Fast Rule-Based Graph Programs

Graham Campbell[a,1], Brian Courtehoute[b], Detlef Plump[b]

[a] *School of Mathematics, Statistics and Physics, Newcastle University, Newcastle upon Tyne, United Kingdom*
[b] *Department of Computer Science, University of York, York, United Kingdom*

Abstract

Implementing graph algorithms efficiently in a rule-based language is challenging because graph pattern matching is expensive. In this paper, we present a number of linear-time implementations of graph algorithms in GP 2, an experimental programming language based on graph transformation rules which aims to facilitate program analysis and verification. We focus on two classes of rule-based graph programs: graph reduction programs which check some graph property, and programs using a depth-first search to test some property or perform an operation such as producing a 2-colouring or a topological sorting. Programs of the first type run in linear time without any constraints on input graphs while programs of the second type require input graphs of bounded degree to run in linear time. Essential for achieving the linear time complexity are so-called rooted rules in GP 2, which, in many situations, can be matched in constant time. For each of our programs, we prove both correctness and complexity, and also give empirical evidence for their run time.

Keywords: Graph transformation, Rooted graph programs, GP 2, Linear-time algorithms, Graph reduction, Depth-first search, Topological sorting

1. Introduction

Rule-based graph transformation was established as a research field in the 1970s and has since then been the subject of countless articles. While many of these contributions have a theoretical nature (see the monograph [1] for a recent overview), there has also been work on languages and tools for executing and analysing graph transformation systems.

Languages based on graph transformation rules include AGG [2], GReAT [3], GROOVE [4], GrGen.Net [5], Henshin [6] and PORGY [7]. This paper focuses on GP 2 [8], an experimental graph programming language which aims

Email addresses: g.j.campbell2@newcastle.ac.uk (Graham Campbell), bc956@york.ac.uk (Brian Courtehoute), detlef.plump@york.ac.uk (Detlef Plump)

[1] Supported by a Doctoral Training Grant from the Engineering and Physical Sciences Research Council (EPSRC) Grant No. (2281162) in the UK.

to facilitate formal reasoning on programs. The language has a simple formal semantics and is computationally complete in that every computable function on graphs can be programmed [9]. Research on graph programs has provided, for example, a Hoare-calculus for program verification [10, 11] and a static analysis for confluence checking [12].

A challenge for the design and implementation of graph transformation languages is to narrow the performance gap between imperative and rule-based graph programming. The bottleneck for achieving fast graph transformation is the cost of graph matching. In general, matching the left-hand graph L of a rule within a *host graph* G requires time $\text{size}(G)^{\text{size}(L)}$ which is polynomial since L is fixed. (We denote by $\text{size}(X)$ the number of nodes and edges in a graph X.) As a consequence, linear-time imperative graph algorithms may be slowed down to polynomial time when they are recast as rule-based graph programs.

To mitigate this problem, GP 2 supports *rooted* graph transformation which was first proposed by Dörr [13]. The idea is to distinguish certain nodes as *roots* and to match roots in rules with roots in host graphs. Then only the neighbourhood of *host graph* roots needs to be searched for matches, allowing, under mild conditions, to match rules in constant time. In [14], *fast* rules were identified as a class of rooted rules that can be applied in constant time if host graphs have a bounded node degree and contain a bounded number of roots.

The condition of a bounded number of *host graph* roots can be satisfied by requiring unrooted input graphs and using in loops only rules that do not increase the number of roots. This simply means that no such rule must have more roots in its right-hand side than in its left-hand side. (A refined condition considers the "root balance" of all rules in a loop body simultaneously.) The condition that host graphs must have a bounded node degree depends on the application domain of a program. For example, traffic networks or digital circuits can be considered as graphs of bounded degree.

The first linear-time graph problem implemented by a GP 2 program with fast rules was 2-colouring. In [14, 15] it is shown that this program colours connected graphs of bounded degree in linear time. The compiled program even matches the speed of Sedgewick's textbook C program [16] on grid graphs of up to 100,000 nodes. Since then, the compiler has received some major improvements, in particular relating to the runtime graph data structure used by the compiled programs [17], which has allowed us to achieve linear time worst-case performance for a wider class of programs than was previously possible, in some cases even on input graph classes of unbounded degree.

In this paper, we continue to provide evidence that rooted graph programs can rival the time complexity of graph algorithms in conventional programming languages. We present five case studies the first three of which are based on graph reduction programs for recognising cycle graphs, trees, and binary DAGs. The other two case studies are based on programs using depth-first search to check connectedness resp. produce a topological sorting of an acyclic graph. Each of these problems is solvable in linear time with algorithms in imperative languages. For each problem, we present a GP 2 program with fast rules, show its correctness, and prove its linear time complexity (on graphs of bounded node

degree for the last two problems). We also give empirical evidence for the linear run time by presenting benchmark results for graphs of up to around 500,000 nodes in various graph classes.

This paper is a revised and significantly extended version of [18], which was written before the improvements to the GP 2 run time system. We now have graph reduction programs which recognise cycle graphs, trees and binary DAGs in linear time among arbitrary input graphs, without the restriction of bounded node degrees. We present the recognition programs together with new proofs of correctness and complexity. They all run in linear time, which previously was possible only if input graphs have a bounded node degree. We also revisit the topological sorting program of [18] and the 2-colouring program of Bak and Plump [15], giving more rigorous analyses. The topological sorting program has been re-worked so that it doubles as a program for checking acyclicity. In addition, we present and analyse a program for checking connectedness using depth-first search.

Finally, it is worth noting that rooted rules per se are not a blueprint for imitating algorithms in imperative languages. This is because GP 2 intentionally does not provide access to the graph data structure of its implementation.

2. The Graph Programming Language GP 2

This section briefly introduces GP 2, a non-deterministic language based on graph-transformation rules, first defined in [8]. An up-to-date version of the syntax of GP 2 can be found in [19]. The language is implemented by a compiler generating C code [15, 17], and the source code is available on GitHub[2].

2.1. Graphs, Rules and Programs

GP 2 programs transform input graphs into output graphs, where graphs are directed and may contain parallel edges and loops. Both nodes and edges are labelled with lists consisting of integers and character strings. This includes the special case of items labelled with the empty list which may be considered as "unlabelled".

The principal programming construct in GP 2 consist of conditional graph transformation rules labelled with expressions. For example, the rule `push` in Figure 17 has three formal parameters of type `list`, a left-hand graph and a right-hand graph which are specified graphically, and a textual condition starting with the keyword `where`.

The small numbers attached to nodes are identifiers, all other text in the graphs consist of labels. Parameters are typed but in this paper, we only need the most general type `list` which represents lists with arbitrary values.

Besides carrying expressions, nodes and edges can be *marked* red, green or blue. In addition, nodes can be marked grey and edges can be dashed. For

[2] `https://github.com/UoYCS-plasma/GP2`

example, rule `push` in Figure 17 contains red and grey nodes and unmarked edges. Nodes and edges in rules can also be marked magenta, which allows the mark to be bound to any mark at the point of rule instantiation (see the next paragraph). Marks are convenient, among other things, to record visited items during a graph traversal and to encode auxiliary structures in graphs. The programs in the following sections use marks extensively.

Rules operate on *host graphs* which are labelled with constant values (lists containing integers and character strings). Formally, the application of a rule to a *host graph* is defined as a two-stage process in which first the rule is instantiated by replacing all variables with values of the same type, and evaluating all expressions. This yields a standard rule (without expressions) in the so-called double-pushout approach with relabelling [20]. In the second stage, the instantiated rule is applied to the *host graph* by constructing two suitable pushouts. We refer to [19] for details and only give an equivalent operational description of rule application.

Applying a rule $L \Rightarrow R$ to a *host graph* G works roughly as follows: (1) Replace the variables in L and R with constant values and evaluate the expressions in L and R, to obtain an instantiated rule $\hat{L} \Rightarrow \hat{R}$. (2) Choose a subgraph S of G isomorphic to $\hat{L}$ such that the dangling condition and the rule's application condition are satisfied (see below). (3) Replace S with $\hat{R}$ as follows: numbered nodes stay in place (possibly relabelled), edges and unnumbered nodes of $\hat{L}$ are deleted, and edges and unnumbered nodes of $\hat{R}$ are inserted.

In this construction, the *dangling condition* requires that nodes in S corresponding to unnumbered nodes in $\hat{L}$ (which should be deleted) must not be incident with edges outside S. The rule's application condition is evaluated after variables have been replaced with the corresponding values of $\hat{L}$, and node identifiers of L with the corresponding identifiers of S. For example, the condition $\mathrm{indeg}(\mathtt{1}) < \mathtt{2}$ of rule `push` in Figure 17 requires that node $g(\mathtt{1})$ has at most one incoming edge, where $g(\mathtt{1})$ is the node in S corresponding to `1`.

A program consists of declarations of conditional rules and procedures, and exactly one declaration of a main command sequence, which is a distinct procedure named `Main`. Procedures must be non-recursive, they can be seen as macros. We describe GP 2's main control constructs.

The call of a rule set $\{r_1, \ldots, r_n\}$ non-deterministically applies one of the rules whose left-hand graph matches a subgraph of the *host graph* such that the dangling condition and the rule's application condition are satisfied. The call *fails* if none of the rules is applicable to the host graph.

The command `if` C `then` P `else` Q is executed on a *host graph* G by first executing C on a copy of G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The `try` command has a similar effect, except that P is executed on the result of C's execution.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body was entered for the last time. The `break` command inside a loop terminates that loop and transfers control to the command following the loop.

In general, the execution of a program on a *host graph* may result in different

graphs, fail, or diverge. This is formally defined in the next subsection.

2.2. Operational Semantics of GP 2

This subsection reviews the semantics of GP 2, except for the definition of rule applications, in the style of structural operational semantics [21]. In this approach, inference rules inductively define a small-step transition relation $\to$ on *configurations*. In the setting of GP 2, a configuration is either a command sequence together with a host graph, just a host graph or the special element fail:

$$\to \subseteq (\text{ComSeq} \times \mathcal{G}) \times ((\text{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\text{fail}\})$$

where $\mathcal{G}$ is the set of GP 2 host graphs. Configurations in $\text{ComSeq} \times \mathcal{G}$, given by a rest program and a host graph, represent states of unfinished computations while graphs in $\mathcal{G}$ are final states or *results* of computations. The element fail represents a failure state. A configuration γ is said to be *terminal* if there is no configuration δ such that $\gamma \to \delta$.

Figure 1 shows the inference rules for the core commands of GP 2. The rules contain meta-variables for command sequences and graphs, where R stands for a call of a rule set or of a rule, C, P, P', Q stand for command sequences, and G, H stand for host graphs. The transitive and reflexive-transitive closures of $\to$ are written $\to^+$ and $\to^*$, respectively. We write $G \Rightarrow_R H$ if H results from host graph G by applying the rule set R, while $G \not\Rightarrow_R$ means that there is no graph H such that $G \Rightarrow_R H$ (application of R fails).

The inference rules for the remaining GP 2 commands are given in Figure 2. These commands are referred to as *derived* commands because they can be defined by the core commands.

The meaning of GP 2 programs is summarised by the semantic function $[\![\_]\!]$ which assigns to each command sequence P the function $[\![P]\!]$ mapping an input graph G to the set $[\![P]\!]G$ of all possible results of executing P on G. The value fail indicates a failed program run while $\bot$ indicates a run that does not terminate or gets stuck. Program P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \dots$ Also, P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$.

The *semantic function* $[\![\_]\!]\colon \text{ComSeq} \to (\mathcal{G} \to 2^{\mathcal{G}^\oplus})$ is defined by

$$\begin{aligned}[\![P]\!]G = \; & \{X \in (\mathcal{G} \cup \{\text{fail}\}) \mid \langle P, G\rangle \xrightarrow{+} X\} \cup \\ & \{\bot \mid P \text{ can diverge or get stuck from } G\},\end{aligned}$$

where ComSeq is the set of GP 2 command sequences and $\mathcal{G}^\oplus = \mathcal{G} \cup \{\bot, \text{fail}\}$. Getting stuck indicates a form of divergence that can happen with a command `if` C `then` P `else` Q or `try` C `then` P `else` Q in case C can diverge from a graph G and neither produce a graph nor fail from G, or with a loop B! whose body B possesses the said property.

$$[\text{call}_1]\ \frac{G \Rightarrow_R H}{\langle R, G\rangle \to H} \qquad [\text{call}_2]\ \frac{G \not\Rightarrow_R}{\langle R, G\rangle \to \text{fail}}$$

$$[\text{seq}_1]\ \frac{\langle P, G\rangle \to \langle P', H\rangle}{\langle P; Q, G\rangle \to \langle P'; Q, H\rangle} \qquad [\text{seq}_2]\ \frac{\langle P, G\rangle \to H}{\langle P; Q, G\rangle \to \langle Q, H\rangle}$$

$$[\text{seq}_3]\ \frac{\langle P, G\rangle \to \text{fail}}{\langle P; Q, G\rangle \to \text{fail}}$$

$$[\text{if}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, G\rangle}$$

$$[\text{if}_2]\ \frac{\langle C, G\rangle \to^+ \text{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{try}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, H\rangle}$$

$$[\text{try}_2]\ \frac{\langle C, G\rangle \to^+ \text{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{alap}_1]\ \frac{\langle P, G\rangle \to^+ H}{\langle P!, G\rangle \to \langle P!, H\rangle} \qquad [\text{alap}_2]\ \frac{\langle P, G\rangle \to^+ \text{fail}}{\langle P!, G\rangle \to G}$$

$$[\text{alap}_3]\ \frac{\langle P, G\rangle \to^* \langle \texttt{break}, H\rangle}{\langle P!, G\rangle \to H} \qquad [\text{break}]\ \langle \texttt{break}; P, G\rangle \to \langle \texttt{break}, G\rangle$$

Figure 1: Inference rules for core commands

$$[\text{or}_1]\ \langle P\ \texttt{or}\ Q, G\rangle \to \langle P, G\rangle \qquad [\text{or}_2]\ \langle P\ \texttt{or}\ Q, G\rangle \to \langle Q, G\rangle$$

$$[\text{skip}]\ \langle \texttt{skip}, G\rangle \to G \qquad [\text{fail}]\ \langle \texttt{fail}, G\rangle \to \text{fail}$$

$$[\text{if}_3]\ \langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}_3]\ \langle \texttt{try}\ C\ \texttt{then}\ P, G\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}_4]\ \langle \texttt{try}\ C\ \texttt{else}\ P, G\rangle \to \langle \texttt{try}\ C\ \texttt{then skip else}\ P, G\rangle$$

$$[\text{try}_5]\ \langle \texttt{try}\ C, G\rangle \to \langle \texttt{try}\ C\ \texttt{then skip else skip}, G\rangle$$

Figure 2: Inference rules for derived commands

2.3. The GP 2-to-C Compiler

GP 2's primary implementation is a GP 2-to-C compiler, and all our complexity assumptions will be compatible with this implementation. The original GP 2 compiler is described in detail by Bak's thesis [19], and recent modifications are described in [17]. These modifications are important for two reasons. The first is that there have been significant changes to the compiler's generated code and runtime system which enable us to write reduction programs in GP 2 that are linear time on graph classes of unbounded degree, which has not been possible before. The second important change is to the theoretical model (and

implementation) of root nodes and morphisms, requiring morphisms to reflect rootedness as well as preserving it. We will assume all morphisms are rootedness reflecting in this paper.

As shown in Figure 3, GP 2 programs are first compiled to C programs using the GP 2-to-C compiler implementation, which are in turn compiled by GCC to a platform dependent executable. This executable then reads an input graph from disk and executes the program, with one of four possible outcomes:

1. the program produces an output graph;
2. the program evaluates to `fail`;
3. the program encounters a runtime error;
4. the program does not terminate.

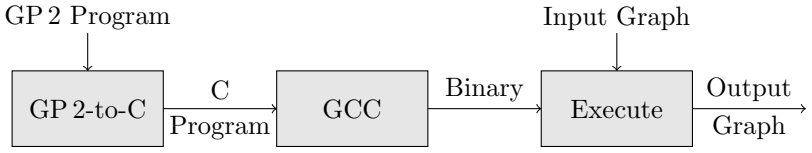


Figure 3: The GP 2-to-C compiler

In this paper, we will ignore runtime errors such as division by zero in a label operation, or the physical memory of the machine being exhausted.

GP 2 programs are graphically visualised throughout this paper. We now look at an example program which computes the transitive closure of a graph (Figure 4). The concrete syntax for the program is available on GitHub[3].

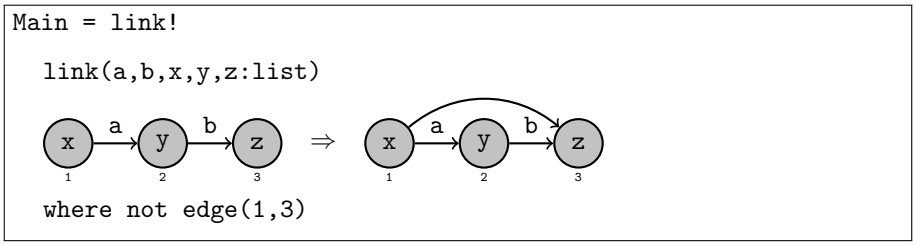


Figure 4: GP 2 program `transitive-closure`

The generated C code for this GP 2 program:

1. Reads and parses the input host graph;
2. Applies the `link` rule as long as possible;
3. Outputs the *host graph* as output.

[3] `https://gist.github.com/GrahamCampbell/c429491366119c898c9d1b5ad0459ab1`

Looking at what it means to apply the `link` rule as long as possible, we attempt to find a *match* satisfying the application condition, and if we find one, we apply the rule, adding a fresh edge. We repeat this (zero or more times) until we can no longer find a suitable match. Finding a match involves first finding a suitable node, then finding a suitable outgoing edge to a suitable node, and then another suitable outgoing edge to a suitable node, and then checking the application condition. We call this the matching algorithm. This is described in general by Bak [19] in his thesis, and updates to the algorithm are defined in [18].

2.4. Reasoning about Time Complexity

When analysing the time complexity of programs, we assume that these are fixed. This is customary in algorithm analysis where programs are fixed and running time is measured in terms of input size [22, 23]. In our setting, the input size is the *size* of a host graph, which we define to be the total number of nodes and edges. Figure 5 shows the complexity of various runtime operations, where n is the size of the current host graph.

Procedure	Description	Complexity
`parseInputGraph`	Parse and load the input graph into memory: the host graph.	$O(n)$
`alreadyMatched`	Test if the given item has been matched in the host graph.	$O(1)$
`clearMatched`	Clear the `is matched` flag for a given item.	$O(1)$
`setMatched`	Set the `is matched` flag for a given item.	$O(1)$
`firstHostNode`	Fetch the first node in the host graph.	$O(1)$
`nextHostNode`	Given a node, fetch the next node in the host graph.	$O(1)$
`firstHostRootNode`	Fetch the first root node in the host graph.	$O(1)$
`nextHostRootNode`	Given a root node, fetch the next root node in the host graph.	$O(1)$
`firstInEdge`	Given a node, fetch the first incoming edge.	$O(1)$
`nextInEdge`	Given a node and an edge, fetch the next incoming edge.	$O(1)$
`firstOutEdge`	Given a node, fetch the first outgoing edge.	$O(1)$
`nextOutEdge`	Given a node and an edge, fetch the next outgoing edge.	$O(1)$
`printHostGraph`	Write the current host graph state as output.	$O(n)$

Figure 5: Runtime complexity assumptions

Throughout the paper, we provide empirical evidence of the time complexity of programs, which includes the time needed to read, parse, and load the input graph into memory, and also the time needed to write the output graph, but not the time spent compiling the program. This evidence supports our complexity proofs and also gives confidence in the accuracy of Figure 5. Formal complexity analysis of those operations is beyond the scope of this paper, and all our complexity results are relative to these basic complexities.

When we discuss the time complexity of programs, we will do this by reasoning about the complexity of the generated code. When considering the complexity of rule applications, it suffices to only reason about the complexity of finding a match because all the programs in this paper satisfy the assumption of the following lemma.

Lemma 2.1 (Constant Time Application)**.** Once a match has been found for a rule that does not modify labels, other than perhaps introducing a fixed label or changing marks, only constant time is needed to complete the process of applying the instantiated rule and building the result graph.

When analysing the programs of this paper, by a *step* we mean a completed or failed rule application, the `break` operation, or the `fail` operation. While `break` and `fail` always finish in constant time, completed and failed rule applications can often be shown to require only constant time by the specific properties of our rules (see below) and the preconditions of some of our programs such as a bounded node degree in input graphs.

2.5. Rooted Programs

The bottleneck for efficiently implementing algorithms in a language based on graph transformation rules is the cost of graph matching. In general, to match the left-hand graph L of a rule within a *host graph* G requires time polynomial in the size of L [14, 15]. As a consequence, linear-time graph algorithms in imperative languages may be slowed down to polynomial time when they are recast as rule-based programs.

To speed up matching, GP 2 supports *rooted* graph transformation where graphs in rules and host graphs are equipped with so-called root nodes. Roots in rules must match roots in the *host graph* so that matches are restricted to the neighbourhood of the host graph's roots. We draw root nodes using double circles. For example, in the rule `prune` of Figure 17, the node labelled `y` in the left-hand side and the single node in the right-hand side are roots.

A conditional rule $\langle L \Rightarrow R, c \rangle$ is *fast* if (1) each node in L is undirectedly reachable from some root, (2) neither L nor R contain repeated occurrences of list, string or atom variables, and (3) the condition c contains neither an `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.

Conditions (2) and (3) will be satisfied by all rules occurring in the following sections; in particular, we neither use the `edge` predicate nor the equality tests. For example, the rules `prune` and `push` in Figure 17 are fast rules.

Theorem 2.2 (Fast Rule Matching [14, 15])**.** Rooted graph matching can be implemented to run in constant time for fast rules, provided there are upper bounds on the maximal node degree and the number of roots in host graphs. Moreover, the GP 2-to-C compiler produces programs that match fast rooted rules in constant time under the above conditions.

Theorem 2.2 is used in the complexity analysis of programs with an input graph class of bounded degree. This approach is central to Section 4. However, it is too coarse to obtain sharp bounds on complexity in general. In particular, in Section 3, we wish to show complexity results with input graph classes not necessarily of bounded degree. We are able to do this via direct reasoning about the complexity of programs generated by the GP 2-to-C compiler.

3. Fast Reduction Programs

The aim of this section is to demonstrate that GP 2 can recognise various graph classes in linear time, by means of simple reduction specifications. As discussed in the previous sections, this is made possible by GP 2's implementation of root nodes. We do not know of any other rule-based graph programming languages that can claim linear time complexity for such tasks.

As well as providing proofs of complexity, we back-up our assumptions by timing the actual execution times of the programs produced by the GP 2-to-C compiler on various graph classes (Figures 6, 7, 8, 9, 10, 11, 12 and 13). The concrete syntax for all the programs in this section is available on GitHub[4].

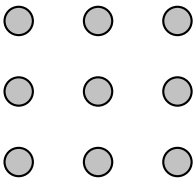


Figure 6: Discrete graph

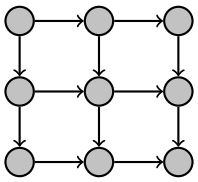


Figure 7: Grid graph

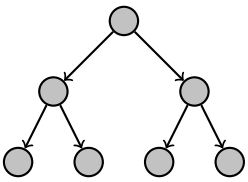


Figure 8: Binary tree

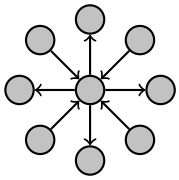


Figure 9: Star graph

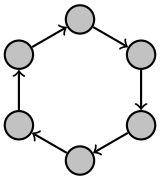


Figure 10: Cycle graph

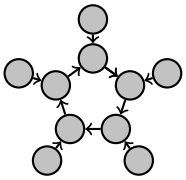


Figure 11: Sun graph

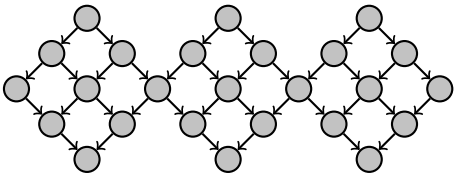


Figure 12: Grid chain

Figure 13: Linked list

For the purposes of Section 3 only, we define:

Definition 3.1 (Input Graph)**.** An *input graph* is an arbitrarily labelled GP 2 *host graph* such that:

1. every node is marked grey;
2. every node is unrooted;
3. every edge is unmarked.

[4] `https://gist.github.com/GrahamCampbell/102eb8ec101ba87f6040ec2e9f3323a2`

3.1. Recognising Cycle Graphs

The class of cycle graphs, up to labelling, consists of the graph containing one node and one edge, and graphs containing n nodes, connected by a directed cycle, for all $n \geq 2$. It is reasonably straightforward to specify this class using reduction rules, yielding a GP 2 program (Figure 14) which, given an *input graph* G, fails if and only if G it is not a cycle graph.

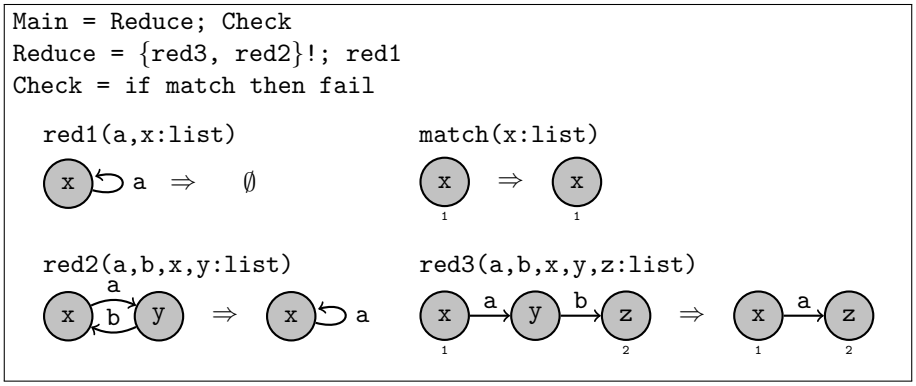


Figure 14: GP 2 program `is-cycle-slow`

Notice that, while the number of computation steps is only linear in the *size* of input graph, the program need not terminate in linear time, due to the fact that the time needed to find a match for each rule application need not be only constant, as we discussed in subsection 2.5. One of the novelties of GP 2 is its implementation of root nodes. Using root nodes, we are able to direct the matching algorithm to only consider a constant *size* subgraph of the input, giving us a program that is genuinely linear time (Figure 15).

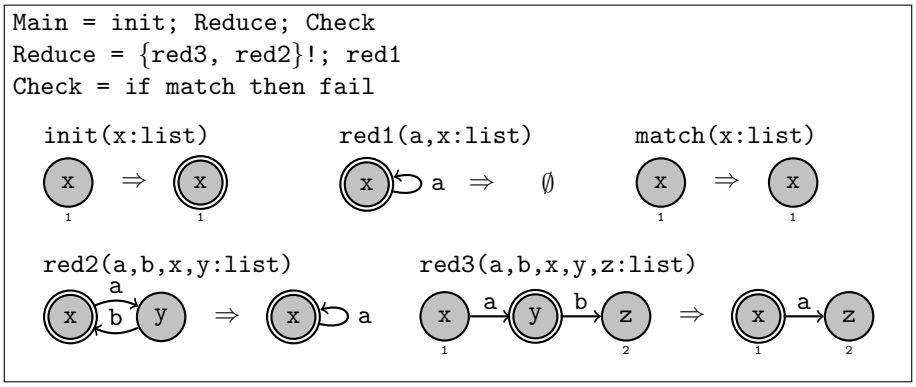


Figure 15: GP 2 program `is-cycle`

Even though this program is simple, we will prove its correctness and complexity now, to give a flavour of the style of such proofs, in preparation for

the later proofs. By *total correctness* of a program with respect to a specification, we mean that on all graphs satisfying the input description, the program terminates with output satisfying the output description.

We must start by showing, by induction on derivation length, that the reduction rules `red2` and `red3` reduce a *rooted cycle graph* to a 1-cycle, and no *rooted input graphs*, where by a *rooted cycle graph*, we mean a cycle graph with exactly one of the nodes a root, and similarly for a *rooted input graph*.

Lemma 3.2 (Cycle Reduction)**.** Applying `red2` and `red3` to a non-empty *rooted input graph* G will terminate after $|V_G| - 1$ steps, yielding a *rooted 1-cycle graph* if and only if G is a *rooted cycle graph*, and otherwise will yield a non-empty *rooted input graph* which is not a *rooted cycle graph*.

Proof. Since both rules are size-reducing, reducing the number of nodes by 1, and require at least 2 nodes to be applicable, we have termination after $|V_G| - 1$ steps.

For closedness, suppose first that G is a *rooted cycle graph* with n nodes. Then the application of either one of the rules has the effect of transforming G into a *rooted cycle graph* with $n - 1$ nodes, if $n > 1$ and if $n = 1$ the rules aren't applicable.

If G is not a *rooted cycle graph*, then either the root node lies in a connected component that is a cycle graph, and the application of one of the rules has the effect of reduction that connected component only, leaving the other components alone, or there is only one connected component and it is not a cycle graph due to an additional edge present. Suppose that extra edge is a loop, then `red2` must not be applicable due to the dangling condition, and if that extra edge is a proper edge, then `red3` is either not applicable because the loop is on the root node, or `red3` is applicable and preserves the additional edge, so preserves the fact that the graph is not a *rooted cycle graph*. □

Theorem 3.3 (Correctness of `is-cycle`)**.** The program `is-cycle` (Figure 15) is totally correct with respect to the specification:

Input: An *input graph*.
Output: Fail if and only if the input is not a cycle graph.

Proof. Termination follows from Lemma 3.2. It remains to show partial correctness. Suppose the input graph empty, then `init` will not be applicable, and the program will terminate with `fail`. It remains to analyze the case where the input graph is non-empty. Clearly, `init` will always be applicable, and will have the effect of rooting exactly one of the nodes of the graph, producing a rooted input graph which is a rooted cycle graph if and only if the original input graph was a cycle graph.

What happens next is that `red3` and `red2` are applied as long as possible. We know by Lemma 3.2 that the result of this computation will be a rooted 1-cycle if the input graph was a cycle graph, and some non-empty non-cycle graph otherwise. In the first case, `red1` will be applied, deleting the graph, and then `match` will not be applicable, so the program succeeds (with the empty graph as

output). Otherwise, either it is the case that `red1` is applicable and then `match` is too, so the program fails, or `red1` is not applicable, so the program fails. □

Lemma 3.4 (Complexity of `init`)**.** `init` terminates in constant time on an *input graph* G.

Proof. The search plan will iterate all nodes in G looking for the first grey unrooted node. Testing if a node is unrooted and grey takes only constant time. Since every node in an *input graph* is unrooted and grey, the search will stop at the first node, or fail in constant time if G is empty. □

Lemma 3.5 (Complexity of `Reduce`)**.** `Reduce` terminates in linear time with respect to the number of nodes in a *rooted input graph* G.

Proof. First, we must argue that `red1`, `red2` and `red3` each take only constant time to either evaluate to `fail` or produce a result graph. Then by Lemma 3.2, we know that {`red3, red2`}`!` runs for only a linear number of steps, and each takes only constant time. So `Reduce` must take only linear time, since it involves executing {`red3, red2`}`!` followed by `red1`.

`red1` is the simplest to analyze. The search plan will look for a root node, and find the first one in constant time, and check that it is grey in constant time, which it is. Next, it will check the incoming and outcoming degrees are both one in constant time. If they are not, then the node is rejected and the search plan looks for another root node and determines there are no more in constant time, and matching fails. If the degrees are correct, next the search plan grabs the first outgoing edge in constant time and checks if it is an unmarked and a loop in constant time. If it is not, then matching fails as before. Otherwise, matching succeeds.

For `red2`, the search plan first looks at the root nodes in exactly the same way as for `red1`, with the same degree checking and rejection handling. The first difference occurs when the search plan looks at the first outgoing edge. It instead checks the edge is proper, also in constant time. It then checks, in constant time, that the target node is grey, unrooted, and has incoming and outgoing degree one. If it is not, the root node is rejected, as before. Otherwise, the first outgoing edge of the new node is grabbed in constant time and checking that it is unmarked and has the root node as a target occurs in constant time. If it is not, then we must look for a new root node, as before, which necessarily fails, and we stop in constant time.

`red3` will turn out to be constant time also, due to the important decision to make the non-interface node the root node. Correctness would not have been impacted had we made a different node in the left-hand side graph the root, but matching complexity would have been impacted! Initially, the search plan proceeds as in `red1`, again, finding a grey root with incoming and outgoing degree one. Next, the search plan grabs the first outgoing edge in constant time and checks it is unmarked and proper in constant time, and that the target node is grey and unrooted. If this fails, then we bailout, as usual, having to look for another root node which fails in constant time. If successful, we then due the dual for the incoming edge of the root node in constant time. □

Lemma 3.6 (Complexity of `Check`)**.** `Check` terminates in constant time on an *rooted input graph* G.

Proof. The search plan will iterate all nodes in G looking for the first grey unrooted node. Testing if a node is unrooted and grey takes only constant time. The search plan will only consider at most 2 nodes. There are three cases to consider. The first case is that the first node we consider is what we want, and we stop. The second case is that the first node we consider is a root node and there are no other nodes, so we stop. The final case is that the first node we consider is a root, and then the second node is the one we are looking for. □

Theorem 3.7 (Complexity of `is-cycle`)**.** The program `is-cycle` (Figure 15) terminates in linear time with respect to the *size* of its input.

Proof. Due to Lemma 3.4 `init` takes only constant time to either evaluate to `fail`, which stops the whole program, or evaluate to a *rooted input graph*. Next, by Lemma 3.5 `Reduce` takes linear time in the number of nodes, either evaluating to `fail`, which stops the whole program, or evaluates to a *rooted input graph* (Lemma 3.2). Finally, `fail` takes only constant time by Lemma 3.6. □

Finally, we have collected empirical timing results for `is-cycle`, supporting our claim that the program runs in linear time, even on graph classes that do not have bounded degree (Figure 16).

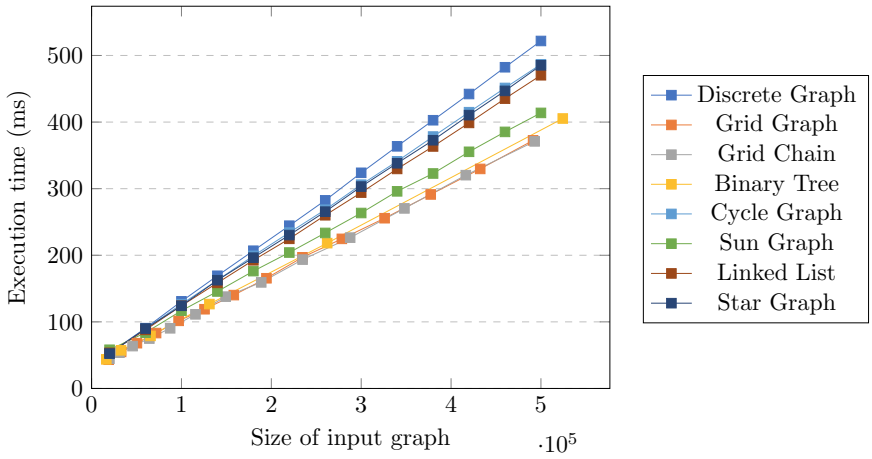


Figure 16: Measured performance of `is-cycle`

3.2. Recognising Trees

A tree is a graph containing a node from which there is a unique directed path to each node in the graph. It is easy to see that it is possible to generate the class of all unlabelled trees by inductively adding new leaf nodes to the discrete graph of *size* one, thus the class can be specified by graph reduction.

The question of linear time recognition of trees was partially solved by Campbell in 2019, producing a GP 2 program to recognise trees in linear time, given the input class has only bounded degree [24]. In this subsection we improve on this result, removing the bounded degree restriction.

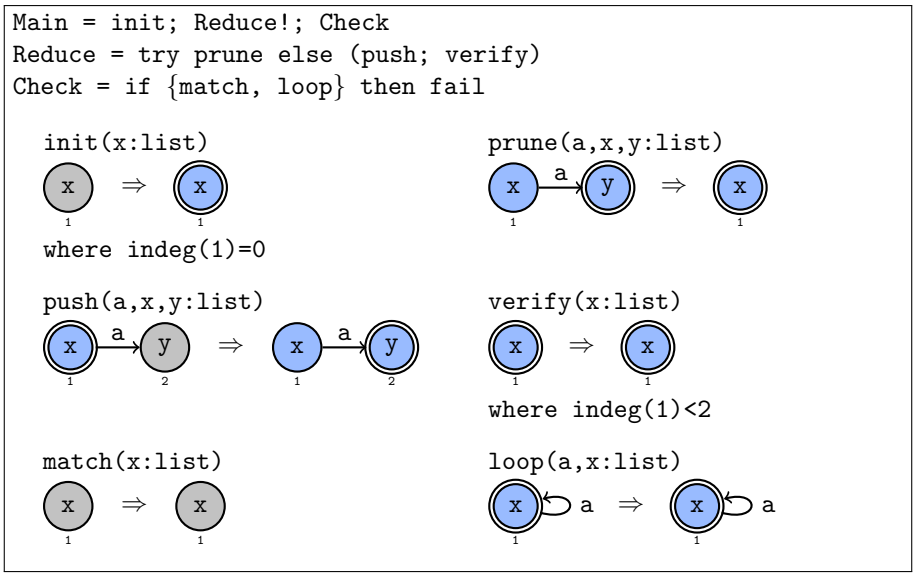


Figure 17: GP 2 program `is-tree`

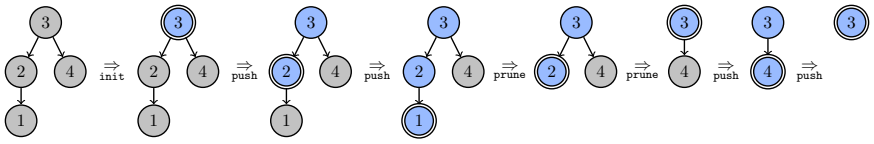


Figure 18: Example tree reduction

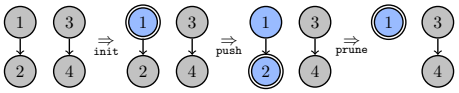


Figure 19: Example forest reduction

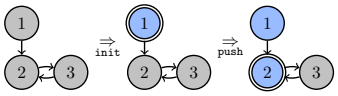


Figure 20: Example reduction with cycle

Intuitively, our new program (Figure 17) works by pushing a special *root* node to the bottom of a branch, and then prunes, repeating the process as long as possible. If we start with a tree and run this until we cannot do it anymore, we must be left with a single root node (Figures 18 and 19).

Notice that we have used both grey and blue node colours. This is necessary in order to ensure termination. Consider an *input graph* that is not a tree, say

it contains a 2-cycle (Figure 20). Then, we would have otherwise been able to push the root node round and round forever! Leaving a trail of blue nodes behind us as we push prevents this from happening.

In order to prove correctness and complexity, we must first show some intermediate results. Using the fact that an equivalent characterisation of a tree is a non-empty connected graph without undirected cycles such that every node has at most one incoming edge, we can show that our reduction procedure preserves the property of either being a tree or not being a tree. The additional `verify` rule is needed to make sure that we stay within linear time complexity, even on graphs of unbounded degree.

Definition 3.8 (Live Graph)**.** A *live graph* (for the purposes of this subsection) is any graph that can be derived from an *input graph* by applying `init` followed by zero or more applications of `push` or `prune` in any order, where these rules are given in Figure 17.

Lemma 3.9 (Live Graph Properties)**.** Live graphs have exactly one blue root node, every other node either grey or blue, and all edges unmarked.

Proof. The only rule that can change the number of root nodes is `init`. Now, `init` is always applied to a graph with no root nodes, so necessarily must increase the number of root nodes by exactly one. No rules have marked edges in their RHSs, and similarly, have only grey and blue nodes in their RHSs. □

Lemma 3.10 (Tree Reduction)**.** `Reduce` is a refinement of {`push, prune`}. That is, if $G \Rightarrow_{\texttt{Reduce}} H$, then there must either be a direct derivation $G \Rightarrow_{\texttt{push}} H$ or $G \Rightarrow_{\texttt{prune}} H$. Moreover, G is a tree if and only if H is.

Proof. For the first part, the observation to make is that the verify rule does not modify graphs. Its only purpose is to possibly evaluate to fail. So, the result then followings from the definition of `try`.

For the second part, it suffices to show that `prune` and `push` preserve the property of being a (non-)tree. Clearly `push` does not modify the structure of graphs, so we just need to analyse `prune`, which has the effect of deleting an edge and its target only when the target has no other incident edges. If G is a tree, then H is also a tree, since all we have done is deleted a leaf node. If G is not a tree, then H must also not be a tree, since if G was disconnected, H must be too, and if G has any undirected cycles, then they cannot have passed through the deleted node due to the condition on other incident edges, so H has the same undirected cycles as G, and if G node with incoming degree at least 2, then so must H since the incoming degree of all the remaining nodes in H is the same as in G and the deleted node had only incoming degree 1. □

Definition 3.11 (Blue Paths and Cycles)**.** A *blue path* (*cycle*) is a *path* (*cycle*) in a graph with all nodes blue. The *length* of a *blue path* (*cycle*) is the number of edges.

Lemma 3.12 (Live Graph Blue Paths)**.** Given a *live graph*, then:

(P1) There is a unique blue node with incoming degree zero, and all other blue nodes have incoming degree one;

(P2) There are no blue cycles, only blue paths;

(P3) All maximal blue paths start from the incoming degree zero node and end at the root node.

Proof. First, note together with Lemma 3.9, this means there is exactly one maximal blue path in each *live graph* (the maximum blue path), otherwise, there would either be a blue cycle or a node with incoming degree of at least 2. Now, to show the properties, we will proceed by induction on derivation length.

Let I be an arbitrary *input graph* and $I \Rightarrow_{\texttt{init}} G$. I and G are the two base cases we must check. By definition, I satisfies the properties. To see that G does, note that all nodes in G must be grey and unrooted apart from one blue root node with incoming degree 0, due to the application condition of `init`.

Suppose now that we have $I \Rightarrow_{\texttt{init}} G \Rightarrow^{n}_{Reduce}$ with H satisfying the properties by inductive hypothesis. We now consider the two possible cases for the successor graph to H. By Lemma 3.10, we have either $H \Rightarrow_{\texttt{prune}} M$ or $H \Rightarrow_{\texttt{push}} M$.

In the first case, M is exactly H but with the bottom of the maximal path deleted and the new bottom node rooted. There is necessarily no blue child of the new root in M, since this would contradict the uniqueness of the original maximal path in H. Since the incoming degree of every remaining node is left unchanged, we conclude that all of the properties hold, as required.

In the second case, M is exactly H but with the bottom of the maximal path extended along an existing edge to an existing previously grey node, becoming the unique blue root in M. Suppose by contradiction that this root had a blue child, then once again this contradicts the uniqueness of the maximal path ion H. Since the incoming degree of every node is unchanged and the application condition of `verify` ensures our replacement root has incoming degree at most 1, then we conclude that all of the properties hold, as required. □

Lemma 3.13 (Live Tree Reduction)**.** Given a *live tree* G, then either G is a single blue root node, or `Reduce` is applicable.

Proof. Suppose G is a live tree. Then by Lemma 3.9, it has a blue root node. If this is the only node G, then G must necessarily be discrete: G is a single isolated blue root node. Alternatively, if G has more than one node, then either this root node has outgoing degree 0 or not. We must analyze these two cases.

If the root has no outgoing edges, then it must have incoming degree 1 since G is a tree with at least 2 nodes. But then by Lemma 3.12, there must be a blue node above it, so `prune` must be applicable. Thus, `Reduce` is applicable, as required.

If the root has at least one outgoing edge, then by Lemma 3.12, all of the nodes below it are grey, so we `push` must be applicable. Then, since the result is a tree, due to Lemma 3.10, we know that `verify` is always applicable, since

all nodes in a tree of incoming degree at most 1. Thus, `Reduce` is applicable, as required. □

Lemma 3.14 (Intermediate Complexity Result)**.** Call the entire application of `init` or `Reduce` a single computation step, let G be an *input graph* with n nodes, and define `P` to be the program `init; Reduce!`. Then `P` terminates after at most $\max(1, 2(n-1))$ steps on G.

Proof. Define the weight $w(G)$ of G to be $2g + b$ where g is the number of grey nodes in G and b the number of blue nodes in G.

From an *input graph* G, we either have $G \Rightarrow_{\texttt{init}} \texttt{fail}$, or $G \Rightarrow_{\texttt{init}} H$, where H is a *live graph*. In the first case, the program has terminated after 1 step, and we're done. In the second case, we have performed one step to derive H, and next we can proceeded to apply `Reduce` as long as possible. By Lemma 3.10, `Reduce` has the same effect as applying either `prune` or `push`. Both of these decrease the weight of the graph by exactly 1, and can only be applied to graphs of at least weight 2, so `Reduce` can be applied at most $2n - 3$ times.

Thus `P`, terminates after 1 step on *input graph*s of weight less then 2, and otherwise terminates in $1 + (2n - 3) = 2(n - 1)$ steps, as required. □

Lemma 3.15 (Correctness of `Check`)**.** `Check` evaluates to `fail` on any graph G if and only if G has *size* strictly greater than one.

Proof. {`match, loop`} passes if and only if G has at least two nodes, or a loop, which happens exactly when the *size* of G is at least two. The result then follows from the fact that `Check` fails if and only if {`match, loop`} passes. □

Theorem 3.16 (Correctness of `is-tree`)**.** The program `is-tree` (Figure 17) is totally correct with respect to the specification:

Input: An *input graph*.
Output: Fail if and only if the input is not a tree.

Proof. Termination follows from Lemma 3.14. To see partial correctness, let G be an *input graph*. We break down our proof into two cases.

First, suppose G is empty. Then `init` evaluates to `fail`, and thus `Main` evaluates to `fail`, as required.

Next, suppose G is non-empty and not a tree. Then, if `init` evaluates to `fail` (such as because G is empty), then `Main` evaluates to `fail`, as required. Otherwise, `init` must have evaluated to a *live graph*, then by Lemma 3.10, `Reduce!` will evaluate to a non-tree, which is non-empty by Lemma 3.9. Finally, by Lemma 3.15, the subprogram `if Check then fail` must evaluate to `fail`, since it was fed a graph of *size* at least 2.

Finally, suppose G is a tree. Then `init` is necessarily applicable, producing a live tree. By Lemmas 3.10 and 3.13, `Reduce!` produces a graph of *size* 1, so by Lemma 3.15, the subprogram `if Check then fail` must not evaluate to `fail`, as required. □

Recall from Lemma 2.1 that all rules that do not modify labels have constant time application, once a match has been found. All rules of `is-tree` satisfy this condition, so in our remaining proofs, we omit reasoning about application complexity, only discussing matching time complexity.

Lemma 3.17 (Complexity of `init`)**.** `init` terminates in linear time with respect to the number of nodes in an *input graph* G.

Proof. The search plan will iterate all the nodes in G looking for the first unrooted grey node with incoming degree 0. Testing if a node is unrooted, grey and has incoming degree 0 is only constant time, so matching takes linear time in the worst case, to either find a match satisfying the application conditions or determine there is no such match. □

Lemma 3.18 (Complexity of `Reduce`)**.** `Reduce` terminates in constant time on a *live graph* G.

Proof. First, the program will try to find a match for `prune`. If it succeeds, there is no more time needed for matching. Otherwise, it will next try to find a match for `push`. If it succeeds, the only other matching code to run is that of `verify` on the result H of applying `push` to G.

When trying to find a match for `prune` in G, first we identify the root in the rule LHS with the root in G (which exists by Lemma 3.9), in constant time. We then check the root is blue in constant time, which it will be by Lemma 3.9. Next, we check that the root has outgoing degree 0 and incoming degree 1 in constant time. If it doesn't, then we must look for a second root node in G, which doesn't exist by Lemma 3.9. We detect this in constant time, then `fail`. Otherwise, the root necessarily has a proper incoming edge. The program will check the edge is unmarked, in constant time (and it will be by Lemma 3.9) and checks that the source node is blue, in constant time (which it will be by Lemma 3.12. We have now found a match (or not) in constant time.

When trying to find a match for `push` in G, first we identify the root in the rule LHS with the root in G (which exists by Lemma 3.9), in constant time. We then check the root is blue, in constant time, which it will be by Lemma 3.9. Next, we check the root has outgoing degree a least 1, in constant time. If it doesn't, then we must look for a second root node in G, which doesn't exist by Lemma 3.9. We detect this in constant time, then `fail`. Otherwise, by Lemma 3.12, the first outgoing edge we consider must be proper and unmarked. After verifying this, in constant time, we then look at the target node, and check it is grey, in constant time. We have now found a match (or not) in constant time.

Finally, when trying to find a match for `verify` in H. we identify the root in the LHS of the rule with the root in H which exists and is blue by the definition of `push` and the fact that H was the result of applying `push` to a *live graph* which had exactly one root by Lemma 3.9. We can now check the root is blue and has incoming degree at most 1 in constant time. If this fails, then we look for another root, and once again, don't find one, in constant time. □

Lemma 3.19 (Complexity of `Check`)**.** `Check` terminates in constant time on a *live graph* G.

Proof. The matching time of {`match, loop`} is the sum of the matching time of `match` and `match` given that `match` was not applicable, which tells us there are no proper edges.

For `match`, we just match the first non-root of G, in constant time. If there are fewer than two nodes, then we fail, in constant time.

For `loop`, matching only ever happens if there is at most one node in G, and so clearly, we can check for the presence of a looped edge in constant time.

Thus, `Check` terminates in constant time, as required. □

Theorem 3.20 (Complexity of `is-tree`)**.** The program `is-tree` (Figure 17) terminates in linear time with respect to the *size* of its input.

Proof. Let G be the input graph. Due to Lemma 3.17 we have that `init` takes only linear time with respect to the number of nodes in G, and it is only applied once, from the definition of the program. By Lemma 3.18 we have that `Reduce` takes only constant time and by Lemma 3.14 is applied only a linear number of times with respect to the number of nodes in G. Finally, by Lemma 3.19 we have that `Check` takes only constant time.

So, the program's `Main` procedure actually has worst-case complexity, a constant function of the number of nodes in the *input graph* G. However, in order to start executing `Main`, the *input graph* must first be loaded into memory, which cannot be done any faster than a constant function of the *size* of G. □

Just as with the `is-cycle` program, we have collected empirical timing results for `is-tree`, supporting our claim that the program runs in linear time, even on graph classes that do not have bounded degree (Figure 21).

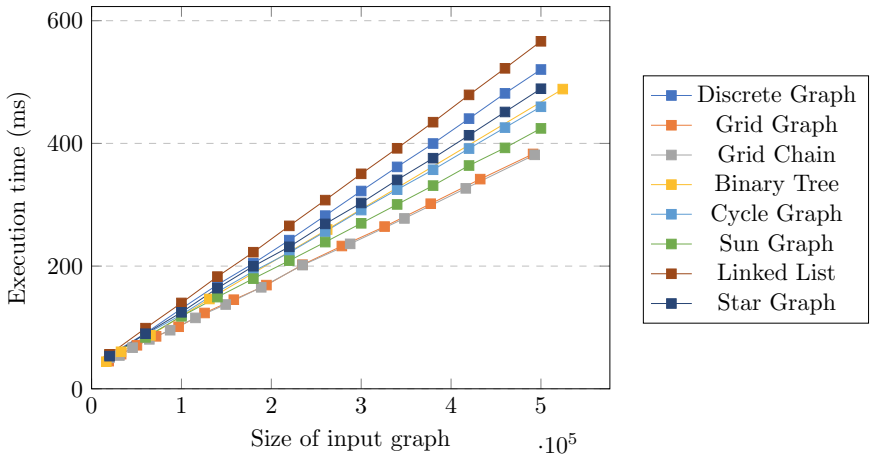


Figure 21: Measured performance of `is-tree`

Finally, we conjecture it is possible to implement this program without needing to use blue nodes, since termination follows from the application conditions preventing the root node from entering a cycle in a graph, which is the only way to instantiate non-termination. We have omitted this program due to the correctness argument being much more fiddly, and the time complexity (and runtime performance) no better.

3.3. Recognising Binary DAGs

Recall that a *directed acyclic graph* (DAG) is a graph containing no directed cycles. A DAG is *binary* if each of its nodes has an outgoing degree of at most two. In this subsection, we present a GP 2 program (Figure 22) that can recognise binary DAGs in linear time.

The program works by finding an incoming degree zero node, then removing it and its edges, if it has the outgoing edges one would expect. This process can be repeated until the entire graph has been deleted, if we have a binary DAG as input (Figure 23). Otherwise, the program necessarily encounters a situation in which reduction cannot continue, and evaluates to fail (Figure 24). Termination is ensured by dashing edges we have visited when searching for an incoming degree zero node.

As with the tree recognition program, we show correctness and complexity, including empirical timing results (Figure 25).

Definition 3.21 (Live Graph)**.** A *live graph* (for the purposes of this subsection) is any graph that appears as an intermediate *host graph* in the process of executing `is-bin-dag` on an *input graph*. Moreover, call a *live graph*:

1. *unrooted* if it has no root nodes;
2. *grey-rooted* if it has exactly one grey root node;
3. *white-rooted* if it has exactly one unmarked root node.

Lemma 3.22 (Live Graph Properties)**.** Let G be a *live graph*. Then:

1. G is either *unrooted*, *grey-rooted*, or *white-rooted*.
2. G has all nodes apart from possibly the root node are grey, and all edges are either unmarked or dashed.
3. G has each node has at most two incoming dashed edges and at most one outgoing dashed edge.

Proof. First, we check that any input graph G satisfies the properties. This is easy to see since G must be an unrooted graph with all nodes grey and all edges unmarked, so all of the three conditions are satisfied. After one step. Next, we analyse what happens to the procedure `init; Reduce!; Guard`. Showing that `init; Reduce!; Guard`(G) is either `fail` or an unrooted live graph if G is an unrooted live graph is sufficient to show the result by induction on the number of iterations of `(init; Reduce!; Guard)!`, since `Check` does not modify the host graph and the input graph is unrooted.

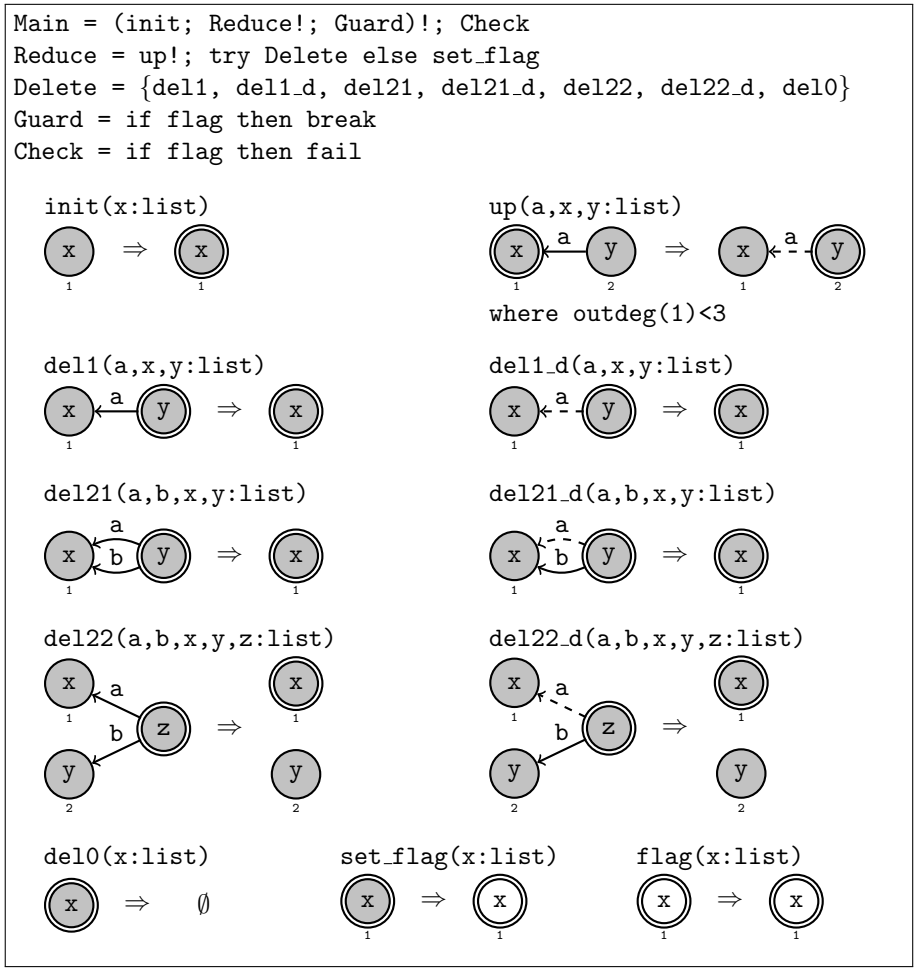


Figure 22: GP 2 program `is-bin-dag`

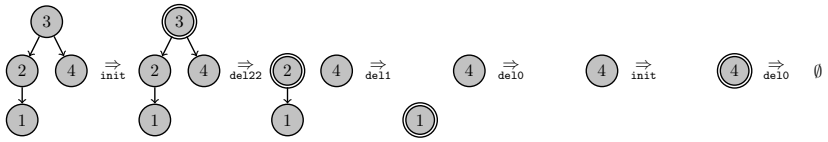


Figure 23: Example tree reduction

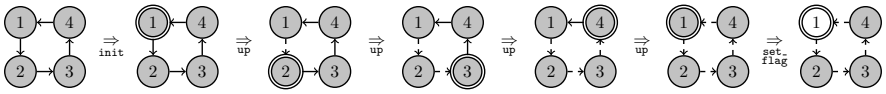


Figure 24: Example cycle reduction

It is clear that $G_0 = \texttt{init}(G)$ is either `fail` or G but with one node turned into a grey root node. We claim that $(\texttt{Reduce!})(G_0)$ is either an unrooted live graph or a white-rooted live graph. To see this, we argue by induction on the number of iterations, showing that $G_{i+1} = (\texttt{Reduce!})(G_i)$ is either `fail` or a graph satisfying the 3 properties of the lemma, and that if it is `fail`, then G_{i+1} must additionally be either unrooted or white-rooted. $G'_i = (\texttt{up!})(G_i)$ must satisfy the 3 properties since `up`'s application condition prevents the 3rd property from being violated, leaving behind a grey-rooted graph. Next, `try Delete else set_flag`(G'_i) clearly either must be the same graph only with the grey root either deleted or unmarked so the conditions are satisfied, as required. Finally, let $H = (\texttt{init; Reduce!})(G)$. Then `Guard`$(H)$ is either H or `fail`. □

Lemma 3.23 (`Reduce` is Structure Preserving)**.** Given a *live graph* G and $G \Rightarrow_{\texttt{Reduce}} H$, then G is a binary DAG if and only if H is.

Proof. We can totally ignore the rootedness and colour of nodes, and the marks of edges for the purposes of this proof, since the same program that operates ignoring everything in this way is certainly compatible, just need not terminate. Since `init`, `up`, `set_flag` and `flag` do not modify the graph structure, then all we need to check is that the 7 remaining rules (ignoring the rootedness, colours, and labels) each individually preserve being a binary DAG and not being a binary DAG, and this is enough to show the result. It is clear that they preserve being a binary DAG, since all of the rules only delete nodes and edges.

To see that `del0` preserves not being a binary DAG is easy, since deleting an isolated node is safe (the remaining connected component must not be a binary DAG). Looking at `del1`, if the input G is not a binary DAG and the rule is applicable, then the dangling condition ensures that the root node has no incoming edges and only one outgoing edge. Now, if G is not a binary DAG then either it has a directed cycle, or a node with outdegree at least 3. The matched root node must not lie on said cycle, nor does it have such outdegree, so removal of the node with its single outgoing edge must leave the directed cycle in place and also the node with outdegree at least 3. The argument for the remaining rules is almost identical. □

Lemma 3.24 (Termination of `is-bin-dag`)**.** On every *input graph* G, the program `is-bin-dag` terminates in $\mathrm{O}(\mathrm{size}(G))$ steps.

Proof. Given any host graph G, let $\mathrm{ug}(G)$ be the number of unrooted grey nodes in G, $\mathrm{g}(G)$ be the number of grey nodes in G, and $\mathrm{ue}(G)$ be the number of unmarked edges in G. Define $\#G = \mathrm{ug}(G) + \mathrm{g}(G) + \mathrm{ue}(G)$. Then

$$0 \leq \#G \leq 2 \times |V_G| + |E_G| \leq 2 \times \mathrm{size}(G). \tag{1}$$

By inspection of the rules in `is-bin-dag`, we have

$$G \underset{r}{\Rightarrow} H \text{ implies } \#H = \#G - 1 \text{ for } r \in \{\texttt{init}, \texttt{up}, \texttt{set\_flag}\} \tag{2}$$

and

$$G \underset{\text{Delete}}{\Rightarrow} H \text{ implies } \#H \leq \#G - 1. \tag{3}$$

Claim: On every host graph G, the loop `Reduce!` terminates after at most $2 \times \text{size}(G)$ successful executions of `Reduce`.

Proof: By (2), the inner loop `up!` terminates as each application of `up` decreases the #-value which cannot get negative. Moreover, by (2) and (3), the command `try Delete else set_flag` either decreases the #-value (if `Delete` or `set_flag` is applied) or terminates the loop (if both `Delete` and `set_flag` fail). As the #-value cannot get negative, it follows that `Reduce!` must terminate. The number of successful executions of `Reduce` is at most $2 \times \text{size}(G)$ because by (3) and (2), both `Delete` and `set_flag` decrease the #-value, and $0 \leq \#G \leq 2 \times \text{size}(G)$ by (1). □

We now show that on every live graph G, `(init; Reduce!; Guard)!` terminates in $O(\text{size}(G))$ steps. This finishes the proof as the execution of `Check` requires at most two steps.

By (2) and (3), each application of `init` decreases the #-value and none of the rules in `Reduce` or `Guard` increases the #-value. Hence each successful execution of the loop body decreases the #-value. Together with the Claim and (1), this implies that the loop terminates after at most $2 \times \text{size}(G)$ successful executions of its body.

Hence, in total, both `init` and `flag` are applied at most $2 \times \text{size}(G)$ times. In addition, there may be one failed application of `init` or one execution of `break`.

By (2), (3) and (1), the rules `up` and `set_flag` and the rule set `Delete` are each applied at most $2 \times \text{size}(G)$ times in total. Each application of `Delete` involves up to six failed applications of rules from the set. Also, there are at most $2 \times \text{size}(G) + 1$ failed applications of `up` because, by the Claim, there are at most $2 \times \text{size}(G)$ successful executions of `Reduce`. By the same argument, there are at most $2 \times \text{size}(G) + 1$ failed applications of both `set_flag` and `Delete`. Each failed application `Delete` amounts to seven failed rule applications.

Thus, altogether, the loop `(init; Reduce!; Guard)!` terminates in a number of steps that is linear in $\text{size}(G)$). □

Theorem 3.25 (Correctness of `is-bin-dag`)**.** The program `is-bin-dag` (Figure 22) is totally correct with respect to the specification:

Input: An *input graph*.
Output: Fail if and only if the input is not a binary DAG.

Proof. Termination follows from Lemma 3.24. We claim that if the input graph is a binary DAG, then the program evaluates to the empty graph, and otherwise, evaluates to fail. In particular, this means the subprogram `(init; Reduce!; Guard)!` must evaluate to the empty graph, or a *white-rooted live graph*, respectively. We argue by induction of the number of iterations of the outer loop.

Suppose the loop runs no times. Then `init` failed, so the input graph was empty, so the subprogram evaluates to the empty graph. Suppose we can make progress, but `Guard` causes a break. Then then `init` must be executed, followed by `Reduce` zero or more times, then `flag` must be applicable, meaning the result must be a *white-rooted live graph*. By Lemma 3.23, this means the input graph must not have been a DAG. Finally, if we can make progress, and no break is executed, then we are ready to try to run another iteration, and again by 3.23, this particular iteration has preserved if the host graph was a binary DAG or not. Suppose now that the loop has run n times. If the loop cannot run again, then we repeat the argument above. Similarly, if we can make progress, then the argument above follows too. □

Lemma 3.26 (Complexity of `init`)**.** `init` terminates in constant time on an *unrooted live graph* G.

Proof. Exactly the same as the proof of Lemma 3.4. The search plan will iterate all nodes in G looking for the first grey unrooted node. Testing if a node is unrooted and grey takes only constant time. Since every node in an *input graph* is unrooted and grey, the search will stop at the first node, or fail in constant time if G is empty. □

Lemma 3.27 (Complexity of `up`)**.** `up` terminates in constant time on a *live graph* G.

Proof. The search plan will look for a root node. If there are no root nodes, matching immediately fails. Otherwise, the first root node is located in constant time, and the check that it is grey takes constant time. If it is not grey, then the matching algorithm will look for the next root node, and determines there are no more (Lemma 3.22) in constant time, and matching fails. In the case that the matching algorithm has determined the root node is grey, it will then check that the incoming degree is at least one and the outgoing degree is at most two (our application condition), which takes only constant time. If the degree checks fail, then the search plan looks for another root node, as before.

If the degrees are correct, next the search plan grabs the first incoming edge in constant time and checks if it is unmarked and proper in constant time. If it is not, then we grab the next incoming edge in constant time. Due to the fact that the outgoing degree is at most 2, there can only be at most two looped edges and due to Lemma 3.22, there can only be at most two proper edges that are not unmarked. So, we know that the search plan will find a suitable edge after a bounded number of attempts. After finding a suitable edge, the search plan then checks the source node is grey and unrooted in constant time, which it will be. □

Lemma 3.28 (Complexity of `Delete`)**.** `Delete` terminates in constant time on an *live graph* G.

Proof. `Delete` takes at most as long as the sum of the time needed for `del1`, `del1d`, `del21`, `del21d`, `del22`, `del22d`, `del0`.

We start with `del0`, the simplest. The search plan will look for a root node. If there are no root nodes, matching immediately fails. Otherwise, the first root node is located in constant time, and the check that it is grey takes constant time. If it is not grey, then the matching algorithm will look for the next root node, and determines there are no more (Lemma 3.22) in constant time, and matching fails. In the case that the matching algorithm has determined the root node is grey, it will then check that the incoming and outgoing degrees are zero, in constant time. If they are, we're done, and if not we must look for another root node, as before.

For `del1` and `del1_d`, the search plan will look for a root node. If there are no root nodes, matching immediately fails. Otherwise, the first root node is located in constant time, and the check that it is grey takes constant time. If it is not grey, then the matching algorithm will look for the next root node, and determines there are no more (Lemma 3.22) in constant time, and matching fails. In the case that the matching algorithm has determined the root node is grey, it will then check that incoming degree is zero and outgoing degree is one, in constant time. If they are not, we must look for another root node, which fails in constant time. Otherwise, we grab the first outgoing edge in constant time, and check that it has the appropriate mark and is proper. If it is not, we proceed to look for the next root node, as before. Otherwise, we then check the target node of the edge is grey and unrooted. If not, we return to looking for the next root, as before.

The story for the remaining rules is similar. The first difference is that the initial outgoing degree check is for exactly two, and an additional edge must be matched. For `del21` and `del21_d`, the search plan additionally checks of the second edge at the end, and for `del22` and `del22_d` the process of checking for a second edge and target node is just an exact repeat of what happened for the first edge and target node. □

Lemma 3.29 (Complexity of `set_flag`)**.** `set_flag` terminates in constant time on an *live graph* G.

Proof. The search plan will look for a root node. If there are no root nodes, matching immediately fails. Otherwise, the first root node is located in constant time, and the check that it is grey takes constant time. If it is not grey, then the matching algorithm will look for the next root node, and determines there are no more (Lemma 3.22) in constant time, and matching fails. □

Lemma 3.30 (Complexity of `flag`)**.** `Guard` and `Check` both terminate in constant time on a *live graph* G.

Proof. `Guard` and `Check` take only as long as `flag`. `flag`'s search plan will look for a root node, and find the first one in constant time, and check that it is unmarked in constant time. If it is unmarked, then matching succeeds. If it is not, then we look for the next root node, and determine there is no other root node in constant time (Lemma 3.22), and fail. □

Theorem 3.31 (Complexity of `is-bin-dag`)**.** The program `is-bin-dag` (Figure 22) terminates in linear time with respect to the *size* of its input.

Proof. Lemma 3.24 tells us that there is only a linear number of steps and Lemma 3.22 and its proofs tells us that the conditions under which the rules can be matched in constant time (Lemmata 3.26, 3.27, 3.28, 3.29, and 3.30) is always preserved. Thus, we have the required result. □

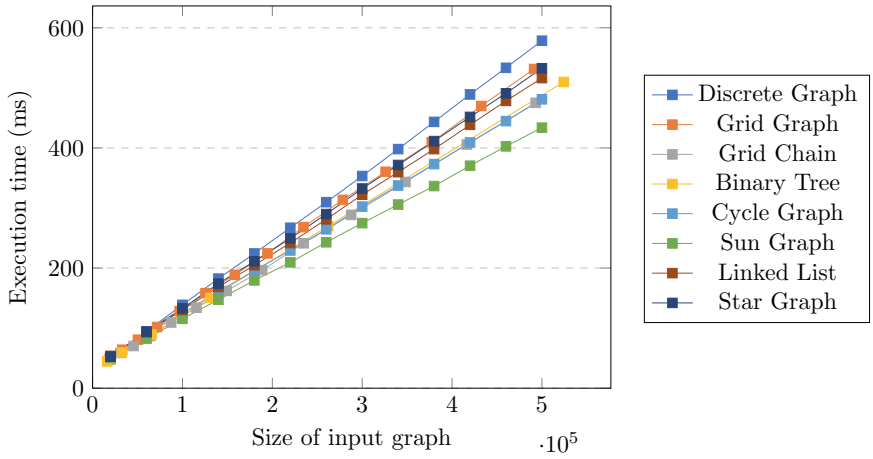


Figure 25: Measured performance of `is-bin-dag`

4. Fast DFS-Based Programs

In this section we will review (undirected) DFS (depth-first search) in GP 2, first implemented by Bak and Plump [14, 19, 15]. We will observe, by means of an introductory example, how it can be used to recognise connected graphs. We then consider the 2-colouring problem, producing a topological sorting for a connected DAG, and how to recognise a connected DAG. The concrete syntax for all the programs in this section is available on GitHub[5].

4.1. Recognising Connected Graphs using DFS

The program `is-connected` (Figure 26) can detect the connectedness of a graph. It fails if and only if its input graph is not connected. This can be achieved by conducting a DFS that turns grey nodes into non-grey ones. Since the DFS cannot propagate beyond the connected component it started in, the presence of a grey node indicates that the host graph is not connected.

First, we show correctness, by rigorously defining what we mean by an input graph, and then showing some intermediate results.

[5] `https://gist.github.com/GrahamCampbell/79f0f62c50d7de5e7ba739ad5d4581e5`

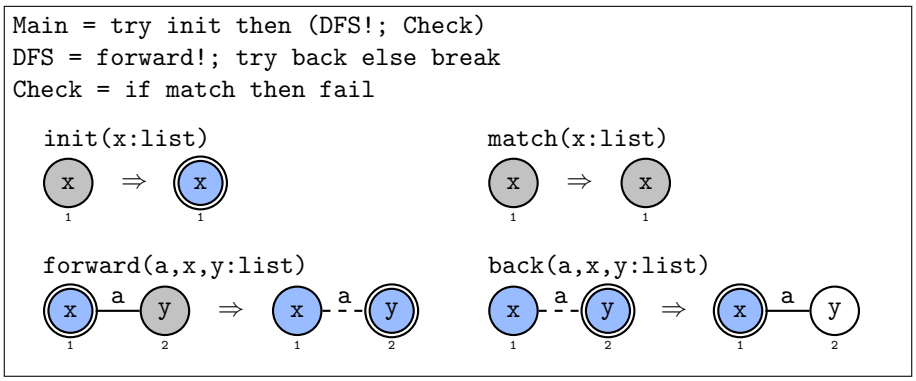


Figure 26: GP 2 program `is-connected`

Definition 4.1 (Input Graph)**.** An *input graph* (for the purposes of this subsection) is an arbitrarily labelled GP 2 *host graph* such that:

1. every node is marked grey;
2. every node is unrooted;
3. every edge is unmarked.

Lemma 4.2 (Invariant of `is-connected`)**.** Throughout the execution of the program `is-connected` on an *input graph*, all non-grey nodes in the *host graph* share a connected component.

Proof. The rule `init` is only called at the start of the program, and turns a grey node into a non-grey one. Subsequently, only applications of `forward` can turn grey nodes into non-grey ones, and only if `init` was successfully applied. So let us inductively show that the invariant is satisfied.

If `init` is applied at the start of the program, it introduces a single blue node into an *input graph*, which does not violate the invariant. If `init` fails, `forward` is never called, and the invariant is trivially satisfied due to a lack of successful rule applications.

Assume by induction that the invariant holds on the current host graph. An application of `forward` turns node `2` in the rule's left hand side blue. However, `forward` can only be applied if said node is adjacent to an existing blue node `1`. Hence it shares a connected component with the other non-grey nodes. □

Lemma 4.3 (Termination of `is-connected`)**.** On any host graph, the program `is-connected` terminates.

Proof. Since the loop body of `forward!` consists of a single rule, `forward` either applies and reduces the number of grey nodes in the host graph, or fails to find a match and terminates the loop. At some point, since the host graph is finite, there are no grey nodes left, and `forward` cannot match, terminating the loop.

For the termination of the loop `DFS!`, consider a couple $\#(G)$ consisting of the number of grey nodes of a host graph G, and the number of dashed edges

of G in that order. By *reducing* the measure # we mean that after changing a host graph G to a graph H, we have $\#(G) > \#(H)$ with respect to the lexicographical ordering, i.e. $\langle a, b\rangle < \langle c, d\rangle$ if either $a < c$ or both $a = c$ and $b < d$.

When calling `DFS` on a host graph G, because of `try back else break`, either `back` is applied, or the loop terminates. When `back` is applied, the measure # is reduced. Indeed, if `forward` is applied at least once the number of grey nodes is reduced (`back` does not modify the number of grey nodes). And if `forward` is not applied, the number of grey nodes remains the same, but the number of dashed edges decreases.

Due to host graphs being finite, # cannot be reduced anymore at some point, which means `back` cannot be applied. Hence `break` is invoked and the loop terminates. □

Lemma 4.4 (Existence of a Non-Grey Connected Component)**.** In the output graph of `try init then DFS!` executed on an *input graph*, there is a connected component consisting of non-grey nodes.

Proof. The lemma is trivially true for an empty *input graph*. In the case of an *input graph* consisting of a single node, `init` marks the entire graph non-grey, satisfying the lemma. So we can assume the input contains at least two nodes.

Assume for the sake of a contradiction that all connected components have at least one grey node in the output graph. Since the input is nonempty, `init` is applied. Consider the connected component of the node `init` was applied to. Let u and v be non-grey and grey nodes of the output graph that are adjacent, respectively. They exist because they share a connected component that has at least one non-grey node (application of `init`) and at least one grey node (assumption). We aim to show that u and v are matched by `forward` during the execution, contradicting our assumption.

If u is unmarked, it must have been matched by `back`. Right before that happened, u must have been a blue root, and `forward` cannot have been applicable to it (note that there is at most one root in the host graph at any given time). However, since v is grey and adjacent to u, `forward` must have been applicable, which is a contradiction. So we may assume u is blue.

Since u is blue in the output graph, it must have been matched by either `init` or `forward` at some point. Either way, after the rule application, u is a blue root, and the program is executing the loop `forward!`. Since `forward` can be applied to u and v, u must have an unmarked neighbour w different from v, otherwise v is marked blue.

For the next argument, let us take a look at the data structure the program creates. The dashed edges form a path of blue nodes, where a node at an end is rooted. This can be seen as a stack of blue nodes, where the root represents the top. Indeed, `init` initialises the stack, `forward` implements the *push* operation, and `back` the *pop* operation. Note however that `back` leaves the popped node with a blue mark, meaning it cannot be pushed again. This prevents the path from becoming a cycle, and also means that throughout the execution of `DFS!`,

the number of blue nodes is reduced. Since the *host graph* has finitely many nodes, `forward` is not applicable anymore at some point, due to unmarked nodes reachable by `forward` being gone. So eventually, only `back` can be applied, popping the top of the stack until the loop terminates.

Coming back to u, v, and w, this reasoning can be applied to the loop `DFS!` from the point where u is first rooted onward, in the subgraph of the nodes reachable from w without going through u. By that reasoning, at some point, the top of the stack is popped until u is the top again. When this happens, `back` is applied and the loop `DFS!` enters its next iteration, which starts with `forward!`. This leaves us in the same situation as previously, where we must assume u has another unmarked neighbour, distinct from v and w. However at some point, there will be no unmarked neighbours to apply the previous argument to (since host graphs are finite), so v will have to matched by `forward`, contradicting our assumption. □

Finally, we show correctness and complexity of `is-connected`.

Theorem 4.5 (Correctness of `is-connected`)**.** The program `is-connected` (Figure 26) is totally correct with respect to the specification:

Input: An *input graph*.
Output: Fail if and only if the input consists of more than one connected component.

Proof. Termination follows from Lemma 4.3. For correctness, first assume *input graph* G has no connected components, i.e. G is the empty graph. Then `init` cannot be applied, and the procedure `Check` is called. The rule `match` cannot be applied either, so the program terminates without failing.

Assume G has exactly one connected component. We know by Lemma 4.3 that `DFS!` terminates. Furthermore, by Lemma 4.4, the output H of `try init then DFS!` has a connected component whose nodes are blue. Since no rule adds or deletes nodes or edges, H is isomorphic to G ignoring marks and roots. Hence the blue connected component must be the entirety of H. The procedure `Check` is called, and `match` cannot find a match in a graph containing only blue nodes. Hence `is-connected` does not fail.

Assume G has more than one connected component. The loop `DFS!` still terminates by Lemma 4.3. Furthermore, by Lemma 4.4, the output H of `try init then DFS!` has a connected component C with blue nodes. Since by Lemma 4.2, all blue nodes share the same connected component, and since H consists of more than one connected component, there is a grey node in $H - C$. Hence the rule `match` matches and the program fails. □

Let us now examine the complexity of `is-connected`. To do this, we consider the following measures of complexity. Let `r` be a rule. Let $s(\texttt{r})$ be an upper bound on the number of steps involving `r`, i.e. how many times `r` is called, whether it is successfully applied or not. Note that the number of steps depends on the program it is called in and the class of input graphs the program is run on. Let $t(\texttt{r})$ be an upper bound on the number of possible matches of `r`

the program considers during its execution according to a GP 2 implementation satisfying Theorem 2.2. We define $K(\mathtt{r}) = s(\mathtt{r}) \cdot t(\mathtt{r})$ to serve as a measure of complexity of a rule. Consider the complexity measure $K(\mathtt{p})$ of a program `p` defined as the sum of terms $K(\mathtt{r})$, where `r` ranges over the rules called by `p`. Note that $K(\mathtt{p})$ is an upper bound, and hence not unique.

Theorem 4.6 (Complexity of `is-connected`)**.** On a class of *bounded degree input graphs*, the program `is-connected` (Figure 26) terminates in linear time with respect to the *size* of its input.

Proof. Since none of the rules add or delete edges, linearity of the *input graph* is equivalent to linearity of all host graphs during the program's execution. So in this proof, we shall call a number that is linear in the number of nodes of the input simply *linear*.

Furthermore, in order for Theorem 2.2 to be applicable, there can only be a constant number of roots in the host graph. This is indeed the case. The only rule that does not preserve the number of roots is `init`, which is only called once. So the host graph can have at most one root at any time.

Let us show the linearity of $K(\mathtt{Main})$ by showing that K of all the rules is linear.

The rule `init` is only called once. In the worst case, there is no match for it, and every node of the *host graph* has to be considered for a match. Hence $K(\mathtt{init})$ is linear.

Similarly, the rule `match` is only called once, and the program has to consider each node of the *host graph* for a possible match in the worst case, making $K(\mathtt{match})$ linear as well.

By Theorem 2.2, `back` matches in constant time on bounded degree graphs, hence $t(\mathtt{back})$ is constant. Let us now examine how many times the rule `back` is called. Since the loop `DFS!` terminates after `back` is successfully matched (see Lemma 4.3), we know that `back` succeeds at each call except for the final one, i.e. it has a constant number of unsuccessful applications. So it is enough to show the linearity of the number of successful applications of the rule. It is easy to see that `back` increases the number of unmarked nodes, while all other rules preserve it. Since `back` cannot match an unmarked node, it can only be applied a linear number of times. Hence $K(\mathtt{back})$ is linear.

The rule `forward` is also matched in constant time by Theorem 2.2, meaning $t(\mathtt{back})$ is constant. Furthermore, by the previous paragraph, `back` is called a linear number of times, meaning the loop `DFS!` has a linear number of iterations. During each of these iterations, `forward` fails exactly once (termination of `forward!` by Lemma 4.3), meaning the linearity of its calls is equivalent to the linearity of its successful applications. The rule `forward` decreases the number of grey nodes, while the other rules called in `DFS!` preserve it. Since it needs a grey node to match successfully, it can only do so a linear number of times. Hence $K(\mathtt{forward})$ is linear. □

Finally, we have collected empirical timing results, supporting our claim that the program runs in linear time on graph classes of bounded degree, but not

necessarily on those that do not have bounded degree (Figures 27 and 28).

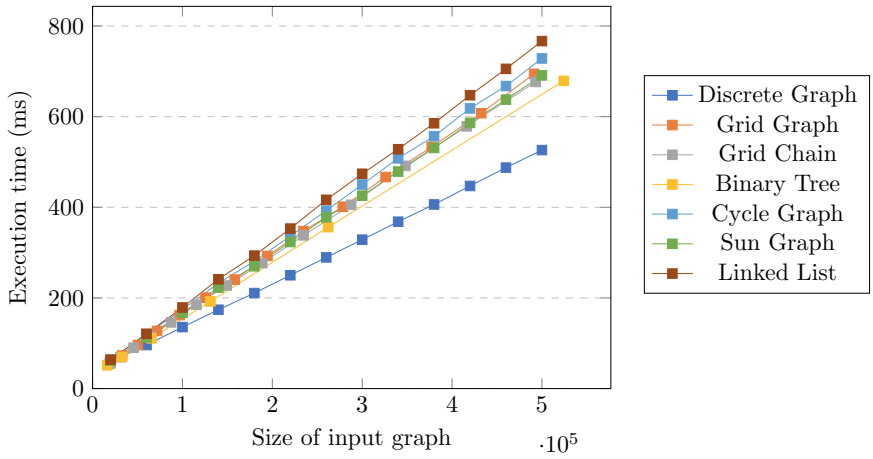


Figure 27: Measured performance of `is-connected`

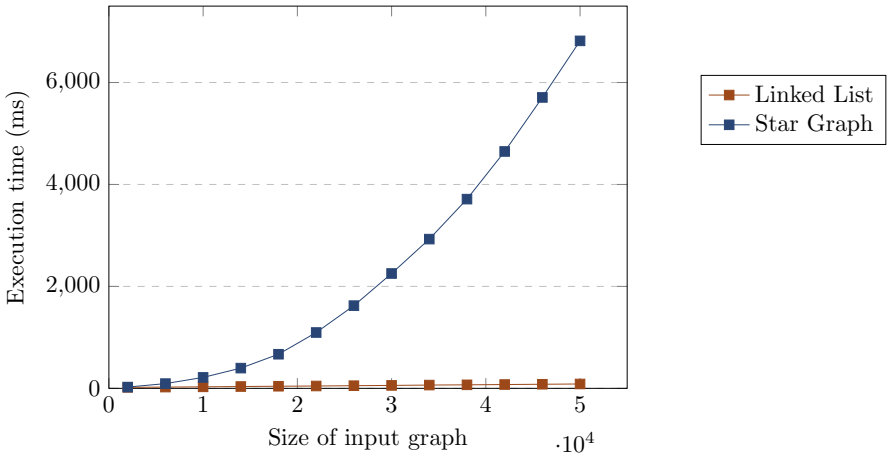


Figure 28: Measured performance of `is-connected`

4.2. The 2-Colouring Problem

Vertex colouring has many applications [23] and is among the most frequently considered graph problems. In 2016 Bak and Plump investigated the possibility of an efficient rule-based algorithm for the 2-colouring problem [15]. We recall this important case study, and provide further empirical evidence for the linear time complexity on graph classes of bounded degree of the compiled program generated by the latest version of the GP 2-to-C compiler.

In order to rigorously define what a 2-colouring program is, in the context of GP 2, we first define what our notion of an input graph should be, and what a 2-colouring is.

Definition 4.7 (Input Graph)**.** An *input graph* (for the purposes of this subsection) is an arbitrarily labelled, *connected* GP 2 *host graph* such that:

1. every node is marked grey;
2. every node is unrooted;
3. every edge is unmarked.

Definition 4.8 (A 2-Colouring)**.** A *2-colouring* H of an *input graph* G is obtained by colouring each of the nodes either red or blue such that no red (blue) node is adjacent to a red (blue) node, respectively.

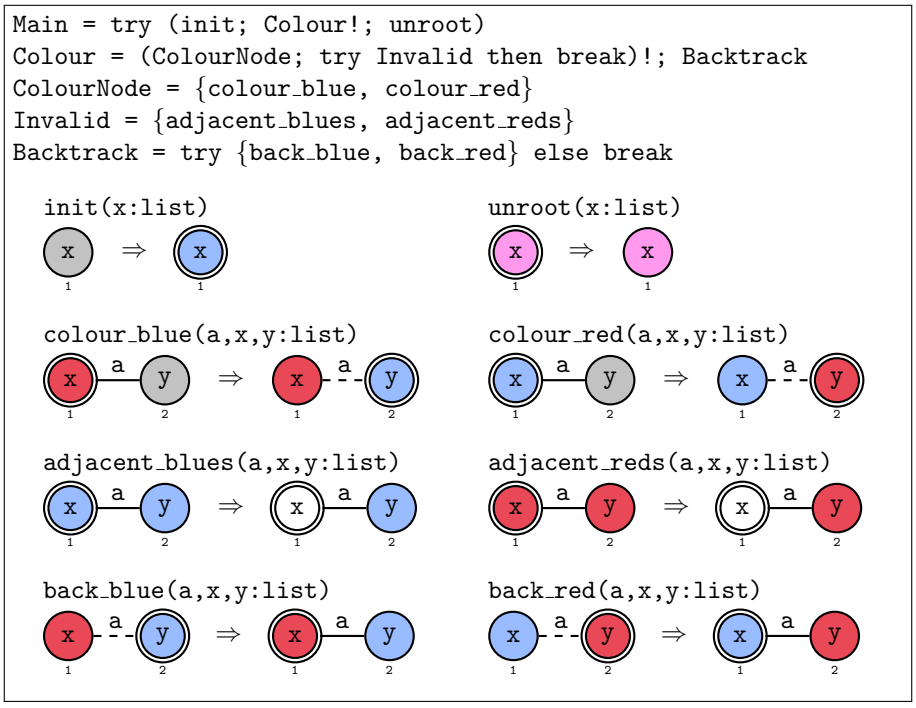


Figure 29: GP 2 program `2-colour`

Theorem 4.9 (Correctness of `2-colour` [19])**.** The program `2-colour` (Figure 29) is totally correct with respect to the specification:

Input: An *input graph*.
Output: Output a *2-colouring* of the input if one exists, otherwise output the input graph unmodified (up to isomorphism).

Theorem 4.10 (Complexity of `2-colour` [19])**.** On a class of *bounded degree input graphs*, the program `2-colour` (Figure 29) terminates in linear time with respect to the *size* of its input.

Note that while the program is only correct on connected graphs, it can be modified to work on arbitrary graphs too, but at a cost. Not only does the program become more complex, but the linear time complexity result fails also, due to there being no way to iterate all the connected components in linear time. Finally, we have collected empirical timing results, supporting our claim that the program runs in linear time on graph classes of bounded degree, but not necessarily on those that do not have bounded degree (Figures 30 and 31).

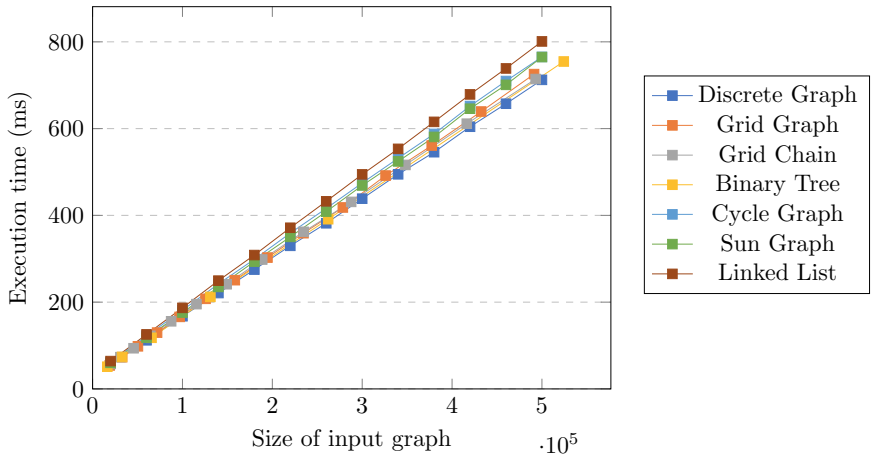


Figure 30: Measured performance of `2-colour`

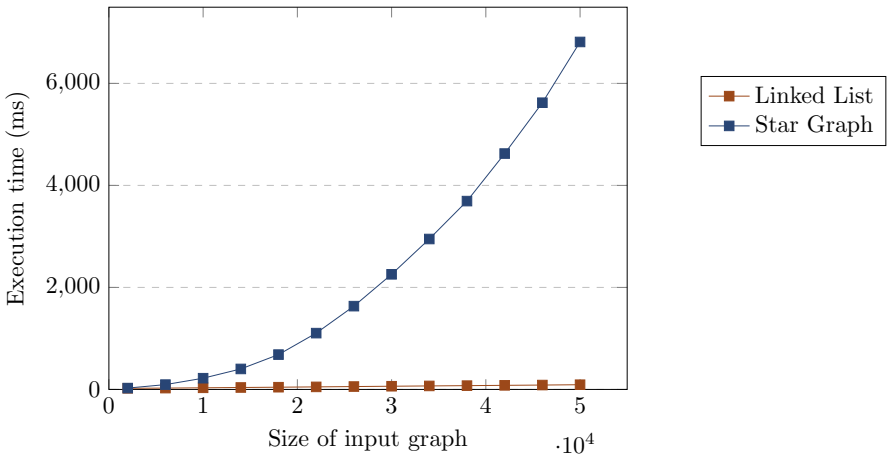


Figure 31: Measured performance of `2-colour`

4.3. Topological Sorting and Recognising DAGs

The GP 2 program `top-sort` (Figures 32, 33, 34) presented in this section has two purposes: recognising whether its connected input graph is a DAG (directed acyclic graph) and if it is, producing a topological sorting of said graph.

The class of *DAGs (directed acyclic graphs)* consists of all graphs that do not contain a directed cycle as a subgraph. A *topological sorting* of a DAG G is a total order (an antisymmetric, transitive, and connex binary relation) $\leq$ on the set of nodes of G, such that for each edge of source u and target v, $u \leq v$ (*topological property*). Topological sortings cannot exist for graphs containing directed cycles, since there is no way to define a total order on the nodes of a cycle such that the topological property is satisfied.

The program uses depth-first search to traverse the host graph in linear time while testing whether it is a DAG and constructing a path of blue edges that define a topological sorting. Moreover, it terminates in linear time on inputs of bounded node degree.

Example executions of `StackNodes!` and `LoopNodes!` can be found in Figures 35 and 36 respectively.

Let us first define what an input graph is for the purposes of this subsection, and then show the program terminates.

Definition 4.11 (Input Graph)**.** An *input graph* (for the purposes of this subsection) is an arbitrarily labelled, *connected* GP 2 *host graph* such that:

1. every node is marked grey;
2. every node is unrooted;
3. every edge is unmarked.

Lemma 4.12 (Termination of `top-sort`)**.** On any host graph, the program `top-sort` terminates.

Proof. Consider the loop {`forward1, forward2`}`!` called in the procedure `StackNodes` (Figure 32). In each iteration, either `forward1` is applied, `forward2` is applied, or both fail and the loop terminates. Whenever one of these rules is applied, the number of grey nodes in the host graph is reduced. Due to host graphs being finite, there are no grey nodes left eventually, and neither rule can match, terminating the loop.

Next, consider the loop `StackNodes!` (Figure 32), a measure # consisting of the number of grey nodes of a host graph paired with the number of dashed edges, and a lexicographical ordering on said pairs. In each iteration, either `back` is applied or the loop terminates due to `break`. If `back` is applied, either the number of grey nodes remains the same and and the number of dashed edges is reduced (i.e. neither `forward1` nor `forward2` are applied), or the number of grey nodes is reduced (i.e. `forward1` or `forward2` have been applied at least once). In either case, # is reduced. Since host graphs are finite, # cannot be reduced anymore at some point, hence `back` is not applicable. Then `break` is invoked and the loop terminates.

```
Main =
  try init then (
    StackNodes!;
    unroot;
    LoopNodes!;
    if flag then fail
  )
StackNodes =
  {forward1, forward2}!;
  try back else break
```

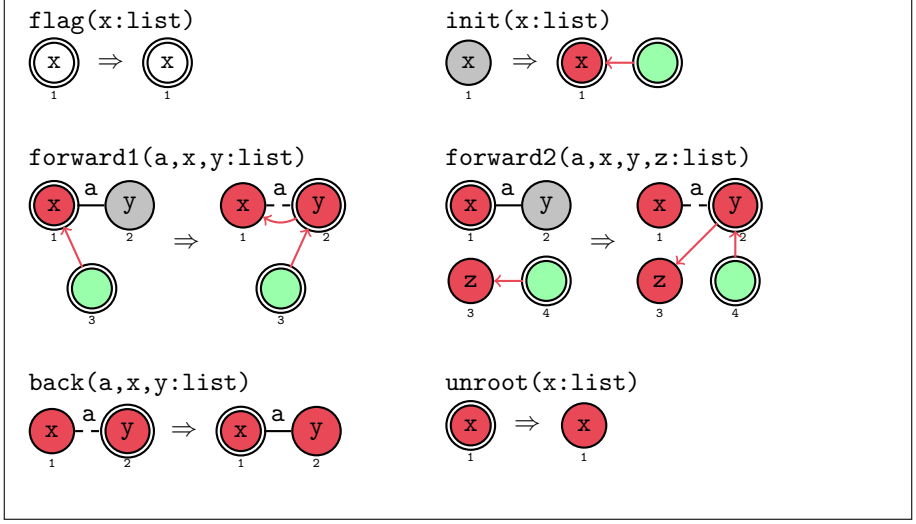


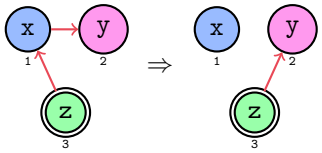
Figure 32: GP 2 program `top-sort` and procedure `StackNodes`

Now consider the loop `forward!` in the procedure `SortNodes` (Figure 34). In each iteration, either `forward` is applied, or the loop terminates. Applying `forward` reduces the number of red nodes in the host graphs. Since there are only finitely many, there will be no red nodes left for `forward` to match eventually. Hence the loop has to terminate.

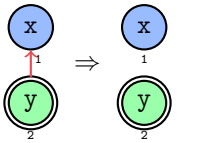
Next, consider the loop `SortNodes!` (Figure 34). In each iteration, either `back_push` applies, `back_first_push` applies, or `break` is invoked. We claim that each iteration either lexicographically reduces the number of red nodes in the host graph paired with the number of grey nodes (let us call this measure #), or terminates the loop. The rules `set_flag`, `grey_push`, and `grey_first_push` only get called in an iteration that terminates the loop, so we do not need to consider them for the purpose of reducing #. Similarly, as rules called in the condition of an `if` statement do not modify the host graph, we do not need to consider `loop`, `two_cycle`, and `back_edge` for reduction purposes either. So consider an iteration where either either tttback_push or `back_first_push` ap-

```
LoopNodes =
  if flag then break;
  try skip1 else (
    try skip2 else (
      try init1 then (
        SortNodes!
      ) else (
        try init2 then (
          SortNodes!
        ) else (
          break
        )
      )
    )
  )
```

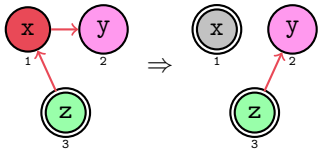
skip1(x,y,z:list)



skip2(x,y:list)



init1(x,y,z:list)



init2(x,y:list)

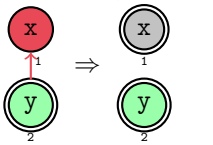


Figure 33: GP 2 procedure `LoopNodes`

plies. If `forward` is applied at least once this iteration, the number of red nodes is reduced, which reduces #. If `forward` is not applied at all, the number of red nodes remains the same while either `back_push` or `back_first_push` reduces the number of grey nodes, reducing #. So eventually, # cannot be reduced any further in the finite host graph, meaning neither `back_push` nor `back_first_push` can be applied, causing the loop to terminate.

Finally, consider the loop `LoopNodes!` (Figure 33). In each iteration, either `break` is invoked, or one of the rules `skip1`, `skip2`, `init1`, and `init2` is applied. Each of these rules reduces the number of red edges in the host graph. So eventually, since host graphs are finite, there are no red edges that can be matched by these rules anymore, so all of them will fail, meaning `break` is invoked due to the structure of the nested `try` statements, and the loop terminates. □

```
SortNodes =
  forward!;
  if {loop, two_cycle, back_edge} then (
    set_flag;
    break
  );
  try back_push else (
    try back_first_push else (
      try grey_push else (
        grey_first_push
      );
      break
    )
  )
```

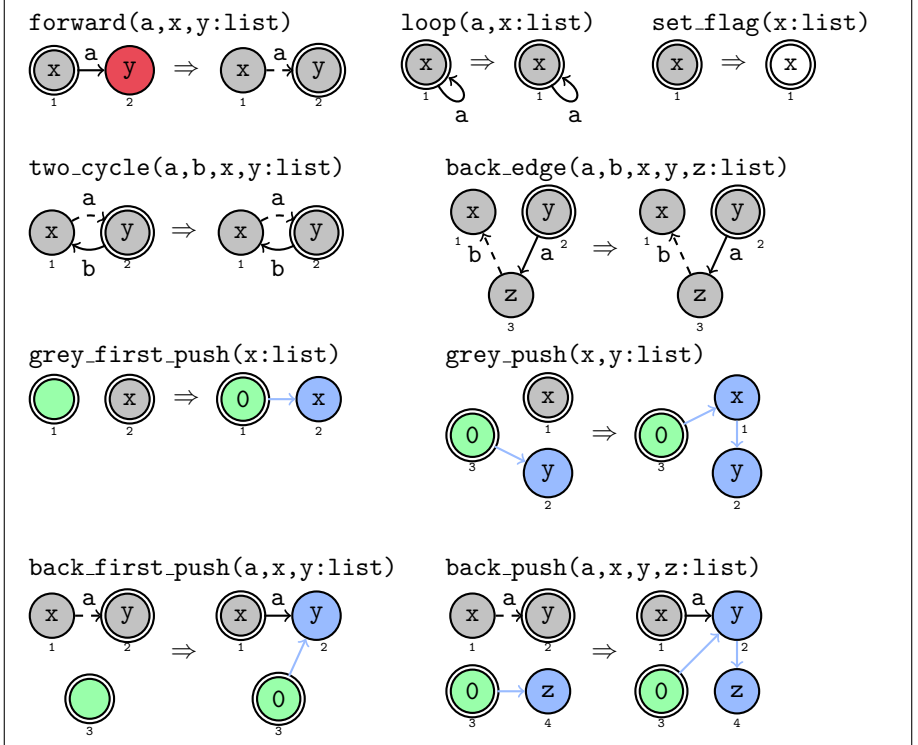


Figure 34: GP 2 procedure `SortNodes`

The command `init; StackNodes!` creates a stack of nodes in order to navigate between strongly connected components while doing a depth-first search that travels in the direction of the edges. It is equivalent to `try init then`

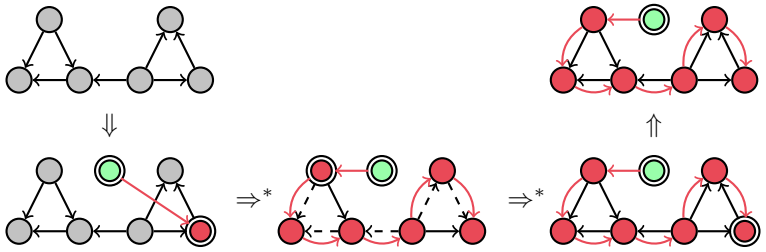


Figure 35: Example execution of `StackNodes!`

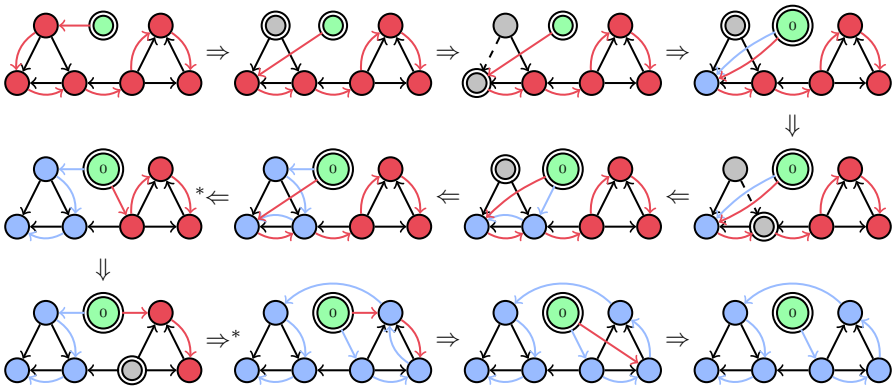


Figure 36: Example execution of `LoopNodes!`

`StackNodes!` if `init` is applied successfully. We shall define stacks via their implementation.

A *stack* is a finite set of red nodes connected by red edges such that the red edges form a path that does not self-intersect. The node in the path that has no incoming red edge from another path node is called the *top* of the stack. Additionally, there is an unlabelled green root node called the *pointer* with only one adjacent edge, namely an outgoing red edge whose target is the top of the stack.

Note that such a stack can also be defined with blue nodes and edges instead of red ones, in which case we shall call it a *blue stack*. In fact, during the execution of `LoopNodes!`, a red and a blue stack coexist using the same green root.

Lemma 4.13 (Correctness of `init; StackNodes!`)**.** The command sequence `init; StackNodes!` is totally correct with respect to the specification:

Input: An *input graph* G.
Output: G where all its nodes are in a *red stack*.

Proof. Termination follows from Lemma 4.12.

The proof that all nodes of G are marked red is analogous to that of Lemma 4.4.

To show that a correctly encoded stack is formed, let us proceed by induction. The rule `init` creates a valid stack containing a single node. Now assume a valid stack is encoded in the host graph. Let us argue that after applying `StackNodes`, that is still the case. Whenever a red edge is created, the target is the top (since it is adjacent to the green root), and the source is a grey node (and hence not part of the path already, since an *input graph* has grey nodes), extending the non-self-intersecting path. The green root now points toward the newly added node, making it the new top. □

Let us now define how to represent a topological sorting in the context of GP 2.

Definition 4.14 (Topological sorting as a graph structure)**.** Consider a graph G and the set of its blue nodes B. Define a binary relation on B by $u \leq v$ if there is a blue edge from u to v, or if $u = v$. G contains a *topological sorting* if the transitive closure of $\leq$ is a topological sorting of the subgraph of G induced by the blue nodes B and the unmarked edges.

With such a structure, one can test whether two nodes are related with a topological sorting by checking whether there is a path of blue edges connecting them.

Lemma 4.15 (Correctness of `LoopNodes!`)**.** On a graph whose nodes are all in a red stack, and whose subgraph induced by its unmarked edges is a DAG, `LoopNodes!` outputs a graph G that contains an unmarked root or a topological sorting.

Proof. Termination follows from Lemma 4.12.

Let us assume G does not contain an unmarked root, and show that it does contain a topological sorting. In fact, since no rules called in `LoopNodes!` can mark or unroot an unmarked node, we can assume that no unmarked root is introduced at any point, i.e. `set_flag` is not applied.

Consider the binary relation $\leq$ on the set of blue nodes defined by blue edges as in Definition 4.14, and let us show it is a topological sorting. It is transitive since it is defined as a transitive closure.

Antisymmetry follows from the fact that $\leq$ is reflexive and the fact the subgraph H of G induced by blue edges does not contain directed cycles. Indeed, H behaves like a stack of blue nodes and edges with a green root pointing towards the top with a blue edge. When the green root is unlabelled, the stack is initialised as a single blue node, and the green root is labelled `0`. Once that label has been established, non-blue nodes are pushed. At no point does the program pop a blue node, or change the mark of a blue node. Hence no blue cycle can be introduced.

Connexity follows from the fact that every node is eventually marked blue, i.e. pushed. By using arguments analogous to those in Lemma 4.4, we can conclude that after `SortNodes!` is applied, the grey root and all nodes reachable from it are marked blue (which we can only conclude because the `if` statement does not change the host graph since we assume `set-flag` is not applied). This

difference is because the steps of the depth-first search are sensitive to edge direction. In order to sort through remaining nodes, `LoopNodes!` skips over blue, i.e. already sorted nodes in the red stack with `skip1` and `skip2`, until it reaches a red, i.e. unsorted node, which is then initialised as grey root with `init1` or `init2`. Then `StackNodes!` is called on that grey root. Since all nodes of the input graph are in the red stack by Lemma 4.13, all nodes are eventually marked blue.

It remains to show that $\leq$ satisfies the topological property, namely that for each unmarked edge from u to v in the input, $u \leq v$, i.e. there is a blue path from u to v in the output graph. Since the blue edges form a stack of all nodes, it is enough to show that u is pushed after v. Consider the iteration of `SortNodes!` that pushes u onto the blue stack with one of the push rules (`grey_first_push`, `grey_push`, `back_first_push`, and `back_push`). The node u has no outgoing unmarked edge with a red node as the source because then `forward!` would have had at least one more iteration, and this would not be the iteration of `SortNodes!` that pushes u. So v is not red. It cannot be grey either because then there would be a path of dashed edges from v to u (since the grey nodes are in a path of dashed edges, and the root, u, is the final node of the path), which would mean there was a cycle of unmarked edges in the input. So v must be blue, i.e. it is pushed before u. □

Now let us show the total correctness of `top-sort`. Note that we include the empty graph in the definition of DAGs. If one wishes to exclude it from the class of DAGs, it suffices to add the `else fail` to the `try` statement in `Main`, since `init` fails on the empty graph.

Theorem 4.16 (Correctness of `top-sort`)**.** The program `top-sort` (Figures 32, 33, 34) is totally correct with respect to the specification:

Input: An *input graph.*
Output: Fail if the input is not a DAG, and G equipped with a topological sorting otherwise.

Proof. If G is the empty graph, `init` is not successfully applied, and the output is G, which defines a valid topological sorting of the empty DAG.

Termination follows from Lemma 4.12.

If G is a DAG, and no unmarked root is introduced in `LoopNodes!`, it follows from the same lemmata and the fact that no rule of `top-sort` changes the structure of the underlying graph of unmarked edges, that the output is G containing a topological sorting. So we need to show that, if G is a DAG, `LoopNodes!` does not introduce an unmarked root. Conversely, if G is not a DAG, we need to show that an unmarked root is introduced in `LoopNodes` because matching `flag` is the only way for `Main` to fail (`unroot` always matches since `SteckNodes` leaves a red root in the host graph).

The only rule that introduces an unmarked root is `set_flag`, which is only called during the `if` statement in `SortNodes`, if the condition is satisfied. So it is enough to show that G is a DAG if and only if neither `loop`, `two_cycle`, nor `back_edge` matches.

As argued in the proof of Lemma 4.15, every non-pointer node is pushed onto the blue stack with one of the push rules. So the `if` statement called right before the push rules are invoked for each non-pointer node while it is a grey root.

If G is a DAG, `loop` and `two_cycle` cannot match since they need a 1-cycle or 2-cycle respectively to be present in G. The rule `back_edge` cannot match either. It contains a path from node 2 to node 1. As the target of a dashed edge, node 1 is in the stack of dashed edges, so there is a path to node 1 to node 2, the top of the stack. This means there is a cycle.

Conversely, assume that G is not a DAG. If it contains a 1- or 2-cycle, either `loop` or `two_cycle` matches. So assume G contains a cycle of length at least 3. Consider the first time a node of that cycle becomes a grey root due to `forward`. Eventually, `forward` is applied to make the next node in the cycle the grey root. We can repeat this argument until the last node in the cycle is the grey root (in the cycle, all edges but one are dashed, all nodes are grey, and the node with an outgoing unmarked edge is rooted). Then `back_edge` can match. □

For the complexity of `top-sort`, let us return to the measures defined in Subsection 4.1, namely $s(\mathtt{r})$ (an upper bound on the number of steps of a rule `r`, i.e. the number of times `r` is called during the execution of its program), $t(\mathtt{r})$ (an upper bound on the number of possible matches for a rule r that have to be considered), and $K(\mathtt{r}) = s(\mathtt{t}) \cdot t(\mathtt{r})$. We shall measure the complexity of a program or procedure `p` with $K(\mathtt{p})$ (the sum of $K(\mathtt{r})$ over the rules `r` called in `p`) as a function of the size of the input graph.

Theorem 4.17 (Complexity of `top-sort`)**.** On a class of *bounded degree input graphs*, the program `top-sort` (Figures 32, 33, 34) terminates in linear time with respect to the *size* of its input.

Proof. Let us first argue that there is a constant number of roots at any given time, so that we can apply Theorem 2.2. The rule `init` introduces a green root. In all other rules, the rootedness and mark no green nodes get modified, and no green nodes get introduced. The rule `init` also introduces a red root. In `StackNodes!`, no rules modify the number of red roots. Then with `unroot`, the one red root is removed. The rules `init1` and `init2` introduce a grey root. Whenever one of them is applied, `SortNodes!` is called. Let us show that `SortNodes!` removes the grey root. This loop can only terminate when `break` is invoked (termination itself has been shown in Lemma 4.13), or when `grey_first_push` does not find a match. In the latter case, either the grey root has already been removed (which is what we want), or the green root has a label. If the green root as a label, `grey_push` or `back_push` would have been applied, and `grey_first_push` never called. Now consider the case where `break` is invoked. This must be preceded by a successful application of `set_flag`, `grey_push`, or `grey_first_push`. In the latter two cases, the grey root is unrooted. In the former case, the grey root is unmarked, and various `break` and `fail` statements are invoked and the program terminates. In either case, the number of roots remains constant.

Let us now show that $K(\texttt{top-sort})$ is linear in the size of the input graph by showing that for each rule `r` called by `top-sort`, $K(\texttt{r})$ is linear.

First consider `init`. It is called once, and every node of the input graph is a valid match, so $K(\texttt{init})$ is constant. Now for each `r` rule apart from `init`, `r` is a fast rule, and hence by Theorem 2.2, $t(\texttt{r})$ is constant. So it remains to show that the number of calls $s(\texttt{r})$ is at most linear.

The rules `unroot` and `flag` are called at most once.

Note that {`forward1, forward2`} is only applied a linear number of times since it reduces the number of grey nodes, of which there can be only linearly many, and no other rule in `StackNodes!` introduces grey nodes. So `back` can only be applied a linear number of times. And the number of times `back` is called but not applied is once, since in that case `break` is invoked. Hence $s(\texttt{back})$ is linear. This also means that the number of iterations of `StackNodes!` is linear. So the number of times `forward1` and `forward2` are called but not applied can only be linear as well. Hence $s(\texttt{forward1})$ and $s(\texttt{forward2})$ are linear.

Therefore there can only be linearly many red edges. In each successful iteration of `LoopNodes!`, one of `skip1`, `skip2`, `init1`, `init2` has to be applied, reducing the number of red edges. So there can only be linearly many iterations of `LoopNodes`, making $s(\texttt{skip1})$, $s(\texttt{skip2})$, $s(\texttt{init1})$, and $s(\texttt{init2})$ linear.

The rule `forward` can only be applied a linear number of times since it reduces the number of red nodes, and no other rule in `LoopNodes!` modifies that number. So there can only be linearly many dashed edges, meaning that combined, `back_first_push` and `back_push` can only be applied a linear number of times. Hence there can only be a linear number of calls of `SortNodes`. So $s(\texttt{r})$ is linear for each rule `r` called only once in `SortNodes`. That also means that the number of times `forward` is called but not applied is also linear, so $s(\texttt{forward})$ is linear. □

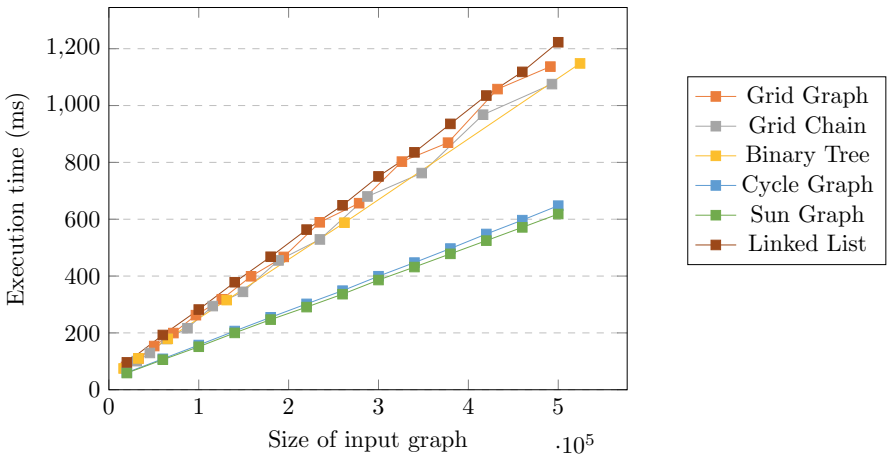


Figure 37: Measured performance of `top-sort`

Like `2-colour`, `top-sort` is only correct on connected graphs, and can similarly be modified to work on arbitrary graphs. Finally, we have collected empirical timing results, supporting our claim that the program runs in linear time on classes of connected graphs of bounded degree (Figure 37).

5. Conclusion

The polynomial cost of graph matching is the performance bottleneck for languages based on standard graph transformation rules. GP 2 mitigates this problem by providing rooted rules which under mild conditions can be matched in constant time. We present rooted GP 2 programs of two types: (1) graph reduction programs which recognise cycle graphs, trees and binary DAGs, and (2) depth-first search programs for checking connectedness and acyclicity resp. for producing a topological sorting. The programs are proved to be correct and to run in linear time either on arbitrary input graphs (in the case of reduction programs) or on graphs of bounded node degree (in the case of depth-first search programs). The proofs demonstrate that graph transformation rules provide a convenient and intuitive abstraction level for formal reasoning on graph programs. We also give empirical evidence for the linear run time of the programs, by presenting benchmark results for graphs of up to 500,000 nodes in various graph classes. For acyclicity checking and topological sorting, the linear behaviour is achieved by implementing depth-first search strategies based on an encoding of stacks in graphs.

In future work, we intend to investigate for more graph algorithms whether and under what conditions their time complexity in conventional programming languages can be reached in GP 2. The more involved the data structures of those algorithms are, the more challenging will be the implementation task. This is because in GP 2, the internal graph data structure is (intentionally) hidden from the programmer and hence any data structures used by an algorithm need to be encoded in host graphs. A simple example of this is the encoding of stacks as linked lists in the programs for acyclicity checking and topological sorting.

Additional future work is the refinement of unrooted programs into more efficient rooted programs. It is not obvious how to do this in general, or what refinement tactics could be used.

The programs using depth-first search need host graphs of bounded node degree in order to run in linear time. A topic for future work is therefore to find a mechanism that allows us to overcome this restriction. Clearly, such a mechanism will require to modify GP 2 and its implementation.

References

[1] H. Ehrig, C. Ermel, U. Golas, F. Hermann, Graph and Model Transformation, Monographs in Theoretical Computer Science, Springer, 2015. doi:10.1007/978-3-662-47980-3.

[2] O. Runge, C. Ermel, G. Taentzer, AGG 2.0 – new features for specifying and analyzing algebraic graph transformations, in: Proc. 4th International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011), Vol. 7233 of Lecture Notes in Computer Science, Springer, 2011, pp. 81–88. doi:10.1007/978-3-642-34176-2_8.

[3] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo, The design of a language for model transformations, Software & Systems Modeling 5 (3) (2006) 261–288. doi:10.1007/s10270-006-0027-7.

[4] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon, M. Zimakova, Modelling and analysis using GROOVE, International Journal on Software Tools for Technology Transfer 14 (1) (2012) 15–40. doi:10.1007/s10009-011-0186-x.

[5] E. Jakumeit, S. Buchwald, M. Kroll, GrGen.NET – the expressive, convenient and fast graph rewrite system, International Journal on Software Tools for Technology Transfer 12 (3–4) (2010) 263–271. doi:10.1007/s10009-010-0148-8.

[6] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced concepts and tools for in-place EMF model transformations, in: Proc. 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Vol. 6394 of Lecture Notes in Computer Science, Springer, 2010, pp. 121–135. doi:10.1007/978-3-642-16145-2_9.

[7] M. Fernandez, H. Kirchner, I. Mackie, B. Pinaud, Visual modelling of complex systems: Towards an abstract machine for PORGY, in: Proc. 10th Conference on Computability in Europe (CiE 2014), Vol. 8493 of Lecture Notes in Computer Science, Springer, 2014, pp. 183–193. doi:10.1007/978-3-319-08019-2_19.

[8] D. Plump, The design of GP 2, in: Proc. 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011), Vol. 82 of Electronic Proceedings in Theoretical Computer Science, 2012, pp. 1–16. doi:10.4204/EPTCS.82.1.

[9] D. Plump, From imperative to rule-based graph programs, Journal of Logical and Algebraic Methods in Programming 88 (2017) 154–173. doi:10.1016/j.jlamp.2016.12.001.

[10] C. Poskitt, D. Plump, Hoare-style verification of graph programs, Fundamenta Informaticae 118 (1–2) (2012) 135–175. doi:10.3233/FI-2012-708.

[11] C. Poskitt, D. Plump, Verifying monadic second-order properties of graph programs, in: Proc. 7th International Conference on Graph Transformation (ICGT 2014), Vol. 8571 of Lecture Notes in Computer Science, Springer, 2014, pp. 33–48. doi:10.1007/978-3-319-09108-2_3.

[12] I. Hristakiev, D. Plump, Checking graph programs for confluence, in: Software Technologies: Applications and Foundations – STAF 2017 Collocated Workshops, Revised Selected Papers, Vol. 10748 of Lecture Notes in Computer Science, Springer, 2018, pp. 92–108. doi:10.1007/978-3-319-74730-9_8.

[13] H. Dörr, Efficient Graph Rewriting and its Implementation, Vol. 922 of Lecture Notes in Computer Science, Springer, 1995. doi:10.1007/BFb0031909.

[14] C. Bak, D. Plump, Rooted graph programs, in: Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012), Vol. 54 of Electronic Communications of the EASST, 2012. doi:10.14279/tuj.eceasst.54.780.

[15] C. Bak, D. Plump, Compiling graph programs to C, in: Proc. 9th International Conference on Graph Transformation (ICGT 2016), Vol. 9761 of Lecture Notes in Computer Science, Springer, 2016, pp. 102–117. doi:10.1007/978-3-319-40530-8_7.

[16] R. Sedgewick, Algorithms in C. Part 5: Graph Algorithms, 3rd Edition, Addison-Wesley, 2002.

[17] G. Campbell, J. Romö, D. Plump, The improved GP 2 compiler, Tech. rep., Department of Computer Science, University of York, UK (2020).
URL https://arxiv.org/abs/2010.03993

[18] G. Campbell, B. Courtehoute, D. Plump, Linear-time graph algorithms in GP 2, in: Proc. 8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019), Vol. 139 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, pp. 16:1–16:23. doi:10.4230/LIPIcs.CALCO.2019.16.

[19] C. Bak, GP 2: Efficient implementation of a graph programming language, Ph.D. thesis, Department of Computer Science, University of York, UK (2015).
URL https://etheses.whiterose.ac.uk/12586/

[20] A. Habel, D. Plump, Relabelling in graph transformation, in: Proc. First International Conference on Graph Transformation (ICGT 2002), Vol. 2505 of Lecture Notes in Computer Science, Springer, 2002, pp. 135–147. doi:10.1007/3-540-45832-8_12.

[21] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60–61 (2004) 17–139. doi:10.1016/j.jlap.2004.05.001.

[22] A. Aho, J. Hopcroft, J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[23] S. Skiena, The Algorithm Design Manual, 2nd Edition, Springer, 2008. doi:10.1007/978-1-84800-070-4.

[24] G. Campbell, Efficient graph rewriting, Bsc thesis, Department of Computer Science, University of York, UK (2019).
URL https://arxiv.org/abs/1906.05170