Linear-Time Graph Algorithms in GP 2

Graham Campbell
Department of Computer Science, University of York, United Kingdom
https://gjcampbell.co.uk/
gjc510@york.ac.uk

Brian Courtehoute
Department of Computer Science, University of York, United Kingdom
https://www.cs.york.ac.uk/people/brianc
bc956@york.ac.uk

Detlef Plump
Department of Computer Science, University of York, United Kingdom
https://www-users.cs.york.ac.uk/det/
detlef.plump@york.ac.uk

Abstract

GP 2 is an experimental programming language based on graph transformation rules which aims to facilitate program analysis and verification. However, implementing graph algorithms efficiently in a rule-based language is challenging because graph pattern matching is expensive. GP 2 mitigates this problem by providing *rooted* rules which, under mild conditions, can be matched in constant time. In this paper, we present linear-time GP 2 programs for three problems: tree recognition, binary DAG recognition, and topological sorting. In each case, we show the correctness of the program, prove its linear time complexity, and also give empirical evidence for the linear run time. For DAG recognition and topological sorting, the linear behaviour is achieved by implementing depth-first search strategies based on an encoding of stacks in graphs.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Graph transformation; rooted graph programs; GP 2; linear-time algorithms; depth-first search; topological sorting

Digital Object Identifier 10.4230/LIPIcs.CALCO.2019.23

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 [8] 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 [17], GReAT [1], GROOVE [10], GrGen.Net [12], Henshin [3] and PORGY [9]. This paper focuses on GP 2 [13], an experimental graph programming language which aims 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 [14]. Research on graph programs has provided, for example, a Hoare-calculus for program verification [15, 16] and a static analysis for confluence checking [11].

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 $\mathrm{size}(G)^{\mathrm{size}(L)}$ (which is polynomial since L is fixed). As a consequence, linear-time imperative graph algorithms may be slowed down to polynomial time when they are recast as rule-based graph

8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019).
Editors: Markus Roggenbach and Ana Sokolova; Article No. 23; pp. 23:1–23:25
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programs. To mitigate this problem, GP 2 allows to distinguish 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 [5], *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. A graph program with fast rules was shown in [6] to 2-colour graphs of bounded degree in linear time. The compiled program matches the speed of Sedgewick's textbook C program [18] on grid graphs of up to 100,000 nodes.

In this paper, we continue this line of research with case studies on three linear-time graph algorithms: recognition of trees, recognition of binary DAGs, and topological sorting. In each case, we present a GP 2 program with fast rules, show its correctness, and prove its linear time complexity on graphs of bounded node degree. We also give empirical evidence for the linear run time by presenting benchmark results for graphs of up to 100,000 nodes in various graph classes. For DAG recognition and topological sorting, the linear behaviour is achieved by implementing depth-first search strategies based on an encoding of stacks.

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 [13]. Up-to-date versions of the syntax and semantics of GP 2 can be found in [4]. The language is implemented by a compiler generating C code [6].

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 my be considered as "unlabelled".

The principal programming construct in GP 2 are conditional graph transformation rules labelled with expressions. For example, the rule `one_of_one` in Figure 10 has four 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 are labels. Parameters are typed but in this paper we only need the most general type `list` which represents arbitrary lists.

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 example, rule `one_of_one` in Figure 10 contains red and blue nodes and a blue edge. 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 integer and string constants). 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 $\mathtt{indeg(1)} = \mathtt{1}$ of rule `one_of_one` in Figure 10 requires that node $g(\mathtt{1})$ has exactly 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. 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. The operational semantics of GP 2 defines a semantic function which maps each host graph to the set of all possible outcomes. See, for example, [14].

2.2 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 [5, 6]. 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 2, 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 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 2 are fast rules.

▶ **Theorem 1** (Complexity of matching fast rules [5]). *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.*

When analysing the time complexity of rules and 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 [2, 19]. In our setting, the input size is the size of a host graph. The implementation of GP 2 does match fast rooted rules in constant time [6].

3 Recognising Trees

A tree is a connected graph without undirected cycles such that every node has at most one incoming edge. It is easy to see that it is possible to generate the collection of all non-empty trees by inductively adding new leaf nodes onto the discrete graph of size one. Thus, given an input graph, if we prune leaf nodes as long as possible and end up with the discrete graph of size one, then the start graph must have been a tree. Figure 1 is implementation of this idea in GP 2.

```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

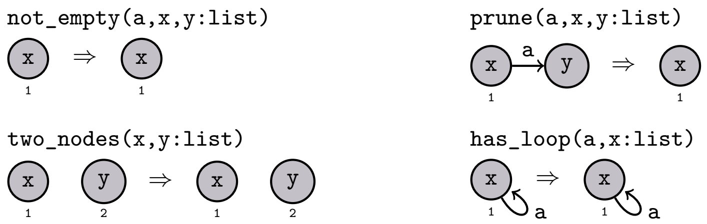


Figure 1 The GP 2 program `is-tree-slow`

▶ **Definition 2** (Tree recognition specification). *The tree recognition specification is as follows.*
- Input*: An arbitrary labelled graph with every node coloured grey, no root nodes, and no other marks.*
- Output*: Fail if and only if the input is not a non-empty tree.*

▶ **Theorem 3** (Correctness of `is-tree-slow`). *The program `is-slow-tree` fulfills the tree recognition specification.*

Proof. Similar to the proof of Theorem 6. ◀

▶ **Proposition 4** (Termination of `prune!`). *`prune!` terminates after at most $|V_G|$ steps.*

Proof. If $G \Rightarrow H$, then $|V_G| > |V_H|$. Suppose there were an infinite sequence of derivations $G_0 \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \cdots$, then there would be an infinite descending chain of natural numbers $|V_{G_0}| > |V_{G_1}| > |V_{G_2}| > \cdots$, which contradicts the well-ordering of $\mathbb{N}$. The last part is immediate since there are only V_G natural numbers less than V_G. ◀

▶ **Theorem 5** (Complexity of `is-tree-slow`). *Given an input graph of bounded degree, `is-tree-slow` will terminate in quadratic time with respect to the number of nodes in the input graph.*

Proof. Clearly `not_empty` and `Check` run in linear time. Unfortunately `prune` is not a fast rule schema, and so it takes linear time to find a match. Finding a match for `prune` takes linear time and so by Proposition 4, `prune!` terminates in quadratic time. ◀

Unfortunately, our program does not run in linear time due to our rules not being such that we have constant time matching. We need to modify the program so that we can always perform a match in constant time. Figure 2 is a refined implementation, using root nodes. We will see that this program is not only correct, but always terminates in linear time.

```
Main = init; Reduce!; if Check then fail
Reduce = {prune, push}
Check = {two_nodes, has_loop}
```

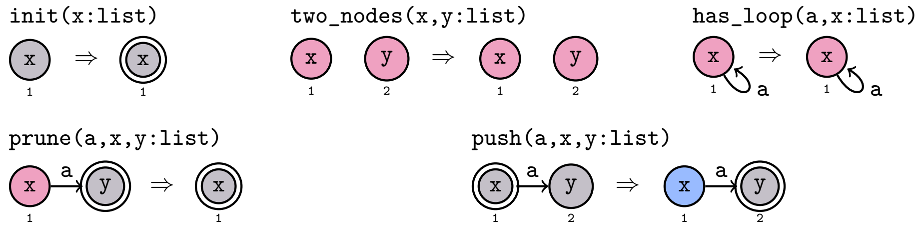


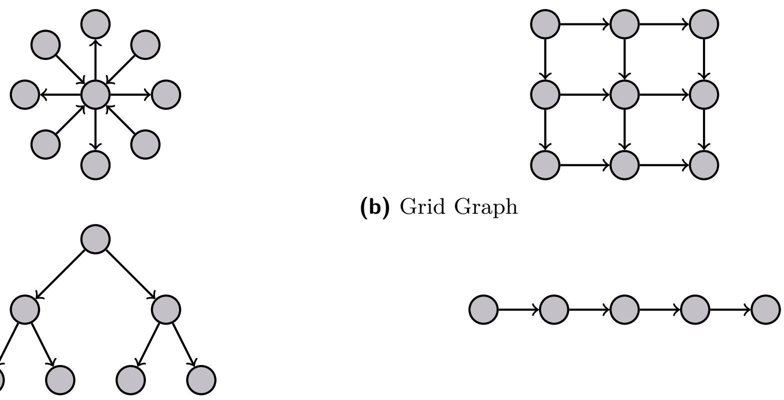
Figure 2 The GP 2 program `is-tree`

▶ **Theorem 6** (Correctness of `is-tree`). *The program* `is-tree` *fulfills the tree recognition specification.*

Proof. The `init` rule will fail if the input graph is empty, otherwise, it will make exactly one node rooted. The `Reduce!` step derives the singleton discrete graph if and only if the input was a tree (Lemmata 16 and 20). Finally, by Lemma 17, `Reduce!` cannot derive the empty graph, so it is sufficient for `Check` to test if there is more than one node, or a loop edge. ◀

▶ **Theorem 7** (Complexity of `is-tree`). *Given an input graph of bounded degree,* `is-tree` *will terminate in linear time with respect to the number of nodes in the input graph.*

Proof. Clearly `init` and `Check` run in linear time. Since `push` and `prune` are fast rules, they take only constant time (Theorem 1), and then by Lemma 15, `Reduce` can only be applied a linear number of times. Thus, `Reduce!` terminates in linear time too. ◀

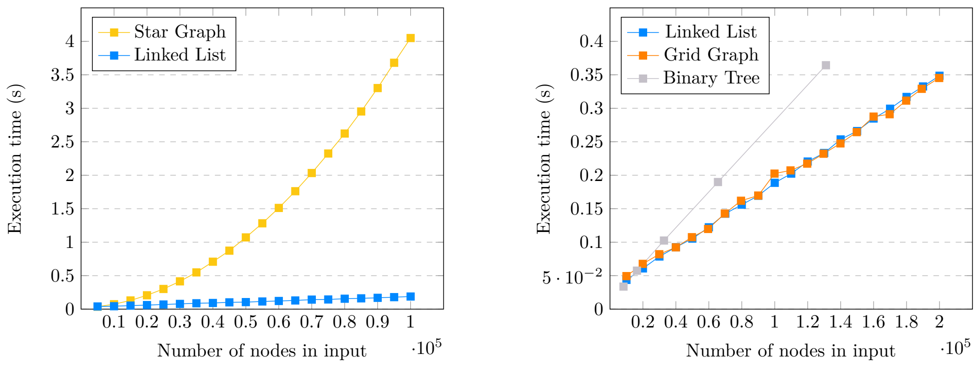


(a) Star Graph **(b)** Grid Graph **(c)** Binary Tree **(d)** Linked List

Figure 3 Types of Graph

We have performed empirical benchmarking to verify the complexity of the program, testing it with Linked Lists, Binary Trees, Grid Graphs, and Star Graphs (Figure 3). Star

Graphs are not of bounded degree, so we saw quadratic time complexity as expected. The other graphs are of bounded degree, thus we observed linear time complexity (Figure 4).



(a) Star Graphs and Linked Lists **(b)** Bounded Degree Input Graphs

Figure 4 Measured performance of `is-tree`

4 Implementing Depth First Search

The depth first search (DFS) seen in Figure 5 is based on the graph traversal done during the GP 2 2-colouring program [6]. Due to the nature of GP 2, it differs from commonly used implementation.

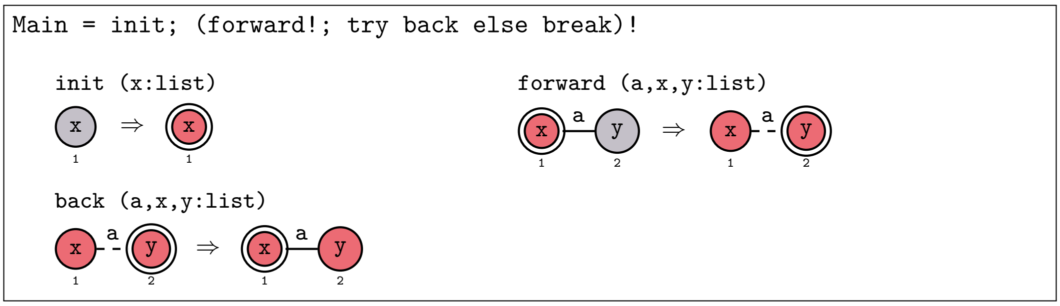


Figure 5 The GP 2 program `general-dfs`

Specifically, standard implementations of DFS [7, 19] loop over the nodes of the input graph, and if the current node has not been visited yet, a recursive function is called on it. Said function marks the node it is called on as visited, and then calls itself on an unvisited neighbour on an outgoing edge. If a program does not loop over all the nodes, and just applies the recursive function to some node v, only the nodes reachable from v are visited, since only outgoing edges are considered at each step. In GP 2 however, there is no obvious linear-time way to loop over all nodes. So instead, the GP 2 program does not require the edges to be outgoing, and applies a command sequence analogous to the recursive function on an arbitrary node.

This approach implements DFS in an undirected graph, but not DFS in a directed graph. So `general-dfs` is guaranteed to visit all nodes of the input graph in linear time, but not necessarily in the order one might expect from a DFS in a directed graph. A DFS implementation that visits the nodes of the input graph in the expected order can be found at the core of the GP 2 program in Section 6.

The program starts by rooting and marking an arbitrary node red. `forward!` moves the root along a path through the unvisited grey nodes until it reaches a node with no unvisited neighbours. This path is marked with dashed edges. Then `back` is applied, bringing the root one step back in the marked path. The program loops until `back` cannot be applied, which is when there is no marked path left, i.e. the root is back at the initially rooted node.

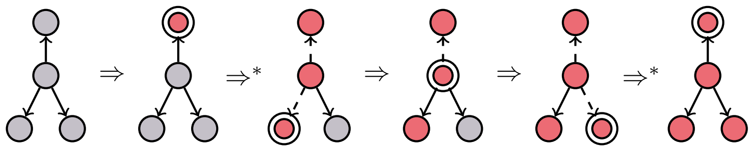


Figure 6 Example execution of `general-dfs`

▶ **Theorem 8** (Correctness and Complexity of `general-dfs`). *Given a connected input graph G of bounded degree with grey unrooted nodes and unmarked edges, `general-dfs` marks all nodes red in linear time.*

Proof. Correctness follows from Lemma 22 and complexity from Corollary 24. They are partially adapted from Bak's [4] proofs of correctness and complexity for a different DFS program. ◀

5 Recognising Binary DAGs

A *directed acyclic graph* (DAG) is a graph containing no directed cycles. A DAG is *binary* if each of its nodes has an outdegree of at most two.

```
Main = try SearchIndeg0Nodes then (if nonempty_stack then skip else fail;
ReduceIndeg0Nodes); if anything then fail
```

Figure 7 The GP 2 Program `is-bin-dag`

The idea behind recognising connected binary DAGs is as follows. First, all indegree-0 nodes of the input graph are identified. Then, if any indegree-0 nodes have been found, one of them is deleted, and all of its children that become a new indegree-0 node get designated as such. This is repeated until no indegree-0 nodes are left. Every time an indegree-0 node is checked, the number of its children are checked as well. If there are any leftover nodes (i.e. nodes that never had indegree-0 in the execution), then there were no directed cycles, and the input graph is a DAG.

▶ **Theorem 9** (Correctness of `is-bin-dag`). *The program `is-bin-dag` fulfills the following specification.*

- Input*: A connected graph G with grey unrooted nodes and unmarked edges.*
- Output*: The empty graph if G minus the blue edges was a binary DAG, and failure otherwise.*

Proof. If G is the empty graph, `SearchIndeg0Nodes` fails by Proposition 10, anything does not match, and the output is the empty graph. So assume G is nonempty.

If G has no indegree-0 nodes, `nonempty_stack` will not match, and `fail` will be invoked. So assume G has indegree-0 nodes.

Then Propositions 10 and 11 can be applied to deduce the following. `SearchIndeg0Nodes` succeeds, `nonempty_stack` matches, then `ReduceIndeg0Nodes` gets applied. If G is a binary DAG, the host graph becomes the empty graph, `anything` will not match, and the output is the empty graph. If G is not a binary DAG, there's failure, or a nonempty graph which results in failure since `anything` is matched. ◀

5.1 Correctness of Procedures

The proof of Theorem 9 depends upon the correctness of the procedures `SearchIndeg0Nodes` and `ReduceIndeg0Nodes`. We will now give their definitions and prove their correctness.

▶ **Proposition 10** (Correctness of `SearchIndeg0Nodes`). *The procedure `SearchIndeg0Nodes` fulfills the following specification.*

- Input*: A connected graph G with grey unrooted nodes and unmarked edges.*
- Output*: If G is the empty graph, then failure. Otherwise G with red non-indegree-0 nodes containing at most one root, and blue indegree-0 nodes that are connected with blue edges forming a path graph (i.e. a linked list). The blue node with no incoming blue edges is a root.*

Proof. If G is empty, `init` cannot match, causing failure. Otherwise, the output conditions are satisfied by Lemmata 26 and 27. ◀

The absence of a red root in the output is an edge case caused by `init` being applied to an indegree-0 node. Because then, either `i0_stack` or `i0_push` will be the last rule that is applied, and the red root becomes a blue root.

The blue nodes linked with blue edges are a GP 2 implementation of stacks. The top of the stack is the only blue root, making it accessible in constant time.

```
SearchIndeg0Nodes = init; (i0_forward!; try i0_push else (try i0_stack);
try i0_back_red else (try i0_back_blue else break))!
```

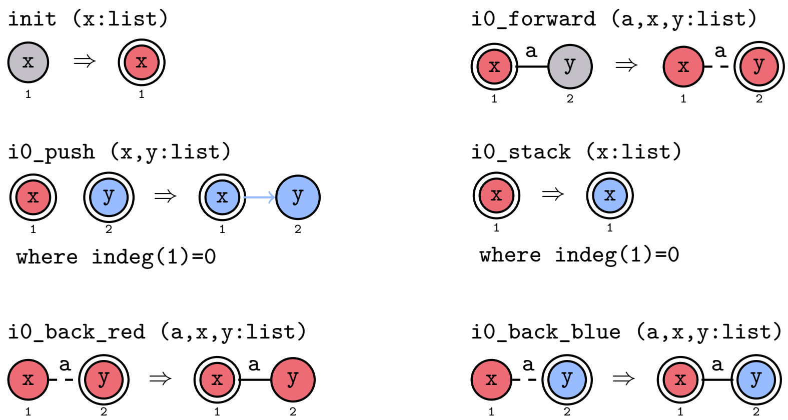


Figure 8 The `SearchIndeg0Nodes` procedure

`SearchIndeg0Nodes`, as seen in Figure 8, is based on the DFS implementation from Section 4, with a few key differences. Using DFS ensures that each node is visited.

Between the forward and back steps lies the command sequence `try i0_push else (try i0_stack)`. Its purpose is to push the node currently visited by the DFS if it has indegree-0. If the stack is nonexistent, there are no blue nodes, and `i0_push` fails. So the program tries to apply `i0_stack`, turning the node into the initial stack element (if its indegree is 0). After the stack has been created, `i0_push` will always be applicable for indegree-0 nodes.

Since the current node may be marked blue by the stack operations after the previous command sequence has been executed, the back step needs to account for that. Hence the program first tries to apply the back rule from the previous DFS program, and if that fails, it tries to apply an alternate version considering a blue current node. In the latter case, the blue node is rooted since we want to keep accessing the top of the stack in constant time.

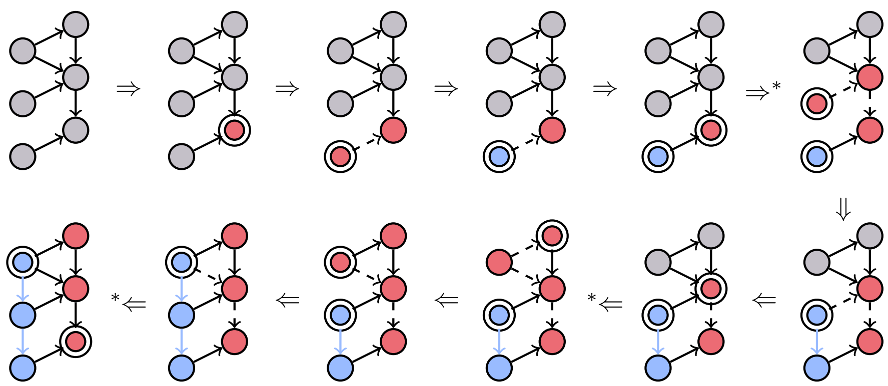


Figure 9 Example execution of `SearchIndeg0Nodes`

▶ **Proposition 11** (Correctness of `ReduceIndeg0Nodes`). *The procedure `ReduceIndeg0Nodes` fulfills the following specification.*

- Input*: A connected graph G with red non-indegree-0 nodes containing at most one root, and blue indegree-0 nodes that are connected with blue edges forming a path graph. The blue node with no incoming blue edges is a root.*
- Output*: The empty graph if G minus the blue edges was a binary DAG, and a nonempty one or failure otherwise.*

Proof. This result follows directly from Lemmata 31 and 32. ◀

The procedure starts by trying to apply `unroot` to get rid of any red roots left over by `SearchIndeg0Nodes`. Then it enters the loop `Reduce!`. The blue root in each iteration shall be called the "top root". First, the program checks whether the top root has more than two children, i.e. whether its outdegree is greater than three, since the blue stack edge needs to be taken into account. If there are too many, the `fail` statement is invoked.

`nontrivial_stack` checks whether the stack has more than one element. If it does not, `add_bottom` artificially adds a node to the bottom of the stack, in order for the following rules to still match.

Next is a non-deterministic choice of rules that cover every case of the number of children the top root has, and how many of those are indegree-0 nodes. In each case, they pop the top root, and push the children that would have indegree 0 after the deletion. `pop!` serves to pop childless indegree-0 nodes for as long as there are any.

```
ReduceIndeg0Nodes = try unroot; Reduce!; pop!; try final_pop
Reduce = if too_many_children then fail; if nontrivial_stack then skip
else add_bottom; {two_of_two, one_of_two, none_of_two, one_of_double,
none_of_double, one_of_one, none_of_one }; pop!
```

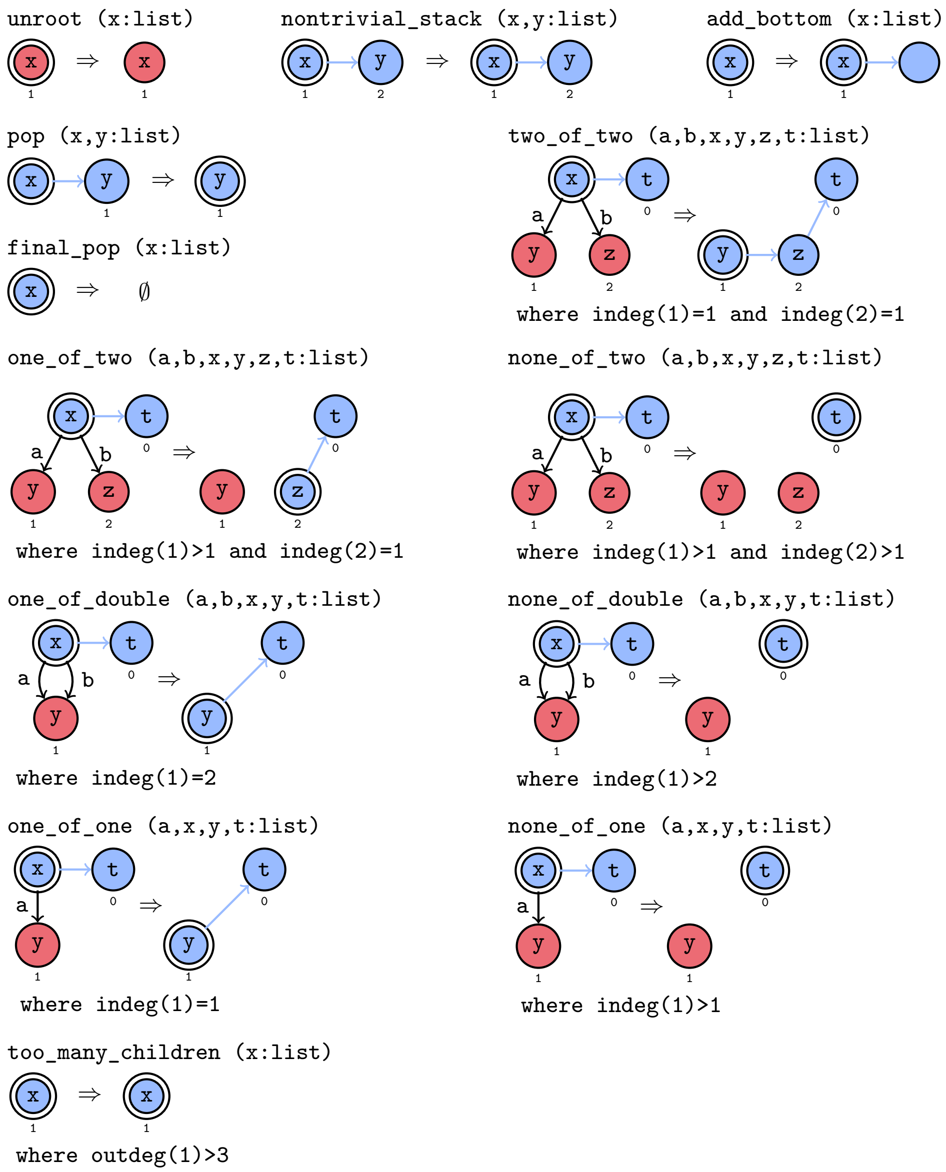


Figure 10 The ReduceIndeg0Nodes procedure

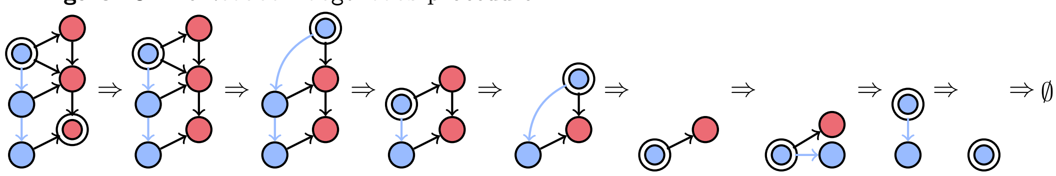


Figure 11 Example execution of ReduceIndeg0Nodes

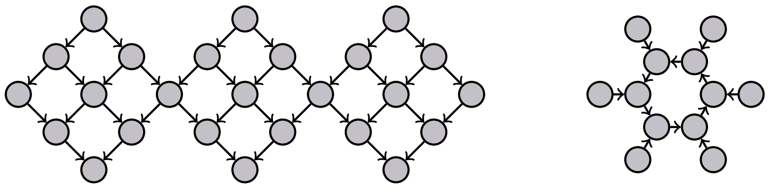
5.2 Performance

We will show that our binary DAG recognition program always terminates in linear time, given an input graph of bounded degree. We have also included empirical evidence for this.

▶ **Theorem 12** (Complexity of `is-bin-dag`). *Given a connected input graph of bounded degree, the program `is-bin-dag` terminates in linear time.*

Proof. The `Main` procedure of `is-bin-dag` contains no loops. `SearchIndeg0Nodes` and `ReduceIndeg0Nodes` terminate in linear time by Lemmata 25 and 33. `nonempty_stack` matches in constant time by Theorem 1 since it is a fast rule schema. `anything` also matches in constant time since any node is a valid match. ◀

In order to support the linear time complexity of `is-bin-dag`, performance will be measured on two graph classes, one consisting of binary DAGs, and the other of non-DAGs.



(a) Grid Chain GC(3)

(b) Sun Graph SG(6)

Figure 12 Input Graph Classes

Consider the following class of binary DAGs. For $n \geq 1$, the *grid chain* GC(n) consists of n grids of size $n \times n$, joint by the nodes of indegree and outdegree 1 in order to form a chain. This class was chosen for having an unbounded number of indegree-0 nodes, meaning that the implemented stack is relatively large.

Now consider the following class of non-DAGs. For $n \geq 3$, the *sun graph* SG(n) consists of a directed cycle of n nodes, each of which has an an additional neighbour connected by an incoming edge. The reason for using this class is, in addition to half the nodes having indegree 0, the other half are part of the cycle, and therefore never get deleted by `ReduceIndeg0Nodes`. This causes an unbounded amount of nodes to be left over.

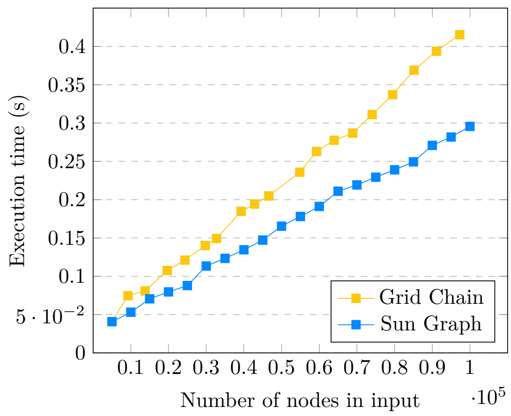


Figure 13 Measured performance of `is-bin-dag`

6 Topological Sorting

Given a DAG G, a *topological sorting* is a total order (an antisymmetric, transitive, and connex binary relation) $\leq$ on V_G, 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. Furthermore, every DAG has a topological sorting.

There are two commonly used linear-time algorithms for finding a topological sorting [19, 18]. One seeks out indegree-0 nodes, adds them to the total order, deletes them, and repeats this process until all nodes have been added to the order. The other, which is used as the basis for the algorithm in this paper, conducts a DFS. Upon completion of a node in the DFS, that node is added as the new minimum element of the linear order. However, unlike the program `general-dfs` from Section 4, the direction of the edges needs to be respected in order to get a topological sorting in the end. Simply turning the bidirectional edges of `general-dfs` into directed edges is not enough since that would only visit the nodes reachable from the initially rooted node, which is not necessarily the entire input graph. Traditional algorithms solve this problem by skipping to the next unvisited node in the data structure representing the graph, and continuing the DFS from there. Similarly, the program `top-sort` uses a DFS implementation with directed edges (`SortNodes`), and once it runs out of unvisited nodes, it uses a DFS similar to `general-dfs` (`SearchUnsortedNodes`) to find a node that has not been visited yet, and to continue the `SortNodes` DFS.

6.1 The Program

We give the GP 2 implementation of topological sorting in Figure 14 and show its correctness. We have added the restriction that the input graph must be connected since in the current version of GP 2, there is no known way to implement a DFS that is linear-time for graphs with an unbounded number of connected components. We have also included an example execution of the program in Figure 15.

▶ **Theorem 13** (Correctness of `top-sort`). *The program `top-sort` fulfills the following specification.*

- Input*: A connected DAG G with no roots whose nodes are all marked grey, and whose edges are unmarked.*
- Output*: G with additional blue edges that define a topological ordering on V_G. The nodes of G are marked blue and each have a red loop. One of these nodes is rooted. Furthermore, there is an additional unlabelled green root node with an outgoing green edge pointing to a node with no incoming blue edges.*

Proof. This theorem follows from Lemma 37. ◀

The additional constructs in the output graph, apart from the blue edges, are needed for the execution of the program. One could define a linear-time cleanup procedure to remove these constructs. The green root and its outgoing edge can be deleted in constant time, since access to roots is constant. Similarly, the blue rooted node can be unrooted in constant time. A DFS similar to the one in Section 4 can be used to remove the red loops or unmark all the nodes in linear time.

The subgraph induced by the blue edges is a path graph containing all the nodes from the input graph. So the binary relation $\leq$ on the set of nodes defined by $u \leq v$ if there is a path of blue edges from u to v is a total order, which is a necessary property for a topological sorting. Similarly to the `SearchIndeg0Nodes` procedure of Subsection 5.1, the blue nodes and edges implement a stack. However, this time the top of the stack is denoted with a green root pointing towards it with a green edge in order not to interfere with a DFS in `SortNodes`.

```
Main = init; SearchUnsortedNodes
SearchUnsortedNodes = ((try unsorted then SortNodes; search_forward)!;
try search_back else break)!
SortNodes = (sort_forward!; try sort_back_push else (try sort_back_stack
else (try red_push else red_stack; break)))!
```

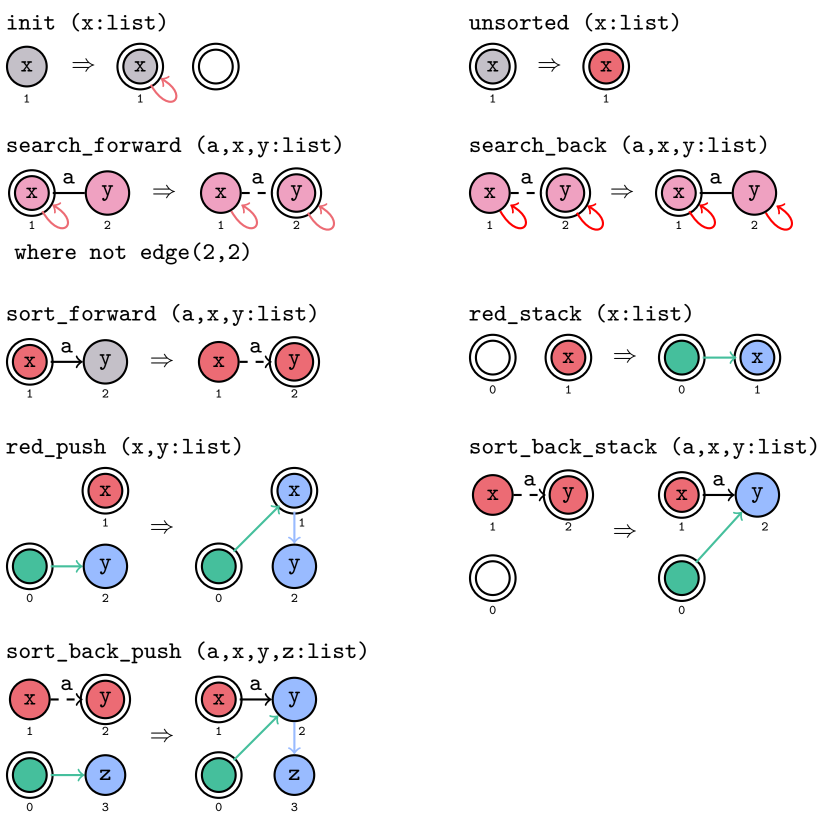


Figure 14 The GP 2 program `top-sort`

The program starts by rooting an input node and endowing it with a red loop, as well as creating an unmarked, unlabelled root that is disconnected from the rest of the graph. This root will point to the top of the stack, and shall hence be called the “pointer”.

`SearchUnsortedNodes` is a DFS implementation similar to `general-dfs` from Section 4 that seeks out a node that has not been visited by `SortNodes` yet. Instead of using a red mark to designate a node as visited, it uses a red loop. Since the input is assumed to be a DAG, it has no loops. This leaves the use of marks to the DFS in `SortNodes`. So in order for the forward step to only match unvisited neighbours of the root, a predicate to forbid loops is needed. The “any” mark ensures that colour does not matter. Right before each application of the forward step, `unsorted` tests whether the current root has been visited by `SortNodes` yet, i.e. whether it is grey. At the same time, said root is initialised for `SortNodes` by being marked red.

Next, `SortNodes` is applied. It performs a DFS with directed edges. Similarly to `SearchIndeg0Nodes` from Section 5, it pushes the current root onto the stack during its back step. `sort_back_push` is applied when the stack has at least one element, otherwise `sort_back_stack` creates the stack. The pointer being green represents the stack being nonempty. The break statement is preceded by `try red_push else red_stack`, since when the back step can no longer be applied, the current root is still pushed onto the stack. Again, two rules are needed to cover the cases of the stack being empty or not. Because of the repeated application of the back step, the root ends up where it was at the beginning of `SortNodes`, meaning that the DFS of `SearchUnsortedNodes` can resume undisturbed.

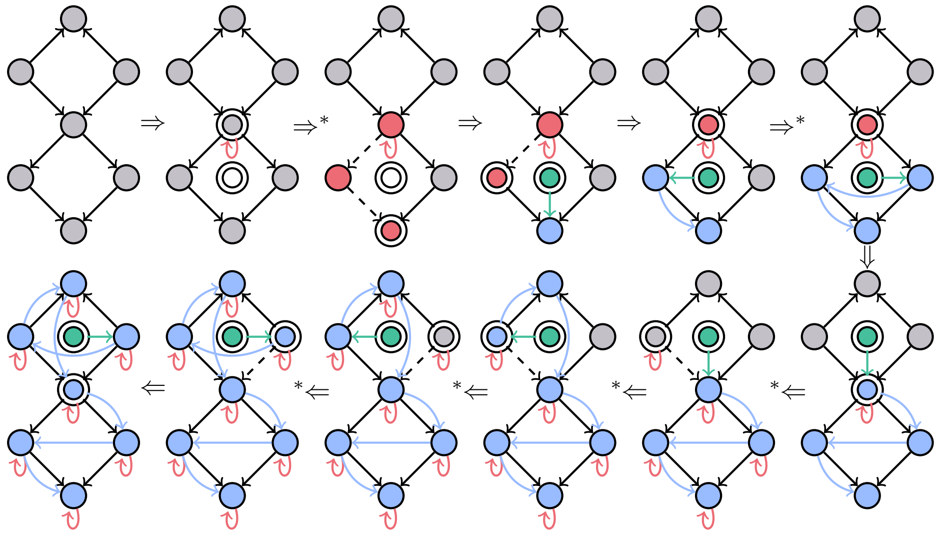


Figure 15 Example execution of `top-sort`

6.2 Performance

Finally, we show that, given a valid input graph of bounded degree, our topological sorting program will always terminate in linear time.

▶ **Theorem 14** (Complexity of `top-sort`). *Given a connected DAG of bounded degree with only grey unrooted nodes whose edges are unmarked as an input, the program* `top-sort` *terminates in linear time.*

Proof. This theorem follows from Lemma 38. ◀

In order to support the linear time complexity of `top-sort`, we make use of the grid chains from Subsection 5.2. They are DAGs, the type of graph `top-sort` is meant to be used on. Furthermore, they have an unbounded number of indegree-0 nodes. Since indegree-0 nodes are unreachable from any other node, and `SortNodes` can only visit nodes reachable from the red root it is called on, `SortNodes` will have to be applied at least once for each indegree-0 node, i.e. an unbounded number of times. Thus these input graphs can adequately illustrate the linearity of `top-sort`. Figure 16 is a plot of the program timings, demonstrating linear time complexity.

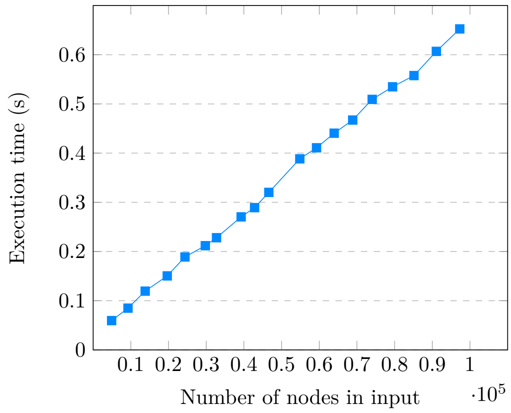


Figure 16 Measured performance of `top-sort` on grid chains

7 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 presented rooted GP 2 programs for three graph algorithms: tree recognition, connected binary DAG recognition, and topological sorting. The programs were proved to be correct and to run in linear time on graphs of bounded node degree. The proofs demonstrate that graph transformation rules provide a convenient and intuitive abstraction level for formal reasoning on graph programs. We also gave empirical evidence for the linear run time of the programs, by presenting benchmark results for graphs of up to 100,000 nodes in various graph classes. For DAG recognition and topological sorting, the linear behaviour was 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 for this is the encoding of stacks as linked lists in the programs for DAG recognition and topological sorting.

The three programs in this paper and also the 2-colouring program of [6] 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 to overcome this restriction. Clearly, such a mechanism will require to modify GP 2 and its implementation.

References

1. Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006. `doi:10.1007/s10270-006-0027-7`.
2. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
3. Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Model*

Driven Engineering Languages and Systems (MODELS 2010), volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010. doi:10.1007/978-3-642-16145-2_9.

4 Christopher Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, Department of Computer Science, University of York, 2015. URL: http://etheses.whiterose.ac.uk/12586/.

5 Christopher Bak and Detlef Plump. Rooted graph programs. In *Proc. International Workshop on Graph Based Tools (GraBaTs 2012)*, volume 54 of *Electronic Communications of the EASST*, 2012. doi:10.14279/tuj.eceasst.54.780.

6 Christopher Bak and Detlef Plump. Compiling graph programs to C. In *Proc. International Conference on Graph Transformation (ICGT 2016)*, volume 9761 of *LNCS*, pages 102–117. Springer, 2016. doi:10.1007/978-3-319-40530-8_7.

7 Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

8 Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation*. Monographs in Theoretical Computer Science. Springer, 2015. doi:10.1007/978-3-662-47980-3.

9 Maribel Fernández, Hélène Kirchner, Ian Mackie, and Bruno Pinaud. Visual modelling of complex systems: Towards an abstract machine for PORGY. In *Proc. Computability in Europe (CiE 2014)*, volume 8493 of *Lecture Notes in Computer Science*, pages 183–193. Springer, 2014. doi:10.1007/978-3-319-08019-2_19.

10 Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012. doi:10.1007/s10009-011-0186-x.

11 Ivaylo Hristakiev and Detlef Plump. Checking graph programs for confluence. In *Software Technologies: Applications and Foundations – STAF 2017 Collocated Workshops, Revised Selected Papers*, volume 10748 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2018. doi:10.1007/978-3-319-74730-9_8.

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

13 Detlef Plump. The design of GP 2. In *Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2012. doi:10.4204/EPTCS.82.1.

14 Detlef Plump. From imperative to rule-based graph programs. *Journal of Logical and Algebraic Methods in Programming*, 88:154–173, 2017. doi:10.1016/j.jlamp.2016.12.001.

15 Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012. doi:10.3233/FI-2012-708.

16 Christopher M. Poskitt and Detlef Plump. Verifying monadic second-order properties of graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2014)*, volume 8571 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014. doi:10.1007/978-3-319-09108-2_3.

17 Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 — new features for specifying and analyzing algebraic graph transformations. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, volume 7233 of *Lecture Notes in Computer Science*, pages 81–88. Springer, 2012. doi:10.1007/978-3-642-34176-2_8.

18 Robert Sedgewick. *Algorithms in C. Part 5: Graph Algorithms*. Addison-Wesley, third edition, 2002.

19 Steven Skiena. *The Algorithm Design Manual*. Springer, second edition, 2008. doi:10.1007/978-1-84800-070-4.

A Appendix: Proofs

This appendix consists of lemmata and proofs omitted from the main sections.

A.1 Tree Recognition Lemmata

By *rooted input graph*, we mean an arbitrary labelled GP 2 input graph with every node coloured grey, exactly one *root* node, and no additional marks. That is, a valid input graph after `init` has been applied. By *rooted input tree*, we mean an rooted input graph that is a non-empty tree. In this appendix, we give the proofs of the lemmata needed to support Theorems 6 and 7 from Section 3.

▶ **Lemma 15.** *`Reduce!` terminates after at most $2|V_G|$ steps.*

Proof. Let $\#G$ be the number of nodes, and $\Box G$ be the number of grey nodes. If $G \Rightarrow_{prune} H$, then $\#G > \#H$ and $\Box G > \Box H$. If $G \Rightarrow_{push} H$ then $\#G = \#H$ and $\Box G > \Box H$. Suppose there were an infinite sequence of derivations $G_0 \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \cdots$, then there would be an infinite descending chain of natural numbers $\#G_0 + \Box G_0 > \#G_1 + \Box G_1 > \#G_2 + \Box G_2 > \cdots$, which contradicts the well-ordering of $\mathbb{N}$. To see the last part, notice that $\Box G \leq \#G$ for all graphs G, so the result is immediate since there are only $2\#G$ natural numbers less than $2\#G$. ◀

▶ **Lemma 16.** *If G is a tree and $G \Rightarrow^*_{Reduce} H$, then H is a tree. If G is not a tree and $G \Rightarrow_{Reduce!} H$, then H is not a tree.*

Proof. Clearly, the application of `push` preserves structure. Suppose G is a tree. `prune` is applicable if and only the second node is matched against a leaf node, due to the dangling condition. Upon application, the leaf node and its incoming edge is removed. Clearly the result graph is still a tree. If G is not a tree and `prune` is applicable, then we can see the properties of not being a tree are preserved. That is, if G is not connected, H is certainly not connected. If G had parallel edges, due to the dangling condition, they must exist in $G \setminus g(L)$, so H has parallel edges. Similarly, cycles are preserved. Finally, if G had a node with incoming degree greater than one, then H must too, since the node in G that is deleted in H had incoming degree one, and the degree of all other nodes is preserved. So, we have shown `Reduce` is structure preserving, and then by induction, so is `Reduce!`. ◀

▶ **Lemma 17.** *If G is a rooted input graph and $G \Rightarrow^*_{Reduce} H$, then H has exactly one root node. Moreover, there is no derivation sequence that derives the empty graph.*

Proof. In each application of `prune` or `push`, the number of root nodes is invariant since the LHS of each rule must be matched against a root node in the host graph, so the other non-roots can only be matched against non-roots, and so the result holds by induction. To see that the empty graph cannot be derived, notice that each derivation reduces $\#G$ by at most one, and no rules are applicable when $\#G = 1$. ◀

▶ **Lemma 18.** *If G is a rooted input graph and $G \Rightarrow^*_{Reduce} H$. Then, every blue node in H either has a blue child or a root-node child.*

Proof. Clearly G satisfies this, as there are no blue nodes. We now proceed by induction. Suppose $G \Rightarrow^*_{Reduce} H \Rightarrow_{Reduce} H'$ where H satisfies the condition. If `prune` is applicable, we introduce no new blue nodes. Additionally, any blue parents of the node 1 are preserved. So H' satisfies the condition. Finally, if `push` is applied, then the new blue node has a root-node child, and the blue nodes in $H' \setminus h(R)$ have the same children, so H' satisfies the condition. ◀

▶ **Corollary 19.** *Let G be a rooted input tree and $G \Rightarrow^*_{Reduce} H$. Then the root-node in H has no blue children.*

Proof. By Lemma 17, H has exactly one root node, and by Lemma 18, all chains of blue nodes terminate with a root-node. If said root-node were to have a blue child, then we would have a cycle, which contradicts that H is a tree (Lemma 16). ◀

▶ **Lemma 20.** *Let G be a rooted input tree and $G \Rightarrow^*_{Reduce} H$. Then, either $|V_H| = 1$ or H is not in normal form.*

Proof. By Lemma 17, $|V_H| \geq 1$. If $|V_G| = 1$, then G is in normal form. Otherwise, either the root node has no children, or it has at least one grey child. In the first case, `prune` must be applicable, and in the second, `push`. Suppose $G \Rightarrow^*_{Reduce} H$. If $|V_H| = 1$, then H is in normal form by the proof to Lemma 17. Otherwise, by Lemma 16 H is a tree and $|V_H| > 1$. Now, the root-node in H (Lemma 17) must have a non-empty neighbourhood. If it has no children, then `prune` must be applicable. Otherwise, `push` must be applicable, since by Corollary 19, there must be a grey node child. So H is not in normal form. ◀

A.2 DFS Lemmata

In this appendix, we give the proofs of the lemmata needed to support Theorem 8 from Section 4.

▶ **Lemma 21** (Termination of `general-dfs`). *Given an input graph G with grey nodes an unmarked edges, `general-dfs` terminates.*

Proof. Consider the following (total) lexicographical ordering $>$ on graphs, defined as $H_1 > H_2$ if H_1 has more grey nodes than H_2, or H_1 and H_2 have the same number of grey nodes and H_1 has more dashed edges than H_2. If `forward` is applied on a graph H_1, yielding H_2, then $H_1 > H_2$ since the rule strictly decreases the number of grey nodes. In particular, `forward!` terminates since eventually, there is no possible match for the left hand side of `forward`. Similarly, applying `back` on a graph H_1 to obtain H_2 strictly decreases the number of dashed edges, and keeps the number of grey nodes constant. So $H_1 > H_2$. Since G has only grey nodes and unmarked edges, there are at most $|V_G| \cdot |E_G|$ (i.e. finitely many) graphs H such that $G > H$. So at some point, neither `forward`, nor `back` are applicable. Since `back` is not applicable, the `break` statement is invoked, causing the program to exit the loop. ◀

▶ **Lemma 22** (Correctness of `general-dfs`). *Given a connected input graph G with grey unrooted nodes an unmarked edges, the subgraph H induced by the edges that have been dashed during the execution of `general-dfs` is a spanning tree, all of whose nodes are marked red.*

Proof. H is a tree if and only if it is connected and has $|V_H| - 1$ edges. This property shall be used to show that H is a tree. The program starts by applying `init`, which roots an input node v and marks it red. If G only has one node, the program terminates and H is the empty graph, which is a tree. Otherwise, `forward` gets applied to a node w (since G is connected) and dashes the edge between v and w, making them part of H. So H has at least two nodes and one edge. The only rule dashing edges is `forward`, which, apart from `init`, is the only rule marking grey nodes red. So the nodes of H are exactly the nodes that are marked red during execution. `forward` matches a red node, a grey node, and the edge between them, ensuring they are all part of H. Since only the red node was part of H already, exactly one node and one edge are added to H. Hence, if `forward` is applied $n \geq 0$ times, we have $|V_H| = 2 + n$ and $|E_H| = 1 + n = |V_H| - 1$. Since `forward` adds an edge and a node to what is known to be part of H already, starting from a connected graph, H is connected.

It remains to show that H is a spanning tree. If $|V_G| \leq 2$, applying `init` followed by `forward` ensures H being a spanning tree. Otherwise, since H is a subgraph of G as well as a tree, it can be extended to a spanning tree T. Assume for the sake of a contradiction that $G \setminus H$ in nonempty. Let w be a (grey) node of $G \setminus H$ adjacent to a (red) node u of H. It exists since G is connected. Each leaf v of H marks the termination of `forward!` (which terminates by Lemma 21), because if `forward` could have been applied again, it would have, and v would have two adjacent dashed edges. If u is a leaf, `forward` would be applicable to u and w, marking w red and causing a contradiction. If u is not a leaf, after all its descendants have been marked red, the root would be moved back to u using applications of the `back` rule. Subsequently, u and w would have been matched by `forward`, causing a contradiction again. ◀

▶ **Corollary 23.** *Given a connected input graph G with grey unrooted nodes an unmarked edges, where $|V_G| \geq 2$, `general-dfs` applies `forward` and `back` exactly $|V_G| - 1$ times each. For $|V_G| < 2$, they are not applied at all.*

Proof. If $|V_G| < 2$, `forward` and `back` do not have enough vertices to match.

Otherwise, since H as in Lemma 22 is a spanning tree, it has $|V_G| - 1$ edges. As seen in the proof of that Lemma, H is constructed a red root by applying `forward` a number of times, which adds an edge and a node each time. So `forward` must have been applied at least $|V_G| - 1$ times. It must also have been applied at most $|V_G| - 1$ times, since that's the number of grey nodes after application of `init`, since it marks every added grey node red, and since no other rule introduces grey nodes.

`back` matches at most $|V_G| - 1$ times since it can only match a dashed edge. It matches at least $|V_G| - 1$ times since its left hand side is matchable throughout the spanning tree. ◀

▶ **Corollary 24** (Complexity of `general-dfs`)**.** *Given a connected input graph G of bounded degree with grey unrooted nodes an unmarked edges, `general-dfs` terminates in linear time.*

Proof. `init` is matched exactly once at the beginning of the program. Since the input has grey nodes only and hence valid matches, the rule matches in constant time. The other rules are fast rules, and hence match in constant time on graphs of bounded degree with a bounded number of roots by Theorem 1. There is always exactly one root since the input has none, `init` introduces one, and the other rules conserve the number of roots. By Corollary 23, `forward` and `back` are applied a number of times linear in the size of the input graph. Since Lemma 21 guarantees the program to terminate, `general-dfs` terminates in linear time. ◀

A.3 Binary DAG Recognition Lemmata

In this appendix, we give the proofs of the lemmata needed to support Propositions 10 and 11 and Theorem 12 from Section 5.

▶ **Lemma 25** (Complexity and Partial Correctness of `SearchIndeg0Nodes`)**.** *Given a connected input graph G with grey unrooted nodes an unmarked edges, `SearchIndeg0Nodes` terminates, and the subgraph H induced by the edges that have been dashed during the execution is a spanning tree. Furthermore, if G has bounded degree, the procedure terminates in linear time.*

Proof. Similar to the proofs in Appendix A.2. ◀

▶ **Lemma 26.** *Given a nonempty connected input graph G with grey unrooted nodes an unmarked edges, at any point of the execution of `SearchIndeg0Nodes`, there is at most one red root.*

Proof. `init` introduces a red root, and is only applied once and in the beginning. The other rules that do not preserve red roots are `i0_push`, `i0_stack` and `i0_back_blue`. If either `i0_push` or `i0_stack` are applied, the red root vanishes. Subsequently, `i0_back_red` cannot be applied. If `i0_back_blue` then gets applied the red root is reintroduced, conserving the existence of a red root within the iteration of the loop. If `i0_back_blue` does not get applied, the break statement is invoked and the procedure terminates. ◀

▶ **Lemma 27.** *Given a nonempty connected input graph G with grey unrooted nodes an unmarked edges, `SearchIndeg0Nodes` outputs G where all the indegree-0 nodes (and only those) are marked blue and connected with blue edges forming a path graph. The blue node with no incoming blue edge is rooted.*

Proof. If G has no indegree-0 nodes, then the lemma is trivially satisfied. So assume G has at least one.

By Lemma 25, `SearchIndeg0Nodes` visits all nodes. Since the right hand side of each rule only contains red and blue nodes, every node is marked either red or blue. The only rules that introduce a blue mark are `i0_push` and `i0_back_blue`, and they turn a red root into a blue root. These rules only get applied if the indegree of said red node is 0. Furthermore, the only edges introduced by `SearchIndeg0Nodes` are blue edges between two blue nodes (in `i0_push`), hence the indegree of a red node is the same as its indegree in the input graph. So only indegree-0 nodes are marked blue. Furthermore, since `SearchIndeg0Nodes` visits, i.e. roots every node of the input graph at some point, all indegree-0 nodes are marked blue, and all non-indegree-0 nodes red.

All rules apart from `i0_push` and `i0_stack` preserve the structure of the subgraph consisting of blue nodes and edges. `i0_stack` only is applied only if `i0_push` is not applicable. But the left hand side of `i0_push` contains a blue root, which can only be created by itself or `i0_stack`. So `i0_push` cannot be applied until `i0_stack` is applied. Since G cannot consist of only indegree-0 nodes (which would mean G is disconnected), `i0_push` can always be matched if the red root has indegree 0. If the red root does not have indegree 0, `i0_stack` cannot be matched either. So the only way for these two rules to match is for `i0_stack` to be matched first and only once, followed by `i0_push` being matched any number of times. Thus, a blue root is created, and then, repeatedly, a new blue node gets connected to the blue root with an outgoing blue edge, while the root moves to the newly added blue node. This construction results in the blue nodes and edges forming a path graph where the node with no incoming edges is a root. ◀

▶ **Lemma 28** (Termination of `ReduceIndeg0Nodes`)**.** *Let G be a connected graph with red non-indegree-0 nodes containing at most one root, and blue indegree-0 nodes that are connected with blue edges forming a path graph. The blue node with no incoming blue edges is a root.*

Given a G as an input, `ReduceIndeg0Nodes` terminates.

Proof. `pop` can only be applied a finite number of times since it reduces the number of nodes in the host graph. So `pop!` terminates.

Claim: During the execution of `ReduceIndeg0Nodes`, `add_bottom` gets applied at most twice.

Proof of Claim: Assume `add_bottom` has already been applied twice, creating the additional nodes u and v. It only gets applied when `nontrivial_stack` cannot be matched, i.e. when the stack consists of only one element. So it adds a blue node to the bottom of the stack. Since every other rule that modifies the stack only does so at the top, u and v must be consecutive and at the bottom. So right after the second application of `add_bottom`, the stack consists of only u and v. Neither u nor v can ever have red neighbours, since there is no rule with an edge incident to a red node in its right hand side. Hence none of the rules in the rule set call between curly braces is applicable, causing `Reduce` to fail, and `add_bottom` never to be applied again.

The rules in the rule set call and `pop` reduce the number of nodes in the host graph by exactly one. So by the claim, they can be applied at most $|V_G| + 2$ times each. So at some point in the loop, they will no longer be applicable. Neither will `add_bottom` since it can only be applied twice. So `Reduce!` terminates. ◀

▶ **Lemma 29.** *Given an input graph G as described in Lemma 28, every node that has no incoming unmarked edges (called* quasi-indegree-0 node*) in some host graph of the execution of `ReduceIndeg0Nodes` gets marked blue.*

Proof. Indeed, the input graph has all quasi-indegree-0 nodes marked blue already. The only rules deleting edges are those from the rule set call (`pop` and `final_pop` cannot delete unmarked edges incident to the node they delete because the dangling condition needs to be satisfied for them to match). So these are the only rules that can create new quasi-indegree-0 nodes. If one of said nodes has indegree 0, it gets detected by the condition of a rule and marked blue. These rules cover each case of how many children their quasi-indegree-0 parent can have in a binary DAG, namely one, one with two parallel edges, and two. The case of no children is covered by `pop` afterwards. They also cover all cases of how many of these children are quasi-indegree-0. So at each execution step, the newly created quasi-indegree-0 nodes get marked blue, proving this lemma. ◀

▶ **Lemma 30.** *Given an input graph G as described in Lemma 28, every node that is marked blue during execution of `ReduceIndeg0Nodes` is not present in the output.*

Proof. Nodes can only be marked blue if an already existing blue node is matched. So it is enough to show that, at some point of the execution, there will be no blue nodes. There are three potential ways to exit the loop `Reduce!`. The first is through the `fail` statement after matching `too_many_children`. This will never happen since the input minus the blue edges is binary, and every rule conserves the blue root having exactly one outgoing blue edge. The second way is for `add_bottom` to fail. This can only happen when there is no blue root. The only rule deleting a blue root is `final_pop`, which is only called after termination of `Reduce!`. Since furthermore, the input is assumed to have a blue root, and every other rule conserves the existence of a blue root, `add_bottom` is always applicable. The third and final way to exit the loop is when none of the rules in the rule set call are applicable. The blue root not having an element below it in the stack cannot be a reason for that, since in that case, `add_bottom` would have been applied. So the current blue root v does not have red neighbours. Since `pop!` has been applied in the previous iteration of `Reduce!`, v was the only blue node in the previous iteration, otherwise it would have been popped. Hence in the current iteration, `add_bottom` was applied, and so the only blue nodes are v and the node created by `add_bottom`, say w. By Lemma 28, `Reduce!` terminates, so this always happens for the given input. As established, v has no children. Neither does w since it was created by `add_bottom` and there is no rule with edges incident to red nodes in its right hand side.

Thus `pop` deletes v, then `final_pop` deletes w, causing all previously blue marked nodes to be deleted. ◀

▶ **Lemma 31** (Correctness of `ReduceIndeg0Nodes` for Binary DAGs)**.** *Given an input graph G as described in Lemma 28, if G minus the blue edges is a binary DAG, `ReduceIndeg0Nodes` yields the empty graph.*

Proof. Assume, for the sake of a contradiction, that the output of `ReduceIndeg0Nodes` contains a node v. By Lemmata 29 and 30, v cannot have been a quasi-indegree-0 node (i.e. an indegree-0 node when ignoring blue edges) at any point during execution. Furthermore, v must have a parent that never was a quasi-indegree-0 node, because otherwise it would have been marked blue by one of the rule set call rules. The same argument can then be applied to the parent's parent, and so on indefinitely. Since the input is finite however, two of these parents must be equal, meaning that there is a cycle. This contradicts the input minus the blue edges being a DAG. ◀

▶ **Lemma 32** (Correctness of `ReduceIndeg0Nodes` for Non-Binary Graphs or Non-DAGs)**.** *Given an input graph G as described in Lemma 28, if G minus the blue edges is either not binary, or not a DAG, then `ReduceIndeg0Nodes` yields a nonempty graph.*

Proof. Assume G is not a DAG. Then it has a directed cycle consisting of consecutive nodes v_1, v_2,$\ldots v_n$. None of these nodes have indegree 0 ignoring blue edges, so they are never matched by the rule set call rules that would mark them blue. Since there are no rules that delete red nodes (only rules that mark them blue), v_1, v_2,$\ldots v_n$ never get deleted. Thus the output is nonempty. Failure cannot occur since every rule and procedure of `ReduceIndeg0Nodes` is either preceded by `try` or followed by `!`.

Now assume that G is a DAG but is not binary. Consider an arbitrary node v of G. The aim is to show that, if v has more than two children (excluding blue edges), then the output is nonempty. By Lemma 29, v gets marked blue at some point of the execution. This can only happen in the rule set call rules. Assume v has just been marked blue by one of these rules. We can also assume that v is rooted since, by Lemma 30, every blue node gets deleted at some point, which can only happen in one of the rule set call rules or in `pop`. The case of it happening in `pop` shall be discarded since that would mean v has no children (disregarding blue edges). Back in the execution right after execution of one of the rule set call rules, since `pop!` cannot fail, the loop `Reduce!` enters its next iteration. The procedure tries to apply `too_many_children` to the blue root. If v has more than two children (disregarding blue edges), it succeeds, and the `fail` statement is invoked, terminating the loop `Reduce!`. Since v has children, both `pop` and `final_pop` do not get applied, for the dangling condition is not satisfied. So the output contains v and is therefore nonempty. ◀

▶ **Lemma 33** (Complexity of `ReduceIndeg0Nodes`)**.** *Given an input graph G as described in Lemma 28 with bounded degree, `ReduceIndeg0Nodes` terminates in linear time.*

Proof. By Lemma 28, the procedure terminates. Every rule is a fast rule schema, and is hence applied in constant time by Theorem 1 (the input is assumed to have bounded degree, and form the input specification, the fact that `unroot` removes a red root if it is present, and the fact that all the other rules conserve the number of roots, there are at most two roots in the host graph at any given point of the execution). So it is enough to show that each of the constantly many rules gets applied a linear number of times. `unroot` and `final_pop` get applied at most once. By the proof of Lemma 28, `add_bottom` gets applied at most twice, and each rule set call rule as well as `pop` at most $|V_G| + 2$ times. `too_many_children` and `nontrivial_stack` can only get reapplied if the rule set call does not fail, which can only happen at most $|V_G| + 2$ times. Hence `ReduceIndeg0Nodes` terminates in linear time. terminates in linear time. ◀

A.4 Topological Sorting Lemmata

In this appendix, we give the proofs of the lemmata needed to support Theorems 13 and 14 from Section 6.

▶ **Lemma 34** (Termination of `top-sort`). *Given a connected DAG G with no roots, grey nodes, and unmarked edges as an input,* `top-sort` *terminates.*

Proof. `sort_forward!` terminates since in each iteration, the number of grey nodes decreases.

For the termination of `SortNodes`, consider the following lexicographical ordering $>$. $H_1 > H_2$ if one of the following three statements are satisfied. H_1 has more grey nodes than H_2, or they have the same number of grey nodes but H_1 has more dashed edges, or they have the same number of grey nodes and dashed edges but H_1 has more red nodes. Let H_1 be the input of an arbitrary iteration of `SortNodes`, and H_2 its output. If `sort_forward` is applied any number of times, $H_1 > H_2$ since the number of grey nodes are reduced. Otherwise, if either `sort_back_push` or `sort_back_stack` is applied, $H_1 > H_2$ since the number of grey nodes is conserved and the number of dashed edges decreases in both rules. Otherwise, either `red_push` or `red_stack` have to be applied, which conserve the number of grey nodes and dashed edges, but decreases the number of red nodes. So in any case, $H_1 > H_2$. For a given graph H_1 consider how many graphs H_2 satisfy $H_1 > H_2$. By definition of $>$, H_1 gives a (finite) upper bound on the number of grey nodes, dashed edges, and red nodes. Hence there are only finitely many possible H_2s. Since `sort_forward!` terminates, and each iteration of the loop reduces the host graph with respect to $<$, `SortNodes` terminates.

Consider `(try unsorted then SortNodes; search_forward)!`. If `search_forward` cannot be applied, the loop terminates. It is the only rule in this loop that increases the number of looped edges in the graph. Due to its predicate, it can only add looped edge to a node if it does not already have one. Furthermore, no rule decreases the number of looped edges. So for an arbitrary input graph H for the loop, at most $|V_H|$ looped edges can be added before `search_forward` fails. Hence the loop terminates (knowing that `SortNodes` also terminates).

Finally, consider the loop that `SearchUnsortedNodes` consists of. Furthermore, consider the lexicographic ordering $>$ defined by $H_1 > H_2$ if H_2 has more nodes with looped edges than H_1, or they have the same number of nodes with looped edges but H_2 has less dashed edges than H_1. By an argument similar to that of Lemma 21, `SearchUnsortedNodes` terminates. ◀

For the correctness of `SortNodes`, the following concepts needs to be defined. In a graph G, a *directed path* from a node v to a node w is a sequence of distinct nodes v_1, v_2,..., v_n such that $v_1 = v$ and $v_n = w$, and for each i where $1 \leq i \leq n-1$, there is an edge of source v_i and of target v_{i+1}.

A directed path from v to w is called *grey-noded* if all the nodes it consists of, except possibly v, are marked grey.

Given a node v in a DAG G, let its *descendants* $\mathrm{Desc}_G(v)$ be defined as the subgraph of G induced by the

$$\{w \in V_G \,|\, \text{there is a directed grey-noded path from } v \text{ to } w\} \cup \{v\}.$$

▶ **Lemma 35** (Correctness of `SortNodes`). *Assume the input graph of* `top-sort` *has no blue edges. Let G be a connected DAG with a single red root v, where the nodes of $Desc_G(v)$ are unrooted. Furthermore, let G have an additional root that is either unmarked and disconnected, or green and connected to the rest of the graph with an outgoing green edge. Let H be the output of* `SortNodes` *applied on G. Consider the binary relation $\leq$ on nodes of $Desc_H(v)$ defined by $u \leq w$ if there is a directed path from u to w, or if $u = w$, such that all of the involved edges are blue. Then $\leq$ defines a topological sorting on $Desc_H(v)$ minus the blue edges.*

Proof. Since the input graph of `top-sort` has no blue edges, any that are present in the host graph were created by rules. Whenever these rules create blue edges, they mark the incident nodes blue. No rule removes a blue mark, so the subgraph of the host graph induced by the blue edges always exclusively consists of blue nodes. Furthermore, every rime a node gets marked blue, the green root points towards it. And when a new blue edge gets created, the target node must also have the green root pointing towards it, and the source node must be a red root. So the procedure only adds a blue edge from a non-blue to the node that has most recently been marked blue. From this construction, we can infer that the graph induced by the blue edges is a path graph. Furthermore, no blue looped edges are introduced. So there can be no path from a node u to a node w and vice versa. Hence if $u \leq w$ and $w \leq u$, u and w must be equal by definition of $\leq$, i.e. $\leq$ is antisymmetric.

From the definition of $\leq$, it is clear that transitivity holds due to path concatenation resulting in paths.

With a proof similar to that of Lemma 22, one can show that `SortNodes` turns every node of $\mathrm{Desc}_G(v)$ into a red root. Furthermore, all the red roots become blue nodes incident to blue edges. So $\leq$ is connex.

To show that the topological property holds, consider two nodes u and w of $\mathrm{Desc}_H(v)$, both of which being distinct from v (v itself will be handled later). So by definition, there is path of non-blue edges from v to u, and one from v to w. We can assume without loss of generality that u becomes a red root before w. If there is no edge between u and w, the topological property imposes no constraint on said pair of nodes. If there is an edge from u to w, `sort_forward` gets applied again, dashing said edge and turning w into a red root. Hence later in the execution, w gets pushed before u, ensuring that the topological property is satisfied. If there is an edge from w to u, there can be no non-blue path from u to w since the input is a DAG. Hence u will be pushed before w, satisfying the topological property again. As for v, any condition involving it must have it as the source node by definition of $\mathrm{Desc}_H(v)$. Since v is pushed last, the topological property is satisfied.

◀

▶ **Lemma 36.** *Given an input G as described in Lemma 35, the output of* `SortNodes` *has the same dashed edges, and the red root in the same place as G.*

Proof. Let v be the red root of G. During the execution of `SortNodes`, there is always a path of dashed edges from v to the current red root, since `sort_forward` is the only rule of `SortNodes` with dashed edges in its right hand side and generates a path graph of red nodes and dashed edges, and since `sort_back_stack` and `sort_back_push` only remove the latest node from that path graph. The only way for their encompassing loop to end is for both of these rules not to be applicable. By the previous argument, this means that there are no dashed edges in said path graph left, and v is the red root when `SortNodes` terminates. ◀

▶ **Lemma 37** (Correctness of `SearchUnsortedNodes`)**.** *Let G be a connected acyclic graph with grey nodes and unmarked edges, except for a disconnected unmarked root and a red looped edge on a unique grey root. Given G as an input, `SearchUnsortedNodes` yields a graph such that the ordering from Lemma 37 extended to the entire output graph is a topological sorting.*

Proof. `SearchUnsortedNodes` is similar to `general-dfs` from Section 4, but instead of read marks, red looped edges are used. Furthermore, `try unsorted then SortNodes` is added, none of whose rules modify red looped edges. Also, after application of `SortNodes`, the red root remains at the same place, and the same edges remain dashed. So a reasoning similar to that in the proof of Lemma 22 can be used to justify that `SearchUnsortedNodes` visits every node of its input graph.

`SearchUnsortedNodes` applies `SortNodes` to each of these visited nodes that are marked grey, say v, and implements a stack on $\mathrm{Desc}(v)$ defining a topological sorting. Clearly, the subgraph induced by the union of all these descendant graphs is just the output graph. So the concatenation of their topological sortings is a topological sorting of the entire output graph. ◀

▶ **Lemma 38** (Complexity of `top-sort`)**.** *Given a connected acyclic graph of bounded degree with grey unrooted nodes and unmarked edges G as an input, `SearchUnsortedNodes` terminates in linear time.*

Proof. First, let us give an upper bound to the number of applications of each rule. `init` is applied exactly once. Since `init` is the only rule having an unmarked root in its right hand side, and the input has no unmarked roots, `red_stack` and `sort_back_stack` can be matched at most once (in total). `unsorted` and `sort_forward` reduce the number of grey nodes by one. Since all the other rules conserve the number of grey nodes, and the input graph has $|V_G|$ grey nodes, they can be applied at most $|V_G|$ times in total. Similarly, `search_forward` (and `init`) reduce the number of nodes with no red looped edge by one. So they can also only be applied at most $|V_G|$ times in total. `red_push` and `sort_back_push` (as well as `red_stack` and `sort_back_stack`) are the only rules not to conserve the number of blue nodes, and reduce the number of non-blue nodes by exactly one. Since the input graph has no blue nodes, they can be applied at most $|V_G|$ times in total. As for `search_back`, a reasoning as in Corollary 23 can be used to justify `search_back` is applied an at most linear amount of times, since `SortNodes` conserves the number of dashed edges by Lemma 36.

`init` is the only rule to increase the number of roots, specifically by two. All the other rules conserve the number of roots. So since the input graph has no roots, there is a constant number of roots at any point during the execution of `top-sort`.

The only rules that are not fast are `init` due to the lack of roots, and `search_forward` due to the `edge` predicate. So by Theorem 1, all the other rules can be matched in constant time since the input has bounded degree. `init` is matched in constant time since it matches any input node. As for `search_forward`, since the input has bounded degree and the rules cannot create an unbounded number of edges incident to a single node, the predicate `edge(2,2)` only has to check a constant number of incident edges.

Since each rule is matched a linear number of times in constant time, and the program terminates by Lemma 34, `top-sort` terminates in linear time. ◀