A Fast Graph Program for Computing Minimum Spanning Trees

Brian Courtehoute and Detlef Plump
Department of Computer Science, University of York, York, UK
{bc956,detlef.plump}@york.ac.uk

When using graph transformation rules to implement graph algorithms, a challenge is to match the efficiency of programs in conventional languages. To help overcome that challenge, the graph programming language GP 2 features *rooted* rules which, under mild conditions, can match in constant time on bounded degree graphs. In this paper, we present an efficient GP 2 program for computing minimum spanning trees. We provide empirical performance results as evidence for the program's subquadratic complexity on bounded degree graphs. This is achieved using depth-first search as well as rooted graph transformation. The program is based on Boruvka's algorithm for minimum spanning trees. Our performance results show that the program's time complexity is consistent with that of classical implementations of Boruvka's algorithm, namely $O(m \log n)$, where m is the number of edges and n the number of nodes.

1 Introduction

GP 2 is an experimental rule-based graph programming language with simple semantics to facilitate formal reasoning. It has been shown that every computable function on graphs can be expressed as a GP 2 program [16].

A challenge in rule-based graph programming is reaching the time efficiency of conventional programs due to the cost of graph matching. In general, finding a match for a graph L in a graph G takes $\text{size}(G)^{\text{size}(L)}$ time, when in practise, we often want to do it in constant time.

Other programming languages based on graph transformation rules include AGG [17], GReAT [1], GROOVE [12], GrGen.Net [14], Henshin [2] and PORGY [11], but we are not aware that any of them are able to match the time complexity of subquadratic graph algorithms.

GP 2 allows to speed up graph matching by using *rooted* graph transformation rules, which was first introduced by Bak and Plump [4]. This enables nodes in the host graph declared as *roots* to be accessed in constant time, making matching locally around those in constant time possible for connected graphs of bounded degree.

In previous work, we developed GP 2 programs that match the time complexity of their conventional counterparts on connected graphs of bounded degree. The first such program produces a 2-colouring, and was shown to match the measured execution times of a tailor-made 2-colouring C program [5]. More GP 2 programs that run in linear time on connected graphs of bounded degree include tree recognition, binary DAG recognition, and topological sorting [8].

Here we continue this work by presenting an efficient GP 2 program computing a minimum spanning tree of a connected graph. Remember that a *spanning tree* of an undirected connected graph G with weighted edges is a subgraph that contains all nodes of G and is a tree. A *minimum spanning tree* (MST) of G is a spanning tree such that the sum of all edge weights is minimum. For example, Figure 1 shows a graph and its minimum spanning tree.

B. Hoffmann and M. Minas (Eds.): Eleventh International Workshop on Graph Computation Models (GCM 2020)
EPTCS 330, 2020, pp. 163–180, doi:10.4204/EPTCS.330.10

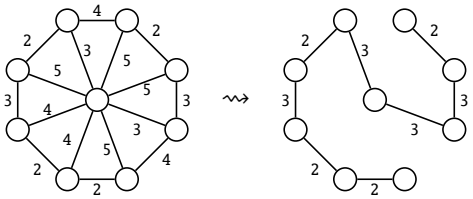


Figure 1: A weighted graph and its minimum spanning tree

MSTs are useful for building networks between a set of nodes by minimising the cost. Such networks include communication, transport, piping, and computer networks. They also provide time efficient approximations to hard problems such as the travelling salesperson problem or the Steiner tree problem [20].

Classical algorithms for finding MSTs given by Prim, Kruskal, and Boruvka all run in time $O(m \log n)$, where m is the number of edges and n is the number of nodes [6]. However, to reach this time bound, the algorithms of Prim and Kruskal need data structures such as binary heaps or union-find data structures. In contrast, Boruvka's algorithm can be implemented efficiently without fancy data structures. Hence we choose to implement this algorithm in GP 2.

In Section 3, we give the GP 2 program `mst-boruvka`, which is based on depth-first search and rooted graph transformation. In Section 4 we give execution time measurements as evidence that on bounded degree graphs, the program's complexity is consistent with the $O(m \log n)$ time bound of implementations of Boruvka's algorithm in conventional languages.

This paper is a revised and extended version of [10].

2 The Graph Programming Language GP 2

This section briefly introduces GP 2, a graph transformation language, first defined in [15]. Up-to-date versions of the syntax and semantics of GP 2 can be found in [3]. The language is implemented by a compiler generating C code [5, 9].

2.1 Graphs, Rules and Programs

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

The principal programming construct in GP 2 consist of conditional graph transformation rules labelled with expressions. For example, the rule `min_s` in Figure 8 has three formal parameters of type `list`, two of type `int`, a left-hand graph and a right-hand graph which are specified graphically, and a textual condition starting with the keyword `where`.

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

Besides carrying expressions, nodes and edges can be *marked* red, green or blue. In addition, nodes can be marked grey and edges can be dashed. For example, rule `root_current` in Figure 4 contains red and unmarked nodes and a red edge. Marks are convenient, among other things, to record visited items

during a graph traversal and to encode auxiliary structures in graphs. The programs in the following sections use marks extensively.

Rules operate on *host graphs* which are labelled with constant values (lists containing integers and character strings). Formally, the application of a rule to a host graph is defined as a two-stage process in which first the rule is instantiated by replacing all variables with values of the same type, and evaluating all expressions. This yields a standard rule (without expressions) in the so-called double-pushout approach with relabelling [13]. In the second stage, the instantiated rule is applied to the host graph by constructing two suitable pushouts. We refer to [3] for details and only give an equivalent operational description of rule application.

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

In this construction, the *dangling condition* requires that nodes in S corresponding to unnumbered nodes in $\hat{L}$ (which should be deleted) must not be incident with edges outside S. The rule's application condition is evaluated after variables have been replaced with the corresponding values of $\hat{L}$, and node identifiers of L with the corresponding identifiers of S. For example, the condition `i < j` of rule `min_s` in Figure 8 requires that the integer label of the edge from node $g(1)$ to node $g(2)$ is smaller than the integer label of the edge from node $g(1)$ to node $g(3)$, where $g(1)$, $g(2)$, $g(3)$ are the nodes in S corresponding to 1, 2, 3.

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

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

The command `if` C `then` P `else` Q is executed on a host graph G by first executing C on a copy of G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The command `try` C `then` P `else` Q has a similar effect, except that P is executed on the result of C's execution. If `then` P or `else` Q are omitted, no additional command is executed in the missing cases.

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

In general, the execution of a program on a host graph may result in different graphs, fail, or diverge. The operational semantics of GP 2 defines a semantic function which maps each host graph to the set of all possible outcomes. See, for example, [16].

2.2 Rooted Programs

The bottleneck for efficiently implementing algorithms in a language based on graph transformation rules is the cost of graph matching. In general, to match the left-hand graph L of a rule within a host graph G requires time polynomial in the size of L [4, 5]. As a consequence, linear-time graph algorithms

in imperative languages may be slowed down to polynomial time when they are recast as rule-based programs.

To speed up matching, GP 2 supports *rooted* graph transformation where graphs in rules and host graphs are equipped with so-called root nodes. Roots in rules must match roots in the host graph so that matches are restricted to the neighbourhood of the host graph's roots. We draw root nodes using double circles. For example, in the rule `root_current` of Figure 4, the nodes labelled `2` are roots and so is the node labelled `1` in the right-hand side.

Rooted graph matching can be implemented to run in constant time under mild conditions, provided there are upper bounds on the maximal node degree and the number of roots in host graphs [4].

3 Boruvka's Algorithm in GP 2

In this section, we take a look at Boruvka's algorithm and its implementation in GP 2. We go through an example execution of the program `mst-boruvka` in Subsection 3.1 in order to give an intuitive understanding of the program and how it relates to the algorithm. Subsections 3.2, 3.3, 3.4, 3.5, and 3.6 contain the program itself and its description.

Prim's, Kruskal's, and Boruvka's algorithms for computing MSTs can all be implemented to run in $O(m \log n)$ time, where m is the number of edges, and n the number of nodes. However Prim's algorithm needs binary heaps to achieve it, and Kruskal's algorithm the union find data structure [6]. The advantage of Boruvka's algorithm is that it does not need fancy data structures to reach that time complexity bound [20]. GP 2 has no predefined data structures except for the host graph that it transforms. Any additional data structures need to be encoded in the host graph itself, which can make a program tricky to read. Hence we choose to implement Boruvka's algorithm in GP 2.

Algorithm 1 shows pseudocode for Boruvka's algorithm. Although it cannot translate directly into GP 2, it is a suitable starting point for the development of a GP 2 program.

Algorithm 1 Boruvka's MST algorithm on an input graph G

1: `Preprocess`: initialise the spanning forest F to be the nodes of G
2: **while** F consists of more than one tree **do**
3: **for** each tree T in F **do**
4: `FindEdge`: select a minimum weight edge between T and $G - T$, prioritising already selected edges if they are minimum
5: **end for**
6: `GrowForest`: add the selected edges to F
7: **end while**

The idea of Boruvka's algorithm is to initialise a forest as the nodes of the input graph without any edges, and to grow that forest by adding minimum-weight edges from between its connected components until it becomes a minimum spanning tree of the input graph.

As illustrated in Figure 2, the input of `mst-boruvka` is a connected graph with unmarked nodes and edges. Nodes are unlabelled, and edges have integer labels. In the output, the subgraph induced by the blue edges are a minimum spanning tree of the input. The additional root with label 1 is an auxiliary construct used in the execution of the program (which could be removed in constant time).

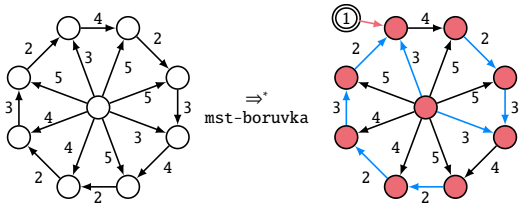


Figure 2: Example input and output of `mst-boruvka`

3.1 Example Execution

Throughout the execution of `mst-boruvka`, the graph induced by the blue edges is a subgraph of the minimum spanning tree highlighted in the output. We shall call this forest F, and its connected components its *trees*. Let us explore how `mst-boruvka` executes using the example in Figure 3, and compare it to the pseudocode in Algorithm 1. The `Main` procedure of `mst-boruvka` is depicted in Figure 4.

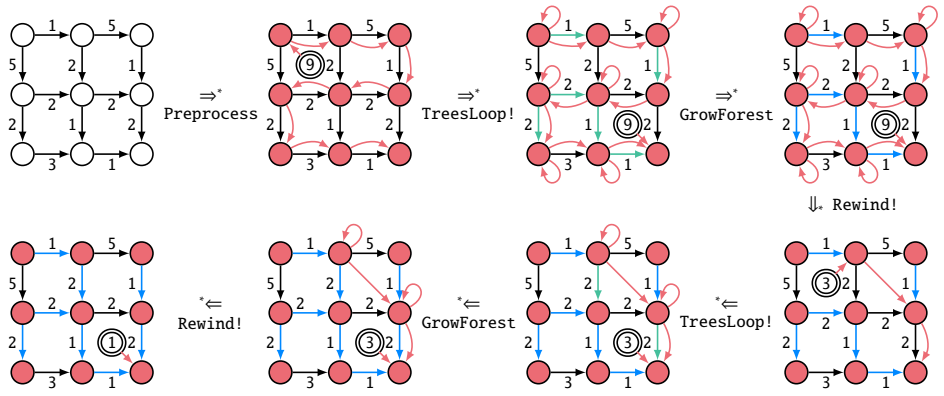


Figure 3: Example execution of `mst-boruvka`

The procedure `Preprocess` initialises the forest F to be just the nodes of the input(see line 1 of the pseudocode). It also sets up a linked list of red edges and red nodes that helps the program loop over the trees of F efficiently. Each tree of F is represented by exactly one of its nodes being an entry in the linked list. Additionally, there is a pointer in the form of an unmarked root node with an outgoing red edge towards the "current" node in the linked list. The pointer also stores the number of trees the forest has in order to efficiently check whether only one tree is left, terminating the main loop (see line 2 of the pseudocode).

The loop `TreesLoop!` moves the pointer through the nodes of the linked list, effectively looping over the trees of F (see line 3 of the pseudocode). On each tree T, the procedure `FindEdge` is called, which selects a minimum weight edge between T and its complement in the host graph by marking it green (see line 4 of the pseudocode). If there is already an adjacent green edge with minimum weight, no new edge is selected since that could introduce a cycle into F. To ensure that only one node of each tree is part of the list, the current tree gets marked for deletion from the list using a red loop under certain conditions. Subsection 3.6 elaborates on this.

The procedure `GrowForest` adds the selected edges to F by green edges into blue ones (see line 6 of the pseudocode).

The loop `Rewind!` serves to maintain the linked list. It moves the pointer back to the beginning

of the list. On the way, it removes nodes that have been marked for deletion with a red loop. It also decrements the pointer's label each time it encounters such a node since that node's tree has been merged with another tree.

3.2 The GP 2 Program `mst-boruvka`

The program `mst-boruvka` is depicted in Figure 4. Most of it has been explained by the example execution in Subsection 3.1. Let us now examine the loop `TreesLoop!`.

```
Main = Preprocess; Loop!
Loop = if one_tree then break else Body
Body = TreesLoop!; GrowForest; Rewind!
TreesLoop = root_current; TraverseTree; MarkForDeletion; CleanUp;
            try next_tree else break
TraverseTree = ColourBlue; FindEdge
CleanUp = ColourRed; unroot_red!
```

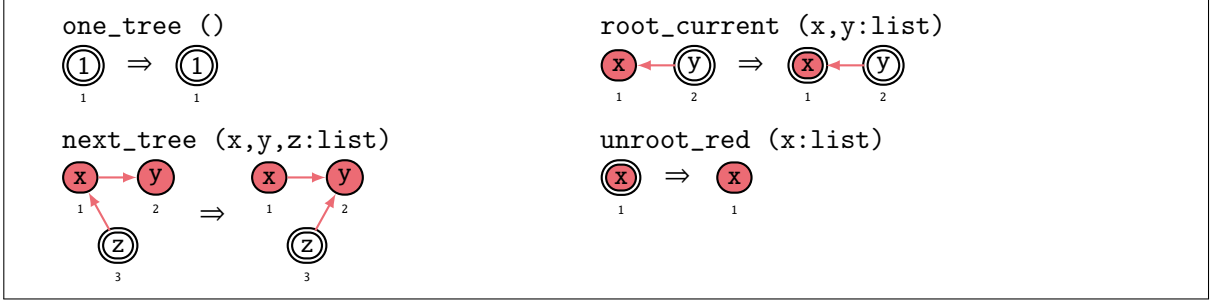


Figure 4: The GP 2 program `mst-boruvka`

The purpose of the loop `TreesLoop!` is to find a minimum weight edge from each tree to its complement and mark it green. It initialises by rooting the node the pointer points to. Then that node's tree is marked blue with the procedure `ColourBlue` so it can easily be distinguished from the rest of the graph. `FindEdge` then finds the minimum edge from the tree to its complement. The procedure `MarkForDeletion` marks the tree for deletion if it will be merged with another one. The procedure `ColourRed` makes the nodes of the tree be red again. The command `unroot_red!` unroots any red roots. The rule `next_tree` then moves the pointer to the next entry in the linked list.

3.3 The Procedure `Preprocess`

The procedure `Preprocess` depicted in Figure 5 uses depth-first search (DFS) to construct the linked list and the pointer. An example of its input and output can be seen in Figure 3.

The rule `pre_init` initialises some node of the input to be the starting point of the DFS, and constructs the pointer. Since initially each node is its own tree, the pointer's label will count the number of nodes encountered during the DFS. Red nodes are considered to be discovered by the DFS, and unmarked nodes undiscovered.

The rules `pre_forward1` and `pre_forward2` are called non-deterministically. They both move the red root to an adjacent unmarked node. The rules contain *bidirectional edges* (without arrowheads) that

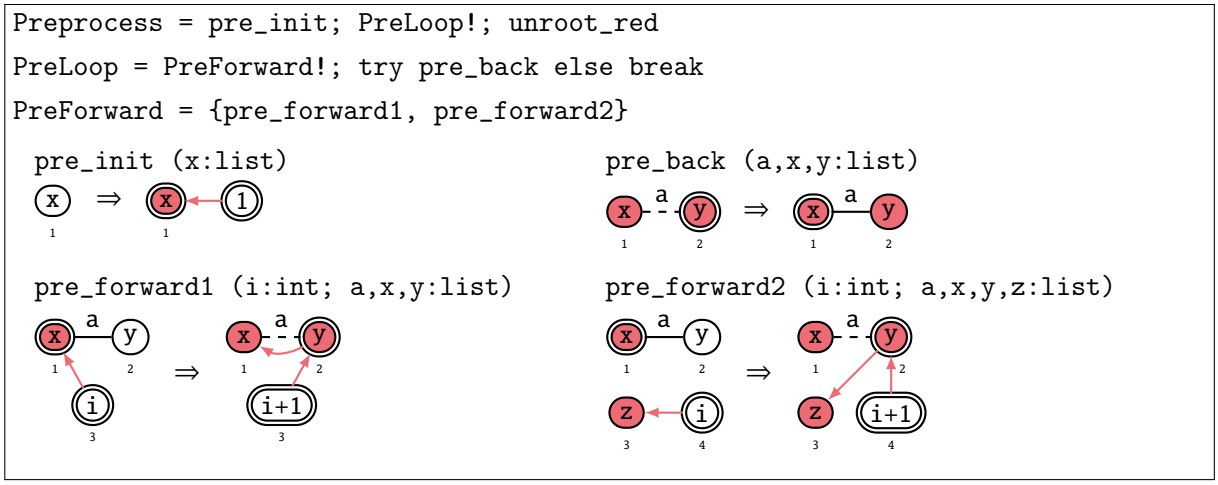


Figure 5: The procedure `Preprocess`

can be matched in either orientation. The rule is a shorthand for a non-deterministic call of copies of the same rule whose bidirectional edges have been replaced with directed edges in all possible combinations of orientation. The dashed edge serves as a way to keep track of the path the DFS has taken, which is backtracked by the rule `pre_back`. The backtracking enables the "forward" rules to find new undiscovered nodes again.

The rules `pre_forward1` and `pre_forward2` also increment the counter and construct the linked list of red edges. The reason we need both rules is to cover both cases of whether the newest entry of the list is also the current red root or not.

3.4 The Procedure `FindEdge`

The procedure `FindEdge`[1], depicted in Figures 7 and 8, serves to find a minimum-weight edge between the current tree (blue nodes) to the rest of the graph (red nodes) using DFS, and to mark it green. If among said minimum edges is an already selected (green) one, it will stay selected, and no additional edge is selected for the current tree. If this were not the case, the selected edges would form a cycle on a 3-cycle whose edges have equal weight for instance, causing the output MST not to be a tree.

Let us examine the example execution of `FindEdge` in Figure 6. It is part of the transition from the fifth to the sixth graph labelled `TreesLoop!` in the example execution of `mst-boruvka` in Figure 3. We start with a graph where the current tree has blue nodes to distinguish it from the rest of the graph. This was done using the procedure `ColourBlue`, which is always called before `FindEdge` as defined in the procedure `TraverseTree` in Figure 4. The nodes of the tree are turned grey, but are still distinguishable from the the rest of the graph, which has red nodes.

The procedure `FindEdge` starts by turning the blue root grey, and creating a green root which serves as a flag indicating whether the minimum edge has been initialised yet. The flag is `1` if initialisation has already happened, and `0` otherwise.

We enter the loop `FindLoop!` and apply `find_forward` to move the root along in the current tree in a depth-first fashion. The flag is not yet set to `1`, so we call `MinSetup` to initialise the minimum edge using `min_init1`. The rule `min_init2` exists in case the grey root's only incident edge has already been

[1]Compared to [10], we added several "`min`" rules to cover all cases.

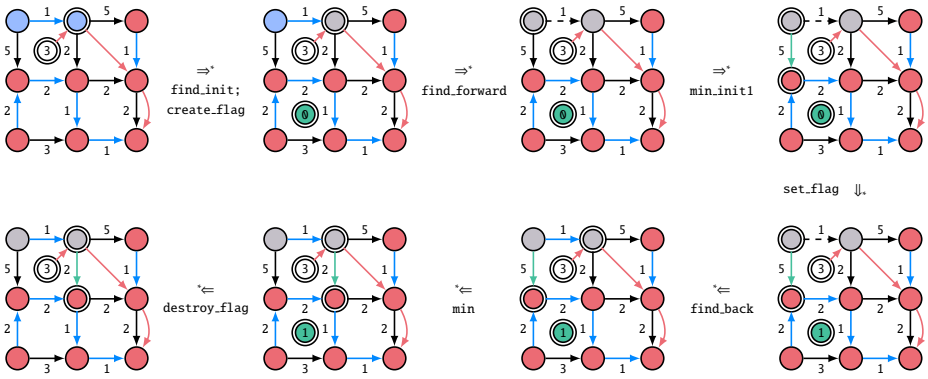


Figure 6: Example execution of `FindEdge`

selected (marked green) when the procedure `FindEdge` was applied to a different tree. An edge selected from both this tree and another tree is represented with a label that is a list consisting of the edge weight followed by a `0`. The currently selected minimum edge of the current tree is represented by a green edge incident to a grey as well as a red root.

We then enter the procedure `Success` which minimises the weight among the unmarked edges incident to the current grey root using the procedure `MinWithS` (which only calls rules that minimise edges incident to the current grey root), and then applies `set_flag` to indicate that the initialisation of the minimum edge is complete.

Next, the rule `find_back` moves the grey root back through the tree in depth-first fashion. We then enter the next iteration of `FindLoop!`. The rule `find_forward` cannot be applied, so we continue with the loop `Minimise!` since the flag has already been set.

The purpose of the loop `Minimise!` is to find an edge incident to the grey root with a smaller weight than the currently selected edge. There are 14 different cases we have to distinguish with the rules that update the minimum edge. They can be seen as combinations of the presence or absence of four flags `s`, `t`, `n`, and `p`, present in the rule names. The flag `s` is present if the new and previous minimum edge share their "source", i.e. the incident grey node in the current tree. The flag `t` is present if the new and previous minimum edge share their "target", i.e. the incident red node outside of the current tree. The flag `n` denotes that the new minimum edge is also a selected minimum edge of a different tree from a previous call of `FindEdge`. The presence of flag `p` indicates that the previous minimum edge has already been selected for a different tree. These edges are denoted by a `0` being appended to their label. They need to be distinguished since their green mark needs to be preserved in order for the program to work correctly.

The "min" rules with both the `s` and `t` flags, i.e. the ones minimising parallel edges, are a special case. We omit the cases that involve previously selected edges (flags `n` or `p`) since such an edge would have already been minimised over its parallel edges by previous applications of `min1_st` and `min2_st`. We use two rules with directed edges labelled `j` instead of one rule with a bidirectional edge labelled `j` due to parallel bidirectional edges being disallowed by GP 2. This is because, if the parallel bidirectional edges are indistinguishable in the left hand side of a rule, the result of the rule application is not necessarily unique up to isomorphism, since it could leave the host graph with an edge in one of two possible directions.

In order to prioritise edges that have already been selected for different trees, we call the rules of the procedure `MinWithN` first. They consist of the rules with flag `n`. We can then call the rest of the rules

```
FindEdge = find_init; create_flag; FindLoop!; destroy_flag
FindLoop = find_forward!; if flag then Minimise!  else
           (try MinSetup); try find_back else break
MinSetup = try min_init2 then Success else (try min_init1 then Success)
Success = MinWithS!; set_flag
Minimise = try MinWithN else MinWithoutN
MinWithS = {min_s, min_sn, min_sp, min_snp, min1_st, min2_st}
MinWithN = {min_n, min_np, min_sn, min_snp, min_tn, min_tnp}
MinWithoutN = {min, min_p, min_s, min_sp, min_t, min_tp, min1_st, min2_st}
```

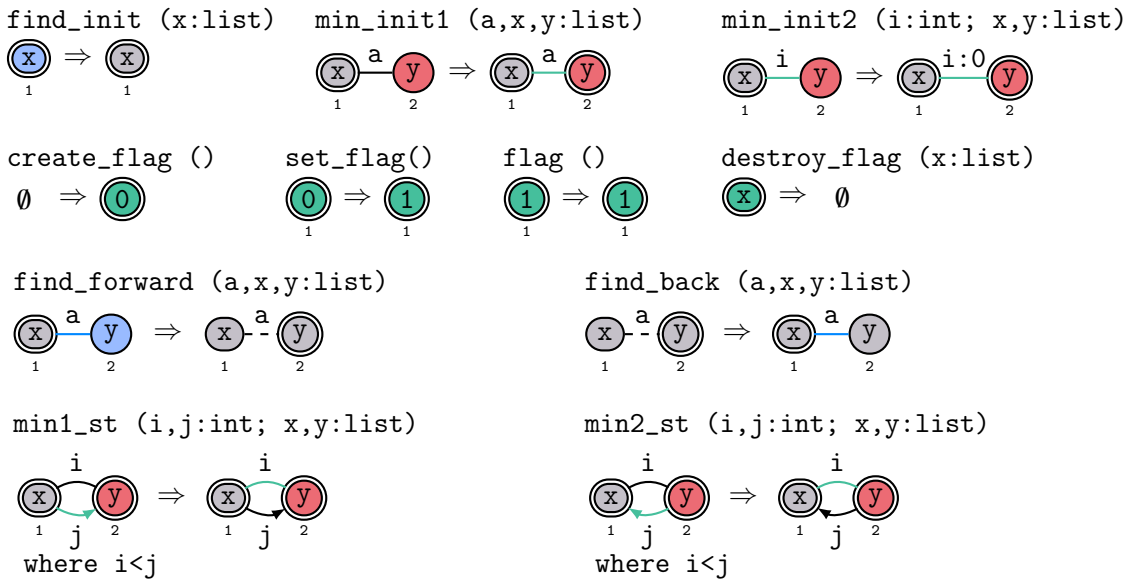


Figure 7: The procedure FindEdge

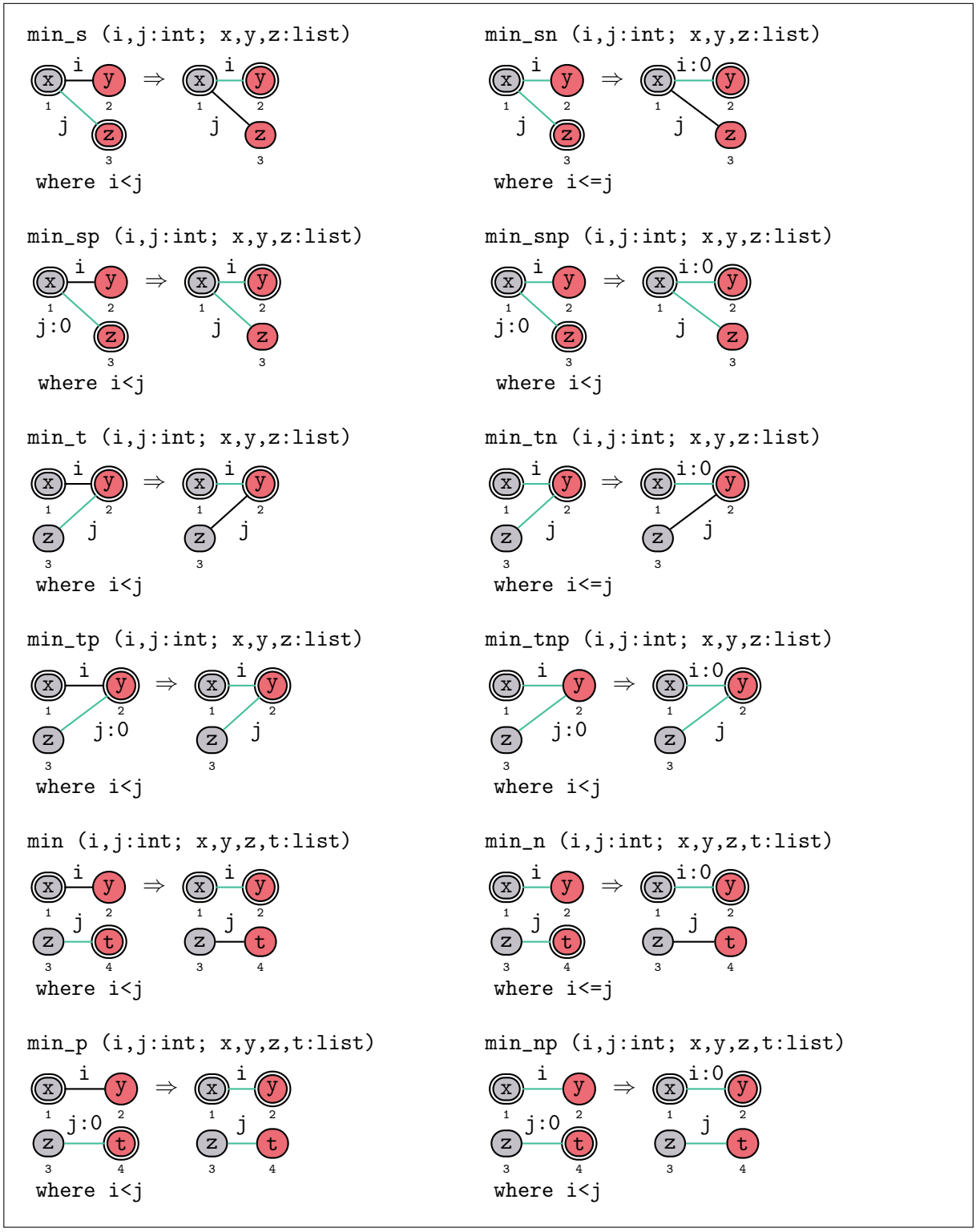


Figure 8: More "min" rules

using `MinWithoutN`. Note that `min_sn`, `min_tn`, and `min_n` (i.e. "min" rules with `n` but not `p`) are the only rules that can be applied if the weights are equal. This is because they are the only ones selecting a previously selected edge, which we prioritise. Making the other rules applicable on equal weights can lead to non-termination. In our example, `min` is applied.

Finally, the DFS terminates and the rule `destroy_flag` deletes the temporary flag needed for this procedure. The flag could have been implemented as an additional list entry of the unmarked root, but was chosen to be its own green root for the sake of semantic clarity.

3.5 The Procedure `GrowForest`

The procedure `GrowForest` depicted in Figure 9 serves to turn the edges selected by `FindEdge!` (green mark) into edges of the forest (blue mark), thus merging some of the trees. Graphs 6 and 7 in the example execution of `mst-boruvka` in Figure 3 exemplify input and output graph of `GrowForest`.

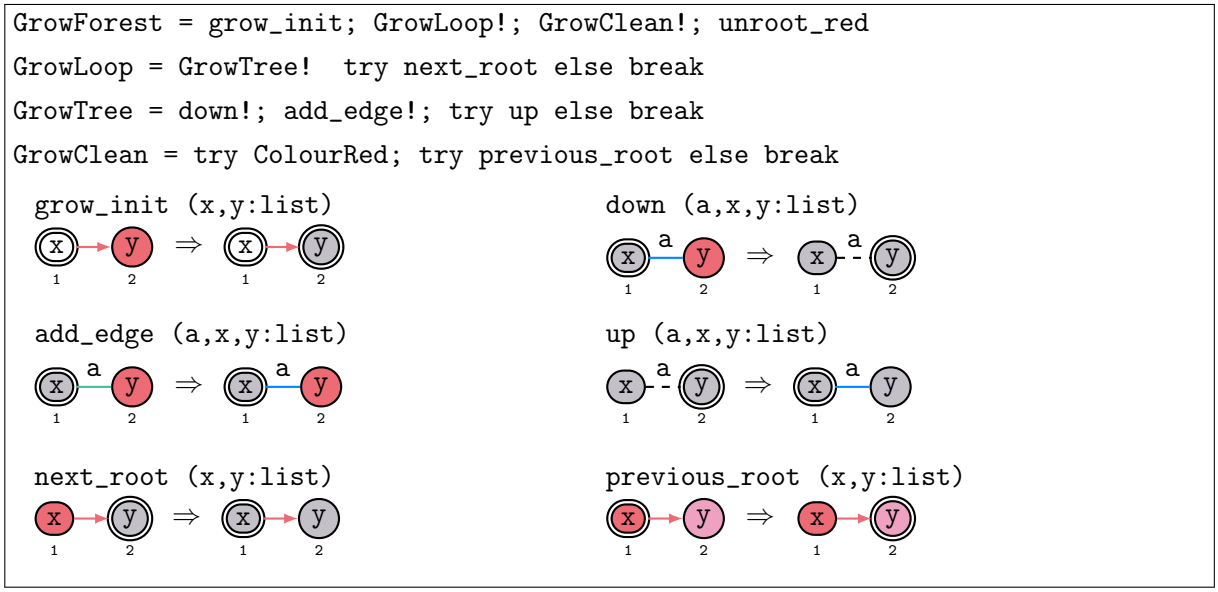


Figure 9: The procedure `GrowForest`

The procedure `GrowForest` traverses the graph by iterating through the list of trees, and conducting a DFS on each tree. `next_root` helps iterate through the list in the direction opposite to the orientation of the red edges.

The rules `down` and `up` play the roles of `forward` and `back` in a DFS. They use blue edges to ensure only the current tree is traversed. `add_edge!` is called right before `up` to turn all green edges adjacent to the grey root blue. After the `up` rule is applied to a grey root, it is not visited again by the DFS, ensuring the new blue edges will not be traversed. Future DFSs will also not traverse these edges since one of its adjacent nodes is grey.

The loop `CleanUp!` iterates through the list of trees in the direction opposite to `GrowTree!` and calls `ColourRed` on each tree to mark the nodes red again.

3.6 Other Procedures

The program `mst-boruvka` calls several procedures to maintain the list data structure or to prepare the graph for the next step. Now we describe these procedures, `ColourBlue` in Figure 10, `ColourRed` in Figure 11, `MarkForDeletion` in Figure 12, and `Rewind` in Figure 13.

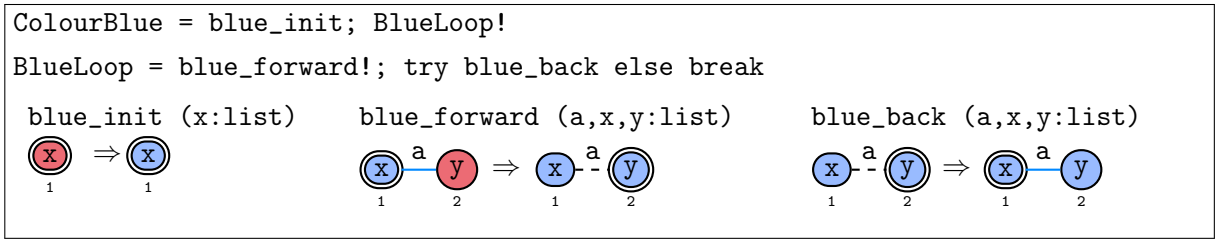


Figure 10: The procedure `ColourBlue`

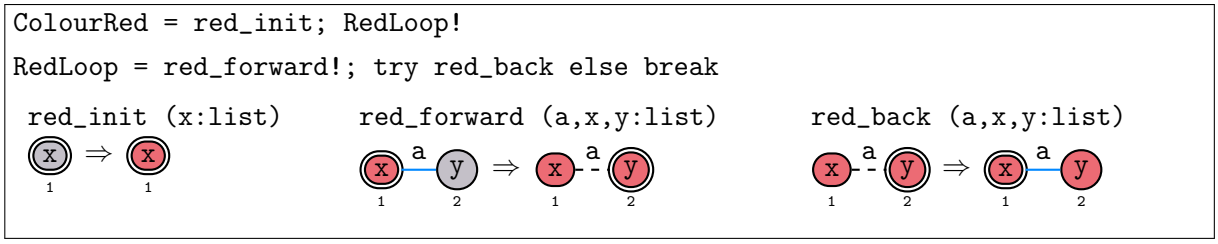


Figure 11: The procedure `ColourRed`

The procedure `ColourBlue` uses DFS to turn the nodes of a tree from red to blue, and the procedure `ColourRed` to turn the nodes of a tree from grey to red.

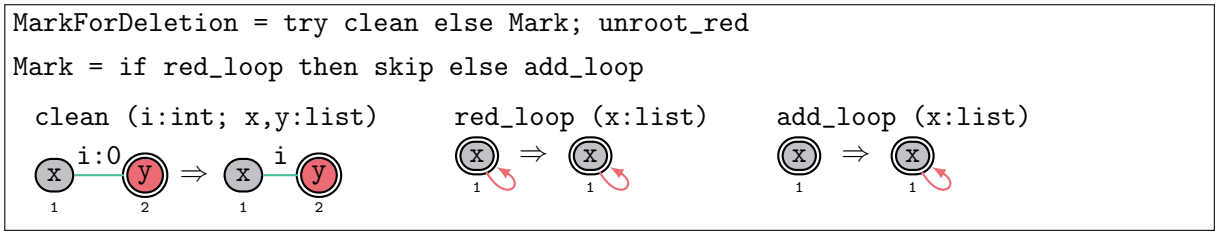


Figure 12: The procedure `MarkForDeletion`

The procedure `MarkForDeletion` determines whether the current tree needs to be removed from the list of trees or not. This needs to be done when the current tree is being merged with another tree in the procedure `GrowForest`. However, in a set of trees that are merged into one tree, exactly one of them needs to be kept as an entry in the list. This is done by exploiting the fact that exactly one of the green edges used to merge that set of trees must have been selected by two different trees. If none of the edges fulfil that condition, the merging would introduce a cycle. If multiple edges fulfil it, the trees are merged into a forest, and not a single tree. Hence the trees that select a previously selected edge are kept as an entry in the list. The rule `clean` easily detects these edges since their label is a list of their edge weight followed by a `0`.

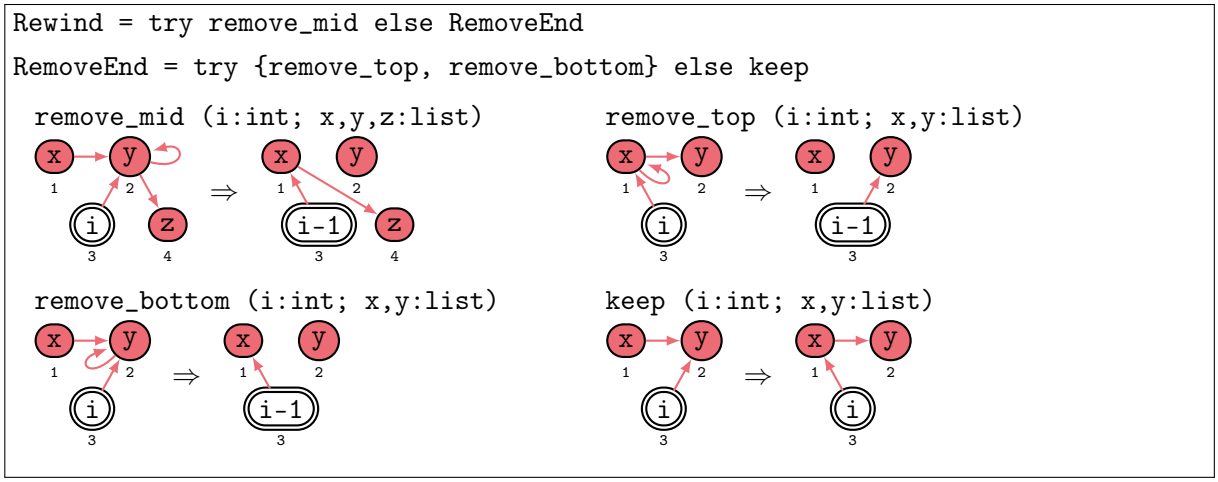


Figure 13: The procedure `Rewind`

The procedure `Rewind` returns the pointer to the beginning of the list of trees. On the way, it removes list entries marked for deletion with a red loop, and updates the pointer's label which represents the number of trees in the list.

4 Empirical Performance Results

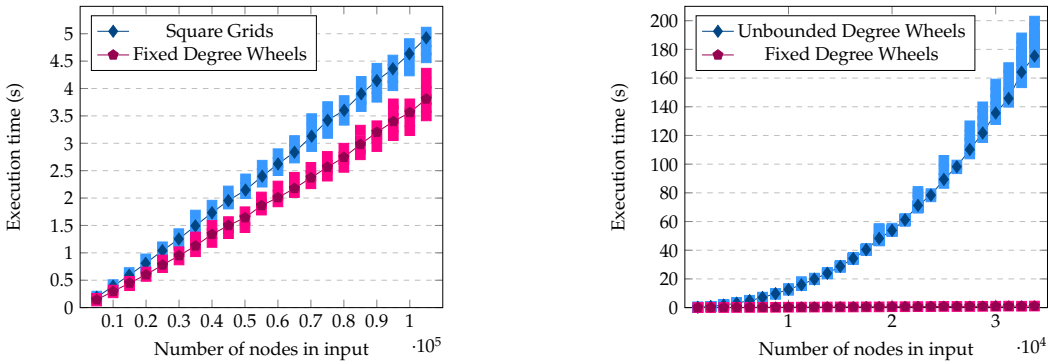
On the graph classes we tested, time measurements as illustrated in Figure 14 show subquadratic growth on square grids and fixed degree wheels, and polynomial growth on unbounded degree wheels.

The execution time of the program `mst-boruvka` has been measured on square grids, fixed degree wheels, and undbounded degree wheels. The k^{th} *square grid* is a $k \times k$ grid graph as depicted in Figure 3. Figure 2 depicts a wheel graph with 8 spokes. The k^{th} *fixed degree wheel* is a wheel graph with 16 spokes, each of which consist of a path graph with k edges. The k^{th} *undbounded degree wheel* is a wheel graph with k spokes.

The edge weights of the input graphs are randomly generated integers between 1 and 1000. The number of nodes of the square grids and fixed degree wheels ranges up to over 100000, and that of the undbounded degree wheels to almost 35000. For each graph of a given size, the execution time depicted with shapes is the average execution time of `mst-boruvka` on copies of that graph with at least 20 random weight distributions. The bars around those data points show the range between the minimum and maximum measured execution time for that graph. The extent of that range can be attributed to differing random weight distributions used for each time measurement. With a fixed weight distribution, that range is much smaller.

Figure 14a shows that `mst-boruvka` is subquadratic and close to linear on fixed degree wheels and square grids. We expect the time complexity to be $\mathrm{O}(m \log n)$, where m is the number of edges and n the number of nodes, akin to those of standard implementations of Boruvka's MST algorithm [20]. However, proving it will be left as future work. Note that, in order to reach this time complexity in GP 2, the use of root nodes is necessary.

In Figure 14b, `mst-boruvka` is seen to be of an order worse than $m \log n$ on unbounded degree wheels. In fact, we conjecture it to be quadratic. GP 2 programs that are non-destructive in that they preserve the input graph seem to require at least quadratic time on unbounded degree input graphs. For



(a) Bounded Degree Graphs (b) Wheels

Figure 14: Execution times of `mst-boruvka` with average

example, consider `MinWithN!` seen in Figures 7 and 8. In each case, it has to match a root (say u) and an adjacent non-root (say v) as long as possible. Assume that in the host graph, a root that is a valid match for u has a linear number of adjacent nodes, all of which are a valid match for v. Assume that the first time `MinWithN` is called, the node with the highest edge weight is matched as v. The program only needs to check one node since every node is a valid match. Then assume that the second time `MinWithN` is called, the node with the next highest edge weight is matched. In the worst case, two nodes have to be checked for a valid match. Summing up the number of nodes that are checked if we continue this pattern, we get a sum of consecutive integers with a linear number of terms, which is quadratic. Hence the quadratic time complexity.

Furthermore, procedures that are based on depth-first search and preserve their input, such as the procedure `ColourBlue` in Figure 10, have quadratic time complexity on unbounded degree graphs. The rule `blue_back` looks for a dashed edge around the blue root. Assume the blue root has degree that is linear in the number of edges of a graph class the input graph belongs to. Only one of its adjacent edges can be dashed since the dashed edges form a path from the blue root to the origin of the DFS. Since there's only one valid match for the dashed edge, the rule application takes linear time. Every node of the input has to play the role of the blue root in `blue_back` at some point.

The execution time on square grids is slower than that on fixed degree wheels by a constant factor. This is likely due to the fact that a large part of fixed degree wheels consists of path graphs, in which separate trees often share a minimum edge. So `MinWithN` is applied more often in fixed degree wheels than in square grids. Hence more rules (those of `MinWithN`) generally have to be called in square grids.

The time measurements were taken on a Lenovo Thinkpad T460 (2.4 GHz Intel Core i5, 16GB RAM) running Manjaro Linux, using the Python 3.8.3 time module. Exact figures of the time measurements can be found at `https://github.com/BrianCourtehoute/BrianCourtehoute.github.io/tree/master/PaperFiles/2020-06/Timings`, and the source program at `https://github.com/BrianCourtehoute/BrianCourtehoute.github.io/blob/master/PaperFiles/2020-06/Code/mst-boruvka.gp2`.

5 Conclusion and Future Work

This paper features an implementation of Boruvka's algorithm for computing minimum spanning trees in the rule-based graph programming language GP 2. We have presented empirical evidence for its time

efficiency on a few bounded degree graph classes that is in accordance with the known $O(m \log n)$ time complexity bound of Boruvka's algorithm implemented in conventional programming languages, where m is the number of edges and n the number of nodes. Furthermore, we have given empirical evidence for the program's non-linear time complexity on a graph class of unbounded degree.

The program is longer than its equivalent in conventional languages. The C implementation of Boruvka's algorithm presented by Sedgewick [18, 19] shown in Appendix A has 62 lines of code (not accounting for lines only consisting of brackets). A textual representation of the GP 2 program has 330 lines using a similar counting method. This is because, as an experimental language with a small syntax, GP 2 has no built-in data structures yet. Every data structure has to be implemented in the host graph itself. The procedures `Preprocess` and `Rewind` for instance only serve the purpose of creating and maintaining a list of trees.

Alternatively, one could omit counting the rule definitions in the GP 2 program, since rules are the basic operations of GP 2, and definitions of basic operations are not included in the line count of the C program. In that case, we count 30 lines in the GP 2 program, which is much more comparable to the length of the C code.

Due to the large number of procedures and rules, it can be rather challenging to understand how the program `mst-boruvka` operates while reading it (there seems to be a trade-off between efficiency and readability). This goes against the GP 2 design philosophy of facilitating formal reasoning, since proving soundness is not obvious. Indeed, giving a correctness proof in a formal proof system like in [21] would be a major undertaking. Hence the proof we plan to provide will not be in a formal proof system.

For the immediate future, we plan to write a longer version of this paper where we prove that $O(m \log n)$ is indeed a time bound of `mst-boruvka` on bounded degree graphs, and that the program indeed produces a minimum spanning tree of its input.

Another goal is to find more graph algorithms that can be implemented in GP 2 to reach their classical time bounds. We also plan to expand our technique for giving time measurements of a program's execution by comparing the timings of a GP 2 program to that of C code, and by using randomly generated inputs. Since the latter can produce wildly different timings however, it is not obvious how to do this in a sensible manner.

Finally, there is the open problem of how to create GP 2 programs that are efficient on unbounded degree graphs. There are GP 2 reduction programs in as yet unpublished work [7] that are efficient on arbitrary inputs. They operate by repeatedly removing nodes and edges from the host graph. This is a sensible approach for recognising whether a graph belongs to a certain graph class. However, when the purpose of a program is to produce a structure based on the input such as minimum spanning trees or topological sortings, this approach is not viable. It is not yet clear how to make these non-destructive programs efficient on arbitrary inputs.

References

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

[2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause & Gabriele Taentzer (2010): *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations*. In: *Model Driven Engineering Languages and Systems (MODELS 2010)*, *Lecture Notes in Computer Science* 6394, Springer, pp. 121–135, doi:10.1007/978-3-642-16145-2_9.

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

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

[5] Christopher Bak & Detlef Plump (2016): *Compiling Graph Programs to C*. In: *Proc. International Conference on Graph Transformation (ICGT 2016)*, *LNCS* 9761, Springer, pp. 102–117, doi:10.1007/978-3-319-40530-8_7.

[6] Cüneyt F. Bazlamaçcı & Khalil S. Hindi (2001): *Minimum-weight spanning tree algorithms A survey and empirical study*. *Computers & Operations Research* 28(8), pp. 767–785, doi:10.1016/S0305-0548(00)00007-1.

[7] Graham Campbell, Brian Courtehoute & Detlef Plump: *Fast Rule-Based Graph Programs*. Work in progress.

[8] Graham Campbell, Brian Courtehoute & Detlef Plump (2019): *Linear-Time Graph Algorithms in GP 2*. In: *Proceedings 8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, Leibniz International Proceedings in Informatics (LIPICS), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, pp. 16:1–16:23, doi:10.4230/LIPIcs.CALCO.2019.16.

[9] Graham Campbell, Jack Romö & Detlef Plump (2020): *The Improved GP 2 Compiler*. *ArXiv e-prints* arXiv:2010.03993. Available at `https://arxiv.org/abs/2010.03993`. 11 pages.

[10] Brian Courtehoute & Detlef Plump (2020): *A Fast Graph Program for Computing Minimum Spanning Trees*. In: *Proc. 11th International Workshop on Graph Computation Models (GCM 2020)*, pp. 165–183. Available at `http://www.cs.york.ac.uk/plasma/publications/pdf/Courtehoute-Plump.GCM.20.pdf`. Pre-proceedings version of this paper.

[11] Maribel Fernández, Hélène Kirchner, Ian Mackie & Bruno Pinaud (2014): *Visual Modelling of Complex Systems: Towards an Abstract Machine for PORGY*. In: *Proc. Computability in Europe (CiE 2014)*, *Lecture Notes in Computer Science* 8493, Springer, pp. 183–193, doi:10.1007/978-3-319-08019-2_19.

[12] Amir Hossein 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.

[13] Annegret Habel & Detlef Plump (2002): *Relabelling in Graph Transformation*. In: *Proc. International Conference on Graph Transformation (ICGT 2002)*, *Lecture Notes in Computer Science* 2505, Springer, pp. 135–147, doi:10.1007/3-540-45832-8_12.

[14] 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.

[15] 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.

[16] Detlef Plump (2017): *From Imperative to Rule-based Graph Programs*. *Journal of Logical and Algebraic Methods in Programming* 88, pp. 154–173, doi:10.1016/j.jlamp.2016.12.001.

[17] Olga Runge, Claudia Ermel & Gabriele Taentzer (2012): *AGG 2.0 — New Features for Specifying and Analyzing Algebraic Graph Transformations*. In: *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, *Lecture Notes in Computer Science* 7233, Springer, pp. 81–88, doi:10.1007/978-3-642-34176-2_8.

[18] Robert Sedgewick (1997): *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching*, 3rd edition. Addison-Wesley.

[19] Robert Sedgewick (2001): *Algorithms in C, Part 5: Graph Algorithms*, 3rd edition. Addison-Wesley.

[20] Steven S. Skiena (2008): *The Algorithm Design Manual*, 2nd edition. Springer, doi:10.1007/978-1-84800-070-4.

[21] Gia Wulandari & Detlef Plump (2020): *Verifying Graph Programs with First-Order Logic*. In: *Graph Computation Models (GCM 2020), Revised Selected Papers, Electronic Proceedings in Theoretical Computer Science* This volume.

A Boruvka's Algorithm in C

In this appendix, we include Sedgewick's C implementation of Boruvka's algorithm for the sake of comparison [18, 19].

Listing 1 contains the program computing a minimum spanning tree [19], which uses Union-Find as well as an adjacency list implementation of weighted graphs.

```
Edge nn[maxV], a[maxE];
void GRAPHmstE(Graph G, Edge mst[])
  { int h, i, j, k, v, w, N; Edge e;
    int E = GRAPHedges(a, G);
    for (UFinit(G->V); E != 0; E = N)
      {
        for (k = 0; k < G->V; k++)
          nn[k] = EDGE(G->V, G->V, maxWT);
        for (h = 0, N = 0; h < E; h++)
          {
            i = find(a[h].v); j = find(a[h].w);
            if (i == j) continue;
            if (a[h].wt < nn[i].wt) nn[i] = a[h];
            if (a[h].wt < nn[j].wt) nn[j] = a[h];
            a[N++] = a[h];
          }
        for (k = 0; k < G->V; k++)
          {
            e = nn[k]; v = e.v; w = e.w;
            if ((v != G->V) && !UFfind(v, w))
              { UFunion(v, w); mst[k] = e; }
          }
      }
  }
```

Listing 1: Boruvka's Algorithm

Listing 2 shows the interface of Union-Find [18].

```
void UFinit(int);
int UFfind(int, int);
int UFunion(int, int);
```

Listing 2: Union-Find ADT interface

Union-Find itself is shown in Listing 3 [18].

```
#include <stdlib.h>
#include "UF.h"
static int *id, *sz;
void UFinit(int N)
  { int i;
    id = malloc(N*sizeof(int));
```

```
    sz = malloc(N*sizeof(int));
    for (i = 0; i < N; i++)
      { id[i] = i; sz[i] = 1; }
  }
int find(int x)
  { int i = x;
    while (i != id[i]) i = id[i]; return i; }
int UFfind(int p, int q)
  { return (find(p) == find(q)); }
int UFunion(int p, int q)
  { int i = find(p), j = find(q);
    if (i == j) return;
    if (sz[i] < sz[j])
         { id[i] = j; sz[j] += sz[i]; }
    else { id[j] = i; sz[i] += sz[j]; }
  }
```

Listing 3: Union-Find ADT implementation

Listing 4 contains the implementation of weighted graphs using adjacency lists [19].

```
#include "GRAPH.h"
typedef struct node *link;
struct node { int v; double wt; link next; };
struct graph { int V; int E; link *adj; };
link NEW(int v, double wt, link next)
  { link x = malloc(sizeof *x);
    x->v = v; x->wt = wt; x->next = next;
    return x;
  }
Graph GRAPHinit(int V)
  { int i;
    Graph G = malloc(sizeof *G);
    G->adj = malloc(V*sizeof(link));
    G->V = V; G->E = 0;
    for (i = 0; i < V; i++) G->adj[i] = NULL;
    return G;
  }
void GRAPHinsertE(Graph G, Edge e)
  { link t;
    int v = e.v, w = e.w;
    if (v == w) return;
    G->adj[v] = NEW(w, e.wt, G->adj[v]);
    G->adj[w] = NEW(v, e.wt, G->adj[w]);
    G->E++;
  }
```

Listing 4: Weighted-graph ADT (adjacency lists)