Linear-Time Graph Programs for Unbounded-Degree Graphs

— Extended Version —

Ziad Ismaili Alaoui and Detlef Plump

Department of Computer Science, University of York
York, United Kingdom
{z.ismaili-alaoui, detlef.plump}@york.ac.uk

Abstract. Achieving the complexity of graph algorithms in conventional languages with programs based on graph transformation rules is challenging because of the cost of graph matching. Previous work demonstrated that with so-called *rooted* rules, certain algorithms can be executed in linear time using the graph programming language GP 2. However, for non-destructive algorithms which retain the structure of input graphs, achieving a linear runtime required that input graphs have a bounded node degree. In this paper, we show how to overcome this restriction by enhancing the graph data structure generated by the GP 2 compiler and exploiting the new structure in programs. As a case study, we present a 2-colouring program that runs in linear time on connected input graphs with arbitrary node degrees. We prove the linear time complexity and also provide empirical evidence in the form of timings for various classes of input graphs.

Keywords: Rooted graph programs · Efficient graph matching · GP 2 · Linear-time algorithms · Depth-first search · 2-colouring

1 Introduction

Designing and implementing languages for rule-based graph transformation, such as GReAT [1], GROOVE [9], GrGen.Net [10], Henshin [12] or PORGY [8], is challenging in terms of performance. Typically, there is a gap between the runtime that can be achieved with programs in conventional imperative languages and rule-based graph programs. The bottleneck for 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 $|G|^{|L|}$, where $|X|$ is the size of a graph X. (This is a polynomial since L is fixed.) As a consequence, linear-time imperative graph algorithms may have a polynomial runtime when they are recast as rule-based graph programs.

To mitigate this problem, the graph programming language GP 2 supports *rooted* graph transformation rules which were first proposed by Dörr [7]. 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 certain conditions, to match rules in constant time. The GP 2 compiler [2] maintains a list of pointers to roots in the host graph, hence allowing to access roots in constant time if the number of roots throughout a program's execution is bounded. In [3], *fast* rules were identified as a class of rooted rules that can be applied in constant time if host graphs contain a bounded number of roots and have a bounded node degree.

The first linear-time graph problem implemented by a GP 2 program with fast rules was 2-colouring. In [3, 4], it is shown that this program colours connected graphs of bounded degree in linear time. Since then, the GP 2 compiler has received some major improvements, in particular relating to the runtime graph data structure used by the compiled programs [6]. These improvements made a linear time worst-case performance possible for a wider class of programs, in some cases even on input graph classes of unbounded degree. See [5] for an overview.

Despite this progress, programs that retain the structure of input graphs, such as the said 2-colouring program, up to now required non-linear runtimes on graphs of unbounded degree. The problem is that during a depth-first search, the number of failed attempts to match an edge incident to a node may increase over repeated visits to this node. As a consequence, in graph classes of unbounded degree, this number may grow quadratically in the graph size, leading to a quadratic program runtime. In graph classes of bounded degree, this undesirable behaviour is ruled out because the maximal number of failed matching attempts per node is constant.

In this paper, we present an update to the GP 2 compiler which mitigates this performance bottleneck. In short, the solution is to improve the graph data structure generated by the compiler in that the edges incident with a node are stored in separate linked lists if they have different *marks* (red, green, blue, dashed or unmarked). This allows the matching algorithm to find an incident edge in constant time. For example, if an unmarked edge is required, a single access to the list of unmarked incident edges will either find such an edge or determine that none exists.

In addition to the new graph data structure, a technique is needed to exploit the improved storage in programs. We demonstrate in a case study of the 2-colouring problem how the new graph representation allows to achieve a linear runtime on graphs with arbitrary node degrees. Our program expects connected input graphs and either detects that a graph is not 2-colourable or colours the nodes blue and red such that all non-loop edges link nodes of different colour.

We provide a detailed proof that this program runs in linear time on arbitrary connected input graphs. To the best of our knowledge, such a demonstration of a rule-based linear-time 2-colouring algorithm does not exist in the literature. We also present the results of timing experiments with the colouring program on six different graph classes containing graphs with up to one million nodes and edges.

2 The Problem with Unbounded-Degree Graphs

We refer to [5] for a description of the GP 2 programming language. Previous versions of non-destructive GP 2 programs based on depth-first-search showed a linear time complexity on graph classes of bounded degree but a non-linear runtime on graph classes of unbounded degree [5].

For example, the program `is-connected` in Figure 1 checks the connectedness of a graph. It fails if and only if an input graph, consisting only of arbitrarily labelled unmarked edges and grey-marked nodes, is not connected. The rule `init` first initialises an arbitrary grey node in the graph as a root, and should that rule apply (i.e. the graph is non-empty), the procedure `DFS!` performs a depth-first search of the connected component of the node initialised by `init`. The rule `forward` marks each newly visited node blue, and `back` unmarks it once it is processed. The procedure `DFS` ends when `back` fails to find a match, indicating that the search is over. The rule `match` in `Check` checks whether a grey-marked node exists in the graph following the execution of `DFS!`, which exists if and only if the input graph contains more than one connected component. If that rule applies, the graph is disconnected, and the program invokes the command `fail`. Otherwise, it terminates without invoking the `fail` command.

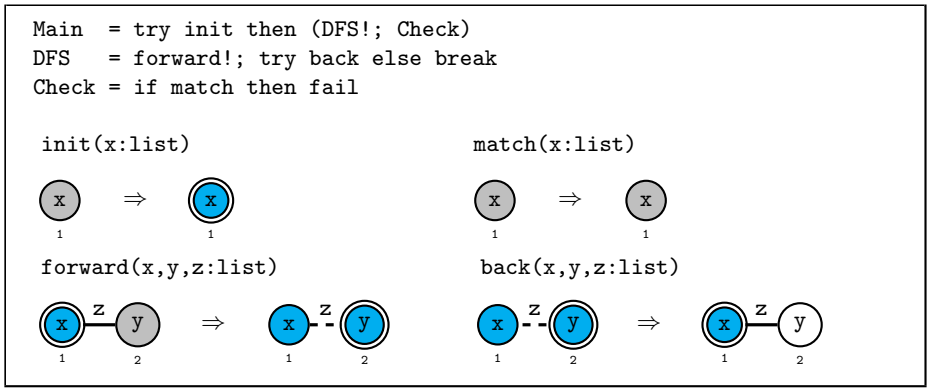

Fig. 1. The old program `is-connected`.

It can be shown that the program `is-connected` runs in linear time for bounded-degree graph classes. However, as the following example shows, the program may require non-linear time on unbounded-degree graph classes.

Figure 2 shows an execution of `is-connected` on a star graph with 8 edges (see also Figure 11). The numbers below the graphs show the ranges of attempts that the matching algorithm may perform. For instance, in the second graph of the top row, either a match is found immediately among the edges that connect the

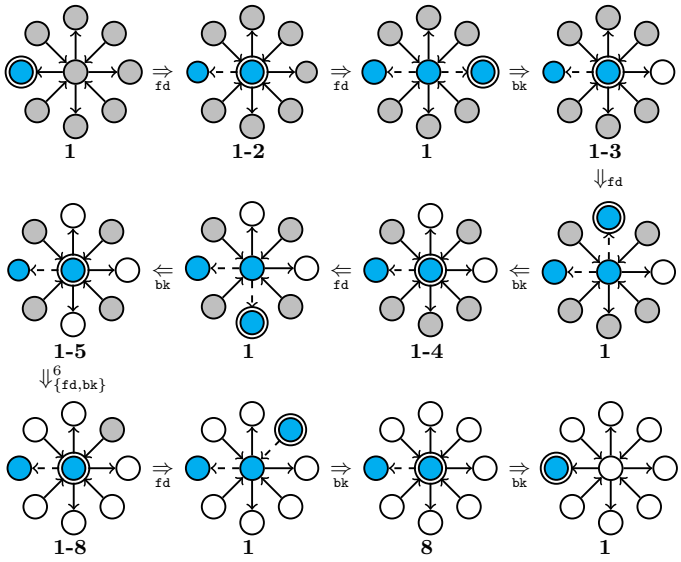

Fig. 2. Matching attempts with the `forward` rule.

central node with the grey nodes, or the dashed edge is unsuccessfully tried first. In order to find a match for the rule `forward`, the matching algorithm considers, in the worst case, every edge incident with the root. When the node central to the graph is rooted and the rule `forward` is called, the matching algorithm may first attempt a match with the dashed back edge and all edges incident with an unmarked node. Therefore, the maximum number of matching attempts for `forward` grows as the root moves back to the central node. As can be seen from this example, the worst-case complexity of matching `forward` throughout the program's execution is $2|E| + \sum_{i=1}^{|E|} i = \mathcal{O}(|E|^2)$ where E is the set of edges.

3 The Updated GP 2 Compiler

To address the problem described in Section 2, we changed the GP 2 compiler described in [6], which we refer to as the *2020 compiler*. We call the version introduced in this paper the *new compiler*[1].

3.1 New Graph Data Structure

The 2020 compiler stored the host graph's structure as one linked list containing every node in the graph, with each node storing two additional linked lists: one for incoming edges and one for outgoing edges. When iterating through edge lists to find a particular match for a rule edge, the 2020 compiler had

[1] Available at: `https://github.com/UoYCS-plasma/GP2`.

to traverse through edges with marks incompatible with that of the rule edge. This resulted in performance issues, especially if nodes could be incident to an unbounded number of edges with marks incompatible with the edge to be matched. For example, consider the rule `blue_red` from Figure 6. Initially, the matching algorithm matches node `1` from the interface with a root node in the host graph. Subsequently, it iterates through the node's edge lists to locate a match for the red edge. In the 2020 compiler, all edges incident to this node were stored within two lists, one for each orientation, irrespective of their marks. However, if the node were incident to a growing number of unmatchable edges (because of mark changes), the matching algorithm would face, in the worst case, a growing number of iterations through the edge lists to find a single red edge.

When considering a match for a rule edge, host edges with incorrect orientation and incompatible marks do not match; thus, the matching algorithm need not iterate through them. By organising edges into homogeneous linked lists as array entries based on their marks and orientations, the matching algorithm can selectively consider linked lists of edges of correct orientation and mark. More precisely, in the new compiler, we update the graph structure of the 2020 compiler by replacing the two linked lists with a two-dimensional array. Each element of the array stores a linked list containing edges of a particular mark and orientation. We also consider loops to be a distinct type of orientation, separate from non-loop outgoing and incoming edges. The 2D array, therefore, consists of 5 rows (unmarked, dashed, red, blue, green) and 3 columns (incoming, outgoing, loop), totalling 15 cells, each one storing a single linked list. Consider

	in	out	loop
unmarked	...	...	...
dashed	...	...	...
red	...	...	...
green	...	...	...
blue	...	...	...

Fig. 3. Two-dimensional array of linked lists of edges.

again the rule `blue_red` of `2-colouring` from Figure 6. In the new compiler, the matching algorithm can access the linked list of non-loop incoming red edges and that of non-loop outgoing red edges in constant time and only consider these edges. Other edges, such as blue edges incident to the matched node, are stored in separate linked lists and thus are not considered. In this specific instance, it can be shown that there is at most one red edge in the host graph throughout the execution of `2-colouring` (Proposition 3). Therefore, there can be at most one edge in either list, and a matching attempt will either find such an edge or determine that none exists, both in constant time. However, under the 2020 compiler, the presence of non-red edges in the list could result in longer, non-constant search times, as, in the worst-case scenario, all edges except the red one would need to be iterated over.

Procedure	Description	Complexity
`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(m)`	**Given a node, fetch the first incoming edge of mark `m`.**	$O(1)$
`nextInEdge(m)`	**Given a node and an edge of mark `m`, fetch the next incoming edge of mark `m`.**	$O(1)$
`firstOutEdge(m)`	**Given a node, fetch the first outgoing edge of mark `m`.**	$O(1)$
`nextOutEdge(m)`	**Given a node and an edge of mark `m`, fetch the next outgoing edge of mark `m`.**	$O(1)$
`firstLoop(m)`	**Given a node, fetch the first loop edge of mark `m`.**	$O(1)$
`nextLoop(m)`	**Given a node and an edge of mark `m`, fetch the next loop edge of mark `m`.**	$O(1)$
`getInDegree`	Given a node, fetch its incoming degree.	$O(1)$
`getOutDegree`	Given a node, fetch its outgoing degree.	$O(1)$
`getMark`	Given a node or edge, fetch its mark.	$O(1)$
`isRooted`	Given a node, determine if it is rooted.	$O(1)$
`getSource`	Given an edge, fetch the source node.	$O(1)$
`getTarget`	Given an edge, fetch the target node.	$O(1)$
`parseInputGraph`	Parse and load the input graph into memory: the host graph.	$O(n)$
`printHostGraph`	Write the current host graph state as output.	$O(n)$

Fig. 4. Updated runtime complexity assumptions. Modified procedures are highlighted in grey, and added ones, in blue. n is the size of the input.

3.2 New Programs

The `is-connected` program of Figure 1 does not yet run in linear time under the new compiler. Figure 5 shows the runtimes of the program on star graphs and, for comparison, linked lists (see Figures 10 and 14). As a consequence, we need a new technique to exploit the improved graph data structure in programs. Indeed, a rule that matches a rooted node adjacent to an unvisited node via an

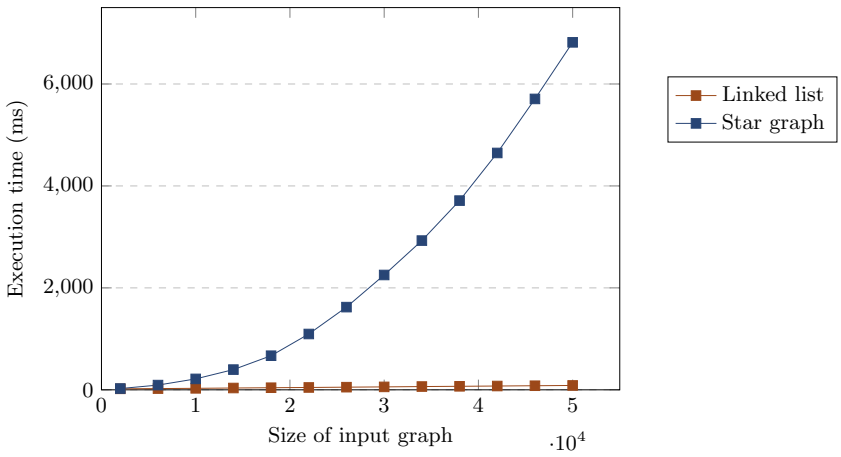

Fig. 5. The program `is-connected` running on the new compiler.

unprocessed edge would require the matching algorithm to iterate through all unprocessed edges incident to the root, which includes cross edges. To mitigate

this problem, we implement the *forward* operation by marking an unprocessed edge incident to the root in a colour such that it is the only edge incident to the root marked of that colour. Then, we check the mark of the node adjacent to the root via the uniquely-coloured edge: if it is marked such that it is visited, we ignore it by marking the uniquely-coloured edge in the mark denoting that it is processed, otherwise, we move the root to the unvisited node and mark the edge as being processed.

In order to reason about programs, it is primordial to lay down assumptions on the complexity of certain elementary operations. We define the *search plan* of a rule as the procedure generated to compute a match satisfying the application condition, should one exist. Figure 4 showcases the complexity assumptions of the basic procedures of the search plan, adapted from [5]. The grey rows indicate existing procedures updated by the changes introduced in this paper and the blue rows, new procedures. The proof of Theorem 2 of the new `2-colouring` program relies on these complexity assumptions, and the empirical evidence showcased in Figure 15 corroborates them.

4 Case Study: Two-Colouring

Vertex colouring is a widely common graph problem in computer science and finds its application in myriad domains, including scheduling, compiler optimisation and register allocation [11]. In 2012, Bak and Plump conducted research on the feasibility of developing an efficient rule-based algorithm for the 2-colouring problem, matching the linear-time complexity achieved by conventional imperative programming languages, using rooted rule schemata [3]. Bak and Plump's 2-colouring GP 2 program achieved linear time complexity on bounded-degree graph classes but exhibited quadratic time complexity on graph classes of unbounded degree [4].

In this section, we present the `2-colouring` program (Figure 6), which achieves linear runtime on both bounded- and unbounded-degree graph classes using the exact program specifications (i.e. input and output conditions) as Bak and Plump's. This achievement is made possible by exploiting the improvements to the compiler described in Section 3, namely, the separation of edge lists with respect to marks and orientation.

The program `2-colouring`[2] expects a host graph satisfying the input conditions laid down in Definition 1 and 2-colours it by performing a depth-first search (DFS) from an initial node, colouring each newly visited node in the colour contrasting that of the node it is visited from (either red or blue). The program fails if an edge is found to be incident to two non-grey nodes of the same mark. Figure 7 provides a sample execution of `2-colouring`.

In contrast to previous implementations of the 2-colouring, the linearity of this program's runtime is primarily attributed to an invariant ensuring that a

[2] The concrete syntax of the program available at: `https://gist.github.com/ismaili-ziad/51cc29fa3ea49a49d1922acd560ce3ee`.

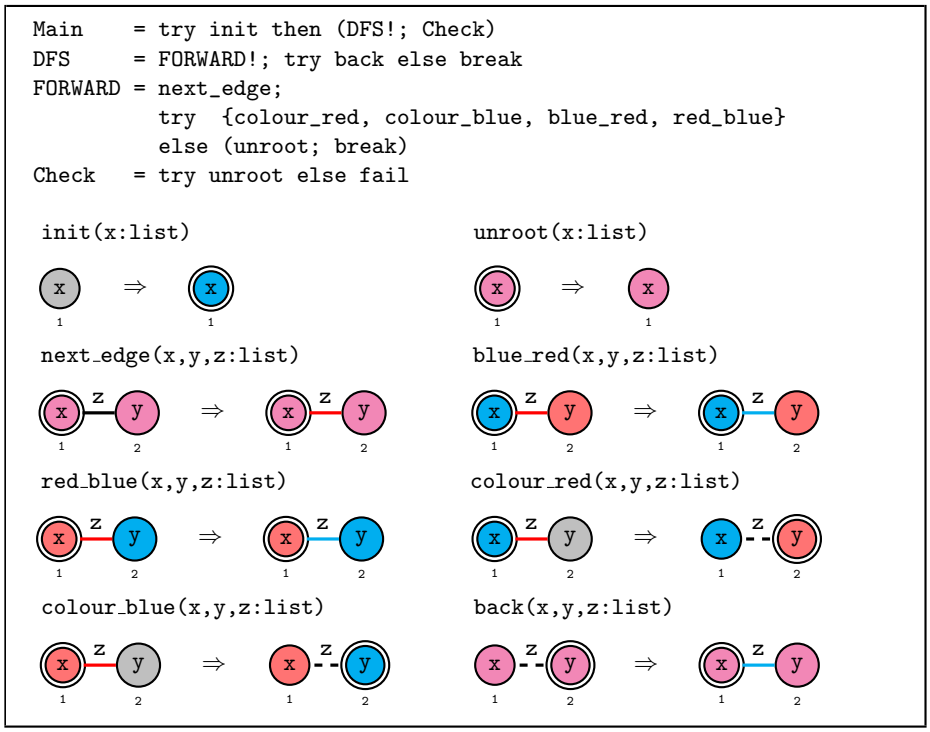

Fig. 6. The program `2-colouring` (magenta represents the mark *any*).

single edge incident to the root is marked red, allowing rules implementing a depth-first search to process both forward and cross edges, instead of forward edges only. Invariant 3 shows that there can be, at most, one red edge at a time in the host graph throughout the execution of the program, allowing for instantaneous access with the compiler's recent optimisations. Furthermore, a blue mark on an edge indicates that its processing has ended, eliminating the need for it to be matched again as a forward or cross edge in the DFS traversal.

We first demonstrate that `2-colouring` is totally correct. Then, we show that the program is linear with respect to the size of the input graph on any class of input graphs. Finally, we provide empirical evidence on various graph classes of bounded and unbounded degrees to corroborate our claim.

For the purposes of this section, the set of nodes of a graph G is denoted as V_G. The cardinality of a set X is represented by $|X|$. We use the notation $A \Rightarrow_r B$ to indicate that B results from applying r on A. For convenience, we define `COLOUR` to be the set {`colour_red, colour_blue`} and `IGNORE`, the set {`blue_red, red_blue`}. When we refer to the application of `COLOUR`, we mean that either `colour_red` or `colour_blue` is applied. Similarly, applying `IGNORE`

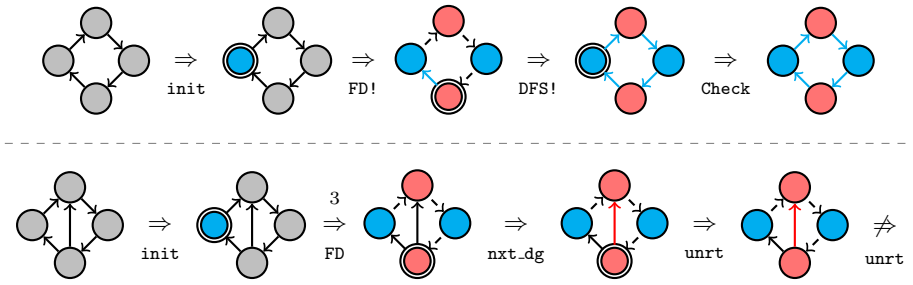

Fig. 7. Sample executions of `2-colouring` on a 2-colourable (top) input graph and a non-2-colourable (bottom) input graph. `FD`, `nxt_dg` and `unrt` denote `FORWARD`, `next_edge` and `unroot`, respectively.

means applying either `blue_red` or `red_blue`. By a *rule call* or a *rule invocation*, we mean a completed or failed rule or procedure application, the `break` operation or the `fail` operation.

We first lay down the definition of an input graph.

Definition 1 (Input graph)**.** An *input graph*, within the context of the two-colourability problem, is an arbitrarily-labelled *connected* GP 2 host graph such that:

1. every node is marked grey,
2. every node is non-rooted, and
3. every edge is unmarked.

Let us first examine the correctness of `2-colouring`.

Invariant 1. Throughout the execution of `2-colouring` on a given input graph, the following invariant holds: edges marked as blue and nodes marked as either blue or red retain their respective marks unchanged.

Proof. Upon inspection of the rules, none modifies the mark of a blue node, a red node, a blue edge or an *any*-marked element. Thus the invariant holds. □

Invariant 2. Throughout the execution of `2-colouring`, the following invariant holds: there is at most one root in the host graph and is incident to at most one dashed edge.

Proof. `init` and `unroot` are the only rules that do not preserve the cardinality of roots in the host graph. The `init` rule increases the number of roots by 1, while the `unroot` rule decreases it by 1. Since `init` is only called once, there is at most one root node throughout the execution of `2-colouring`.

For the following argument, observe that `colour_red` and `colour_blue` are the only rules generating dashed edges in the host graph. Let u, v and e denote node `1`, node `2` and the edge of either rule, respectively.

Consider the step $G \Rightarrow_{\texttt{COLOUR}} H$ and let the nodes u' and v' be the images of u and v by that production, respectively. Define $d(x)$ as the number of dashed edges incident to node x. Then, $d(v') = d(v) + 1$ and $d(u') = d(u) + 1$. To maintain the invariant, $d(v)$ must be 0. To show this, consider the fact that v is grey. Upon inspection of the program, there is no rule that generates a dashed edge incident to a grey node. Indeed, any generated edge must be incident to either a red or a blue node. Furthermore, as established in Proposition 1, red and blue nodes retain their mark, implying that a non-grey node incident to a dashed edge cannot subsequently turn grey.

Hence, since v is grey, it is guaranteed not to be incident to a dashed edge prior to the application of `COLOUR`; thus, $d(v') = d(v) + 1 = 1$. Given that the rule `back` does not increase the incidence of a node to dashed edges, the property remains satisfied. □

Proposition 1. Upon the execution of `2-colouring` on an input graph G, the rule `unroot` in `DFS!` is applied if and only if G is not 2-colourable.

Proof. The lemma is trivially true if G is empty. Assume G contains at least one node. Let us split the proof into two distinct cases.

Case 1. *G is 2-colourable.* Let X and Y be disjoint subsets of V_G with respect to the bipartition of G such that $X \cup Y = V_G$ and every edge in G shares endpoints in both X and Y. Thus, no pair of distinct nodes within the same subset are adjacent in G. Clearly, given that the program is structure-preserving, X and Y remain the same throughout the execution of `2-colouring`. Let G' be the initialised graph such that $G \Rightarrow_{\texttt{init}} G'$. Suppose that `init` roots and marks a node blue in X. Consider the graph H such that $G' \Rightarrow^{n}_{\texttt{FORWARD}} H$ for some $n \geq 0$. By induction on n, we show that two non-grey nodes sharing the same mark are never adjacent.

When $n = 0$ (base case), the body `DFS!` has not been invoked once. Hence, H consists of (possibly zero) grey nodes and a single blue root node in X created by `init`, which does not violate the property.

Now, assume the property holds in H for some n; we show it still holds for $n + 1$. At the invocation of `FORWARD`, the rule `next_edge` is called. If there is no unmarked edge incident to the root node, the rule fails to match, and the loop exits, satisfying the condition. Otherwise, `next_edge` matches node `1` to the root node and node `2` to some arbitrarily marked node adjacent to the root via an unmarked edge. Since the host graph is 2-colourable, `next_edge` preserves marks and it is assumed, by the induction hypothesis, that no pair of adjacent nodes are both marked blue or red, node `2` must be part of the subset opposite to the root's and marked differently. That is, if node `1` is in X, node `2` is in Y, and vice versa. Otherwise, it would imply two nodes within the same subset share an edge, violating the assumption on G. At the invocation of `try {colour_red, colour_blue, blue_red, red_blue}`, following `next_edge`, node `2` must either have a mark opposite to the root's (i.e. blue if the root is red and vice versa) or be grey. Hence, one rule within that body must apply and move the root to the opposite subset while alternating its mark, preserving the property.

Therefore, upon the execution of `FORWARD`, either `next_edge` fails to apply and exits the loop, or it does apply and a rule in the `try` condition consequently matches and applies. Given that the rule `back` does not affect marks, `unroot` in `DFS!` is never invoked.

Case 2. *G is not 2-colourable.* By definition, there exists no assignment of marks from the set $\{blue, red\}$ to every node in G such that every edge has endpoints of different marks. Thus, a program that colours nodes in G either blue or red has to violate the two-colourability condition in that at least one pair of adjacent nodes share the same colour. Indeed, the node-marking rules of `DFS!` are `colour_red` and `colour_blue`, and each rule colours a node in the contrasting colour to that of the adjacent root. As such, an eventual application of these rules will result in two non-grey nodes sharing the same mark. Let w be node `2` following such an application of either `colour_red` or `colour_blue` on the host graph. w is either blue or red, rooted and adjacent to some node x that shares its mark. Any edge that connects w and x at this instance must be unmarked since a mark would imply that the edge was previously matched by `next_edge`, and subsequently by a rule in `try {colour_red, colour_blue, blue_red, red_blue}`. One of two situations occurs at the next invocation of `FORWARD`: either `next_edge` matches w and x (1), or it matches w and a different neighbour distinct from x (2).

1. Each rule in `COLOUR` and `IGNORE` is invoked, but none matches. Consequently, the `unroot` rule, applicable to w, is invoked and applied, leading to the call of `break` and the termination of the `DFS!` loop.
2. Let y be the neighbour of w distinct from x matched by the rule `next_edge`. Three subcases emerge: y admits of the same mark as w (a); y is marked in the colour opposite to w (b); y is grey (c).
   (a) The same reasoning as Situation (1) applies.
   (b) As both w and y are of contrasting colours and connected by a red edge, `IGNORE` applies. This ends the current instance of `FORWARD`, although not necessarily terminating the loop, and brings us back to the beginnings of Situations (1) and (2).
   (c) After `COLOUR` is applied, w is unrooted, and y becomes the new root, marked with the colour opposite to w. Additionally, the edge matched by the rule is dashed. Invariant 2 establishes that the root node is incident to at most one dashed edge. Consequently, any subsequent application of `back` successfully backtracks the root to its ancestral node in the depth-first search tree.
   The loop `DFS!` terminates upon the failure of the `back` rule. There are two possible scenarios: either the root eventually backtracks to node w before `DFS!` terminates, returning us to Situations (1) and (2), or `DFS!` breaks prior to w being rooted again. The latter can only happen if `back` is no longer applicable.
   However, since `COLOUR` creates a path of dashed edges with a single endpoint rooted, the inapplicability of `back` can only occur due to the removal of the root, i.e. the application of `unroot`. In either case, `unroot` is invoked, causing `FORWARD!` to break, `back` to fail, and `DFS!` to terminate.

Therefore, executing `2-colouring` on a non-2-colourable input graph results in the eventual application of `unroot` in `DFS!`. □

The next lemma demonstrates that termination of `2-colouring` is ensured by showing that the body of each loop reduces a measure that assigns a non-negative integer to each host graph, thereby showing that the loop body eventually fails.

Lemma 1 (Termination of `2-colouring`)**.** On any host graph, the program `2-colouring` terminates.

Proof. The program `2-colouring` contains two looping procedures: `DFS!` and `FORWARD!`. To show termination, consider a measure $\#(X)$ consisting of the number of unmarked edges in the host graph X. The rule `next_edge` is invoked at the beginning of `FORWARD`, and if it fails to match, the loop breaks. Clearly, an application of `next_edge` reduces the measure $\#$. Since the number of edges is finite and no rule in `2-colouring` creates or unmarks an edge, `FORWARD!` terminates. We now show that the upper body `DFS!` terminates. This time, consider $\#(X)$ to consist of the number of non-blue edges in the host graph X. The loop `DFS!` breaks if and only if `back` is called and fails to apply. Let H be the resulting graph of an application of `back` on G, that is, $G \Rightarrow_{\texttt{back}} H$. Clearly, $\#(H) < \#(G)$ since the rule marks a dashed edge blue and preserves the size (i.e. $|H| = |G|$). Similarly, given that blue edges retain their marks throughout the execution of `2-colouring` (Invariant 1), the number of edges is finitely fixed and `FORWARD!` is known to terminate, dashed edges are eventually exhausted, the `break` command is called, and `DFS!` terminates. □

Lemma 2 shows that the program 2-colours the graph, should it be 2-colourable. It demonstrates that the existence of an unvisited (grey) node following the termination of `DFS!` on a 2-colourable graph leads to a contradiction.

Lemma 2. Consider a *2-colourable* input graph G. Upon termination of `DFS!` on G, the host graph contains no grey nodes.

Proof. The lemma is trivially true if G is empty since the rule `init` fails to apply and the body `(DFS!; Check)` is not invoked. Suppose G consists of at least one node, thereby making `init` applicable, and consider G' and H such that $G \Rightarrow_{\texttt{init}} G' \Rightarrow_{\texttt{DFS!}} H$. For the sake of contradiction, assume that a grey node exists in H. The rule `init` creates a blue node in G'. Note, upon inspection of the rules, that the program is structure-preserving. Therefore, both G' and H are 2-colourable and connected. As per Invariant 1, once a node is turned blue, its mark is no longer modified. This implies at least one grey node in H (assumption) is adjacent to some non-grey node by the connectedness of H. Let u and v be the non-grey (blue or red) and grey nodes, respectively. We show that u and v are matched by a rule in `COLOUR` prior to the termination of `DFS!`, thereby contradicting the assumption.

Given that u is non-grey in G, and non-grey nodes preserve their mark, it must have been matched by either `init` or a rule in `COLOUR`. Either way, u

must have been a non-grey root node. Recall that `DFS!` terminates if and only if `back` fails to apply, which can only occur following the termination of the loop `FORWARD!`. The latter breaks if either condition holds: `next_edge` fails to apply; no rule in `COLOUR` ∪ `IGNORE` applies. The second condition implies that the body `(unroot; break)` is then called. However, as shown in Proposition 1, `unroot` is never invoked if G is 2-colourable. Therefore, the second condition can never hold; that is, a rule in `COLOUR` ∪ `IGNORE` is always applicable upon its invocation. Regarding the first condition, let us examine the implicit data structure `DFS!` generates. The dashed edges form a path of non-grey nodes, wherein an endpoint is rooted. This models a stack of nodes where the root represents the top element. Specifically, `init` initialises the stack, `COLOUR` executes the *push* operation, and `back` performs the *pop* operation. Given that a root node can be incident to, at most, one dashed edge, it is guaranteed to backtrack to its ancestral node in the depth-first search tree as `back` is applied. The inapplicability of `back` can be seen as an exhaustion of the stack, which can only occur if there are no dashed edges in the host graph (as previously stated, the application of `unroot` is not a possibility). It is easy to observe that the remaining root at the end of `DFS!` is the initial node of the stack (i.e. the first pushed node). If u is rooted, then it is the initial node (i.e. the node `init` was applied to). The invocation of `back` follows the termination of `FORWARD!`, which only occurs if `next_edge` is not applicable, provided the graph is 2-colourable. However, `next_edge` is applicable on u and v, hence a contradiction. If u is non-rooted, it must have been a root at some point during the execution of `DFS!` prior to it being popped from the so-called stack. Again, an analogous argument shows that this leads to a contradiction.

Since u and v ought to have been matched by `next_edge`, one of two outcomes must have occurred: either `COLOUR` successfully applied, or the statement `(unroot; break)` was invoked. However, since G is 2-colourable, the `unroot` rule cannot be applied. Therefore, `COLOUR` must have been applicable, resulting in v being marked non-grey. As established in Invariant 1, non-grey nodes retain their colour assignment. Hence, v cannot be grey in H.

Given the following properties:

- nodes in the host graph can only be grey, blue or red;
- a node adjacent to a non-grey node cannot be grey;
- G is 2-colourable and connected;
- blue and red nodes retain their mark; and
- the rule `init`, prior to `DFS!`, creates one non-grey node in the host graph;

it follows that every node in H (i.e. the resulting graph following the execution of `DFS!`) is non-grey. □

Building upon the previous lemmata and propositions, we now show the correctness of `2-colouring`.

Theorem 1 (Correctness of `2-colouring`)**.** The program `2-colouring` is totally correct with respect to the following specifications:

Input: An *input graph.*
Output: The program fails if and only if the input is not 2-colourable. Otherwise, it outputs a 2-colouring of the input such that every node is either blue or red, and no pair of adjacent nodes share the same mark.

Proof. Termination follows from Lemma 1. Loop edges do not affect the 2-colouring and are omitted in the program. Let G be the input graph. If G is empty, `init` fails to match, and the program terminates, outputting the empty graph and satisfying the specifications. Suppose G consists of at least one node. We then split the remainder of this proof into two cases.

Case 1. G is 2-colourable. Since it is nonempty, `init` applies and `DFS!` is invoked. Consider G' and H such that $G \Rightarrow_{\texttt{init}} G' \Rightarrow_{\texttt{DFS!}} H$. It follows from Lemma 2 that H only contains red and blue nodes. Furthermore, as per Proposition 1, the rule `unroot` in `DFS` is never invoked, indicating that no violation of the 2-colouring has been encountered and the existence of a single root in H. Upon termination of `DFS!`, the procedure `Check` is called, and the rule `unroot` unroots the unique root node. The program then terminates and returns the 2-coloured host graph.

Case 2. G is not 2-colourable. Since it is nonempty, `init` applies and `DFS!` is invoked. Analogously to the previous argument, consider G' and H such that $G \Rightarrow_{\texttt{init}} G' \Rightarrow_{\texttt{DFS!}} H$. It follows from Proposition 1 that `unroot` is applied at the last execution of `FORWARD!`, breaking the `DFS!` loop. Therefore, there is no root node in H. The procedure `Check` is called, and since H contains no root, the rule `unroot` fails to match, and the `fail` command is invoked, failing the entire program. □

We now examine the complexity of `2-colouring`. Prior to doing so, we establish another invariant of the program in Invariant 3 so as to argue for the constant-time matching of rules in `COLOUR` ∪ `IGNORE`.

Invariant 3. Throughout the execution of `2-colouring`, there is at most one red edge in the host graph.

Proof. Upon inspection of the rules, `next_edge` is the only rule that creates a red edge in the host graph. Following its application, either a rule in the body {`colour_red, colour_blue, blue_red, red_blue`} applies and removes the unique red edge, or `(unroot; break)` is called, breaking the `FORWARD!` loop, making `back` inapplicable and, subsequently, terminating the `DFS!` loop. No rule involves red edges afterwards. □

Theorem 2 (Complexity of `2-colouring`)**.** On any class of input graphs, the program `2-colouring` terminates in time $\mathcal{O}(|V|+|E|)$, where $|V|$ is the number of nodes and $|E|$, the number of edges.

Proof. We first show that, for every rule, there is at most one matching attempt with respect to the complexity assumptions of the updated compiler (Figure 4).

The rule `init` matches a single node. If the graph is nonempty, `init` immediately applies on the first match as every node is grey. and is therefore constant. Otherwise, no node is considered, and the matching fails immediately. Therefore, there is at most one matching attempt for `init`. The rule `unroot` matches a single node. Given that there is a bounded number of roots in the graph (Invariant 2), the number of matching attempts is also bounded.

Since there is at most one root in the host graph, node `1` of `next_edge`, `blue_red`, `red_blue`, `colour_red`, `colour_blue` and node `2` of `back` match in constant time. The rule `next_edge` matches any unmarked edge incident to the root incident to an *any*-marked node. Since any node adjacent to the root is marked, and non-loop edges are stored in distinct lists with respect to their marks, there is at most one matching attempt for `next_edge`. An analogous argument can be made for all rules beside `init` and `unroot`, as it is known from Invariant 3 that there is at most one red edge in the host graph, and Invariant 2 establishes that a root node is incident to at most one dashed edge, hence limiting the number of possible matches to a single one. Therefore, there is at most one matching attempt for every rule in `2-colouring`. Now, let us look at the number of calls during the execution of the program for each rule. We define a call to be the invocation of a rule. For the purpose of this proof, let n and m be the number of nodes and edges, respectively.

The rule `init` is only called once at the beginning. It succeeds if the input graph is nonempty; otherwise, it fails. Let us show that `back` is called at most $m + 1$ times. The number of calls is the sum of successful applications and unsuccessful ones. Since the loop `DFS!` terminates at the inapplicability of `back`, there can be, at most, one unsuccessful application. Observe that the rule `back` marks an edge blue. It has been established in Invariant 1 that blue edges retain their mark. Hence, since there are m edges, there can be at most m successful applications.

Similarly, we demonstrate that the rules `blue_red`, `red_blue`, `colour_red` and `colour_blue` are called at most $n + m$ times each. Since the reasoning applies analogously to each of these rules, let `r` represent one of them. It can be observed that an unsuccessful application of `r` does not necessarily trigger the invocation of `(unroot; break)`, as the latter is called if and only if every single rule in `COLOUR` ∪ `IGNORE` fails. Let us then look at the number of successful applications first. The rules `red_blue` and `blue_red` mark an edge blue. As previously stated, blue edges retain their mark. Therefore, there can be at most m successful applications of `COLOUR`. The rules `colour_red` and `colour_blue` mark a grey node blue or red. Non-grey nodes retain their mark, thus implying that there can be at most $n - 1$ successful applications of `IGNORE` (there are n nodes and `init` has already turned one node non-grey, hence $n - 1$). Given that the failure of every rule in `COLOUR` ∪ `IGNORE` triggers the invocation of `(unroot; break)` and terminates both `FORWARD!` and `DFS!`, each rule can fail to apply at most as many times as some rule in `COLOUR` ∪ `IGNORE` succeeds and one more time, marking the terminations of the loops `FORWARD!` and `DFS!`. Hence, each rule application can be unsuccessful at most $(n - 1) + (m) + 1 = n + m$ times.

The rule `next_edge` marks an unmarked edge, implying that there can be at most m successful applications. An unsuccessful application of `next_edge` terminates the loop `FORWARD!` and invokes the rule `back`, which either succeeds or terminates the `DFS!` loop. Thus, there can be at most as many unsuccessful applications of `next_edge` as there are successful applications of `back`, that is, m. The number of calls of `next_edge` is bounded to $m + m = 2m$. Finally, it is easy to see that `unroot` is called at most twice. Once in `DFS!` (its call terminates the loop) and once in `Check`.

Figure 8 offers an overview of the maximum number of calls for each rule of the program. Therefore, taking into account that all rules are constant, the overall time complexity of the program `2-colouring` is

$$\begin{aligned} & 2 \cdot 1 + 2 \cdot 1 + 2 \cdot m + 4 \cdot (n + m) + 2 \cdot (n - 1) + 2 \cdot m + (m + 1) \\ = \;& 6n + 9m + 3 = \mathcal{O}(|V| + |E|). \end{aligned}$$

□

Rules	Unsuccessful	Successful
`init`	1	1
`unroot`	1	1
`next_edge`	m	m
`blue-red`	$n + m$	m
`red-blue`	$n + m$	m
`colour_red`	$n + m$	$n - 1$
`colour_blue`	$n + m$	$n - 1$
`back`	1	m

Fig. 8. Bounds on the number of rule calls for each rule throughout an entire execution. n is the number of nodes and m, edges.

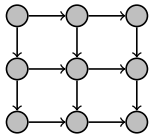

Fig. 9. Grid graph.

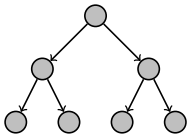

Fig. 10. Binary tree.

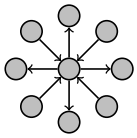

Fig. 11. Star graph.

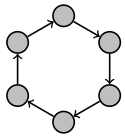

Fig. 12. Cycle graph.

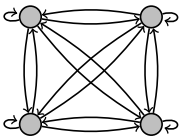

Fig. 13. Complete graph.

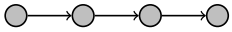

Fig. 14. Linked list.

Figure 15 showcases the empirical benchmarks of `2-colouring` on the graph classes of Figures 9, 10, 11, 12, 13 and 14. The measured runtimes do not account for graph parsing, building and printing, as these operations have a linear time

complexity with respect to the input size (Figure 4). Compilation time is also not included.

As evidenced, both unbounded-degree graph classes, complete graphs and binary trees, exhibit linear runtime performance in these tests. It is interesting to note that complete graphs exhibit almost constant runtime. Since any complete graph K_n with $n \geq 3$ is not 2-colourable, the GP 2 program can detect a violation early during execution. This consistent behaviour is primarily attributed to the deterministic nature of the compiler implementation. In theory, matches in GP 2 are nondeterministic, and it is conceivable that a GP 2 compiler strictly adhering to this nondeterminism would visit every node in an arbitrary complete graph before encountering such a violation.

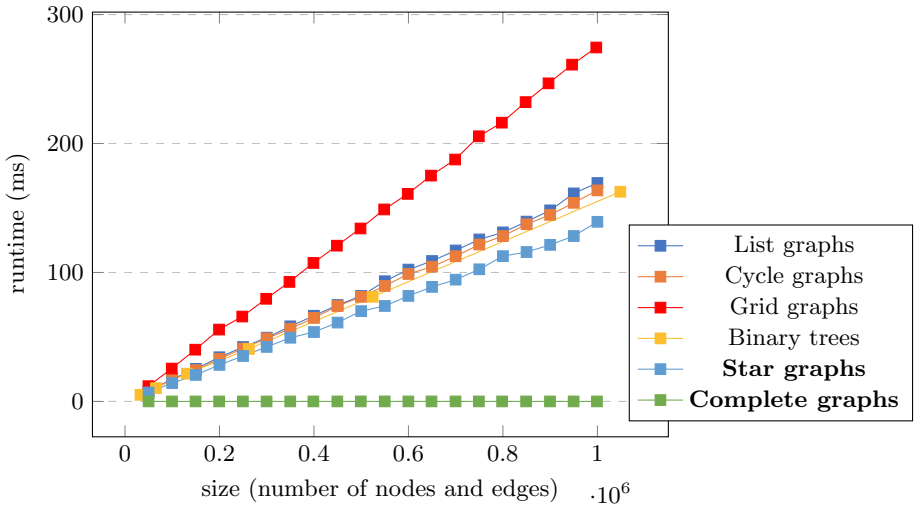

Fig. 15. Measured performance of the program `2-colouring`. Unbounded-degree graph classes are boldened in the legend.

5 Conclusion

We have presented an approach to implement conventional linear-time graph algorithms, such as 2-colouring and connectedness checking, with rule-based graph programs that have a linear runtime on input graphs with arbitrary node degrees. Removing the condition of bounded-degree input graphs has been an open problem since the publication of the first paper on rooted graph transformation [3].

So far, only certain reduction programs that destroy their input graphs could be designed to run in linear time on unbounded-degree graphs [5].

Our solution consists in both improving the graph data structure generated by the GP 2 compiler and devising a technique to exploit the new representation in programs. Previously, the graph data structure of the C program generated by the compiler stored lists of incoming and outgoing edges with each node. These lists were searched by the matching algorithm to quickly find an edge corresponding to an incoming or outgoing edge in the left-hand graph of a processed rule. However, in the presence of incident edges with different marks, these searches became linear-time operations that prevented constant-time rule matching. With the new data structure, finding an incident edge with a particular mark requires only constant time.

We exploit the new graph representation by designing a rule assigning a unique mark to an edge (`next_edge` in `2-colouring` resp. `is-connected`) and constructing other rules that perform actions on the uniquely marked edge and its linked nodes depending on the marks of the nodes.

In future work, we plan to overcome the remaining restriction that programs such as `2-colouring` require connected input graphs. By creating a separate node list for each node mark, it should be possible to find an unprocessed connected component of the host graph in constant time. For example, after 2-colouring a connected component, an uncoloured connected component could be found by searching for an arbitrary grey node.

We speculate that it will ultimately be possible to implement all DFS-based linear-time graph algorithms by linear-time GP 2 programs. Such algorithms include, for example, the non-destructive recognition of acyclic graphs, the topological sorting of acyclic graphs, the construction of Eulerian cycles, and the generation of strongly connected components.

References

1. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Software & Systems Modeling **5**(3), 261–288 (2006). https://doi.org/10.1007/s10270-006-0027-7
2. Bak, C.: GP 2: Efficient Implementation of a Graph Programming Language. Ph.D. thesis, Department of Computer Science, University of York, UK (2015), https://etheses.whiterose.ac.uk/12586/
3. Bak, C., Plump, D.: Rooted graph programs. In: Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012). Electronic Communications of the EASST, vol. 54 (2012). https://doi.org/10.14279/tuj.eceasst.54.780
4. Bak, C., Plump, D.: Compiling graph programs to C. In: Proc. 9th International Conference on Graph Transformation (ICGT 2016). Lecture Notes in Computer Science, vol. 9761, pp. 102–117. Springer (2016). https://doi.org/10.1007/978-3-319-40530-8_7
5. Campbell, G., Courtehoute, B., Plump, D.: Fast rule-based graph programs. Science of Computer Programming **214**, 102727 (2022)

6. Campbell, G., Romö, J., Plump, D.: The improved GP 2 compiler. Tech. rep., Department of Computer Science, University of York, UK (2020), https://arxiv.org/abs/2010.03993
7. Dörr, H.: Efficient Graph Rewriting and its Implementation, Lecture Notes in Computer Science, vol. 922. Springer (1995). https://doi.org/10.1007/BFb0031909
8. Fernández, M., Kirchner, H., Pinaud, B.: Strategic port graph rewriting: an interactive modelling framework. Mathematical Structures in Computer Science **29**(5), 615–662 (2019). https://doi.org/10.1017/S0960129518000270
9. Ghamarian, A., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. International Journal on Software Tools for Technology Transfer **14**(1), 15–40 (2012). https://doi.org/10.1007/s10009-011-0186-x
10. Jakumeit, E., Buchwald, S., Kroll, M.: GrGen.NET – the expressive, convenient and fast graph rewrite system. International Journal on Software Tools for Technology Transfer **12**(3–4), 263–271 (2010). https://doi.org/10.1007/s10009-010-0148-8
11. Skiena, S.S.: The Algorithm Design Manual. Springer, third edn. (2020). https://doi.org/10.1007/978-3-030-54256-6
12. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for EMF model transformation development. In: Proc. 10th International Conference on Graph Transformation (ICGT 2017). Lecture Notes in Computer Science, vol. 10373, pp. 196–208. Springer (2017). https://doi.org/10.1007/978-3-319-61470-0_12