# Visualizing and Simulating Algorithms using GROOVE

M.M.T.W. Heuvink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.m.t.w.heuvink@student.utwente.nl

## ABSTRACT

Algorithms can be quite difficult to understand. Traditional ways of presenting algorithms, such as by giving pseudocode or in natural language, are often not sufficient for a complete and fast understanding of the algorithm. GROOVE provides a means to simulate and animate an algorithm by using graph transformation rules. A library of GROOVE transformation rules can be an aid to understand algorithms more easily. User tests have been designed that can be used to assess the effectiveness of GROOVE in the better understanding of algorithms in comparison to traditional pseudocode.

## Keywords

Algorithm, Visualization, Simulation, GROOVE, Graph, Transformation, User Evaluation.

## 1. INTRODUCTION

The study of computer algorithms is a recurring subject in computer science studies and should be in the bag of every computer science student. New algorithms are being developed and existing ones revised. There are several ways to present algorithms to students, such as listing pseudocode or explaining its workings in natural language. However, *students find it difficult to understand certain algorithms.* The problem might lie in the representation method itself. One can experience difficulties visualizing the inner workings of an algorithm when studying pseudocode. Often a visual representation is needed. Furthermore, with multi-core computers becoming today's standard the study and use of distributed algorithms are becoming more and more important. These algorithms pose another problem. To fully understand distributed algorithms usually one cannot rely on solely understanding pseudocode. The workings of a distributed algorithm can often better be grasped by seeing the algorithm in action. The following example will illustrate this statement. One of the downsides of devising distributed algorithms is the threat of race conditions, which can occur on shared data. Process $x$ is about to increase a counter in the shared data, but that is the intention of process $y$ as well. If both processes increase the counter and some process $z$ wants to read the counter, $z$ may not read the expected value. Some locking mechanism is required. Such a race condition can be difficult to identify when studying pseudocode and can sometimes only be identified by simulation of the algorithm. Even in cases in which simulation may prove insufficient in identifying such race conditions, simulation of the algorithm can at least contribute to better understanding of the algorithm.


*19$^{th}$ Twente Student Conference on IT* June 24$^{th}$, 2013, Enschede, The Netherlands.


A way to study algorithms is to visualize and simulate them through *graphs*. An algorithm's input and the context on which that input is applied (e.g. some state of a datastructure) is represented by a starting graph $G$. One step in the algorithm implies a graph transformation rule $r$. Applying $r$ to graph $G$ will result in graph $G'$. Subsequently applying rules on $G'$ will result in a representation of the execution of the algorithm. A tool called GROOVE can be used for this purpose. GROOVE allows one to define graphs and transformation rules that act on them. Moreover, the tool has a function to automatically animate the transformations in the current graph.

Figure 1.1 depicts an example of a graph transformation rule created with GROOVE. It uses a pattern matching mechanism on the current state of the system (i.e. some graph) to decide which rule or subgraph to apply next. The transformation rule in the figure concerns inserting an integer element in a *binary search tree*. Actually, this rule is the last step of the algorithm: an element with value *val* is about to be inserted either at the left or right of some leaf of the tree. The element to be inserted is represented by the node named *other*. The thick lines in the graph emphasize creation, while the dashed lines emphasize deletion. In the example node *other* is inserted at the leaf node, depicted by the flag *isLeaf*. The rule application works as follows. First the expression `other.val < val` is evaluated. Only if the expression evaluated to `true` the *insert* edge is removed. The same way the *isLeaf* flag is deleted. Now node *other* becomes the leaf and is placed at the left of the old leaf. Note that if the expression `other.val < val` would evaluate to false, pattern matching would fail. In Figure 1.2 a graph is given that matches the the "addLeft" rule of Figure 1.1. As depicted in Figure 1.3, after application of the transformation rule, the *insert* edge has been removed and replaced by a *left* edge, because the value of the node that was inserted is smaller than the leafnode. Furthermore, the *isLeaf* flag is now part of the inserted node.

In this paper some algorithms that were simulated using GROOVE will be highlighted. For each algorithm a translation is made from part(s) of the pseudocode to their corresponding GROOVE graph transformation rules. Although the translations of the pseudocode have been made to look as natural and logical as possible, no reasoning about their correctness will be presented. *The goal of*

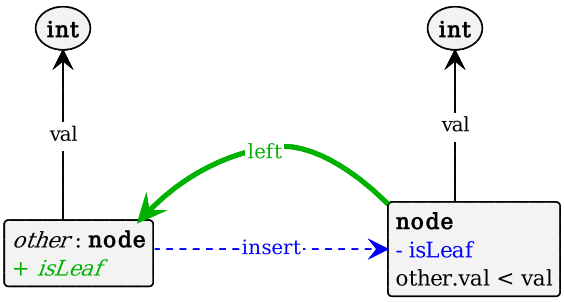



**Figure 1.1. The "addleft" rule of a binary search tree**

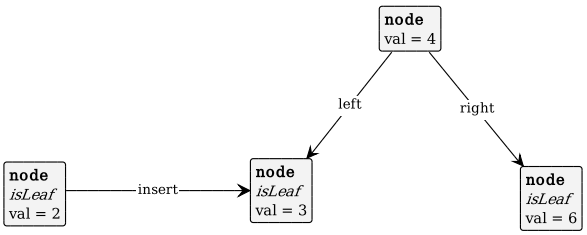



**Figure 1.2. A graph that matches the addleft rule**

*this research is to design a user evaluation study to assess GROOVE's capabilities* to present algorithms in a different, concise and clear way so that they can be used as an extension to the (formal) material that is readily available for those algorithms. To attain this goal user tests are presented to compare the GROOVE transformations against pseudocode and in some cases the formal mathematical representation of the algorithm. Participants complete a series of questions regarding each algorithm to assess their understanding of that particular algorithm. The results of those tests should adequately reflect the effectiveness of using GROOVE to aid in the understanding of the algorithms presented in this paper.

## 2. BACKGROUND

Visualization of algorithms is not a new subject. There are several tools that manipulate graphs and can be used to model an algorithm. An example given by Parduhn et al. [5] uses a program called LTVA Visualiser to visualize inserting a node in a binary search tree. The technique is based on shape analysis on shape graphs. There are other tools that act on graph besides GROOVE. For instance, PORGY is a graph-rewriting program that uses a strategy language for the application of rules [1]. However, the main reason of using GROOVE for graph transformations is that the rules can be created *graphically*. This fact can lead to easier understanding and faster creation of rules.

The part of GROOVE called GROOVE Simulator allows one to create graphs and transformation rules that act on them. For convenience, GROOVE Simulator is referred to as GROOVE in the remaining sections of this paper. The concept used by GROOVE will be concisely illustrated by the following example. A rule that deletes a certain node from graph $G$ should be modelled by taking a subgraph of $G'$ and defining some actions to create subgraph $H'$. An action in this example could be to delete a certain node in $G'$. GROOVE defines a set of actions that can be used in the transformation rules. These actions include: testing some expression, changing the fields in the nodes and edge relabelling. The tool also supports non-determinism.

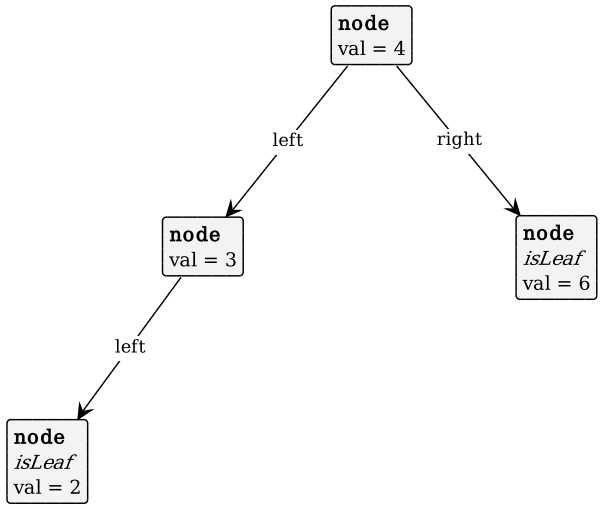



**Figure 1.3. The resulting graph after the rule application**

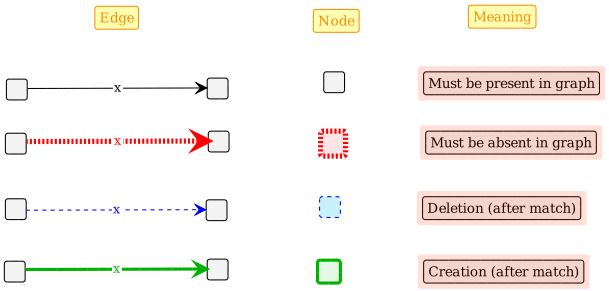



**Figure 2.1. GROOVE: basic syntax**

If at some point in the application of subsequent graph transformations more than one transformation could be applied, only one of those transformations is (randomly) chosen.

To simulate an algorithm with GROOVE one has to decide how to represent the various variables and data types in the algorithm. An integer could be represented by a single node. For a string there are multiple options. A single node representation could suffice for one algorithm, but a subtree that comprises multiple singe character nodes may be more useful for some other algorithm. Thus the choice of a data structure could prove essential for some algorithms to fully understand its concepts.

### 2.1 GROOVE Syntax

The basic syntax of GROOVE needed to understand the graph transformation rules given in this paper is shown in Figure 2.1. A short description of the function of the syntactical elements is also provided in the figure. GROOVE has some advanced feature that were used in some of the transformation rules in the examples. Those are listed in Figure 2.2. For a more extensive overview of the features and syntax of GROOVE one may consult GROOVE's user manual [6].

## 3. ALGORITHM SELECTION

This section gives an overview of some of the algorithms that have been implemented using GROOVE. However, only the parts of these algorithms that give key insights or otherwise seem worth mentioning are highlighted. Each subsection covers one algorithm and is divided into three

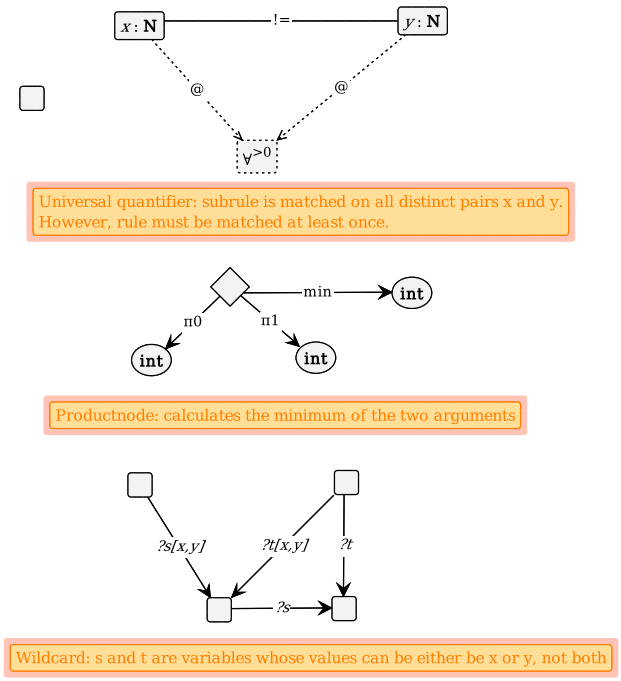



**Figure 2.2. GROOVE: more features**

parts. Firstly, the pseudocode of the relevant sections of the algorithm is given and shortly covered. Next, a GROOVE transformation rule concerned with the relevant sections of the pseudocode is depicted and also covered. The subsection ends with a comparison of GROOVE's implementation and the pseudocode. Furthermore, some new aspects or insights of the algorithm that GROOVE might have revealed is discussed.

## 3.1 Red-black trees

An important and efficient datastructure in computer science is a Red-black tree. A Red-black tree is in essence a binary-search tree with some unique properties. Although the structure of a Red-black tree is in general easily understood, the operations that can be applied to the tree may seem non-trivial. Inserting and deleting nodes are the two operations that can be applied to the tree. These operations are essentially the same as those for a binary search tree. The difference lies in the fact that in some cases a property of a Red-black tree is violated having applied such an operation. The algorithms concerned with restoring the properties are presented here and simulated by GROOVE. An example of a Red-black tree is given in Figure 3.1[1]. A concise summary of the properties of a Red-black tree taken from Cormen et al. [3] is presented below.

1. A node can be either red or black
2. The root is black
3. All the leafs are black
4. The children of a red node are black
5. Every path from a given node to any of its leaves contains the same number of black nodes.

[1]Source: http://en.wikipedia.org/wiki/Red-black_tree

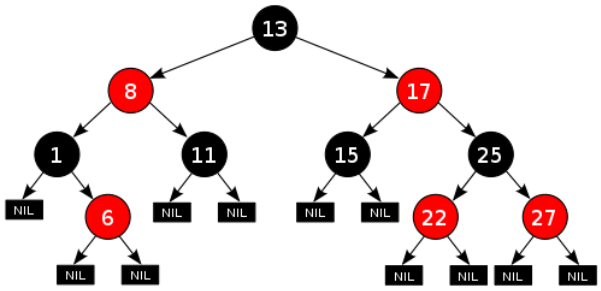



**Figure 3.1. A Red-black tree**

```
void insert_case5(Node n)
{
    Node g = grandparent(n);

    n.parent.color = Color.BLACK;
    g.color = Color.RED;
    if (n == n.parent.left)
        rotate_right(g);
    else
        rotate_left(g);
}
```

**Figure 3.2. Pseudocode: insertion case 5**

### 3.1.1 *Red-black tree insertion: pseudocode*

The pseudocode of the insert operation on a Red-black tree is given below. Five different cases that comprise the operation are generally distinguished. However, this paper covers only one of those cases.

### 3.1.2 *Red-black tree insertion: GROOVE*

In Figure 3.3 GROOVE's transformation rule is depicted for the pseudocode of the `insert_case5`-method. In fact, each case in the pseudocode corresponds to only one GROOVE transformation rule. This makes for simple and quick comparison.

### 3.1.3 *Comparison to pseudocode*

An insertion of some (random) node in a Red-black tree could result in the application of `insert_case5`. A tree example that triggers that method is given in Figure 3.4[1]. As shown in the figure, the tree on the left violates prop-

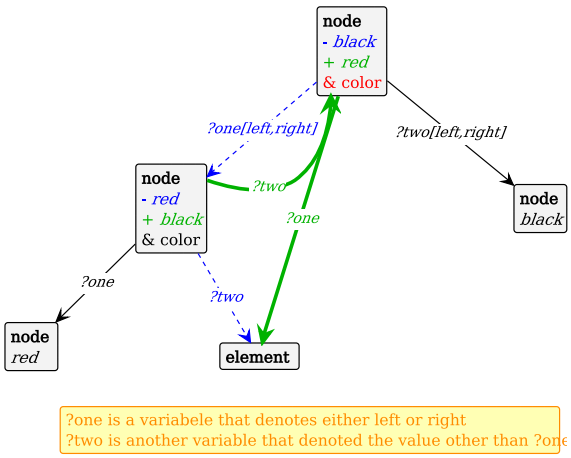



**Figure 3.3. Transformation rule: insertion case 5**

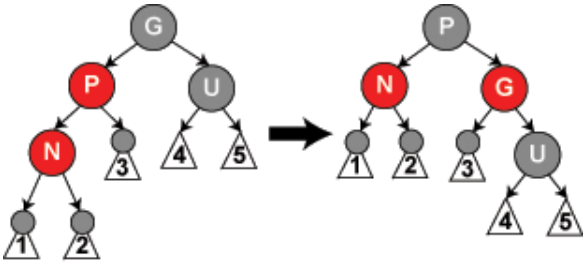



**Figure 3.4. Example insertion case 5**

erty 4 and needs to be rebalanced. After application of `insert_case5` the tree does not violate any property and insertion is finished.

The pseudocode of `insert_case5` is not hard to understand. It clearly states the steps to transform the tree on the left in Figure 3.4 to the one on the right. However, the code hardly triggers an easy visualization in one's mind. GROOVE's transformation rule in Figure 3.3 depicts exactly the same tree structure of the left tree in Figure 3.4. At first glance, when looking at the transformation in Figure 3.4, one might think the tree has been through some radical changes. However, GROOVE's rule shows that only 3 nodes are involved in the transformation and that only 2 of them have changed color. This pattern might not have been so easily identified using the pseudocode. However, due to the syntax and visual limitations (e.g. node coloring) of GROOVE, it might take some time to realize that the tree on the left in Figure 3.4 is similar to the tree pattern in Figure 3.3.

Because of GROOVE's ability to use variables on edges the transformation rule not only matches the left tree in Figure 3.4, but also the *reflected* version of the tree around the y-axis. The only difference in matching the reflected tree is the labels of the edges: the reflected tree contains opposite edge labels compared to the original tree. In the pseudocode of `insert_case5` the distinction between those two trees is made using the `if`-clause.

## 3.2 DFA minimization

Deterministic Finite Automata (DFAs) is a recurring subject in computer science studies and are often used to model some real-world concept. One can think of modelling a computer program or the behaviour of an elevator. A DFA $M$ is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where $Q$ denotes a finite set of states, $\Sigma$ a finite set of input symbols (the alphabet), $\delta$ a state transition function $\delta : Q \times \Sigma \to Q$, $q_0 \in Q$ the start state, and $F \subseteq Q$ a set of end states. Any DFA accepts some regular language $L$. Two DFAs are considered to be equivalent if they accept the same language $L$. A DFA is said to be minimal if it accepts $L$ with a minimum number of states.

Minimizing a given DFA $M$ consists of two steps:

1. Determinining the equivalent states of $M$
2. Merging the equivalent states into a new minimal DFA $M'$

In this paper, only the determinination of equivalent states (step 1) is covered.

### 3.2.1 Find equivalent states: pseudocode

The algorithm in pseudocode of Figure 3.5 finds the equivalent states of some DFA. The input of the algorithm is

input: DFA $M = (Q, \Sigma, \delta, q_0, F)$

1. (Initialization)
   **for** every pair of states $q_i$ and $q_j$, $i < j$, **do**
   1.1. $D[i, j] := 0$
   1.2. $S[i, j] := \emptyset$
   **end for**
2. **for** every pair $i, j$, $i < j$, **if** one of $q_i$ or $q_j$ is an accepting state and the other is not an accepting state, **then** set $D[i, j] := 1$
3. **for** every pair $i, j$, $i < j$, with $D[i, j] = 0$, **do**
   3.1. **if** there exists an $a \in \Sigma$ such that $\delta(q_i, a) = q_m$, $\delta(q_j, a) = q_n$ and $D[m, n] = 1$ or $D[n, m] = 1$, **then** $DIST(i, j)$
   3.2. **else** for each $a \in \Sigma$, do: Let $\delta(q_i, a) = q_m$ and $\delta(q_j, a) = q_n$
   **if** $m < n$ and $[i, j] \neq [m, n]$, **then** add $[i, j]$ to $S[m, n]$
   **else if** $m > n$ and $[i, j] \neq [n, m]$, **then** add $[i, j]$ to $S[n, m]$
   **end for**

   $DIST(i, j)$;
   **begin**
   $D[i, j] := 1$
   **for** all $[m, n] \in S[i, j]$, $DIST(m, n)$
   **end**

**Figure 3.5. Pseudocode: determination of non-equivalent states**

a DFA $M = (Q, \Sigma, \delta, q_0, F)$. The output is an array $D$, where $D[i, j] = 1$ means that state $i$ is considered to be *distinguishable* from state $j$. State $i$ and $j$ are considered equivalent if and only if $D[i, j] = 0$ upon termination of the algorithm. Part 3.1. will be covered in detail. That part tests for a pair of states $q_i$ and $q_j$ that are not yet distinguishable making a transition to their respective states $q_m$ and $q_n$ using the same inputsymbol. If pair $(q_m, q_n)$ has already been declared as distinguishable, then $(q_i, q_j)$ is also marked as distinguishable.

The code of Figure 3.5 is taken from Sudkamp's version of the DFA minimization algorithm[7, p. 179] and is not modified. This particular pseudocode is chosen because it is being used at the University of Twente and may be difficult to understand due to the formal language used.

### 3.2.2 Find equivalent states: GROOVE

Figure 3.6 depicts the GROOVE transformation rule that covers part 3.1. of the pseudocode. The rule matches any pair of two states (i.e. nodes with type `S` and identifiers `i` and `j`) that are distinct and have a transition to another distinct pair of states `m` and `n` with the same edge value (stored in variable `x`). State distinction in the figure is denoted by the `!=` edge between the states.

### 3.2.3 Comparison to pseudocode

GROOVE does not have support for 2-dimensional arrays. In the pseudocode, two of them are used. It seemed therefore important to represent those arrays in the transformation rule for the comparison to pseudocode to be as intuitive as possible. The pseudocode makes use of a 2-dimensional state-array $S$ and a 2-dimensional boolean-array $D$. In Figure 3.6 the $D$-array is represented as a node of type `D`. Setting the value `true` is modelled as two states that both have an `is`-edge to a `D`-node. In fact, they share a `D`-node. Setting `false` is represented by the fact

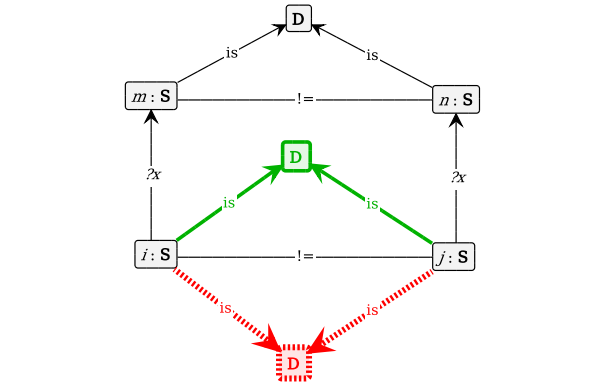



**Figure 3.6. GROOVE: determination of non-equivalent states**

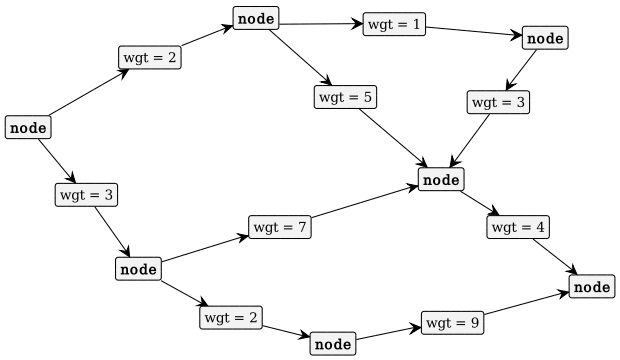



**Figure 3.7. GROOVE: weighted graph**

that the states do *not* share a `D`-node. The array $S$ in the pseudocode is not modelled in this manner. GROOVE does not make use of the values of $S$, but only uses the keys. The key $[i, j]$ of $S$ in the pseudocode is being visualized as a distinct pair of `S`-nodes in the figure.

The pseudocode of part 3.1. in Figure 3.5 also makes use of some *DIST*-function. *DIST*, together with the $S$-array improves the efficiency of the algorithm by storing intermediate states and recursively updating them. GROOVE's version does not use this explicit optimization, because this research does not concern efficiency of algorithms, solely their intuitive representation. The fact that the optimization is not represented by GROOVE could lead to better understanding of the algorithm. GROOVE shows exactly how the algorithm works, not how effeciently it works. However, GROOVE does use the first line in the *DIST*-function: $D[i, j] := 1$, namely marking state $i$ and $j$ as distinguishable.

Besides the differences discussed above the GROOVE transformation rule works exactly like the description given in Section 4.3.1.

## 3.3 Dijkstra's shortest path algorithm

An algorithm that might easily be represented by graphs is Dijkstra's shortest path algorithm. The algorithm operates on *directed*, *weighted* graphs, hence the choice for implementing GROOVE transformation rules that act on

```
function Dijkstra(Graph, source):
1.  for each vertex v in Graph:
        dist[v] := infinity;
        previous[v] := undefined;
    end for

    dist[source] := 0;
    Q := the set of all nodes in Graph;
2.  while Q is not empty:
        u := vertex in Q with smallest value
                in dist[];
        remove u from Q;
        if dist[u] = infinity:
            break;
        end if

3.      for each neighbor v of u:
            alt := dist[u] + dist_between(u, v);
            if alt < dist[v]:
                dist[v] := alt;
                previous[v] := u;
                decrease-key v in Q;
            end if
        end for
    end while
return dist;
```

**Figure 3.8. Pseudocode: Dijkstra's shortest path algorithm**

them. Dijkstra's algorithm calculates the shortest path costs of a given start node to any other node in some directed, weighted graph. The only condition is that the weights be nonnegative.

### 3.3.1 *Dijkstra path cost update: pseudocode*

The path costs from the start node to some node $v$ in the graph is stored in a `dist`-array. Note that this array only stores the path costs, not the paths themselves. Although the paths can easily be stored using the existing algorithm, they are not covered in this paper and modelled by GROOVE. In Figure 3.8[2] the pseudocode of the algorithm is given. Only the code within the `for each` loop at 3. is covered in this paper. In the loop, the minimal path costs for each neighbour `v` of `u` is calculated (i.e., `v` already has a shorter path or there exists a shorter path via `v`). In short, the path costs of `v` will become `min(dist[u] + dist_between(u, v), dist[v])`.

### 3.3.2 *Dijkstra path cost update: GROOVE*

The transformation rule that updates the path cost of all the neighbours of some node $v$ currently being inspected is depicted in Figure 3.9. As one can conclude from Figure 3.7 (from the 'path' node), GROOVE does not support weighted graphs.

### 3.3.3 *Comparison to pseudocode*

Dijkstra's algorithm works on graphs. Therefore, it seems logical that a graph transformation rule of the algorithm would be trivial to understand. However, the transformation rule of GROOVE shows quite a number of nodes to model the pseudocode. Although a closer look indeed reveals that the rule depicts the path cost update formula `min(dist[u] + dist_between(u, v), dist[v])`, it

[2]Source: http://en.wikipedia.org/wiki/Dijkstra's_algorithm

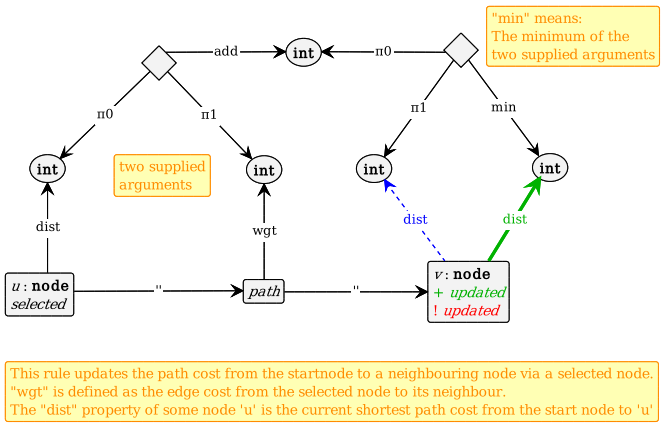



**Figure 3.9. GROOVE: updating the shortest path**

might be questionable that GROOVE's transformation rule is more easily understood than the pseudocode.

## 3.4 Other algorithms

There are other algorithms that have been implemented in GROOVE, but are not covered in some detail in this paper. These include sorting algorithms such as Bubblesort and Quicksort. However, in the design of the user evaluation study these algorithms are indeed covered. For a complete package of the GROOVE transformation rules of those algorithms and the additional transformation rules that were omitted in the coverage of the algorithms presented in this paper one may contact the author via e-mail.

# 4. USER EVALUATION

This section covers the design of a user evaluation concerning GROOVE's ability to aid in student's understanding of algorithms. Firstly, the test setup is elaborated on. Secondly, the test method that will be used is covered. Next, information is given on the interpretation of the data to be obtained. The question when logical conclusion can be drawn from the data is also covered. To conclude this section, possible design weaknesses and possible improvements are discussed that could lead to better test results.

## 4.1 The test setup

The user tests comprises one *controlled experiment* for each of the algorithms presented in this paper. A standard experiment design approach that is widely used in the field of psychology has been followed. The *experimental units* (i.e. the population) consists of undergraduate computer science students of the University of Twente. Those units have been divided between a *treatment group*, who will be working with GROOVE, and a *control group* working with solely pseudocode. For the experiment to yield representative results *random assignment* of the units to one of the groups was the obvious choice. The randomness of the selection should ensure that on average the subjects have equal existing knowledge of the algorithms and equal ability to study them.

Each test is concerned with an algorithm that is recently presented to the subjects in some course that they were taking at that time. Therefore, the assumption that the subjects have some existing knowledge of the algorithms presented is justified.

Firstly, a small presentation will be given to both the control group and the treatment group (at the same time) about the syntax of GROOVE. Besides the fact that only one group will be actually working with GROOVE, the groups should be treated as equally as possible for the experiment to yield reliable results. The duration of the presentation is 10 minutes. Then, both groups are given a handout that initially explains in short which algorithm they will be studying. On the handout for the *control group* the algorithm is presented in pseudocode. The subjects are given 15 minutes to study the pseudocode in detail. Afterwards the subjects will take part in a test that comprises questions regarding the algorithm studied. The test is used to assess the subject's understanding of the algorithm in question.

The *treatment group* will be given the same handout as the control group and the subjects too will study the pseudocode. However, there is an important difference. The subjects are only given 5 minutes to study the code. Afterwards, they receive an additional handout that explains the GROOVE syntax needed for understanding the transformation rules of the algorithm being studied. After the syntax is fully understood, the subjects are placed in front of a computer that is running the GROOVE Simulator program of the algorithm. The subjects are given 10 minutes to study the transformation rules and to relate those rules to the corresponding parts of the pseudocode. Next, the treatment group will also take part in the test regarding their understanding of the algorithm.

For both the treatment and control group a questionnaire is given afterwards. The purpose of the questionnaire is to obtain additional data besides the data from the answers to the questions regarding the algorithms. The questions in the questionnaire for the control group give regard to the algorithm before and after they studied the pseudocode. Also, the amount of existing knowledge of the algorithm and the need for additional material to aid them in better understanding of algorithms are part of the questions.

The questions of the treatment group are the same as the control group. However, an additional questionnaire that relate to GROOVE will be presented. The questions are concerned with the use of GROOVE as an aid to a better understanding of the algorithm studied, the GROOVE syntax in general, and the ability of the subjects to link the transformation rules to their corresponding parts of the pseudocode. The questionnaires can be found in Appendix A. The test questions for each algorithm is presented in Appendix B. The answers for each of the test questions has been marked by an asterisk (*).

## 4.2 The test design

Considering the fact that the users are split up into a control group and an experimental group and assuming a normal distribution and variance of the population it is advisory to follow the commonly used unpaired 2-sample t-test (Student's t-test) as the test method of choice [4, p. 207]. The fact the test is unpaired follows from the fact that the control and treatment groups are tested independently of each other and no group is tested twice. Furthermore, it is assumed that equal samples size for both groups are used. The sample size estimation is given below. Moreover, to aid in the normal distribution of the population, it is advisory to use solely first-year students in the samples.

Small tests already conducted on users (but not covered in this paper) indicate that if GROOVE would make a difference in understanding algorithms that difference would not be large. Taking that property into account, a statistical effect size of 0.5 is required, i.e. the tests should yield a moderate difference between the control group and the experimental group. Consulting Cohen's sample size table for 2-sample t-tests [2] and requiring a universally accepted statistical power of 0.8 one needs to gather a sample size of 64 for each group. If one wants to test for a *slight* difference between the groups an even larger sample size of 393 is needed per group [2]. However, such a large sample may not be attainable due to the limited number of first-year students. The data to be gathered consists of the points scored on the test questions for each algorithm (see Appendix B for the point distribution). The *subjective* data gathered by the questionnaires can be used to assess, for instance, the feelings towards GROOVE and towards GROOVE as an aid in the understanding of algorithms. However, only the analysis of *objective* data gathered from the test questions is covered.

## 4.3 Analysis of results

As stated in the previous section it is assumed that the groups are equal in size and in variance. Using the scored points as the data to analyse and following the 2-sample t-test approach, the $t$-statistic can be identified using the following formula [4]:

$$t = \frac{\bar{X_1} - \bar{X_2}}{\sqrt{\frac{1}{2}(S^2_{X_1} + S^2_{X_2})}\sqrt{\frac{2}{n}}}$$

In this formula, $\bar{X_1}$ is the mean value of group 1, $n$ the number of participants, and $\sqrt{\frac{1}{2}(S^2_{X_1} + S^2_{X_2})}$ the pooled standard deviation. Consulting the appropriate t-table using the value $t$ will give the answer whether there exists a moderate difference between the groups.

## 4.4 Possible design weaknesses

The user evaluation might not yield good results only by analysing the total points scored by the subject. The analysis might be improved by including data that is concerned with the time in which subjects complete a test question of some algorithm. This data can be added to the existing test scores and may lead to better results. Another consideration to take into account is whether to use the full set of GROOVE's transformation rules for an algorithm. It might be the case that the user evaluation yields better results if only some carefully chosen transformation rules (known to simplify complex pseudocode) are included in the material of the experiment. Lastly, the learning curve of GROOVE might have a significant impact on the test results. The subjects in the treatment group might not be sufficiently familiar with GROOVE's syntax and semantics in the 10 minutes they are allowed to work with GROOVE. In that case, the test results might reject the hypothesis that the use of GROOVE makes a moderate difference in understanding algorithms better. A possible improvement might be to extend the duration of the initial presentation about GROOVE (given to both groups) until the subjects are familiar with GROOVE.

# 5. CONCLUSION

The designed user tests that have been shown in this paper can be used to assess wheter there exists a moderate (positive) difference between the treatment group and the control group. In other words, GROOVE's capabilities to act as an appropriate tool to help students understand algorithms better can be tested. The test results might indicate that GROOVE is helpful only for some algorithms. However, if the results yield that in most cases GROOVE does not make a difference in the understanding of the algorithm, one should carefully consider possible design weaknesses. Another consideration is to increase the sample size. When using a larger sample size one can test for a smaller difference between the groups.

# 6. FUTURE WORK

There exists more classes of algorithms that can be simulated using GROOVE, such as the class of distributed algorithms. Simulating various processes that uses certain datastructures in a distributed algorithm using graph transformations might be an intuitive aid in the understanding of such an algorithm. The user evaluation study of such distributed algorithms will in principle not be any different from the one conducted in this paper.

Future work might also be triggered by the continuous development of GROOVE. For example, GROOVE is often not suited for proper animation of the algorithms, because it does not support programming of graph layout changes. If such a feature would be implemented, the effectiveness of GROOVE as an aid in understanding algorithms might be increased. In that case, new user evaluations would be needed to support this claim.

# 7. ACKNOWLEDGEMENTS

Firstly, two people have continuously given their support in the process of the research conducted in this paper. Therefore, gratitude goes to dr. M.M. Fokkinga (as the instructor) and prof.dr.ir. A. Rensink (as the founder of the research topic). Furthermore, all the people who reviewed this paper are worth mentioning.

# 8. REFERENCES

[1] O. Andrei et al. Porgy: Strategy-driven interactive transformation of graphs. In *6th International Workshop on Computing with Terms and Graphs*, pages 54–68, 2011.

[2] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, New York, 1977.

[3] T. Cormen et al. *Introduction to Algorithms*. The MIT Press, Cambridge, second edition, 2001.

[4] D. Kenny. *Statistics for the Social and Behavioral Sciences*. Little, Brown and Company Ltd., Boston, 1987.

[5] S. Parduhn et al. Algorithm visualization using concrete and abstract shape graphs. In *SoftVis '08 Proceedings of the 4th ACM symposium on Software visualization*, pages 33–36, 2008.

[6] A. Rensink et al. User manual for the groove tool set. April 2007.

[7] T. Sudkamp. *Languages and Machines*. Addison Wesley, New York, third edition, 2006.

# APPENDIX

# A. QUESTIONNAIRES

## A.1 Standard questionnaire

Algemeen

**Aantal jaren dat u Informatica studeert: _**
**Leeftijd _**
**Geslacht** man/vrouw

Inhoudelijk

**Hoeveel procent van het algoritme kende u op voorhand al?**
□ 0-20%
□ 20-40%
□ 40-60%
□ 60-80%
□ 80-100%

**U heeft 15 minuten de tijd gekregen het algoritme te bestuderen. Hoeveel procent van de code hebt u volledig kunnen begrijpen?**
□ 0-20%
□ 20-40%
□ 40-60%
□ 60-80%
□ 80-100%

In hoeverre bent u het met de volgende uitspraken eens?

**Het te bestuderen materiaal was voldoende om het algoritme volledig te begrijpen**
Helemaal mee oneens □ □ □ □ Helemaal mee eens

**Het vormde voor mij geen probleem om de interne werking van het algoritme te visualiseren**
Helemaal mee oneens □ □ □ □ Helemaal mee eens

## A.2 Questionnaire concerning GROOVE

In hoeverre bent u het met de volgende uitspraken eens?

**De graaf-transformatie regels hebben mij een helderder beeld gegeven bij de werking van het algoritme**
Helemaal mee oneens □ □ □ □ Helemaal mee eens

**De GROOVE transformatieregels waren eenvoudig te begrijpen**
Helemaal mee oneens □ □ □ □ Helemaal mee eens

**De relatie tussen een GROOVE transformatieregel en een (of meerdere) stap(pen) uit de pseudocode kon ik gemakkelijk en vrijwel direct leggen**
Helemaal mee oneens □ □ □ □ Helemaal mee eens

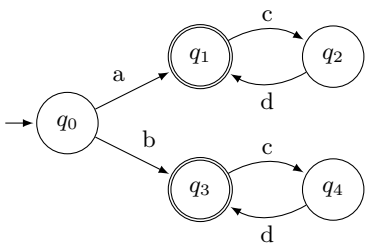
**Door de GROOVE transformatieregel in actie te zien met behulp van de GROOVE Simulator kon ik een beter beeld krijgen van de interne werking van dat specifieke deel van het algoritme.**
Helemaal mee oneens □ □ □ □ Helemaal mee eens

# B. TEST QUESIONS

## B.1 DFA Minimization Questions

**Question 1**:

Consider the following DFA:



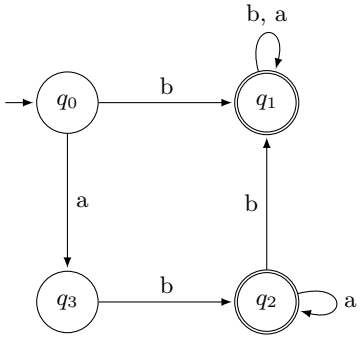

Clearly, $q_1$ and $q_3$ are equivalent states. Furthermore, $q_2$ and $q_4$ and are equivalent states. However, the equivalence of the pair $(q_2, q_4)$ depends on the equivalence of $(q_1, q_3)$. But the equivalence of $(q_1, q_3)$ in turn depends on the equivalence of $(q_2, q_4)$ again. These facts suggest an infinite loop.

Explain why in general the algorithm studied does not end up in an infinite loop while determining equivalent states.

*Answer: *The algorithm visits the pairs of nodes only once. It only revisits if some pair of nodes is distinguishable, which in this example does not happen (each of the two pairs cannot be set as distinguishable in the first visit, because they are dependent of each other)*

**Question 2**

Consider the following DFA:



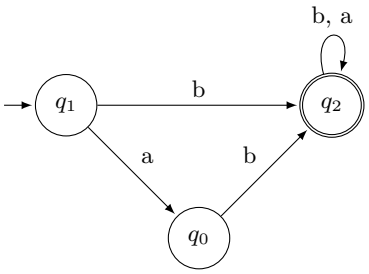

Which pair(s) of states (if any) will be determined equivalent (non distinguishable) after the execution of the algorithm?

A $(q_1, q_2)$ and $(q_0, q_3)$
*B $(q_1, q_2)$
C $(q_0, q_3)$
D no pair can be determined

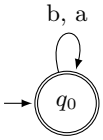
**Question 3**

If asked to minimize the DFA of Question 2, which of the following automata would be the result?

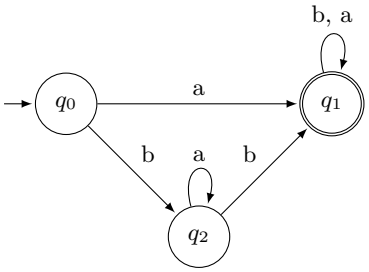
*A



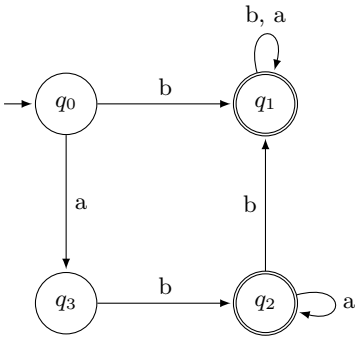

B




C




D




## B.2 Bubblesort Questions

**Question 1**:

Using Bubblesort, write down the result of the 3rd *bubble* operation using the following initial list of numbers:

| 2 | 9 | 3 | 7 | 11 | 4 | 1 | 15 | 8 | 13 |

| — | — | — | — | — | — | — | — | — | — |

*Answer:
| 2 | 3 | 4 | 1 | 7 | 8 | 9 | 11 | 13 | 15 |

**Question 2**:

Using Bubblesort, how many iterations are needed to fully sort the following initial list of numbers?

| 1 | 2 | 3 | 7 | 11 | 14 | 12 | 15 | 8 | 13 |

A 2
B 3
*C 4
D 5

## B.3 Quicksort Questions

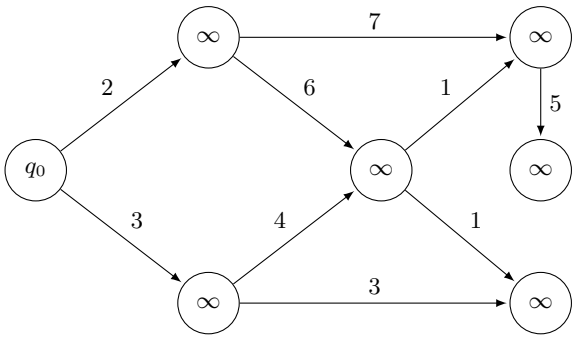
The same questions used for Bubblesort will be asked for Quicksort, but with different lists of numbers.

## B.4 Dijkstra's Algorithm Questions

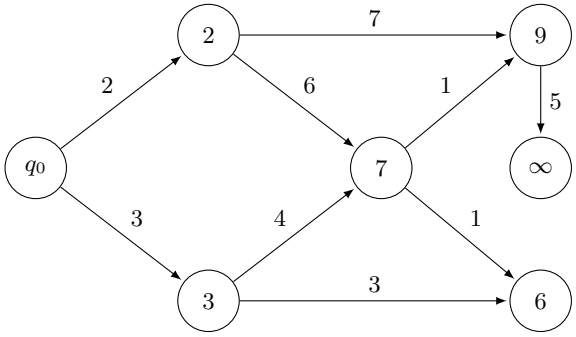
**Question 1** [1 point]

Consider the following weighted graph $G$:

Use Dijkstra on node $q_0$ and fill in the path costs from $q_0$ to all other nodes of $G$ after the 4th iteration in the `while`-loop.




Answer:
*The following graph:*




*If one or more path costs are incorrect: 0 points.*

**Question 2** [1 point]

Why, in general, does Dijkstra's algorithm work only on graphs with the property: for all edge weights $w$: $w \geq 0$?

*Answer: *Dijkstra's algorithm is classified as a greedy algorithm. The fact that from node $u$ it picks some neighbouring node $v$ with the smallest $w(u, v)$ [0.5 points] as the next node in the path guarantees that there exists no other path from $u$ to $v$ with a smaller path cost to $v$. This can only be guaranteed if all weights are nonnegative. [0.5 points]*

**Question 3** [1 point]

To clarify your answer for Question 2, draw some graph $G'$ that does not fulfill the property of Question 2 and mark a path from some start node to some other node for which Dijkstra's algorithm does not yield the smallest path cost.

*Answer:
*Some path on a graph with negative edge weights that fulfills the above requirement [1 point]*

## B.5 Red-black Trees Questions

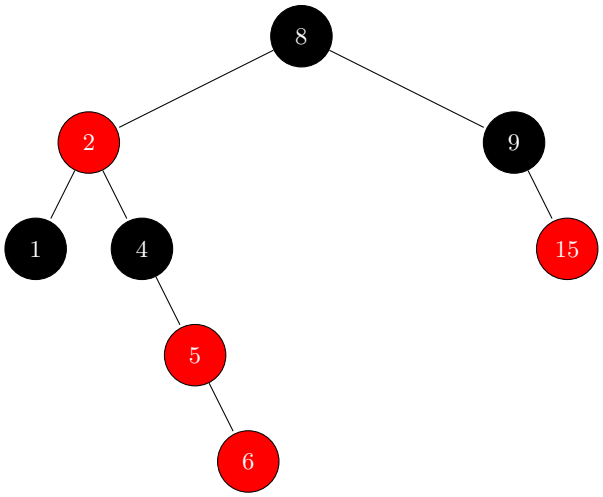
**Question 1** [1 point]

Explain why application of case 5 does not violate property 5 for Red-black trees.

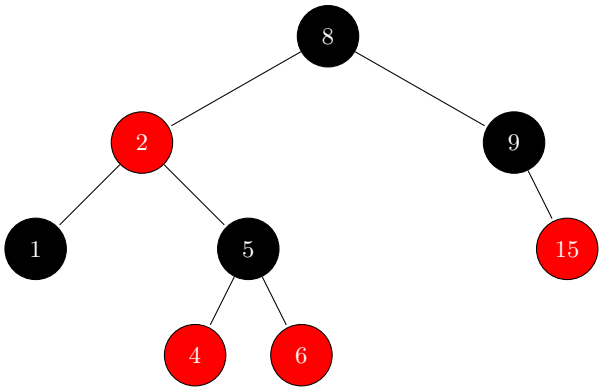
*Answer:
*The inserted node will always be red. If the tree before the application did not violate any property, then property 5 will not be violated. The number of black nodes will remain the same, regardless of an insertion of any red node [1 point].*

**Question 2** [1 point]

The Red-black tree below clearly violates property 4. Apply the appropriate transformations to the tree and draw the resulting Red-black tree so that everry property is satisfied.




*Answer:
*The following Red-black tree:*




*If resulting tree differs in any way: 0 points.*