Linear-Time Graph Programs for Unrestricted Graphs

Ziad Ismaili Alaoui and Detlef Plump
Department of Computer Science, University of York
York, United Kingdom
{z.ismaili-alaoui, detlef.plump}@york.ac.uk

Achieving the complexity of graph algorithms in conventional languages using graph transformation rules is challenging due to the cost of graph matching. Previous work demonstrated that with *rooted* rules, certain algorithms could be executed in linear time using the graph programming language GP 2. However, for non-destructive algorithms that retain the structure of input graphs, achieving linear runtime required the input graphs to be connected. In this paper, we overcome this restriction by enhancing the graph data structure generated by the GP 2 compiler and exploiting this new structure in programs. As case studies, we present a cycle detection program, a breadth-first search program (previously achievable only in quadratic time), and a program for numbering connected components, all running in linear time on all classes of input graphs, both connected and disconnected, with arbitrary node degrees. We empirically substantiate the linear time complexity with timings for various classes of input graphs.

1 Introduction

Designing and implementing languages for rule-based graph rewriting, such as GReAT [1], GROOVE [8], GrGen.Net [9], Henshin [12], and PORGY [7], poses significant performance challenges. Typically, programs written in these languages do not achieve the same runtime efficiency as those written in conventional imperative languages like Java. The primary obstacle is the cost of graph matching, where matching the left-hand graph L of a rule within a host graph G generally requires time $|G|^{|L|}$, with $|X|$ denoting the size of graph X. (Since L is fixed, this is a polynomial.) Therefore, linear-time imperative graph algorithms may exhibit polynomial runtimes when modelled as rule-based graph programs.

To address this issue, the graph programming language GP 2 [11] supports *rooted* graph transformation rules, initially proposed by Dörr [6]. This approach involves designating certain nodes as *roots* and matching them with roots in the host graphs. Consequently, only the neighbourhoods of host graph roots need to be searched for matches, which can often be done in constant time under specific (and usually mild) conditions. The GP 2 compiler [2] maintains a list of pointers to roots in the host graph, facilitating constant-time access to roots if their number remains bounded throughout the program's execution. In [3], *fast* rules were identified as a class of rooted rules that can be applied in constant time, provided 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, 2], 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, particularly related to the runtime graph data structure used by the compiled programs [5]. 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 [4] for an overview.

Submitted to:
Fifteenth International Workshop on Graph
Computation Models (GCM 2024)

© Z. Ismaili Alaoui & D. Plump
This work is licensed under the
Creative Commons Attribution License.

Despite this progress, programs that retain the structure of input graphs, such as the aforementioned 2-colouring program, have until now required non-linear runtimes on disconnected graphs. The problem is that, after a connected component is visited, the number of failed attempts to match a non-visited node in a different connected component may increase. Consequently, in disconnected graph classes, this number may grow quadratically in the graph size, leading to a quadratic program runtime. In connected graph classes, this undesirable behaviour is ruled out because all nodes are reachable from a single undirected depth-first search.

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 so that nodes are stored in separate linked lists based on their *marks* (red, green, blue, dashed, or unmarked). This allows the matching algorithm to find a node of a specific mark in constant time. For example, if a red-marked node is required, a single access to the list of red-marked nodes will either find such a node 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 first demonstrate in a case study of the acyclicity problem how the new graph representation allows for a linear runtime. Our program expects connected or disconnected input graphs and either detects that a graph admits of a cycle or returns a graph isomorphic to the input up to node and edge marks. Then, we present two other programs that exploit this structure: one numbering the connected components of a graph, and another carrying out a breadth-first search of a graph. For each program, we offer empirical benchmarks to substantiate our claims regarding their time complexities.

2 Compiler Enhancements

We refer to [4] for a description of the GP 2 programming language. In this section, we introduce the modifications to the graph data structure generated by the GP 2 compiler. We first state the motivation behind the changes.

2.1 Motivation

Consider the program is-discrete from Figure 1.[1] The program fails if and only if the input graph is discrete, that is, it contains no edges. Here, we assume that the input graph is unmarked (nodes and edges are not coloured, and edges are not dashed). The program is non destructive in that the output graph is isomorphic to the input graph up to node marks, should it not fail. `is-discrete` is composed of a looping procedure followed by a test. The rule `mark` in the looping procedure marks and roots an arbitrary unmarked node in the host graph, and `isolated` checks whether the degree of the node rooted by `mark` is 0. Notice that the right-hand side node of the rule `isolated` is not in the interface; hence, for it to match, it must satisfy the dangling condition (i.e. have no edge outside the rule incident with it). If `isolated` does not apply after an application of `mark`, a node whose degree is non-zero was found and the looping procedure breaks. The rule `root` checks if a root exists in the host graph, which can happen if and only if `isolated` failed to apply.

The GP 2 compiler, prior to this article, matched the rule `mark` with a complexity of $\mathcal{O}(n)$, with n being the number of nodes in the host graph at the application of `mark`. Indeed, in order to find an

[1] Node labels such as x are written inside nodes, whereas small integers below nodes are their identifiers. Nodes without identifiers on the left-hand side are to be deleted; nodes without identifiers on the right-hand side are to be added. Nodes with the same identifier on each side are to be kept.

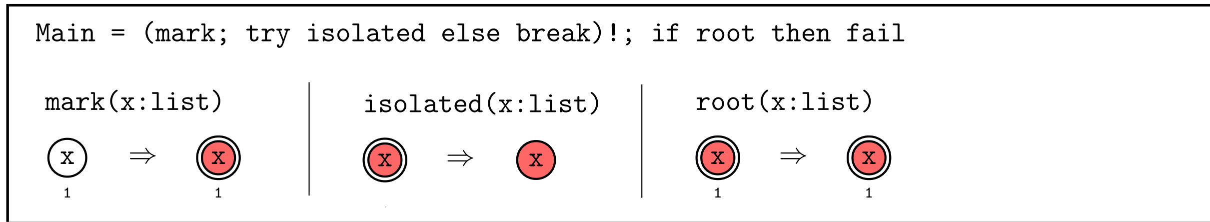

Figure 1: The non-destructive program `is-discrete`.

unmarked node, the compiler had to iterate through a unique linked list consisting of all nodes in the host graph. As a result, since the looping procedure can be invoked at most n times, the overall complexity of `is-concrete` was $\mathcal{O}(n^2)$, as empirically evidenced in Figure 2.

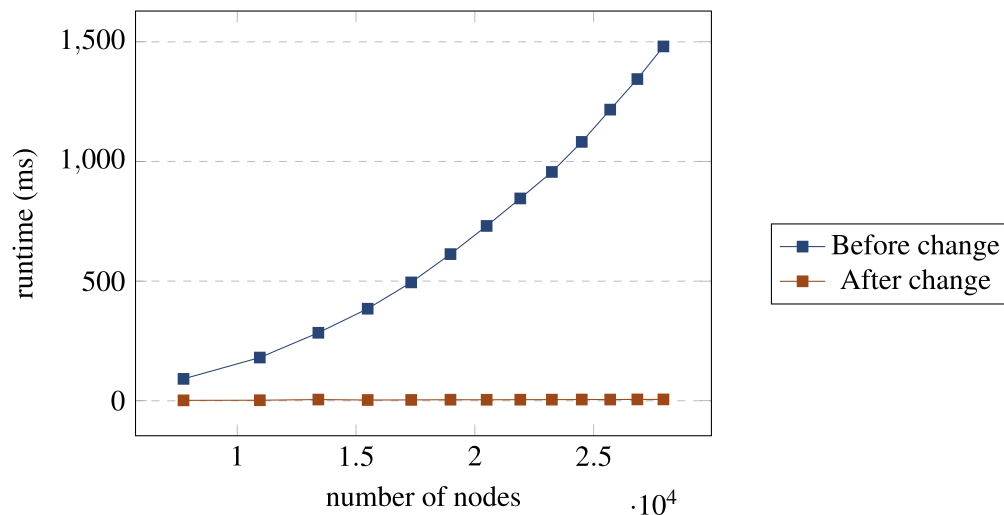

Figure 2: Measured performance of the program `is-discrete` on discrete graphs under the previous and the new compilers.

2.2 New Graph Data Structure

To address the problem described in Subsection 2.1, we modified the internal graph data structure the GP 2 compiler generates. We employ the terms *unmodified compiler* and *modified compiler* to refer to the versions prior to and after this paper, respectively.

The unmodified compiler stored the host's graph data structure as one linked list containing of every node in the graph, with each node storing a two-dimensional array where each cell stored a linked list consisting of incident edges of a particular mark and orientation relative to the node. This model allowed for constant access to edges of relevant orientations and marks, in contrast to an earlier model which stored, for each node, all edges incident with it in two linked lists: one for incoming edges and one for outgoing edges [5]. One major problem with the structure used to store nodes in the unmodified compiler is the unnecessary lookups the matching algorithm carried out to find a match for a node in the host graph, especially when there existed other nodes whose marks could not match. For instance, in order to match node `1` of the rule `mark` in `is-discrete` (Figure 1), the matching algorithm would

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(m)`	**Fetch the first node of mark `m` in the host graph.**	$O(1)$
`nextHostNode(m)`	**Given a node of mark `m`, fetch the next node of mark `m` 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.	$O(n)$

Figure 3: Updated runtime complexity assumptions. Modified procedures are highlighted in grey. n is the size of the input.

have to iterate through a single linked list consisting of every node in the host graph, including those of incompatible marks (such as red-marked nodes). Hence, in the case where every node in the host graph were red-marked beside an unmarked one, the matching algorithm could, in the worst case, look up (i.e. attempt matching with) every red-marked node in the host graph until it finally reaches the unmarked node. It is clear that under such a model, matching `mark` is linear, provided the number of non-red-marked nodes in the class under which the host graph falls is not bound to a constant.

A solution presented in this paper, supported by the modified compiler, is the storage of nodes in several linked lists, each consisting of nodes of a particular, distinct mark (unmarked nodes are stored in a distinct linked list). Each linked list is stored in a unique cell of a global one-dimensional array, allowing for constant access to the first element of each linked list. This model permits the rule `mark` to match in constant time, since the matching algorithm need not iterate through nodes of incompatible marks. Indeed, the matching algorithm directly inspects the linked list consisting of all red-marked nodes in the host graph. Since the rule `mark` allows for any red-marked node of arbitrary label to be matched, the first element found in the linked list by the matching algorithm matches successfully. As a result of these changes, the time compelexity of the program `is-discrete` under the modified compiler is reduced down to $\mathcal{O}(n)$. Figure 2 highlights the difference in runtimes of the program `is-discrete` run under both compilers.

To reason about programs, it is crucial 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 3 showcases the complexity assumptions of the basic procedures of the search plan, adapted from [4]. The grey rows indicate existing procedures updated by the changes introduced in this paper. The empirical evidence collated from various case stud-

ies showcased in this paper corroborates the complexity assumptions implied by the changes brought to the modified compiler.

3 Case Study: Recognising Acyclicity

Cycle detection is a foundational problem in graph algorithms. Several GP 2 programs were previously proposed to decide whether a given directed graph is acyclic (i.e. contains no directed cycle), however, they were either restricted to binary DAGs (directed acyclic graphs wherein each node is incident to at most two outgoing edges) or destructive, in that they partially or totally deleted the input graph [2, 4].

3.1 Program

The program `is-dag` from Figure 4 recognises acylic graphs with respect to the following specifications.

Input: An arbitrarily-labelled GP 2 host graph such that:

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

Output: A host graph isomorphic to the input graph up to marks if the input graph is acyclic, otherwise the program fails.

It is claimed that `is-dag` is non-destructive; however, a cyclic input graph results in the invocation of the `fail` command, returning no output graph. Nonetheless, the program can easily be turned completely non-destructive by replacing the `fail` command in the procedure `Check` by a rule creating a distinct flag (such as an unmarked node), signalling the existence of a cycle. Figure 5 illustrates a sample execution of the program on a cyclic input graph.

The program `is-dag` implements a directed depth-first search of the host graph by marking visited nodes in two distinct colours: red and blue. A red mark on a node indicates that it is being visited by the depth-first search. Visualising the DFS (depth-first search) as modelled with a stack, nodes inside the stack are marked red. A blue mark on a node indicates that it was previously in the stack and was popped from it, meaning that it no longer requires further visitation. The head node of the stack is rooted (an invariant of the program is that there is at most one root in the host graph throughout the execution of `is-dag`). A cycle exists if and only if, during the DFS, an edge from the rooted node to a red-marked node is found. Indeed, let u be the root and v, a red-marked node adjacent to u via an edge from u to v. Since v is marked red, it implies that it is being visited (therefore part of the stack) and that there exists a directed path from v to u. Since there exists an edge from u to v, we obtain a cycle.

The program first calls a looping procedure in `Main`: `(init; DFS!; try unroot else break)!`. The rule `init` acts as an initialiser: it selects an arbitrary node to start a directed DFS from. Then, the looping procedure `DFS!` is invoked. The purpose of the procedure `DFS` is to move the root in a DFS fashion throughout the host graph. The procedure is structured around a `try-else` construct. If first tries to determine if there is any unprocessed edge (i.e. unmarked edge) incident with the root by invoking the rule `next_edge` (the magenta colour in the visual representations of the programs in this paper denote the wildcard mark `any`). Should one be found, the rule marks the edge red so that it can be uniquely processed by the rest of the procedure. If none is found, the root can no longer move forward

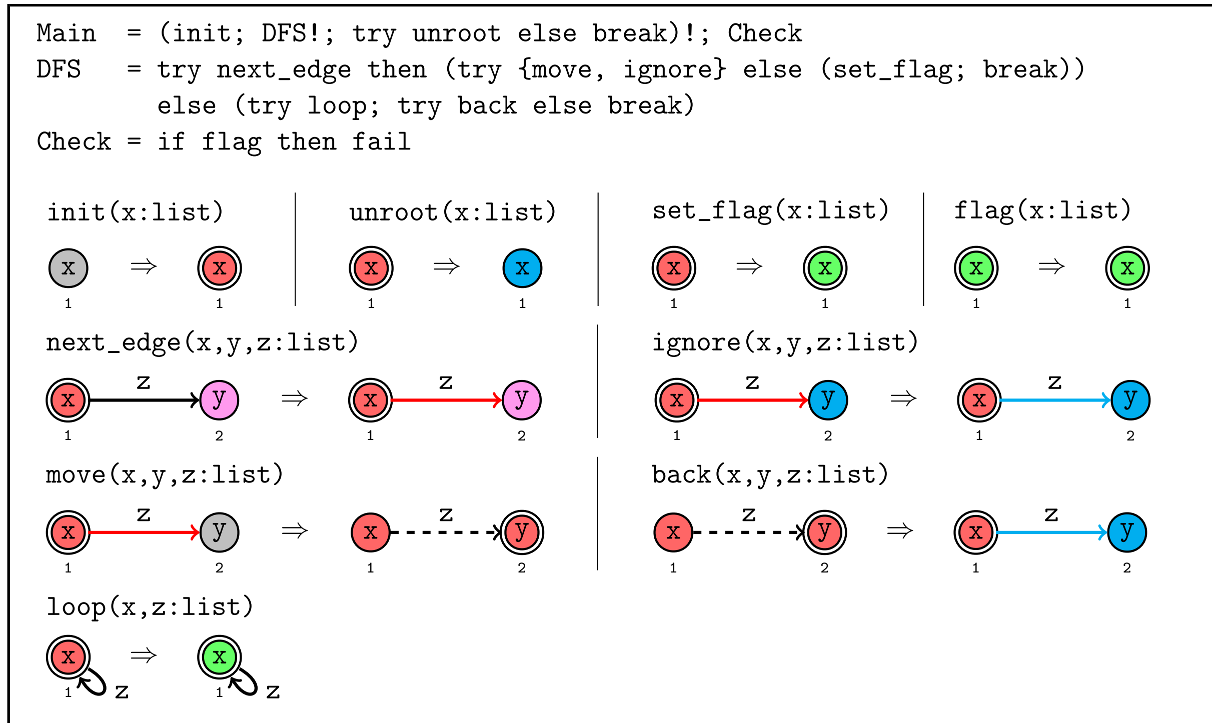

Figure 4: The program `is-dag`.

and the `else`-statement (on the next line) is invoked instead. Back to `next_edge`; after a successful application of the rule, there are three possible scenarios: the root is adjacent to (1) a grey-marked node, (2) a blue-marked node, or (3) a red-marked node. If the root is adjacent to a grey-marked node, the rule `move` moves the root forward to that node, marks it red and dashes the traversed edge. Dashed edges represent the ongoing path followed by the directed DFS. If the root is adjacent to a blue-marked node, the edge is ignored by the rule `ignore` marking it blue, so that it does not match with `next_edge` any further. If the root is adjacent to a red-marked node, no rule in the ruleset `{move, ignore}` applies, hence invoking the rule `set_flag`, which marks the rooted node green, indicating the existence of a cycle. Prior to applying `next_edge`, if no unmarked edge is incident with the root, the latter rule fails, and `(try loop; try back else break)` is invoked. The rule `loop` is first tried to ensure that the root does not admit of a loop. If a loop is found, the rule turns the rooted node green, akin to `set_flag`. The rule `back` implements the *pop* operation by backtracking the root via an incoming dashed edge. If no incoming dashed edge is found, it implies that the rooted node is the only element of the stack; hence, the `break` command is called, terminating the looping procedure `DFS!`.

Following the termination of `DFS!`, the rule `unroot` is tried in `Main` to ensure that no flag was set during the execution of `DFS` and to unroot the sole red node of the stack. The only rules remarking (i.e. changing the marks of) rooted nodes in `DFS` are `set_flag` and `loop`, which are applied if and only if a cycle was found. Therefore, a failure to apply `unroot` entails the existence of a cycle. Given that input graphs can be disconnected and the DFS is directed, all nodes are not necessarily visited after an initial DFS from an arbitrary node. Hence, the rule `init` is called again following a successful application of `unroot`. The looping procedure terminates when `init` no longer applies, indicating the visitation of all nodes, or when `unroot` fails, indicating the discovery of a cycle.

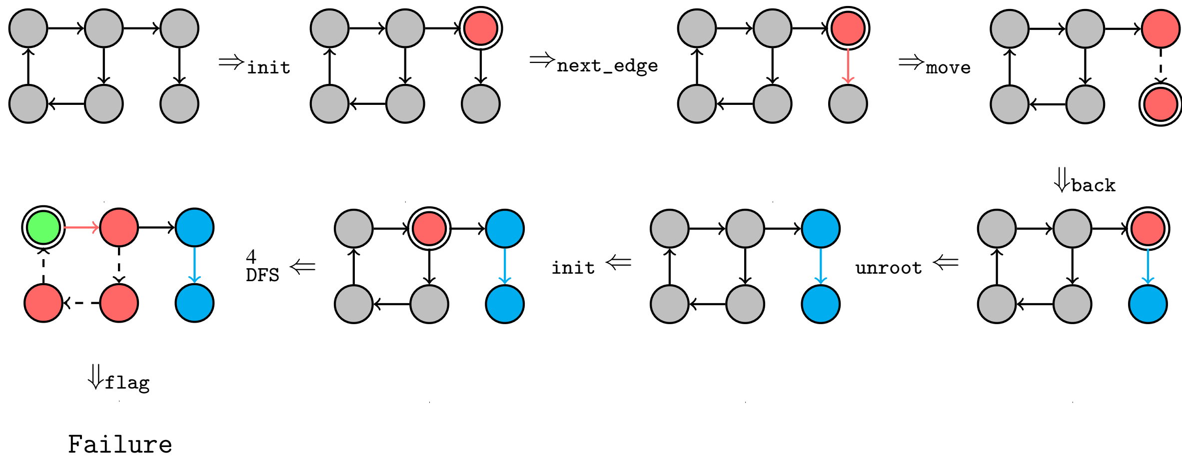

Figure 5: Sample execution of `is-dag` on a cyclic graph.

Finally, the procedure `Check` is invoked, checking whether a flag exists in the host graph. Should one exist, a cycle was found and the command `fail` is called, failing the program. Otherwise, the program terminates and returns the host graph, isomorphic to the input graph up to marks.

3.2 Time Complexity

All rules of the program `is-dag` apply in constant time under the complexity assumptions of the modified GP 2 compiler (Figure 3). The rule `init` applies in constant time since any arbitrarily labelled grey-marked node is a match for the rule. As the generated graph data structure keeps a node linked list separate to each mark, the matching algorithm can select a grey-marked node in constant time regardless of the number of non-grey-marked nodes in the host graph at the time of the rule application.

It can be observed that a node and an edge never repeat their marks. Since all rules, except `flag` called at most once in `Check`, remark at least one element, the overall program runtime is linear in the size of the graph, i.e. $\mathcal{O}(n+m)$ with n and m being the number of nodes and edges in the input graph, respectively. To support that claim, we have collected empirical timing results, showcased in Figure 13, for various graph classes of bounded (Figures 6, 7, 9, 10 and 12) and unbounded degree (Figure 12).

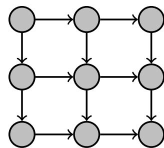

Figure 6: Grid graph.

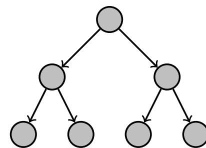

Figure 7: Binary tree.

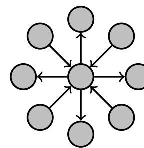

Figure 8: Star graph.

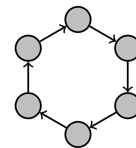

Figure 9: Cycle graph.

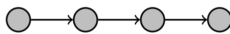

Figure 10: Linked list.

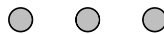

Figure 11: Discrete graph.

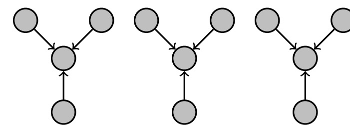

Figure 12: k k-star graphs.

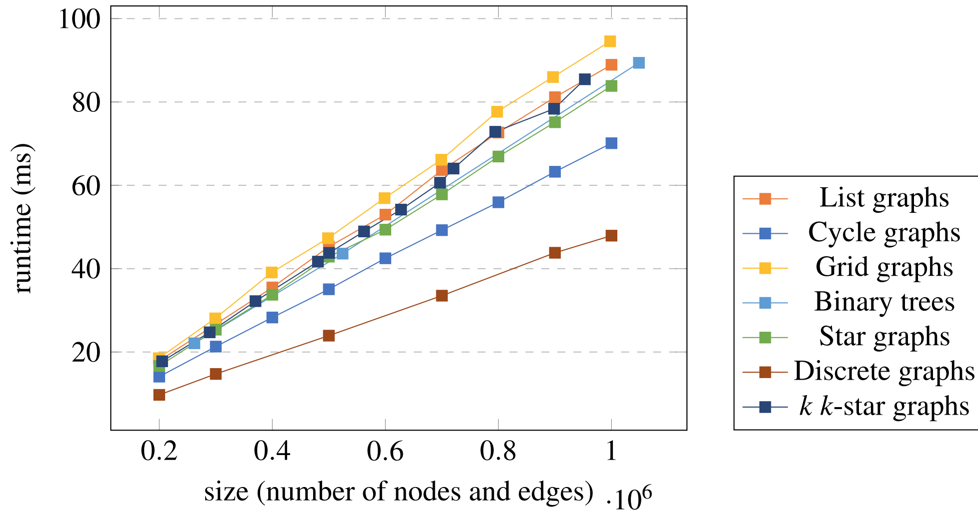

Figure 13: Measured performance of the program `is-dag` under the modified compiler.

4 Case Study: Numbering Connected Components

A clear advantage of the new data structure is the ability to match an arbitrarily-labelled node of a particular mark (or determine that none exists) in constant time. A natural choice of a program that can be constructed under that paradigm is one that numbers all connected components of an input graph.

The program `component-numbering` from Figure 14 appends a number to the list of each node of an input graph unique to the connected component the node belongs to with respect to the following specifications.

Input: An arbitrarily-labelled GP 2 host graph such that:

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

Output: A host graph structually isomorphic to the input graph where a number is appended to the list of each node, denoting the unique identifier of the connected component it belongs to.

4.1 Program

The program `component-numbering` works by first evoking the rule `init`, which marks an arbitrarily-labelled node grey, roots it and appends `1` (first component identifier) to its list label. If the rule fails to match, the graph is empty and the program terminates, given that `unroot` fails. The procedure `DFS` works analogously to that of `is-dag`, except it is undirected. The rule `move` propagates the identifier.

The first looping body `DFS!` propagates the numbering `1` in a single connected component of the graph. The next looping procedure repeats the process, except it invokes `next` instead of `init`. The rule `next` simply unroots the current rooted node, roots another unvisited (grey-marked) node and appends the identifier of the previous rooted node, incremented by 1. Once all unvisited nodes are exhausted, the rule `next` fails to match and the `break` command is called. The rule `unroot` unroots the sole root of the graph.

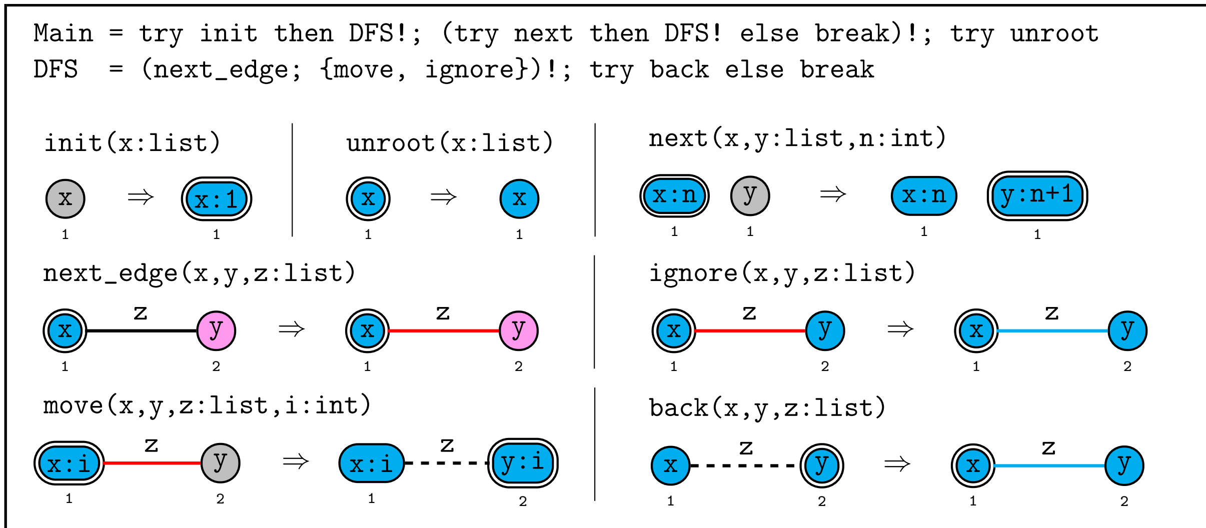

Figure 14: The program `component-numbering`.

4.2 Time Complexity

The time complexity of the program `component-numbering` is linear. That is largely attributed to the fact that `init` and `next` match in constant time, which would have not been possible under the unmodified compiler. Figure 15 offers corroborative evidence. As expected, the program exhibits a linear runtime on discrete graphs, which are disconnected and require $n-1$ calls of the rule `next`, with n being the number of nodes.

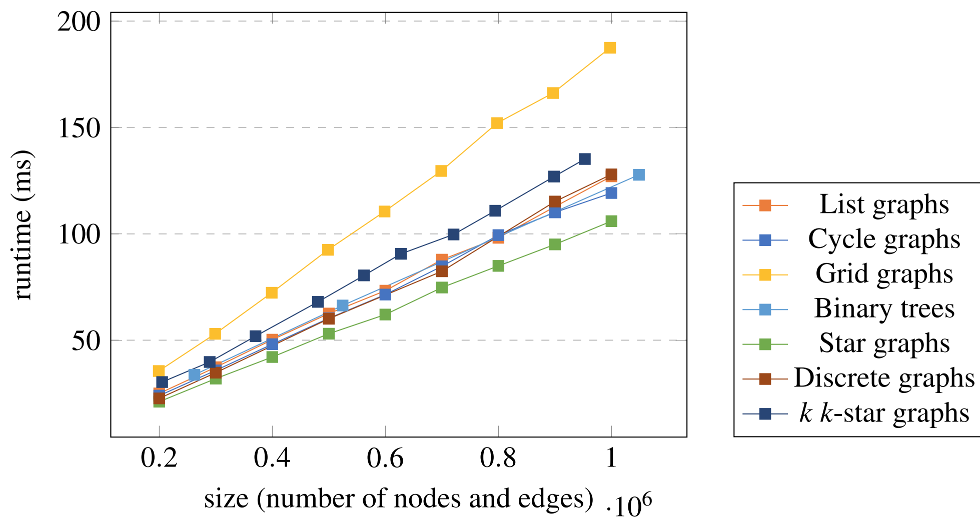

Figure 15: Measured performance of the program `component-numbering` under the modified compiler.

5 Case Study: Breadth First Search

The problem of traversing a graph in a breadth-first search (BFS) fashion in GP 2 is interesting, as previous techniques used to traverse graphs non-destructively in linear time involved variations of a depth-first search. Bak proposed, in his PhD thesis [2], an implementation of the BFS algorithm in GP 2. However, that program ran in quadratic time. That is due to there being no way, prior to the compiler enhancement of this paper, to find a node to expand from in constant time, given that the next node to expand from is not necessarily adjacent to the one being expanded during a BFS.

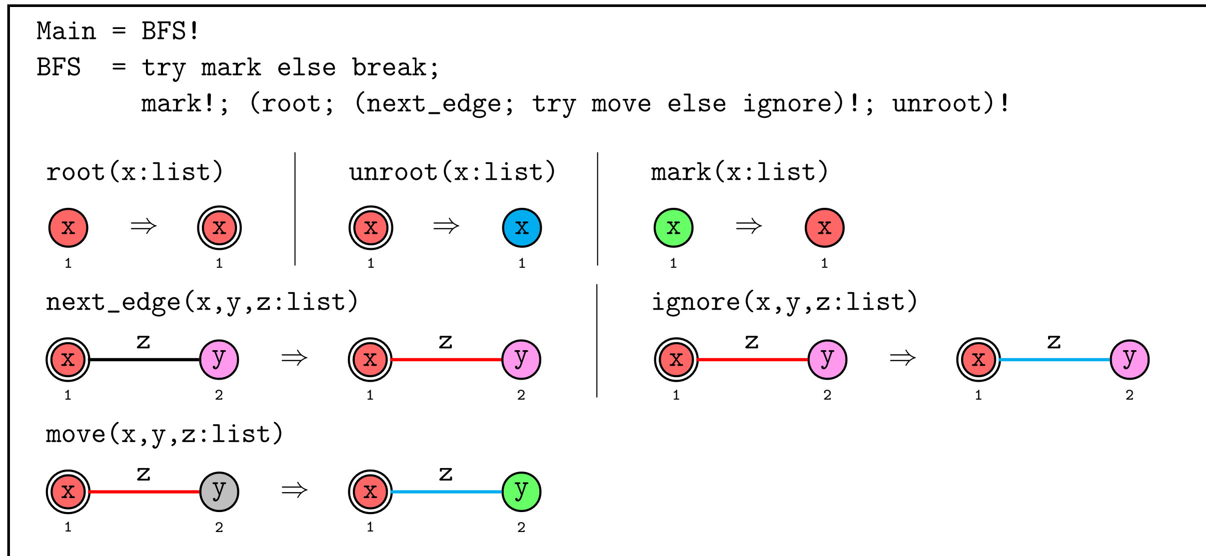

Figure 16: The program bfs.

In this section, we present the program bfs, capable of carrying out a breadth-first search of a graph in linear time with respect to its size and the following specifications.

Input: A non-empty arbitrarily-labelled GP 2 host graph such that:

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

Output: A host graph structually isomorphic to the input graph where all nodes and edges are blue-marked.

5.1 Program

The program exploits the advantages of the compiler modifications so as to find the next node to expand from in constant time. The program bfs consists of a looping procedure, BFS!, which marks all green nodes in the host graph red and, subsequently, marks all grey nodes directedly adjacent to the red nodes green. Once a red node has been expanded from, it is marked blue.

Intuitively, at the beginning of the execution of BFS, the program seeks to mark all nodes directedly adjacent to green-marked. Firstly, the rule mark is applied as long as possible to mark all green-marked nodes red. Then, for as long as possible, the program picks some red node (which was previously green)

with the ryke `root`, and expands from it as long as possible. The nested looping procedure ends when `next_edge` is no longer applicable, implying that there is no expansion left from the node picked. The rule `unroot` then marks the chosen node blue, indicating that it was fully processed, and moves on to the next red-marked node. The upper parenthetical looping procedure within `BFS!` terminates when `root` no longer applies, indicating that all nodes that were previously green at the beginning of `BFS` have been processed.

5.2 Time Complexity

The program `bfs` runs in linear time with respect to the size of the graph (i.e. the number of nodes and edges). Discrete graphs exhibit a seemingly constant time complexity since they consist of connected components of at most one node.

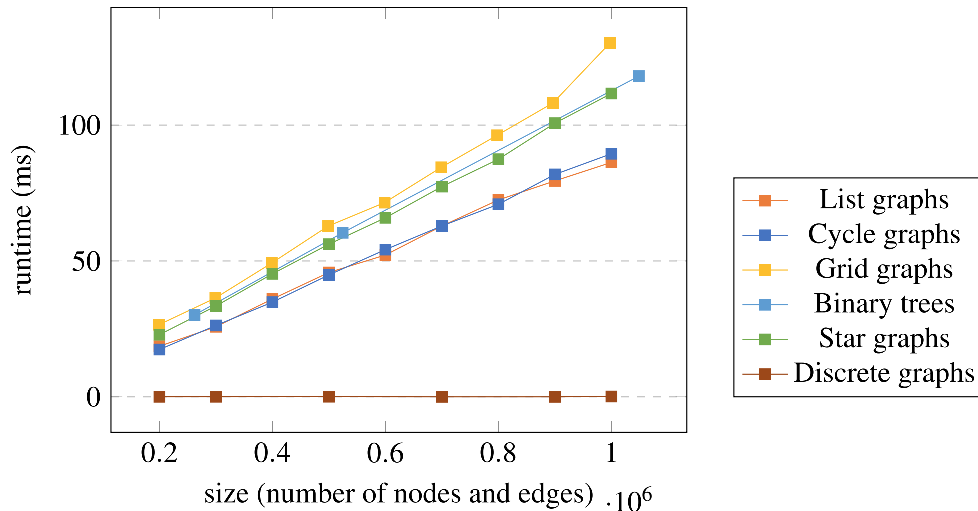

Figure 17: Measured performance of the program `bfs` under the modified compiler.

6 Conclusion

We have demonstrated an approach to implement various depth-first search-based algorithms for disconnected graph classes using a rule-based graph program that achieves linear runtime on input graphs with arbitrary node degrees. Addressing the challenge of disconnected input graphs has been an open problem since the publication of the first paper on rooted graph transformation [3], as well as a limiting issue for colouring programs in [4], which required input graphs to be connected. Until now, only certain reduction programs that destroy their input graphs could be designed to run in linear time on unbounded-degree and disconnected graphs [4].

Our approach involves both enhancing the graph data structure generated by the GP 2 compiler and developing a technique to leverage this new representation in programs. Previously, the graph data structure in the C program generated by the compiler stored all nodes in the host graph in a single linked list. However, when nodes had different marks, searches within this list became linear-time operations, hindering constant-time rule matching. The new data structure allows for finding a node with a particular mark in constant time.

We speculate that it will eventually be possible to implement virtually all graph- and tree-based data structures with their conventional time complexities as GP 2 programs. This includes structures such as heap data structures and AVL trees. Such implementations would enable faster GP 2 versions of conventional DFS-based algorithms from the literature that utilise advanced data structures, such as Dijkstra's algorithm.

References

[1] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi & Attila Vizhanyo (2006): *The design of a language for model transformations*. *Software & Systems Modeling* 5(3), pp. 261–288, doi:10.1007/s10270-006-0027-7.

[2] Christopher Bak (2015): *GP 2: efficient implementation of a graph programming language*. Ph.D. thesis, University of York.

[3] Christopher Bak & Detlef Plump (2012): *Rooted graph programs*. In: *Proc. International Workshop on Graph Based Tools (GraBaTs 2012)*, 54.

[4] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast rule-based graph programs*. *Science of Computer Programming* 214, p. 102727.

[5] Graham Campbell, Jack Romo & Detlef Plump (2020): *The improved GP 2 compiler*. *arXiv preprint arXiv:2010.03993*.

[6] Heiko Dörr (1995): *Efficient Graph Rewriting and its implementation*. Springer.

[7] Maribel Fernández, Hélène Kirchner & Bruno Pinaud (2019): *Strategic port graph rewriting: an interactive modelling framework*. *Mathematical Structures in Computer Science* 29(5), pp. 615–662, doi:10.1017/S0960129518000270.

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

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

[10] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Arend Rensink, Louis Rose, Sebastian Wätzoldt, Markus Lepper & Steffen Mazanek (2014): *A survey and comparison of transformation tools based on the transformation tool contest*. *Science of Computer Programming* 85, pp. 41–99, doi:10.1016/j.scico.2013.10.009.

[11] Detlef Plump (2012): *The design of GP 2*. *arXiv preprint arXiv:1204.5541*.

[12] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf & Matthias Tichy (2017): *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* 10373, Springer, pp. 196–208, doi:10.1007/978-3-319-61470-0_12.