Towards Automatic Advice in Visual Algorithm Simulation

Artturi Tilanterä

School of Science

Thesis submitted for examination for the degree of
Master of Science in Technology.
Helsinki 26.4.2020

Supervisor

Senior University Lecturer
Ari Korhonen

Advisor

Otto Seppälä, D.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of the master's thesis

Author Artturi Tilanterä

Title Towards Automatic Advice in Visual Algorithm Simulation

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Senior University Lecturer Ari Korhonen

Advisor Otto Seppälä, D.Sc. (Tech.)

Date 26.4.2020 **Number of pages** 85 **Language** English

Abstract

Visual Algorithm Simulation (VAS) exercise is an interactive application which teaches an algorithm or a data structure. The exercise shows the student a visual representation of a data structure with initial data. The student imitates the execution of the algorithm by interacting with the visual representation. The student's solution is graded automatically. A misconception about the algorithm being learned can manifest itself as systematic errors which can be modelled as a new algorithm. It is assumed that a VAS exercise which could detect automatically a misconception and give corrective feedback would further support learning.

This thesis includes a literature review on algorithm misconceptions and an empirical study. Four VAS exercises of OpenDSA e-textbook are reviewed by their program code: Evaluating a postfix expression, Build-heap, Quicksort, and Dijkstra's algorithm. A dataset of 1430 Build-heap VAS submissions is analysed manually with ad-hoc software. The submissions are then automatically classified based on the misconceptions found.

The main result extends the set of known misconceptions of the Build-heap VAS exercise. 52% of the submissions were correct, 17% were misconceptions and the rest 31% had a logical explanation. 95% of submissions classified as misconception have multiple explanations with heap size of 10. The thesis presents a Python software which can automatically classify the known misconceptions. Theory on how to generate VAS inputs which support detection of misconceptions is discussed. The theory is applied by improving the input generation of Dijkstra's algorithm exercise.

The thesis concludes that studying misconceptions in the VAS exercises of OpenDSA currently requires exercise-dependent work. Not all OpenDSA VAS exercises record enough data for later analysis. Moreover, the player and analysis software must be written separately for each exercise. There is need to develop the OpenDSA-related software libraries to produce detailed exercise recordings. It should be studied how the heap size in the Build-heap exercise affects detection of misconceptions.

Keywords computing education, algorithm visualisation, automatic assessment, misconception

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Artturi Tilanterä

Työn nimi Kohti automaattisia vihjeitä visuaalisessa algoritmisimulaatiossa

Koulutusohjelma Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

Pääaine Tietotekniikka **Pääaineen koodi** SCI3042

Työn valvoja Vanhempi yliopistonlehtori Ari Korhonen

Työn ohjaaja TkT Otto Seppälä

Päivämäärä 26.4.2020 **Sivumäärä** 85 **Kieli** Englanti

Tiivistelmä

Visuaalinen algoritmisimulaatiotehtävä (VAS-tehtävä) on vuorovaikutteinen sovellus, joka opettaa algoritmin tai tietorakenteen. Tehtävä näyttää opiskelijalle kuvan tietorakenteesta lähtödatalla. Opiskelija mukailee algoritmin suoritusta vuorovaikuttamalla kuvaesityksen kanssa. Opiskelijan ratkaisu arvostellaan automaattisesti. Väärinkäsitys VAS-tehtävässä on opiskelijan järjestelmällinen väärinymmärrys, joka voidaan kuvata algoritmilla. On oletus, että VAS-tehtävä, joka tunnistaisi automaattisesti väärinkäsityksen ja antaisi korjaavaa palautetta, tukisi oppimista entisestään.

Tämä opinnäytetyö sisältää kirjallisuustutkimuksen algoritmien väärinkäsityksistä sekä empiirisen tutkimuksen. Sähköisen OpenDSA-kirjan neljä VAS-tehtävää on tutkittu niiden ohjelmakoodiltaan: Postfix-lausekkeen evaluointi, binäärikeon rakentaminen, pikajärjestäminen ja Dijkstran algoritmi. Tietoaineisto, jossa on 1430 tallennetta Binäärikeon rakentaminen -tehtävästä, on analysoitu käsin tätä varten kehitetyllä ohjelmalla. Tallenteet on sitten automaattisesti luokiteltu löydettyjen väärinkäsitysten perusteella.

Työn päätulos laajentaa binäärikeon rakentaminen -tehtävän väärinkäsityksien joukkoa. 52 % tehtäväpalautuksista oli oikein, 17 % väärinkäsityksiä ja loput 31 % voidaan selittää loogisesti. 95 %:lla niistä palautuksista, jotka luokiteltiin väärinkäsitykseksi, oli useampi yhtä hyvä selitys, kun keon koko oli 10. Työ esittää Python-ohjelman, joka voi automaattisesti luokitella tunnettuja väärinkäsityksiä. Työ esittää myös teoriaa, kuinka tuottaa VAS-tehtävien lähtödataa siten, että se tukisi väärinkäsitysten tunnistamista. Teoriaa on sovellettu parantamalla Dijkstran algoritmi -tehtävän syötteen tuottamista.

Johtopäätöksenä OpenDSA:n VAS-tehtävien väärinkäsitysten tutkiminen vaatii nykyisellään tehtäväkohtaista työtä. Kaikki OpenDSA:n VAS-tehtävät eivät tallenna riittävästi dataa myöhempää analyysiä varten. Lisäksi tehtävätoistin ja analyysiohjelma pitää kirjoittaa erikseen joka tehtävälle. On tarve kehittää OpenDSA:n ohjelmakirjastoja tuottamaan yksityiskohtaisia tehtävätallenteita. Binäärikeon rakentaminen -tehtävässä pitäisi tutkia keon koon vaikutusta väärinkäsitysten tunnistamiseen.

Avainsanat tietotekniikan opetus, algoritmien havainnollistus, automaattinen arviointi, väärinkäsitys

Acknowledgements

I want to thank Senior University Lecturer Ari Korhonen for supervising the thesis. He first hired me as a Teaching Assistant for the Data Structures and Algorithms Y course in 2017 and then allowed me to do develop the course software. For this thesis, Ari provided valuable practical insight, such as the grid-based approach for input generation for the Dijkstra's algorithm exercise, and multiple literature references. His calm guidance ensured that this work eventually converged into a Master's thesis.

I also want to thank University Lecturer Otto Seppälä for providing practical and literature advice for the thesis, especially the encouragement to study the problem of input generation from software testing perspective. Otto's cornucopia of ideas and playful thinking provided many inspiring discussions which show paths for future research far beyond of this thesis.

The Aalto University Department of Computer Science has generously funded this thesis. I wish the work I have constructed will soon benefit directly other students. Advancing research in Learning + Technology group has been a privilege for its relaxed atmosphere, interesting discussions and possibilities to develop new, cutting-edge technology.

Professor Lauri Malmi was also interested in the progress and provided me with fruitful material on misconceptions [36]. Thanks to him, the Discussion section has general-purpose ideas for misconception mining which might help the future work with other exercises than Build-heap.

Lecturer Lassi Haaranen analysed the software architecture of the VAS exercises on the Aalto University course "Data Structures and Algorithms Y", which helped the exploration of the OpenDSA program code.

Professor Mario Di Francesco and Daniel Bruzual Balzan for provided the original code for JSAV downloader. Jaakko Kantojärvi introduced me to the A+ API. Without you three the data analysis would have been much harder.

I want to thank Viivi and Gabriell for cheering me up and reminding that there is life outside work: drawing, fiction, music, adventures, non-engineering issues, and you two. Also Ruoste and Aino provided valuable insight to the importance of catness, the entertainability estimates of various household objects, and doing nothing.

Finally, I want to thank Merja and Teemu for their continuous support throughout my life, even when it did not seem to proceed linearly. It has been a privilege to study computing and energy sciences, and I wish the knowledge I possess benefits future students. I am also thankful to Miranda for taking me out in the woods once in a while, to appreciate the complexity of the inartificial realm.

Helsinki, 31.3.2020

Artturi Tilanterä

Contents

Abstract 2

Abstract (in Finnish) 3

Acknowledgements 4

Contents 5

Abbreviations and Acronyms 7

1 Introduction 8
- 1.1 Motivation . . . 8
- 1.2 Objectives . . . 10
- 1.3 Research questions and methods . . . 11
- 1.4 Scope and structure . . . 11

2 Background 13
- 2.1 Pedagogics of VAS and misconceptions . . . 13
- 2.2 Literature study . . . 15
- 2.3 Algorithmic detection of known misconceptions . . . 17
- 2.4 Misconception-aware input in VAS . . . 19
- 2.5 Software testing perspective . . . 20
  - 2.5.1 Relevant testing terminology . . . 21
  - 2.5.2 Test data generation by symbolic execution . . . 22
- 2.6 Software context . . . 24
- 2.7 String matching for VAS . . . 24

3 Methodology 28
- 3.1 System overview with risk analysis . . . 28
- 3.2 The essential features of an algorithm . . . 33
  - 3.2.1 Evaluating Postfix expression . . . 34
  - 3.2.2 Quicksort . . . 34
  - 3.2.3 Build-heap . . . 35
  - 3.2.4 Dijkstra's algorithm . . . 38
- 3.3 Reproducibility of JSAV exercise recordings . . . 43
  - 3.3.1 Evaluating postfix expression . . . 44
  - 3.3.2 Quicksort . . . 44
  - 3.3.3 Build-heap . . . 44
  - 3.3.4 Dijkstra's algorithm . . . 45
- 3.4 Improved graph algorithm exercises . . . 46
- 3.5 Input generation by random search . . . 50

4 Empirical methods and data 52
4.1 Replication as a scientific method 52
4.2 Direct replication of the Build-heap studies 52
4.3 Forming hypotheses for Build-heap misconceptions 55
4.4 Machine classification of new Build-heap misconceptions 57
4.5 Software tools for misconception study 58

5 Empirical results 61
5.1 Direct replication of the Build-heap studies 61
5.2 Forming hypotheses for Build-heap misconceptions 63
5.3 Machine classification of new Build-heap misconceptions 65

6 Discussion 68
6.1 Contributions 68
6.2 Evaluation 68
6.3 Discussion 69
6.4 Recommendations 71
6.4.1 Improving the Build-heap VAS exercise 71
6.4.2 Further research questions 72

A VAS exercises on the Data Structures and Algorithms course 74

B Direct replications of Build-heap study 75

C Details for alternative Build-heap hypothesis 76

D Revised sequence similarity algorithm 79

Abbreviations and Acronyms

A+ LMS	A+ Learning Management System
AMM	Algorithmic Misconception Model
CS	Computer Science
CS2	Computer Science 2
CSS	Cascading Style Sheets
DFS	Depth-First Search
DSA	Data Structures and Algorithms
E-learning	Electronic Learning
FIFO	First In, First Out
GUI	Graphical User Interface
HTML5	HyperText Markup Language
IR	Input Requirement
JSAV	JavaScript Algorithm Visualization Library
JSON	JavaScript Object Notation
LMS	Learning Management System
LR	Left to right
RL	Right to left
RR	Reproducibility Requirement
RQ	Research Question
SV	Software Visualisation
TRAKLA2	a VAS software
VAS	Visual Algorithm Simulation
VPS	Visual Program Simulation

1 Introduction

This section explains briefly the purpose and the subject of study of this thesis. It refers to the relevant concepts and scientific disciplines. The research objective and scope are defined in this section.

1.1 Motivation

The purpose of this thesis is to improve computing education at higher education level. Aalto University has a course on elementary data structures and algorithms. One of the teaching methods on the course is *Visual Algorithm Simulation* (VAS), which is a software consisting the following interaction cycles. First, a computer displays a student visual representation of a data structure. The student interacts with the visualisation, typically with a mouse click, to manipulate the state of the data structure. Finally, the computer shows the updated state. A *Visual Algorithm Simulation Exercise* (VAS exercise) allows students to practise an algorithm in this abstract, graphical environment. The student is given a data structure in its initial state, and the student tries to simulate the execution steps of an algorithm according to their mental model. The student can choose different manipulation actions at each step. However, there is only one sequence of actions which is the correct solution of the exercise. When the student thinks they have finished their steps, the student request the exercise software to grade their performance. The student receives instant, automatic feedback on how closely they followed the steps of the correct algorithm. [24, p. 8–10, 25, p. 1]

Figure 1 is a screenshot of the graphical user interface of a VAS exercise in progress. The exercise shows an instruction text, a piece for program code for reference, and the visualisation of the data structure in its current state. The exercise in the figure is about inserting random integer keys into a binary search tree. Initially, the tree was empty, and the student inserted key 22 into it by clicking the empty root node of the tree. After an insertion, the tree expands by showing two empty children of the inserted node. The student inserted then other five keys similarly by clicking their correct locations. The exercise shows a remaining stack of five keys, value 15 at top, to be inserted to the tree.

VAS exercises studied in this thesis are computerised, but they have analogues. Historically, students have solved algorithm simulation exercises on pen and paper, and teaching assistants have graded them manually. The automatic grading software TRAKLA and graphical, computerised simulation environments (Tred, Matrix) were developed in 1990s at Helsinki University of Technology. The obvious benefits of VAS exercises are personally tailored assignments for each student, direct manipulation of graphical representation, automatic, instant feedback, and a possibility to revise each's own solution. [24, p. 14] Another analogue for VAS exercises could be the puzzle game of Rubik's cube such that there are certain predefined operations, turns, which lead from one state to another. Unlike Rubik's cube, where the only the end result matters, the correct solution of a VAS exercise must also have correct intermediate steps.

Undo | Reset | Model Answer | Grade

Ohjeet: Use the BST Insert algorithm to insert values as they are shown at the top. Click on any empty node in the tree to place the value to be inserted there. Remember that equal values go to the left.

```
1. private BSTNode inserthelp(BSTNode rt, Comparable e) {
2.   if (rt == null) return new BSTNode(e);
3.   if (rt.element().compareTo(e) >= 0)
4.     rt.setLeft(inserthelp(rt.left(), e));
5.   else
6.     rt.setRight(inserthelp(rt.right(), e));
7.   return rt;
8. }
```

Figure 1: A visual algorithm simulation exercise on insertion into a binary tree.

VAS is part of the larger field of *software visualisation* (SV) which studies graphical representations of algorithms and program code [46, p. 140]. The major purpose of SV is to assist software professionals in software engineering and software development. The methods of SV include analysing structure of software systems, viewing program execution behaviour over time, and mapping evolution of software as development history. [32] However, VAS as SV is used for particularly educational purpose. A closely related form of VAS is *visual program simulation* (VPS): another computerised pedagogic technique, but for introductory programming. VPS shows the execution of a particular program written in particular programming language and given input to the student. The student simulates the execution of the program, instruction by instruction, similar to VAS. VAS is more abstract version of VPS, as it discusses visually the dynamic behaviour of algorithms and data structures rather than any specific program, programming language, or hardware. [46, p. 141–185]

The key problem studied in this thesis is that the automatic feedback of the VAS exercises is very limited. Student only knows by score the relative correctness of their steps after submitting a solution they consider finished. After that they can also view a model answer. Some research has already been conducted to simulate systematic mistakes, *algorithm misconceptions*, in the students' solution sequences [19, 25, 42, 43]. The key idea is that if a student has an incorrect mental model of the algorithm, they repeat their mistake in their solution. Then an instructor or a

researcher might be able to write an algorithm which corresponds to the student's mental model. From now on the instructor-written algorithm is referred as *algorithm misconception model* (AMM). The AMM can be used to simulate student's mistake. It allows to automatically classify a student's solution as a known misconception. Furthermore, if a solution is classified as known misconception, the student can be given automatic feedback which corrects the misconception. This would further support learning. [19, 43]

There are theoretical problems related to detection of misconceptions. Each time a student tries to solve a VAS exercise, they receive pseudorandom, personalised initial state. This is to counteract plagiarism and allow practising the same exercise several times. The automatic creation of a personalised initial state is called *input generation* hereafter. The problem is that the method which is used for input generation affects on what kind of AMMs can be detected. Some inputs produce the same steps both with the correct algorithm and an AMM. Another time, when looking at a student's solution sequence, two AMMs could have produced the same sequence. The inputs that are designed to counteract these problems are called *misconception aware inputs*. [25]

E-learning, VAS exercises, and studying misconceptions is essentially part of *learning analytics*. Generally, learning analytics is a vast and evolving field. which is typically used on large online courses on to measure, collect and analyse data about learners. The online course software, a *learning management system* (LMS), can, for example, record students' answers on exercises, submission times, or how the students interact. The LMS will then generate reports and statistics for the teacher to support understanding and optimisation of learning. [2, 30] The field of learning analytics combines computer science and statistics with learning science, sociology, and psychology. In higher education it aims to detect academically at-risk students, predict individual learning needs, and reveal critical learning obstacles and success factors. Learning analytics can even support and guide each learner individually in addition to peer students and teaching staff. [8, p. 26–27, 15] Learning analytics in VAS exercises could mean analysing misconceptions to improve instructions in the learning material, detecting parts of algorithms that are typically difficult to understand, and giving automatic, corrective feedback to students.

1.2 Objectives

The planned contribution of this thesis is to:

1. Develop a tool to replay students' solutions to VAS exercises. This would allow an instructor to study candidates for AMMs.

2. Design a tool which matches AMMs to students' solutions in VAS exercises. This allows the teacher to validate their candidate AMM and detect further misconceptions.

3. Confirm the results of the earlier studies [19, 43] by implementing the AMMs of the Build-heap algorithm.

1.3 Research questions and methods

The research in this thesis consists three substudies: a literature study, a development of methodology, and an empirical study. The literature study consists of research questions RQ1 and RQ2. Its purpose is to support an empirical study. Research questions RQ3-RQ6 arise from background problems which affect the empirical study. Research question RQ7 is the empirical study. It is also the main research question of this thesis.

- RQ1: What earlier research is there on misconceptions in VAS?
- RQ2: What known misconceptions are there related to the topics of the implemented VAS exercises?
- RQ3: How can a VAS exercise input support detection of misconceptions?
- RQ4: How can the communication fail in a VAS system between the student, the software, and the instructor?
- RQ5: Is the input generation of the current VAS exercises particularly *misconception aware* in terms of [25]?
- RQ6: Is it possible to construct a ”player application” for the current submissions of JSAV exercises?
- RQ7: *Which AMMs for the VAS exercises have matches in the submission data?*

The submission data is collected during years 2016–2019 from Aalto University course *Data Structures and Algorithms Y*.

1.4 Scope and structure

This thesis discusses a number of misconceptions about a selected set of algorithms. These data structures and algorithms taught on the corresponding Aalto University course. The course also has programming exercises on the same subjects, but this thesis concentrates on the VAS exercises listed in in Appendix A. The *Build-heap* exercise was the main subject, because it had both reconstructible exercise recordings from several years and earlier studies which could be replicated [19, 43]. The input generation of the Build-heap exercise was studied, and then software tools were developed to analyse students' misconceptions both manually and automatically. The input generation and record reproducibility of the *Dijkstra's algorithm* exercise was also studied, but the emphasis was on modifying the program code for improved input generation. Also the exercises *Evaluating postfix expression* and *Quicksort* were studied for input validness and reproducibility. However, the scope of this thesis was narrowed only to the data of Build-heap exercise recordings to be able to study it comprehensively.

Although the thesis discusses misconceptions, its discipline is computer science, not pedagogics. The thesis studies what kinds of misconceptions there seem to be in a VAS exercises. The thesis summarises an earlier discussion on how the misconceptions form, and whether the teacher has understood the student's mental model correctly. However, no further research of these questions is conducted.

The thesis is balanced between the methodology of misconception detection and developing a finished software product. The software product, a misconception detector, is written for the Build-heap exercise for demonstration. The methodological work addresses problems related to detection of misconceptions. Its purpose is to notify future researchers about the nontriviality of the software design task.

The rest of this thesis is organised as follows. Section 2 is an overview on the related research. It contains the results of the literature study. Section 3 provides theory related to detection of misconceptions. Section 4 describes the methods and data of the empirical study. Section 5 represents results of the empirical study. Its substructure follows the previous section. Section 6 discusses the reliability and meaning of the results and recommends actions for future research.

2 Background

This section provides detailed background information which is required to understand the new research of the thesis. The section begins with an overview on how VAS exercises help students learn and why misconceptions should be studied. Then it reviews research on known algorithm misconceptions to support the empirical study in this thesis. Subsection 2.3 concentrates on technology for detecting misconceptions that are known to exist. Subsections 2.4 and 2.5 proceed deeper in the theory on how to design VAS exercises that support automatic detection of misconceptions. Finally Subsection 2.6 describes the software context in which the empirical study is conducted.

2.1 Pedagogics of VAS and misconceptions

This part answers to RQ1: *What earlier research is there on misconceptions in VAS?*

Data structures and algorithms are essential in Computer Science. It is difficult to teach them by static images in a textbook, because algorithms are dynamic: they are executed step by step over time, and the execution depends on an initial state and a set of rules. An algorithm animation software is the next step. It simulates the execution of an algorithm with given input. However, a student which only views an animation will not learn much. Therefore a VAS exercise is a further improvement: the student simulates the execution of an algorithm. This higher level of ”constructive engagement” is important: the student must use their reasoning to achive the correct end result. Therefore a VAS exercise aids the student to build a working mental model. [20, p. 172, 24, pp. 11–15, 96]

”Mental model” is a central term in Cognitive Science. Because the discipline is vast, there are many definitions of the concept. In the context of this thesis, the definition comes from Johnson-Laird’s theory. A *mental model* is a simplified representation of the real world in a person’s mind. It is an explanation on how some part of the world works. The person has constructed the model themselves, and the model allows solving reasoning problems. [12, p. 346, 14, pp. 5–6] Inevitably, different persons have more or less different mental models. Learning can be explained as people constructing meaningful mental models [41, p. 86].

The high level of abstraction in VAS is essential in a pedagogic sense. Compared to VPS, it hides details specific to hardware or programming language, such as types of variables and memory layout. The data structure concepts used in JSAV-based VAS exercises are arrays, nodes, connections between nodes, trees, and graphs; JSAV supports creating these abstract objects [1]. Examples of algorithmic concepts in JSAV-based exercises are temporary variable, function call stack, and swap of two elements. Students are supposed to integrate these concepts in their mental models of data structures and algorithms. Therefore the purpose of high-level abstract representation in VAS is to reduce the cognitive load of the student.

[1] `http://jsav.io/datastructures/`

When students try to solve visual algorithm simulation exercises, they often make mistakes. These mistakes are either due to carelessness, trial and error, or a systematic misunderstanding, a *misconception* [43]. A student *conception* is a belief or theory which explains some scientific phenomenon. The conception becomes a misconception, or *alternative framework*, when it is in conflict with the accepted scientific theories. A misconception can be seen as a mental model which is incomplete or contradicting compared to the model that is supposed to be learned. Therefore misconceptions are systematic but erroneous, and they vary based on students' earlier knowledge. A particular problem is that student might think they have learned the case, but their mental model is incorrect or partially correct. It is known that novice programmers form misconceptions by reasoning about program examples. The user interface can also be a source of misconceptions. [19, p. 62] To summarise, VAS exercises help students build viable mental models of data structures and algorithms. Misconceptions happen in learning. It is assumed that knowledge of misconceptions helps improve VAS exercises.

As mentioned in Section 2.2, Seppälä, Malmi, and Korhonen [43] have studied student misconceptions in visual algorithm simulation exercise of build-heap in the TRAKLA2 environment. They identified seven AMMs which described 60 % of the students' incorrect solutions.

There are two pedagogical theories which explain many misconceptions in VAS. The first is a learning strategy called *imitative problem solving*. The algorithms textbooks and the model solutions of the VAS exercises show examples of correct sequences with some particular algorithm and input. The student might try to look a model solution and map its *surface features* into another instance of the same exercise: they try to perform the same operations on the data structure on given data that was in the data of the example. [38, 43, p. 252]

According to *repair theory*, some students have systematic errors. There exists a procedure for producing the erroneous answers. Typically these procedures are minor variations of the correct procedure. In addition to the systematic errors, there are errors called ”slips”. When the student makes a slip, they intended to do something, but did something else instead. These slips are not systematic. An interesting case is when the student follows rigorously their own procedure, and then think they have arrived in a dead end, an impasse. Then the student tries to ”repair” their procedure by inventing a new rule. [5, p. 379–381] Some submissions to the Build-heap VAS can be explained with the repair theory. The student first executes a non-recursive algorithm variant and then applies additional, systematic steps to ensure the heap property. [43, p. 253] If the repair theory applies in some situations, it is still unknown whether a student has tried to review the learning material, construct a rule from a model answer, or invent a totally new rule. The slips are essentially carelessness, like clicking a nearby, similar object in the VAS exercise, or forgetting to do one step once. The systematic procedure and the slips can be thought mathematically as a model or function with random noise.

2.2 Literature study

The list of algorithms covered in the Aalto University Data Structures and Algorithms Y course is given in Table 13, Appendix A. This section presents results from a literature study on misconceptions on these algorithms, RQ2. Algorithm analysis and recursion are included as they are discussed on the Data Structures and Algorithms Y course. This set of topics is typically called a Data Structures and Algorithms course or a *CS2 course*. [37, 53, p. 169].

The cited literature also discusses the causes behind each misconception, but this is omitted to only have a brief overview here.

The following searches were done on scientific literature:

- ”algorithm misconception” on the ACM Digital Library [2]
- ”algorithm misconception” limiting to Computer Science articles on Scopus [3]

Also the concept inventory by Porter [36] had many fruitful references as a starting point.

There are synonymous, related terminology regarding misconceptions. When a catalogue of misconceptions is refined and validated, the misconceptions can be used to form a *concept inventory*. Concept inventories are used for testing students' correct understanding and for improving teaching to better encounter the misunderstandings. [22, 36] A misconception may also be called a *partial understanding*, an *incorrect understanding*, a *student-constructed rule*, a *mistake*, or a *bug* [46, p. 358].

The results of the literature review are shown in the following tables. Table 1 summarises the algorithm misconception literature reviewed in this thesis. The table merely categorises discussed misconceptions; it does not evaluate how common or strongly validated each misconception is.

Table 1: Reported algorithm misconceptions in reviewed literature.

Category	Number of misconceptions	References
Binary heap	8	[19, 35, p. 31, 43]
Binary tree	11	[7, p. 23, 53, pp. 172–175]
Efficiency	16	[9, 11, 53, p. 173, 34]
Linked list	5	[53, p. 171]
Recursion	7	[12, 35, p. 32]
Sorting	1	[48]
Total	48	11

Table 2 shows the diversity in the methodology of the reviewed misconception studies. Most studies have collected data directly from students: either by written

[2] `https://dl.acm.org/`
[3] `https://www.scopus.com/`

questionnaires having open-ended, multiple-choice or fill-the-black questions. Think-aloud interviews for students and flash tests in class are similar tools. Other main type was analysis of student's productions that already existed: analysis of exam answers or solutions to visual algorithm simulation exercises. Taherkhani, Korhonen and Malmi [48] studied machine learning for classification of programming exercises and detected inefficiency in the implementations, and this is regarded as a misconception in this thesis. Farghally, Koh, Ernst, and Shaffer [9] used a *Delphi process*: a panel of computer science instructors formed an agreement on the most difficult and important concepts for algorithm analysis topics. The study populations were mostly first-year university students. Özdener [34] included and Gal-Ezer and Zur [11] exclusively studied high school students.

Table 2: Research methods in the algorithm misconception studies.

Study	Open-ended writing	Multiple-choice	Fill-blanks writing	Interview	Exam	VAS	Other
[11]	x	x					
[35]	x	x	x				
[53]	x			x			
[34]	x	x					
[7]		x		x	x		Flash test
[12]					x		
[19]						x	
[43]						x	
[9]							Delphi
[48]							Prog.ex.

Recently, the Basic Data Structures Inventory (BDSI) [36] was created as an inventory of students' concepts and misconceptions on basic data structures. The inventory was collected by presenting open-ended and multiple-choice questionnaires to students. Finally, the concepts are also validated by interviewing students. The BDSI database has questionnaires which teachers can use to analyse their students' understanding and difficulties on the following subjects: linked list, binary search tree, binary tree, stack, set, and implementing an abstract interface with several data structures. Specific work has been performed to analyse the validity of the questionnaire instrument: topics are relevant to instructors; the questions are meaningful and address key concepts and typical difficulties, and students interpret the questions correctly. The specific BDSI content is stored on an access-limited web forum, and therefore the questions and misconceptions are not described in detail in this thesis.

Danielsiek, Paul, and Vahrenhold [7], and also Zingaro et al. [53] acknowledge that most work on misconceptions in computer science is related to object-orientation or basics of programming. Sorva [46, p. 358–368] has listed 162 common novice programmer misconceptions , thus demonstrating the vastness in which the "CS1

course" has been studied. Therefore the literature study can be concluded as that data structure and algorithm misconceptions have been studied less than other types of misconceptions in computer science education.

Misconceptions in VAS seem to have been studied even less than algorithm misconceptions in general. The misconceptions for the binary heap VAS are unique, although many of them are related to understanding of recursion and how the program execution proceeds [19, 43]. The binary heap misconceptions are studied in detail in later sections of this thesis.

2.3 Algorithmic detection of known misconceptions

This section continues RQ1 and discusses algorithmic methods of detecting misconceptions in VAS exercises. If summarises earlier work.

The person who studies students' solutions to a VAS exercise is referred as the *instructor* throughout the thesis. The instructor is a role: it means a person which has knowledge on teaching of algorithms and VAS. The person can be a course lecturer or a researcher. The instructor studies a dataset of students' solutions to find new misconceptions and improve the learning material.

Seppälä, Malmi and Korhonen [43] studied students' answers to a VAS exercise on the Build-min-heap algorithm. The exercise features a tree view of a binary heap with random values, and the student must manipulate the heap by swapping keys in the heap. Figure 2 shows the user interface of the exercise. The exercise would also display a pseudocode of the algorithm for reference, but it is omitted in the figure for compactness.

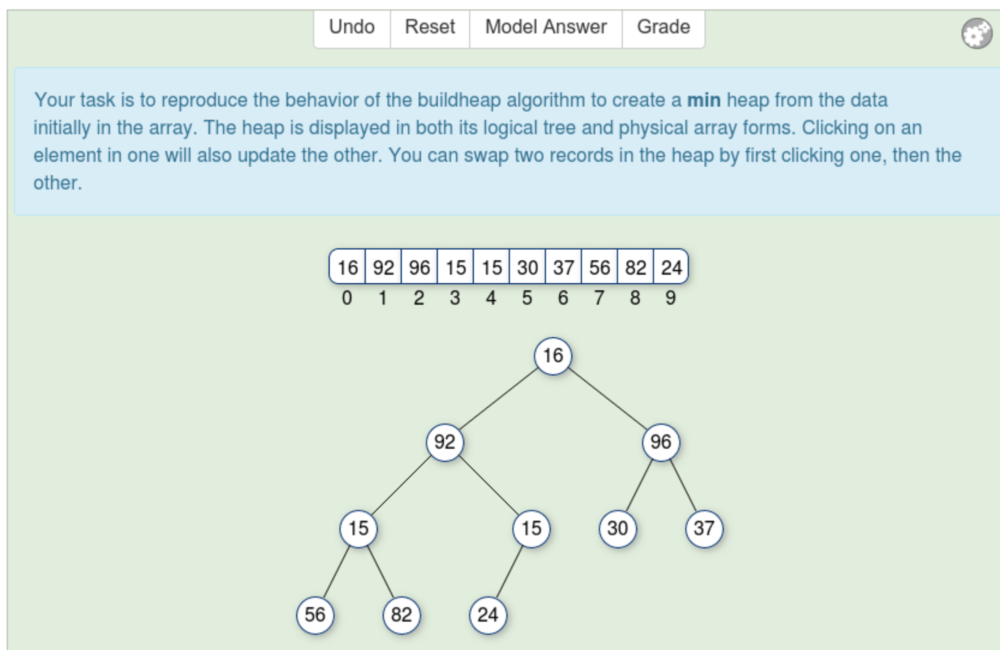


Figure 2: Graphical user interface of the Build-min-heap exercise implemented with JSAV.

The pseudocode of the correct Build-min-heap algorithm is shown later in Section 3.2.3. A correct solution must the same swaps in the same order as the correct algorithm. Thus each student's *solution sequence*, or a *trace*, is a sequence of states in the binary heap so that each swap is a transition from one state to another. Seppälä et al. reviewed these *solution sequences* manually, step by step, trying to find systematic errors that could be described as variants of the correct algorithm. These known algorithm variants are the AMMs: each of them is a hypothesis of an incorrect mental model described precisely as program code.

The next step towards automatic detection of misconceptions is to have a *comparison algorithm* which decides whether a given trace follows the correct algorithm or an AMM. This algorithm has two purposes. First, it strengthens the evidence that some hypothetical AMM is present in the set of manually studied submissions. Second, it allows to automatically detect the same misconception later in new submissions. Seppälä et al. [43] describe how to measure the similarity of a trace to the correct solution in a VAS exercise. The trace contains the initial state of the heap which is random integers in random order. Both the correct algorithm and all the AMMs are given this input, and each of them produces a sequence of states. In the case of Build-heap algorithm, each state is the contents of the heap array. Then the comparison algorithm computes the similarity between the state sequence of the student's trace and the state sequences of each AMM, including the correct algorithm.

The comparison algorithm iterates over the candidate sequence. On each step in the candidate sequence, it tries to find a matching state in student's sequence. Essentially, the algorithm tries to find maximum number pairs of equal states between sequences so that the states are in the same order. The algorithm is discussed in detail later in Section 2.7. The candidate algorithm whose sequence has the highest number of matching states from the beginning is decided to be the best matching algorithm. However, to classify a solution sequence as a misconception, the sequence of an AMM should match at least two steps more than the correct solution. The two-step threshold has been chosen to separate actual misconceptions from single-step, *slip*-type errors, which are due to carelessness. [43, p. 246]

Because student's solution sequence to a VAS exercise can include both a misconception and a slip, Seppälä [42] further studied modelling the sequence with code mutation. Initially, for some VAS exercise, there are the correct algorithm and several AMMs. Then the instructor modifies one of the algorithms for the exercise: they choose a points in the code that can be mutated, called *mutation points*. The result is called the *metamutant*, an algorithm that can be altered during execution. Each mutation point can be altered by a *mutation operator*: integer offset by one, negation of comparison operator, rounding versus truncating division, and skipping a subsection of the code. These mutations are not permament, but can happen at some states of code execution, modelling a carelessness error of a student at some point of their simulation sequence. [42]

Correspondingly, the mutating misconception classifier tries to map the student's solution sequence onto one metamutant at a time. It begins executing the metamutant and mapping the solution sequence from the beginning. When it encounters a mutation point, it chooses the option of the mutation operator that matches with

the sequence. If the sequence proceeds in a way where the same mutation point is encountered multiple times, the option of the mutation operator can be chosen independently each time. The combinations of explored mutations are recorded in a *mutation tree* where the each node is linked to a specific step in the algorithm sequence and a mutation point in the metamutant. The matching is done by a depth first search (DFS) from the beginning of the sequence with the initial version of the metamutant. The intention is to match steps of the sequence as far as possible, backtrack to previous mutation point if necessary, try another variant of the mutation and continue. This DFS of the mutation tree ends when a perfect match is found. This a specific path of the tree matches perfectly to the student's sequence. Another ending condition is that all combinations of mutations are explored and none of them fully matches the sequence. [42]

2.4 Misconception-aware input in VAS

This section addresses RQ1 regarding the automatic exercise instance generation in VAS and its effects on misconceptions. It combines theoretical development of earlier studies.

Korhonen, Seppälä, and Sorva [25] studied the idea of *misconception aware input sets* in VAS exercises. The target is that if an input data for a VAS exercise is random, it should be validated to ensure that each input reveals all the known misconceptions. The article describes two approaches to input validation. First, a random input could be run against AMMs, and only instances where AMMs produce different steps are accepted. This is called *constrained input.* Second, a VAS could generate its seemingly random input by a *preconfigured template*, which is designed by an instructor to reveal a misconception. These templates describe a relative order of the input, and the input can be randomised according to the rules in the template while always leading to the same simulation steps with the same template. [19, 25] For comparison sorting algorithms, such as selection sort, mergesort or quicksort, this is clear: they sort the keys in their relative order, and thus the absolute values of the keys does not matter.

Table 3 collects requirements of misconception aware input from several VAS studies. These properties will be referred as *input requirements* (IR) in this thesis. The next paragraphs discuss each IR briefly.

IR1 is required to counteract plagiarism. Fulfilling IR1 requires in practise automatic generation of exercise inputs, as there can be several submission attempts and hundreds of students. [24, p. 120] Using pseudorandom data is useful for generating inputs which have high variance.

IR2 rises from the idea of benefits of random, constrained input. Korhonen et al. [25, p. 4] note that the method of preconfigured templates may hinder the discovery of new misconceptions, because the templates restrict the input space.

IR3 is for both to make the student think through all essential cases of the algorithm for learning, and also to ensure that the student will thoroughly understand all the essential features correctly. For IR3 of Build-heap, [43, p. 253] notes the use of recursion in the exercise as an example. Practical examples of IR3 are discussed

Table 3: Requirements of misconception aware input in VAS exercises.

Requirement	Description	Reference
IR1	The input should be personalised to vary between exercise instances and students.	[24, p. 120]
IR2	The input should be as random as possible to maximise possibilities for detecting new misconceptions.	[25, p. 4]
IR3	The input should always result in a sequence which shows all the essential features of the algorithm.	[43, p. 253]
IR4	The correct algorithm and an AMM should produce different sequences with the same input.	[43, p. 253]
IR5	Each AMM should produce a different sequence with the same input.	[43, p. 253]
IR6	The input should have optimal length for learning with reasonable effort.	[19, p. 64]

later in Section 3.2.

IR4 ensures that all the sequences of all AMMs are different from the correct one; otherwise the student could have a known misconception, but some inputs would not reveal it. IR5 is similar to IR4, but ensures separation of known misconceptions. This is important if we want to assign a written, corrective feedback for each AMM that can be shown automatically to the student when a misconception is detected. IR3, IR4, and IR5 often require an input long enough to distinguish between the misconceptions. It is also possible to split the exercise into several subexercises. [19, p. 68].

IR6 has the assumption that on average, longer input causes higher workload for the student. The length of an input typically affects the asymptotic time complexity of an algorithm: a longer input generally means longer execution time. Too high a workload will likely cause some students to skip the exercise. Therefore IR6 constrains the length of the input.

As a practical case for the input length, the current JSAV exercise of the Build-heap algorithm has an input of 10 items, while the older TRAKLA2 implementation had 15 items. Regardless of this, all the misconceptions detected in the TRAKLA2 submissions were also found from the JSAV data. However, not every single exercise instance is guaranteed to reveal all of the known misconceptions. [19]

2.5 Software testing perspective

This subsection answers to RQ3: *How can a VAS exercise input support detection of misconceptions?* We first study the problem of VAS exercise input generation from the software testing perspective. We then review existing research which proposes automatic input generation for programming exercises. Finally, we discuss whether the input generation method could be applied to VAS exercises. This subsection is a

theoretical summary for future work; it is not applied in practise in this thesis.

2.5.1 Relevant testing terminology

From a software testing point of view, A VAS exercise can be seen as a *unit test* for the student's mental model of algorithm: give input, run algorithm, and measure correctness by examining output. This is essentially a *black-box test*, as we cannot see the student's thoughts. The *specification* is the correct algorithm. If we know some AMMs, it is possible to deduce new tests from their program code, which can be considered as white-box testing. [45, p. 210–216]. IR3–IR5 form the *test adequacy criteria*. The criteria are generally *error-based testing*, as we know what kind of errors in the output the AMMs may cause. [16, p. 12–13]

Compared to unit testing with actual program code run on computer hardware, we have the following additional difficulties and requirements, which come from Table 3:

- One input provides all test cases.
- The execution might malfunction at some random steps (slip errors).
- The software may change between test runs (learning, trial and error).
- The output that is tested is an execution trace (simulation steps).

For the single student case, the challenge is informally: "Generate one input which reveals all the known bugs." The emphasis is on the "known bugs", because as it generally is with software testing, any amount of testing cannot prove that the program is always working correctly [45, p. 210–216]. Accepting this, the challenge is related to the method of *path testing* whose aim is to ensure that all execution paths in the program have been covered. An *execution path* is a sequence of program states from the beginning of the program to the end with some particular input. Two execution paths diverge in an IF-ELSE branch. A WHILE or FOR loop in the code causes a loop in an execution path. To ensure that all execution paths are covered, the program must be run with different inputs. This set of test inputs together ensure that each program line is executed with some input. [44]

In testing terms, IR3 aims for *test adequacy* by maximising *branch coverage*: all alternative IF-ELSE branches of the program should be covered. This also maximises *statement coverage*: if all branches are covered, then also all lines of the program are executed. [52, pp. 367, 369, 374]. However, as the program is executed only once, we should find a criterion for only one input which produces an *execution path* that still maximises the statement and branch coverage. Intuitively, longer input enables more possibilities for the coverage. Although there is only one input, the input still contains all the test cases derived from IR3–IR5.

2.5.2 Test data generation by symbolic execution

Regarding Java programming exercises, Ihantola [16] studied the possibility to generate sets of test data which are seemingly random, but together explore all possible *execution paths* of a program. *Symbolic execution* of programs uses symbolic values and variable substitution instead of concrete values. Figure 3 shows an example: if the `power` function is given is given input `x = a, y = 3`, the symbolically executed result is `a * a * a`. More generally, if `y = b > 0`, the result is a to the power of b.

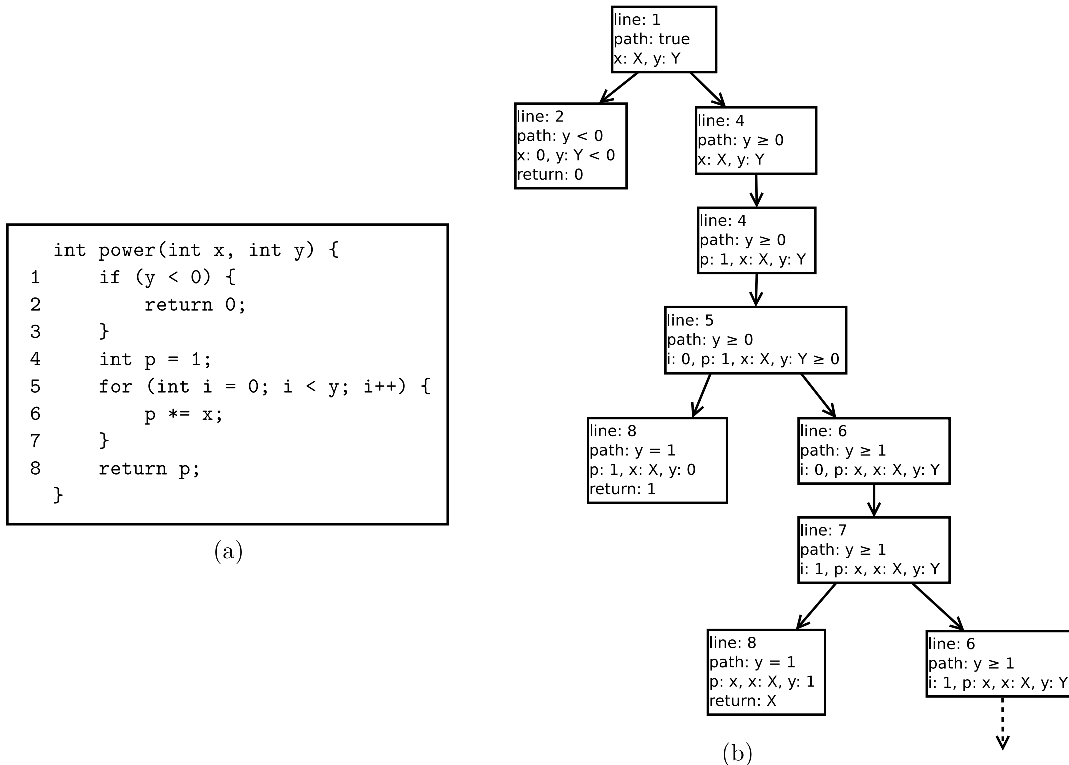


Figure 3: (a) program for computing an integer power of an integer; (b) the symbolic execution tree of the same program. *Line* denotes program counter and *path* denotes path condition.

Symbolic execution runs program code with conditional expressions as values of variables. The state of the program consists of values of variables, a *path condition* and a program counter (the line of program code that will be executed next). The path condition describes constraints of the known variables. At each control statement (IF, ELSE, FOR, WHILE) it is decided which branch can be taken. Then the execution of program, making choices at control statements, creates a sequence of states of the symbolic execution, and this is called an *execution path*. The diverging execution paths of a program form together the *execution tree* of a program. [16, p. 18–19]

Figure 3 (b) shows the execution tree of the `power` program. The leftmost execution path ends in the case where the input variable $y < 0$ and the output is 0 (regardless of x). The next execution path to the right describes the case where $y = 0$ and the output is 1. The third path to the right describes case $x = X, y = 1$

with output X. The rightmost path shows how the tree expands recursively as the `for` loop on line 5 in Figure 3 (a) proceeds depending on value of y.

The example of symbolic execution in Figure 3 also features *lazy initialisation*: values of variables are uninitialised in the beginning, and their constraint expressions are refined as the program reads or writes these variables [23, p. 557].

A more compact representation of a symbolic execution tree is a *control flow graph*. The main idea is to analyse essentially different execution paths and then derive the corresponding conditions for each $(input, path)$ pair. [16]

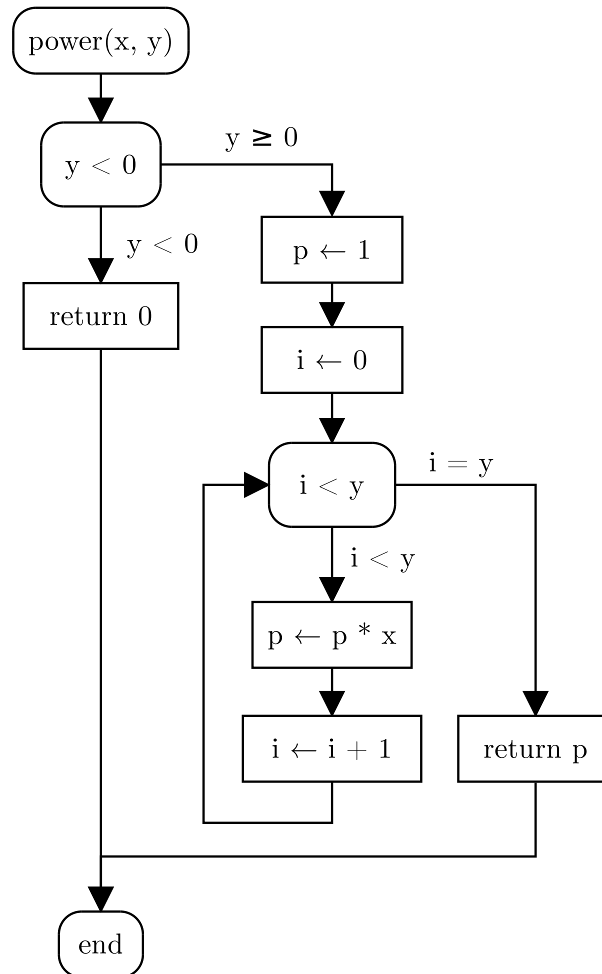


Figure 4: Control flow graph of Figure 3 (a).

Symbolic execution with control flow graphs is described in [16, p. 18–20] and [52, p. 372–374]. Figure 4 shows an example of a control flow graph. Ihantola's thesis aims to derive test data from both the student-written program and the correct model solution together. The correct model solution in VAS exercise context is the correct algorithm, and the AMMs can be thought as student-written programs.

The main idea of symbolic execution with lazy initialisation and path exploration is summarised here from [23]. Begin the execution of a program with an undefined input set, similarly to example in Figure 3. When an IF, FOR, or WHILE statement is

encountered, choose one of the branches. This creates a restricting condition on the input according to the expression of the conditional statement. If some execution path reaches a contradiction, the search must backtrack until another branch can be explored. Eventually the program execution reaches its end by some path, resulting in conditions on the input. Then fixed input values causing that execution path can be generated from the input conditions of the path. Path exploration in this context means a depth-first search for the possible input paths. [23] Therefore symbolic execution for input generation is a depth-first search in the control flow graph.

2.6 Software context

The software context of the thesis has several components. The A+ Learning Management System [18] provides the Aalto University course "CS-A1141 Tietorakenteet ja algoritmit Y" (Data Structures and Algorithms Y), which has several kinds of content: VAS exercises, multiple-choice exercises, programming exercises, and tutorial text. The VAS exercises are implemented with the JavaScript Algorithm Visualisation Library (JSAV), which allows creating both slideshows and interactive exercises with HTML5, CSS, and JavaScript [21]. In addition to the JSAV library, each VAS exercise has its own JavaScript code, which creates random input for each exercise instance, provides simulation functionality, grades student's answer automatically, and provides a visualisation of the correct answer. The JSAV-based VAS exercises are a part of the OpenDSA project, which is an open-source online textbook for DSA courses [20]. The Aalto University DSA course uses OpenDSA as the textbook, but the A+ LMS also provides the same VAS exercises and also records students' solutions to them.

2.7 String matching for VAS

The process of grading a student's solution for a VAS exercise requires computing a similarity between the student's solution sequence and the sequence produced by the correct algorithm. Also the automatic detection of an AMM requires computing the similarity between the student's solution sequence and sequences of each AMM. These sequences are steps produced by the student or an algorithm, and generally, comparing them is the task of *string matching*.

The JSAV library grades a VAS exercise used on the DSA course with its *default grader* which is independent of the type of the exercise. The grader is shown in Algorithm 1 from the JSAV source code[4]. The student's sequence is S and the correct sequence is C.

This grader has two notable properties. First, it gives points only until the first mismatching state. If the student makes one slip and then continues simulating the correct algorithm, the position of the slip determines the score. Therefore the algorithm might give a confusingly low score for a student who made only one mistake in otherwise correct simulation. This behaviour can be observed easily in the

[4]`https://github.com/vkaravir/JSAV/tree/master/src/exercise.js` lines 128–165. Visited on 09/16/2020.

Algorithm 1 Default grader of JSAV

```
1: procedure JSAV-GRADE(C, S)
2:     i ← 0
3:     while i < |C| and i < |S| and C[i] = S[i] do
4:         i ← i + 1
5:     end while
6:     return i / |C|
7: end procedure
```

Insertion sort and *Build-heap* exercises. Swap elements at the same random indices twice. The state of the array is then the same as the initial state. Then execute the correct algorithm and click the grade button. The submission will receive zero points.

Second, if the first $|C|$ steps of the student's sequence are correct, but the student continues performing steps after the correct algorithm has stopped, they will still receive a 100% grade. It could be said that the grading algorithm does not catch cases where the student does not understand the ending condition of the algorithm. It must be noted that nearly all students still receive full points from a VAS exercise after some attempts [19, p. 65–67]. The essential feedback that the default JSAV grader gives is whether the submission was correct or not.

Seppälä et al. [43, p. 246] have used a more elaborate similarity algorithm when they have studied the misconceptions in the Build-heap VAS exercise. It is possible that the same algorithm has also been used in the earlier TRAKLA2 VAS exercise environment: the TRAKLA2 grader ”reports the number of orrect steps out of total number of required steps” [27, p. 271]. The grading algorithm is described in the article [43] as follows.

> ”The comparison procedure iterates over the candidate sequence, selecting a single state at a time, and iterating over the student's states trying to find that very same state. The comparison then selects the next state in the candidate sequence and starts its search in the student state located after the previous match. This is repeated until no more equal states can be found.
>
> The candidate that finds a match furthest away in the student's sequence is the one that best explains the student's sequence. In most cases, there exist a number of variants that match the student's sequence equally well. In these cases, we have selected the most conservative option available. Basically, this will be the candidate that best resembles the correct algorithm or the most general one available.
>
> Sometimes, the sequence created by the correct algorithm matches the student's sequence if it is not required that every single state must exist in the student sequence.” [43, p. 246]

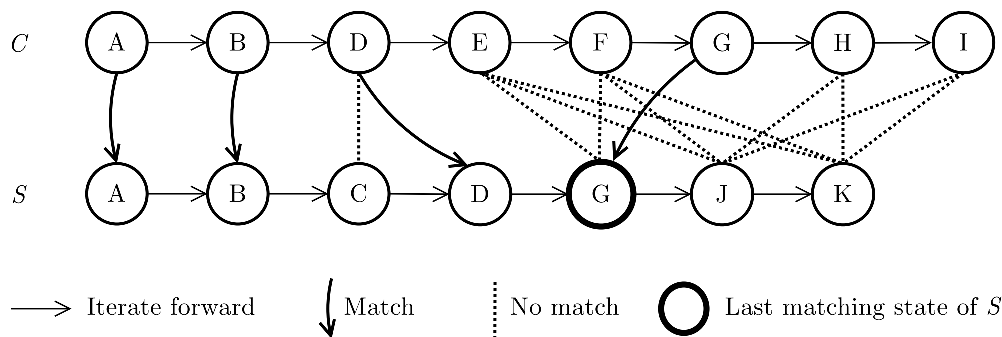
Algorithm 2 is a formal interpretation of the comparison algorithm in [43, p. 246]. Input parameter S is the student's solution sequence containing states, that is,

Algorithm 2 Sequence similarity following Seppälä et al. [43, p. 246]

```
 1: procedure STATE-SIMILARITY(C, S)
 2:     i, j ← 0
 3:     while i < |C| and j < |S| do
 4:         k ← j
 5:         while k < |S| and C[i] ≠ S[k] do
 6:             k ← k + 1
 7:         end while
 8:         if k < |S| then
 9:             j ← k + 1
10:         end if
11:         i ← i + 1
12:     end while
13:     return j
14: end procedure
```

contents of the data structure after each modification: array, tree, or graph. $S[0]$ contains the random input generated by the VAS exercise. C is the *candidate sequence*: a sequence similar to S obtained by running an algorithm variant with the input $S[0]$. Particularly $C[0] = S[0]$.

The procedure matches explicitly *states* in the candidate sequence to equal states in student's solution sequence. The loop on lines 3–12 in Algorithm 2 iterates over candidate sequence: i is index of C. The innermost loop on lines 5–7 iterates over student's sequence: k is index of S. Variable j stores the index of next currently unmatched state in student's sequence. Thus the algorithm tries to map states from candidate sequence to student's sequence. The return value is the location of the furthest state in student's sequence that had a match in the candidate sequence.



Pairs of states (c, s) tested for equality $(c \in C,\ s \in S)$:
(A,A), (B,B), (D,C), (D,D), (E,G), (E,J), (E,K), (F,G), (F,J), (F,K), (G,G), (H,J), (H,K), (I,J), (I,K).
Return value: 4.

Figure 5: An execution example of Algorithm 2.

Formally, each state in C has at most one corresponding state in S, and also each state in S has at most one corresponding state in C; thus $(C[i] = S[m]) \wedge (C[j] = S[m]) \Leftrightarrow i = j$. The mapped pairs of states have the same relative order: $(C[i] = S[m]) \wedge (C[j] = S[n]) \wedge (i < j) \rightarrow m < n$. Figure 5 clarifies this by an example: the states are labeled by letters A–K, but they could be the contents of any data structure. Korhonen, Seppälä and Sorva [25, p. 3] give similar visual description of what seems to be Algorithm 2. Figure 5 emphasizes that Algorithm 2 can skip states both in the candidate sequence and the student's sequence.

3 Methodology

This section provides an in-depth study on problems which make detection of misconceptions harder. Subsection 3.1 provides an overview of the human-computer system we are studying. It also discusses what situations might rise in such a system and how serious they are.

Subsection 3.2 determines IR3 for four VAS exercises, i.e. what are the essential features of each algorithm. It is a demonstration on how IR3 can be defined by program code analysis. The four VAS exercises are *Evaluating Postfix expression*, *Quicksort*, *Heap build*, and *Dijkstra's algorithm*.

Subsection 3.3 concentrates on a practical issue: do the currect JSAV-based exercise recordings support study of misconceptions? The exercises are the same as in the previous subsection.

Subsection 3.4 is a demonstration on JSAV-based exercise software development. The input generation of *Dijkstra's algorithm* JSAV exercise is improved to meet IR3.

Finally, Subsection 3.5 presents an algorithm which tries to generate random input for a VAS exercise fulfilling as many input requirements as possible. This algorithm is one solution attempt for the input requirement problem. It is presented for future research.

3.1 System overview with risk analysis

This section answers RQ4: *How can the communication fail in a VAS system between the student, the software, and the instructor?* It performs a simple risk analysis for a proposed VAS system that gives automatic feedback based on students' misconceptions. The purpose of the analysis is to identify points of communication failure in the system, as there are both human and software components.

The methods of the risk analysis used here are the following. First a flowchart of the system is drafted for *fault identification* to be able to identify events and components. Then an *event tree* is built based on the flowchart. An event tree depicts chains of events and their consequences. Finally, unwanted consequences, *hazards*, are identified. Their probability and severity is briefly discussed to provide a qualitative risk analysis. [3, pp. 34–35, 49, 72]

The system has three components. The *student* tries to learn an algorithm using a VAS exercise. The *VAS software* generates an instance of a VAS exercise for the student, grades the student's answer, and tries to classify the answer as a misconception and give corrective feedback, respectively. The *instructor* analyses students' answers which the VAS software has graded and classified. The instructor's objective is to improve the feedback of the VAS software by finding new misconceptions in students' answers, implementing them as AMMs, and writing corrective feedback.

Figure 6 shows a flowchart of the VAS system with the three components. The flowchart depicts the communication and key decisions in the system, and its purpose is to facilitate the fault analysis. The process begins when the VAS software generates an input for an exercise. The input generator tries to fulfill IR3–IR5 as described before. The student works with the exercise with the given input and submits their

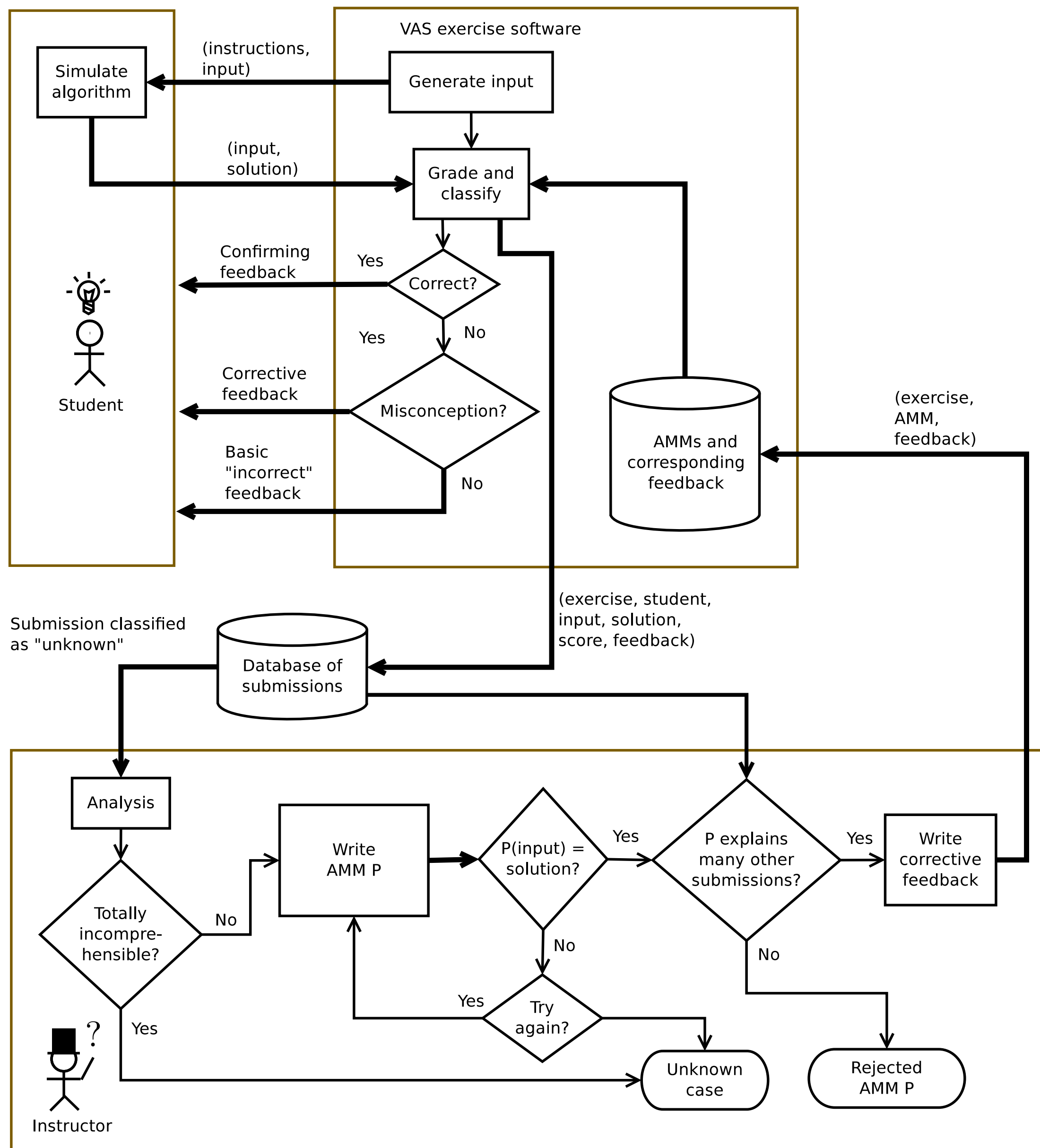


Figure 6: Communication flowchart for a VAS exercise system with automatic misconception detection.

solution. The VAS software grades and classifies student's answer and tries to give either confirming feedback on a correct solution or corrective feedback for a detected misconception.

Figure 7 shows an event tree for student–VAS system interaction. The tree begins from the left with the initiating event *Student submits a solution*. The chain of events proceeds to the right. Each branch has several *reactive events* and only one of them is chosen. The path from an initiating event with some choices of reactive events ends in an *outcome*, which can be either successful or an unwanted situation. [3, p. 72–74] The outcomes are the rounded shapes on the right. There is only one outcome

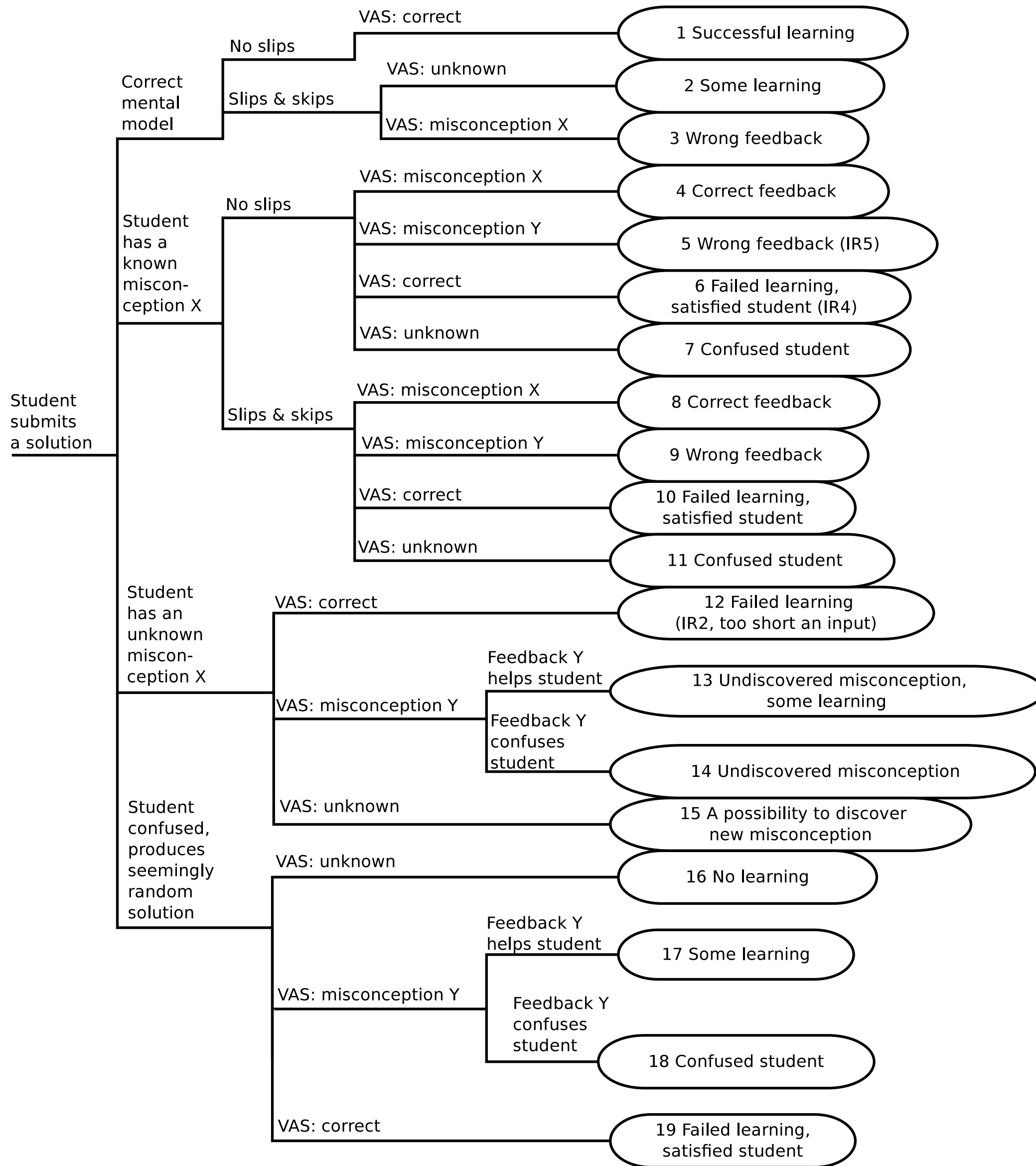


Figure 7: Event tree for student–VAS system interaction

for each path in the tree.

The event tree has three variables: *student's mental model* (correct / known misconception / unknown misconception / confused), *carelessness* (no slips / slips and skips), and the *classification result* of the VAS software (correct / misconception / unknown). It is assumed that some combinations are not possible. For example, if the student has the correct mental model and performs the algorithm without slips and skips, it is assumed that the VAS system always classifies the result as correct. Moreover, if the student has an unknown misconception or are confused, it is assumed that slips and skips are irrelevant for the answer of the VAS software, as the VAS software cannot produce corrective feedback for an unknown misconception.

The consequences of the event tree are assessed in Table 4. The assessment

Table 4: Risk analysis for the VAS system.

No	Consequence	Likelihood	Risk
1	Positive	Likely	-
2	Slightly negative	Likely	Low
3	Slightly negative	Unknown	Low
4	Positive	Likely	-
5	Slightly negative	Likely	Low
6	Negative	Unlikely	Medium
7	Slightly negative	Unknown	Low
8	Positive	Likely	-
9	Slightly negative	Unknown	Low
10	Negative	Unlikely	Low
11	Slightly negative	Likely	Low
12	Negative	Unknown	Medium–High
13	Positive	Unlikely	-
14	Negative	Likely	High
15	Positive	Likely	-
16	Neutral	Likely	-
17	Positive	Unlikely	-
18	Slightly negative	Unlikely	Low
19	Negative	Very unlikely	Very low

is qualitative and based on estimates of the thesis author's knowledge. There are three *types of consequences*: student learns (positive), student is confused (slightly negative), student passes the exercise with a misconception (negative). The *likelihood* of each outcome is estimated qualitatively: very unlikely, unlikely, likely, or unknown. The *risk* is estimated by multiplying consequence and likelihood such that high risk involves both highly negative consequence and high likelihood. The risk is reported only if it has a negative consequence.

The most severe and likely risks relate to Table 4 are Based on the risk assessment, consequences 6, 10, and 12, where the VAS system classifes the solution as correct when the student has an incorrect mental model, are the most severe and likely. Consequence 6 results from failing IR4 and can be counteracted by misconception aware input generation. Consequence 10 is that a known misconception passes by random chance due to slips and skips. The misconception classifier described in Section 4.2 counteracts this by requiring that a a submission is only classified as correct if it matches exactly to the sequence of the correct algorithm. The probability of Consequence 10 can be decreased simply by requiring long enough an input. Consequence 12 results from too short of an input. An example of this is the detection of *Wrong-duplicate* misconception in Build-heap exercise [19, p. 63]: the case of duplicate values for the same children of a parent node is rare, and therefore in most cases this misconceived algorithm produces the same sequence as the correct algorithm. Consequence 2 can be counteracted to some extent by requiring IR3 and enough long a random input. This is the best effort to support detecting the

unknown.

Consequence 19 is the theoretical possibility that student solves the exercise without knowing what they have done. This can be made highly improbable by simply requiring IR2 and IR3. Pattern *Swaps-resemble-build-heap* in Section 5.3 implies that some students might try to complete the Build-heap exercise with minimal understanding by copying a model answer. However, this strategy fails due to random input.

Slips and skips make detection of known misconceptions harder. Consequences 3, 5, 9, 13–14, and 17–18 depict the case where the VAS system gives feedback for a wrong misconception. The classifier in Section 4.2 counteracts this by requiring that in a case where the student's sequence does not match exactly to any misconceived algorithm, the submission is classified as a misconception only if one misconceived algorithm explains the sequence two steps further than the correct algorithm. Otherwise the submission is classified as unknown, which results in Consequences 2, 7, 11, 15, or 16. Then the student receives a response "Your solution only X% correct". This likely confuses student, but they will probably review the learning material and try again.

Consequence 3 implies careful design of automatic feedback. In this case, the student has a viable mental model, but due to slips and skips, the VAS system classifies the student's solution as a misconception. Although the probability of this risk can be mitigated as described before, it is still greater than zero. If the student has a viable mental model and the automatic feedback states that the student has misunderstood the algorithm, the feedback might guide the student away from their viable mental model. It is safer to have automatic feedback which always emphasises a feature of the correct algorithm.

Consequences 13 and 14 are interesting, because then the VAS system masks an unknown misconception by classifying it as a known misconception. This is possible if the input is so short that student's unknown misconception X and the known misconception Y produce the same sequence with the given input. In this case the unknown misconception will be left unstudied, if the instructor only analyses submissions classified as unknown.

Table 5 shows a summary of the risk analysis.

The proposed system has still other issues outside the student–VAS software interaction cycle. The instructor's actions in Figure 6 introduce other faults which are independent of the student and the VAS exercise software. By default, the instructor retrieves a submission from the database for analysis. The instructor reviews the solution sequence and by their expert knowledge, tries to write an AMM candidate P. The hypothetical new algorithm P must pass two tests to be accepted. First, P must match exactly to the student's solution with the corresponding input. Second, P must be practically significant: it must match to many other submissions in the submission database, especially previously unknown submissions.

Notice that it is irrelevant whether the instructor guesses correctly the mental model P from the submission of a particular student S. Student S will never receive the corrective feedback related to P in practise. Students work on one VAS exercise for one or two days, while the instructor might not be able to analyse students' submissions and improve feedback during the teaching period of the course.

Table 5: Mitigation of negative risks

No	Description	Risk	Mitigation
2	Some learning	Low	IR3, input length
3	Wrong feedback	Low	Classifier design
5	Wrong feedback	Low	Classifier design
6	Failed learning, satisfied student	Medium	IR4
7	Confused student	Low	Unavoidable
9	Wrong feedback	Low	Classifier design
10	Failed learning, satisfied student	Low	Input length
11	Confused student	Low	Unavoidable
12	Failed learning	Medium–High	Input length
14	Undiscovered misconception	High	Classifier design
18	Confused student	Low	Classifier design
19	Failed learning, satisfied student	Very low	IR2, IR3

The instructor's decision to update the database of *Known misconception algorithms and corresponding feedback* should be even more careful and conservative. In practise, when a new algorithm P is added, the misconception classifier might decide that some submissions already classified as algorithm Q are now algorithm P. This means that adding a new AMM alters the classification frequencies of the existing misconceptions. The instructor must carefully examine whether the benefits of adding P–the improved feedback for some cases–are more significant than the changing feedback of submissions previously classified as misconceptions.

Finally, the validation of misconceptions is left outside of the system design in Figure 6, but it is still important. Humans encounter failures even in direct interpersonal communication, where two persons discuss face-to-face in the same physical space. One should expect additional difficulties when one of the humans is trying to learn and then a computer software is between the humans, making decisions on which instructor-written messages should be given to which student. The students should be at least asked whether the feedback they receive is relevant. Therefore Section 6.4.1 discusses validation briefly.

3.2 The essential features of an algorithm

This subsection studies RQ5: *Is the input generation of the current VAS exercises particularly* misconception aware *in terms of [25]?*

The input requirements (IR) for VAS exercises were defined in Section 2.4. Therefore RQ5 is about studying the VAS exercises against these input requirements.

A complete input requirement validation procedure would test each exercise against IR1–IR6 in Table 3 in Section 2.4. In practise, IR1 and IR2 are known to be fulfilled, as each exercise instance has randomly generated initial state. This is validated when inspecting IR3.

Fulfilling IR3 was already discussed in Section 2.5.1. The ”essential features of the algorithm” are assumed to be the full branch coverage. The pseudocode of the

algorithm in the VAS exercise is inspected manually and conditions for a good input is decided based on the branch conditions. Moreover, because the input is randomly generated, the probability at which a random input fulfills IR3 must be studied. Therefore another part of the research data here is the source code of the JSAV-based exercises in OpenDSA [20].

Studying IR4 and IR5 requires knowledge of existing misconceptions. In the scope of this thesis, these requirements are only studied for the input generation of the *Heap build* (Build-heap) exercise under RQ7. Studying IR6 is omitted as it is out of the scope of this thesis; Section 6.2 discusses the related problems briefly.

Four JSAV-based VAS exercises from Appendix A are selected for inspection: *Evaluating Postfix expression*, *Quicksort*, *Heap build*, and *Dijkstra's algorithm*. Together they represent a high variety of algorithm classes: a stack algorithm, a recursive sorting algorithm, a tree algorithm, and a graph algorithm.

3.2.1 Evaluating Postfix expression

This exercise features a *postfix form* of an arithmetic expression containing integers, multiplication, and summation signs. An example of a random input generated by the exercise is ”`7 2 + 5 5 + 7 * * 7 2 + + 9 +`”. The student must evaluate the value of the expression using a stack. The exercise is part of the OpenDSA[5]. IR3 for the ”Evaluating postfix expression” exercise were defined as: (1) The expression must always be evaluatable; and (2) The expression must always have both $+$ and $*$ operators. Detailed study of the program code showed that IR3 holds with the exception that the generated postfix expression may have only one type of operator at probability of 1/128.

3.2.2 Quicksort

This exercise features the Quicksort algorithm [6, p. 170–190]. It is defined in the OpenDSA[6]. The exercise presents an array of ten random integers from range 10–124. The pivot is chosen automatically, and the student must perform the swaps of PARTITION function, select a range for a recursive function call, and also indicate when a function call returns.

The IR3 for this exercise was defined such that the student must perform at least one swap in the partitioning phase. Therefore the integers in the input should never be in a monotonically increasing order. Detailed study of the program code showed that the probability for generating an acceptable input is approximately 0.9999996.

[5] Instructions: `https://github.com/OpenDSA/OpenDSA/tree/master/AV/Development/postfixEvaluationPRO.json` Exercise: in `https://github.com/OpenDSA/OpenDSA/tree/master/AV/Development/postfixEvaluationPRO.json`. Visited on 09/16/2019.

[6] Algorithm source code: `https://github.com/OpenDSA/OpenDSA/tree/master/SourceCode/Processing/Sorting/Quicksort.pde` Exercise: `https://github.com/OpenDSA/OpenDSA/tree/master/AV/Development/quicksort2PRO.js`. Visited on 09/16/2019.

3.2.3 Build-heap

The algorithm for this exercise is Algorithm 3. Note that this is a version where the heap array indexing begins from zero.

Algorithm 3 Build-Min-Heap [43].

```
1: procedure BUILD-MIN-HEAP(A)
2:     for i ← ⌊heap-size(A) / 2⌋ - 1 downto 0 do
3:         MIN-HEAPIFY(A, i)
4:     end for
5: end procedure

6: procedure MIN-HEAPIFY(A, i)
7:     l ← LEFT-CHILD-INDEX(i)
8:     r ← RIGHT-CHILD-INDEX(i)
9:     if l ≤ heap-size(A) and A[l] < A[i] then
10:        smallest ← l
11:    else
12:        smallest ← i
13:    end if
14:    if r ≤ heap-size(A) and A[r] < A[smallest] then
15:        smallest ← r
16:    end if
17:    if smallest ≠ i then
18:        SWAP(A[i], A[smallest])
19:        MIN-HEAPIFY(A, smallest)
20:    end if
21: end procedure

22: LEFT-CHILD-INDEX(i) = 2i + 1
23: RIGHT-CHILD-INDEX(i) = 2i + 2
```

Based on the build-heap algorithm, IR3 for the exercise are:

1. The minheap property must not hold in the beginning.
2. There must be at least one swap with left child.
3. There must be at least one swap with right child.
4. There must be at least one node in the main loop where a swap is not required.
5. MIN-HEAPIFY must call itself at least once such that a swap happens in the recursive call.
6. MIN-HEAPIFY must call itself at least once such that a swap does not happen in the recursive call.

Requirement 1 is obvious. Requirements 2–4 ensure the three different swap cases (left, right, none) are performed. Requirement 5 ensures the recursive nature of the algorithm. Requirement 6 shows that if a swap has been performed, it does not always cause a swap in the recursive step.

The input generation of Build-heap VAS exercise is shown in Listing 1 from OpenDSA[7].

Listing 1: Input generation of the Build-heap VAS exercise

```
12 function init() {
13   var nodeNum = 10;
14   if (bh) {
15     bh.clear();
16   }
17   $.fx.off = true;
18   var test = function (data) {
19     bh = av.ds.binheap(data, {size: nodeNum, stats: true,
                                  tree: false});
20     var stats = bh.stats;
21     bh.clear();
22     return (stats.swaps > 3 && stats.recursiveswaps > 0 &&
                 stats.leftswaps > 0 &&
23               stats.rightswaps > 0 &&
                 stats.partlyrecursiveswaps > 0);
24   };
25   initData = JSAV.utils.rand.numKeys(10, 100, nodeNum,
                 {test: test, tries: 50});
26
27   // Log the initial state of the exercise
28   var exInitData = {};
29   exInitData.gen_array = initData;
30   ODSA.AV.logExerciseInit(exInitData);
31
32   bh = av.ds.binheap(initData, {heapify: false});
33   swapIndex = av.variable(-1);
34   av._undo = [];
35   $.fx.off = false;
36   return bh;
37 }
```

The length of the input is 10 set on line 13. Line 19 sets other parameters for the input: random integer keys from range 10–99, validate the input with internal function `test` on lines 18–24, and run the generate-validate cycle at most 50 times. In the generator-validator function, variable `bh` on line 19 refers to a new a binary heap object whose class is defined in another file[8]. This binary heap is given the

[7] https://github.com/OpenDSA/OpenDSA/blob/master/AV/Binary/heapbuildPRO.js

[8] https://github.com/OpenDSA/OpenDSA/blob/master/DataStructures/binaryheap.js
Both URLs were visited on 04/25/2020.

random input and it is then heapified storing statistics on the process. Line 22 of Listing 1 compares straightly to IR3 of minheap described earlier:

- `stats.swaps > 3` corresponds requirement 1
- `stats.leftswaps > 0` corresponds requirement 2
- `stats.rightswaps > 0` corresponds requirement 3
- `stats.recursiveswaps > 0` corresponds requirement 5
- `stats.partlyrecursiveswaps > 0` corresponds requirement 6

Listing 2 from OpenDSA[9] shows how the build-heap statistics are computed. The condition on line 194 is executed if the recursively called HEAPIFY had performed a swap on one level higher. This means that at two consecutive levels a swap is performed, and therefore counter `recursiveswaps` increases. If a swap at the current level was performed, but another swap at the recursive step is not needed, the counter `partlyrecursiveswap` increases instead.

Listing 2: OpenDSA Minheap implementation for input validation

```
167 bhproto.heapify = function (pos, options) {
168   var size = this.heapsize(),
169     lpos = pos * 2,
170     rpos = pos * 2 + 1,
171     smallest = pos,
172     comp = this.options.compare,
173     step = this.options.steps ? this.jsav.step :
              function () {};
174   if (lpos <= size && comp(this.value(lpos - 1),
                              this.value(pos - 1)) < 0) {
175     smallest = lpos;
176   }
177   if (rpos <= size && comp(this.value(rpos - 1),
                        this.value(smallest - 1)) < 0) {
178     smallest = rpos;
179   }
180   if (smallest !== pos) {
181     if (this.options.stats) {
182       this.stats.swaps++;
183       if (smallest === lpos) { this.stats.leftswaps++; }
184       else { this.stats.rightswaps++; }
185     }
186     if (options && options.noAnimation) {
187       var tmp = this.value(pos - 1);
188       this.value(pos - 1, this.value(smallest - 1));
```

[9]Again, this is https://github.com/OpenDSA/OpenDSA/blob/master/DataStructures/binaryheap.js

```
189        this.value(smallest - 1, tmp);
190      } else {
191        this.swap(smallest - 1, pos - 1);
192      }
193      step.apply(this.jsav);
194      if (this.heapify(smallest, options) &&
             this.options.stats) {
195        this.stats.recursiveswaps++;
196      } else if (this.options.stats) {
197        this.stats.partlyrecursiveswaps++;
198      }
199      return true;
200    } else if (this.options.stats) {
201      this.stats.interrupted = true;
202    }
203    return false;
204  };
```

Requirement 4 for having a no-swap case is not guaranteed. It would be satisfied by adding condition
`stats.swaps - stats.recursiveswaps < Math.floor(nodeNum / 2)`
onto line 22 in Listing 1. Here the left side of the inequality is the number of swaps which are not recursive, meaning that they result from the main loop directly calling Min-Heapify. The right side of the inequality is the number of iterations in the main loop.

Another theoretically problematic case is the one where a valid input could not been generated within 50 random trials. By empirical test, it seems that function `test` in Listing 1 rejects the random input at 25% = 1/4 probability based on 1000 repeats. Adding condition `stats.swaps < nodeNum` does not alter the probability significantly. The probability that a valid input could not been generated after 50 trials is

$$\left(\frac{1}{4}\right)^{50} \approx 10^{-30},$$

which is practically never.

Requirement 4 is therefore the only one which is not explicitly satisfied. By empirical test, the exercise currently presents the student an input which fails requirement 4 at probability of 0.15. Adding this requirement to the input validation increases the failure probability of a single generation trial from 25% to 41%. Still, after maximum 50 trials, the total failure probability would be $0.41^{50} \approx 10^{-20}$, which is tolerable. Therefore this exercise does not currently meet IR3, but it could be corrected very easily.

3.2.4 Dijkstra's algorithm

The Aalto DSA course has a VAS exercise on Dijkstra's algoritm for single-source shortest paths in a graph that has positive edge weights. The graph in the exercise has integer weights on edges and the edges are not directed.

Listing 3: Dijkstra's algorithm in Java

```
1  // Find the unvisited vertex with the smalled distance
2  static int minVertex(Graph G, int[] D) {
3    int v = 0;  // Initialize v to any unvisited vertex;
4    for (int i=0; i<G.nodeCount(); i++)
5      if (G.getValue(i) != VISITED) { v = i; break; }
6    for (int i=0; i<G.nodeCount(); i++)
       // Now find smallest value
7      if ((G.getValue(i) != VISITED) && (D[i] < D[v]))
8        v = i;
9    return v;
10 }
11 // Compute shortest path distances from s, store them in D
12 static void Dijkstra(Graph G, int s, int[] D) {
13   for (int i=0; i<G.nodeCount(); i++)    // Initialize
14     D[i] = INFINITY;
15   D[s] = 0;
16   for (int i=0; i<G.nodeCount(); i++) {
       // Process the vertices
17     int v = minVertex(G, D); // Find next-closest vertex
18     G.setValue(v, VISITED);
19     if (D[v] == INFINITY) return; // Unreachable
20     int[] nList = G.neighbors(v);
21     for (int j=0; j<nList.length; j++) {
22       int w = nList[j];
23       if (D[w] > (D[v] + G.weight(v, w)))
24         D[w] = D[v] + G.weight(v, w);
25     }
26   }
27 }
```

Listing 3 shows an example version of Dijkstra's algorithm in Java, as presented in the OpenDSA textbook[10]. It is the array-based version: it has a boolean array to indicate which vertices are visited, and an integer array (array `D` on line 14), to store distances from the initial vertex `s`. The algorithm produces a shortest-paths tree: for each vertex that is reachable from the initial vertex, there is the distance from the initial vertex and a rooted tree which describes the shortest paths.

The algorithm iterates through all vertices beginning from line 16. For each vertex, it uses an auxiliary function `minVertex` and tries to find a vertex that is not yet visited, but is closest to the current vertex. If such a vertex is found and is reachable, then edge (i, v) is added to the shortest-paths tree. Adding the edge to the tree is not mentioned in Listing 3, but it is crucial for producing a useful output; Cormen, Leiserson, Rivest, and Stein [6, p. 658] state this step explicitly in their pseudocode.

On lines 20–25, the algorithm iterates the neighbor vertices of the current vertex

[10] `https://github.com/OpenDSA/OpenDSA/blob/master/SourceCode/Java/Graphs/Dijkstra.java`. Visited on 04/25/2020.

`i`. It updates the distance `D[j]` of neighbor `j` if the path from `i` is shorter than what is currently in `D`. The algorithm simply chooses the best neighbor at each vertex and updates the shortest paths for the neighbors of that neighbor.

IR3 is addressed in Listing 3 on lines 5, 7, 19 and 23, which check different acceptance criteria.

Line 5 is always executed unless the graph has no edges.

Line 7 has a condition which is true when there are several unvisited vertices, and the distance of the first one of them (in the vertex indexing order) is not smallest. In a larger perspective, there are three important cases: (i) minVertex returns the first vertex; (ii) there is only one valid choice in minVertex; (iii) there are several valid choices in minVertex. Case (i) happens when all the vertices are visited; this happens at the last iteration of the main loop on lines 16–26. Note that in the VAS exercise the student does not need to perform the iterations in minVertex but just choose a vertex if applicable.

Line 19 implies the graph should have at least one vertex which is unreachable from the initial vertex.

Line 23 updates the distance of a vertex if the current path explored is shorter than the previously seen one. This implies that there should be at least two distinct vertices, v_1 and v_2, in the graph such that when Dijkstra's algorithm is begun from v_0, there are two different paths from v_0 to v_1, and also two different paths from v_0 to v_1. In the case of v_1 the distance array `D` is updated, but in the case of v_2 the distance array is not updated. This means that the connected component of the graph including v_0 should have at least $x + 1$ edges when there are x vertices, thus at least two loops.

1. At some point of algorithm, there is unique choice for closest unvisited vertex.

2. At some point of algorithm, there are multiple equal choices for closest unvisited vertex.

3. There is at least one vertex which is unreachable from the initial vertex v_0.

4. There is a vertex u such that there are at least two different paths, p_1 and p_2, such that both lead from v_0 to u, p_1 is explored before p_2, and p_2 has lower weight than p_1.

5. There is a vertex u such that there are at least two different paths, p_1 and p_2, such that both lead from v_0 to u, p_1 is explored before p_2, and p_2 has equal or greater weight than p_1.

Listing 4: Input generation in Dijkstra's algorithm exercise

```
18  graph = jsav.ds.graph({
19    width: 400,
20    height: 400,
21    layout: "automatic",
22    directed: false
23  });
24  graphUtils.generate(graph, {weighted: true});
```

Listing 4 from OpenDSA[11] shows that the exercise actually calls the `graphUtils.js` module of the OpenDSA to generate a graph. The `generate` function on line 24 maps to function `generateGraph` in Listing 5 from OpenDSA[12]. Lines 65–67 show the default parameters: 7 vertices and 10 edges. On lines 77–79, each vertex is labeled with letters A, B, C, ... The function calls another function in the same file, `generateRandomEdges`, to generate the edges. This is shown in Listing 6 from OpenDSA[13].

Listing 5: Random graph generation in OpenDSA (1/2)

```
63  function generateGraph(graph, options) {
64    var defaultOptions = {
65      weighted: false,
66      nodes: 7,          // number of nodes
67      edges: 10          // number of edges
68    };
69    var opts = $.extend(defaultOptions, options),
70        weighted = opts.weighted,
71        nNodes = opts.nodes,
72        nEdges = opts.edges,
73        nodes = new Array(nNodes),
74        edges,
75        i;
76
77    // Generate the node values
78    for (i = 0; i < nNodes; i++) {
79      nodes[i] = String.fromCharCode(i + 65);
80    }
81    // Generate edges
82    edges = generateRandomEdges(nNodes, nEdges, weighted);
83    // Add the nodes to the graph
84    for (i = 0; i < nNodes; i++) {
85      graph.addNode(nodes[i]);
86    }
87    // Add the edges to the graph
88    for (i = 0; i < nEdges; i++) {
```

[11] `https://github.com/OpenDSA/OpenDSA/blob/master/AV/Graph/DijkstraPE.js`
[12] `https://github.com/OpenDSA/OpenDSA/blob/master/AV/Development/graphUtils.js`
[13] Also from `graphUtils.js`. Both URLs were visited on 04/25/2020.

```
89        var gNodes  = graph.nodes(),
90            start   = gNodes[edges[i].startIndex],
91            end     = gNodes[edges[i].endIndex],
92            eOpts   = edges[i].weight ?
                        {weight: edges[i].weight} : {};
93
94        graph.addEdge(start, end, eOpts);
95      }
96    }
```

Listing 6: Random graph generation in OpenDSA (2/2)

```
4   function generateRandomEdges(nNodes, nEdges, weighted) {
5     var edges = new Array(nEdges),
6         adjacencyMatrix,
7         index1,
8         index2,
9         i, j;
10
11      // Utility funciton to check whether the edge already
            exists
12      function isEligibleEdge(startIndex, endIndex) {
13        if ((startIndex === endIndex) ||
14            (adjacencyMatrix[startIndex][endIndex] === 1) ||
15            (adjacencyMatrix[endIndex][startIndex] === 1)) {
16          return false;
17        }
18        return true;
19      }
20
21      //Create the adjacencyMatrix
22      adjacencyMatrix = new Array(nNodes);
23      for (i = 0; i < nNodes; i++) {
24        adjacencyMatrix[i] = new Array(nNodes);
25      }
26      //Initialize the adjacency matrix
27      for (i = 0; i < nNodes; i++) {
28        for (j = 0; j < nNodes; j++) {
29          adjacencyMatrix[i][j] = 0;
30        }
31      }
32      for (i = 0; i < nEdges; i++) {
33        do {
34          index1 = Math.floor((Math.random() * nNodes));
35          index2 = Math.floor((Math.random() * nNodes));
36        } while (!isEligibleEdge(index1, index2));
37        edges[i] = {
38          startIndex: index1,
```

```
39        endIndex: index2
40      };
41      if (weighted) {
42        edges[i].weight = 1 + Math.floor(
            (Math.random() * 9));
43      }
44      // add the edge to the matrix
45      adjacencyMatrix[index1][index2] = 1;
46      // adjacencyMatrix[index2][index1] = 1;
47    }
48
49    return edges;
50  }
```

The loop on line 32 in Listing 6 adds exactly `nEdges` edges to the graph. Lines 34–35 and function `isEligibleEdge` on lines 12–19 define the conditions for creating each edge:

1. Start and end vertices of the edge are chosen randomly and independently.
2. Each vertex has equal probability to be chosen.
3. Start and end vertices must be different.
4. There can be only one edge from vertex u to vertex v.
5. If there is edge from u to v, there cannot be an edge from v to u.

It is known that the spanning tree of a undirected graph with only one connected component has $|V| - 1$ edges; it takes that amount of edges to connect $|V|$ vertices. Then, if Listing 6 has already created $|V| - 1$ edges and they connect all the $|V|$ vertices, adding randomly $nEdges-(|V|-1)$ more edges creates $[1, nEdges-(|V|-1)]$ loops.

Clearly the proposed IR3 requirements for Dijkstra's algorithm are not guaranteed. The random graph generation conflicts with the requirements. For example, the subrequirement "There is at least one vertex which is unreachable from the initial vertex v_0" is currently fulfilled at probability of 0.03 based on 100 generation of exercise instances. Therefore this exercise does not meet IR3.

3.3 Reproducibility of JSAV exercise recordings

This subsection studies RQ6: *Is it possible to construct a "player application" for the current submissions of JSAV exercises?*

To be able to construct a player application for the current, JSAV-based VAS exercise recordings on the Aalto University DSA course, the data of the actual exercise recordings must be inspected. The exercise recording must meet two requirements in order to support replaying the student's actions. These requirements are equal for manual analysis and automatic detection of misconceptions. The *Reproducibility Requirements* (RR) are:

- RR1. The input must be included in the recording, or a unique, unambiguous input can be deduced from the steps in the recording.
- RR2. All the steps in the recording must have a unique, unambiguous interpretation on student's choices such that it is clear on which step the student's steps begin to differ from the execution path of the correct algorithm.

The same subset of JSAV-based VAS exercises is used here than in Section 3.2.

3.3.1 Evaluating postfix expression

The following describes the exercise recordings from years 2016–2019 on the Aalto University DSA course. The recording of this exercise consists of the states of the output array. The initial input is not given, therefore the exercise fails RR1. The exercise also fails RR2, as it is not indicated which particular left and right parentheses in the input are processed at which steps. Therefore the submissions to the *Evaluating postfix expression* exercise are not currently reproducible.

3.3.2 Quicksort

The recording of this exercise consists of the states of the array to be sorted. The initial input is given as the first state, therefore RR1 is satisfied. The swap operations can be reconstructed by comparing pairwise states. This alone would be enough to meet RR2. Moreover, the recording has the same color coding as the GUI of the exercise: pivot, active area for current recursive call, and inactive areas, both processed and unprocessed are shown. Therefore all student's actions can be reproduced, and the exercise is reproducible.

3.3.3 Build-heap

The Build-heap exercise records the binary heaps as an array, and at each step, the whole array is recorded. Listing 7 shows an example step. The first step shows the array before any swaps have been performed, and therefore the first step is the actual input. Therefore the exercise satisfies RR1.

Listing 7: Example step of a JSON recording of the Build-heap exercise.

```
{
  "ind": [ {"v":43}, {"v":55}, {"v":64}, {"v":65}, {"v":26},
           {"v":48}, {"v":41},{"v":55},
           {"v":54,"cls":["jsavhighlight"]}, {"v":41}
         ],
  "style" : "height: 60px; width: 301px;",
  "classes" : ["jsavcenter"]
},
```

Otherwise it seems that the recording presents a single swap in two adjacent steps. In the first step of a swap, the lower element involving the swap has additional

attribute `cls` having value `jsavhighlight`. In the second step of the swap, there is no highlight, but comparing the values of the array to the previous step, the previously highlighted element and its parent have been swapped. The final state in the recording shows the output of the algorithm.

The swap operation is at only one location in the Build-heap algorithm (see Algorithm 3). Furthermore, we know that the subalgorithm MIN-HEAPIFY works recursively in a top-down manner. Every time we see a swap that involves a lower array index than what was previously encountered, we know that a new round in the main loop of BUILD-MIN-HEAP has begun. Clearly Build-Heap also satisfies RR2, and therefore it is reproducible.

3.3.4 Dijkstra's algorithm

Listing 8 shows a step of a recording of the *Dijkstra's algorithm* exercise. Field `n` defines seven vertices labeled as A . . . G. The subfield `n.css` contains CSS code which defines the absolute coordinates of the vertex in the graphical layout of the graph. Field `e` defines edges. Each entry has indices of start and end vertices and then additional information on the edge. The `path` field defines the start and end coordinates of the edge in the graphical layout. Field `w` defines the weight of the edge, which is a positive integer. Thus every step records both the graph and its layout entirely. This also confirms that RR1 is satisfied for Dijkstra's algorithm JSAV exercise.

Listing 8: An example step of a recording of the Dijkstra's algorithm exercise.

```
{
  "n": [{
    "v": "A",
    "cls": ["marked"],
    "css": "position: absolute; left: 218.611px; top: 306px;"
  }, ... {
    "v": "G",
    "css": "position: absolute; left: 0px; top: 0px;"
  }],
  "e": [
    [0, 2, { "a": { "fill": "none",
        "stroke": "#000",
        "path": [ ["M", 235, 307], ["L", 193, 163] ],
        "width": 400,
        "height": 400,
        "opacity": 1
      },
      "l": "4",
      "w": 4
    }],
    [1, 4, { ... "w": 9}], [2, 1, { ... "w": 6}],
    [2, 4, { ... "w": 7}], [3, 2, { ... "w": 7}],
    [4, 3, { ... "w": 3}], [5, 2, { ... "w": 6}],
```

```
    [5, 0, { ... "w": 6}], [5, 3, { ... "w": 1}],
    [6, 2, { ... "w": 7}]
  ]
}
```

Attribute `"cls": ["marked"]` is added to some vertices in the further steps of the recording. Otherwise the steps are identical. The ”marked” attribute indicates whether the vertex is marked as visited. This is similar to the line 18 in Listing 3. Therefore the recording lists the order in which the next-closest vertices are added into the single-source shortest paths spanning tree. Moreover, incorrect submissions also have edges with the ”marked” attribute if both of the vertices of the edge were visited or unvisited. Based on this property, the *Dijkstra's algorithm* exercise is reproducible.

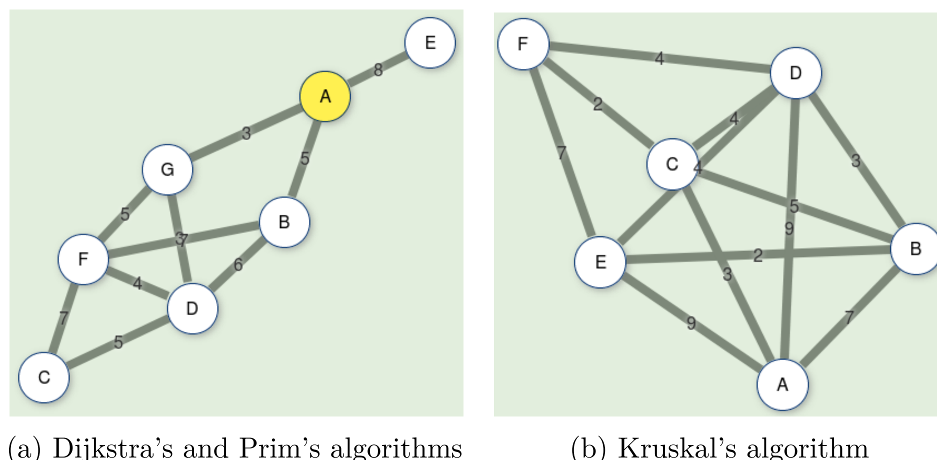
3.4 Improved graph algorithm exercises

This subsection provides a practical demonstration on how to improve the misconception awareness of a VAS exercise. Therefore it is a continuation of Section 3.2.

The software related to the VAS exercise *Dijkstra's algorithm* is further developed to meet IR3. The original need for this work was to correct a bug in the exercise. However, as the bug could be corrected by writing a new algorithm which both generates the input and decides the graph layout, the algorithm was written to fulfill IR3. Thus the description of this practical software development work was also included in this thesis. The solution was also directly applicable to the *Prim's algorithm* and *Kruskal's algorithm* exercises.

This subsection discusses three types of algorithms: elementary graph algorithms, input generator algorithms for VAS exercises, and algorithms which create visual representations of graphs. The first paragraph gives a brief introduction to principles of visualising graphs. Then the discussion continues on OpenDSA exercises featuring elementary graph algorithms. It is shown that these exercises often produce low-quality visualisations. Finally, the subsection proposes an improvement on the exercises based on principles of graph visualisation.

Graph drawing is methods and algorithms which automatically generate a visual representation of graph data for human analysis. Thus graph drawing is an intersecting field of algorithmics and information visualisation. Typically graph drawing methods produce a two-dimensional, static diagram of the data where vertices (nodes) are circles and edges are line segments connecting the nodes. Examples of these graph visualisations are shown in Figure 8. A *graph layout algorithm* computes coordinates for the vertices and routing for the edges. To maximise visual readability for humans, a layout algorithm tries to meet *aesthetic requirements*. Typical requirements are: showing symmetry, avoiding crossing lines, maximising angles in crossings, and minimising the drawing area. One well-known example is the *force-directed layout algorithm* which uses physical simulation: nodes are modelled as repelling, opposite charges to each other, while the edges are modelled as springs which try to maintain an optimal length. The locations of the edges will stabilise after sufficient number of iterations. [4, 10, 47, 50, pp. 7–9] If the graph is relatively small (low number of



(a) Dijkstra's and Prim's algorithms (b) Kruskal's algorithm

Figure 8: The graph algorithm VAS exercises with the original input generator and layout algorithm.

vertices) and is relatively sparse (low edge-vertex ratio), one can expect that the force-directed layout has no crossing edges.

OpenDSA has three VAS exercises on graph algorithms that have problems with showing the graph data structure visually: *Dijkstra's algorithm*, *Prim's algorithm*, and *Kruskal's algorithm*. Dijkstra's algorithm was already discussed in Section 3.2.4. For reference, Prim's and Kruskal's algorithms are described in Cormen et al. [6, p. 631–636]. The program code implementing these exercises is in the files `PrimAVPE.*`, `DijkstraPE.*`, and `KruskalPE.*` in the OpenDSA[14]. Essentially, the VAS exercises of all these algorithms feature an undirected graph with positive integer weights.

Figure 8 shows examples of initial situations of these exercises. Depending on the exercise, the student must either click on the nodes or on the edges to simulate the algorithm. Dijkstra's and Prim's algorithm exercises have seven (7) nodes and ten (10) edges. Kruskal's algorithm exercise has six (6) nodes and twelve (12) edges. The graph is generated randomly and then a force-directed layout algorithm is used to compute the locations of the nodes on the screen. The input generator algorithm is in the file `graphUtils.js`[15]. The graph layout algorithm is in the JSAV library[16] under the comment `Graph layout algorithm based on Graph Dracula`.

The graph layouts tend to have overlapping elements. Edges (B, F) and (D, G) in Figure 8 (a) are one example. These two edges have overlapping weight labels "7" and "3". It is hard to determine which label belongs to which edge. Figure 8 (b) hows how node C overlaps with the weight label "4" edge (D, E). Also the visual labels "5" and '7" of edges (B, C) and (A, D) is confusing. There are also situations where two nodes have distance of less than two node diameters from each other. Then it becomes hard to read or interact with the edge between them. The probability that the layout is "problematic" is 40% for Prim's and Dijkstra's algorithm exercises,

[14]`https://github.com/OpenDSA/OpenDSA/blob/master/AV/Graph/`. Visited on 09/25/2020.

[15]`https://github.com/OpenDSA/OpenDSA/blob/master/AV/graphUtils.js`.

[16]`https://github.com/vkaravir/JSAV/blob/master/src/graph.js`. Both URLs were visited on 09/25/2020.

and 50% for Kruskal's algorithm exercise, based on visual inspection of 100 sample layouts. Clearly the aesthetic requirements for graph drawing are not often met.

There are several causes of overlap. First, force-directed layout algorithm in JSAV does only consider overlap of vertices, not edges and edge weight labels. Second, Kruskal's algorithm exercise has dense, random graph, which additionally increases overlap. One solution is to modify the algorithm by adding repulsive forces between all pairs of edge labels and nodes. Symmetric graph elements can still cause crossing diagonals. The vertex quartet (B, D, F, G) in Figure 8 (a) is an example of this.

Instead of modifying the force-directed layout algorithm, a new graph generator with static layout was written. The updated source code for the exercises is published in [49], particularly in the files `AV/Development/graphUtils.js`, `AV/Graph/DijkstraPE.*`, `AV/Graph/KruskalPE.*`, and `AV/Graph/PrimPE.*`.

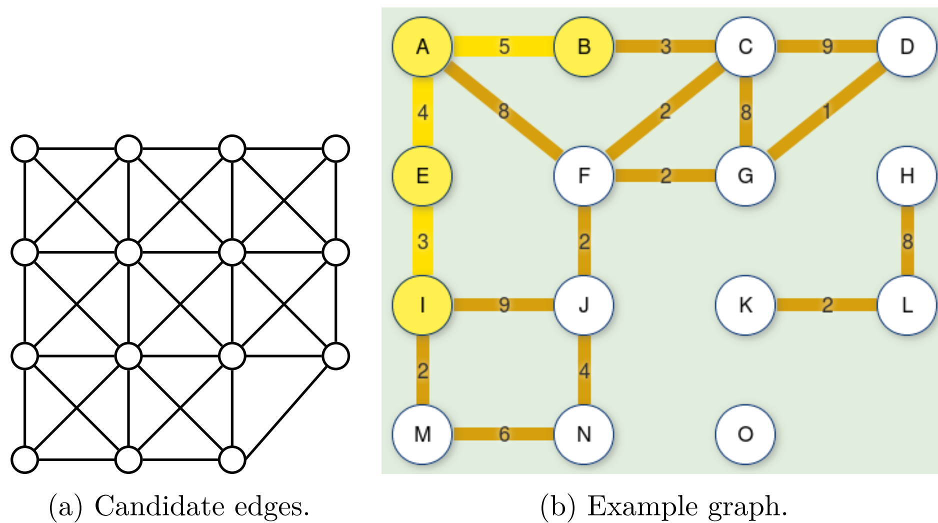
The new input generator produces a planar graph with no crossing edges and vertices placed on a rectangular grid. The basis of the graph is shown in Figure 9 (a): there are 15 nodes and the set of *candidate edges*: each rectangle has four edges and two diagonal edges. Diagonal edges in the same rectangle are mutually exclusive, and when the set of candidate edges is created, one of the two diagonals is chosen randomly for each rectangle. Function `candidateEdges()` in *graphUtils.js* creates the candidate grid.

The generator algorithm continues by assigning vertices into two connected components; see function `verticesToComponents()` in *graphUtils.js*. Component C_1 has 12 vertices while C_2 has 3 vertices. First all vertices are initialised to C_1, then a random vertex is chosen from the rightmost column in Figure 8 (a), and finally a random depth-first search is run from it on the candidate edge graph with depth limit of 2, until total 3 vertices have been visited. After vertices are assigned to components, the edges in the candidate grid that lead from one component to the other, are filtered out and the rest of the edges are assigned to either component: E_1 for C_1 and E_2 for C_2. For reference, see function `edgesToComponents()` in *graphUtils.js*. Finally, 12 edges from E_1 and 2 edges from E_2 are chosen randomly, and the weights for each edge is chosen randomly and independently.

The grid-based approach which creates a planar graph with two components is already a step forward to meet the input requirements, but the random graph that is obtained is still validated. File *DijkstraPE.js* has a validator function `testDijkstra()` which checks the IR3 as discussed in Section3.2.4. In practise, the Dijkstra's algorithm is run on the input and statistics for each essential feature of the algorithm is counted. Input generation is repeated until all the requirements are met or 100 repeations have passed. In the latter case the random graph that met most requirements is selected.

The result of improved Dijkstra's VAS exercise is shown in Figure 9 (b). The colors and line width of the edges in the exercise are altered to have higher contrast between the edge and its weight label for both explored and unexplored edges. The visualisation has also been tested for color vision deficiencies with Coblis Color Blindness Simulator [51].

Kruskal's algorithm exercise has the same generator algorithm as Dijkstra's algorithm exercise, but with one connected component having $|V| = 11$ vertices



(a) Candidate edges. (b) Example graph.

Figure 9: The improved *Dijkstra's algorithm* VAS exercise.

and $|E| = 14$ edges. No specific validator function is used. Regarding IR3, this is enough to activate all essential features of the algorithm. The number of connected components does not affect the execution of Kruskal's algorithm; it is only essential that the initial graph has more edges than what is required for the minimum spanning tree (or forest). The edges are given random integer weights from range $[1, 9]$, and the student is instructed to choose an edge with lowest alphabetic vertex label, when there are multiple equal choices.

Prim's algorithm is almost identical to Dijkstra's algorithm with the exception that distance of an unvisited vertex is computed from the nearest visited neighbor, not from the shortest path of the start vertex. Therefore the essential features of Prim's algorithm are the same as Dijkstra's algorithm. The random input in Prim's algorithm VAS exercise is generated with the same parameters as in the Dijkstra's algorithm VAS exercise. The input is validated with Prim's algorithm in function `testPrim()` in the file *PrimPE.js*.

3.5 Input generation by random search

Algorithm 4 Brute-force misconception aware input generation for VAS

```
1: procedure Generate-VAS-input-BF(exercise, Algorithms, length, maxT)
2:     bestInput ← ε
3:     bestUniqueness ← 0
4:     t ← 0
5:     repeat
6:         Sequences ← ∅
7:         input ← Random-Input(exercise, length)
8:         for a ∈ Algorithms do
9:             Sequences ← Sequences ∪ a(input)
10:        end for
11:        if |Sequences| > bestUniqueness then
12:            bestInput ← input
13:            bestUniqueness ← |Sequences|
14:        end if
15:        t ← t + 1
16:    until |Sequences| = |Algorithms| or t = maxT
17:    return bestInput
18: end procedure
```

Algorithm 4 shows a proposal for a brute-force approach of misconception aware input generation. A random input is generated and then validated against the requirements IR3–IR5. If it does not pass the requirements, a new random input is generated.

Input parameters are the following. Parameter *exercise* specifies the type of exercise as in, for example, Table 13 in Appendix A. It is just an identifier value for the exercise. Parameter *length* denotes the length of the input. Its semantics depend on the type of the exercise. It is a length of an array for example, sorting and binary heap. For graph algorithm exercises the length of an input could be a set of parameters, like the number of vertices and edges. Parameter *Algorithms* is a set containing both the correct algorithm and the known AMMs for the exercise. When an algorithm from this set is run, its output is the execution sequence (program trace) of that algorithm with the given input. Parameter $maxT > 0 \in \mathbb{Z}$ is the number of iterations in the random search. The generation algorithm tries at most $maxT$ times to generate a random input with which all the *Algorithms* produce a different sequence.

The subprocedure Random-Input generates a random input with the given parameter *length*. It tries to fulfill IR3 for the given *exercise*. By default, it is assumed that Random-Input always returns an input regardless of whether IR3 could be fulfilled with given *length*.

The algorithm works as follows. Variable *bestInput*, initially the empty string, stores the best random input generated thus far. Its goodness is depicted by variable $bestUniqueness \geq 0 \in \mathbb{Z}$: how many different sequences the current value of

bestInput produces when all algorithms in *Algorithms* are run with that input. One trial on an input is simple. Generate a random input candidate on line 7. Run this input on each given algorithm on lines 8–13. If some algorithm generated a different sequence than what was previously seen, store this sequence into set of *Sequences*. Lines 14–17 update *bestInput* and *bestUniqueness* if a new record in number of different sequences was hit.

The algorithm ends successfully when there is an input that produces unique sequence for all given algorithms. In this case, IR4 and IR5 are fulfilled. Alternatively, *maxT* trials was reached, and IR4, IR5, or both have failed. However, the algorithm always returns the best candidate stored in *bestInput*.

The algorithm is a *randomised algorithm*: it makes random choices during execution, and this also affects the running time and the output. By default, it is assumed that *maxT* has a finite value. This is the *Monte Carlo* version of the algorithm: the running time is fixed, but the output may be incorrect at some probability. Correspondingly, the *Las Vegas* version is the one where $maxT \to \infty$: the output is always correct but the running time varies. [29, p. 9] Incorrectness in this case means that the input does not fulfill IR3, IR4, or IR5. The optimal value for *maxT*, whether being finite or infinite, should be studied empirically. Also note that the algorithm could be modified to prefer IR4 over IR5, if both of then cannot be fulfilled.

Input generation by algorithm GENERATE-VAS-INPUT-BF is easy from programmer's view, but it has unclear computational complexity. Hypothetically the practical running time grows by size of set *Algorithms*. However, longer input length offers larger solution space, and therefore finding an longer input which produces different sequence with each algorithm might require less trials. As this is a random search of a large input space, a second terminating condition *maxT* must be set to ensure that input generation for a single exercise instance does not take too long.

4 Empirical methods and data

This section discusses empirical methods and data for the research question RQ7: *Which AMMs for the VAS exercises have matches in the submission data?* The scope is narrowed to the Build-heap VAS exercise. Subsection 4.1 discusses the importance of replication. Subsections 4.2–4.4 describe methods of several replicated studies. Subsection 4.5 describes tools that were developed for the empirical study.

4.1 Replication as a scientific method

A *replicated study* is a scientific experiment where an experimental procedure of a prior study has been repeated. The purpose of a replicated study is to either confirm or disconfirm the hypotheses of an earlier study. Therefore replication is a basic principle of the scientific method. [13, 40]

Replication has several specific functions [13, 40]:

- Control the sample error: if only a subset of subjects (people, other biological bodies, or inanimate objects) were randomly chosen for the experiment, the random choice affects the results.
- Control for fraud or artifacts: it is possible that the researchers have had an effect on the results when doing an experiment, and this might be either on purpose or by unconsciousness. For example, different researcher or different laboratory setting might affect the results. Regarding computing education research, demographics, classroom climate (teachers, students' mood), and culture of the school affect results.
- To generalise the results for a larger or different population.
- To confirm the hypothesis of the earlier study using a different method. For example, the Avogadro number in chemistry has been verified with different experimental approaches such as X-ray diffraction and electrolysis.

Schmidt [40] defines two categories of replication studies which have widely been used. A *direct replication* is a study which uses the same sampling techniques and experiments than the prior study. A *conceptual replication* uses different methodology, but still aims to confirm the same hypothesis.

The current need of replication in computing education research is high. The amount of replication studies in computing education seems to be only 2–3% on years 2009–2018, when concerning articles that explicitly discuss replication. [13] This thesis aims to perform both a direct replication and a conceptual replication.

4.2 Direct replication of the Build-heap studies

The first part of answering RQ7 is the direct replication of studies [19] and [43]. The purpose of this part is to experiment with a different dataset to confirm the earlier results. This would support the hypothesis on the existence of specific, misconceived

algorithms for Build-heap similar to Karavirta, Korhonen, and Seppälä [19] that performed a conceptual replication of Seppälä, Malmi, and Korhonen [43].

The data for the current experiment contains the Build-heap submissions from years 2016–2019 with total of $N = 1430$ submissions. Both in this experiment and the earlier study by Karavirta et al. the data consists of exercise recordings from a JSAV-based Build-Heap VAS exercise on a Data Structures and Algorithms course, intended to be taken at undergraduate level at Aalto University. Section 3.2.3 shows the essential Algorithm 3.

Table 6: Differences with Build-heap misconception studies.

Study	Data from years	VAS software	Heap size	Submissions
Seppälä et al. [43]	2005	TRAKLA2	15	884
Karavirta et al. [19]	2012	JSAV	10	373
This thesis	2016–2019	JSAV	10	1430

The differences with the studies are shown in Table 6. There are several variables changed between [43] and [19]: the VAS software, the heap size, and sample size. However, the same Java-based implementations of AMMs and the submission classifier were used in [43] and [19]. The software has been re-implemented in this study. The sample size is also larger.

AMMs similar to the study of Karavirta et al. [19] are implemented in the Misconception matcher. The following AMM descriptions are from the article.

Heapify-with-Father This misconceived algorithm does (possibly two) swaps with the father node, if necessary. This is easy to recognize by the swaps the student does in case both children are smaller than the father. Typically, first the left child is swapped and if it is smaller than the right child, a new swap follows. The correct algorithm does at most only one swap.

Left-to-Right In this misconceived algorithm, the iterative heapify goes through each level of the binary tree from left to right, and, after reaching the rightmost node, moves one level up.

No-Recursion The recursive step at the end of the algorithm is missing, resulting in a violation of the heap-order property in some cases (i.e., in case a swap occurs, and there should be a new swap somewhere below the node).

Single-Skip In this variant, the student follows the correct solution, but a single step is missing.

Top-Down This is a misconceived algorithm that applies Min-Heapify for all nodes starting from the root node.

Delayed-Recursion This is a misconception related to No-Recursion in which the student later realizes that the heap-order property is not satisfied, and then returns to fix the property to hold again.

Smallest-Instantly-Up This is a misconceived algorithm in which the tree is traversed in level order, swapping the traversed key with the smallest key found in the subtree rooted at the node being traversed.

Maximum-Heap The student builds a maximum heap algorithm instead of a minimum heap.

Wrong-Duplicate This misconception, in order to be revealed, requires that the input data contains two equal keys in such a way that they appear as left and right child for a single parent node that needs to be swapped. While the correct algorithm swaps the the left child in this case, the Wrong-Duplicate version swaps the right child.

Other This category includes several other misconceived algorithms and combinations of these (i.e., Smallest-Instantly-Up done in Right-to-Left order) that only catch a couple of student solution instances. [19, p. 63]

In this study, no source code for AMMs was available, and therefore some interpretation is required. Seppälä et al. [43, p. 249] do not discuss the order of children in *Heapify-with-Father*, but mention that this variant results in a valid heap ”if executed recursively”. Therefore all subvariants of *Heapify-with-Father* are considered relevant: both ”left child first” and ”right child first”, and both recursive and nonrecursive versions. However, the order of comparison for children is assumed constant.

The *Other* category was only described to include the ”Smallest-Instantly-Up done in Right-to-Left order” [19, p. 63], which was interpreted as the Build-heap main loop variant (0,2,1,4,3). See Section 5.2 for elaborate discussion on the main loop variants. Seppälä et al. [43, p. 251] included the *Left-to-Right* AMM in the *Other* category, but in Karavirta et al. [19] and this thesis it has its own category.

Table 7: Main submission classes in [43] and [19].

Algorithm	Finished	Unfinished
Correct	A	–
Misconceived	B	C
Unknown	–	D

The similarity between the student's sequence and each AMM is computed following Algorithm 2. The student's sequence is classified into one of the four main categories in Table 7 as follows. First, if the student's sequence matches perfectly to the correct solution, it is labeled as class A (Correct, Finished). Second, if there is a perfect match with sequence of an AMM, the submission is labeled as class B (Misconceived, Finished). Third, if no perfect match with any AMM was found and the sequence of some AMM matches at least two steps further than the correct algorithm, choose class C (Misconceived, Unfinished). Finally, if any of the previous rules do not apply, choose class D (Unknown, Unfinished). If there are equally well matching AMMs after choosing class B or C, choose the first applicable AMM in the

following order: *Wrong-Duplicate*, *Heapify-with-Father*, *Left-to-Right*, *No-Recursion*, *Single-Skip*, *Top-Down*, *Delayed-Recursion*, *Smallest-Instantly-Up*, and *Other* [43, 19]. Note that the Maximum-Heap AMM was not mentioned in the preference list, which means this AMM is not chosen if there are multiple equally matching algorithm AMMs.

The *Single-Skip* AMM produces several candidate sequences with the same input [43, p. 250]. Single-Skip is implemented in this thesis such that first the correct Build-Min-Heap algorithm is run on the input, and the sequence of *swaps* is stored with information on which of the swaps are *recursive*: their cause is in Algorithm 3 when Min-Heapify calls itself. Then a Single-Skip *sequence generator* algorithm produces k sequences: the correct sequence has k swaps, and each Single-Skip sequence has one of these swaps omitted. Additionally, the recursive swaps immediately following the omitted swap are omitted, and the sequence continues from the next nonrecursive swap. All of the resulting Single-Skip sequences are matched against student's sequence with Algorithm 2, and the highest score is stored as the similarity value for Single-Skip.

The *Delayed-Recursion* AMM also produces several candidate sequences. There are two variants: the first one executes recursive swaps after each level of the binary tree representation, and the second one executes all recursive swaps in the end of the algorithm. However, the research article by Seppälä et al. [43] offers possibility for interpretation: there is no explicit statement that the relative order between recursive swaps is constant. Implicitly, the number of subvariants for Delayed-Recursion is reported to be "two", whereas the Single-Skip has "variants". [43, p. 250]. Karavirta et al. [19, p. 63] redescribe Delayed-Recursion as the student "fixing" the heap property. To be realistic, these "fixing" swaps can happen in an arbitrary order. Allowing variance in the order of recursive swaps requires either creating a candidate sequence for each permutation of the recursive swaps, or modifying Algorithm 5 such that it has a specific "recursive mode" for matching consecutive recursive swaps in arbitrary order. Therefore, in the replicated study executed in this thesis, the Delayed-Recursion AMM produces exactly two candidate sequences as described above. The recursive swaps are delayed by storing them into a *First In, First Out* (FIFO) queue which is emptied between levels or in the end, depending on the variant. These two candidate sequences of Delayed-Recursion are then matched against student's sequence with the same Algorithm 2 as the other AMMs.

4.3 Forming hypotheses for Build-heap misconceptions

Misconceptions of the Build-heap exercise, related to RQ7, were studied further in a *conceptual* replication of the earlier studies [19, 43]. This section describes manual analysis of Build-heap submissions. The purpose of manual analysis is to (i) verify the existence of known misconceptions, (ii) to find hypotheses for new misconceptions, and (iii) to establish a ground truth for evaluating machine classification of misconceptions.

This thesis and the previous studies use a different process to form hypotheses for AMMs. Seppälä et al. [43, p. 245] manually inspected submissions that could

not be explained with already known AMMs, wrote new hypothetical AMM for each unexplained submission, and run the misconception classifier including the new AMM. This process is *iterative*: each time a new hypothetical AMM is introduced, it matches automatically to some submissions, and thus the set of unknown submissions decreases.

This thesis uses a *batch analysis process*: first all submissions are analysed manually, then hypothetical AMMs are written, and finally the existence of hypothetical AMMs are tested against the same data automatically. This exhaustive noniterative procedure is used to counteract a hypothesis bias that could arise from building hypotheses on hypotheses. In other words, this process ensures that even if the sequence of AMM A matches perfectly to submission s, submission s can still be analysed manually and there is possibility to form another hypothesis based on it.

The data here is the same as in the direct replication study with total $N = 1430$ submissions. First all of the year 2016 and 2018 submissions were reviewed. Then the submissions from 2017 and 2019 which received less than 100% grade were reviewed. The 2017 and 2019 which were graded automatically as correct were given the ”correct” manual class based on assumption that they are either exactly correct or have first the correct sequence and then some extra steps in the end.

Initially each submission was tried to be classified into one of the following seven classes: either *correct*, *inexplicable*, or one of the five known AMMs described in [43]: *No-recursion*, *Heapify-with-Father*, *Delayed recursion*, *Left-to-right*, or *Smallest-instantly-up*.

Additional information was also recorded for hypothesis building. Some *features* were also marked for submissions, meaning independent Boolean valued indicating some property holding in the submission sequence. These features are: (i) *Lesser-down swaps*: an erroneous swap with a parent node and its child node where the parent is less than the child. (ii) *Jump swaps*: any swap that is not performed with a parent node and its child. (iii) *Correct but unfinished*: correct steps from the beginning, but the end of the sequence is missing. (iv) *Slip or skip*: some swap is skipped or a single swap with wrong child was performed. Moreover, some submissions were given a written description, such as ”indices of main loop could be (3,4,1,2,0) with delayed recursion, but there are several skips or slips”, or ”Either correct with a slip or Heapify-with-Father”.

The variance in AMMs seemed larger than what could be described with the aforementioned categorisation. The correct Build-heap algorithm contains two parts: the main loop iteration and the Heapify subprocedure. Therefore a hypothetical, two-dimensional categorisation was formed where one variable is the *main loop variant* and another variable is the *Heapify variant*. The results of this categorisation is shown in section 5.2.

When there were several equally matching algorithm variant candidates for a submission, the principle of Occam’s razor was used: choose the simplest possible explanation. First, if a sequence contained single pair of steps where both of the children were swapped with their parent node, it was interpreted as the correct performance with a slip; a Heapify-with-Father decision was done only if there was several Heapify-with-Father swap pairs. Second, there might be several traversal

orders for the main loop, if some nodes did not require swaps in the Heapify procedure. The order of preference was *Correct*, *Zigzag RL*, *Level RL*, and other. This order was decided by the frequency of main loop variants in the manual analysis of year 2018 submissions; this required two examination passes . Third, a *No-Recursion* variant was chosen only if there were several recursive steps missing; otherwise the submission was decided to be a correct submission with a skip. Fourth, if a submission could be interpreted either as an AMM with several slips and skips, or as an unrecognised submission where the student had just played with the exercise without further understanding, it was labeled as the latter.

4.4 Machine classification of new Build-heap misconceptions

This subsection is the second part of the conceptual replication of Build-heap study, and it still relates to RQ7.

The existence of hypothetical AMMs described in Section 5.2 is tested by implementing them in the Misconception matcher. Also the fallback features are examined in the following order: No-Swaps, Extra-Steps-After-Correct, Swaps-Resemble-Build-heap, Nonsystematic-Build-heap, Legal-swaps, and Legal-swap-indices. The first applying feature decides the correspondingly named *fallback class* for the submission. If none of the features apply, the submission is classified as Unrecognised.

Testing properties *No-Swaps*, *Extra-Steps-After-Correct* and *Nonsystematic-Build-heap* is trivial. The rules for properties *Legal-swap-indices* and *Legal-swap* can be easily constructed from Algorithm 3.

Algorithm 5 Build-min-heap explainability of a swap sequence

```
 1: procedure Swaps-Resemble-Build-Heap(swaps, heapSize)
 2:     if |swaps| < 2 then return False
 3:     swapIndex ← 0
 4:     mainLoopIndex ← ⌊heapSize/2⌋
 5:     previousChild ← −1
 6:     for i = 0 ... |swaps| − 1 do
 7:         swap ← swaps[i]
 8:         if ⌊(swap.child − 1)/2⌋ ≠ swap.parent then return False
 9:         if previousChild ≠ swap.parent then
10:             if parent ≥ mainLoopIndex then return False
11:             mainLoopIndex ← swap.parent
12:         end if
13:         previousChild ← swap.child
14:     end for
15:     return True
16: end procedure
```

Algorithm 5 tests property *Swaps-Resemble-Build-heap*. Input $heapSize \in \mathbb{Z}_+$ is the size of the heap array. Input *swaps* is a sequence of pairs $(a_i, b_i), 0 \leq i < |swaps|$ such that each pair is a pair of indices in the heap array where the first index is less

than the second index: $0 \leq a_i < b_i < heapSize$. Line 8 tests the *Legal-swap-indices* property. Line 9 tests whether the parent index of the current swap is the same than the child index of the previous swap; if so, it is assumed that this is a recursive swap in the MIN-HEAPIFY subprocedure. Otherwise it is assumed that the main loop index in the BUILD-MIN-HEAP has decreased, and thus the parent index of the current swap should be the new main loop index.

Finally, some statistics of the machine classification are studied: number of recognised AMMs, frequencies of fallback classes, and the failure rates of IR4 and IR5. This is done both for the *Default* classifier (conceptual replication study) and the *Test* classifier (direct replication study). Because both of the classifiers have conceptually similar classes, the mapping in Table 8 is used to facilitate comparison.

Table 8: Class mapping from Test classifier to Default classifier.

	Main loop variant							
Heapify variant	Correct	Zigzag up/RL	Zigzag up/LR	Level LR	Top-down	Zigzag down/LR	Zigzag down/RL	Inorder
Correct	Correct	Other		Left-to-Right	Top-down	Other		
No-recursion	No-recursion							
Delayed recursion	Delayed-recursion							
Heapify-with-father (LR, LR recursive, RL, RL recursive)	Heapify-with-father Heapify-with-father Heapify-with-father Heapify-with-father							
Heapify-up	Other							
Max-heapify	Maximum-heap							
Wrong-duplicate	Wrong-duplicate							
Path-Bubblesort	Other							
Smallest-instantly-up	Smallest-instantly-up							

4.5 Software tools for misconception study

Several software applications were developed as tools for the empirical study. The source code for these tools are available at the author's GitHub account[17].

JSAV downloader retrieves JSAV exercise submissions from the A+ LMS. It is based on a similar Python script, which originally downloaded programming exercise submissions from the A+ LMS. Each *exercise instance* is specified manually. The exercise instance is a tuple (x, y), where x is the type of the exercise, such as Build-

[17] `https://github.com/atilante/JSAV-tools/releases/tag/masterthesis`

heap, y is the course instance, such as "2016" for the respective year. Submissions from each exercise instance are downloaded into their own JSON file.

JSAV inspector creates slideshows of exercise submissions. It runs in a web browser as single, static web page, meaning that no server setup is needed. The application is implemented in HTML5, CSS and JavaScript and it utilises the JSAV library. Another choice of technologies could have been Python with PyQt for GUI[18]. Choosing Python would mean that all software would be developed in Python, but also that GUI with PyQt and algorithm slideshows with a vector graphics library had to be developed. JSAV already support creating algorithm animations from exercise recording, and therefore JavaScript was chosen. The JSAV inspector tool can open a JSON file produced with the JSAV downloader and display students' solutions to the Build-heap exercise.

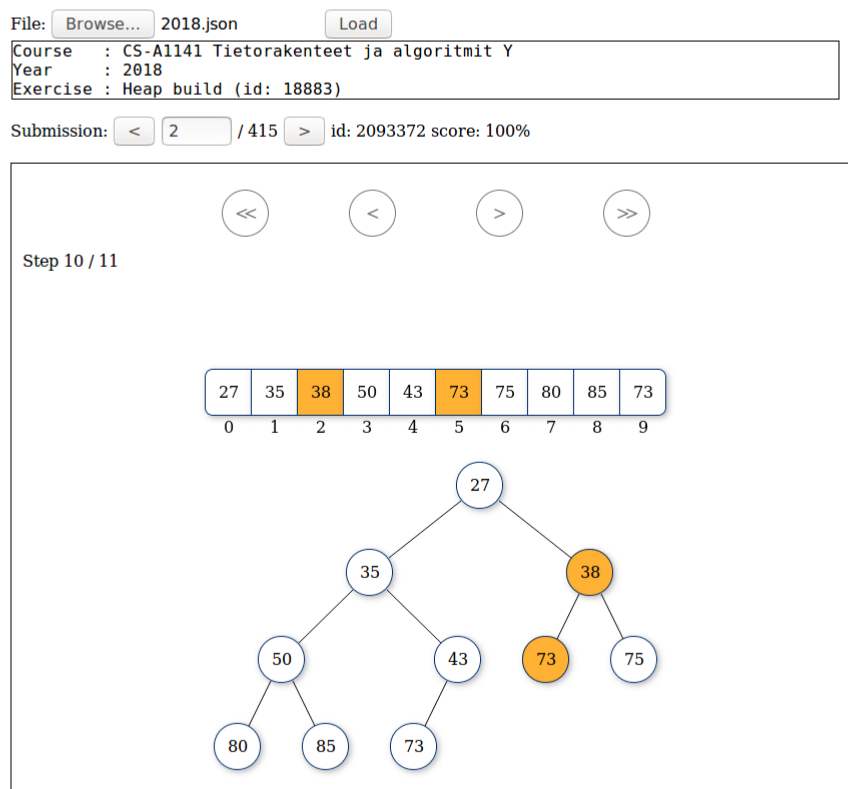


Figure 10: A screenshot of the JSAV inspector

Figure 10 shows the graphical user interface of the JSAV inspector. The rectangle at top displays information on the exercise recording file, here *2018.json*: name of the course, yearly instance, and exercise type. The submission selector below it allows the user to choose a submission for further inspection. The selector shows the submission identifier given by the A+ LMS and the relative, automatic score given by the JSAV library; both of these are included in the recording and not computed

[18]https://wiki.python.org/moin/PyQt

by the JSAV inspector.

The rectangle at bottom in Figure 10 displays a slideshow of the exercise recording, reconstructed by JSAV inspector and implemented with the JSAV library. The slideshow features both the array and tree views similar to the student's exercise interface. JSAV inspector shows state 1 as the initial state with random input, and for the Build-heap exercise, two steps for each swap: first a *highlight* step, where elements which are swapped next are colored orange, and then the *result* step, which shows the state of the heap after the swap.

The third tool is the *JSAV matcher* which compares the students' solution sequences against sequences produced by AMMs. The matcher software was implemented in Python. A+ LMS already supports Python-based *graders*: a server-side software that receives an exercise submission, assess it automatically and generates feedback for students[19]. JSAV matcher is discussed in detail in Section 5.3.

[19] `https://github.com/apluslms/grade-python`. Visited on 04/26/2020.

5 Empirical results

This section describes the results for the research question RQ7: *Which AMMs for the VAS exercises have matches in the submission data?* The structure of this section corresponds to the structure of the previous section.

5.1 Direct replication of the Build-heap studies

This section discusses the replicated Build-heap misconception study following methods described in Section 4.2. The essential results are shown in Figure 11. The corresponding numerical data can be found in Appendix B. The three main categories of submissions are: 743 *Correct* (52.0%), 311 *Misconceived* (21.7%, both finished and unfinished), and 376 *Unknown* (26.3%).

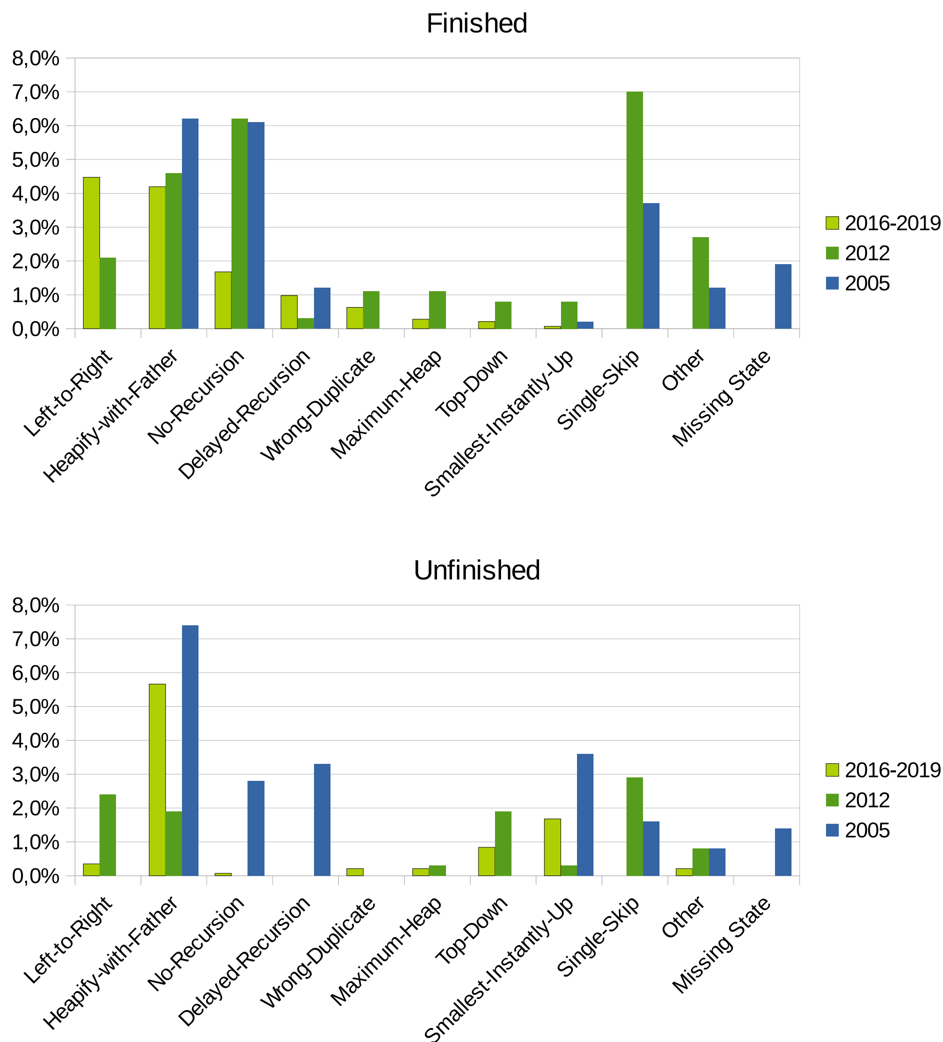


Figure 11: Replicated misconception study for Build-heap VAS exercise submissions.

The misconception categories have been changed between the studies [43] and

[19], using data from years 2005 and 2012, respectively. Left-to-Right is included in ”Other” category in 2005. AMMs Wrong-Duplicate, Top-Down and Maximum-Heap were discovered in 2012, and the Missing State category was omitted at the same time.

Generally, the same misconceptions still exist based on the statistics. The frequency of the *Correct* category according to JSAV-matcher is 52.0%, while the Build-Heap JSAV exercise give 100% automatic score to 53.7% of submissions. Of finished variants, *Heapify-with-Father* seems to be the most steady misconception; its frequency has lowest variance over the years. *No-Recursion* has lower frequency than before. No *Single-Skip* sequences could be detected, although they have been one of the most frequent variants. The yearly differences in the *Other* category suggest that there have been several undescribed variants in the earlier studies. The frequency of ”Unknown” category is similar to earlier studies.

Detailed analysis of the classification provided some explanations. Section 2.7 discussed how the JSAV library and the JSAV-matcher have different similarity algorithms. This causes minor differences in how the two algorithms recognise a ”correct” submission. All Finished *Correct* submissions received 100% automatic score from the Build-Heap JSAV exercise, as it requires exact state-by-state match. Meanwhile, JSAV gives 100% grade even if the student’s sequence as extra states after otherwise correct performance, and this explains the additional 1.7% of 100% exercise scores. JSAV-matcher labels 21 of 25 of these submissions as *Unknown*, two as *Heapify-with-Father* and two as *Left-to-Right*.

A *Single-Skip* variant actually matches perfectly to 50 (3.5%) submissions from years 2016–2019. However, 24 of them are labeled as Finished *No-Recursion* and 26 as Finished *Heapify-with-Father*. This means that for these submissions, the random input have caused identical sequence for student’s sequence, No-Recursion, Heapify-with-Father, and Single-Skip. Because Single-Skip has the lowest preference of them, it will never be chosen. This is an example of failing IR5 related to Section 2.4.

Generally, input requirements IR4 and IR5 fail often. A misconception could pass as the correct solution for 722 submissions (97.1% of correct ones). When the solution is not correct ($1430 - 743 = 687$ cases), there are 101 cases for multiple, equal misconceptions (7.06% of all submissions and 14.7% of incorrect ones).

Consecutive submission attempts eventually lead to success. First note that the exercise had 2.01 submission attempts per student on average, and that the average best score per student for 2016–2019 is 99.56 %. Figure 12 shows submission types by each student’s consecutive submission attempt. The *Main categories* plot groups submissions as follows. *Correct* denotes the exactly correct answer, *Misconceptions* denote both finished and unfinished misconceived algorithms, and *Unknown* is the same fallback category as before. Ten (10) submission attempts is the maximum to receive points from the exercise.

The frequencies of specific misconceptions follow the same decay trend, and due to this trend, only the overall most frequent misconceptions (Left-to-Right, Heapify-with-Father, No-Recursion, Delayed Recursion) exist at submission attempts 4–12.

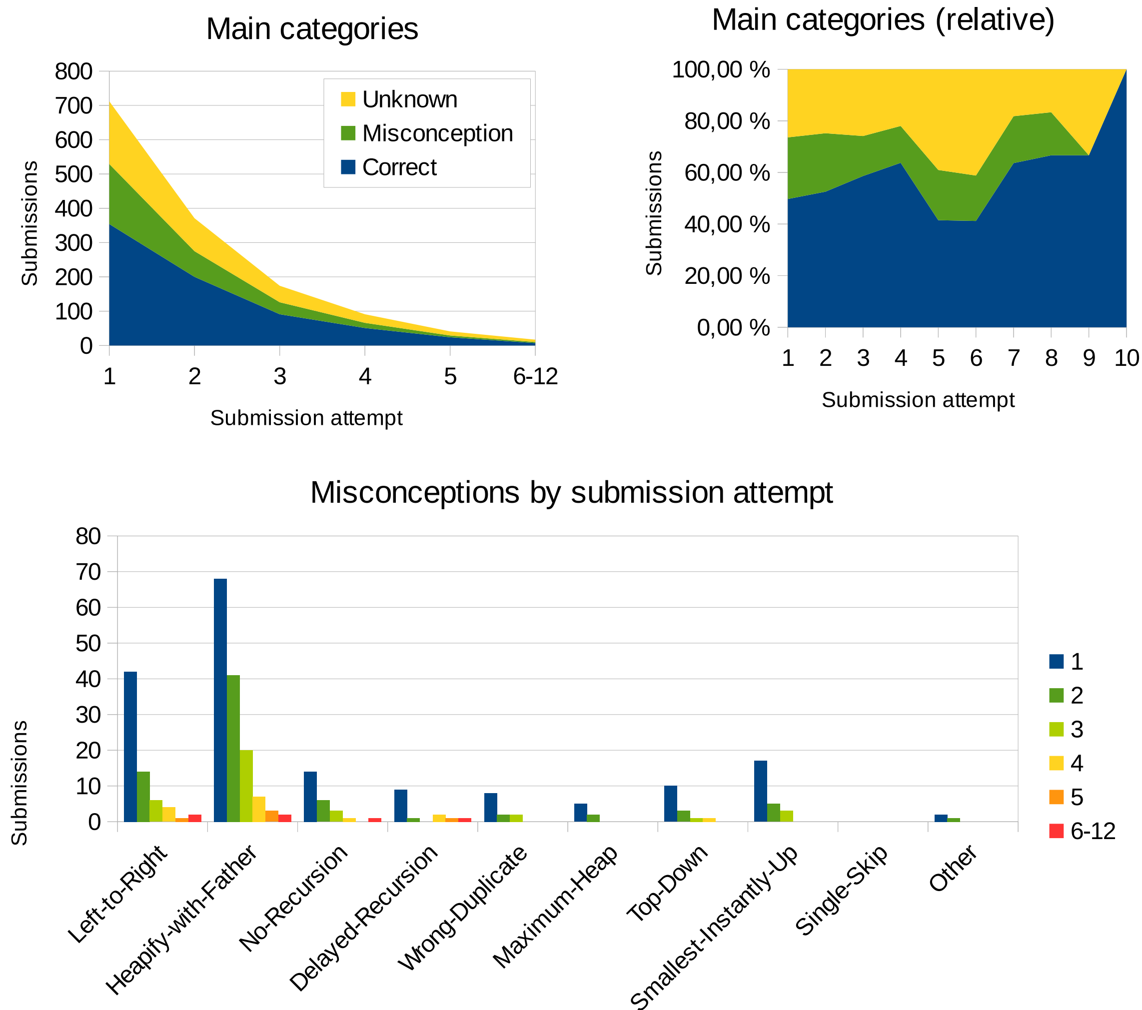


Figure 12: Build-heap submissions by student's attempt for 2016–2019 data.

5.2 Forming hypotheses for Build-heap misconceptions

Table 9 shows new categories of misconceptions. This set of AMMs is called the *main loop variant hypothesis*.

The proposed main loop variants are the following. The *Correct* main loop is the decreasing *for* loop in Algorithm 3. In practise, when the size of the heap array in the exercise is 10 and the corresponding indices are $0 \dots 9$, the array index sequence for the correct main loop is $(4, 3, 2, 1, 0)$. *Level LR* and *Top-down* both iterate each level from left to right, but in bottom-up and top-down order, respectively. The *Zigzag* variants are especially new, and they represent iteration where even levels are traversed from left to right and odd levels right to left, resulting in a "zigzag" pattern. The *Inorder* loop variant iterates the heap from left to right regardless of the height of each node, which is the same as the inorder traversal of a binary tree. See Appendix C for visual description of the variants.

Table 9: Manual classification of year 2016–2019 Build-heap submissions.

	Main loop variant							
Heapify variant	Correct	Zigzag up/RL	Zigzag up/LR	Level LR	Top-down	Zigzag down/LR	Zigzag down/RL	Inorder
Correct	927	68	14	26	4	–	–	14
No-recursion	51	12	4	2	2	–	–	3
Delayed recursion	24	3	1	3	–	–	–	1
Heapify-with-father LR	0	1	–	–	–	–	–	–
Heapify-with-father LR recursive	1	3	–	–	–	–	–	–
Heapify-with-father RL	5	1	–	–	–	–	–	–
Heapify-with-father RL recursive	22	1	–	1	–	–	–	–
Heapify-up	5	–	–	–	–	1	1	–
Max-heapify	4	–	–	1	–	–	–	–
Wrong-duplicate	11	1	–	–	1	–	–	1
Path-Bubblesort	1	–	–	–	–	–	–	–
Smallest-instantly-up	–	–	–	–	1	–	–	–

Miscellaneous categories	
Unrecognised	208

The *Path-Bubblesort* Heapify variant runs the *Bubblesort* sorting algorithm on the path of nodes between the root node and a leaf node. However, the first step is equal to the first step of Min-Heapify: only the lesser child is swapped with the parent node, if necessary.

Some submissions that are neither correct or an AMM seem to have phenomena which are called here as the *fallback features*. Each of the feature is binary: a submission either has the feature or it does not have it. These features not included in Table 9, but in the list below. Their existence is studied later in Sections 4.4 and 5.3.

No-Swaps A submission that does not have any swaps.

Extra-Steps-After-Correct The submission contains first the correct Build-min-heap sequence, but there are extra swaps in the end. Typically the last Heapify operation continues its swaps sequence until a leaf is encountered regardless of whether this is correct.

Swaps-Resemble-Build-Heap The submission has parent-child swaps that represent a correct execution of the Build-heap algorithm, but not with the given input. This kind of submissions are characterised by swaps where a lower priority element is swapped upwards, or several times the wrong child is swapped

with its parent. The student might have tried to imitate a model solution which has a different input.

Nonsystematic-Build-heap The submission produces a valid minimum heap with parent-child swaps, but the sequence itself is not systematic.

Legal-swap-indices All swaps of the sequence have pair of indices (i, j) such that j is child of i.

Legal-swaps All swaps of the sequence have pair of indices (i, j) such that j is child of i. When the heap array is A, also $A[j] < A[i]$, i.e. lower values are swapped upwards.

5.3 Machine classification of new Build-heap misconceptions

The three main categories of submissions with the main loop hypothesis are: 743 *Correct* (52.0%), 242 *Misconceived* (16.9%, both finished and unfinished), and 445 *Fallback*, (31.1%, including *Unknown*). The details are shown in Table 10. It seems that in general, both the main loop variants and the fallback features exist. The misconceived submissions have two subcategories: $851 - 743 = 108$ variants with correct main loop, and $242 - 108 = 134$ variants with altered main loop. The amount of algorithmically explained submissions (correct or misconception) is 845 (59.0%), and the fallback features explain almost all other cases $(445 - 2)$. Quantitatively, the classifier is able to give some explanation for 99.9% of submissions.

Examination of Table 10 shows that the *Delayed recursion* Heapify variant and the *Zigzag top-down LR* main loop variant seem absent. The main loop variants exist mostly for *Correct* and *No-recursion* Heapify variants (total 109 submissions), where as the main loop variants are rare (25 submissions) for other Heapify variants.

Detailed examination of the *Delayed recursion* variant shows that there were 93 cases where Delayed recursion was among best candidates, but the submission was classified as other misconception. In these cases the final classes were main loop variants of the correct algorithm: 48 *Zigzag RL*, 24 *Level LR*, 10 *Inorder*, 8 *Zigzag LR*, and 3 *Top-down*. The 14 submissions which the previous classifier labeled as Delayed recursion were classified mostly as Unrecognised.

Table 11 compares the classifiers: the current *Test* classifier and the *Default* classifier of the direct replication study. The number of submissions classified as misconceptions has decreased from 311 (16.9%) to 242 (21.7%). The Default classifier recognises additional 124 submissions as misconception where the Test classifier has given a fallback class. In the reverse situation, the Test classifier explains 55 submissions that were unknown to the Default classifier. Thus the both classifiers agreed that a submission is an AMM in 187 cases.

Figure 13 shows misconception category differences with the machine classifiers after the classification result of the Test classifier has been mapped to the classes that the Default classifier uses, following Table 8. AMMs *Heapify-with-Father*, *Left-to-Right*, and *No-Recursion* have highest frequentices with both classifiers still after the mapping. The increase in *No-Recursion* and *Other* categories support the explanation

Table 10: Machine classification of Build-heap submissions with the main loop variant hypothesis.

	Main loop variant								
Heapify variant	Correct	Zigzag up/RL	Zigzag up/LR	Level LR	Top-down	Zigzag down/LR	Zigzag down/RL	Inorder	sum
Correct	743	48	8	24	3	–	–	10	836
No-recursion	51	9	2	3	1	–	–	1	67
Delayed recursion	–	–	–	–	–	–	–	–	–
Heapify-with-father LR	2	2	–	–	–	–	–	3	7
Heapify-with-father LR recursive	5	1	–	–	–	–	–	1	7
Heapify-with-father RL	3	1	–	–	–	–	1	–	5
Heapify-with-father RL recursive	25	1	–	–	1	–	–	–	27
Heapify-up	4	1	–	2	2	–	2	1	12
Max-heapify	4	–	–	1	–	–	–	–	5
Wrong-duplicate	7	–	–	–	–	–	–	–	7
Path-Bubblesort	7	–	–	1	–	–	1	2	11
Smallest-instantly-up	–	–	–	–	1	–	–	–	1
sum	851	63	10	31	8	0	4	18	985

Fallback categories	
No-swaps	21
Extra-steps-after-correct	17
Swaps-Resemble-Build-heap	150
Nonsystematic-Build-heap	153
Legal-swaps	32
Legal-swap-indices	70
Unknown	2
sum	445

that the main loop variants of the *Correct* and *No-Recursion* Heapify variant have boosted these categories.

Table 12 compares the human and test classifiers. As stated in the end of Section 9, the manual classification did not include the fallback features. The classification correctness, where the human and the test classifiers gave the exactly same class, is 899/1430 submissions (62.9%). Of these, the submissions labeled as correct (743) received 100% agreement as expected. Interesingly, the rest of the 156 agreed submissions are the AMMs. This means that if the test classifier recognises the submission as an AMM, it agrees with the human on the exact variant at 156 of 189 cases (82.5%).

Regarding performance, the machine classification of 1430 submissions took 2.4

Table 11: Comparison of the default and the test classifiers.

Parameter	Default	Test	Common
Hypothetical misconceptions	10	95	9
Fallback features	0	6	0
Result			
Submission as correct	743	743	743
Submission as misconception	311	242	187
Submission as fallback feature	0	443	0
Submission as unknown	376	2	2
IR4 failures	722 (97%)	743 (100%)	–
IR5 failures	101 (32%)	230 (95%)	–
Submissions explained	1119 (78.3%)	1428 (99.9%)	–

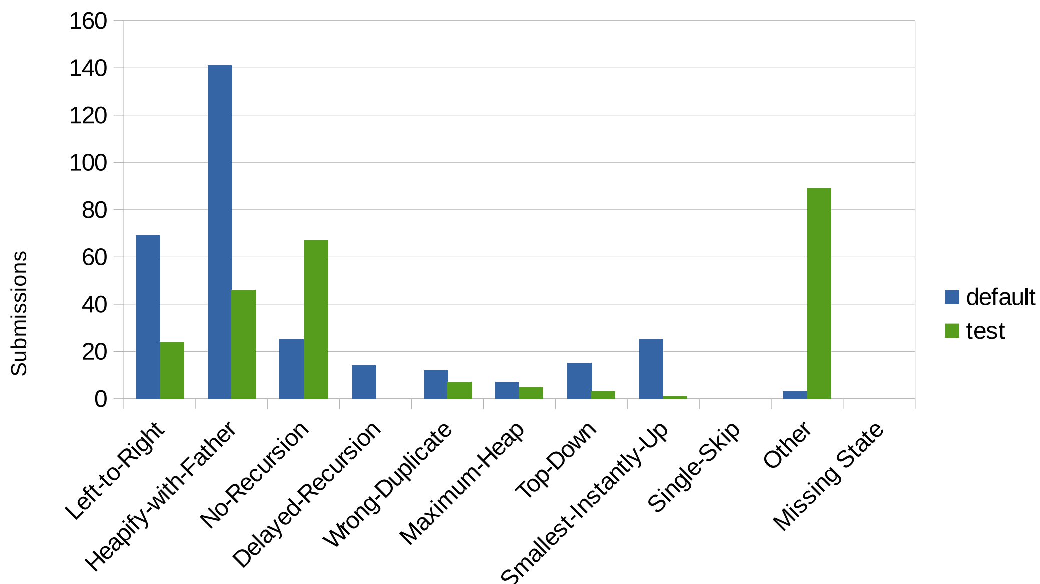


Figure 13: Differences in misconception recognition with the *Default* and *Test* classifiers.

Table 12: Comparison of the human and the test classifiers.

	Test classifier				
Human	Correct	Misconception	Fallback	Unknown	sum
Correct	743	29	155	0	927
Misconception	0	189	106	0	295
Unknown	0	24	182	2	208
sum	743	242	443	2	1430

seconds with single-threaded Python 3.5.2, Intel Core i5-7300U 2.60GHz CPU, and 16 GB of RAM. The classification time is thus 2 ms per submission, which looks promising for using JSAV-matcher as a grader software in production environment.

6 Discussion

This section concludes the thesis. Section 6.1 lists the key contributions of the thesis. Section 6.2 provides a reliability analysis of the methods and the results. Section 6.3 summarises the answers the research questions and provides explanations for the results. Section 6.4 provides recommendations for future research.

6.1 Contributions

The research objectives defined in Section 1.2 were met for the Build-heap VAS exercise. A web application was developed to replay students' solutions to VAS exercises. A Python application was developed to match AMMs to students' solutions. The empirical study confirmed results of the earlier studies [19, 43]. Moreover, the main loop variant hypothesis and fallback features seem to explain 99% of the Build-heap submissions.

The thesis also provided methodology on how to design VAS exercises that support detection of misconceptions. Six input requirements were compiled from earlier studies. Essential features of four algorithms were defined to improve corresponding VAS exercises: Evaluating postfix expression, Quicksort, Build-heap, and Dijkstra's algorithm. There is a proposal for a randomised algorithm that generates high-quality inputs for any VAS exercise.

Three JSAV-based exercises were improved: Dijkstra's algorithm, Prim's algorithm, and Kruskal's algorithm. Now these exercises force the student to apply all the essential features of the corresponding algorithm. The graph layout in these exercises is guaranteed to be easily visually readable according to aesthetic principles of graph drawing.

A risk analysis showed that the most significant problems in a misconception detecting VAS system are related to the input requirements. The risks can be mitigated by input requirements, input length, and classifier design.

6.2 Evaluation

One source of error in the direct replication study of Build-heap in Section 4.2 are the AMMs and the classifier which are written based on natural-language descriptions. Some interpretation was required, and it is not sure whether the resulting software is functionally exact to the one in the earlier studies.

Particularly the Algorithm 2 in Section 2.7 for sequence matching is a likely cause for the inexistence of the *Single-Skip* Build-heap AMM in Section 5.1. As discussed in that section, students' sequences which could be classified as *Single-Skip* are classified as *No-Recursion* or *Heapify-with-Father* instead. This happens because Algorithm 2 continues to match a candidate sequence to student's sequence even when one state is missing. Appendix D discusses an alternative interpretation of the original description: Algorithm 6.

Section 4.3 compares iterative and batch analysis processes for constructing hypotheses for AMMs. Although hypotheses for new Build-heap misconceptions were

found in Section 5.2, it is not enough to claim that the batch analysis process is always superior. Moreover, the known AMMs affect the hypothesis formulation for new Build-heap misconceptions. The hypotheses would be different if several human raters analysed the data independently and formed their own AMMs without prior knowledge of misconceptions in the earlier studies.

The manual assignment of class codes in Section 5.2 has more conservative interpretation of algorithm variants than the automatic classifier. One example is a recording where only one Heapify-with-Father double-swap happened. This is manually interpreted as the correct algorithm with a slip. However, in this case the manual classifier may give higher match with the Heapify-with-father AMM than the correct algorithm. This is one reason for classification errors. Another reason for classification errors is that IR4 and IR5 fail, as was already shown in the replicated study.

Moreover, manual classification in Section 5.2 cannot be reliably performed with single person, as the person might have an unconscious tendency to prefer one algorithm variant over another. In this thesis each submission was classified as the first applying variant that was recognised; it was not considered which of the many variants could apply. Due to failing input requirements and nearly 100 hypothetical algorithm variants, refining the 2016–2019 Build-heap submissions into a reliable test dataset using multiple human raters is a very laborious task.

One unknown factor in the Build-heap misconception hypothesis forming is the graphical user interface of the exercise. Even when there is evidence for certain solution patterns, some of them might be caused by difficulties using the VAS exercise and not that the student had a faulty mental model. The *Level LR* main loop, which is a mirror case of the correct main loop, could be a user difficulty with the GUI.

6.3 Discussion

Regarding RQ1, the only research on VAS exercise misconception are the articles [19, 25, 42, 43].

Regarding RQ2, eleven (11) articles consider misconceptions in data structures and algorithms. When VAS is excluded, these articles are [7, 9, 11, 12, 34, 35, 48, 53].

Regarding RQ3, VAS exercises can support detection of misconceptions by six input requirements defined in Section 2.4. According to risk analysis in Section 3.1, the length of input and classifier design also affect detection of misconceptions. Classifier design was discussed in Section 4.2. Section 3.2 studied one input requirement for four VAS exercises. Section 3.5 proposed a random search algorithm for input generation. Section 2.5 discussed another input generation method based on symbolic execution.

RQ4 was answered in Section 3.1. The analysis showed that the student–VAS software interaction risks can be mitigated. When an instructor forms a hypothesis for a new AMM, the new AMM should explain submissions that are previously unexplained. Moreover, there is need to validate AMMs by discussing with students.

Regarding RQ5, it was shown that input generation of the current VAS exercises is not misconception aware compared to input requirements. This area needs large

amount of further research.

Regarding RQ6, constructing a player application for a JSAV-based exercise currently requires separate work for each type of exercise. Moreover, not all exercises produce recordings that support playback and analysis, and therefore not all exercise recordings from years 2016–2019 are useful. Studying misconceptions in a JSAV exercise requires both modifying the exercise recording code and writing a visual player code at the worst case. Therefore, there is a need to develop the JSAV library, and the related OpenDSA libraries, such that all JSAV exercises would automatically record sufficient data for replay. This work includes deciding a general-purpose record format, an *algorithm animation language*, such as XAAL [17]. The language must support both manual and automatic analysis. The corresponding general-purpose analysis software must also be developed. Giacomo Mariani's Master's Thesis will contribute to this task [28].

Only AMMs of the Build-heap exercises were studied regarding RQ7. Already known AMMs seem to still exist based on submission data from year 2016–2019 course instances. New AMMs were proposed in the Build-heap main loop variant hypothesis. However, this extended set of AMMs explain the data to less extent than the original set.

A higher number of hypothetical AMMs seems to decrease misconception detectability. The direct replication study had 10 AMMs which matched to 311 submissions. The conceptual replication study had 95 AMMs which matched to 242 submissions. This is 22% less than with the original hypothesis. The alternative hypothesis includes all the algorithm variants in the original hypothesis except *Single-skips*, as shown in Appendix C. Therefore it is more likely that the cause of the phenomenon is the number of hypothetical algorithm variants rather than the alternative hypothesis itself. Moreover, the failure rates for IR4 and IR5 are also higher when there are more algorithm variants. This is understandable, as then it is less probable that some AMM explains the student's sequence two steps further than the correct algorithm.

The existence of fallback features also seems strong, as only the *No-swaps* and *Unrecognised* categories, total 1.6% of all submissions, do not have any logic that would resemble the Build-heap algorithm. No-swaps has two explanations: the student have clicked the *Grade* button when they have meant the *Model Answer* button on its left side, or the student has tried their chance to get some points without any effort. The *Extra-steps-after-correct* feature is either a slip or a sign of misconception that each call of Heapify always continues to the leaf node.

The seemingly 150 cases of *Swaps-Resemble-Build-Heap* can be explained with the imitative problem solving strategy discussed in Section 2.1. The student tries to perform the exactly same swaps that are in an example somewhere in the learning material (or the Internet), or then they have opened a model answer of the VAS exercise in a separate web browser and tried to blindly copy it. The equally popular 153 cases of *Nonsystematic-Build-heap* indicate that the student has understood the heap property but not the algorithm. The *Legal-swaps* and *Legal-swap-indices* indicate that the student has understood, to some extent, how a single swap is performed.

The question about which hypothesis, the original or the one with main loop variants, is the truth about Build-heap AMMs still remains unanswered. Several factors cause unreliability: there is only one human rater, the input generation is not misconception aware, and the misconceptions have not been validated with the students.

The replicated Build-heap studies imply that misconceptions still exist for the exercise. The heap size of 10 elements is clearly too small, as misconceptions can pass as correct (IR4), many misconceptions explain one sequence equally well (IR5), and it is possible to form different hypotheses based on the data.

6.4 Recommendations

6.4.1 Improving the Build-heap VAS exercise

This subsection gives the most important recommendations that are required to refine the current Build-heap misconception classifier into a working product.

1. A set of feedbacks for Build-heap AMMs should be written and assigned to the algorithm variants. It is likely that not every class in the main loop hypothesis requires a different feedback text. If the submission is a main loop variant misconception, one could first instruct the student on following the correct main loop, maybe with the No-Recursion variant. Then, after the student performs a sequence matching an AMM with the Correct main loop variant, another feedback would guide them towards the correct, recursive Min-heapify. Thus the number of feedback categories might be limited to 10 or 20, which might improve classification correctness.

2. The heap size should be extended to 20 to improve detectability of misconceptions. This should still fulfill IR6.

3. Algorithm 4 should be used to find random inputs which fulfill IR3–IR5. IR3 is currently implemented in the Build-heap JSAV exercise code and it can be imported into JSAV-matcher with some work. First it should be studied how hard it is to fulfill IR4 with heap size of 20. If this succeeds, IR5 should be added to the requirements. If finding good inputs is computationally intensive, these inputs could be pregenerated.

4. The JSAV Matcher should be integrated into the A+ LMS. This requires:

    (a) Packaging the JSAV Matcher into a Python-based grader container;

    (b) Modifying the current web server code which receives a JSAV exercise submission to run the grader container and then send the results back to the JSAV exercise;

    (c) Modifying the Build-heap JSAV exercise show the feedback.

5. There should be a mechanism to validate the feedback with the students. For example, the Build-heap exercise could ask whether the given feedback was useful, and this bit of information would be recorded into the A+ LMS along with the exercise input, student's solution, grade, and feedback. More elaborate choices for student's feedback for automatic feedback could be a *How do you feel?* question with answers such as ”confused”, ”new information”, ”already knew this”. If the submission is classified as unknown, the exercise could request the student to write about what they were thinking.

Alternatively, steps 2 and 3 can be postponed if we want to have the heap size fixed and only study how students' behaviour changes when the exercise gives written feedback. Overall, not all improvements cannot be done at the same time if we want to study the effectiveness of each improvement.

6.4.2 Further research questions

This subsection discusses less urgent research questions which are related to the topic of this thesis.

It should be studied how the modified graph algorithm exercises in Section 3.4 affect students' workload and learning. Did the submission statistics change? Are there any new issues in feedback from students?

The problem of misconception aware input against the input requirements in Section 2.4 should be studied further. It should be verified that increasing number of algorithm variants decreases the amount of submissions classified as misconceptions and increases input requirement failures. This could be verified with the existing Build-heap submission dataset by choosing $1, 2, \ldots, 95$ algorithm variants of the main loop hypothesis and classifying the dataset each time.

Misconception-aware input generation by symbolic execution Section 2.5.2 in could be studied if random search in Section 3.5 seems computationally expensive.

The earlier VAS misconception research articles raised several questions and proposals which still require further study [19, p. 68, 43, p. 254]. Thus far the automatic part of misconception recognition has been done for *classification* of given submissions as predefined algorithm variants. This has required large amount of manual work: students' solutions reviewed in detail to be able to form hypotheses for algorithm variants. Then the hypothetical algorithm variants must be written in a programming language to, which is also laborious.

Data mining could provide automatic support for generation of hypotheses. Precisely, it might be possible to create a *hierarchical clustering* of the submissions for one exercise using unsupervised machine learning methods [1, pp. 143–161]. Although the algorithm simulation sequences are strings of states by definition, the actions leading from a state to another resemble steps in a time series: swaps from one array index to another array index, or direction of lower value in the swap. In this case the Dynamic Time Warping [39] or other *elastic similarity measures* [33] could be used with together Kruskal's algorithm to obtain a similarity hierarchy of submissions. This requires defining a distance function for two data structure operations: how similar are two swaps in a binary heap, or two a visits node of a tree? The resulting

similarity hierarchy is a tree whose root is all the submissions and each of the branch is a subset of submissions which have some degree of computed similarity. The further a branch extends from the root, the less it contains submissions, but the more similar are the submissions. Finally, a human could explore this hierarchy of submissions and decide where to cut each branches so that it represents a single misconception. If one or more human raters agree that the similarity tree represents category of misconceptions, there is no need to write program code for AMMs: it is possible to classify new submissions as misconceptions based on how they relate to the tree.

An extended literature review of misconceptions related to data structures and algorithms could support teaching. The already mentioned misconceptions in Section 2.2 could be extended into a catalogue Similar to Sorva's doctoral thesis [46, p. 358–368]. The search scope should be extended to related keywords such as *partial understanding*, *incorrect understanding*, *student difficulties*, *CS2*, *student-constructed rules*, *mistakes*, and *bugs*. Also the The Cambridge Handbook of Computing Education Research might provide essential background [26].

A VAS exercises on the Data Structures and Algorithms course

This appendix lists the JSAV-based VAS exercises that were used on Aalto University course *CS-A1141 Data structures and algorithms Y* on the year 2018. The list of exercises is shown in Table 13. All of these exercises are a subset of JSAV-based exercises in the OpenDSA [20, 31]

Table 13: List of visual algorithm simulation exercises on the DSA course. The numbers refer to the weekly chapters in the electronic learning material.

2. Linear structures	**8. Search structures**	**10. Hashing**
Evaluating postfix expression	Binary search	Open hashing
Infix to postfix	Interpolation search	Linear probing
5. Sorting	Binary search tree search	Quadratic probing
Insertion Sort	Binary search tree insert	Double hashing
Selection Sort	Binary search tree remove	Rehashing
Mergesort	**9. Balanced search trees**	**11. Graphs**
Quicksort	Rotation	Depth first search
Radix exchange sort	Double rotation	Breadh first search
6. Tree traversals	AVL tree insertion	Prim's algorithm
Pre-order traversal	Red-black tree coloring	Dijkstra's algorithm
In-order traversal	Red-black tree insertion	Kruskal's algorithm
Post-order traversal	Digital search tree	**12. Bonus**
Level-order traversal	Radix trie	B+ tree insertion
7. Priority queues	Common trie	Modified B+ tree insertion
Heap insert		
Heap remove		
Heap build		
Heapsort		

B Direct replications of Build-heap study

Table 14: Replicated misconception study for Build-heap VAS exercise submissions. F denotes a finished, U an unfinished category.

	2016–2019				2012		2005	
Variant	F	U	F	U	F	U	F	U
Correct	743	–	52.0%	–	29.2%	–	34.3%	–
Wrong-Duplicate	9	3	0.6%	0.2%	1.1%	–	–	–
Heapify-with-Father	60	81	4.2%	5.7%	4.6%	1.9%	6.2%	7.4%
Left-to-Right	64	5	4.5%	0.3%	2.1%	2.4%	–	–
No-Recursion	24	1	1.7%	0.1%	6.2%	–	6.1%	2.8%
Single-Skip	0	0	0.0%	0.0%	7.0%	2.9%	3.7%	1.6%
Top-Down	3	12	0.2%	0.8%	0.8%	1.9%	–	–
Delayed-Recursion	14	0	1.0%	0.0%	0.3%	–	1.2%	3.3%
Smallest-Instantly-Up	1	24	0.1%	1.7%	0.8%	0.3%	0.2%	3.6%
Maximum-Heap	4	3	0.3%	0.2%	1.1%	0.3%	–	–
Missing State	–	–	–	–	–	–	1.9%	1.4%
Other	0	3	0.0%	0.2%	2.7%	0.8%	1.2%	0.8%
Unknown	–	376	–	26.3%	–	33.8%	–	24.2%
Total	1430		100%		100%		100%	

Table 14 shows the numerical results of the replicated Build-heap study. Data from years 2016–2019 is analysed in this thesis, data from year 2012 is from [19], and data from year 2005 is from [43].

C Details for alternative Build-heap hypothesis

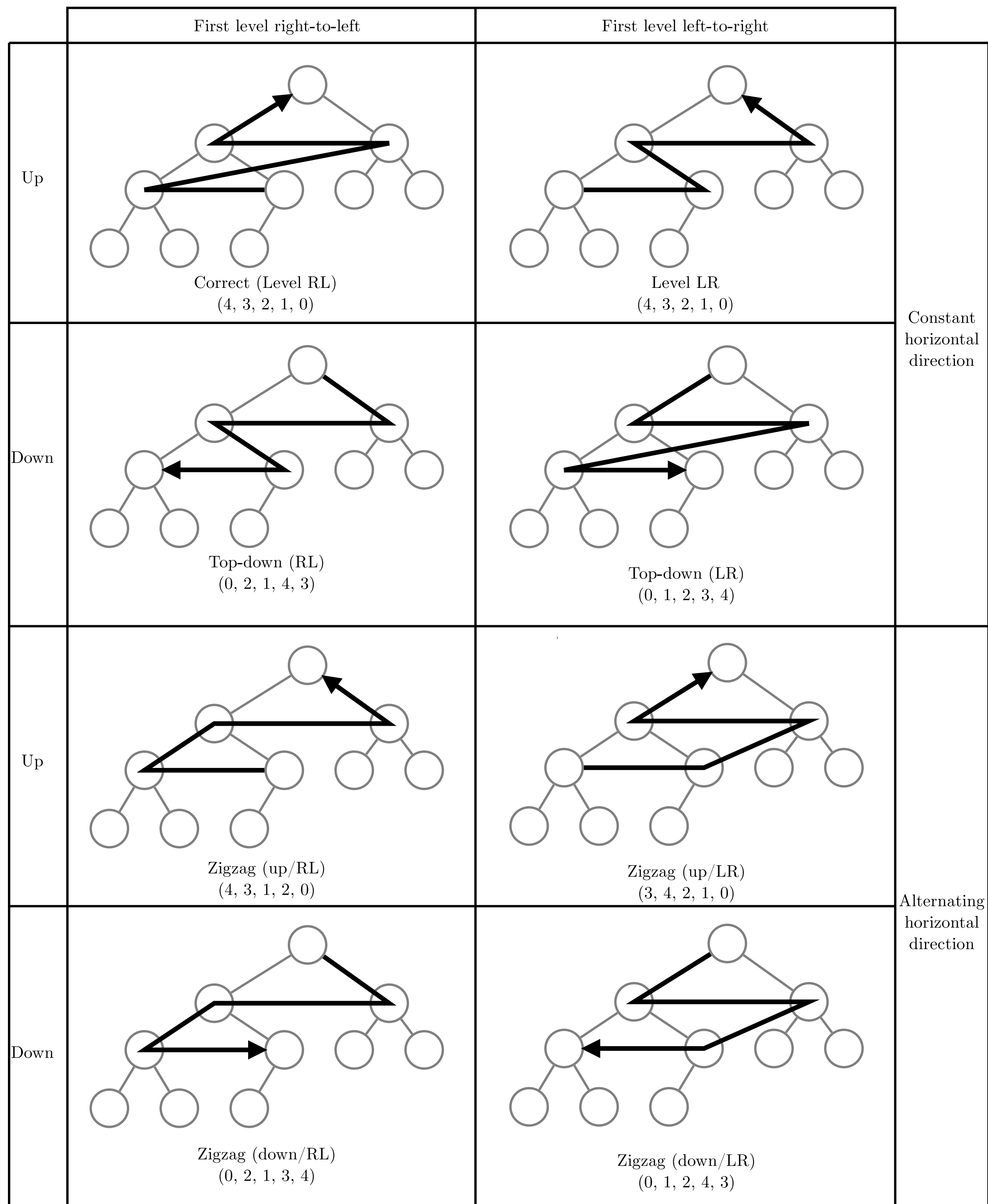


Figure 14: Main loop variants of the Build-heap AMMs.

Figure 14 shows visualisation of the Build-heap main loop variants. There are eight variants, and in each of them the order in which the main loop traverses the binary tree representation is shown as arrow. The figures have heap size of 10, but the patterns are not limited by the size of the heap. Each of then can be generated

with an algorithm.

Table 15: Order of preference for algorithm variants in the Build-heap main loop hypothesis.

	Main loop variant							
Heapify variant	Correct	Zigzag up/RL	Zigzag up/LR	Level LR	Top-down	Zigzag down/LR	Zigzag down/RL	Inorder
Correct	0	1	6	3	14	23	22	7
No-recursion	2	8	13	20	32	59	58	18
Delayed recursion	4	15	30	17	41	67	66	29
Heapify-with-father LR	26	55	76	65	83	95	92	73
Heapify-with-father LR recursive	19	37	57	47	68	89	88	56
Heapify-with-father RL	10	31	49	42	61	85	84	48
Heapify-with-father RL recursive	5	16	35	27	43	70	69	34
Heapify-up	11	28	46	40	60	80	79	45
Max-heapify	12	36	52	44	62	87	86	51
Wrong-duplicate	9	21	39	33	50	78	77	38
Path-Bubblesort	25	54	75	64	82	94	91	72
Smallest-instantly-up	24	53	74	63	81	93	90	71

Table 15 shows the order of preference in the Build-heap *Test* classifier when there are multiple algorithm variants which explain a submission equally well. Lower value means higher priority. Thus the three most preferred heapify–main loop variant pairs are *Correct–Correct*, *Correct–Zigzag up/RL*, and *No-recursion–Correct*.

Table 16 compares the mapping of algorithm variants between [19], the direct replication in thesis, and the conceptual replication in this thesis. Column *Alt.hypot.* indicates whether the algorithm variant has been included in the conceptual replication (the main loop hypothesis).

Table 16: Build-heap variant mapping between Karavirta et al. [19] and this thesis.

Algorithm in [19]	Direct replication		In alternative hypothesis
	Main loop	Heapify	
Correct	4,3,2,1,0	Correct	yes
Wrong-Duplicate	4,3,2,1,0	Wrong-Duplicate	yes
Heapify-with-Father	4,3,2,1,0	Heapify-with-Father LR recursive	yes
Heapify-with-Father	4,3,2,1,0	Heapify-with-Father LR recursive	yes
Left-to-Right	3,4,1,2,0	Correct	yes
No-Recursion	4,3,2,1,0	No-Recursion	yes
Single-Skip	4,3,2,1,0	Correct	–
Top-Down	0,1,2,3,4	Correct	yes
Delayed-Recursion	4,3,2,1,0	Delayed-Recursion	yes
Smallest-Instantly-Up	0,1,2,3,4	Smallest-Instantly-Up	yes
Maximum-Heap	4,3,2,1,0	Max-Heapify	yes
Other	0,2,1,4,3	Smallest-Instantly-Up	yes

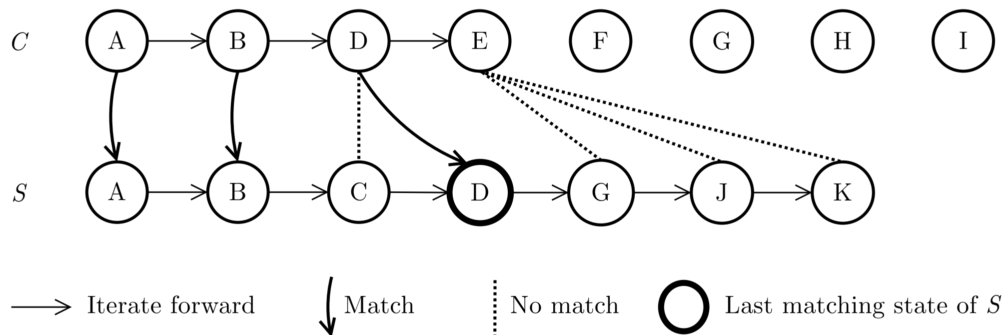
D Revised sequence similarity algorithm

Algorithm 6 is another interpretation of the grading algorithm in [43, p. 246]. It is similar to Algorithm 6 in Section 2.7, but does not allow skips in the candidate sequence. Input C is the candidate sequence produced by an AMM. Input S is the student's sequence.

Algorithm 6 Sequence similarity without skips in the candidate sequence.

```
1:  procedure State-Similarity(C, S)
2:      i, j ← 0
3:      while i < |C| and j < |S| do
4:          if C[i] = S[j] then
5:              i ← i + 1
6:          end if
7:          j ← j + 1
8:      end while
9:      return j
10: end procedure
```

Figure 15 shows an execution example corresponding to Figure 5 in Section 2.7. It is similar to Figure 2 in [25, p. 3].



Pairs of states (c, s) tested for equality ($c \in C,\ s \in S$):
(A,A), (B,B), (D,C), (D,D), (E,G), (E,J), (E,K).
Return value: 3.

Figure 15: An execution example of Algorithm 6.

References

[1] Ethem Alpaydin. *Introduction to machine learning*. 2nd ed. Cambridge, Massachusetts, USA: The MIT Press, 2010. ISBN: 978-0-262-01243-0.

[2] Ari-Matti Auvinen. "Mistä oppimisanalytiikassa keskustellaan?" In: *SeOPPI* 1.1 (2017), pp. 6–7. ISSN: 1795-3251.

[3] Bilal M. Ayuub. *Risk analysis in engineering and economics*. Boca Raton, Florida, USA: Chapman & Hall/CRC, 2003. ISBN: 1-58488-395-2.

[4] Ulrik Brandes. "Drawing on physical analogies". In: *Drawing Graphs : Methods and models*. Ed. by Michael Kaufmann and Dorothea Wagner. Lecture Notes in Computer Science, vol. 2025. Berlin, Germany: Springer-Verlag, 2001, pp. 71–86. ISBN: 978-3-540-44969-0. DOI: `10.1007/3-540-44969-8`.

[5] John Seely Brown and Kurt VanLehn. "Repair Theory: A Generative Theory of Bugs in Procedural Skills". In: *Cognitive Science* 4.4 (1980), pp. 379–426. ISSN: 0364-0213. DOI: `https://doi.org/10.1207/s15516709cog0404_3`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog0404%5C_3`.

[6] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. ed. Cambridge, Massachusetts, USA: The MIT Press, 2009. ISBN: 978-0-262-53305-8.

[7] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. "Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 21–26. ISBN: 978-1-4503-1098-7. DOI: `10.1145/2157136.2157148`. URL: `http://doi.acm.org/10.1145/2157136.2157148`.

[8] Shahira El Alfy, Jorge Marx Gómez, and Anita Dani. "Exploring the benefits and challenges of learning analytics in higher education institutions: a systematic literature review". In: *Information Discovery and Delivery* 47.1 (2019), pp. 25–34. ISSN: 2398-6247. DOI: `10.1108/IDD-06-2018-0018`.

[9] Mohammed F. Farghally et al. "Towards a Concept Inventory for Algorithm Analysis Topics". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: ACM, 2017, pp. 207–212. ISBN: 978-1-4503-4698-6. DOI: `10.1145/3017680.3017756`. URL: `http://doi.acm.org/10.1145/3017680.3017756`.

[10] Rudolf Fleischer and Colin Hirsch. "Graph drawing and its applications". In: *Drawing Graphs : Methods and models*. Ed. by Michael Kaufmann and Dorothea Wagner. Lecture Notes in Computer Science, vol. 2025. Berlin, Germany: Springer-Verlag, 2001, pp. 1–22. ISBN: 978-3-540-44969-0. DOI: `10.1007/3-540-44969-8`.

[11] Judith Gal-Ezer and Ela Zur. "The efficiency of algorithms—misconceptions". In: *Computers & Education* 42.3 (2004), pp. 215–226. ISSN: 0360-1315. DOI: `10.1016/j.compedu.2003.07.004`.

[12] Tina Götschi, Ian Sanders, and Vashti Galpin. “Mental Models of Recursion”. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’03. Reno, Navada, USA: ACM, 2003, pp. 346–350. ISBN: 1-58113-648-X. DOI: 10.1145/611892.612004.

[13] Qiang Hao et al. “A Systematic Investigation of Replications in Computing Education Research”. In: *ACM Transactions on Computing Education* 19.4 (Aug. 2019). DOI: 10.1145/3345328. URL: https://doi.org/10.1145/3345328.

[14] Carsten Held, Markus Knauff, and Gottfried Vosgerau. “General Introduction”. In: *Mental models and the mind: current developments in cognitive psychology, neuroscience, and philosophy of mind*. Ed. by Carsten Held, Markus Knauff, and Gottfried Vosgerau. Advances in Psychology, 138. Elsevier Science & Technology, 2006, pp. 5–22. ISBN: 978-0-444-52079-1.

[15] Kaisa Honkonen and Leena Vainio. “Times of day, submission dates, learning statistics, interaction charts – it this what is meant by ”learning analytics”?” In: *SeOPPI* 1.2 (2008). ISSN: 1795-3251.

[16] Petri Ihantola. *Automatic test data generation for programming exercises with symbolic execution and Java PathFinder*. Master’s Thesis. Espoo, Finland, 2006.

[17] Ville Karavirta. *XAAL - Extensible Algorithm Animation Language*. Master’s Thesis. Espoo, Finland, 2005. URL: http://www.cs.hut.fi/Research/SVG/publications/karavirta-masters.pdf (visited on 02/21/2020).

[18] Ville Karavirta, Petri Ihantola, and Teemu Koskinen. “Service-Oriented Approach to Improve Interoperability of E-Learning Systems”. In: *2013 IEEE 13th International Conference on Advanced Learning Technologies*. IEEE, July 2013, pp. 341–345. DOI: 10.1109/ICALT.2013.105.

[19] Ville Karavirta, Ari Korhonen, and Otto Seppälä. “Misconceptions in Visual Algorithm Simulation Revisited: On UI’s Effect on Student Performance, Attitudes, and Misconceptions”. In: *2013 Learning and Teaching in Computing and Engineering*. IEEE, Mar. 2013, pp. 62–69. DOI: 10.1109/LaTiCE.2013.35.

[20] Ville Karavirta and Clifford A. Shaffer. “Creating Engaging Online Learning Material with the JSAV JavaScript Algorithm Visualization Library”. In: *IEEE Transactions on Learning Technologies* 9.2 (Apr. 2016), pp. 171–183. ISSN: 1939-1382. DOI: 10.1109/TLT.2015.2490673.

[21] Ville Karavirta and Clifford A. Shaffer. “JSAV: The JavaScript Algorithm Visualization Library”. In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’13. Canterbury, England, UK: ACM, 2013, pp. 159–164. ISBN: 978-1-4503-2078-8. DOI: 10.1145/2462476.2462487. URL: http://doi.acm.org/10.1145/2462476.2462487.

[22] Kuba Karpierz and Steven A. Wolfman. "Misconceptions and Concept Inventory Questions for Binary Search Trees and Hash Tables". In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. Atlanta, Georgia, USA: ACM, 2014, pp. 109–114. ISBN: 978-1-4503-2605-6. DOI: 10.1145/2538862.2538902. URL: http://doi.acm.org/10.1145/2538862.2538902.

[23] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. "Generalized Symbolic Execution for Model Checking and Testing". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Hubert Garavel and John Hatcliff. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, 2003, pp. 553–568. ISBN: 978-3-540-36577-8.

[24] Ari Korhonen. "Visual Algorithm Simulation". PhD thesis. Espoo, Finland: Helsinki University of Technology, Department of Computer Science and Engineering, 2003. ISBN: 951-22-6788-8.

[25] Ari Korhonen, Otto Seppälä, and Juha Sorva. "Automatic recognition of misconceptions in visual algorithm simulation exercises". In: *2015 IEEE Frontiers in Education Conference (FIE)*. IEEE, Aug. 2015. DOI: 10.1109/FIE.2015.7344046.

[26] Colleen M. Lewis, Michael J. Clancy, and Jan Vahrenhold. "Student Knowledge and Misconceptions". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V. Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 773–800. DOI: 10.1017/9781108654555.028.

[27] Lauri Malmi et al. "Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2". In: *Informatics in Education* 3.2 (2004), pp. 267–288. ISSN: 2335-8971. URL: https://www.mii.lt/informatics_in_education/pdf/INFE048.pdf.

[28] Giacomo Mariani. *Design of an application to collect data and create animations from Visual Algorithm Simulation exercises*. Master's Thesis, planned publication on 2020. The Title is from the thesis presentation. Espoo, Finland, 2020.

[29] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge, UK: Cambridge University Press, 1995. ISBN: 0-521-47465-5.

[30] n.d. *Learning Analytics & Knowledge Conference 2019 | General Call*. 2019. URL: https://lak19.solaresearch.org/general-call/ (visited on 07/29/2019).

[31] n.d. *OpenDSA/AV/Development at master · OpenDSA/OpenDSA · GitHub*. 2019. URL: https://github.com/OpenDSA/OpenDSA/tree/master/ (visited on 09/16/2019).

[32] n.d. *VISSOFT2019 - seventh IEEE Working Conference on Software Visualization*. 2019. URL: http://vissoft19.dcc.uchile.cl/ (visited on 07/31/2019).

[33] Izaskun Oregi et al. “On-line Elastic Similarity Measures for time series”. In: *Pattern Recognition* 88 (2019), pp. 506–517. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2018.12.007. URL: http://www.sciencedirect.com/science/article/pii/S003132031830428X.

[34] Nesrin Özdener. “A comparison of the misconceptions about the time-efficiency of algorithms by various profiles of computer-programming students”. In: *Computers & Education* 51.3 (2008), pp. 1094–1102. ISSN: 0360-1315. DOI: 10.1016/j.compedu.2007.10.008.

[35] Wolfgang Paul and Jan Vahrenhold. “Hunting High and Low: Instruments to Detect Misconceptions Related to Algorithms and Data Structures”. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE ’13. Denver, Colorado, USA: ACM, 2013, pp. 29–34. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445212. URL: http://doi.acm.org/10.1145/2445196.2445212.

[36] Leo Porter et al. “BDSI: A Validated Concept Inventory for Basic Data Structures”. In: *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ICER ’19. Toronto ON, Canada: ACM, 2019, pp. 111–119. ISBN: 978-1-4503-6185-9. DOI: 10.1145/3291279.3339404. URL: http://doi.acm.org/10.1145/3291279.3339404.

[37] Leo Porter et al. “Developing Course-Level Learning Goals for Basic Data Structures in CS2”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18. Baltimore, Maryland, USA: ACM, 2018, pp. 858–863. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159457. URL: http://doi.acm.org/10.1145/3159450.3159457.

[38] S. Ian Robertson. *Is analogical problem solving always analogical?: the case for imitation.* Tech. rep. HCRL Technical Report 97. HCRL, The Open University, 1993. URL: https://www.researchgate.net/publication/2248507 (visited on 07/31/2019).

[39] Hiroaki Sakoe and Seibi Chiba. “Dynamic programming algorithm optimization for spoken word recognition”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (Feb. 1978), pp. 43–49. ISSN: 0096-3518. DOI: 10.1109/TASSP.1978.1163055.

[40] Stefan Schmidt. “Shall we Really do it Again? The Powerful Concept of Replication is Neglected in the Social Sciences”. In: *Review of General Psychology* 13.2 (2009), pp. 90–100. DOI: 10.1037/a0015108. URL: https://doi.org/10.1037/a0015108.

[41] Norbert M. Seel. “Mental Models in Learning Situations”. In: *Mental models and the mind : current developments in cognitive psychology, neuroscience, and philosophy of mind.* Ed. by Gottfried Vosgerau Carsten Held Markus Knauff. Advances in Psychology, 138. Elsevier Science & Technology, 2006, pp. 85–107. ISBN: 978-0-444-52079-1.

[42] Otto Seppälä. “Modelling Student Behavior in Algorithm Simulation Exercises with Code Mutation”. In: *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*. Baltic Sea ’06. Uppsala, Sweden: ACM, 2006, pp. 109–114. DOI: 10.1145/1315803.1315822. URL: http://doi.acm.org/10.1145/1315803.1315822.

[43] Otto Seppälä, Lauri Malmi, and Ari Korhonen. “Observations on Student Misconceptions—A Case Study of the Build – Heap Algorithm”. In: *Computer Science Education* 16.3 (2006), pp. 241–255. DOI: 10.1080/08993400600913523.

[44] Ian Sommerville. *Path Testing*. 2008. URL: https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Testing/PathTest.html (visited on 09/02/2019).

[45] Ian Sommerville. *Software Engineering*. 9th. Boston, Massachusetts, USA: Pearson Education, 2011. ISBN: 978-0-13-705346-9.

[46] Juha Sorva. “Visual Program Simulation in Introductory Programming Education”. PhD thesis. Espoo, Finland: Aalto University, School of Science, 2012, p. 428. ISBN: 978-952-60-4625-9. URL: https://aaltodoc.aalto.fi/handle/123456789/3534.

[47] Kozo Sugiyama. *Graph drawing and applications for software and knowledge engineers*. River Edge, New Jersey, USA: World scientific, 2002. ISBN: 978-981-4489-24-9. DOI: 10.1142/9789812777898.

[48] Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. “Automatic Recognition of Students’ Sorting Algorithm Implementations in a Data Structures and Algorithms Course”. In: *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. Koli Calling ’12. Koli, Finland: ACM, 2012, pp. 83–92. ISBN: 978-1-4503-1795-5. DOI: 10.1145/2401796.2401806. URL: http://doi.acm.org/10.1145/2401796.2401806.

[49] Artturi Tilanterä. *Improved Dijkstra’s, Kruskal’s and Prim’s algorithm practise exercises · atilante/OpenDSA@280cdbc*. Retrieved 27.1.2020. 2020. URL: https://github.com/atilante/OpenDSA/commit/280cdbc10327bea9d7df942700db4c7d87508daa.

[50] Artturi Tilanterä. *Verkkojen piirtäminen hierarkkisella ryhmittelyllä*. Bachelor’s Thesis. Aalto University. Espoo, Finland, 2014. URL: https://aaltodoc.aalto.fi/handle/123456789/13233.

[51] Matthew Wickline and Human-Computer Interaction Resource Network. *Coblis – Color Blindness Simulator*. Retrieved 29.1.2020. 2018. URL: https://www.color-blindness.com/coblis-color-blindness-simulator/.

[52] Hong Zhu, Patrick A. V. Hall, and John H. R. May. “Software Unit Test Coverage and Adequacy”. In: *ACM Computing Surveys* 29.4 (Dec. 1997), pp. 366–427. ISSN: 0360-0300. DOI: 10.1145/267580.267590.

[53] Daniel Zingaro et al. “Identifying Student Difficulties with Basic Data Structures”. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER ’18. Espoo, Finland: ACM, 2018, pp. 169–177. ISBN: 978-1-4503-5628-2. DOI: 10.1145/3230977.3231005. URL: http://doi.acm.org/10.1145/3230977.3231005.