HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences

Jarno Ruokokoski

Automatic Assessment in University-level Mathematics

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology in the Degree Programme in Engineering Physics.

Espoo, 27.8.2009

Supervisor: Professor Gustaf Gripenberg
Instructor: Ph.D. Antti Rasila

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta

Diplomityön tiivistelmä

Tekijä:	Jarno Ruokokoski
Koulutusohjelma:	Teknillinen fysiikka
Pääaine:	Matematiikka
Sivuaine:	Operaatio- ja systeemitutkimus
Työn nimi:	Automaattinen tarkastaminen yliopistotasoisessa matematiikassa
Title in English:	Automatic Assessment in University-level Mathematics
Professuurin koodi ja nimi:	Mat-1 Matematiikka
Työn valvoja:	Professori Gustaf Gripenberg
Työn ohjaaja:	FT Antti Rasila

Tiivistelmä:

Tässä diplomityössä tutkitaan satunnaistettujen, yliopistotasoisten tekniikan alan matematiikan tehtävien automaattista tarkastamista. Työssä käytetään yhtä tällaiseen käyttöön suunniteltua ohjelmaa ja toteutetaann sillä tehtäväsarjoja eri matematiikan aihealueilta.

Aihealueiksi valitaan laskento ja graafiteoria. Lisäksi tutkitaan hieman matemaattista todistamista. Aiheet sopivat tekniikan alan matematiikan opetukseen automaatio- ja systeemitekniikan sekä tietotekniikan opiskelijoille. Työssä oletetaan laskentoon liittyvät aiheet tunnetuiksi, mutta graafiteoria esitetään sillä tarkkuudella, että sen ymmärtäminen ilman muuta tuntemista on mahdollista.

Työtä varten kirjoitettiin noin 80 tehtävää. Tehtävien luonnissa käytettiin viimeisimpiä teknisiä innovaatioita automaattisen tarkastamisen saralla ja saatuja tehtäviä kokeiltiin lukuvuoden 2008-2009 aikana kahdella noin 300 opiskelijan kurssilla. Tässä työssä esitetään mielenkiintoisimmat tehtävät, jonkin verran tehtyjen kokeilujen tuloksia ja saatua palautetta.

Sivumäärä: 77 Avainsanat: automaattinen tarkastaminen, modSTACK, satunnaistaminen, yliopistotasoiset matematiikan tehtävät, Maxima, graafiteoria, laskento

Täytetään osastolla

Hyväksytty: Kirjasto:

Helsinki University of Technology Abstract of master's thesis
Faculty of Information and Natural Sciences

Author:	Jarno Ruokokoski
Degree Programme:	Engineering physics
Major subject:	Mathematics
Minor subject:	Systems and Operations Research
Title:	Automatic Assessment in University-level Mathematics
Title in Finnish:	Automaattinen tarkastaminen yliopistotasoisessa matematiikassa
Chair:	Mat-1 Mathematics
Supervisor:	Professor Gustaf Gripenberg
Instructor:	Ph.D. Antti Rasila

Abstract:

In this thesis we study automatic assessment of randomized, university level mathematics exercises. We use one system meant for this purpose and implement exercise collections from different mathematical topics.

The selected topics of mathematics are calculus and graph theory. Moreover, something about mathematical proving is presented. The topics were selected to fit for teaching university level mathematics in the department of automation and systems technology and the department of computer science and engineering. Calculus is assumed to be familiar, but graph theory are presented such that understanding is possible without earlier knowledge.

We wrote about 80 exercises for this thesis. We used the latest technical innovations in automatic assessment in the writing of the exercises, and we tested received results in two courses, in which participate about 300 students during the semester 2008-2009. We will present the most interesting exercises, some results of the experimentations and received feedback in this thesis.

Number of pages: 77 Keywords: automatic assessment, university level mathematics exercises, randomization, modSTACK, Maxima, graph theory

Department fills

Approved: Library code:

Preface

This thesis has been written at the Department of Mathematics and Systems Analysis in the Helsinki University of Technology during the years 2008 and 2009. It was written to the project of automatic assessment in mathematics. I would like to thank my instructor Antti Rasila, the coordinator of the project, for his guidance and numerous discussions during the writing process of this work. I also wish to thank my supervisor, professor Gustaf Gripenberg, for his recommendations and comments about this thesis.

I am grateful to Matti Harjula for his technical support regarding STACK and modSTACK, and for his new implementations to modSTACK. Without them the thesis would never have become completed. I also wish to thank numerous students for their raw feedback regarding exercises.

Finally, I would like to thank my parents and siblings for securing a comfortable childhood home. Especially, I wish to thank my girlfriend Annika for her love and support.

Otaniemi, August 27, 2009

Jarno Ruokokoski

Contents

Chapter 1

Introduction

In this thesis we deal with the problem of creating automatically assessed mathematics exercises at the university level by using computer aided assessment (CAA) systems, which can generate and assess exercises. These CAA systems almost always include a computer algebra system (CAS), like Mathematica or Maxima, which execute all computations. Our main goal is to research possibilities and restrictions of these CAA systems, when we use them to write randomized exercises at university level.

By randomization we mean that the parameters of an exercise are chosen at random so that, with a high probability, every student gets a different instance of the exercise. Only the general idea of the solution process will remain stable through different instances of the exercise.

The key questions motivating our research are the following:

1. What kind of exercises can we write?
2. Is it possible to write sufficiently many instances of the exercise such that they can be solved by using a same method?
3. Can the instances be equally difficult to solve?
4. What benefits do we get from using such system compared to traditional exercises?

We will research these key questions by using a particular CAA system. Several CAA systems, such as STACK [31, 35], AIM [30] and Maple T.A. [20], have been developed to generate exercises mainly at the high school level. Also many suitable exercises have already been developed at that level [30]. A tool for generating, checking, and grading exercises, specific at university level, have not existed until recently. Matti Harjula marked up a suitable system by further developing STACK in his master's thesis [12].

Harjula examined the usage of a CAA system and then considered the features such a system should have and how some of them can be implemented. His thesis also includes statistics from a test course using this modified STACK, which we call modSTACK in this thesis. We will concentrate to use this modSTACK, because we have already compared different systems [27, 29], and these STACK and modified STACK were our choice. We will outline the usage of STACK and modSTACK in Chapter 2.

Let us consider our key question number one. Juhana Yrjölä has developed some exercises mainly from the linear algebra and ordinary differential equations [38]. The main question in this thesis is how to develop exercises at university level using modSTACK such that the exercises are randomized, innovative, and go beyond linear algebra and ordinary differential equations.

The key questions number two and three deal with the number of different instances. It is almost always possible to write in principle infinite number of different instances of the exercise. Seldom they can also be equally difficult, but very often they cannot. It is very important to recognize, to which case each exercise belongs. In this thesis we will show examples about both cases.

In the first case we do not need to include all possible equally difficult instances of an exercise to modSTACK. Our upper bound for number of different instances was selected to be about 900000, because then each of 300 students gets a different instance with a higher probability than 0.95, if every instance is equally probable. 300 is a typical number of students in our test courses, and five per cent is a typical risk level. In real life it is not possible to achieve the uniform distribution, but it is a good approximation.

The last case highlights a somehow better way to randomize exercises. By this new way we need only as many instances, as there are students in a course. In that case the CAA system works such that every time, when the system is opened, a new unused instance is selected, too. ModSTACK is not able to operate in this way yet.

The difficulty level between different instances of the exercise is troublesome to measure. We have only used intuition to measure it in this thesis. However, this step is important, although it would be difficult, too.

In Chapter 3 we will present the writing process of a randomized exercise. In general, the writing process includes four steps: the writing of an exercise, the search for a correct answer, the construction of a grading tool, and the construction of a worked solution. Sometimes we can combine some of these steps, and in general anyone of these steps can be extremely difficult. We do not include the fourth step in this thesis in greater detail, because our main research topic was the writing of the university level exercises. If we want to produce functional exercises, we do not need the worked solutions necessarily. Moreover, Tri Quach researches the possibilities to write worked

solutions in his technical report [26].

In the remaining chapters we will consider all of our key questions by presenting and considering the material we have written. For this thesis we wrote about 80 exercises, some of which will be presented in these chapters. Three different topics of university level mathematics were chosen for this thesis. The topics of mathematics were chosen to answer the need of the three selected basic courses in engineering mathematics in Helsinki University of Technology. The three courses together are called C courses. They includes courses C1, C2, and C3. We will introduce these courses in Chapter 4. The selected three topics of mathematics are calculus, graph theory, and mathematical proofs.

Chapter 5 introduces the abilities of modSTACK. This introduction has been written by using calculus exercises, and thus it includes our first topic. However, this chapter does not present any theorem or proof about the topic. We take them all for granted. If the reader is not familiar with the results of calculus, we recommend to check them from any book about the topic, for example from Robert A. Adams's *Calculus, A Complete Course* [1].

Chapter 5 shows advantages and disadvantages of the automatically assessed exercises. We have chosen five example exercises from this calculus topic, which will show the advantages and disadvantages of modSTACK. We have also chosen one example, which we cannot implement to modSTACK yet.

Our second topic begins in Chapter 6, and it considers graph theory, and basic graph algorithms. We will define all results in this chapter, we do not take anything for granted. In general, it is not difficult to solve the graph problems, which are presented in this chapter. Instead, it is very difficult to write different but still appropriate instances of the exercise from the topic. For example, it is not difficult to solve the shortest minimum path problem for the given graph by using Dijkstra's algorithm. The generation and drawing of the connected, planar, and always same-sized graph to this problem is quite another story. We study a couple of methods to generate above mentioned graphs and introduce them in this chapter. Moreover, we will briefly present the Graphviz program, which draws our graphs, and its connection to modSTACK.

The third topic is the automatically assessed mathematical proofs. It is presented in Chapter 7. This chapter is a very short introduction, because it mostly includes the introduction of the Principle of Induction and one example exercise from this principle. Maybe this topic is the most interesting topic in this thesis, but it is also very difficult topic, and needs a lot of further research.

Chapter 2

ModSTACK

The CAA system chosen for experimental usage at Helsinki University of Technology is STACK, which is free open source software. The CAS system behind STACK is Maxima, which is free open source software as well. The researchers of the Helsinki University of Technology have modified STACK to suit emerging needs. That is why we call the system modSTACK instead of STACK. In this chapter we will present this modSTACK briefly. This introduction includes material from Christopher Sangwin's article [32] and from Harjula's master's thesis [12]. Moreover, this chapter presents some new abilities, which Harjula has marked up for our emerging needs.

We will present modSTACK in two sections. In the first section we will consider some points in the form for defining a question. We will consider the implementation of a computer aided assessment system for mathematics, or the architecture of the STACK, only very briefly. Mainly our assumption is that the modSTACK system has been constructed, and we have the form for defining a question before us.

In the second section we will present some new abilities in modSTACK. They are an if statement and a for loop, an ability to use Graphviz program to draw graphs, and a new method to assess students' answers. Note that we have also programmed a lot of functions for Maxima to help modSTACK. However, we do not present every one of the functions in this thesis, only a few; and we do not present them in this chapter, but in the chapters in which they are needed for the first time.

2.1 The Form for Defining a Question

In this section we will give only the necessary information about the writing of the exercise, and for other information we refer to the earlier mentioned

article and thesis. We assume that we have *the form for defining a question* before us. It is a webform, to which we write exercises.

We have showed the form for defining a question in two parts in Figures 2.1 and 2.2. When we write exercises, there is always a possibility to see an instance of the exercise by pressing the button: Try question. A window that will open then is called the *students' view*. We will see a few examples about students' views among others in Chapter 5.

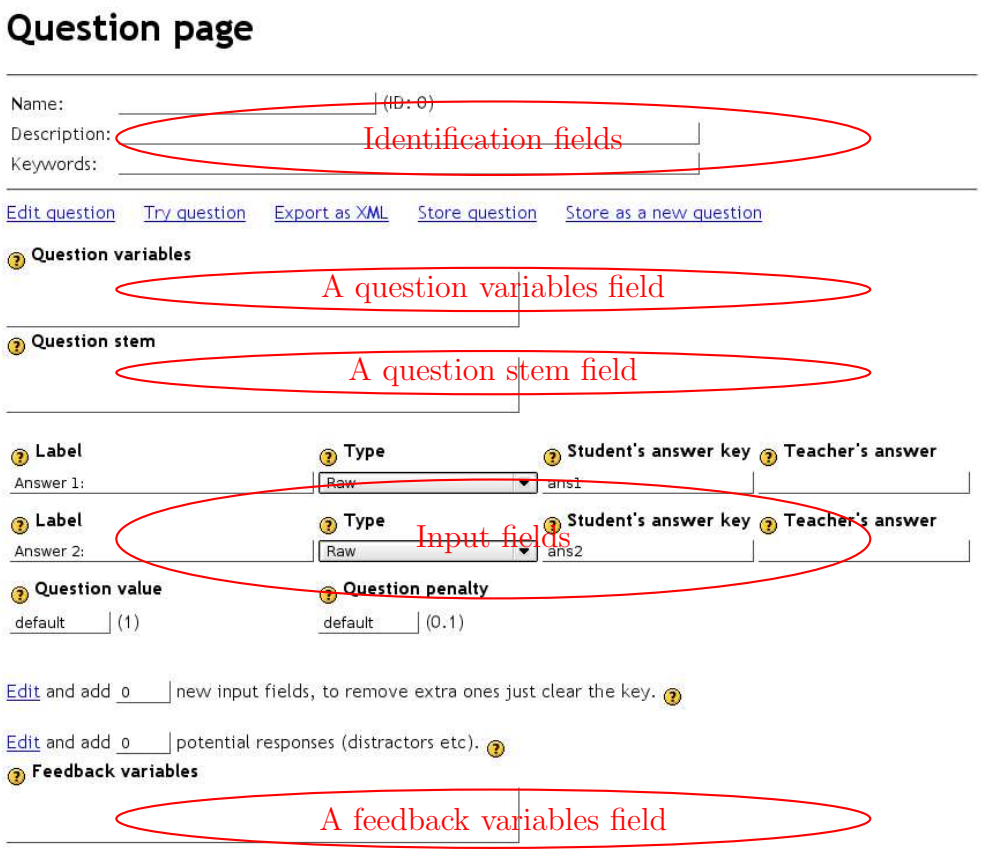


Figure 2.1: The exercise creation form, part 1

The form for defining a question includes the following areas: identification fields, a question variables field, a question stem field, input fields, a feedback variables field, a response processing tree, a worked solution field, and options. The first five areas can be seen in Figure 2.1 and the rest in Figure 2.2. We have highlighted the areas by putting the names of the areas by red color inside red ellipses on each of them.

The first area is *identification fields*. In this area we have three fields,

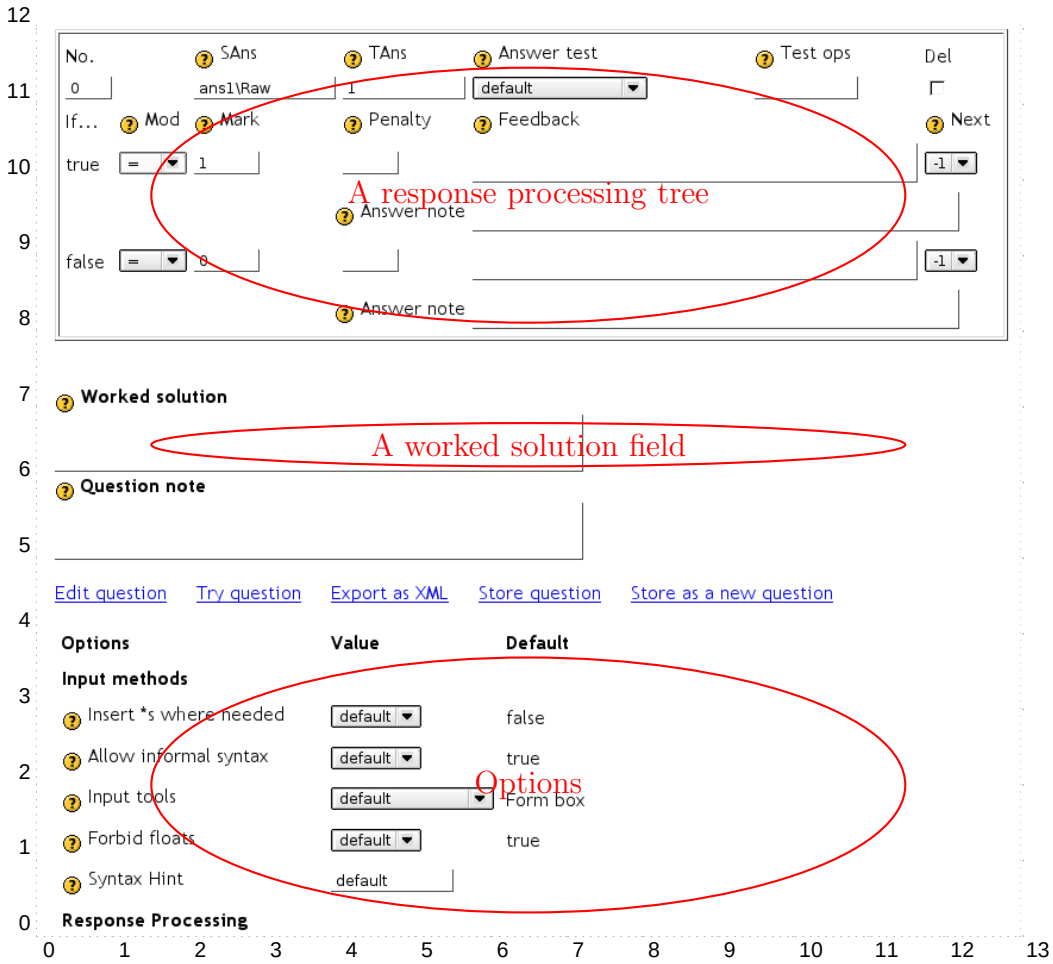


Figure 2.2: The exercise creation form, part 2

for which we can write a name, description, and keywords of the question. If modSTACK has a lot of exercises, then it is possible to search certain exercises by using keywords.

The second area contains *a question variable field*. In this field we set the variables of the question. Every line in this field have to be in form $key = value$. Note that an equals sign is an assign operator in the exercise creation form, whereas a colon is an assign operator in Maxima. The *value* on the right hand side can include Maxima commands, and if an assign operator is needed there, it have to be a colon. Next we will present examples about

possible lines in a question variable field.

$$a = 2 \tag{2.1}$$
$$b = 2 + i \tag{2.2}$$
$$c = true \tag{2.3}$$
$$l = [2, 3, 4, 5, 6, 7] \tag{2.4}$$
$$s = \{2, 3, 4, 5, 6, 7\} \tag{2.5}$$
$$A = \text{matrix}([4, 6, 0], [8, 0, 5], [6, 2, 2]) \tag{2.6}$$
$$d = \text{determinant}(A) \tag{2.7}$$
$$r = \text{rand}(y) \tag{2.8}$$
$$g = \text{for } i : 1 \text{ thru } 100 \text{ do } (a : a + l) \tag{2.9}$$
$$a = a \tag{2.10}$$
$$f = \text{jrRandom3}(-10, 10, [2, l]) \tag{2.11}$$

When an instance of the exercise is generated, STACK and modSTACK send these variables to the CAS system Maxima, which executes them. After that the received results return to STACK or modSTACK, and they will be used in other areas.

Let us consider example lines above. Assignments (2.1), (2.2), and (2.3) set that the variable a is the real number 2, the variable b is the complex number $2 + i$, and the variable c is the Boolean value *true*, respectively. Assignment (2.4) constructs a list, which has the elements $2, 3, 4, 5, 6$, and 7; whereas Assignment (2.5) constructs a set, which has the same elements as well. The difference between lists and sets is that the order of the elements is relevant for lists and irrelevant for sets. In this thesis we call this square bracketed list by a *Maxima list*, because it represents a list in Maxima. Assignment (2.6) constructs the following matrix:

$$\begin{pmatrix} 4 & 6 & 0 \\ 8 & 0 & 5 \\ 6 & 2 & 2 \end{pmatrix}$$

This matrix command is called a *Maxima matrix* in this thesis. Note that a list whose elements are lists is never a matrix in Maxima.

Assignment (2.7) uses Maxima command *determinant*. We can use almost any Maxima function. A comprehensive list of Maxima functions can be found, for example, from the official web paged for the Maxima project [21]. However, some of the commands are prohibited and some of them are replaced by other commands. A comprehensive list of allowed and replaced functions can be found, for example, from the official wiki for the STACK

project [36]. Assignment (2.8) is an example about a substituted function. It randomly selects an element from the list l and sets that r is this selected element.

Assignments (2.9) and (2.10) are here to clarify one technical restriction. Remember that every line in a question variable field have to be in form $key = value$. The variable g is *done*, when Maxima has executed Assignment (2.9). However, this line modifies the value of the variable a. It is done on the right hand side of assignment and that is why an assignment operator have to be a colon.

If we want that the variable a is correct in a response processing tree, which is presented later in this section, then we have to put Assignment (2.10) below Assignment (2.9).

We can also use own Maxima functions, if we have coded them to a distinct text file, and this file have been connected to modSTACK. We have written about 70 Maxima functions for this thesis, and we will present some of them in context of our example exercises in Chapters 5, and 6, if it is necessary because of clarity. The one example about our own Maxima functions is Assignment (2.11). This $jrRandom3$ is a random function, which needs three parameters. The first is a lower bound, the second is an upper bound and the third is a list, which contains all prohibited values. When our example assignment is executed, the returned value belongs to the set $\{x \in \mathbf{Z} | x \leq 10, x \geq -10, x \neq 2, x \neq l\}$. Note that we do not know the value of l beforehand, but we know that f cannot get the same value as l.

A question stem field generates the layout of the exercises to the students' view. Basically, this field includes LaTeX code. We do not present here anything about LaTeX, and if the reader is not familiar with the software, a good manual to study use of the software is Frank Mittelbach and Michel Goossens's The LaTeX Companion [22]. The question stem field can include variables of the question variable field, if there are @ symbols before and after the name of the variable. For example, if we need to use the value of the x variable, we can use it by putting @x@ in the LaTeX code in the question stem field.

The fourth area is *input fields*. This area generates the answer fields to the student. We have put two answer fields, Answer 1 and Answer 2 to Figure 2.1. Note that the layout of this fourth area in the students' view, can be complicated. We can generate multiple input fields [12, p. 54], and if we use so-called custom layout [12, p. 58], we can put input fields practically anywhere in the LaTeX code, as we can see from the example exercise 3 in Figure 5.3. We can also ask intermediate steps, and we will discuss these steps later in Chapter 5. There are a couple of examples about a students' view, which include the results of these input fields and a question stem field

in Chapter 5 as well.

The fifth area is *a feedback variable field*. To this field we put the tests, which are needed to check before the assessment is executed. This field is like a question variable field. The comments about a question variable field also relate to this field.

The next area is *a response processing tree*, which is also called a potential response tree, which consists of linked potential response nodes. We will briefly present a principle of the tree here, it is described more fully in Sangwin and M.J Grove's conference proceeding [32]. Each of the nodes may be traversed once only, which means that technically the tree is an acyclic directed graph. Each potential response node provides a mechanism, by which two expressions can be compared by using a specified Answer Test. Answer Tests are presented, for example, on the official wiki for the STACK project [36]. Depending on the result of Answer Test, either the true or false branch is executed. Both branch have the opportunity to do any of the following:

1. Adjust the mark and penalty for this attempt.

2. Generate and add specific feedback.

3. Generate and add a specific answer note, used by the teacher.

4. Proceed to another node, or end the process.

We have put one node of a response processing tree to Figure 2.1, and we will show one example about the principle of the response processing tree in Chapter 5. In general, the principle of the response processing tree is easy to understand, but it is not very effective, if the exercise is complicated. Then the tree is very hard to make and it's editing is tedious afterwards [38, p. 39]. For example, we had to use about 50 potential response nodes in one of our exercise. Then it is very troublesome to add, for example, one node between the nodes 10 and 11.

To the *worked solution field* we can set the worked solution of the exercise. We do not go into this area in detail, because our main goal is to study the creation of the exercise. Tri Quach generated exercises, which included accurate worked solutions, and presented them in his technical report [26].

The last area in the form for defining a question is *options*. There are a lot of predominantly Boolean-valued switches, for which we can for example accept decimal numbers et cetera.

2.2 The New Abilities in modSTACK

In this section we present four new abilities, which Harjula programmed in order to meet our needs. The first ability is a modified grading system. With the normal STACK, the most popular grading system is so-called linear system. It works so that a penalty grows linearly for every return time after the first one. For example, let us assume that it is possible to get one point from an exercise and there are ten return times available. Then the penalty grows 0.1 for every return time. This penalty is then reduced from the received points in that return time. The maximum of all received points is the final result.

With this new modified grading system it is possible to give three free times to try, and the penalty begins to grow after these three tries have been used. The penalty grows as in the linear penalty system. Feedback that was collected from students during the semester 2008-2009 gave a more positive response to this system than the earlier mentioned linear penalty system. That is why we recommend to use this modified system later, too.

The second ability is an if statement and the third a for loop. We need them in modSTACK, because sometimes we need to put some symbols to student's view, and sometimes not, and the LaTeX version *ifthen* is not enough. Moreover, we need both methods in graph drawing in Chapter 6. Formally an if statement looks as follows:

```
|$ begin if @#condition#@ $|
symbols                    ,
|$ end if $|
```

where condition is a Maxima command, which will return true or false, symbols are symbols, which we will see in the student's view, if the condition is true.

Formally a for loop looks as follows:

```
|$ begin for _variable1_ in @#set1#@
             _variable2_ in @#set2#@ ... $|
something
|$ end for $|
```

where can have so many "_variableX_ in @#setX#@" in the first line as are needed. That is why we have put ... in this line. Note that the sizes of the index sets in the first line have to be equal. Moreover, index sets have to be normal Maxima lists marked by square brackets. This means that the matrices of Maxima are not suitable.

We can use both of these abilities within each other. We can use the same variable, which we have used in the earlier for loop, in the next if statement or for loop instead of @#condition#@ or @#set1#@.

The following artificial example will clarify these comments. We have determined in the question variable field that

```
x = [0,1,2,3,4,5,6,7,8]
p = reverse(x)
d = create_list(i^2,i,x)
t = create_list(is(i^2<9-i),i,x).
```

We have put to the question stem field the following code:

```
\begin{tabular}{r|l|l}
|$ begin for _x_ in @#x#@ _p_ in @#p#@
              _d_ in @#d#@ _t_ in @#t#@ $|
|$ begin if _t_ $|
_x_ & _p_ & _d_ \\
|$ end if $|
|$ end for $|
\end{tabular}.
```

Now the result in the students' view looks like the following:

0	8	0
1	7	1
2	6	4

We will show another example, which uses the if statement and the for loop in Chapter 6. This example also includes Graphviz codes.

The fourth new ability is the inclusion of the Graphviz program. This program is an open source graph visualization software [10], which can draw graphs from a simple text language. We do not need every advanced ability of the program. We will only use the NEATO subprogram, which uses the DOT language. Stephen North has written a suitable manual about NEATO [25], which can serve as a tutorial for understanding how to use the subprogram. Some parts in this manual are obsolescent, the most up-to-date documentation is provided on the earlier mentioned web pages.

We will not present the DOT language here very exactly. We will only show how we can use it in modSTACK. The notation is very simple as we can see below.

```
|$ begin dot neato options$|
graph G {
 connections
}
|$ end dot $|
```

Note that the lines: "graph G {*connections*}" include the source data, which have been written by the DOT language as such. The options include commands, which we may use on the command line on the normal

terminal. For example, it can include options like $-Gepsilon = 0.001$ $-Gmaxiter = 1000$ [25, p.8]. The lines "graph G {*connections*}" can include earlier mentioned if statements and for loops, and thus it is possible to draw random graphs. We will return to this observation in Chapter 6. We will also show some DOT code in that chapter, too.

Chapter 3

Writing Process of Exercises

In this chapter we present the writing process of CAA exercises on a very general level. It includes four steps: the writing of an exercise, the search for a correct answer, the creation of a grading tool, and the creation of a worked solution.

The first step contains the writing of a question. In this step we choose an exercise type and a method to solve the exercise. For example, an exercise requests integration such that every instance of the exercise needs the integration by parts equally many times. The first step includes also the randomization of the parameters of the exercise. The randomization have to be such that every instance is solvable by the earlier selected method and every solution have to be equally difficult. Moreover, the simpler the solutions are, the better the exercise is. In practice, students make a lot of mistakes, if correct solutions are complicated

We will show an example about this problem. Let us assume that an exercise asks the characteristic values and vectors of a given matrix. The matrix have been chosen such that these characteristics are complex-valued. Let us also assume that two students get the following matrices:

$$\mathbf{A} = \begin{pmatrix} 11 & 5 \\ -5 & 5 \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} 14 & 10 \\ -10 & 5 \end{pmatrix}$$

Before calculation we see that both matrices are equally simple. If we calculate the characteristic values of the matrices, we get that $\lambda_{1,2}(\mathbf{A}) = 8 \pm 4i$ and $\lambda_{1,2}(\mathbf{B}) = (19 \pm \sqrt{319}i)/2$. It is presumable that the student that gets the matrix $\mathbf{B}$ will make more mistakes. So we have to be careful, when we randomize exercises.

The second step in the creation process is a search for a correct answer. We need this step, because modSTACK always generates a teacher's answer to the exercise. In addition, this answer helps us to test the grading tool in

step three, naturally the teacher's answer has to give the maximum points available. Sometimes the correct answer is unique, and then it is useful and nimbler to randomize the correct answer, and then go backward to get the question. In that case the second step includes the first step.

The previous exercise, which considered matrices $\mathbf{A}$ and $\mathbf{B}$, was a good example about this backward process. We can set that the correct answers are, for example, $\lambda_{1,2}(\mathbf{A}) = a \pm bi$, where $a, b \in \{z \in \mathbb{Z} \big| |z| < 10\}$. Then a suitable matrix is

$$\mathbf{A} = \begin{pmatrix} \alpha & \frac{b^2+(\alpha-a)^2}{-\gamma} \\ \gamma & 2a - \alpha \end{pmatrix},$$

where α and γ are free variables. For example, they can belong to the same set as a and b. Now every correct answer is equally difficult to mark.

In the situation of many correct answer we cannot unite the steps. In addition, if the number of the intermediate steps grows, then the probability, that the correct answers of one or more answer field are not unique, grows also. Because our experience and feedback from our students tell us that intermediate steps are necessary, this situation is the most typical one. In this thesis we will give example exercises about both situations.

The third step is the creation of a grading tool. A good rule of thumb in this step is as such obvious, but nontrivial to guarantee: Students can get full marks if and only if their answers are correct. Because there can be a great number of correct answers, it is not advisable to search and collect all of them to the certain set, and then check that the student's answer belongs to this set. We have to invent other ways. For example, if we request the characteristic values and vectors of a matrix $\mathbf{A}$, then it is easy to check that answers satisfy the characteristic equation $\mathbf{Ax} = \lambda\mathbf{x}$. We will see more examples about these other ways in Chapter 5.

The fourth and the last step is the creation of the worked solution. But as we noted in Chapter 2, we do not go through this step in details.

Chapter 4

Three Basic Courses in Mathematics

During fall 2008 and spring 2009 we have generated several exercises to modSTACK such that all of them are suitable to the so-called C courses at Helsinki University of Technology. The C courses include three basic course in mathematics, all of which are spread over one semester. The three courses are called C1, C2, and C3, respectively, although C3 has been split into two courses nowadays. However, it is not essential in this thesis. The C courses is designed for first and second year students, who study mainly in the department of automation and systems technology, and the department of computer science and engineering.

The C1 course includes material from three topics: discrete mathematics, linear algebra, and real variable integral calculus with numerical methods. The C2 course is a direct advanced course to the C1 course. It includes material about one variable differential equations, series, partial derivation, multiple integration, extreme values of functions defined on restricted domains, and introduction to algebra. The C3 course is a direct advanced course to the C2 course. It includes introduction to the function theory of one complex variable; material about Fourier, Z, and Laplace transformations; an introduction to differential equation systems; and advanced material about linear algebra.

The course books of these courses are Adams's *Calculus, A Complete Course* [1], David Lay's *Linear Algebra and its Applications* [17], Norman Biggs's *Discrete Mathematics* [2] and Erwin Kreyszig's *Advanced Engineering Mathematics* [16].

Chapter 5

Calculus Exercises

In this chapter we will discuss the abilities of modSTACK by using calculus exercises such that we will not present any theorem or proof. We take all of them for granted. If the reader is not familiar with the results of calculus, we recommend to check them from any book about this section, for example from Adams's *Calculus, A Complete Course* [1], which, however, does not include linear algebra enough. Something about it can be read, for example, from Lay's *Linear Algebra and Its Applications* [17]. This chapter concentrates on introducing exercises written from this area. Then it shows advantages or disadvantages of automatically assessed exercises. We have chosen five example exercises from this topics to do so. We have also chosen one example, which we cannot implement in modSTACK yet. There are also other exercise types whose implementations are very difficult or impossible. One of them is mathematical proving. We will discuss this topic in Chapter 7. All examples look almost same as the official students' view. Answer fields, for which the students will write their answers, are blue in every one of the example figures.

Our first example exercise in Figure 5.1 is very easy. The exercise requests an integral of a rational function. The students' view includes only one field, the result of the integral. We can write this kind of exercises with the normal STACK, so the exercise does not include any advanced abilities.

The exercise is randomized, and it has 16000 different instances. The number of different instances is smaller than our upper bound, because it is not possible to write equally difficult, sufficiently easy instances from the question of this type. We want to write an exercise, which requests to integrate a given function. We want that the integration needs a partial fraction decomposition. Thus we set that the denominator have been gotten from the form $(x + a)(x + b)$, where a, and b are unequal integers such that, for example, $|a| + |b| = 11$, and $a, b \neq 0$.

We demand that the partial fraction is $\frac{\alpha}{x+a} + \frac{\beta}{x+b}$, where α and β are

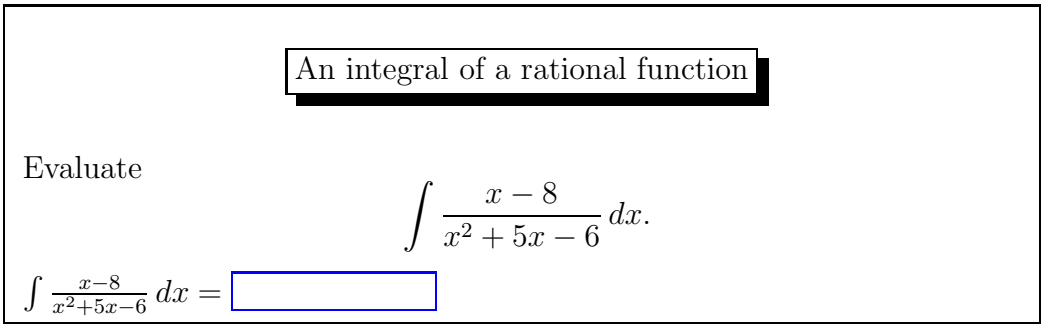


Figure 5.1: The first example exercise

integers. We get the integrand by expanding the partial fraction, and simple instances if we set, for example, that $a, \alpha, \beta \in \{x \in \mathbb{Z} | |x| \leq 10, x \neq 0\}$ and $b = \pm(11 - |a|)$. In that case the number of different instances is $2 \cdot 20^3 = 16000$.

We think that randomization is a main benefit in CAA systems, because every student gets a different exercise with a high probability. So, students cannot copy an answer from their friends, because a correct answer of their exercise is not the same as their friends' correct answer. However, randomization makes the writing of the exercises much more difficult, as we will see later.

Other benefits in CAA systems are freedoms of return times and places, and the immediacy of feedback. Students need only a connection to the Internet, and then they can return their answers at home at midnight if they want to. Note that STACK separates two kinds of feedback. The first feedback is associated with the syntax of the student's answer, the second with its interpretation semantics. We do not consider syntax problems at all, whereas interpretation semantics is considered in context of each example exercise. More about these two kinds of feedback can be found, for example, from Sangwin's article [32].

Now we will search any disadvantage in our first example exercise in Figure 5.1. Let us assume that some students can form a partial fraction decomposition to this function. Let us also assume that they are not able to integrate the decomposition. So, they get no results, nor points. Then an easy result is to put some other field, which will request the partial fraction decomposition. When students see this version, they may think that this function needs a partial fraction decomposition. Then students need not to observe it by self. This matter highlights a difficult problem: on the one hand we need to ask intermediate steps, on the other hand intermediate steps guide students toward a right answer. Maybe in the future we can use completely

Limit of a function of two variables

Investigate the limiting behavior of

$$f(x,y) = \frac{12x^2y}{4y^2 + 8x^2y + y - x^2}$$

as (x, y) approaches $(0, 0)$.
If the limit exists, it is enough to answer to the multiple choice question.

$$\lim_{(x,y)\to(0,0)} \frac{12x^2y}{4y^2 + 8x^2y + y - x^2} = \begin{cases} \square & \text{The limit does not exist} \\ \square & 0 \\ \square & 1 \\ \square & 2 \\ \square & 3 \\ \square & -1 \end{cases}$$

If the limit does not exist, show two curves, along which the function does not approach the same limit.

$$f(\square, \square) \to \square.$$
$$f(\square, \square) \to \square.$$

To the right hand side of the arrow you shall put the limits. To the left hand side you shall put the curves. The computer interprets your answer such that it substitutes the x of the function by your first variable on the left hand side and y by your second variable. Then it computes the limit such that your only variable goes to the zero.
Thus, these answers cannot be acceptable: $f(0, 1) \to 0$ (no variables), $f(x, 4 * y) \to 2$ (too many variables) and $f(x, 4) \to 3$ (do not approach the origin). These answers are acceptable: $f(0, x) \to 1$ (we approach the origin by straight line $x = 0$) and $f(x, 4 * x) \to 3$ (we approach the origin by the straight line $y = 4x$).

Figure 5.2: The second example exercise

free answer fields, nowadays it is not possible. According to the collected feedback it is better to put intermediate steps than not. Besides, we think that rest of our examples, which includes intermediate steps, have their own

place in basic courses in mathematics. They are an easy way to familiarize oneself with the subject and that is why they are a good part of teaching. However, it is important to remember that these exercises cannot substitute standard paper exercises completely yet.

Let us see an another example exercise in Figure 5.2. This exercise requests a limit of a given function. We have written the exercise such that the correct answer is always the same: $f(x, ax) \to 0$ for all $a \in \mathbb{R}$ but $f(x, x^2) \to 1$. Note that the exercise includes a multiple choice question and multiple answer fields.

We will briefly show the creation process of this exercise. We begin by setting that the general function is:

$$f(x,y) = \frac{ax + bx^2 + cy + dy^2 + exy + fx^2y + gxy^2 + hx^2y^2}{ix + jy + kx^2 + ly^2 + mxy + nxy^2 + ox^2y + px^2y^2}.$$

We want that the following statements are always true:

$$f(0,0) = \text{"}\frac{0}{0}\text{"}, \tag{5.1}$$

$$\lim_{x\to 0} f(x,0) = 0, \tag{5.2}$$

$$\lim_{y\to 0} f(0,y) = 0, \tag{5.3}$$

$$\lim_{x\to 0} f(x,\alpha x) = 0 \text{ and} \tag{5.4}$$

$$\lim_{x\to 0} f(x,x^2) = 1, \tag{5.5}$$

where $\alpha \in \mathbb{R}\backslash\{0\}$. Statement (5.1) is always true. Statement (5.2) gives that $a = 0$ and $b = 0$, and Statement (5.3) gives that $c = 0$ and $d = 0$. From Statement (5.4) we get that $j \neq 0$, and from Statement (5.5), that $i = 0$, $h = p$, $g = n$, $e = m$, $f = l + o$ and $k = -j$.

Hence we have got the following result

$$f(x,y) = \frac{exy + (o+l)x^2y + gxy^2 + hx^2y^2}{jy - jx^2 + ly^2 + exy + gxy^2 + ox^2y + hx^2y^2},$$

where e, g, h, j, l, o are otherwise free real-valued variables, except $j \neq 0$ and $o \neq -l$. However, we think that the exercise is too difficult yet, so we set that $e = g = h = 0$, and get the following function:

$$f(x,y) = \frac{(o+l)x^2y}{jy - jx^2 + ly^2 + ox^2y}.$$

This result establishes the function of the exercise. In our test exercise we have set that $j, l \in \{x \in \mathbb{Z} \big| x \leq 10, x \neq 0\}$ and $o \in \{x \in \mathbb{Z} \big| x \leq 10, x \neq 0, x \neq -l\}$.

The third example exercise is in Figure 5.3. This exercise shows, how complicated the input fields can be. Note especially the answer fields for the lower and upper limits. This example exercise shows how we can practice the use of the spherical coordinates with modSTACK. A careful choice of the intermediate steps can lead a good result.

Triple integral and the spherical coordinates

In this exercise we practice the spherical coordinates. Evaluate

$$\iiint_V 4\sqrt{z^2+y^2+x^2}\,dV,$$

where V is the octant of the ball such that $x \leq 0$, $y \leq 0$, $z \geq 0$ and $z^2+y^2+x^2 \leq 25$. You have to use spherical coordinates. Use instead of the standard coordinates $[\rho, \phi, \theta]$ the letters $[r, f, t]$. Note that f measures the angle between the line, which goes through origin and (x, y, z), and the positive z-axis. Put to the first step your integral with the spherical coordinates and in the each of the following steps your results, when you have integrated one variable away. Note that in the iterated integral $\iiint_V f(x, y, z)\,dxdydz$, the integral $\int f(x, y, z)\,dx$ is the innermost integral and it must be evaluated first.

$$\iiint_V 4\sqrt{z^2+y^2+x^2}\,dV =$$

$$\int_{\square}^{\square}\int_{\square}^{\square}\int_{\square}^{\square} \square\, d\square\, d\square\, d\square =$$

$$\int_{\square}^{\square}\int_{\square}^{\square} \square\, d\square\, d\square =$$

$$\int_{\square}^{\square} \square\, d\square =$$

$$\square$$

Figure 5.3: The third example exercise

Let us see another example exercise in Figure 5.4. We can implement matrix formed answer fields very easily. This matrix notation is introduced

more precisely in Harjula's master's thesis [12, p.55]. We do not need to fix the dimensions of the matrix beforehand, because modSTACK generates matrices from scratch. The only drawback is that students cannot decide matrix dimensions by themselves. So, if the exercise is on the fundamental level, and it asks, for example, to compute $\mathbf{A} \cdot \mathbf{B}$, then the student do not need to deduce the dimensions of the multiplication matrix.

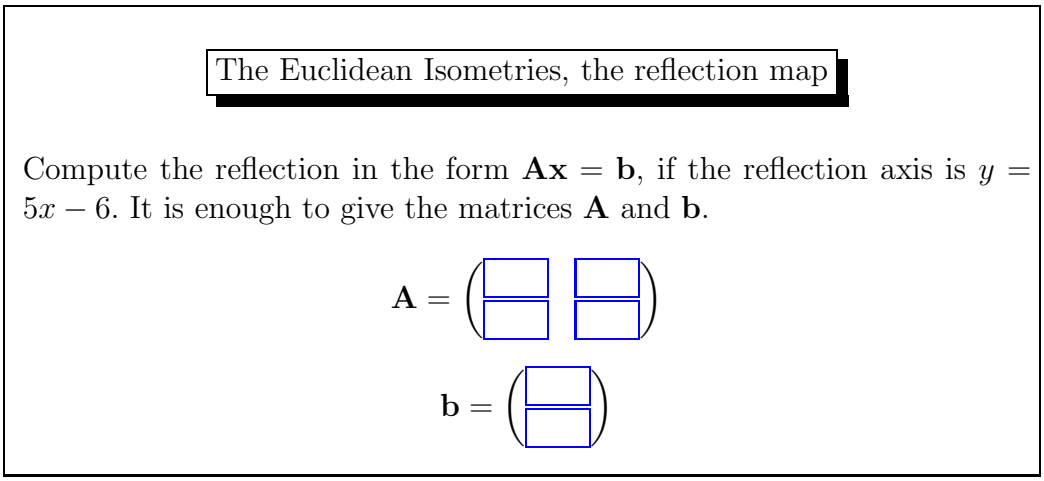


Figure 5.4: The fourth example exercise

The last example exercise is also a matrix form question, which is presented in Figure 5.5. We have also presented the potential response tree of this question in Figure 5.6. Every potential response node has been presented by a blue box. It contains a node number, a considered question, and true and false branches. The branches include adjustments to points. If both branches lead to the same node, then there is only one arrow below the box. Otherwise there are two arrows, one for either of branch. The correct answers of this question are

$$\begin{pmatrix} 6 & -3 & 6 & 4 \\ 0 & 9 & -1 & 2 \\ 0 & -9 & 1 & -2 \\ 0 & -18 & 2 & -4 \end{pmatrix} \mathbf{x} = \begin{pmatrix} -9 \\ 2 \\ k-45 \\ h-54 \end{pmatrix}.$$

$$k = 43$$
$$h = 50$$
$$\mathbf{x} = \begin{pmatrix} \frac{-25-17a-14b}{18} \\ \frac{2+a-2b}{9} \\ a \\ b \end{pmatrix}.$$

The exercise have been written such that the matrices of every instance are singular such that the Gaussian algorithm produces two zero rows. This means that the final result includes two variables, which are marked by a and b in our model answer. Students may not understand the need for both variables, but then the algorithm gives partial marks.

The consistence of a linear system

Determine the values of h and k such that the given linear system is consistent. Moreover, compute a solution with your h and k. If you need any parameter, you can use any letters, except k, h, e and i.

$$\begin{pmatrix} 6 & -3 & 6 & 4 \\ 6 & 6 & 5 & 6 \\ -30 & 6 & -29 & -22 \\ 36 & 0 & -34 & -28 \end{pmatrix} \mathbf{x} = \begin{pmatrix} -9 \\ -7 \\ k \\ h \end{pmatrix}.$$

You have to give one intermediate step about the Gaussian elimination. The intermediate step need to be such that all elements of the first column are zero, except the first one; and you have not calculated more zeros elsewhere yet. Of course, it is possible to get zeros elsewhere in pursuance of computing the zeros of the first column.

$$\begin{pmatrix} \square & \square & \square & \square \\ \square & \square & \square & \square \\ \square & \square & \square & \square \\ \square & \square & \square & \square \end{pmatrix} \mathbf{x} = \begin{pmatrix} \square \\ \square \\ \square \\ \square \end{pmatrix}.$$

$k = \square$

$h = \square$

$\mathbf{x} = \begin{pmatrix} \square \\ \square \\ \square \\ \square \end{pmatrix}$

Figure 5.5: The fifth example exercise

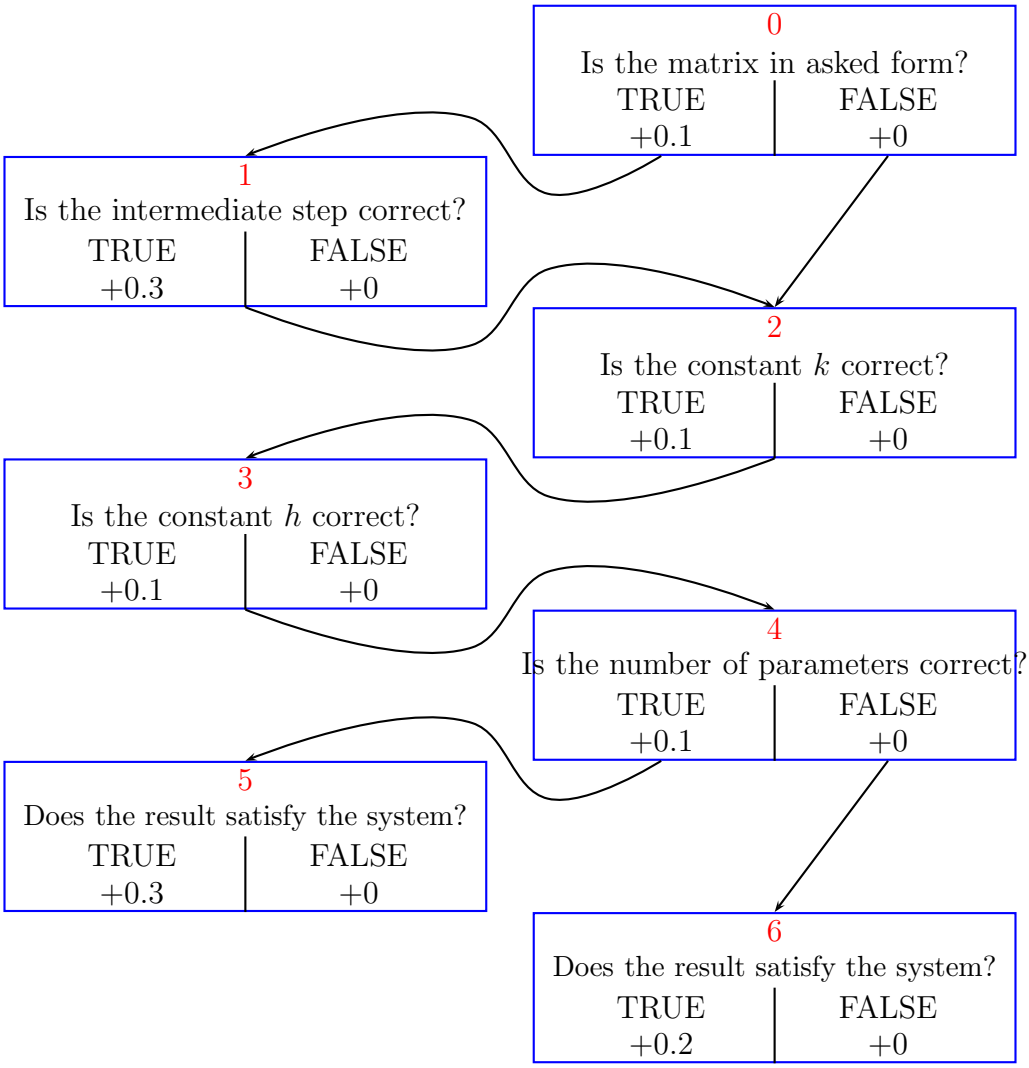


Figure 5.6: The response processing tree

We have shown five example exercise of mathematics at university level. All of them have been executable examples of the modSTACK exercises. Next we show an example, which is impossible to implement to modSTACK. The exercise have been presented in Figure 5.7.

We can see that there are only two possible answers to this exercise: the series converges or diverges. In general, the modSTACK exercises can be returned several times and for every return the student get some penalty to the maximum points available. If there are only two possible choices, the student will get 1 point or 0.9 points from this exercise without understanding

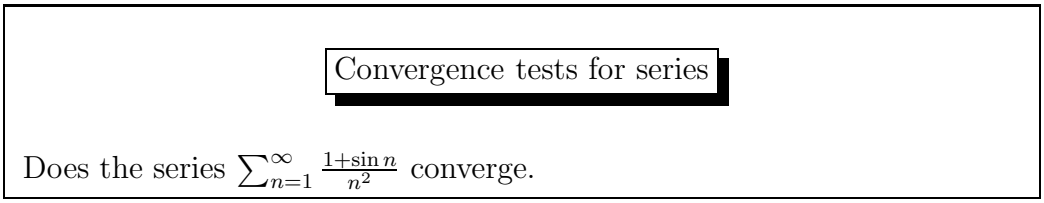


Figure 5.7: The sixth example exercise

anything about it.

Hence we need also to ask some intermediate steps for the student. But the part of the solution of this exercise is to select appropriate test to check the convergence. So, if we select the correct convergence test beforehand, and ask some intermediate steps about the use of this test, a student needs not to select an appropriate test self. Thus the exercise becomes too easy.

That is why this exercise type cannot be implemented to modSTACK yet. If we could put so-called modified input fields to exercises, this exercise became accessible. By modified input fields we mean fields, which can interact with the rest of the fields in the question. For example, there could be a multiple choice question about the appropriate test in example in Figure 5.7. Whenever the student chooses anyone of the test, there appear more questions about just this selected test, like in Figure 5.8, where a student have selected a comparison test.

Convergence tests for series

Test the series $\sum_{n=1}^{\infty} \frac{1+\sin n}{n^2}$ for convergence.

Select an appropriate convergence test:
- ☐ An integral test
- ☒ A comparison test
- ☐ A limit comparison test
- ☐ A ratio test
- ☐ A root test

So, you have selected the comparison test. Please answer to the following questions:
If you want to use the comparison test, you need a function $f(n)$ and a constant k, which satisfy the following inequality:

$$0 \leq \frac{1+\sin n}{n^2} \leq kf(n), \qquad \text{for } n = 1, 2, 3 \ldots$$

You have to select a function $f(n)$ so, that it is known to formulate a converging or diverging series $\sum_{n=1}^{\infty} f(n)$ (check this list). Please put your function $f(n)$ and constant k to the next fields:

$f(n) =$ [ ], and
$k =$ [ ].

Does the series converge or not?
- ☐ Yes
- ☐ No

Figure 5.8: The sixth example exercise (continued)

Chapter 6

Graph Theory

In this chapter we will consider writing of graph theory exercises. We will begin by defining the basic properties and definitions of a graph in Section 6.1. They have been described more fully, for example, on a general level in the Douglas B. West's *Introduction to Graph Theory* [37], and with a view toward the computer programming in Cormen et al.'s *Introduction to Algorithms* [5].

Secondly we will consider graph theory exercises, and how they fit to our four-step-model in Chapter 3. We will analyze the first three steps of this model in Sections 6.2, 6.3, 6.4, and 6.5.

The first step, the creation of an exercise, includes two different and very difficult problems. The first problem is the illustration of graphs. ModSTACK is able to generate layout to all of our previously presented exercises in Chapter 5, because they mainly include normal LaTeX code. However, it is not so easy to draw graph using LaTeX . Moreover, drawing graphs is a difficult problem, in fact, it form a whole discipline. We will consider this drawing problem very briefly in Section 6.2.

The second problem is the generation of suitable randomized graphs. This problem leads us to the theory of random graphs. It was originated in a series of papers published in the period 1959-1968 by two outstanding Hungarian mathematicians, Paul Erdös and Alfred Rényi. In the early days, the literature on the subject was scattered around several probabilistic, combinatorial and general mathematics journals. In the late seventies, Béla Bollobás became the leading scientist in the field and contributed dozens of papers, and an outstanding monograph *Random Graphs*, whose second edition was printed in 2001 [4]. The appearance of that book stimulated the research further, shaping up a new theory. Over the nineties several new, beautiful results have been proved and numerous fine techniques and methods have been introduced. The three highly respected members of the discrete mathematics community, Svante Janson, Tomasz Luczak and Andrzej Rucinski

have published some of these results in their book *Random Graphs* [14] making them easily accessible.

The model of the random graphs introduced by Erdös [7] is very natural, but it is not suitable for us. Our goal is to write randomized exercises to the C courses about graph theory, and that is why we need small, but still randomized graphs. In general, an assumption is that the problem is simpler then, but, it is not true. We will give more details about this situation in Section 6.3.

In the second step in our four-step-model we need to find a correct answer. Our typical exercise requests to find something about the given graph. Because we need the correct answer, we need to be able to solve the given problem with the computer. That is why we need basic algorithms from the graph theory, and we will present them in Section 6.4. Moreover, we will study what kind of graph theory exercises we can write from them to mod-STACK in this section also. We do it by presenting 6 different graph theory exercises written from the algorithms. The presented algorithms have been taken from two different references [5, 8].

In the third step we need to check the student's answer. In general, the graph theory exercises often have several correct answers and that is why this step is nontrivial. Luckily, this step is often not difficult. We will consider this step in Section 6.5. We consider some of the six graph exercises from Section 6.4 and present routines, which handle the student's answers. We will especially consider question: If there are many correct answers, what must be checked to guarantee the correctness of student's answer.

6.1 Definitions

In this section we will give basic properties and definitions of a graph. We will present them as briefly as possible because our main goal is the creation of suitable graph exercise, not certain graph theoretical results. First, we will define what is a graph.

Definition 6.1. A *graph* G is a triple consisting of a vertex set $V(G)$, an edge set $E(G)$, and a relation that associates with each edge two unordered vertices (not necessarily distinct) called its *endpoints*. A *weighted graph* is a graph with numerical labels on the edges.

Note that the graphs defined by Definition 6.1 are called undirected, because the order of the endpoints is irrelevant. If the order is relevant, we use the following definition:

Definition 6.2. A *directed graph* G or A *digraph* G is a triple consisting of a vertex set $V(G)$, an edge set $E(G)$, and a function assigning each edge an ordered pair of vertices. The first vertex of the ordered pair is the *tail* of the edge, and the second is the *head*.

An undirected graph can always be made into a directed graph by letting every edge has both directions. However, the converse is not always true, and that is why it is important to check that the presented algorithm can also operate on directed graphs. In this thesis we do not need them, however, the presented exercises uses only undirected graphs. That is why we will only speak about mere graphs, which always mean undirected graphs. Sometimes definitions are same for both graph types, sometimes not. In the latter case, there are often own definitions for digraphs. If the reader want to compare different definitions, we recommend to use West's book [37].

Note also that it is possible to consider graphs, whose number of vertices, or edges is infinite. We define as follows:

Definition 6.3. A graph G is *finite* if its vertex set and edge set are finite. Otherwise it is *infinite*.

We do not consider infinite graphs in this thesis at all.

In Figure 6.1 we can see a typical illustration of a graph. This example graph has the vertex set $\{1, 2, 3, 4, 5, 6, 7\}$. Each edge have been represented by using a straight line from one endpoint to the other. Moreover, the example graph is a weighted graph, we have marked numerical labels besides the edges.

Definition 6.4. A *loop* is an edge whose endpoints are equal. *Multiple edges* are edges having the same pair of endpoints. A *simple graph* is a graph having no loops or multiple edges. Then we specify a simple graph by its vertex set and edge set, treating the edge set as a set of unordered pairs of vertices and writing $e = (u, v)$ (or $e = (v, u)$) for an edge e with endpoints u and v. When u and v are the endpoints of an edge, they are *adjacent* and are neighbors.

The following definition complements the previous definition.

Definition 6.5. A graph is *loopless* if it can include multiple edges, but not loops. If the vertex v is an endpoint of the edge e, then v and e are *incident*. The *degree* of the vertex v in a loopless graph is the number of incident edges.

Our first example graph in Figure 6.1 is an example about simple graphs. In Figure 6.2 we can see a graph, which has one loop in the vertex 4 and

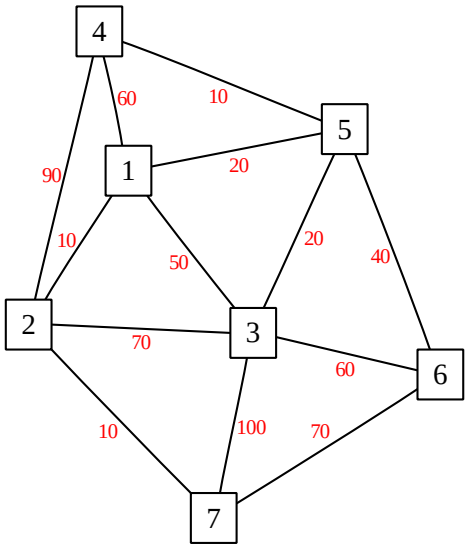


Figure 6.1: An example graph

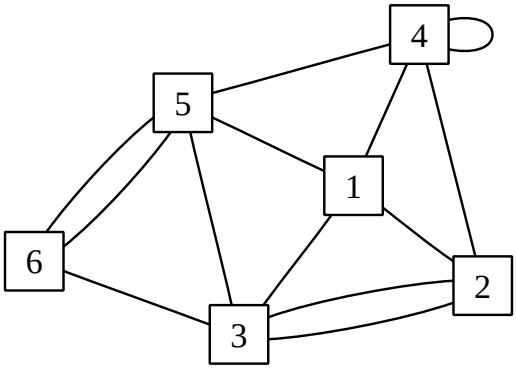


Figure 6.2: A second example graph

four multiple edges, two between the vertices 5 and 6, and two between the vertices 2 and 3. In this thesis we restrict our attention to simple graphs.

Next we want to analyze routes in graphs. We begin by defining a path and a cycle.

Definition 6.6. A graph is a *path*, if it is a simple graph whose vertices can be ordered so that two vertices are adjacent if and only if they are consecutive in the list. The first and last vertex in the list are called the *endpoints of the path*. The path is denoted by normal brackets, and any path from the vertex

u to the vertex v is denoted by using $\rightsquigarrow$. A graph is a *cycle*, if it is a simple graph with an equal number of vertices and edges, and every degree of the vertex is 2.

For example, if a path from the vertex u to the vertex v exists, we denote it by $u \rightsquigarrow v$, and we list the vertices of this path by using the notation $[u, u_1, u_2, \ldots, u_k, v]$. In Figure 6.3 we can see a path. We use this definition of the path to create "routes" in the graph. We do it by using subgraphs.

Definition 6.7. A *subgraph* of graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ and the assignment of endpoints to edges in H is the same as in G. Then $H \subseteq G$ and say that "G *contains* H".

By saying: a graph has a path, we mean that we can construct a subgraph which is a path. This note highlights the following definition.

Definition 6.8. A *maximal path* in a graph G is a path P in G that is not contained in a longer path.

We can also say that a graph has a cycle, which means that we can construct a subgraph which is a cycle.

Next we complement the definition of a path and a cycle.

Definition 6.9. A *walk* is a list $v_0, e_1, v_1, \ldots, e_k, v_k$ of vertices and edges such that for $i \in \{1, 2, \ldots, k\}$, the edge e_i has endpoints v_{i-1} and v_i. A walk is closed, if $v_0 = v_k$. A *trail* is a walk with no repeated edge, and a closed trail is a closed walk with no repeated edge. A *u,v-walk* or *u,v-trail* has first vertex u and last vertex v; these are its *endpoints*. A *u,v-path* is a path whose vertices of degree 1 (its endpoints) are u and v; the others are internal vertices.

The edges are listed in a walk to distinguish among multiple edges when a graph is not simple. In a simple graph, a walk or a trail is completely specified by its ordered list of vertices. That is why we usually name a path, cycle, trail, or walk in a simple graph by listing only its vertices in order, even though it consists of both vertices and edges. When we discuss a cycle, we can start at any vertex and do not repeat the first vertex at the end. We use parentheses to distinguish a cycle from a path.

Now we ask: can we be sure that a graph has a path such that the endpoints of the path are arbitrary two vertices in the graph? We need a definition of a connectivity.

Definition 6.10. A graph G is *connected* if $u \rightsquigarrow v$ for all $u, v \in V(G)$. Otherwise, G is *disconnected*.

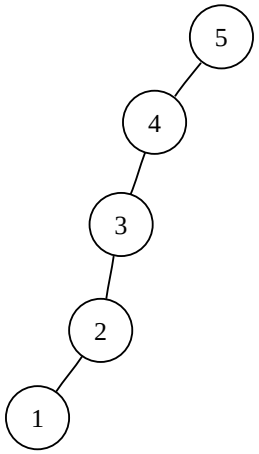


Figure 6.3: A third example graph

In Figure 6.1 a graph is connected, because we can construct a path from any vertex to any other vertex. Whereas a graph in Figure 6.4 is disconnected, because there is no, for example, 2, 3-path. A maximal path for this example graph is, for example, $[1, 8, 2, 4, 7]$

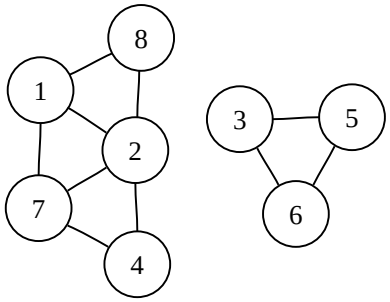


Figure 6.4: An fourth example graph

Next we define a maximal connected subgraph, components and a decomposition of a graph.

Definition 6.11. A *maximal connected subgraph* of a graph G is a subgraph that is connected and is not contained in any other connected subgraph of G. *Components* of G are its maximal connected subgraphs. A *decomposition* of a graph is a list of subgraphs such that each edge appears in exactly one subgraph in the list.

In this thesis we mainly need graphs, which have only one component. We say that a list of subgraphs decomposes a graph, if the list is a decomposition of the graph. The example graph in Figure 6.4 has 2 components.

If we want to consider the deletion of vertices, we need the definition of connectivity and k-connected.

Definition 6.12. A *connectivity* of G is the minimum size of a vertex set S such that $G\backslash S$ is disconnected or has only one vertex. The connectivity is written by $\kappa(G)$. A graph G is k-connected if its connectivity is at least k.

We need also some special graphs and subgraphs: an acyclic graph, a tree, a spanning subgraph, and a spanning tree.

Definition 6.13. A graph with no cycle is *acyclic*. A *tree* is a connected acyclic graph. A *leaf* is a vertex of degree 1. A *spanning subgraph* of G is a subgraph with vertex set $V(G)$. A *spanning tree* is a spanning subgraph that is a tree.

Note that if a graph has n vertices, then every its spanning tree has $n-1$ edges. The example graph in Figure 6.3 is a tree. It has two leafs: the vertices 1 and 5.

The following definition gives a special case of a tree.

Definition 6.14. A *rooted tree* is a tree G with one vertex s chosen as a *root*. For each vertex $v \in V(G)$, let $P(v)$ be the unique v, s-path on the rooted tree. The *parent* or the *predecessor* of v is its neighbor on $P(v)$; its *children* are its other neighbors. Its *ancestors* are the vertices of $P(v)\backslash v$. Its *descendants* are the vertices u such that $P(u)$ contains v. *Leaves* are the vertices with no children.

The following definition needs the definition of the spanning tree. It have only been defined for weighted graphs.

Definition 6.15 (MST)**.** The minimum spanning tree (MST) is a spanning tree, which minimizes the sum of its edge weights.

Now we will consider how we can represent a graph on a computer. Two standard ways to do it are a collection of adjacency list, and an adjacency matrix.

Definition 6.16. The *adjacency-list representation* of a graph G consists of an array A of $|V(G)|$ lists, one for each vertex in $V(G)$. For each $u \in V(G)$, the adjacency list $A(u)$ contains all the vertices v such that there is an edge $e = uv \in E(G)$. That is, $A(u)$ consists of all the vertices adjacent to $u \in V(G)$. The vertices in each adjacency list are typically in an arbitrary order.

Definition 6.17. For the *adjacency-matrix representation* of a graph G, assume that vertices have been numbered $1, 2, \ldots, |V(G)|$ in some arbitrary manner. Then the adjacency-matrix representation consists of a $|V(G)| \times |V(G)|$ matrix $\mathbf{A} = (a_{ij})$ such that a_{ij} is the number of edges from i to j.

For example, the adjacency-matrix representation for the example graph in Figure 6.1 is the following:

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Communications between Maxima and modSTACK can only be simple, which means that data can only be single numbers, lists, or matrices. Thus it is not possible, for example, to send graphs of Maxima from modSTACK to Maxima. Maxima does not recognize walks, or cycles neither. We need to present all of them by normal square bracketed Maxima lists. Moreover, note that the earlier presented adjacency-matrix representation for the example graph in Figure 6.1 loses data, it does not include weights of the graph. Because our exercises only use simple graphs, we can use a new way to move graphs between modSTACK and Maxima. In this thesis this new way is called a distance-matrix representation. It is not a standard way to present weighted, and simple graphs, however.

Definition 6.18. Let us assume that G is a simple, weighted graph, and the vertices of this graph are numbered $1, 2, \ldots, |V(G)|$ in some arbitrary manner. Then a *distance-matrix representation* of the graph G consists of a $|V(G)| \times |V(G)|$ matrix $\mathbf{A} = (a_{ij})$ such that, if the edge $(i, j), i \neq j$ exists, then a_{ij} is its weight, otherwise it is ∞. The diagonal elements of $\mathbf{A}$ are zeros.

Note that in the distance-matrix representation, 0 elsewhere than in the diagonal of the matrix means the 0-weighted edge. We have defined that all diagonal elements are 0, because then we can use our algorithms from Section 6.4 to this representation easily. The distance-matrix representation

of the example graph in Figure 6.1 is the following:

$$\begin{pmatrix} 0 & 10 & 30 & 60 & 20 & \infty & \infty \\ 10 & 0 & 70 & 90 & \infty & \infty & 10 \\ 30 & 70 & 0 & \infty & 20 & 60 & 100 \\ 60 & 90 & \infty & 0 & 10 & \infty & \infty \\ 20 & \infty & 20 & 10 & 0 & 40 & \infty \\ \infty & \infty & 60 & \infty & 40 & 0 & 70 \\ \infty & 10 & 100 & \infty & \infty & 70 & 0 \end{pmatrix}.$$

Note that now this matrix includes all information about the example graph.

The following definition is here, because we need it later. The definition was born, when we implemented our algorithms. They often construct so-called predecessor lists to handle trees, especially rooted trees. ModSTACK does not allow students to draw their own graphs. That is why exercises request to give the answer tree by this predecessor list. Moreover, the development trend of modSTACK do not probably advance to this direction, because the valid system, which is called TRAKLA [15], already exists. Note that the viewpoints of TRAKLA and modSTACK are different, TRAKLA considers graphs and algorithms from the computer science viewpoint, whereas modSTACK considers them from the mathematical viewpoint.

Definition 6.19. Let G be a rooted tree, and let the vertices of this graph be labelled by $1, 2, \ldots, |V(G)|$ in some arbitrary manner. A *predecessor list* of this tree is a list, which have square brackets, and whose i:th element is the predecessor of the i node in the tree. It are often denoted by π. The predecessor of the root of the tree is marked by 0.

Note that in our example exercises walks, trails, cycles, paths, and predecessor lists are all marked by square brackets, because it is only way to present lists in Maxima.

Let us consider these predecessor lists. If an exercise requests a rooted tree from a student by this predecessor list, we have to be sure that the given list produces a rooted tree. Note the following matters: There is one component for each 0 in the predecessor list. Moreover, it is possible to construct a predecessor list, which have only one 0, but two or more components. The following theorem proves, what we need to check to guarantee the correctness of student's predecessor list.

Theorem 6.1. *Let us assume that we have an arbitrary predecessor list π. Let the graph generated from π be G_π. Then G_π is a rooted tree if and only if G_π contains exactly one component and π contains exactly one* 0.

Proof. Let us assume that G_π is a rooted tree. Then it has a unique source vertex s. Then the s:th element in π is 0. No other zeros exists. If G_π is a rooted tree, then it has only one component.

On the contrary, let us assume that G_π contains exactly one component and π contains exactly one zero. Then we know that G_π is connected by Definition 6.11. We need to show that G_π is acyclic. Let us assume that one cycle C exists in this G_π. Let us list the vertices of this cycle by $(v_0, v_1, v_2, \ldots, v_k)$.

Let us assume that the vertex s, whose predecessor is 0, is on this cycle. Because degrees of the vertices on the cycle are at least 2, there exist at least two vertices v_k and v_l, whose predecessor is s. Then there can be vertices, whose predecessors are v_k or v_l, and so on. But it is not possible to unite these branches, because every vertex has only one predecessor. So the source vertex s cannot belong to C.

Let us take an arbitrary vertex v_i from the cycle C. Then the predecessor of this v_i is either v_{i-1} or v_{i+1}. Without loss of generalization we can assume that the predecessor of v_i is v_{i-1}. Then the predecessor of v_{i-1} is v_{i-2} and so on. Because C is a cycle, we get finally that predecessor of v_{i+1} is v_i. Then the graph G_π does not have a s, v_i path, so it has at least 2 components, which is a contradiction.

Thus G_π is acyclic. So, it is a tree. The root of this G_π is the vertex, whose predecessor is 0. □

The following two definitions and Fáry's theorem are needed in our randomized graphs.

Definition 6.20. A *planar graph* is a graph that can be drawn in the plane so that its edges are curves of the plane, its vertices are points of the plane, and if two edges intersect, they do so only at a common endpoint. A graph is said to be *planar* if it is a planar graph. A *plane graph* is a graph that is already drawn in such a fashion. A plane graph separates its complement, i.e. the rest of the plane, into regions, which are called *faces*.

The following theorem says that planar graphs can always be drawn such that all edges are line segments. The proof of the theorem can be found, for example, from I. Fáry's article [9].

Theorem 6.2 (Fáry's theorem)**.** *All planar graphs can be drawn such that all edges in the respective plane graphs are line segments.*

Definition 6.22 is needed in the construction of random graphs in Section 6.3. We need some basic definitions from the measure theory. More about metric spaces and metrics can be read, for example, from Walter Rudin's *Principles of Mathematical Analysis* [28].

Definition 6.21. Let (X, d) be a metric space. Assume that all points and sets mentioned next are understood to be elements and subsets of X. A neighborhood of p is a set $B(p, r)$ consisting of all q such that $d(p, q) < r$, for some $r > 0$. The number r is called the radius of $B(p, r)$.

Definition 6.22. The unique unbounded face of a plane graph is called the *exterior face*, and all other faces are called interior faces. The boundary of the exterior face is called the boundary of the graph. Let us take an arbitrary edge e and a point p, such that the line segment representing e goes through the point p and p is not the endpoint of this line segment. Then the edge e is called a *boundary edge* if a neighborhood $B(p, r)$ includes points of exterior and interior faces for all $r > 0$. Respectively, the edge e is called an *interior edge* if a neighborhood $B(p, r)$ includes only points of interior faces for some fixed R such that $0 < r < R$.

The following definition is here, because we can write an interesting exercise about it.

Definition 6.23. A graph is *Eulerian* if it has a closed trail containing all edges. We call a closed trail *a circuit* when we do not specify the first vertex but keep the list in cyclic order. An *Eulerian circuit* or *Eulerian trail* in a graph is a circuit or a trail containing all the edges. An *even graph* is a graph with vertex degrees all even. A vertex is *odd (even)* when its degree is odd (even).

Note that a graph has a circuit means the same as it has a cycle. We present also some results about these Eulerian graphs, because we need them in our exercise. The following lemma is needed in the proof of Theorem 6.3.

Lemma 6.1. If every vertex of a finite graph G has degree at least 2, then G contains a cycle.

Proof. Let P be a maximal path in G, and let u be an endpoint of P. Since P cannot be extended, every neighbor of u must already be a vertex of P. Since u has degree at least 2, it has a neighbor v in $V(P)$ via an edge not in P. The edge (u, v) completes a cycle with the portion of P from v to u. □

The following theorem states, when a graph is Eulerian.

Theorem 6.3. *A graph G is Eulerian if and only if it has at most one nontrivial component and its vertices all have even degree.*

Proof. Suppose that G has a closed trail $T = v_0, e_1, v_1, \ldots, e_k, v_0$, where each edge $e_i \in E(G)$ exists once only. Because there is one edge before, and one after each vertex, every vertex has even degree. Note that the edges beside the vertex v_0 are e_k and e_1. In addition to, two edges can be in the same trail only when they lie in the same component. Thus the graph has at most one nontrivial component.

Conversely, suppose that G has one nontrivial component, its degree of all vertices is even, and it has m vertices. We prove this by the strong induction principle (Theorem 7.2).

The induction basis: $m = 0$. A closed trail consisting of one vertex suffices.

The induction step: Let us assume that the statement is true for all $m - 1 \geq 0$. We show that the statement is true for m. With even degrees, each vertex in the nontrivial component of G has degree at least 2. By Lemma 6.1, the nontrivial component has a cycle C. Let G' be the graph obtained from G by deleting $E(C)$. Since C has 0 or 2 edges at each vertex of G, each component of G' is also an even graph. By using the induction hypothesis, we know that since each of the component of G' has fewer than m edges, they all have an Eulerian circuit. We form the Eulerian circuit to G by traversing C. When a component of G' is encounter for the first time we circle along an Eulerian circuit of that component. This circuit ends at the vertex where we began the circle. When we have traversed the C, we have gotten round to an Eulerian circuit of G. □

We also need the following theorem to guarantee, that a graph G has one or more Eulerian trails.

Theorem 6.4. *If a connected nontrivial graph G has exactly $2k$ odd vertices, then the minimum number of trails that decompose it is* $\max\{k, 1\}$.

Proof. A trail contributes even degree to every vertex, except that a non-closed trail contributes odd degree to its endpoints. Hence, a partition of the edges into trails must have some non-closed trail ending at each odd vertex. Because each trail has only two ends, we must use at least k trails to satisfy $2k$ odd vertices. We also need at least one trail, because G has an edge, and Theorem 6.3 implies that one trail suffices when $k = 0$.

Now we need to prove that k trails suffice when $k > 0$. Given such a graph G, we pair up the odd vertices of G in arbitrary manner and form G' by adding for each pair an edge joining its two vertices. In other words, we add k edges to G. The resulting graph G' is still connected and even, and thus by Theorem 6.3 it has an Eulerian circuit C. Now we traverse C in G', and we start a new trail in G each time we traverse an edge of $G' \backslash E(G)$. We

get $2k$ trails, and k of them are these added edges. Thus k trails decompose G. □

With Theorem 6.4 we can formulate the following trivial corollary.

Corollary 6.1. A graph G has a non-closed Eulerian trail if and only if it has exactly 2 odd vertices.

Proof. Suppose that a graph G has a non-closed Eulerian trail

$$T = v_0, e_1, v_1 \ldots, e_k, v_k.$$

Then we know that this trail contributes even degree to every vertex, except endpoints. The degrees of endpoints are odd. There are only two endpoints, and thus G has exactly 2 odd vertices.

Conversely, suppose that G has exactly 2 odd vertices. Then by setting $k = 1$ in Theorem 6.4, we complete the proof. □

We need Theorem 6.3 and Corollary 6.1 in the creation of an example exercise in Section 6.4. The exercise requests an Eulerian Circuit and Trail from a given graph.

We write also exercises, which requests a Hamiltonian Circuit and Trail and a solution to a single-source shortest-paths problem. They are defined as follows.

Definition 6.24. A *Hamiltonian circuit* of a graph G is a closed trail containing all vertices of G. A graph is *Hamiltonian* if it has a Hamiltonian circuit. A *Hamiltonian trail* in a graph is a trail containing all the vertices.

Definition 6.25. A *shortest-paths problem* is a problem, which we have a weighted, directed graph G, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The weight of path $p = v_0, v_1, \ldots, v_k$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

We define the shortest-path weight from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

In this thesis we shall focus on the single-source shortest-paths problem: given a graph G, we want to find shortest path from a given source vertex $s \in V(G)$ to each vertex $v \in V(G)$. Many other problems can be solved by the algorithm for the single-source problem, as a so-called single-pair shortest-path problem. It asks the shortest path from u to v for given vertices u and v. Actually our exercises are practically single-pair shortest-path problems, but according to Cormen et al. no algorithm for this problem is known that runs asymptotically faster than the best single-source algorithms in the worst case [5, p.581].

6.2 Drawing Graphs

In this section we will consider problems of drawing graphs. Generally speaking, the graph drawing is a whole discipline. We could present a whole book about drawing planar graphs only, but it is not the goal of this thesis. If the reader want to familiarize with it, we recommend Takao Nishizeki and Md. Saidur Rahman's *Planar Graph Drawing* [24].

As we have mentioned in Section 2.2, our selected program is Graphviz, and its subprogram NEATO. This program is only a temporary solution, because it does not always draw planar graphs correctly. The program is not even able to draw planar graphs correctly, if it is known that the graph is planar. There seems to be no elegant open source software to draw these planar graphs correctly in every case, and therefore we have to be satisfied with this NEATO.

The DOT code, which draws the example graph in Figure 6.1 is the following:

```
graph G {
  node [height,width=.1,.1, shape=box]
  edge [fontcolor=red, len = 1.5, fontsize= 10]
  1 -- 2 [label=10];
  1 -- 3 [label=50];
  1 -- 4 [label=60];
  1 -- 5 [label=20];
  2 -- 3 [label=70];
  2 -- 4 [label=90];
  2 -- 7 [label=10];
  3 -- 5 [label=20];
  3 -- 6 [label=60];
  3 -- 7 [label=100];
  4 -- 5 [label=10];
  5 -- 6 [label=40];
  6 -- 7 [label=70];
}
```

Let us look at the codes, which draw random graphs in modSTACK. We write the following commands for the question variable field.

```
nONodes = 7
adjac = jrCreateRandomConnectedGraph(nONodes)
graphicsAdjac=jrCreateBooleanAdjacMatrix(adjac,nONodes)
nodes = create_list(i,i,1,nONodes)
nmatrix = jrWeightTheGraph(adjac)
graphicsMatrix = jrCreateNodeMatrix(nmatrix,nONodes)
```

The variable nONodes is the number of nodes in the graph. *jrCreateRandomConnectedGraph* constructs random, connected graphs. It returns an adjacency-matrix representation of the graph. We will consider the algorithm beneath this function in Section 6.3. *jrCreateBooleanAdjacMatrix* returns a Maxima list, whose elements are Maxima lists. Each element corresponds each rows of the adjac matrix, such that there is true, if the element of the row is 1, and false, if the element of the row is 0. Technically, the returned graphicsAdjac is not same as a Maxima matrix. We need this graphicsAdjac in our drawing code in the question stem field, and it cannot be a normal matrix.

The graphicsAdjac for the example graph in Figure 6.1 is the following

$$\begin{array}{llllllll} [& [false, & true, & true, & true, & true, & false, & false\ ], \\ & [true, & false, & true, & true, & false, & false, & true\ ], \\ & [true, & true, & false, & false, & true, & true, & true\ ], \\ & [true, & true, & false, & false, & true, & false, & false\ ], \\ & [true, & false, & true, & true, & false, & true, & false\ ], \\ & [false, & false, & true, & false, & true, & false, & true\ ], \\ & [false, & true, & true, & false, & false, & true, & false\ ]]. \end{array}$$

The variable nodes is a list, which include all nodes, and in our example it is $[1, 2, 3, 4, 5, 6, 7]$. The function *jrWeightTheGraph* constructs and returns a distance-matrix representation of the adjac matrix. The weight of edges have been chosen at random. The function *jrCreateNodeMatrix* returns nmatrix in the same notation as the graphicsAdjac, i.e. a list, which includes lists.

The question stem field includes the following commands:

```
|$ begin dot neato -Gepsilon=.0001
 -Gsplines=true -Goverlap=scale $|
graph G {
  node [height,width=.1,.1, shape=box]
  edge [fontsize=8,fontcolor=red,len=1.2]
  |$ begin for _vect_ in @#graphicsMatrix#@ _begin_
   in @#nodes#@ _adjacVect_ in @#graphicsAdjac#@ $|
    |$ begin for _condition_ in _adjacVect_
```

```
      _end_ in @#nodes#@ _dist_ in _vect_ $|

       |$ begin if _condition_ $|
        _begin_ -- _end_ [label= _dist_];
       |$ end if $|

     |$ end for $|
   |$ end for $|
}
|$ end dot $|.
```

Note how the code includes nested for loops and how there are a couple of indices inside the for loops. Inside the first for loop are three indices: *_vect_*, *_begin_*, and three index sets: @#*graphicsMatrix*#@, @#*nodes*#@, and @#*graphicsAdjac*#@, respectively. The sizes of the index set are same for all index sets, and the indices get the values at the same time, which means that, for example, every index in the first for loop gets the first value of the corresponding index set at the same time. Then everyone gets the second value, and so on.

Finally, every graph in the figures of this chapter have been drawn by Graphviz. We have tried to select graphs such that they would show the good and bad sides of Graphviz. We will comment on the graphs if needed, as they are shown.

6.3 Generation of a Random Graph

If we want to write randomized, modSTACK exercises from graph theory, we need to study random graph theory. The student's assignment is not to work out the graph, but solve the asked problem. That is why we want to use small, planar, and connected graphs. By a small graph we mean a graph, which has about ten nodes.

So our key question is: Can we construct this kind of small, planar, and connected graphs uniformly at random? An elegant solution to this question would be an algorithm, which constructs a graph G with n vertices uniformly at random. The algorithm would return the graph by using the adjacency-matrix representation. We found one algorithm, which gave promising results. The algorithm can construct connected, planar graphs uniformly at random [3, 33, 34]. However, we had not enough time to implement this algorithm yet, and thus further research is needed.

We also tested a worse algorithm, which uses so-called Delaunay triangulation. We chose it, because it is easy to implement and it always creates planar and connected graphs. However it is not a final solution to our key

question, as we will see.

The Delaunay Triangulation

The algorithm which constructs so-called Delaunay triangulations was chosen to create randomized graphs to the exercises, because it is easy to implement and it constructs always a connected planar graph. The algorithm is not efficient, but the number of vertices in our exercise is small, and thus the efficiency is not important.

It has also restrictions and we will analyze them in this section. There is also a very small probability, that our algorithm does not work correctly. We will also consider this issue in this section. We will begin by defining the classical Euclidean distance in $\mathbb{R}^2$.

Definition 6.26. The Euclidean distance between points $\mathbf{x} = (x_1, x_2)$,$\mathbf{y} = (y_1, y_2) \in \mathbb{R}^2$ is

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}.$$

Next we define collinearity, a triangulation, a circumcircle of a triangle, a Voronoi diagram, and a Delaunay triangulation.

Definition 6.27. Let us assume that we have three or more points in the space $\mathbb{R}^2$. If all of them lie on the same line, then they are *collinear*.

Definition 6.28. A 2-connected plane graph G is called *a triangulation* if every interior face is bounded by a triangle and the boundary of G is a convex polygon. *A circumcircle* is a circle which passes through all the vertices of the triangle.

Definition 6.29. Let S be a set of distinguished points in $\mathbb{R}^2$ such that not all points are collinear. For each $\mathbf{s} \in S$, *the Voronoi region* generated by $\mathbf{s}$ is the set of points closer to $\mathbf{s}$ than to any other point in S. The collection of all Voronoi regions generated by points of S is called *the Voronoi diagram* generated by S. *The Voronoi dual* of S is defined to be the straight-line geometric dual ([11]) of the Voronoi diagram. If no more than three Voronoi regions meet at any point in the Voronoi diagram generated by S, then the Voronoi dual of S is a triangulation, called *the Delaunay triangulation* of S. In this case the Delaunay triangulation is said to be *nondegenerate*. If the Voronoi dual is *degenerate*, which means that more than three Voronoi regions meet at any point, then a Delaunay triangulation is any triangulation obtained by adding edges to the Voronoi dual.

Next, we define what is a circumcentre.

Definition 6.30. The *circumcentre* of the vertices of triangle is a point, which is equidistant from each vertices. Hence, it is the midpoint of the circumcircle.

The next trivial theorem computes the circumcentre of the vertices of a triangle.

Theorem 6.5. *If the vertices of triangle are* $\mathbf{x} = (x_1, x_2)$, $\mathbf{y} = (y_1, y_2)$ *and* $\mathbf{z} = (z_1, z_2)$, *then the coordinates of the circumcentre are given by*

$$\begin{cases} c_1 &= \frac{y_2 z_1^2 - (y_1^2 + y_2^2) z_2 + y_2 z_2^2 + x_1^2(-y_2 + z_2) + x_2^2(-y_2 + z_2) + x_2(y_1^2 + y_2^2 - z_1^2 - z_2^2)}{2(x_2(y_1 - z_1) + y_2 z_1 - y_1 z_2 + x_1(-y_2 + z_2))} \\ c_2 &= \frac{x_1^2(y_1 - z_1) + x_2^2(y_1 - z_1) + z_1(y_1^2 + y_2^2 - y_1 z_1) - y_1 z_2^2 + x_1(-y_1^2 - y_2^2 + z_1^2 + z_2^2)}{2(x_2(y_1 - z_1) + y_2 z_1 - y_1 z_2 + x_1(-y_2 + z_2))} \end{cases}.$$

Now we can state an algorithm, which constructs the Delaunay triangulation, from which a random graph is constructed. The algorithm contains two technical restrictions and we will consider these restrictions later.

Algorithm 6.1. The algorithm chooses n points randomly from the unit square $[0, 1] \times [0, 1]$, and it connects an ordinal number to each of them. The first randomly chosen point gets a number one, the second gets a number two, and so on. These chosen points will constitute the vertices of a graph G, and the ordinal numbers will be the names of the vertices. The algorithm calculates the circumcentre for each thee points of $V(G)$. Then, for each circumcentre x, all the points in $V(G)$ are checked to make sure that the three points used to find x are the three closest points of $V(G)$ to x. If they are not, x is rejected from consideration. Then, for each circumcentre x not rejected, the algorithm draws lines between the points that were used to find this circumcentre. These lines will constitute the edges of the graph G. Finally the algorithm constructs an adjacency-matrix representation from these vertices and edges.

Let us consider our algorithm more closely. If the reader is not familiar with the O-notation, see, for example, from Cormen et al.'s book [5]. The time needed for the calculation of the circumcentres is $O(n^3)$, the time needed for the rejection of the circumcentres is $O(n^4)$, and the time needed for the creation of the adjacency-matrix representation is $O(n)$. Hence, the time needed for the algorithm is $O(n^4)$.

This result is not good, but still the algorithm is sufficiently fast, because the number of vertices in our graph exercises is small, typically 10 or less. D.T. Lee and B.J. Scachter represents a divide-and-conquer algorithm in their article [18], which runs in $O(n \log n)$ time. However, it is much more complicated, and our tests in the C courses shows that our algorithm is sufficiently fast.

Let us consider two restrictions in our algorithm. First, if we choose three points uniformly at random from the unit square, then the probability that we get two equal points or three points, which are on the same line, is zero [13]. However, in the real life we cannot choose real numbers uniformly at random, but only approximated real numbers, which means that they have only finite digits in the decimal representation. That is why it is possible to get the same point two or more times, or three or more points from the same line. Second, based on the previous discussion, it is also possible to get four points from the same circle.

In the second case, the Delaunay triangulation will be degenerate, and the algorithm constructs four different three-membered combinations, all of which have the same circumcentre. However, our algorithm does not work like in Definition 6.29, because the algorithm does not reject any of these circumcentres, and the resulting graph is not a triangulation and it may be a planar graph neither. We do not check this error, however, because the probability that it happens is very small. Moreover, the check of this error needs a lot of CPU time, and the algorithm, which draw our graphs is neither always able to draw planar graphs correctly 6.2.

The first restriction is solved by the following theorem. If we choose three points from the unit square, we get always one of the following four different situations:

1. Three different points, do not belong to the same line.
2. Three different points, belong to the same line.
3. Two different points.
4. One point.

Now we can formulate a theorem, with which we can handle the conditions 2,3, and 4.

Theorem 6.6. *If one of the three conditions 2,3 or 4 is true then the denominators* $2(x_2(y_1 - z_1) + y_2z_1 - y_1z_2 + x_1(-y_2 + z_2))$ *of the both coordinates of the circumcentre of Theorem 6.5 are zero.*

Proof. Let us assume that the condition 2 is true. Then we can calculate a slope to the points or the points are on the line $y = k$.

If the slope exists then we can set that

$$\begin{cases} x_2 &= kx_1 + b, \\ y_2 &= ky_1 + b, \\ z_2 &= kz_1 + b. \end{cases}$$

Then we can calculate that

$$\begin{aligned}
&2(x_2(y_1 - z_1) + y_2z_1 - y_1z_2 + x_1(-y_2 + z_2)) \\
=&2((kx_1 + b)(y_1 - z_1) + (ky_1 + b)z_1 - y_1(kz_1 + b) + x_1(kz_1 + b - ky_1 - b) \\
=&0
\end{aligned}$$

If the points are on the line $y = k$ then we know that $x_1 = y_1 = z_1$ and

$$\begin{aligned}
&2(x_2(y_1 - z_1) + y_2z_1 - y_1z_2 + x_1(-y_2 + z_2)) \\
=&2(x_2(x_1 - x_1) + y_2x_1 - x_1z_2 + x_1(z_2 - y_2)) \\
=&0
\end{aligned}$$

Let us assume that the condition 3 is true. Then we have two different points, and we can assume, that $x_1 = y_1$ and $x_2 = y_2$.

$$\begin{aligned}
&2(y_2(y_1 - z_1) + y_2z_2 - y_1z_2 + y_1(z_2 - y_2)) \\
=&2(y_2y_1 - y_2z_1 + y_2z_1 - y_1z_2 + y_1z_2 - y_1y_2) \\
=&0
\end{aligned}$$

Let us assume that the condition 4 is true. Then all three point are same point, which means that $x_1 = y_1 = z_1$ and $x_2 = y_2 = z_2$.

$$2(y_2(y_1 - z_1) + y_2z_2 - y_1z_2 + y_1(z_2 - y_2)) = 0$$

□

By using Theorem 6.6 we can be sure that we almost always get a valid Delaunay Triangulation if we check that the denominator is always nonzero. Note that we do not know anything about the opposite direction of Theorem 6.6. Moreover, note that the word "almost" is only there because of the small possibility that four or more points are on the same circle.

Finally, note that Algorithm 6.1 returns an adjacency-matrix representation, and it does not include the coordinates of the vertices or the coordinates of the selected circumcentres. We know that a valid Delaunay triangulation always corresponds to a planar graph, but it is not clear that the graph is planar, when only the adjacency-matrix representation of the graph is considered. The Graphviz program is not able to draw planar graphs correctly, but it tries to minimize unnecessary cross connections, and thus it needs CPU time. Hence, the better way in this situation would be such that the modSTACK draws the points and lines, which Algorithm 6.1 constructs. Then the result graph is always planar, and the Graphviz program is not needed. We do not implement this method, however, because the Delaunay

triangulation is also a temporary solution in certain cases. For example, we know that the random graphs generated by the Delaunay triangulation are Hamiltonians with high probability [6]. Moreover, we conjecture that random graphs generated by the Delaunay triangulation are never Eulerians, and they always have a Hamiltonian trail and seldom an Eulerian trail.

6.4 Algorithms

In this section we will present some basic graph algorithms, and some exercises written from them. The algorithms are BFS, DFS, Dijkstra's algorithm, MST-PRIM, searchEulerianCircuit and searchEulerianTrail. In addition, we will present an exercise that requests a Hamiltonian circuit and a Hamiltonian trail. We call these six algorithms main algorithms in this section, because there are some auxiliary algorithms linked to each of them.

The algorithms are presented such that the operating principle can be understood. The exact Maxima code of the some algorithms is given in Appendix A. For the rest algorithms, a pseudocode about them can be found, for example, in Cormen et al.'s *Introduction to Algorithms* [5].

There exist theorems, which proves how each algorithm really solves the asked problem. We do not present them in this thesis, but the reference to the theorem will be given. Moreover, we do not calculate computation times for the algorithms. The algorithms are not necessary the fastest known implementation to each problem. If clearly faster algorithm exists, the reference to it is given.

All presented algorithms work on graphs, whose n vertices are labelled by numbers $\{1, 2, \ldots, n\}$. The algorithms can also work on the graphs, whose vertices are labelled in a different way, but in this thesis we will only use these numbers, because it is simpler. For example, we can ask: Is there an edge from the vertex 4 to the vertex 5 in graph G? The command $A[4, 5]$ will tell this, if the adjacency-matrix representation of the graph G is stored in the variable A.

In this section the word "list" always means a Maxima list. In Maxima they can be constructed, for example, by command $Q : [1, 4, 7]$. Then Q is a list, which includes elements 1,4, and 7. We can get the i:th element of the list Q by command $Q[i]$. Thus in our example $Q[2]$ returns 4. In this section we also use two operations of Maxima: The assign operator :, and the comparison operator =. More about these matters can be found, for example, from official web pages for Maxima [21].

We begin by defining some elementary data structures to maintain our data in the algorithms.

A queue is a set in which the element removed from the set is prespecified. In a queue, the deleted element is always the one that has been in the set for the longest time, i.e. the queue implements a first-in, first-out policy. In this thesis we implements the queue with the list Q, which can include at most n-1 elements, although its length is n. We need two operations for a queue: Enqueue, which inserts elements to the queue, and Dequeue, which deletes an element from the queue. When a queue is constructed, we need this list Q and two variables, a *head* and a *tail*. In the beginning the list is empty and $head = tail = 1$, which means that they point to the first element of the list Q. Moreover, if $head = tail$, the list is empty. We think that the list is in circular order, the next element for the element n is 1. Both operations, enqueue and dequeue, are very simple with these set-ups.

Algorithm 6.2 (Enqueue)**.** Enqueue inserts one element to the queue. It put the element to the *tail*:th position in the list Q. Then it set that *tail* points to the next element of the list. Remember the circular order.

Algorithm 6.3 (Dequeue)**.** Dequeue deletes one element from the list Q. It returns *head*:th element from the list Q and set that *head* points to the next element of the list.

The second elementary data structure is a priority queue. We do not need it in general level in this thesis. If the reader want to familiarize with it more closely, we recommend *Introduction to Algorithms* [5, p.138]. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. In this thesis we use a min-priority queue and the operation Extract-min. Priority queues also have other operations, but we do not need them here. The operation Extract-min returns the element of S whose key is the smallest one, and deletes the element from the set S. The operation is used to lists, although it is also usable for the elementary data structure called heap [5, p.127].

A priority queue needs two lists: Q, and d. The list Q includes elements in arbitrary order. General level they can be anything, but in this thesis they are only numbers. Let us assume that when the priority queue is constructed, the length of Q is n. Then the length of d has to be at least n. The list d includes keys for each element from the set $\{1, 2, \ldots, n\}$. Note that Q does not need to include all numbers from the set $\{1, 2, \ldots, n\}$, but if there is the number k in Q, then the key of k is k:th element in the list d.

Algorithm 6.4 (Extract-min)**.** The algorithm constructs two variables: *value*, and *min*. In the beginning, the value of *value* is ∞ and the value of *min* is -1. The algorithm goes through the list Q and updates the variables

value and *min*. If the key of $i \in Q$ is smaller than *value*, then *value* is the key of i and *min* is i. Finally the algorithm deletes *min* from the Q and returns it.

With this queue structure, we can present our first main algorithm, which is Breadth-first search (BFS). It is one of the simplest algorithm for searching a graph. By searching a graph we mean that we systematically follow the edges of the graph to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of the graph. We present two graph-searching algorithms. The first is Breadth-first search, and it explores the edges of a graph to find every vertex that is reachable from s. Moreover, it computes the distance from s to each reachable vertex. The distance is here the smallest number of edges between endpoints. BFS also produces a rooted tree with root s, which is called "breadth-first tree", that contains all reachable vertices. For every vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G.

Let us assume that there is a graph G and a distinguished source vertex s. Both of them are parameters of BFS.

Algorithm 6.5 (BFS)**.** First, the algorithm constructs one queue Q, and three lists: *color*, d and π. *color* is a list, whose v:th member tells the color of $v \in V(G)$, d is a list, whose i:th member tells the distance from the node i to the source vertex s, π is a predecessor list, like in Definition 6.19. In the beginning, the queue Q is empty, and every element of the *color* is white. They may later become gray and then black. Every element of the d is ∞, and every element of the π is 0.

The algorithm enqueues the source vertex s to the queue Q. Then it does the following so long, as there are elements in the queue Q: It dequeues one element u from the queue Q and search all adjacent vertices v to u, and if the color of the vertex v is white, then the algorithm colors it gray and set that $d[v] : d[u] + 1$, $\pi[v] : u$ and enqueues the vertex v to the queue. When every adjacent vertex to v have been searched, the algorithm colors the vertex u to black. Finally the algorithm returns a list, which includes two elements, π and d.

The theorem which states, that the BFS algorithm discovers every vertex $v \in V(G)$ that is reachable from the source vertex s, and computes the earlier defined distance from every vertex $v \in V(G)$ to s, is given with proof in *Introduction to Algorithms* [5, p. 535].

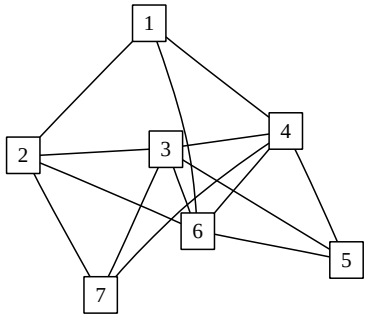
The exercise written from the breadth-first search algorithm is in Figure 6.5. The exercise requests the earlier constructed π and d. The correct

answer to this question is unique, because there is this clause: “Always choose the vertex, which has the smallest label.”

Note how the edge $(1, 6)$ is not a straight-line segment. We know by Fáry's theorem that all planar graph have a plane graph, whose edges are line segments. We also know that this graph is a planar graph, because it is generated by Algorithm 6.1. The Graphviz program is not always able to draw planar graphs correctly, and that is why we have corrected the code by allowing drawing edges with spline curves, although it is not necessary in the optimal cases. It is done by command $-Gsplines = true$ [25, p.11]. This command have also been presented in the question stem field code on page 40.

The breadth-first tree

Form the breadth-first tree (BFS) with root 5 about the following graph. Always choose the vertex, which has the smallest label.



Give your answer by using predecessor- and distance list. The nth element of the predecessor list is the predecessor of n. The nth element of the distance list is the distance between the n node and the root. Mark the 5th elements of both lists by 0.

The predecessor list is

The distance list is

Figure 6.5: The exercise which requests breadth-first tree

The second main algorithm is Depth-first search (DFS). It is also a graph-

searching algorithm, and it searches "deeper" in the graph whenever possible. The algorithm explores out of the most recently discovered vertex v that still has unexplored adjacent vertices. When all of v's edges have been explored, the algorithm "backtracks" to explore vertices adjacent to vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex s. If there are any undiscovered vertices, then the smallest of them is selected as a new source vertex, and the search is repeated from that vertex. The algorithm ends when all vertices have been explored.

This algorithm is slightly modified version about the algorithm introduced in *Introduction to Algorithms* [5, p. 540]. The only difference is, that this algorithm starts from the given source vertex s.

It may seem arbitrary that BFS is limited to only one source vertex whereas DFS may search from multiple source vertices. It is possible that BFS proceeds from multiple sources, and DFS is limited to only one source. However, this approach reflects how the results of these searches are typically used [5].

Let us assume that there is a graph G and a distinguished source vertex s, which are parameters of DFS.

Algorithm 6.6 (DFS)**.** In the beginning depth-first search colors each vertex white. The color of the vertex is stored in the list *color*. The vertices may later become first gray and then black. Moreover, it constructs two lists Q, and π. Q includes the vertices such that the first element is s, and the rest are in increasing order. π is a predecessor list, like in Definition 6.19, and in the beginning every element of π is 0. The algorithm goes through Q, and if the color in the same place in list *color* is white, then the algorithm visits in this node by using the DFSVisit algorithm. Finally, the algorithm returns the list π.

The following auxiliary algorithm uses same variables as DFS. It has one parameter, the visiting vertex v.

Algorithm 6.7 (DFSVisit)**.** The algorithm colors the visiting vertex v gray. Then it goes through all vertices of G such that if there are any white vertices u, adjacent to v, then the algorithm sets that predecessor of u is v and calls itself recursively with parameter u. Finally algorithm colors the visiting vertex v black, and returns nothing.

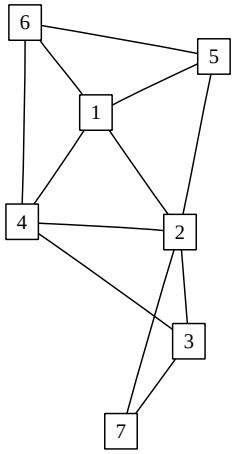
The predecessor of the source vertex s is 0. If there are two or more components in the graph, then there are two or more zeros in the predecessor list, too.

Biggs gives the theorem with proof, which states that the result π contains a spanning tree for all components of G in his book [2, p.203].

The exercise written from the depth-first search algorithm is in Figure 6.6. The exercise requests the earlier constructed π. The correct answer to this question is unique, because there is this clause: "Always choose the vertex, which has the smallest label." Note how there is an unnecessary edge intersection in the given graph.

The depth-first tree

Form the depth-first tree (DFS) with root 7 about the following graph. Always choose the vertex, which has the smallest label.



Give your answer by using a predecessor list. The nth element of the predecessor list is the predecessor of n. Mark the 7th element of the list by 0.

The predecessor list is

Figure 6.6: The exercise which requests depth-first tree

The next main algorithm is Dijkstra's algorithm, which solves the single source shortest-paths problem on a weighted graph G for the case in which all edge weights are nonnegative. The proof that the algorithm solves the problem is given in *Introduction to Algorithms* [5, p. 597].

Let us assume that the graph G has n vertices, and a set S includes the vertices whose final shortest-path weights from the source s have already been determined. In the beginning, the set S is empty. The algorithm repeatedly selects a vertex $u \in V(G)\backslash S$ which has a minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u by using Algorithm 6.9. It does this until $V(G)\backslash S$ is empty.

Note that the algorithm solves the single source shortest-paths problem, but our example exercise only requests the shortest path between two given vertices (remember the comments below Definition 6.25). Therefore our implementation returns a list, which includes two elements: the length of the shortest path between the given vertices, and a list, which includes this shortest path including the source vertex in the beginning and the goal vertex in the end.

The Dijkstra's algorithm needs three parameters, the examined graph by the distance-matrix representation G, the source vertex s and the goal vertex *goal*. Note that the notation $G[i, j]$ means the element in matrix G such that i is the row index, and j is the column index.

Algorithm 6.8 (Dijkstra's algorithm)**.** In the beginning, the algorithm constructs three lists: d, π and Q. d includes upper bounds on the weights of a shortest path from source vertex s to all vertices of G and these upper bounds are called shortest-path estimates. π is a predecessor list, like in Definition 6.19, and Q represent the set $V(G)\backslash S$. In the beginning, every element of d is ∞, every element of π is -1 and Q is $V(G)$. The algorithm begins by setting that $d[s] : 0$. After that the algorithm do n times the following things: It uses Algorithm 6.4 to get the smallest element u from Q, and goes through all vertices $j \in V(G)$ such that if $G[u, j] < \infty$, it uses Algorithm 6.9 to the vertices u and j.

Finally the algorithm constructs a list, whose first element is $d[goal]$ and the second element is the shortest path from s to *goal*. The second element is constructed from π.

The following auxiliary algorithm is needed in Dijkstra's algorithm. For the relaxation we mean technique, which maintains a predecessor list π, and a distance attribute $d[v]$ for each vertex $v \in V(G)$.

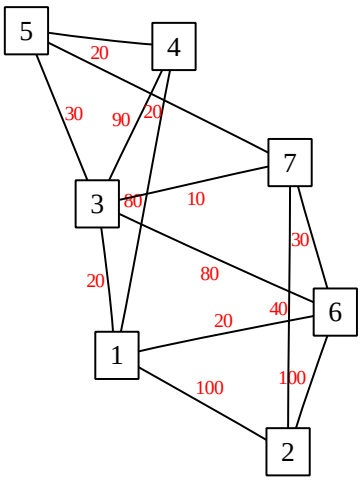
Algorithm 6.9 (Relaxation)**.** The Relaxation algorithm gets two parameters: u, and v. If $d[v] < d[u] + G[u, v]$ then the algorithm sets that $d[v] : d[u] + G[u, v]$, and $\pi[v] : u$. It returns nothing.

The exercise written from Dijkstra's algorithm is in Figure 6.7. Note that the exercise requests just the elements returned by Dijkstra's algorithm. The

given graph is admittedly a little bit ambiguous. What is the weight of the edge $(2, 7)$? 100? 40? 30? The Graphviz program always inserts the edge weights beside the midpoint of the edge, but still NEATO will not be enough.

Dijkstra's algorithm

Let us consider the following weighted graph.



Search the shortest route from the node 6 to the node 5 by using Dijkstra's algorithm. Give the length of the route and the route in form [6,12,11,5] such that the answer includes the endpoints of the route.

The length of the route is

The route is

Figure 6.7: The exercise about single source shortest-paths problem

The fourth main algorithm is a MST-PRIM algorithm. It constructs a minimum spanning tree from the given, weighted, connected graph. The algorithm constructs a rooted tree, which is also a minimum spanning tree, and the root of the tree is always 1. In theory, the root of the tree can be any vertex of the graph G.

MST-PRIM needs one parameter, the distance-matrix representation of the examined graph G. Let us assume that the graph has n vertices. Again,

the notation $G[i, j]$ means the element in matrix G such that i is the row index, and j is the column index.

Algorithm 6.10 (MST-PRIM)**.** In the beginning, the algorithm constructs three lists, Q, d, and π. Q includes all vertices that are not in the tree, and it forms the min-priority queue used in this algorithm. The $d[v]$ is the minimum weight of any edge connecting v to a vertex in the tree and the keys for the min-priority queue Q, π is a predecessor list, like in Definition 6.19. The minimum spanning tree T forms to this list π. In the beginning every element of d is ∞, every element of π is 0 and Q includes all vertices of G. First, the algorithm sets $d[1] : 0$. After that it does n times the following things: It uses Extract-min to get the vertex u from Q, and goes through all adjacent vertices v of u, which are in Q, such that if $G[u, v] < d[v]$ then $\pi[v] : u$ and $d[v] : G[u, v]$. Finally the algorithm computes the following sum

$$wT = \sum_{i=1}^{n} d[i],$$

which is the sum of the edge weights of the tree, and returns a list, which has two elements. The first element is this wT and the second is π.

The following two theorem proves the correctness of the MST-PRIM algorithm.

Theorem 6.7. *Let G be a connected, weighted graph, and suppose that T is a graph constructed from G by the MST-PRIM algorithm. Then T is a spanning tree.*

Proof. First we show that the algorithm produces a tree. It never chooses an edge that completes a cycle. If the result graph has more than one component, then no edge can join two of them, because such an edge would be accepted. Since G is connected, some such edge exists. Thus the final graph is connected and acyclic, which means that it is a tree by Definition 6.13.

The algorithm extract-mins n times the vertex i from the Q and adds it to the tree T. Thus every vertex is added to the T, which means that it is a spanning tree. □

In the following theorem $w(T)$ is the sum of the edge weights of the tree T. With Theorem 6.7 and Biggs's proof [2, p. 201] we have proved the following theorem.

Theorem 6.8. *Let G be a connected, weighted graph, and suppose that T is a graph constructed from G by the MST-PRIM algorithm. Then*

$$w(T) \leq w(U)$$

for any spanning tree U of G.

The exercise written from MST-PRIM is in Figure 6.8. It just requests the predecessor list written from the minimum spanning tree. Note that the predecessor list is defined for a rooted tree and a minimum spanning tree need not to be a rooted tree. However, we did not invent a handy way to present trees by single lists without determining the root of the tree. Of course it is possible to request adjacency-matrix representations, but students cannot construct them easily. Still, we have written the exercise such that it does not determine the root of the tree beforehand. Students can choose it themselves.

The graph in this exercise looks good. Note that there are 8 vertices in this graph. We have only changed value of the variable nONodes in the question variable field code on page 40.

The following exercise in Figure 6.9 requests to establish the Hamiltonian circuit or trail. The solutions of the exercise are easy to find because Maxima has functions: hamilton_path and hamilton_cycle, which find Hamiltonian trail and cycle, respectively. That is why we do not consider this exercise more closely. The graph in the exercises is an example about the Delaunay triangulation, which is not Hamiltonian. However, it has a Hamiltonian trail.

The latest two main algorithm in this section are searchEulerianCircuit and searchEulerianTrail. They are little bit complicated, and need many auxiliary algorithms. They are based on the algorithm presented by James R. Evans and Edward Minieka [8].

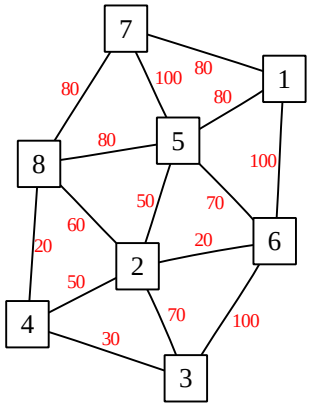
First, we define a couple of auxiliary algorithms. The first algorithm answers to two question: Has a graph G an Eulerian trail? Is a graph G Eulerian? It is based on Theorem 6.3 and Corollary 6.1. The algorithm gets one parameter: an adjacency-matrix representation of the examined graph G. Our implementation is not the most effective known algorithm for this problem. Its compute time is $O(|V(G)|^2)$, and there exists, for example, algorithm in S.A.M. Makki's article [19], whose compute time is $O(|V(G)|)$.

Algorithm 6.11 (hasEulerianTrailOrCircuit)**.** The algorithm computes the degrees of every vertices by computing sum of the every row in adjacency matrix. If there are exactly 2 odd vertices, the graph has an Eulerian trail. If there are 0 odd vertices, the graph is Eulerian, which have both an Eulerian trail, and an Eulerian circuit. Otherwise the graph has neither of them. The algorithm returns a list, which has 2 elements. The first element is true, if an Eulerian tour exists, otherwise it is false. The second element is true, if an Eulerian circuit exists, otherwise it is false.

The following 5 auxiliary algorithms are very easy to understand and that is why we present them very shortly.

MST-PRIM

Let us consider the following weighted graph.



Search the minimum spanning tree. Give the weight of the tree and the tree by a predecessor list. The nth element of the predecessor list is the predecessor of n. Mark the root of the tree in the list by 0.

The weight of the tree is

The predecessor list is

Figure 6.8: The exercise about minimum spanning tree.

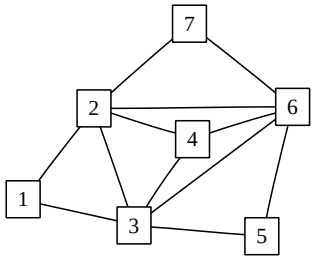
Algorithm 6.12 (ComputeDegree)**.** ComputeDegree gets two parameters, a graph $tMatrix$ in an adjacency-matrix representation and a vertex n. It returns the degree of the vertex n in the graph $tMatrix$.

Algorithm 6.13 (iZR)**.** The name of the algorithm is abbreviation for the name isZeroRow. The algorithm gets two parameters, a graph $tMatrix$ in an adjacency-matrix representation and a vertex n. It returns true, if the vertex n has no adjacent vertices, otherwise it returns false.

Algorithm 6.14 (sFO)**.** The name of the algorithm is abbreviation for the name searchFirstOne. The algorithm gets two parameters, a graph $tMatrix$ in an adjacency-matrix representation and a row number n. If $n = 0$, then it returns the coordinates of the first 1 in $tMatrix$. The search order is

Hamiltonian circuit and trail

Let us consider the following graph.



Is it possible to create a Hamiltonian circuit or trail to this graph? Mark your possible circuit or trail like $[3, 4, 5, \ldots, 10]$. If you find the circuit, put the first vertex at both ends of your answer list. If you are not able to find a circuit or a trail, you need not to put anything in the respective answer fields.

The graph has an Hamiltonian trail? ☐ Yes ☐ No

If your answer was yes, what is the trail? ☐

The graph has an Hamiltonian circuit? ☐ Yes ☐ No

If your answer was yes, what is the circuit? ☐

Figure 6.9: The exercise about Hamiltonian circuit and trail

$tMatrix[1,1], tMatrix[1,2], \ldots, tMatrix[2,1], \ldots$ If $n \neq 0$, then the algorithm only goes through the n:th row of the matrix $tMatrix$. If the suitable 1 does not exist, then the algorithm returns $[0, 0]$.

Algorithm 6.15 (iIPTM)**.** The name of the algorithm is abbreviation for the name isItPossibleToMove. The algorithm gets three parameters, a graph $tMatrix$ in an adjacency-matrix representation and two vertices, *begin* and *end*. If $tMatrix[begin, end] = 1$ then the algorithm returns *true*, else it returns $false$.

Algorithm 6.16 (isItZeroMatrix)**.** The algorithm gets one parameter, a graph $tMatrix$ in an adjacency-matrix representation. It returns $true$ if all elements of $tMatrix$ are 0, otherwise it returns $false$.

Algorithm 6.17 (searchBeginEnd)**.** The algorithm gets one parameter, a graph $tMatrix$ in an adjacency-matrix representation. It returns all vertices, whose degree is odd. The vertices are put in a normal Maxima list $[a, b]$.

Our two main algorithms are too long to explain verbal only. That is why the exact Maxima codes of the algorithms are given in Appendix A. The codes are given such that the standard functions of Maxima are green and our own functions are red. Here we will only give the operating principle of both algorithms. Note that searchEulerianCircuit assumes that a given graph G is Eulerian and searchEulerianTrail assumes that a given graph has an Eulerian trail. Thus it is very important to test these properties by hasEulerianTrailOrCircuit algorithm. If the graph G is not Eulerian, do not try to use searchEulerianCircuit to it. Both algorithm gets one parameter, the adjacency-matrix representation of the graph G.

Algorithm 6.18 (searchEulerianCircuit)**.** The algorithm advances in four steps:

1. The algorithm selects an arbitrary vertex as the source vertex s. From this vertex it constructs an arbitrary circuit C_0 by first visiting any edge (s, a) adjacent to vertex s. It marks the edge (s, a) as visited, and visits from a an edge which is not visited before. The algorithm repeats the process until the visit returns to s. The process will return to s because the vertices have even degree and each time a vertex is visited an even number of edges is used. Thus, every time the visit reaches a vertex except possibly the source vertex, there is an edge to exit from that vertex.

2. The algorithm goes to step 4 if the union of constructed circuits $\cup_{i=0}^{k} C_i$ contains all the edges of G. Otherwise the remaining subgraph $G_{k+1} = G \backslash E(\cup_{i=0}^{k} C_i)$ must be Eulerian, because each vertex of the circuits C_0, $C_1, \ldots, C_k$ contains an even number of edges. The subgraph G_{k+1} and one of $C_0, C_1, \ldots, C_k$ must have at least a common vertex v_{k+1}, since G is a connected graph.

3. The algorithm constructs a circuit C_{k+1} in the subgraph G_{k+1} from v_{k+1}. Then it returns to step 2.

4. Finally, the algorithm inserts each constructed circuit C_k after v_k in circuit $\cup_{i=0}^{k-1} C_i$.

The next algorithm searches the Eulerian trail for the given graph G. It is only a small modification to the algorithm searchEulerianCircuit. Note that the graph has exactly two odd vertices.

Algorithm 6.19 (searchEulerianTrail)**.** This algorithm differs from searchEulerianCircuit in that the first circuit is not a circuit but a trail. It begins from the one odd vertex and ends to the other odd vertex. Note that in the step 2, the possible remaining subgraph $G_1 = G \backslash E(C_0)$ is Eulerian, because each degree of vertices in $G_1 = G \backslash E(C_0)$ is even.

The following exercise in Figure 6.10 requests to establish the Eulerian circuit or trail.

6.5 Checking the Student's Answer

In general, quite often there is only one correct answer in mathematical exercises, and that is why it is relatively easy to check whether the student's answer is correct or not. In general, graph theory exercises have many correct answer, even numerable infinite, and it can sometimes be almost impossible to obtain all of them. Luckily, checking the correctness of a given answer is often easier.

In this section we will present algorithms, which will check student's answer in each of our example exercise in Section 6.4, if the correct answer is not unique. The solutions use the same Maxima notation as in the previous section.

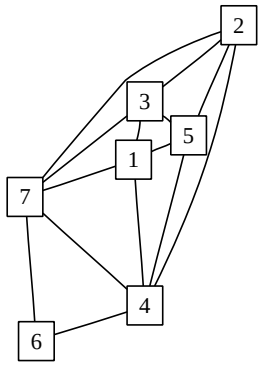
The first exercise whose correct answer is not unique is Dijkstra's algorithm exercise in Figure 6.7. There is a small probability that the given graph has two or more minimum routes from the given source vertex to the given end vertex, although the length of the routes are equal. Therefore the correct minimum route constructed by Dijkstra's algorithm is not necessarily unique. Nonetheless, the correctness of the student's minimum route is easy to check. Let us assume that G is the distance-matrix representation of the given graph, and the unique weight of the minimum route is w. The exercise computes the following sum

$$l = \sum_{i=1}^{k-1} G[a[i], a[i+1]],$$

where a is the student's minimum route, and k is the length of this a. If $l = w$ and student's minimum route begins from the source vertex and ends to the end vertex, it is correct.

Eulerian circuit and trail

Let us consider the following graph.



Is it possible to construct an Eulerian circuit or trail to this graph? Mark your possible circuit or trail as $[3, 4, 5, \ldots, 10]$. If you find the circuit, put the first vertex at both ends of your answer list. If you are not able to find a circuit or a trail, you need not to put anything in the respective answer fields.

The graph has an Eulerian trail? ☐ Yes ☐ No

If your answer was yes, what is the trail? ☐

The graph has an Eulerian circuit? ☐ Yes ☐ No

If your answer was yes, what is the circuit? ☐

Figure 6.10: The exercise about Eulerian circuit and trail

The second exercise whose correct answer is not unique is MST-PRIM algorithm exercise in Figure 6.8. In this exercise, the weight of the tree is unique, but the minimum spanning tree is not. Let us assume that G is the distance-matrix representation of the given graph, n is the number of vertices in the graph, the unique weight of the minimum spanning tree is w, and the

student's predecessor list is π. The exercise computes the following sum

$$l = \sum_{i=1,\pi[i]\neq 0}^{n} G[\pi[i], i],$$

the number of zeros nOZ in π, and the number of components nOC in the graph G_π generated from π. By Theorem 6.1 and the unambiguity of w we know that if $nOC = nOZ = 1$ and $l = w$, the student's minimum spanning tree is correct.

The Hamiltonian circuit and trail exercise in Figure 6.9 is also an exercise, which has many correct answers. The algorithm, which checks the correctnesses of students' answers, is given in Listing A.3 in Appendix A. The algorithm gets five parameters, the first parameter is the adjacency-matrix representation of the given graph G, and the rest four parameters corresponds the answer fields in Figure 6.9, respectively. The algorithm returns a list, which has two elements. The first element $isTrail$ tells something about the student's trail and the second element $isCirc$ tells something about the student's circuit. The value of $isTrail$ or $isCirc$ belongs to the set $\{1, 2, 3, 4, 5, 6, 7\}$. The meaning of each value is given in Table 6.1. Note that the word: route can be a Hamiltonian trail or Hamiltonian circuit. $isTrail$ cannot get the value 6, it is reserved for $isCirc$. Moreover, by Definitions 6.9, and 6.24, if the value is not 1, it always belongs to the set $\{2, 3, 4, 5, 6\}$, i.e. it is not possible to construct a wrong answer, for which no statement in Table 6.1 is true. The value 7 is a special value. Then student's answer is not list.

Value	Meaning
1	The route is correct
2	Some of the vertices does not belong to the route.
3	There are at least two times the same vertex.
4	The route contains an edge, which does not exist.
5	The route contains a vertex, which does not exist.
6	The first vertex and the last vertex of the circuit are not same.
7	The route is not in requested form.

Table 6.1: The meaning of the values of $isTrail$ and $isCirc$ in Hamiltonian circuit and trail exercise

The Eulerian circuit and trail exercise in Figure 6.10 is also an exercise, which has many correct answers. The algorithm, which checks the correctnesses of students' answers, is given in Listing A.4 in Appendix A. The

algorithm gets six parameters, the first parameter is the adjacency-matrix representation of the given graph G, and the next four parameters corresponds the answer fields in Figure 6.10, respectively. The last parameter is the number of edges in the graph. The algorithm returns a list, which has two elements, and they have the same meaning as in Hamiltonian exercise above, except the meaning of each possible value of both element is slightly different. The meaning of each value is given in Table 6.2. Again the word "route" can be an Eulerian trail or Eulerian circuit. Moreover, by Definitions 6.9, and 6.23, if the value is not 1, it belongs always to the set $\{2, 3, 4, 5\}$, i.e. it is not possible to construct a wrong answer, for which no statement in Table 6.2 is true. There is also a special value 6. The value of function is 6, if the student's answer is not list.

Let us suppose that a graph is Eulerian. Then a student cannot give an Eulerian circuit such that it does not begin from, and end in the same point; and includes all the edges at the same time. In this situation our algorithm returns 2 or 3

Value	Meaning
1	The route is correct.
2	Some of the edges does not belong to the route
3	There are at least two times the same edge
4	The route contains an edge, which does not exist.
5	The route contains a vertex, which does not exist.
6	The route is not in requested form.

Table 6.2: The meaning of the values of $isTrail$ and $isCirc$ in Eulerian circuit and trail exercise

Chapter 7

Mathematical Proofs and the Principle of Induction

In this chapter we study the possibility to use our system for formal proofs. In general, this area is very difficult. It is easy to say that it is impossible to write computer exercises from the mathematical proofs. We have a vision that in the future students can prove theorems selected by the teacher by computers. Moreover, a computer can give hints to the student, whose proof gets nowhere.

We succeeded in creating one type of proof exercise in modSTACK, and we give it here. The example exercise is only small introduction to this topic, but we believe that in the future there are more papers about this topic. Our example exercise involves the use of the Principle of Induction.

Theorem 7.1 (The Principle of Induction)**.** *Let $S(n)$ denote a mathematical statement (or set of such open statements) that involves one or more occurrences of the variable $n \in \mathbb{Z}^+$. If*

a) *$S(1)$ is true; and*

b) *if whenever $S(k)$ is true for some arbitrarily chosen $k \in \mathbb{Z}^+$, then $S(k+1)$ is true;*

then $S(n)$ is true for all $n \in \mathbb{Z}^+$. The item a) is called an induction basis. The condition in the item b) is called an induction hypothesis, and the proof, which shows the statement in the item b), is called an induction step.

Before we prove the principle, we state that it belongs to the set of three logically equivalent principles. The rest principles are Strong Induction, and the Well-Ordering Axiom.

Theorem 7.2 (Strong Induction)**.** *Let m be an integer and, for each $n \geq m$, let p_n be a statement. Suppose the following conditions are satisfied.*

1. *p_m is true.*

2. *If $k \geq m$ and all of p_m,p_{m+1},...,p_k are true, then p_{k+1} is also true.*

Then p_n is true for every $n \geq m$.

Definition 7.1 (The Well-Ordering Axiom)**.** Every nonempty subset of $\mathbb{Z}^+$ contains a smallest element. We express this by saying that $\mathbb{Z}^+$ is well ordered.

If we assume any of the three principles, we can prove all the others [23, p. 33]. Thus we can set the following statement:

$$\text{Well-Ordering} \Rightarrow \text{Induction} \Rightarrow \text{Strong Induction} \Rightarrow \text{Well-Ordering}.$$

However, we cannot prove any of the principles by using only basic axioms of the integers. We need to assume one of the principles, and it is the Well-Ordering Axiom in this thesis, because it is usually the selection. By using the Well-Ordering Axiom, we can prove the Principle of Induction, as we assumed above.

Proof. Let $S(n)$ be such an open statement satisfying conditions a) and b), and let $F = \{t \in \mathbb{Z}^+ | S(t) \text{ is false}\}$. We want to prove that $F = \emptyset$, and we do it by assuming that $F \neq \emptyset$ leads to a contradiction. Then by the Well-Ordering Axiom, F has a least element s. Since $S(1)$ is true, it follows that $s > 1$, and consequently $s - 1 \in \mathbb{Z}^+$. With $s - 1 \notin F$, we have that $S(s-1)$ is true. So by condition b) it follows that $S(s - 1 + 1) = S(s)$ is true, contradicting $s \in F$. Consequently, we know that $F = \emptyset$. □

The Principle of Induction is very useful tool, which is used in many situations. It is a good and easy method to practice formal proving. That is why it is important to practice its use. Moreover, it is possible to write exercises about this principle to modSTACK. One implementation can be seen in Figure 7.1. The symbols a and b in the example are randomized constants, which, for example, can get values from the set $\{x | x \in \mathbb{Z}, |x| \leq 10, x \neq 0\}$.

This example exercise was used during the fall 2008 and about 300 students did the exercise. We noticed the following two things:

First, we can use many answer fields, but it is not enough. At the present, students are not able to edit the number of the answer field in the induction

The Principle of Induction

Prove the following equation by using the Principle of Induction.

$$\sum_{r=1}^{n}(br+c) = \frac{n(bn+2c+b)}{2}$$

The induction basis
Put your answers to the form: left-hand side without simplification = simplified answer = right-hand side without simplification.

$$\square = \square = \square$$

The induction hypothesis

$$\sum_{r=1}^{n}(br+c) = \square$$

The induction step
Do not use $\sum$ nor ... in your answers.

$$\begin{aligned}\sum_{r=1}^{n+1}(br+c) &= \square \\ &= \square \\ &= \square \\ &= \square \\ &= \square\end{aligned}$$

Figure 7.1: The Exercise about the Principle of Induction

step, and this is a restriction, as we have mentioned in Chapter 5. For

example, we can calculate that

$$\begin{aligned}\sum_{r=1}^{n+1}(6r+7) &= \frac{n(6n+20)}{2} + 6(n+1) + 7 \\ &= \frac{n(6n+20) + 12(n+1) + 14}{2} \\ &= \frac{6n(n+1) + 12(n+1) + 14(n+1)}{2} \\ &= \frac{(n+1)(6n+26)}{2} \\ &= \frac{(n+1)(6(n+1)+20)}{2}.\end{aligned}$$

If we put the above calculation to the answer sheet of the exercise, we use all the answer fields of the induction step. What about if we want to use more fields, or we are so swift that we do not need all fields?

Second, some students used the variable k instead of n, because the model answers in manual exercises used k. As we can see from our answer sheet, this have been forbidden by using the ready formulas on the left-hand side of the answer fields. Thus we can think that students, which make this mistake, are unsophisticated. However, maybe this attitude is not the correct one.

There is always a possibility to replace k by n automatically. However, it is not a good idea, because mathematics and computer sciences do not include automatic operations in general. It is good to demand that students are accurate.

There is also a possibility to give feedback, which says that you cannot use the variable k. We think that this is the best way to react to this mistake. However, in that case the evaluation tree becomes larger and thus more difficult to administer. Consequently, we need a better evaluation tree again.

Chapter 8

Conclusions

The final impression after this thesis is an incompleteness of the topic. The automatic assessment at university level mathematics and writing of the exercises to this level are topics, which need a lot of further research. During the writing of this thesis we have seen that there are a lot of requirements in the universities for this kind of CAA system. These systems have developed much during the few past years. Still, they have restrictions, which need further work.

The writing of randomized exercises, which was our topic in this thesis, is in general difficult. The writing of an exercise can sometimes take very much time. Although the solution method for the exercise is trivial, the writing of the exercise can be almost impossible. Sometimes we encountered problems which are under research just now. In addition to, the present knowledge does not know some results, which are needed to write certain elegant and powerful randomized exercises to university level.

In this thesis we presented some topics of mathematics, and wrote some exercises about them. Selected three topics were calculus, graph theory, and mathematical proofs. Furthermore, this thesis presents some new abilities implemented to modSTACK by Matti Harjula.

The first topic was calculus. From this topic we presented the first exercises, which include intermediate steps. During the semester 2008-2009 we collected feedback from students, and in practical we observed that the only way to produce good exercises at university level is add intermediate steps to them. In general, these addings are not trivial, and carefulness is needed. The thesis also considered matters, which have to take account in these intermediate steps. The next two topics do not contain intermediate steps, and therefore further study is needed.

The second topic was graph theory. This section included two very difficult problem: the construction of random, connected, planar graphs, and

graph drawing. We presented one solution to the first problem, namely the Delaunay triangulation. This solution has severe restrictions, and that is why we also mentioned a good, but slower method to create these randomized graphs. The implementation of this algorithm left outside this thesis, it is a good further research topic.

For the solution to the second difficult problem Harjula plugged the Graphviz program into modSTACK. In this thesis, we presented the operational principle of this software in modSTACK. Graphviz is a freeware software, but it has limitations, and therefore a second software will be needed in the future.

The third topic was mathematical proofs. This section is also very difficult, and in this thesis we have only given a small introduction to it. We considered only the Principle of Induction, and wrote an exercise about it. Again, lot of further work is needed.

Bibliography

[1] Robert A. Adams. *Calculus, Complete Course*. Pearson Addison Wesley, 6 edition, 2006.

[2] Norman L. Biggs. *Discrete Mathematics*. Oxford University Press, 2 edition, 2002.

[3] Manuel Bodirsky, Gröpl Clemens, and Mihyun Kang. Generating labeled planar graphs uniformly at random. *Theoretical Computer Science*, 379(3):377–386, 2007.

[4] Béla Bollobás. *Random Graphs*. Cambridge University Press, 2 edition, 2001.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT press, 2 edition, 2001.

[6] Michael B. Dillencourt. Toughness and Delaunay Triangulations. *Discrete & Computational Geometry*, 5(1):575–601, 1990.

[7] Paul Erdös. Some remarks on the theory of graphs. *Bulletin of the American Mathematical Society*, 53(4):292–294, 1947.

[8] James R. Evans and Edward Minieka. *Optimization Algorithms for Networks and Graphs*. Marcel Dekker, Inc., 2 edition, 1992.

[9] I. Fáry. On straight Line Representation of Planar Graphs. *Acta Scientiarum Mathematicarum*, 11:229–233, 1948.

[10] Graphviz. `http://www.graphviz.org` (referenced August 27, 2009).

[11] Frank Harary. The Geometric Dual of a Graph. *Annals of the New York Academy of Sciences*, 555:216–219, 1989.

[12] Matti Harjula. *Mathematics Exercise System with Automatic Assessment*. Master's thesis, Helsinki University of Technology, 2008.

[13] Jean Jacod and Philip Protter. *Probability Essentials*. Springer, 2 edition, 2004.

[14] Svante Janson, Tomasz Luczak, and Andrzej Rucinski. *Random Graphs*. John Wiley & Sons, Inc., 1 edition, 2000.

[15] Ari Korhonen, Lauri Malmi, and Panu Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of Kolin Kolistelut /Koli Calling - Third Annual Baltic Conference on Computer Science Education*, pages 48–56, 2003.

[16] Erwin Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, Inc., 9 edition, 2006.

[17] David C. Lay. *Linear Algebra and Its Applications*. Pearson Addison Wesley, 3 edition, 2006.

[18] D.T. Lee and B.J. Scachter. Two algorithms for Constructing a Delaunay Triangulation. *International Journal of Computer and Information Sciences*, 9(3):219–242, 1980.

[19] S.A.M. Makki. A distributed algorithm for constructingf an eulerian tour. In *Performance, Computing, and Communications Conference*, pages 94–100. IEEE, feb 1997.

[20] Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. *Maple T.A.* `http://www.maplesoft.com/products/mapleta/index.aspx` (referenced August 27, 2009).

[21] The official web page for Maxima. `http://maxima.sourceforge.net/` (referenced August 27, 2009).

[22] Frank Mittelbach and Michel Goossens. *The LaTeX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, 2 edition, 2004.

[23] W. Keith Nicholson. *Introduction to Abstract Algebra*. John Wiley & Sons, Inc., 2 edition, 1999.

[24] Takao Nishizeki and Md. Saidur Rahman. *Planar Graph Drawing*, volume 12 of *Lecture Notes Series On Computing*. World Scientific Publsihing Company, Inc., 1 edition, 2004.

[25] Stephen North. Drawing Graphs with NEATO. Technical report, AT&T, apr 2004. http://www.graphviz.org/pdf/neatoguide.pdf (referenced August 27, 2009).

[26] Tri Quach. Stack ja opetuksen kehittäminen. Technical report, Helsinki University of Technology, Department of Automation and Systems Technology, 2008.

[27] Antti Rasila, Matti Harjula, and Kai Zenger. Automatic assessment of mathematics exercises: Experiences and future prospects. In *ReflekTori 2007 - Symposium of Engineering Education*, pages 70–80, Finland, dec 2007. TKK: Teaching and Learning Development Unit.

[28] Walter Rudin. *Principles of Mathematical Analysis*. McGraw-Hill Companies,Inc., 3 edition, 1976.

[29] Jarno Ruokokoski. Matematiikan verkkokokeet. Technical report, Helsinki University of Technology, Department of Mathematics and Systems Analysis, 2005.

[30] Christopher J. Sangwin. Assessing mathematics automatically using computer algebra and the Internet. *Teaching Mathematics and its Applications: An International Journal of the IMA*, 23(1):1–14, 2004.

[31] Christopher J. Sangwin. STACK: Making many fine judgements rapidly. In *CAME 2007 - The Fifth CAME Symposium*, Pécs, Hungary, June 2007. Computer Algebra in Mathematics Education.

[32] Christopher J. Sangwin and Michael Grove. STACK: addressing the needs of the "neglected learners". In *Proceedings of the WebAlt Conference*, pages 81–95, Eindhoven, Netherlands, January 2006. WebAlt.

[33] Gilles Schaeffer. *Conjugaison d'arbres et cartes combinatoires aléatoires*. PhD thesis, Université Bordeaux I, 1998.

[34] Gilles Schaeffer. Random Sampling of Large Planar Maps and Convex Polyhedra. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing (STOC' 99)*, pages 760–769. ACM Press, May 1999.

[35] STACK. http://www.stack.bham.ac.uk/ (referenced August 27, 2009).

[36] The official wiki for the STACK project. http://stack.bham.ac.uk/wiki/index.php/Main_Page (referenced August 27, 2009).

[37] Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, Inc., 2 edition, 2001.

[38] Juhana Yrjölä. Matematiikan tehtävien tarkastaminen STACK-järjestelmällä. Technical report, Helsinki University of Technology, Department of Information and Computer Science, 2007.

Appendix A

Maxima Codes

```
searchEulerianCircuit(G) := block([retVal:[],
                                   oner:[0,0],
                                   vertex,
                                   tMatrix,
                                   cycles:[],
                                   cycle,
                                   iFT:true
                                  ],
  tMatrix:copymatrix(G),
  for i:1 unless (isItZeroMatrix(tMatrix)) do (
    if iFT then (
      oner:sFO(tMatrix,0)
    ) else (
      for k:1 thru length(cycles) do (
        for l:1 thru (length(cycles[k])-1) do (
          if( not(iZR(tMatrix,cycles[k][l])) ) then (
            oner:sFO(tMatrix,cycles[k][l])
          )
        )
      )
    ),
    cycle:[],
    setelmx(0,first(oner),last(oner),tMatrix),
    setelmx(0,last(oner),first(oner),tMatrix),
    cycle:endcons(first(oner),cycle),
    cycle:endcons(last(oner),cycle),
    vertex:last(oner),
    for j:1 unless (vertex=first(oner)) do (
      if iIPTM(tMatrix,vertex,first(oner)) then (
        cycle:endcons(first(oner),cycle),
        setelmx(0,first(oner),vertex,tMatrix),
        setelmx(0,vertex,first(oner),tMatrix),
        vertex:first(oner)
```

```
34          ) else (
35              two:sFO(tMatrix,vertex),
36              cycle:endcons(last(two),cycle),
37              vertex:last(two),
38              setelmx(0,first(two),last(two),tMatrix),
39              setelmx(0,last(two),first(two),tMatrix)
40          )
41      ),
42      cycles:endcons(cycle,cycles),
43      iFT:false,
44      oner:[0,0]
45    ),
46    retVal:combineCycles(cycles,length(cycles)),
47    return(retVal)
48  )
```

Listing A.1: The source code of the algorithm, which constructs Eulerian Circuit for the given matrix

```
1   searchEulerianTrail(G) := block([retVal:[],
2                                    tMatrix,
3                                    oner:[0,0],
4                                    vertex,
5                                    cycles:[],
6                                    cycle,
7                                    iFT:true,
8                                    b,
9                                    e,
10                                   temp,
11                                   f
12                                  ],
13    tMatrix:copymatrix(G),
14    temp:searchBeginEnd(G),
15    b:first(temp),
16    e:last(temp),
17    for i:1 unless (isItZeroMatrix(tMatrix)) do (
18      if iFT then (
19        oner:sFO(tMatrix,b)
20      ) else (
21        for k:1 thru length(cycles) do (
22          for l:1 thru (length(cycles[k])-1) do (
23              if(not(iZR(tMatrix,cycles[k][l])))then(
24                oner:sFO(tMatrix,cycles[k][l])
25              )
26          )
27        )
28      ),
29      cycle:[],
30      setelmx(0,first(oner),last(oner),tMatrix),
```

```
31      setelmx(0,last(oner),first(oner),tMatrix),
32      cycle:endcons(first(oner),cycle),
33      cycle:endcons(last(oner),cycle),
34      vertex:last(oner),
35      f:first(oner),
36      for j:1 unless (vertex=f) do (
37        if (iFT and iIPTMove(tMatrix,vertex,e)) then (
38            cycle:endcons(e,cycle),
39            setelmx(0,e,vertex,tMatrix),
40            setelmx(0,vertex,e,tMatrix),
41            vertex:f
42        ) elseif(not(iFT) and iIPTMove(tMatrix,vertex,f))
43                                                then (
44            cycle:endcons(first(oner),cycle),
45            setelmx(0,first(oner),vertex,tMatrix),
46            setelmx(0,vertex,first(oner),tMatrix),
47            vertex:f
48        ) elseif(not(iFT) and iIPTMove(tMatrix,vertex,f))
49                                                then (
50            cycle:endcons(first(oner),cycle),
51            setelmx(0,first(oner),vertex,tMatrix),
52            setelmx(0,vertex,first(oner),tMatrix),
53            vertex:f
54        ) elseif (iFT and (last(oner)=e)) then (
55            vertex:f
56        ) else (
57            two:sFO(tMatrix,vertex),
58            cycle:endcons(last(two),cycle),
59            vertex:last(two),
60            setelmx(0,first(two),last(two),tMatrix),
61            setelmx(0,last(two),first(two),tMatrix)
62        )
63      ),
64      cycles:endcons(cycle,cycles),
65      iFT:false,
66      oner:[0,0]
67    ),
68    retVal:combineCycles(cycles,length(cycles)),
69    return(retVal)
70  )
```

Listing A.2: The source code of the algorithm, which constructs Eulerian Trail for the given matrix

```
1  checkHamiltonians(adjac,a1,a2,a3,a4):=block([isTrail,
2                                                isCirc,
3                                                tCounter,
4                                                cCounter,
5                                                nOVertices
```

```
6                                                          ],
7     nOVertices:length(adjac),
8     isTrail:1,
9     isCirc:1,
10    tCounter:makelist(0,i,1,nOVertices),
11    cCounter:makelist(0,i,1,nOVertices),
12    if a1 then (
13      if (listp(a2)= false) then isTrail:7
14      elseif lmax(a2) > nOVertices then isTrail:5  else (
15        for i:1 thru (length(a2)-1) do (
16          if (adjac[a2[i],a2[i+1]]=1) then
17          tCounter[a2[i]]:tCounter[a2[i]] + 1
18          else (isTrail:4,return(done))
19        ),
20        tCounter[last(a2)]:tCounter[last(a2)] + 1,
21        if (member(0,tCounter) and isTrail # 4)
22        then isTrail:2
23        elseif (lmax(tCounter)>1 and isTrail # 4)
24        then isTrail:3
25      )
26    ),
27    if a3 then (
28      if (listp(a4) = false) then isCirc:7
29      elseif (first(a4)#last(a4) ) then isCirc:6
30      elseif (lmax(a4) >nOVertices) then isCirc:5  else (
31        for i:1 thru (length(a4)-1) do (
32         if (adjac[a4[i],a4[i+1]]=1) then
33         cCounter[a4[i]]:cCounter[a4[i]] + 1
34         else (isCirc:4,return(done))
35        ),
36        if (member(0,cCounter) and isCirc # 4)
37        then isCirc:2
38        elseif (lmax(cCounter)>1 and isCirc # 4)
39        then isCirc:3
40      )
41    ),
42    return([isTrail,isCirc])
43  )
```

Listing A.3: The source code of the algorithm, which checks the correctnesses of student's Hamiltonian circuit and trail

```
1
2   checkEulerians(adjac,a1,a2,a3,a4,n):=block([isTrail:1,
3                                                isCirc:1,
4                                                tM1,
5                                                tM2,
6                                                temp1,
7                                                temp2,
```

```
8                                                    nOVertices
9                                                   ],
10    nOVertices:length(adjac),
11    tM1:zeromatrix(nOVertices,nOVertices),
12    tM2:zeromatrix(nOVertices,nOVertices),
13    if (a1) then (
14      if listp(a2) = false then isTrail:6
15      elseif lmax(a2) > nOVertices then isTrail:5
16      elseif length(a2) < n then isTrail:2
17      else (
18        for i:1 thru (n-1) do (
19          temp1:a2[i],
20          temp2:a2[i+1],
21          if (tM1[temp1,temp2] = 1) then (isTrail:3,
22                                          return(done))
23          else (
24              setelmx(1,temp1,temp2,tM1),
25              setelmx(1,temp2,temp1,tM1)
26          )
27        ),
28        if is(tM1 # adjac and isTrail # 3) then isTrail:4
29      )
30    ),
31    if (a3) then (
32      if listp(a4) = false then isCirc:6
33      elseif lmax(a4) > nOVertices then isCirc:5
34      elseif length(a4) < n then isCirc:2
35      else (
36        for i:1 thru (n-1) do (
37          temp1:a4[i],
38          temp2:a4[i+1],
39          if (tM2[temp1,temp2] = 1) then (isCirc:3,
40                                          return(done))
41          else(
42              setelmx(1,temp1,temp2,tM2),
43              setelmx(1,temp2,temp1,tM2)
44             )
45        ),
46        if is(tM2 # adjac and isCirc # 3) then isCirc:4
47      )
48    ),
49    return([isTrail,isCirc])
50  )
```

Listing A.4: The source code of the algorithm, which checks the correctnesses of student's Eulerian circuit and trail