Proceedings Paper:
Ismaili Alaoui, Ziad and Plump, Detlef (2025) Linear-Time Graph Programs without Preconditions. In: Proceedings 14th and 15th International Workshop on Graph Computation Models (GCM 2023 and GCM 2024). Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 39-54.

https://doi.org/10.4204/EPTCS.417.3

Linear-Time Graph Programs without Preconditions

Ziad Ismaili Alaoui *
Department of Computer Science, University of Liverpool
Liverpool, United Kingdom
ziad.ismaili-alaoui@liverpool.ac.uk

Detlef Plump
Department of Computer Science, University of York
York, United Kingdom
detlef.plump@york.ac.uk

We report on a recent breakthrough in rule-based graph programming, which allows us to reach the time complexity of imperative linear-time algorithms. In general, 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 can 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 input graphs to be connected and of bounded node degree. In this paper, we overcome these preconditions by enhancing the graph data structure generated by the GP 2 compiler and exploiting the new structure in programs. We present three case studies, a cycle detection program, a program for numbering the connected components of a graph, and a breadth-first search program. Each of these programs runs in linear time on both connected and disconnected input graphs with arbitrary node degrees. We give empirical evidence for the linear time complexity by using timings for various classes of input graphs.

1 Introduction

Designing and implementing languages for rule-based graph rewriting, such as GReAT [1], GROOVE [10], GrGen.Net [11], Henshin [15], and PORGY [9], poses significant performance challenges. Typically, programs written in these languages do not achieve the same runtime efficiency as those written in conventional imperative languages such as C or 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.) As a consequence, standard imperative graph algorithms running in linear time (see, for example, [7, 14]) may exhibit non-linear, polynomial runtimes when recast as rule-based graph programs.

To address this issue, the graph programming language GP 2 [13] supports *rooted* graph transformation rules, initially proposed by Dörr [8]. 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 mild conditions. The GP 2 compiler [3] 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 [4], *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 was 2-colouring. In [4, 3], it is shown that this program colours connected graphs of bounded node degree in linear time. Since then, the GP 2 compiler has received some major improvements, particularly related to the runtime graph data

*This author's work was done while he was affiliated with the University of York.

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 39–54, doi:10.4204/EPTCS.417.3

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 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 two updates to the GP 2 compiler, one being introduced in [2], which allow lifting the preconditions that host graphs must be connected and have a bounded node degree. In short, the solution is to improve the graph data structure generated by the compiler. Nodes are now stored in separate linked lists based on their *marks* (red, green, blue, grey or unmarked), and each node comes with a two-dimensional array of linked lists storing all incident edges based on their marks (red, green, blue, dashed or unmarked) and orientation (incoming, outgoing or looping). This enables the matching algorithm to find in constant time a node with a specific mark or an edge with a specific mark and orientation. For instance, if a red node is needed, a single access to the list of red nodes will either locate such a node or confirm its absence. Similarly, if an outgoing green edge is required, a single access to the corresponding linked list will either find such an edge or determine that none exists.

In addition to the new graph data structure, a programming technique is needed to take advantage of the improved storage. In a case study, we demonstrate how to recognize acyclic graphs in linear time with the new graph representation. Our program admits both connected and disconnected input graphs with arbitrary node degrees. It either detects that a graph is cyclic or returns an acyclic graph that is isomorphic to the input graph up to marking. Then, we discuss two more programs relying on the improved compiler: one numbering the connected components of a graph, and another performing a breadth-first search. For both programs, we give empirical evidence that they run in linear time on different classes of input graphs.

2 The Problem with Graph Classes of Unbounded Degree

The current section and the next explain the problem caused by classes of input graphs with an unbounded node degree, and how the updated compiler overcomes this problem. We include this material from [2] for the benefit of the reader. For a description of the GP 2 programming language, we refer to [5].

Previously, non-destructive GP 2 programs based on depth-first-search ran in linear time on graph classes of bounded node degree but in non-linear time on graph classes of unbounded degree [5]. For example, consider the program `is-connected` in Figure 1 which checks whether a graph is connected.[1] Input graphs are arbitrary GP 2 host graphs with grey nodes and unmarked edges. The program fails on a graph if and only if the graph is disconnected.

Rule `init` picks an arbitrary grey node as a root (if the input graph is non-empty) and then the loop `DFS!` performs a depth-first search of the connected component of the node chosen by `init`. The rule `forward` marks each newly visited node blue, and `back` unmarks it once it is processed. Procedure `DFS`

[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.

ends when `back` fails to match, indicating that the search is complete. Rule `match` checks whether a grey-marked node still exists in the graph following the execution of `DFS!`. This is the case if and only if the input graph contains more than one connected component. In this situation the program invokes the command `fail`, otherwise it terminates by returning the graph resulting from the depth-first search.

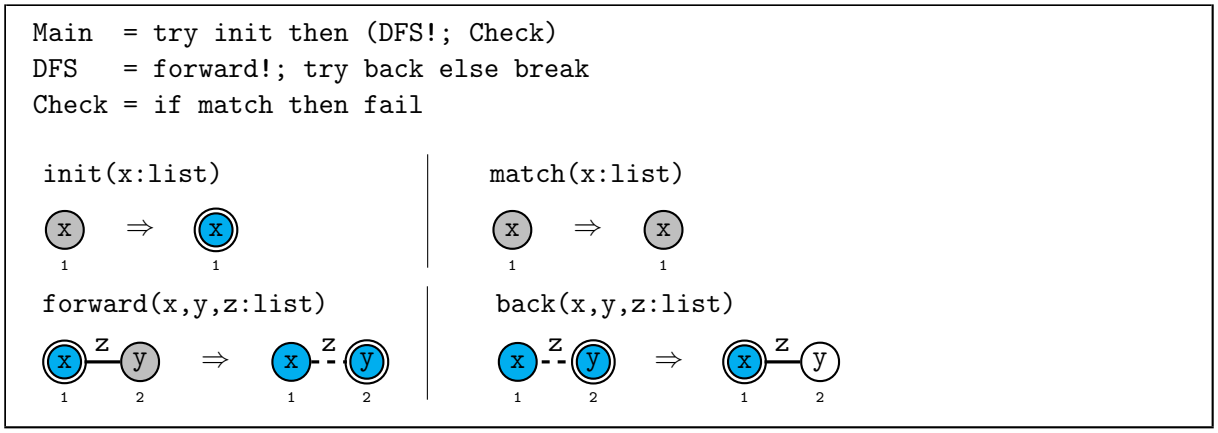

Figure 1: The old program `is-connected`.

It can be shown that the program `is-connected` runs in linear time on classes of graphs with bounded node degree [5]. 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 13). 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 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 $2|E| + \sum_{i=1}^{|E|} i = \mathrm{O}(|E|^2)$ where E is the set of edges.

3 First Compiler Enhancement

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*[2]. The 2020 compiler stored the host graph's structure as one linked list containing every node in the graph, with each node storing two linked lists of edges: 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 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.

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

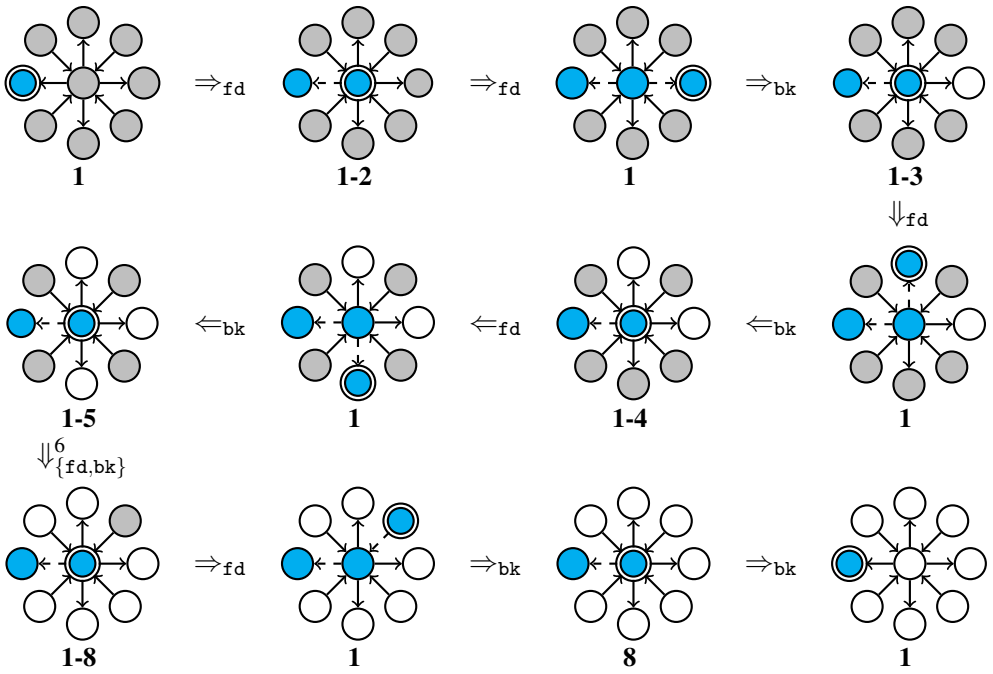

Figure 2: Matching attempts with the `forward` rule. `fd` and `bk` denote `forward` and `back`, respectively.

For example, consider the rule `move` from Figure 8. 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 is 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 or incompatible marks do not match; thus, the matching algorithm need not iterate through them. By organising the edges incident to a node into linked lists based on their mark and orientation, the matching algorithm can selectively consider linked lists of edges of correct mark and orientation. More precisely, in the new compiler, we updated the graph structure of the 2020 compiler by replacing the two linked lists with a two-dimensional array of linked lists of edges. 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 array, therefore, consists of 5 rows (unmarked, dashed, red, blue, green) and 3 columns (incoming, outgoing, loop), totalling 15 cells that each store a single linked list. See Figure 3 for an illustration.

4 Finding Nodes in Constant Time

In this section, we explain the problem of disconnected input graphs. For example, consider the program `is-discrete` in Figure 4. The program fails if and only if the input graph is discrete, that is, contains no edges. We assume that the input graph is unmarked. The program is composed of a loop followed by a test. The rule `mark` in the loop marks and roots an arbitrary unmarked node while the rule `isolated` checks whether the node rooted by `mark` is isolated. Notice that the node in the left-hand

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

Figure 3: Two-dimensional array of linked lists of edges.

side of `isolated` is to be deleted and the right-hand side node is to be created. Hence, by the dangling condition, the rule is applicable only to a red root node. If `isolated` is not applicable, the node rooted by `mark` is not isolated and the loop is terminated by the `break` command. Finally, the rule `root` checks if a red root exists in the host graph, which is the case if and only if the application of `isolated` failed.

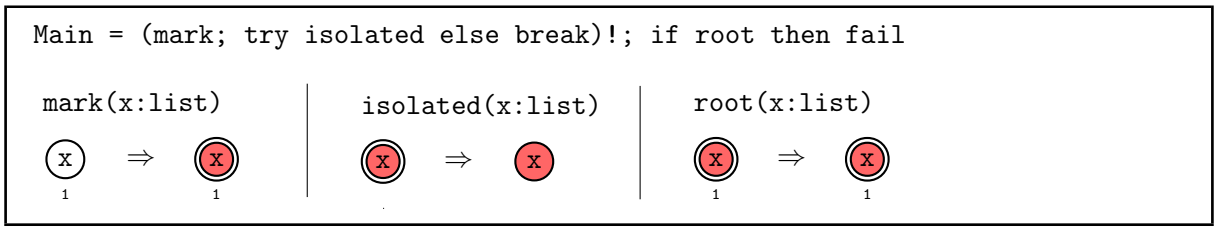

Figure 4: The program `is-discrete`.

The 2020 compiler matched the rule `mark` with a complexity of $O(n)$, where n is the number of nodes in the host graph. In order to find an unmarked node, the matching algorithm had to iterate through the linked list containing all nodes. As the loop body is executed at most n times, the overall complexity of `is-concrete` was $O(n^2)$, as illustrated by the timing diagram in Figure 5.

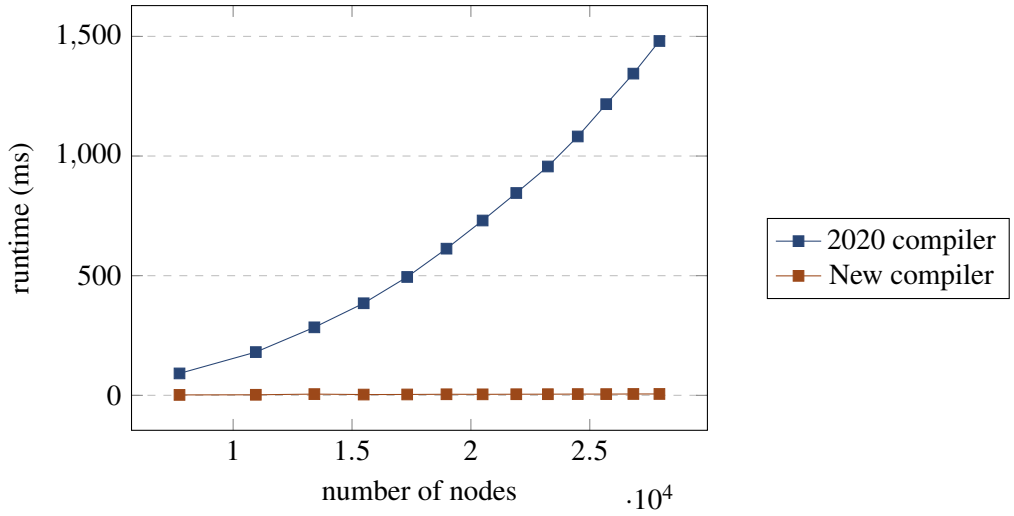

Figure 5: Measured performance of `is-discrete` on discrete graphs under the 2020 compiler and the new compiler.

unmarked	...
grey	...
red	...
green	...
blue	...

Figure 6: Array of linked lists of nodes.

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 7: Updated runtime complexity assumptions. Procedures modified in this paper are highlighted in grey. Procedures modified in [2] are highlighted in light blue. n is the size of the input.

5 Second Compiler Enhancement

To overcome the problem described in Section 4, the new graph data structure stores host-graph nodes in five global linked lists. Each list corresponds to one of the node marks red, green, blue, grey or unmarked, and holds all nodes with that mark. The first element of each linked list can be accessed in constant time and hence, for example, the rule `mark` in Figure 4 can be matched in constant time. The matching algorithm inspects the first element in the linked list of unmarked nodes. As the left-hand side of `mark` requires an unmarked node with an arbitrary label, the first node in the list will match. In case the list of unmarked nodes is empty, the host graph does not contain any unmarked node and thus the application of `mark` fails.

As a result of these changes, the time complexity of the program `is-discrete` under the new compiler is reduced to $O(n)$. Figure 5 highlights the difference in runtimes of the program `is-discrete` run under both compilers.

To reason about programs, we need to make assumptions on the complexity of certain elementary

operations. Figure 7 shows the complexity of basic procedures of the C code generated by the GP 2 compiler, adapted from [5]. The grey rows indicate existing procedures updated by the changes introduced in this paper. The time complexities are consistent with the runtimes observed in all our case studies with the new compiler.

6 Case Study: Recognising Acyclic Graphs

Checking whether a given graph contains a directed cycle is a basic problem in the area of graph algorithms [14]. A GP 2 program solving this problem is given in [5], but to run in linear time it requires input graphs of bounded node degree. The same paper contains a program for the related problem of recognising binary DAGs, which are acyclic graphs in which each node has at most two outgoing edges. This program has a linear runtime on arbitrary input graphs but is destructive in that the input graph is partially or totally deleted.

6.1 Program

The program `is-dag` in Figure 8 recognises acylic graphs with respect to the following specification.

Input: An arbitrary GP 2 host graph such that

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

Output: If the input graph is acyclic, a host graph that is isomorphic to the input graph up to marks. Otherwise *failure*.

Strictly speaking, this program is destructive in case the input graph is cyclic because the `fail` command in the procedure `Check` does not return an output graph. However, `is-dag` could easily be made completely non-destructive by replacing the `fail` command with a rule creating a distinct structure (such as an unmarked node) which signals the existence of a cycle.

Figure 9 illustrates an execution of `is-dag` on a cyclic input graph, and Figure 10, on an acylic graph. The program implements a directed DFS (depth-first search) of the host graph that marks the visited nodes red or blue, where the red nodes are currently being visited. When modelling the DFS with a stack, red nodes are currently on the stack while blue nodes have been previously on the stack and were popped because they require no further visits. Moreover, the top of the stack is a root node.

It is an invariant of `is-dag` that there is at most one root in the host graph throughout the program's execution. The graph contains a cycle if and only if the search finds an edge from the root to a red node. In fact, if u is the root and v is a red node adjacent to u via an edge from u to v, then there must exist a directed path from v to u. Hence a directed cycle has been found.

Consider the loop `(init; DFS!; try unroot else break)!` of `is-dag`'s main procedure. Rule `init` selects an arbitrary grey node as a root to start a directed DFS. The loop `DFS!` moves the root in depth-first fashion through the host graph. The procedure uses a `try-else` command to find any unprocessed (that is, unmarked) edge outgoing from the root. It does this by calling `next_edge`.[3] If there is such an edge, the rule marks it red so that it can be uniquely identified by the rest of the procedure. If no such edge exits, the root can no longer move forward and the `else` statement is invoked instead.

After a successful application of `next_edge`, the root is adjacent to either (1) a grey node, (2) a blue node, or (3) a red node. In case (1), the rule `move` moves the root to the grey node, marks it red and

[3] In the programs of this paper, we use the magenta colour to represent the wildcard mark `any`.

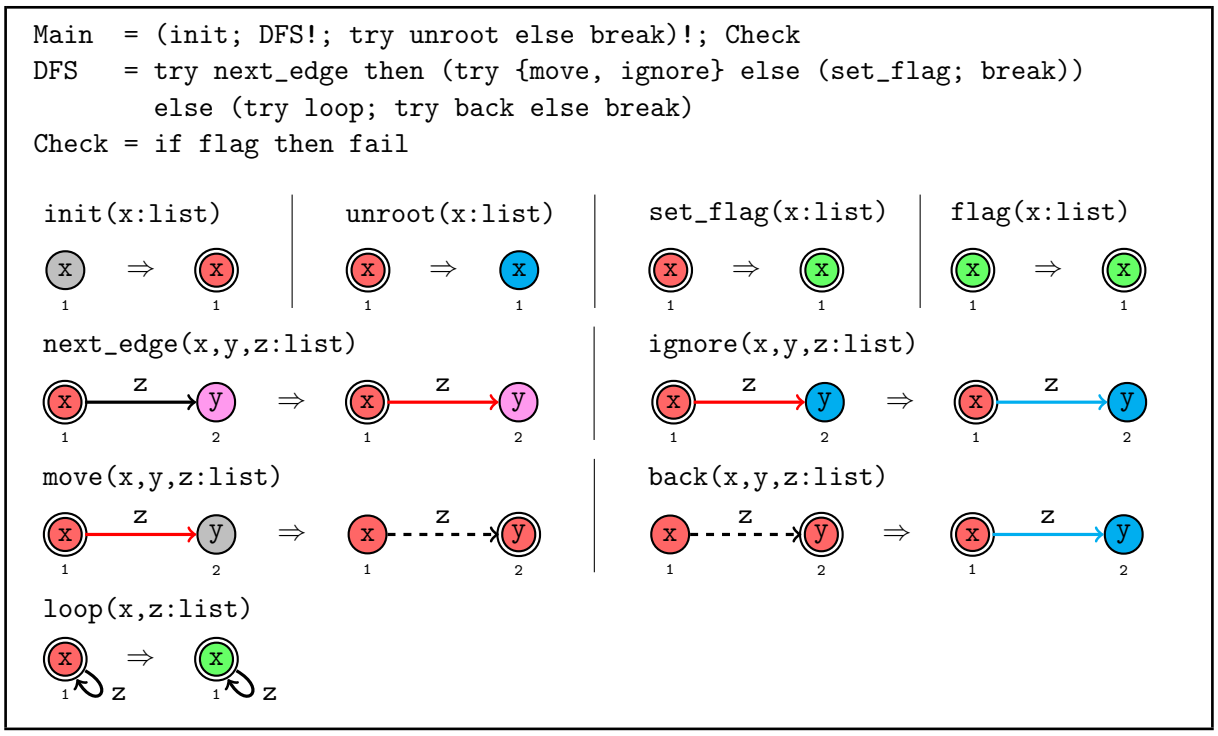

Figure 8: The program `is-dag`.

dashes the traversed edge. Dashed edges represent the path followed by the directed DFS. In case (2), the red edge is marked blue by the rule `ignore` so that it can no longer be matched by `next_edge`. In case (3), neither `move` nor `ignore` is applicable so that `set_flag` marks the root green, indicating the existence of a cycle.

If `next_edge` is not applicable, the command sequence `(try loop; try back else break)` is executed. Rule `loop` checks whether there is a loop attached to the root. If this is the case, the rule marks the root green, similar to `set_flag`. Then rule `back` is tried which implements the *pop* operation in the above mentioned stack model. The rule moves the root backwards along an incoming dashed edge. If no incoming dashed edge is present, the root must be the only element on the stack so that the `break` command terminates the loop `DFS!`.

Upon termination of `DFS!`, the rule `unroot` attempts to turn the root into an unrooted blue node. If this is not possible, the root must have been marked green by `set_flag` or `loop`. This implies the existence of a cycle and hence the outer loop of `is-dag` is terminated.

If rule `unroot` could be applied, there may still be nodes that have not been visited by the DFS. These are nodes that are not directly reachable from the initial nodes chosen so far. In this case the execution of the outer loop is continued until `init` is no longer applicable or `unroot` fails.

Finally, the procedure `Check` tests whether a green flag exists in the host graph. Should this be the case, a cycle was found and the command `fail` terminates the program with failure. Otherwise, the program terminates by returning a host graph which is isomorphic to the input graph up to the generated marks.

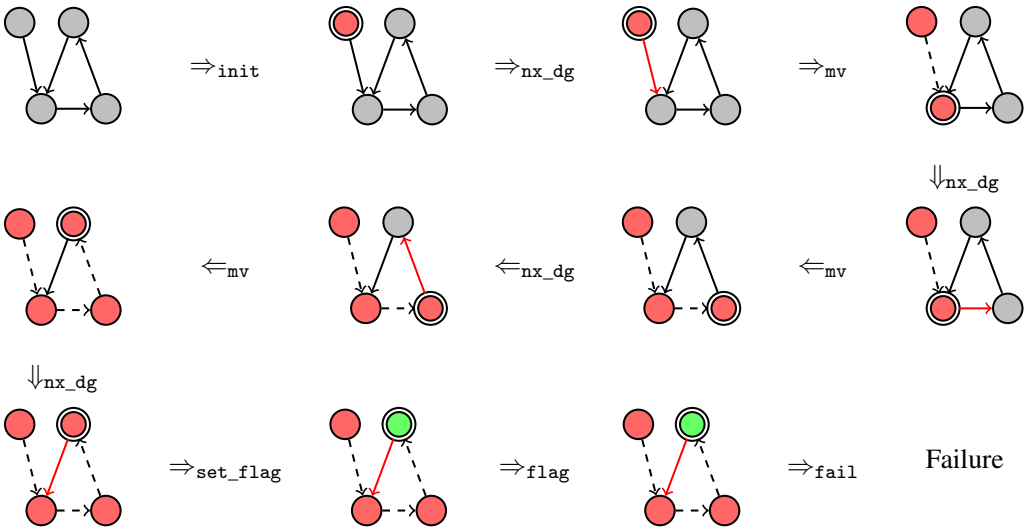

Figure 9: Sample execution of `is-dag` on a cyclic graph.

6.2 Time Complexity

All rules of the program `is-dag` apply in constant time under the complexity assumptions of the modified GP 2 compiler (Figure 7). The rule `init` applies in constant time since any grey-marked node is a match for the rule. As the generated graph data structure keeps a linked list of nodes for each node mark, the matching algorithm can select a grey node in constant time regardless of the number of non-grey nodes in the host graph.

It can be observed that nodes and edges are never remarked by a mark they previously had. 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. $O(n)$ where n is the number of nodes and edges in the input graph. To support this claim, we conducted runtime experiments on various classes of bounded-degree graphs (Figures 11, 12, 14, 15 and 16), and unbounded-degree graphs (Figures 13 and 17). The timing results are shown in Figure 18.

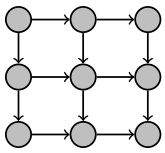

Figure 11: Grid graph.

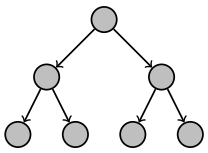

Figure 12: Binary tree.

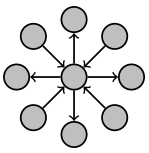

Figure 13: Star graph.

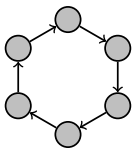

Figure 14: Cycle graph.

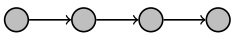

Figure 15: Linked list.

Figure 16: Discrete graph.

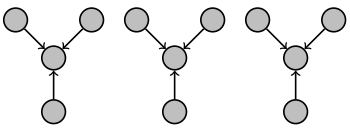

Figure 17: k k-star graphs.

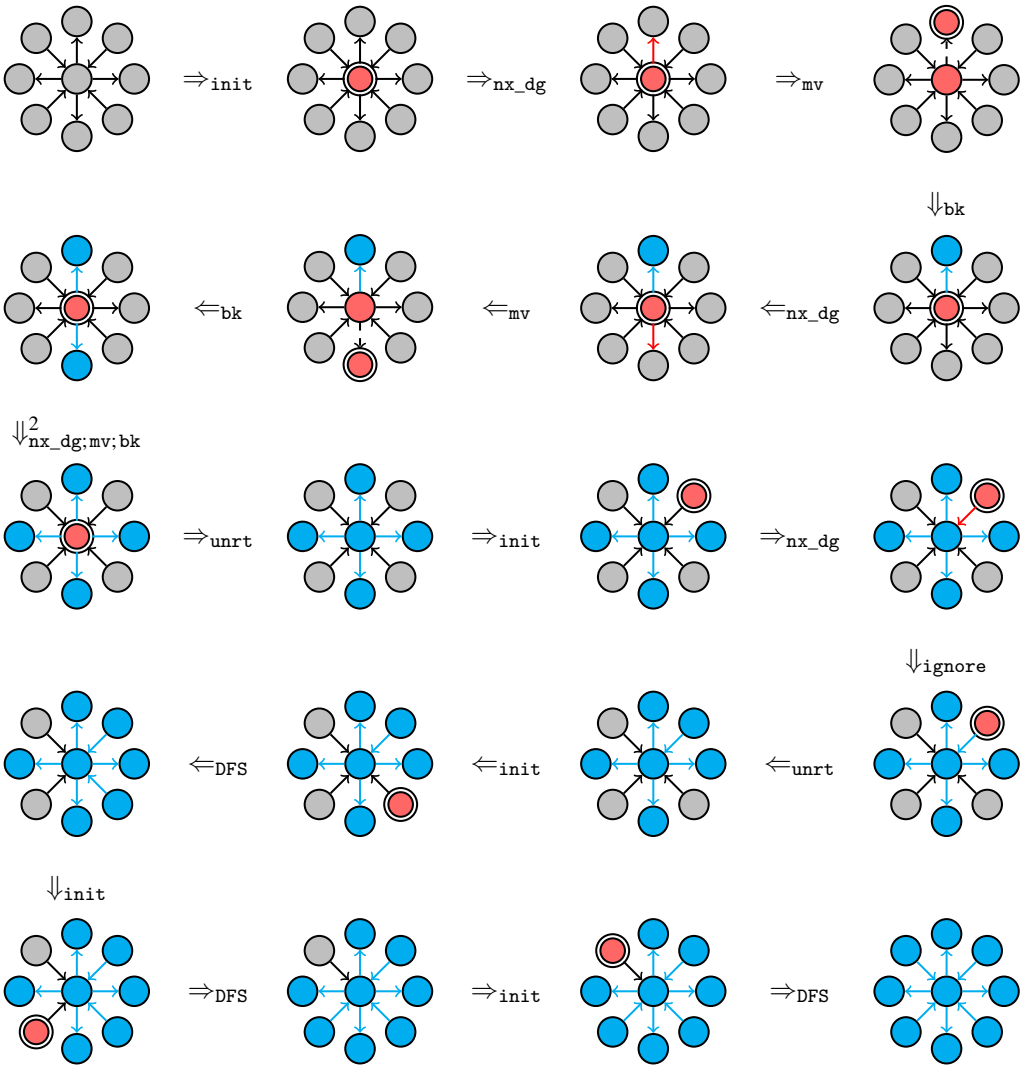

Figure 10: Sample execution of `is-dag` on an acyclic graph.

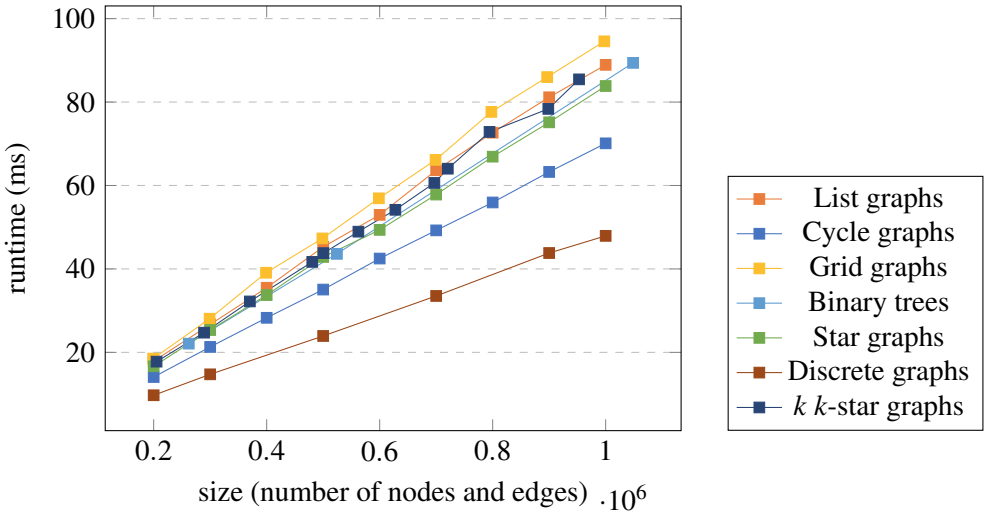

Figure 18: Measured performance of the program `is-dag` under the modified compiler.

7 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 19 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 specification.

Input: An arbitrary GP 2 host graph such that

1. each node is non-rooted and marked grey, and
2. each 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.

7.1 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.

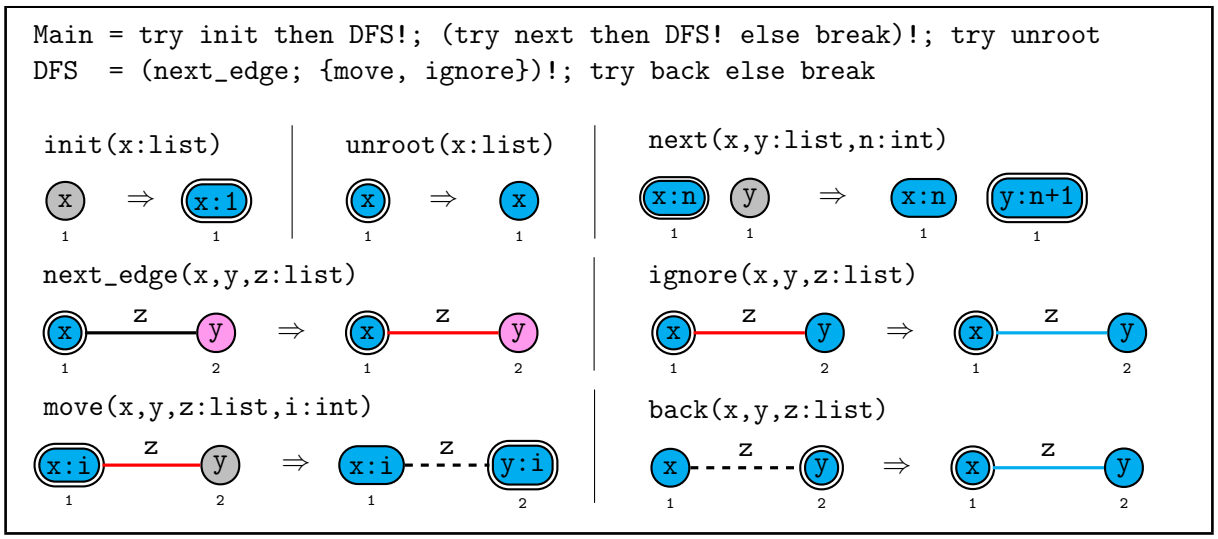

Figure 19: The program `component-numbering`.

7.2 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 20 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.

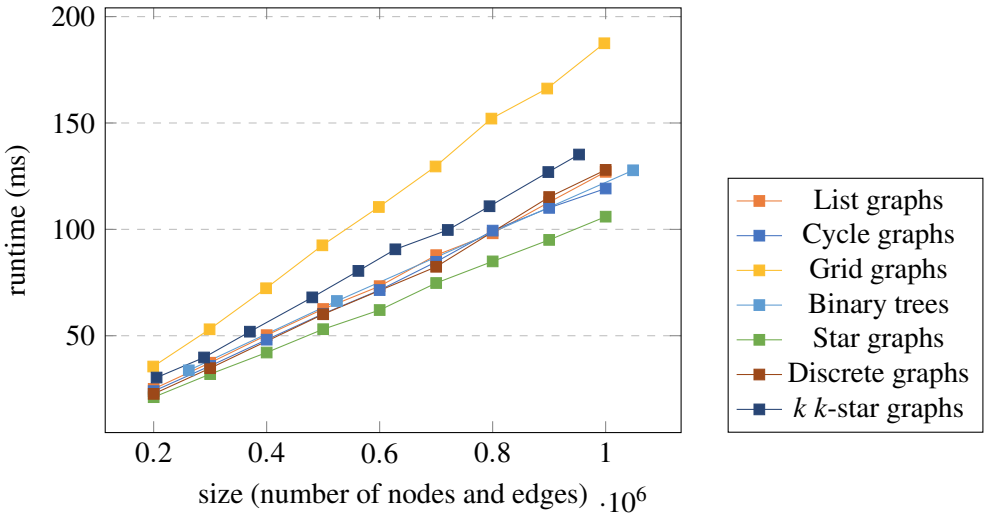

Figure 20: Measured performance of the program `component-numbering` under the modified compiler.

8 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 [3], 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.

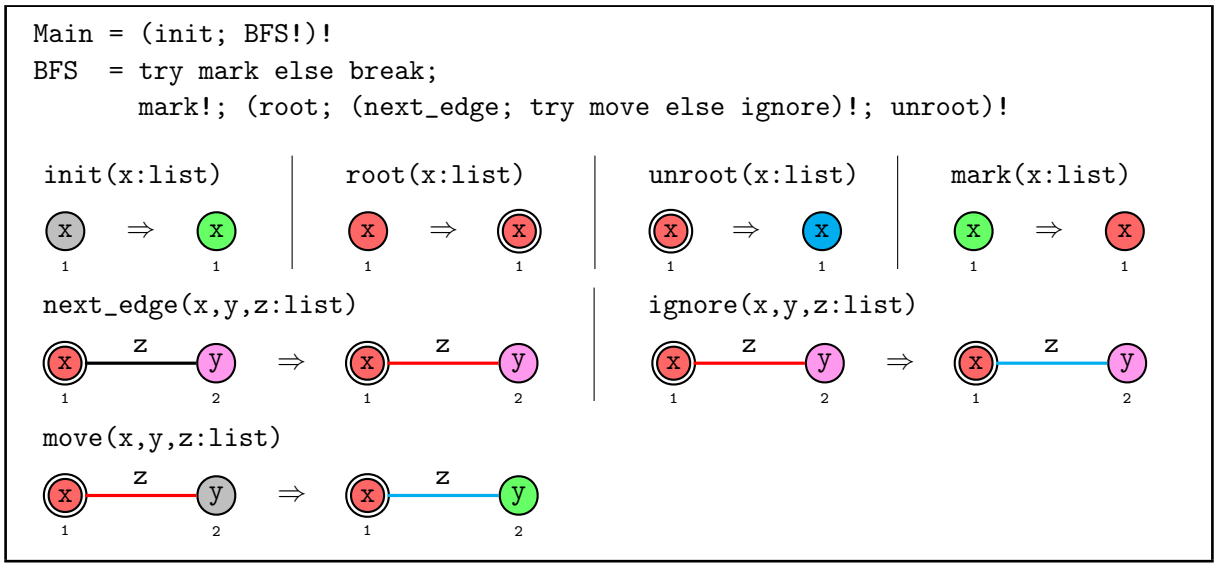

Figure 21: The program `bfs`.

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 specification.

Input: An arbitrary GP 2 host graph such that

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

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

8.1 Program

The program exploits the advantages of the compiler modifications in order to find the next node to expand from in constant time. The program `bfs` consists of a loop, `(init; BFS!)`, which itself contains a 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. That procedure loops as long as the connected component of the node marked green by `init` contains non-blue nodes.

Intuitively, at the beginning of the execution of `BFS`, the program seeks to mark all nodes directedly adjacent to the node marked green by `init`. 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 rule `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. Once `BFS!` terminates, the rule `init` is called again to start a bread-first search procedure from a different non-visited connected component. The loop continues until all nodes in the host graphs are visited.

8.2 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).

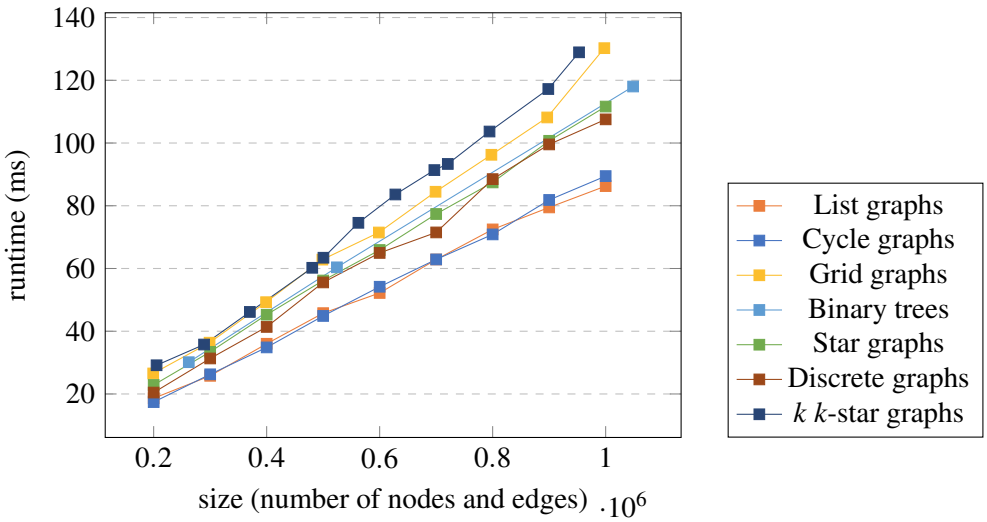

Figure 22: Measured performance of the program `bfs` under the modified compiler.

9 Conclusion

We have demonstrated by case studies how to implement in GP 2 graph algorithms based on depth-first and breadth-first search such that a linear runtime is achieved, even if input graphs have an unbounded node degree or are possibly disconnected. Addressing the issues of unbounded degree and disconnectedness has been an open problem since the publication of the first paper on rooted graph transformation [4]. Up to now, only certain graph reduction programs that destroy their input graphs could be designed to run in linear time on graph classes that have an unbounded node degree or contain disconnected graphs [5].

Our approach involves both enhancing the graph data structure generated by the GP 2 compiler and developing a programming technique that leverages the new graph representation. Previously, the graph data structure in the C program generated by the compiler stored all host-graph nodes in a single linked

list. Hence, if host graph nodes may have different marks, searches within this list required linear time, preventing constant-time rule matching. In contrast, the new data structure finds a node with a given mark or an edge with a given mark and orientation in constant time.

We speculate that all linear-time graph algorithms based on depth-first or breadth-first search can be implemented as GP 2 programs running in linear time. More generally, we intend to implement a GP 2 library of advanced data structures, such as priority queues, Fibonacci heaps, or AVL trees, to support programmers in constructing GP 2 versions of conventional graph algorithms that match the time complexity achievable in the imperative setting.

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] Ziad Ismaili Alaoui & Detlef Plump (2024): *Linear-Time Graph Programs for Unbounded-Degree Graphs*. In: *Proc. 17th International Conference on Graph Transformation (ICGT 2024)*, *Lecture Notes in Computer Science* 14774, Springer, pp. 3–20, doi:10.1007/978-3-031-64285-2_1.

[3] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language*. Ph.D. thesis, Department of Computer Science, University of York, UK. Available at `https://etheses.whiterose.ac.uk/12586/`.

[4] Christopher Bak & Detlef Plump (2012): *Rooted Graph Programs*. In: *Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, *Electronic Communications of the EASST* 54, doi:10.14279/tuj.eceasst.54.780.

[5] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs*. *Science of Computer Programming* 214, p. 102727, doi:10.1016/j.scico.2021.102727.

[6] Graham Campbell, Jack Romö & Detlef Plump (2020): *The Improved GP 2 Compiler*. *ArXiv e-prints* arXiv:2010.03993, doi:10.48550/arXiv.2010.03993. 11 pages.

[7] Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest & Clifford Stein (2022): *Introduction to Algorithms*, fourth edition. The MIT Press.

[8] Heiko Dörr (1995): *Efficient Graph Rewriting and its Implementation*. *Lecture Notes in Computer Science* 922, Springer, doi:10.1007/BFb0031909.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] Steven S. Skiena (2020): *The Algorithm Design Manual*, third edition. Springer, doi:10.1007/978-3-030-54256-6.

[15] 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.