MASTER'S THESIS

ESTIMATING STUDENT KNOWLEDGE LEVELS IN AN ITS FOR PROPOSITIONAL LOGIC

Joosse, C.W.

Award date:
2023

ESTIMATING STUDENT KNOWLEDGE LEVELS IN AN ITS FOR PROPOSITIONAL LOGIC

by

C.W. Joosse

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Science
Master Software Engineering
to be defended publicly on Tuesday June 20, 2023 at 2:00 PM.

Course code:	IMA0002	
Thesis committee:	dr. Bastiaan Heeren (chairman),	Open University
	dr. Josje Lodder (supervisor),	Open University

Contents

List of Figures

LIST OF TABLES

ABSTRACT

This thesis tests the use of a student model in an intelligent tutoring system (ITS) for propositional logic. ITSs are widely used in modern education programs and are applied for different purposes. Student models are used in ITSs to determine if a student masters the subject of the ITS or how a student solves an exercise. Often, the student models are used to generate exercises that are relevant for the student.

In this thesis, we model and implement a student model that can track the knowledge levels of students on propositional rewrite rules. We will base the student model on Bayesian Knowledge Tracking, but extend the basic structure of the classical Bayesian Knowledge Tracking model. The student model will classify the difficulty level of the exercises in the ITS based on the knowledge level of the individual students. We will gather feedback from the students on exercises in the ITS, build student models for each of the students, and compare the feedback with the classification of the student models.

We will show that the model works as intended, but that we need to refine the model to provide better classifications.

1

INTRODUCTION

The past two decades have seen an increase in the availability of online learning platforms. Well-known examples are Khan Academy[1] in 2008, Udemy[2] in 2010 and Coursera[3] in 2012. These platforms aim to make courses on all kinds of subjects available to anyone with an internet connection. The classes in these courses are conducted online and are often pre-recorded so that each student can view the classes in her own time. In these courses, assignments have to be made and quizzes need to be taken by the student to show that the student masters the subject of the course.

One of the problems with these courses is that the number of students makes it infeasible to assign human tutors to students that provide feedback on the assignments. Most courses offer feedback by providing a forum where human tutors can answer questions, but personalized feedback on the quizzes and assignments is not provided in most cases.

1.1. INTELLIGENT TUTORING SYSTEMS

This is where the field of artificial intelligence can contribute. The processing power of computers is now advanced enough to allow online personalized algorithmic feedback on assignments and quizzes. Systems that can provide automatic feedback are part of a class of systems that are called Intelligent Tutoring Systems (ITS) [VanLehn, 2006].

The use of an ITS does not end with providing feedback, as the student's progress and knowledge level can also be tracked. The level of sophistication in providing feedback and estimating the students' knowledge levels can vary between systems.

1.2. NEED FOR A STUDENT MODEL

Students often differ in their level of understanding of the knowledge components in a course. However, when an ITS, teacher or lecturer needs to create exercises, the exercises generally will be judged on the average level of all students, if the exercise is not meant for a specific student. This is not so much a problem for exams, as exams assume a certain knowledge level of the students, but when creating exercises to practice the course material, it might be good to generate exercises for several difficulty levels.

[1] https://www.khanacademy.org
[2] https://www.udemy.org
[3] https://www.coursera.org

However, in most cases, students master the topics of a course on a different level. This leads to differences in how students perceive the difficulty of a specific exercise. Because of this discrepancy, we think that students might be offered exercises, or students might select exercises, that are too difficult or too easy for their level. As a result, students won't learn as much as they could have learned, or they might be demotivated to continue during the exercise. This way, the difficulty level of an exercise can directly influence the performance of a student. If an exercise requires too many cognitive resources, the performance of the student will drop [Allscheid and Cellar, 1996; Cornelisz and Klaveren, 2018].

It would therefore be better to offer exercises based on the knowledge level of the student and motivate students by offering challenging exercises that are not too difficult. Several ITSs exist that use student models to select exercises of the appropriate level [Gluz et al., 2014; Mostafavi et al., 2015]. Other ITSs exist that can generate exercises [Dostálová and Lang, 2007], but no ITS combines both approaches, as far as we know.

We propose to create a student model that serves two purposes: the first purpose is to track the knowledge level of a student based on their interactions with an ITS for propositional logic. By looking at the interactions, correct steps and incorrect steps that students take, we hope to estimate the knowledge level of students.

The second purpose is to classify exercises based on the knowledge level of a specific student. As mentioned before, if a student would like to work on an easy assignment, the 'easy' part is open for discussion. By tracking the knowledge level of an individual student, we can make a judgment on an individual basis about the exercise for a specific student.

Our scientific contribution is that we combine the classification of exercises and knowledge level tracking in the same student model for propositional logic, as we have found no other student models that do the same. Furthermore, we will apply the model in a small experiment to determine if the model works as expected.

1.3. Outline

In this thesis, we will investigate what characteristics can be used in a student model, propose a specific model and test that model in a live session with actual students. Chapter 2 will provide an overview of the research on student models in ITSs. Chapter 3 introduces the ITS that we use in our research (LogEx) and describes the research questions that will be answered in this thesis. Chapter 4 provide more theoretical background on Bayesian Knowledge Tracing models. Chapter 5 will describe the student model in detail. Chapter 6 will describe the experiment we conducted and provide the data gathered in the experiment. Chapter 7 contains a conclusion and proposes several areas for further research.

2

RELATED WORK

Intelligent tutoring systems (ITS) have been researched extensively in the past decades. A comprehensive overview of the inner workings of an ITS is given by VanLehn [2006], which breaks an ITS down into two loops: an inner loop and an outer loop.

The inner loop focuses on the tasks that can help the student solve a problem. These tasks include providing feedback on steps taken by the student, providing hints on the next step to be taken, and reviewing the student's solution.

The selection of exercises usually takes place in the outer loop. The complexity of the decisions made in this loop can vary in complexity, ranging from simply offering a fixed list of exercises to advanced systems that select the exercise for the student based on the knowledge level of the student.

VanLehn [2006] describes two approaches for an automated decision: *Mastery learning*, where a student is offered exercises from a single knowledge component until she has mastered that component, and *Macroadaptive learning*, where each exercise is coupled to a set of knowledge components. The student model keeps track of the mastery of a student for each component. Exercises are selected based on the overlap of the components and the mastery level of the student.

2.1. STUDENT MODELS

Sison and Shimura [1998] define a student model as a qualitative representation that accounts for the *student behavior* in terms of the student's *knowledge background*. The student behavior is the student's *observable* response to a specific problem in the tutor's domain. It can be measured on assignment-level and on the level of steps in the exercise.

Chrysafiadi and Virvou [2013] provide an overview of how a student model is developed. They pose three questions that should be answered when constructing a student model:

1. What are the characteristics of the user we want to model?

2. How do we model them?

3. How do we use the student model?

2.1.1. CHARACTERISTICS

According to Guangbing Yang et al. [2010], an effective personalized student model must contain both domain-dependent and domain-independent characteristics. Jeremić et al. [2012] state that some characteristics are static and can be prefilled, such as age, tongue language, and some are dynamic and must be derived from the student's interaction with the system.

Dynamic characteristics can include knowledge and skill, errors and misconceptions, preferences, affective states, and (meta-)cognitive factors. The required domain knowledge is often broken down into knowledge components: facts, rules, procedures, or concepts that should be known or mastered.

2.1.2. TYPES OF STUDENT MODELS

Chrysafiadi and Virvou [2013] describe several student model approaches. These classifications of student models do not exclude each other. Often, models use multiple techniques to model student knowledge. In particular, the student models described in 2.3 use a combination of machine learning techniques and one of the classifications described here to model student behavior.

Overlay The overlay model assumes that the knowledge of a student is a subset of the domain knowledge. Domain knowledge is often defined by a domain expert. The most basic model assigns Boolean values to knowledge components that are assumed to be known or unknown by a student. Since overlay models cannot model the misconceptions of a student, nor the learning styles and affective states, Rivers [1989] argues that overlay models cannot be used for sophisticated models.

Perturbation Perturbation models are based on overlay models, with the extension of a bug library. The bug library enables the modeling of misconceptions of a student and contains buggy rules that model these misconceptions. The buggy rules are either provided beforehand by domain experts (enumerative models) or deducted from interactions with students (generative models).

Stereotypes Stereotype models cluster students according to the characteristics of groups. These groups are called stereotypes. Students are classified with one or more stereotypes based on their performance on the exercises in the ITS.

Machine-learning-based models Most machine-learning techniques are based on observing students' interactions with the system and are used to learn which characteristics are needed to model the students. There are two areas that often use machine learning techniques:

1. (online) learning of the knowledge level of students, based on the behavior of the students while using an ITS,

2. to automatically extend or construct the bug library

Cognitive-theory based models These models use known cognitive theories to model student behavior. Cognitive theories, related to learning, try to model the learning process by explaining thinking and understanding processes in humans. For example, the Human Plausible Reasoning theory [Collins and Michalski, 1989] categorizes inferences in a set of frequently occurring inference patterns and transformations on these patterns. This allows for computers to use more humanlike reasoning models. An example of an ITS that applies the HPR theory is F-Smile [Virvou and Kabassi, 2002], which applies HPR to diagnose errors and generate humanlike advice.

KNOWLEDGE TRACING AND MODEL TRACING

Besides the classification of ITS types, another distinction can be made between student models that track the knowledge state of students (*knowledge tracing*) and models that track how students solve problems (*model tracing*) [Corbett and Anderson, 1994]. Knowledge tracing models focus on assessment: how well does a student master concepts in the domain, while model tracing models focus on helping students who reach an impasse while solving an exercise [Millán et al., 2010]. Model tracing works by providing a set of rules and buggy rules to the ITS. When given a solution by a student, the ITS then uses these rules to reconstruct the steps that the student took to come to her solution. If only valid rules were used, the student didn't make a mistake [Mitrovic et al., 2003].

2.2. ITEM GENERATION

An important feature of ITSs is that they provide a student with enough exercises to improve their skills. Traditionally, course tutors had to generate exercises manually, which is very labor-intensive.

In ITSs the task of generating exercises is often automated, where templates of exercises are provided by human editors and a program creates exercises based on these templates [Prados et al., 2005; Zavala and Mendoza, 2018], where the templates are generated by a program [Ahmed et al., 2013], or where the templates are generated semi-automatically with human intervention [Rajamani et al., 2012].

Another approach is to use Answer Set Programming (ASP), which is based on logic programming languages. In ASP, the domain is modeled in constraints consisting of facts and rules, which are defined in first-order logic predicates, and a SAT-solver is used to find solutions that satisfy these constraints. This approach is used in several studies [Andersen et al., 2013; O'Rourke et al., 2019; Smith et al., 2012]. The resulting solution would be an exercise that is solvable. The rules for ASP can contain both the rules of the domain and the constraints that the designers would like to impose on the created exercises. For instance, O'Rourke et al. used ASP to generate algebra exercises, where the rules contained constraints on the number of terms in an equation and the number of steps that were needed to rewrite the equation [O'Rourke et al., 2019].

Ahmed et al. [2013] argue that item generation has two other advantages. It can help to avoid copyright issues since it is not necessary to copy exercises from textbooks to create a fresh set of exercises. It can also help in preventing plagiarism since students are presented with their own unique set of exercises, and sharing answers to exercises with other students is therefore irrelevant.

2.3. Intelligent Tutor Systems for Logic

Lodder et al. [2016] describe several propositional logic ITSs. Most of these ITSs use either a fixed set of exercises or allow users to define exercises, but two (Organon and FMA) also generate exercises. The paper investigated ITSs that do not support student models. However, other logic ITSs exist that do use student models. We will discuss the ITSs that generate exercises or use student models here.

2.3.1. DeepThought

DeepThought (Mostafavi et al. [2015]) is a logic tutor for building logical derivation proofs. It uses a variation of the mastery learning student model. The tool is developed to provide custom hints and worked examples to the students and to suggest the next difficulty level using data-driven methods.

DeepThought uses a mastery learning model called Knowledge Tracing, described by Corbett and Anderson [1994], where the proficiency of students on knowledge items is traced based on the performance of the student compared to other students. The exercises are categorized into several levels, and students are required to solve exercises with a lower difficulty level first. The speed at which students progress depends on the number of exercises that a student must solve, which, in turn, depends on the proficiency of a student at the current level. The proficiency at the next level is then determined based on a weighted score on the student's performance at the current level. Experts provided the weights in previous versions of DeepThought, but the latest version of DeepThought uses a classification method to determine each level's weights. Mostafavi and Barnes [2017] describe the evolution of DeepThought.

Students can also view worked examples of exercises and receive hints based on the steps taken. Exercises are provided by domain experts.

2.3.2. FMA

FMA [Prank, 2014] is a tool that offers exercises to rewrite propositional formulas to disjunctive and conjunctive normal form. FMA analyzes the steps that the student takes while solving an exercise. The article describes how a feedback analyzer could generate feedback by comparing each step of a student to a generated solution of FMA. This solution is generated using an algorithm described by Prank [Prank, 2014]. Each deviation by a student to the standard solution is considered to be a mistake, as the algorithm will generate the most optimal solution.

2.3.3. Heráclito framework and EvoLogic

The Heráclito framework The Heráclito ITS framework is described in Gluz et al. [2014] and Galafassi et al. [2019]. It is a tool that helps to teach natural deduction in propositional logic to students. The framework contains three agents: the *mediator*, the *specialist* and the *student profile*.

The mediator selects pedagogical strategies and controls the tutoring process based on the student model. The specialist module is responsible for evaluating the student item solutions and the steps taken by the student. Heráclito has evolved over the years and now generates possible solutions to solving the current exercise given the steps that the student provides.

The Heráclito framework does not contain a module to generate new exercises, as far as we know.

EvoLogic The EvoLogic specialist model [Galafassi et al., 2020] is an evolution on the Heráclito framework. It uses a genetic algorithm to generate all possible paths to a solution of an exercise. This allows EvoLogic to follow the steps of a student and generate hints that are in line with the reasoning of the student. One of the problems in the specialist model of Heráclito was that Heráclito would generate confusing hints, by using derived rules as a hint, while the student would only use or understand basic rules. EvoLogic is able to recognize this and suggest hints that are more in line with the student's solution.

2.3.4. ORGANON

Organon [Dostálová and Lang, 2007] is a logic ITS that is developed by the University of West Bohemia. It consists of a database containing exercise templates on several logical domains (categories), such as propositional logic and predicate logic, an assessment module that can generate exercises based on these templates, and a practice module. This module can show example solutions, check the student's answers, and provide hints. Each category contains a hierarchy of types. A type is made out of the steps that need to be taken to solve an exercise and thus denotes the difficulty of these exercises.

2.3.5. UNIVERSAL PROOF GRAPH

Ahmed et al. [2013] describe the Universal Proof Graph method to generate exercises for natural deduction problems automatically. It uses the truth tables of equivalent canonical propositions, inference rules and rewrite rules to search for equivalent propositions. The Universal Proof Graph is a hypergraph, where the nodes are the truth tables of propositions with the same number of variables n and with the same maximum size s. The set of nodes is built by calculating all canonical propositions and their truth tables for all propositions with n variable and of size s. A hyperedge is a set of tuples that are obtained by applying inference rules on premises and where the conclusion, using the rewrite rules, can be mapped to a truth-table node. The tuples consist of a list of all premisses and their conclusion. Similar exercises can be found by searching the graph. The initial exercise is solved using a breadth-first search in the graph. This results in proofs for that exercise. Similar proofs are then found by performing a back-tracking backward search.

3

RESEARCH DESCRIPTION

3.1. THE GOAL OF THIS RESEARCH

The ITSs that are described in the previous chapter all focus on either generation of helpful feedback (DeepThought, FMA, Heráclito, EvoLogic, Organon) or the creation of exercises (Organon, Universal Proof Graph). DeepThought and Heráclito also take the difficulty of exercises into account in the student models that they use. They also suggest exercises based on the knowledge levels of the student and the difficulty of the exercises.

However, as far as we know, there is no logic tutor system that combines these approaches. The discussed systems that generate exercises do not track the knowledge of students who are using the system. The systems that model the student knowledge use predefined exercises that are classified manually.

3.1.1. LOGEX

Propositional logic is part of the programs of mathematics, computer science, and philosophy studies. However, courses related to formal languages are often considered difficult and suffer from high failure rates [Ehle et al., 2018; Mandrioli, 1982; Surma, 2012].

Several ITSs have been developed that can help students pass these courses. The LogEx ITS [Lodder et al., 2016] is developed to practice rewriting propositional formulas to disjunctive normal form, conjunctive normal form, and to prove logical equivalence between two propositional formulas. LogEx can provide feedback on steps taken to solve the exercises and provide hints on the steps to take towards the solution.

Figure 3.1 shows the selection of exercises that a student can make in LogEx. A student can select an exercise from a fixed set of exercises, online randomly generated exercises, or manually specify formulas to rewrite. The randomly generated exercises are categorized into three levels. The student can choose in which difficulty level she wants to solve an exercise.

LogEx uses model tracing to generate hints and feedback. When a student submits a step in rewriting a formula, the LogEx compares the solutions of students to a solution generated from the steps the student took and a possible solution for the rest of the exercise. It can detect mistakes made by a student, provide feedback, and generate hints on how to proceed. Feedback is given on syntactical errors and on mistakes when applying rewrite

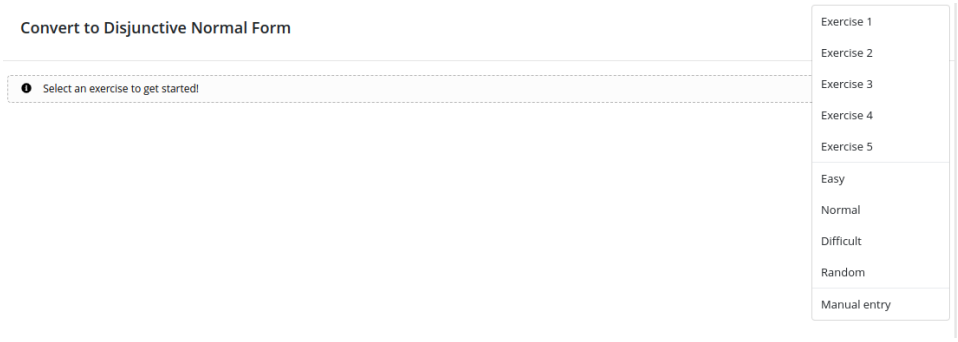


Figure 3.1: The selection options of a student in LogEx.

rules. When a student enters a step that diverges from an optimal strategy, LogEx will recalculate the best strategy from there.

The backend system for LogEx consists of several services that write records for each request to a service to the database. These records capture the input and output of each request. Appendix A describes the attributes of the database records.

3.1.2. Difference in difficulty

Table 3.1 shows three formulas in each of the different levels of difficulty as currently classified by LogEx , along with the number of steps needed to rewrite the formula to DNF and the number of distinct rules to rewrite the formula, according to the preferred way by LogEx. Other solutions might be possible, correct and possibly contain fewer steps. The steps to rewrite the formulas to DNF are also generated by LogEx . The number of steps to rewrite a formula to DNF is low for the easy formulas compared to the other two levels, as might be expected, and increases as the difficulty increases.

Although the sample is too small to make any claims, it appears that the difference between the easy exercises and medium exercises is more pronounced than the difference between the medium and difficult exercises.

Pol [2010] proposed a more refined method to classify DNF rewrite exercises based on the complexity of the formulas and rewrite rules that are used when rewriting a formula to DNF. Each rewrite rule is given a complexity score based on the lowest value of the complexity of either the premise or consequent of the rule. This complexity is described by Feldman [2003] and is based on the number of literals in the *minimal equivalent formula* of the rule. The minimal equivalent formula is the shortest formula that is equivalent and contains the least number of literal occurrences.

The total complexity of an exercise can then be calculated by counting the number of steps or by summing the complexity of each rule used in the solution. The resulting value can be mapped to a scale that denotes the 'basic complexity' of an exercise. Further differentiation can be done by calculating an average of the complexity of the rewrite rules and adding this to the basic complexity. This differentiation might, for example, help to separate exercises that have the same number of steps (and the same basic complexity), but use different rules. One exercise might use more complex rules than the other, and as a consequence, should be regarded as more difficult.

Table 3.1: Formulas generated by LogEx for DNF rewrite exercises. The number of steps is not necessarily the smallest number of steps needed but indicates the number of steps needed in the preferred solution by LogEx.

Difficulty	Formula	Steps	Distinct rules
Easy	$q \rightarrow \neg(p \vee \top)$	4	4
	$(q \rightarrow r) \rightarrow (p \vee q)$	4	3
	$(p \vee q) \wedge \neg q$	3	3
Medium	$(r \wedge (p \leftrightarrow p)) \rightarrow ((r \vee q) \vee \neg p)$	8	5
	$(r \rightarrow p) \rightarrow \neg r \vee (r \rightarrow p)$	6	4
	$((p \leftrightarrow r) \vee p) \rightarrow p$	10	8
Difficult	$((s \vee q) \rightarrow q) \wedge (p \vee s)$	10	7
	$((r \vee s) \rightarrow s) \vee ((p \vee r) \leftrightarrow s)$	7	5
	$\neg((r \rightarrow p) \rightarrow (p \vee s))$	8	7

For example, the minimal equivalent formula of the formula $q \wedge ((q \wedge r) \vee \neg q)$ is $q \wedge r$. To calculate the basic complexity we need to count the number of steps it will take to rewrite the formula into DNF. Appendix B shows the tables for the basic complexity scores, and figure 3.2 shows a derivation generated by LogEx. It takes four rules to rewrite the formula into DNF. This means that the basic complexity of the formula is 1, based on the number of rules used in the derivation. According to the complexity of the minimal equivalent formula, the basic complexity score is 1 as well, as the minimal equivalent formula $q \wedge r$ contains two literals. The additional complexity is calculated by averaging the complexity of the rules, according to table B.3, and adding the average to the basic complexity score. In the case of this formula, the total complexity score would be $1 + (4/4) = 2$.

STEP		FORMULA	RULE
1		$q \wedge ((q \wedge r) \vee \neg q)$	
2	$\Leftrightarrow$	$(q \wedge q \wedge r) \vee (q \wedge \neg q)$	Distribution
3	$\Leftrightarrow$	$(q \wedge q \wedge r) \vee F$	F-rule complement
4	$\Leftrightarrow$	$q \wedge q \wedge r$	F-rule disjunction
5	$\Leftrightarrow$	$q \wedge r$	Idempotency

Figure 3.2: The derivation for the formula $q \wedge ((q \wedge r) \vee \neg q)$ into disjunctive normal form as generated by LogEx.

Although this method might result in a more refined judgment in general than the current method in LogEx , we think that the *perceived* difficulty is more relevant for a student. Since this perception is highly individual, it is hard to capture in standardized rules and tables. As a result, a more individualized method to classify the difficulty level is needed.

3.1.3. RESEARCH PROBLEM AND QUESTIONS

As described in the introduction of this section, we are unaware of an ITS for propositional logic that has a student model and can use the student model to classify the difficulty level of an exercise based on this student model. Our research aims to extend the existing capability of LogEx with a student model to keep track of the knowledge level of individual students and to propose a method to classify the generated exercises for an individual student based on the student model of that student.

In our research, we will search for an answer to the following question:

Main research question: *How can a student model be used to automatically estimate the difficulty level of an exercise in an ITS for propositional logic based on the knowledge level of an individual student?*

We will try to find characteristics of an exercise that can be used to express the level of difficulty of an exercise. We will use these characteristics as input to create a student model for LogEx and implement the student model. Then we will have students interact with LogEx and ask for their feedback on the difficulty level of the exercises they make. We can use this feedback to see if the model can correctly predict the difficulty level of an exercise.

We have chosen to use the Bayesian Knowledge Tracing model [Corbett and Anderson, 1994] as the base model, and make small changes to it. We chose this model as it can model the relation between the interactions of the student and the knowledge level of a student in an intuitive way. This makes it easy to understand the model and define how the model should be updated. In section 5.3 we will describe in more detail how this relation is modeled.

The main question is divided into three subquestions:

Q1 Which characteristics can be used to classify the difficulty level of a propositional logic exercise?

We need to find the characteristics of propositional formulas that can differentiate between the difficulty levels. By asking domain experts to classify randomly generated formulas and the characteristics they look at to classify the formulas as such, we can analyze their answers and pick characteristics that can be used to classify formulas and indicate the knowledge level as well, and that we can track in LogEx.

Q2 How can we construct a student model to track the knowledge level of students, and how can the model be updated?

We will use the framework described in Millán et al. [2010] to define student models using a Bayesian network. We will describe how the model can be updated using the interactions of the student with LogEx.

Q3 How well does the model align with the student's perception of the difficulty level of an exercise?

We will need to compare the classifications of our model to the student's perception of the exercises that they did, so we can verify that the model works as intended. We will hold an experiment where we can gather the student's feedback on the difficulty

levels of the exercises they do in LogEx. We will build student models and classify the same exercises, so we can compare the answers of the students to the classification of the student models.

4

BAYESIAN STATISTICS

4.1. A QUICK PRIMER ON BAYESIAN STATISTICS

Probability theory can be used to create models to help diagnostic or predictive hypotheses. Whenever, for a set of variables in a domain, the joint distribution $P(x_1, \ldots, x_i, \ldots, x_n)$ is known, then any kind of analysis can be made on certain events in that domain.

Example If a car mechanic wants to diagnose why a car has a flat tire, he could think about the reasons why a tire could run flat (i.e., glass on the road or the tire is old and worn out). He could then estimate the chances of a tire running flat because of these factors and make an educated guess about why the tire has run flat. Usually, the decision is based on the option with the highest probability.

4.1.1. THE DIFFERENCE BETWEEN PROBABILITY AND LIKELIHOOD

The word 'likelihood' is often used as a synonym for probability in everyday use. However, in statistics, likelihood is not the same as probability. To see the difference assume some probability function $P(E \mid \theta)$, where E is some observation, and θ is a set of variables on which the probability function depends. We can reason about this in two different ways.

If we observe event e and know the values in θ, we speak of probability when calculating $p(e \mid \theta)$. In other words: probability is the expectation of e occurring, given that we know θ.

Likelihood is used when we do not know θ, and we know e. This is, for example, the case if we know e, we have several options for θ and would like to reason about which valuation of θ explains e the best. That is: we have some observation of an event and want to find the parameters that would maximize the likelihood $p(e \mid \theta)$. A common way to do this is the **maximum likelihood estimation**.

4.1.2. A BAYESIAN NETWORK

A joint distribution table is a table that enumerates all possible values of the variables in the model and the likelihood of each combination. However, in the optimistic case that all the variables take boolean values, constructing a full joint distribution takes 2^n values, which is often not feasible in a real-world domain. If the variables are conditionally independent of each other, a Bayesian network can be used to model the same joint distribution, but with fewer values [Pearl, 1985].

A Bayesian network is a directed acyclic graph (DAG), where the nodes in the graph represent the variables in the model, and the arcs represent the relations between the variables. The variables in the model take *mutually exclusive* and *exhaustive* values, and can either be discrete or continuous values. When there are discrete but infinite values possible, a rest case can be added to ensure there are finite possibilities. If the values are continuous, binning can be applied to map the values to discrete values. Still, a density function (such as a Gaussian distribution) is often used to model the probabilities.

A relationship indicates that a node has some influence on another node: a node is, to some degree, the cause of some effect in the other node. If there is an arc from node X to node Y, then X is a parent node of Y. Each node X_i contains a conditional probability distribution table $P(X_i|\pi(X_i))$, where $\pi(X_i)$ are all the parents of node X_i. The topology of the network is often designed by domain experts, but can also be learned from data [Russell and Norvig, 2014].

Each record in the conditional probability table indicates a probability for some 'possible world' of the parent nodes: the probability for the valuation of the node variable, given a specific valuation of the parent nodes. The probabilities in each row must add up to 1. In the case of boolean variables, it is enough to specify the probability for one of the values (p), as the probability of the negation will be $1-p$.

An important assumption underlying Bayesian networks is that of *conditional independence* [Millán et al., 2010; Pearl, 1985]:

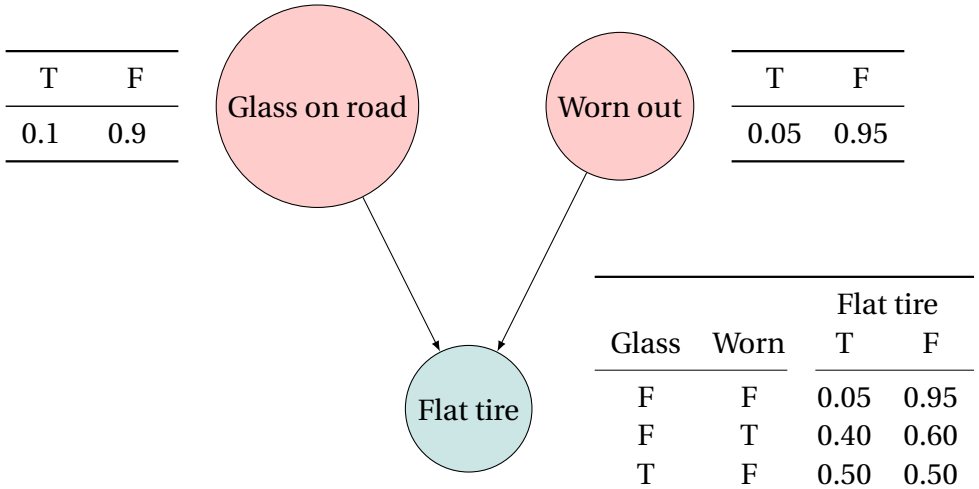
Definition 4.1 (Conditional independence) *If V is the set of all nodes in the model, then the model is conditional independent iff for each $x \in V$ and for each $y \in V \setminus \{x\} \setminus \pi \setminus \gamma$, x is independent of y, where π is the set of direct parent nodes of x and γ is the set of the immediate children of y.*

When the model meets the conditional independence condition, the joint distribution can be calculated as the product of the conditional probabilities of the relevant variables (**Chain rule**): [Pearl, 1985]

$$P(X_1 \wedge X_2 \wedge \cdots \wedge X_n) = \prod_{i=1}^{n} P(X_i|\pi(X_i)) \tag{4.1}$$

Bayesian networks can be used to determine the probabilities of events that are modeled by the network. This process is called inference and results in the **posterior** probability of some event. These probabilities are different from the probabilities as defined in the probability tables in the network, which are called **priors**, as an event leads to evidence: some observed variables in the network have fixed values given an event. For example: a customer observed that her tire is flat. Not all variables in the network need to be observed. The variables that are not observed are called the *unobserved* variables.

Example The car mechanic has estimated the chances of glass on the road and found that glass is present in 10% of all roads. He also found that 5% of all cars drive around with worn-out tires. Therefore, the chance of a flat tire when there is no glass on the road and



T	F
0.1	0.9

T	F
0.05	0.95

Glass	Worn	Flat tire T	Flat tire F
F	F	0.05	0.95
F	T	0.40	0.60
T	F	0.50	0.50
T	T	0.95	0.05

Figure 4.1: A small Bayesian network for a car mechanic

the tires are not worn-out is small, but the probability of a flat tire increases when these conditions occur. A Bayesian network for the example above would look like figure 4.1.

Now, suppose a customer arrives with a flat tire, and she wants to know the odds that her driving causes the flat tire through the glass. This type of question is a diagnostic question (what causes my flat tire?), and the network can answer it.

The probability of glass on the road given a flat tire is $p(g \mid f)$, which is a short notation for $p(\text{Glass} = \text{true} \mid \text{Flat tire} = \text{true})$. Using Bayes rule we know that

$$\begin{aligned} p(g \mid f) &= \frac{p(f \mid g)p(g)}{p(f)} \\ &= \frac{p(f \wedge g)}{p(f)} \\ &= \frac{\sum_{w \in W} p(f \wedge g \wedge w)}{\sum_{g \in G, w \in W} p(f \wedge g \wedge w)} \\ &= \frac{p(f \wedge g \wedge w) + p(f \wedge g \wedge \neg w)}{p(f \wedge g \wedge w) + p(f \wedge g \wedge \neg w) + p(f \wedge \neg g \wedge w) + p(f \wedge \neg g \wedge \neg w)} \end{aligned} \tag{4.2}$$

We can now use equation 4.1 and the probabilities specified in the network to answer the question.

$$\begin{aligned} p(f \wedge g \wedge w) &= p(f \mid g \wedge w)p(g)p(w) &&= 0.95 \times 0.1 \times 0.05 = 0.00475 \\ p(f \wedge g \wedge \neg w) &= p(f \mid g \wedge \neg w)p(g)p(\neg w) &&= 0.5 \times 0.1 \times 0.95 \quad = 0.0475 \\ p(f \wedge \neg g \wedge w) &= p(f \mid \neg g \wedge w)p(\neg g)p(w) &&= 0.4 \times 0.9 \times 0.05 \quad = 0.018 \\ p(f \wedge \neg g \wedge \neg w) &= p(f \mid \neg g \wedge \neg w)p(\neg g)p(\neg w) &&= 0.05 \times 0.9 \times 0.95 = 0.04275 \end{aligned}$$

These values can now be used in equation 4.2. This leads to the probability of $p(g \mid f) \approx 0.462$. That is: there is a chance of about 46% that the flat tire is caused by glass on the road.

4.2. Dynamic probability models

A traditional Bayesian network models the current state of affairs, but does not consider dynamic variables, such as time. Several types of models can capture dynamic information, such as Hidden Markov Models and Dynamic Bayesian models. These two types are relevant to our model and will be discussed here.

4.2.1. Hidden markov models

A Hidden Markov Model (HMM) is a probabilistic model modeling a single variable over time. The states of the model are the possible values of the variable. A probability is defined for each transition between two states, indicating how likely that transition between these two states will occur. The goal of an HMM is to model unobservable (hidden) states based on a single observable variable.

Hidden Markov Models are based on the assumption of a Markov Process, which is a process that models sequences of events, where the probabilities of the next event are only influenced by the state obtained by the current event. That is: events before the current event do not influence the probabilities of the next event. If X is a hidden state, Y is an observable event, and t denotes the current time, then:

$$P(Y_t \mid X_t, X_{t-1}, \ldots, X_1) = P(Y_t \mid X_t) \tag{4.3}$$

If more than one variables need to be modeled by an HMM, the variables in the domain can be combined into a single variable by creating tuples of the possible valuations of the variables. However, this increases the state space considerably, and it might be better to look for other approaches to model the domain.

4.2.2. Dynamic Bayesian Networks

A solution to the state space problem in HMM is to use a Dynamic Bayesian Network (DBN). A DBN is a model like a 'normal' Bayesian network, but the probabilities are updated over time. For each tick in time, the structure of the model is copied and probabilities are updated with new information from the previous event.

One of the reasons to model a multi-variable domain in a DBN is that a DBN is more sparse than an HMM. It might be clear that an HMM can be represented as a DBN with a single state variable and a single evidence variable. However, as indicated in the last paragraph of 4.2.1, an HMM can model multiple variables, and so a DBN can also be transformed into an HMM. This can be done by transforming a DNB with n states into a single HMM variable that consists of a n-tuple.

Consider a domain with 20 boolean variables and where each variable has three boolean parent variables. The DBN will have to model 20×2^3 probabilities. In contrast, an HMM needs to model 2^{20} possible states and even more transitions and probabilities, which makes a DBN more suitable for larger problem domains.

To construct a DBN, one needs to specify the prior distributions over the variables $P(X_0)$, the rules to update the probabilities for $P(X_t \mid X_{t-1}, \ldots, X_1)$ (also known as the update model) and a probability distribution for the events $P(E_t \mid X_t)$ (sensor model).

The update model is the essential difference between an HMM and a DBN. A DBN does not assume to be a Markov Process, and therefore the update model does not necessarily

only depend on the previous state of the model. For example, it might not take the history into account for some of the variables (but at least one variable as it is no longer dynamic otherwise), and it might depend on more states in time than only the previous state.

4.3. BAYESIAN KNOWLEDGE TRACING MODELS

As described in section 2.1.2, the goal of knowledge tracing models is to estimate the knowledge level of students. For example, Corbett and Anderson [1994] proposed a model that can keep track of the change in the student knowledge level during practice in an ITS for programming concepts in LISP, Prolog, and Pascal.

The ITS was based on coding rules, which should be learned when writing programs. Each coding rule can either be in a learned state or an unlearned state. While reading and practicing exercises, the state can transition from unlearned to learned. The model did not account for forgetting (switching from learned to unlearned), but to allow for errors a slip probability was taken into account.

The Bayesian Knowledge Tracing (BKT) model can be seen as both an overlay model and a Hidden Markov Model. As described in 2.1.2, an overlay model keeps only track of the knowledge components in the domain (in this case: the rules). For each rule, an HMM is constructed to estimate the knowledge state of the rule.

Each time the student has an opportunity to apply a coding rule, the estimated knowledge level is updated. The knowledge level is modeled as the probability $p(L_n)$ of the coding rule being in a learned state:

$$p(L_n) = p(L_{n-1} \mid \text{evidence}) + (1 - p(L_{n-1} \mid \text{evidence})) \times p(T) \tag{4.4}$$

This update consists of two separate probabilities:

- $p(L_{n-1} \mid \text{evidence})$: the probability that the rule is in the learned state, given the evidence. This probability is inferred using Bayes' rule.
- $(1 - p(L_{n-1} \mid \text{evidence})) \times p(T)$: the probability that the rule is in unlearned state, but transitions to the learned state.

The context of the ITS was to provide mastery learning for the students. Therefore, the ITS was divided into several sections, where each section offered a set of coding rules. A student could practice the rules in a section until he obtained a sufficient knowledge level (0.95 in the original model).

5

BUILDING THE STUDENT MODEL

This section will explain how the student model is built. We will use the framework described by Millán et al. [2010], which describes how student models can be constructed as Bayesian models to predict or analyse a problem.

5.1. FACTORS CONSIDERED BY DOMAIN EXPERTS

To get more insight into attributes that could be of interest to use in the student model, we generated three sets of 15 formulas. We asked three domain experts (from the teaching staff at Open Universiteit) to give their opinion on how difficult they would think a student would find these formulas to rewrite to DNF and CNF and which factors they considered when they classified the formulas.

The following list contains the factors that the domain experts identified:

- The number of steps required to rewrite the formula
- The number of rules that are applicable in each step
- The order in which to apply the rules
- The length of the original formula
- Which rules should be applied and whether these rules should be applied on atomic propositions or on sub-formulas
- Whether the formula has nested conjunctions or disjunctions after the elimination of equivalence or implication
- Whether the rewriting requires distribution of negation over a sub-formula
- The ease with which simplifications might be identified

All three domain experts agree that a major factor that determines the difficulty level of a formula is the number of steps taken in rewriting a formula. LogEx currently considers this when generating a formula for an exercise. However, this metric is the same for each student, given a particular exercise. To make a more individualized judgment on the difficulty level of an exercise, a student model could track if a student tends to perform worse

on exercises that take more steps to rewrite. The same can be done with the length of a formula and the level of nesting in a formula. These are simple metrics to track and are easy to understand.

However, we would like to be able to make a more fine-grained distinction between the students' knowledge levels. One factor that makes an exercise more difficult for a student is the set of rules that should be applied and the ability of the student to understand how to apply a rule. As the rules are applied in each step in an exercise, we can track the performance of the student in each step, and not on the level of exercises. We expect that the model is more accurate if it looks at the individual steps in an exercise, and as such, we will need to track the performance of a student on the specific rules. In the context of measuring the students' knowledge level in a student model, it makes more sense to track the performance on the level of rules than on any other level, as the rules are the knowledgable items in this domain.

For this reason, we decided to classify formulas based on the rules that should be applied when rewriting the formula.

5.2. Design questions

To model the knowledge level of the students, we will use the framework from Millán et al. [2010]. The following questions can be used as a starting point when constructing a student model:

- What is the problem that needs to be answered, and how can we use a Bayesian network to solve this?

- Which variables are relevant for the domain of the problem?

  The knowledge concepts in a knowledge domain should be taken as a starting point, but other factors can also be modeled as variables. Examples are the emotional states of a student or behavioral patterns.

  Section 5.3 describes the context in which our student model is applied and which variables are used in the model.

- Which states should be used in the network, and which relationships exist between the states?

  The variables identified in the previous question should be modeled as the nodes in the Bayesian network. We must then define the relations between nodes. For example, some knowledge items are prerequisites for knowing other knowledge items. If such relationships exist, then the required knowledge items are parents of the related knowledge items.

  Students' interactions with the system can be modeled as evidence nodes. A link between a knowledge component node and an evidence node indicates that a student must know the corresponding knowledge component in order to apply a step with the corresponding evidence node. But the links can also be reverted. If a student applies a knowledge component correctly (evidence), she shows some level of mastery of that knowledge component.

- Is the model static or dynamic? Do we need to take time into account, and how can we do this?

  When using a Dynamic Bayesian Network as the model, the ITS might be able to track the change of the student's knowledge over time.

5.3. Defining the student model

In this section, the answers to the questions above will be provided, and an explanation is given about the choices made in the student model for LogEx.

5.3.1. Problem description

Students using LogEx can choose exercises from a fixed set of exercises that are labeled with a difficulty level. There are currently five levels in LogEx : Very Easy, Easy, Medium, Difficult, and Very Difficult, of which only Easy, Medium and Difficult are used for LogEx. The framework behind LogEx can generate exercises on demand, where the student must provide the desired difficulty level. The difficulty level of the generated exercises is based on the number of steps needed to rewrite the formula to DNF or CNF, the number of equivalences in the original formula, and by checking the number of subformulas in the starting formula. Subformulas are the parts of a formula that are connected by logical operators. These parts also form logical formulas on their own. The smallest subformula is an atomic proposition.

The goal of the student model is to keep track of the knowledge level of a student and classify the difficulty level of generated exercises for each student based on the knowledge level of the rewrite rules. Ideally, whenever a student wants to start an exercise, the student model should be able to judge the difficulty of an exercise for that student.

Currently, the system estimates the difficulty level by counting the number of steps needed to rewrite a formula to DNF or CNF. As described in section 5.1, we will assume that the difficulty level of an exercise only depends on the knowledge level of the student and the rules that need to be applied. Therefore, our model no longer considers the number of steps taken to 'solve' the exercise.

We hypothesize that students can have differences in how well they understand the rewriting rules and how well they can identify which rule should be applied in a given situation. This is why the model needs to estimate the knowledge level per student.

The underlying assumption in our model is that whenever a student applies a rule, the student gives information about how well she understands the rule that needs to be applied. If the student applies a rule correctly, the student indicates that she understands that the rule must be applied in the current situation. However, if a student incorrectly applies a rule, we assume this indicates the opposite. Even if the student unintentionally applies a rule (in)correctly, the knowledge level will be updated. However, knowing if a student applied a rule intentionally or unintentionally is impossible. This means that we must account for the probability that the student made a correct guess or a mistake.

The knowledge level of the student can be estimated, using these assumptions, by updating the student model with each step that the student takes in LogEx. This means that

we need to identify the following types of variables:

- Evidence variables
- Knowledge variables
- A classification variable

The classification variable acts as a filter where rewrite rules that do not apply for a given exercise are ignored. Only the relevant rules for the current exercise are taken into account when classifying the exercise.

5.3.2. Evidence variables

The evidence variables capture the interaction of the student with the system. Each application of a rule will be recorded and taken into account when updating the probabilities for this variable. The evidence variables take boolean values since each rule application can be correct or incorrect.

We will refer to the probability distribution in an evidence variable of rule r as $p_r(E)$, the probability of a correct application of rule r as $p_r(e)$ and an incorrect application as $p_r(\neg e)$. We will add a superscript to indicate the probability at a specific moment in time.

Apart from the first few applications of a rule, the probability of a student correctly applying a rule is simply the number of correct applications over the total number of applications of the corresponding rule. In the first applications, the probability distribution in an evidence variable is a fixed distribution, which is provided when initializing the model. See section 5.4.1 for the reason behind this decision.

The evidence variables act as conditional variables for the knowledge variables. This structure is chosen because the correct or incorrect application of a rule in an exercise influences the system's belief in how well the student understands the applied rule. The other way around is also a valid option since the knowledge level also influences the probability that the student applies a rule correctly. However, we chose our structure because we try to estimate the knowledge level based on the interactions of the students. Since we update the knowledge level based on the evidence, the update of the evidence and the knowledge level becomes more intuitive when we use this direction.

The probability distribution in the evidence variables is initialized with the same probability for each possible value. This means that we have chosen to model that the students are not likely to apply one rule better than others. The probability distributions in the evidence variables could also be initialized using domain knowledge. In fact, in chapter 6 we will show that our assumption is incorrect.

The probabilities in the evidence variables are updated when the student has applied the corresponding rule a predefined number of times. Section 5.4.1 will describe the update in more detail.

5.3.3. Knowledge variables

The knowledge variables track how well a student knows a rewrite rule. Whether a student knows a rule can also be modeled as facts using boolean values: either the student

knows or doesn't know the rule. As discussed in the previous section, the knowledge variables depend on their evidence variable counterparts. That is: the knowledge level of a student depends on the correctness of the rule applications that the student made. As a consequence, the probability distributions of the knowledge variables can be seen as the system's confidence level in how well a student knows the rewrite rules.

We will refer to the probability distribution for the knowledge variable of rule r as $p_r(K)$, the probability of a student knowing rule r as $p_r(k)$ and not knowing as $p_r(\neg k)$. We will add a superscript n to indicate the probability at a specific moment in time.

The probabilities in the knowledge variables are updated whenever the student applies a rule. This is done by inference on the network: what is the probability that the student knows a rule, given that the student applied a rule correctly (or incorrectly)? Whenever we calculate this probability, we must also keep in mind that the student might have guessed the answer or that the student has made a mistake, which is also known as a slip. This structure resembles the structure of a Bayesian Knowledge Tracing (BKT) model [Corbett and Anderson, 1994]. Figure 5.1 shows the structure of this original model and indicates the use of the different probabilities.

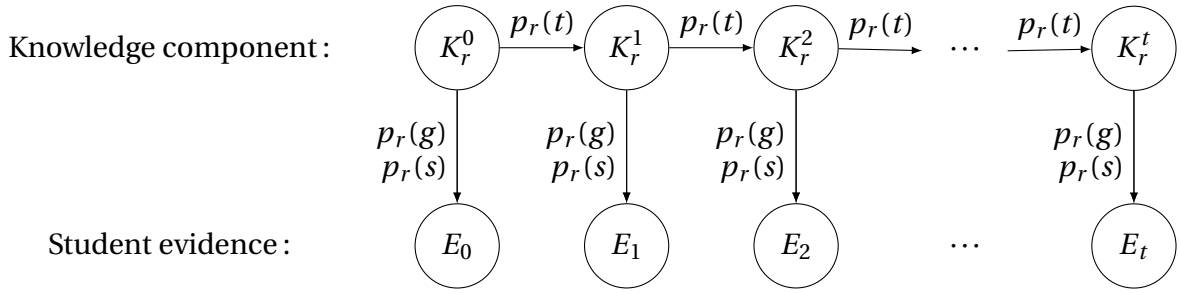


Figure 5.1: The relation between the probabilities in a traditional BKT model as modeled by Corbett and Anderson [1994]. The probability $p_r(t)$ determines the state of the next application of a rule. The probabilities $p(g)$ and $p(s)$ are used to determine the probability of the evidence.

We have introduced a parameter to model the probability of 'forgetting'. We see a slip as that the student must know the rule, then applies it incorrectly, but then realizes that she made a mistake. This implies that the student must still know the rule. This, in turn, implies that there is no transition between states in terms of knowledge: no transition from state 'learned' to state 'not learned', or vice versa, since the student must realize that she made a mistake.

Forgetting does imply this transition since to forget a rule, you first must know the rule. It is debatable whether or not the probability to forget should be dependent on the rule, as some rules might be harder to understand than others, which may cause the rule to be forgotten sooner than the easier rules. To keep the model simple, we decided to define a general probability in the model for forgetting.

As a result, in contrast to the parameter for slip ($p_r^n(s)$), the probability $p(f)$ is fixed in time and not specific to a rule, nor is it dependent on the fact that the student should know the rule before applying the rule incorrectly. See equations 5.7, 5.9 and 5.10 for the difference in usage of these two parameters.

Definition 5.1 (Student model probabilities) *A Bayesian Knowledge Tracing model estimates the probability of a rule being known to a student using four parameters. In addition to the original BKT model, we will also use a probability $p(f)$, to model the probability that a student forgets how to apply a rule:*

- $P_r^0(K)$*: the initial probability distribution of the knowledge variable when initializing the model. Each rule in our model will be initialized with the same distribution, but it might be the case that the accuracy of the model can be improved if each rule has its distribution.*
- $p_r(t)$*: the probability that the student learns a rule for each opportunity to apply the rule (transition).*
- $p_r^0(s)$*: the initial probability that the student makes a mistake (slip) when the rule is known.*
- $p_r^0(g)$*: the initial probability that the rule is unintentionally correctly applied (guess).*
- $p(f)$*: the probability that the student forgets how to apply the rule correctly.*

Figure 5.2 shows the updated model.

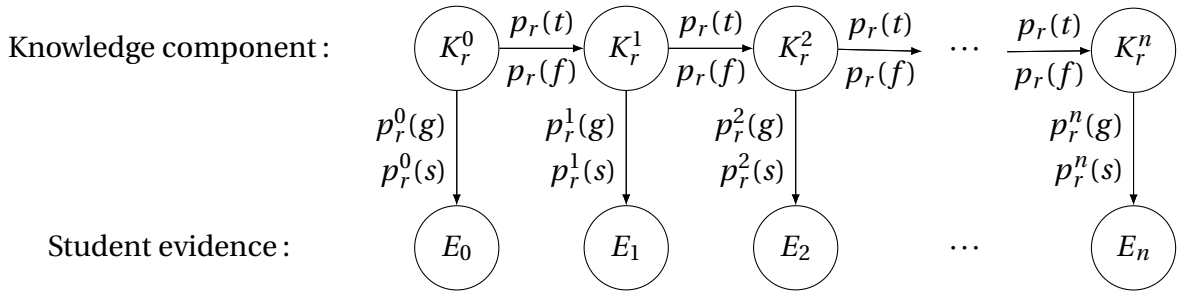


Figure 5.2: The updated picture of our student model, based on the original BKT model.

The model by [Corbett and Anderson, 1994] uses a single variable for each rule, and doesn't include an evidence variable for a rule and it had to specify $p_r^0(K)$. Since we have an evidence variable as parent of the knowledge variable, we need to specify $p_r^0(k \mid e)$ and $p_r^0(k \mid \neg e)$. We can calculate the probability that the student knows rule r as follows:

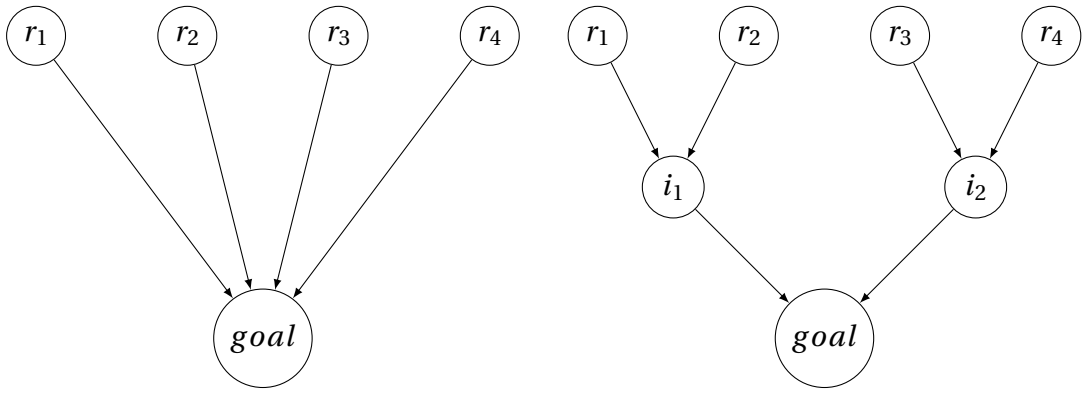
$$\begin{aligned} p_r^n(k) &= p_r^n(k \wedge e) + p_r^n(k \wedge \neg e) \\ &= p_r^n(k \mid e)\, p_r^n(e) + p_r^n(k \mid \neg e)\, p_r^n(\neg e) \end{aligned} \tag{5.1}$$

The probabilities $p_r^n(k \mid e)$, $p_r^n(k \mid \neg e)$, $p_r^n(e)$ and $p_r^n(\neg e)$ can be taken from the model directly as they are specified by the conditional probability tables of the variables.

The model will update the probability $p_r^{n+1}(k \mid e)$ for a rule r each time the student applies that rule correctly, or $p_r^{n+1}(k \mid \neg e)$ if the student applies the rule incorrectly. Section 5.4.2 will discuss the updates in more detail.

The original model used static values for $p_r(s)$ and $p_r(g)$, which would not update given the evidence and the state of the knowledge rule. In our model, the probabilities for $p_r^n(s)$ and $p_r^n(g)$ are not static values since they are equivalent to $p_r^n(\neg e \mid k)$ and $p_r^n(e \mid \neg k)$ respectively. We think that this update of these probabilities is an improvement on the original model, given that students become better with practice and as a result, the probability to slip or guess will differ over time.

5.3.4. GROUPING OF VARIABLES



(a) A simple model with a knowledge variable for each rule. The goal variable has four parent variables. If these variables are boolean variables, the conditional probability table must specify the probability distribution for 2^4 conditional values.

(b) Intermediate variables reduce the conditional probability tables for dependent variables. Since the goal variable now has two parent variables, the conditional probability table now has 2^2 values. The downside is that it might be hard to specify the probabilities in the intermediate variable.

Figure 5.3: The effect of introducing grouping variables.

Our model will use a knowledge variable for each rewrite rule. A more concise model can be built by creating variables that can act as a grouping of rules. One drawback of this approach is that it is no longer possible to distinguish between rules within a group. As such, this approach might lead to a loss of information. For instance, the current model has a variable for each of the De Morgan rules. This can also be modeled as a single variable since these rules are very similar. So we can argue that, in the case of the De Morgan rules, adding a variable for each variant is not adding information.

Reducing the number of variables helps in reducing the complexity of the model and reduces the conditional probability table of the dependent variables. In our model, this is not relevant, as each knowledge variable has only one parent variable, and grouping rules will only reduce the number of conditional probability tables and not the size of the tables.

Another approach is to create a grouping of variables by adding intermediate variables. Although this increases the number of variables, this can also help in understanding the relationships between rules. It can also help reduce the state in conditional probability tables as well. However, intermediate variables must have a meaning as it might be hard to specify the probabilities in such a variable otherwise.

5.3.5. THE CLASSIFICATION VARIABLE

The student model will also contain a single variable to classify the difficulty level of an exercise. When classifying an exercise in LogEx , the student model will be given a set of rewriting rules that are used by the default derivation, and so the classification variable will depend on the knowledge variables that are representing these rules.

If we were to model this variable as a normal variable in a Bayesian network, the conditional probability table for the classification variable would be very large, considering that each exercise may contain any subset of all the rewriting rules. This means that to create a conditional probability table for the classification variable, we would need to enumerate all possible combinations and specify probabilities for each of these combinations, for each possible difficulty level.

Since it would be practically impossible to generate this table, we must define the output of the classification variable differently. We will use several functions to assign an activation value of each of the difficulty levels, given an average knowledge level of the rewrite rules in the exercise. The variable will act as a simple perceptron model.

Let $R = \{r_1, \ldots r_n\}$ be the set of all the rewrite rules in an exercise. We can calculate the probability $p_r^n(k)$ for each of these rules using formula 5.1. We assume that if the student has to apply the same rule multiple times, the student will judge all applications equally regarding the difficulty.

This is not entirely correct since we will average the probabilities for the rules. Multiple applications of the same rule will then influence this average. An improvement to our model would be to keep multiple applications in the calculation for the difficulty level.

We can then calculate a knowledge level (α) for the exercise for a student by calculating the average probability of the student applying the rule correctly given that the student knows the rule for the set R of rules in the derivation, as described in equation 5.2

$$\alpha(R) = \frac{\sum_{r \in R} p_r^n(e \mid k)}{|R|} \tag{5.2}$$

For each difficulty level, we have to define a function that will give an activation value of that difficulty level, given this average knowledge level: $f(\alpha)$. Let $f_{easy}(\alpha), f_{med}(\alpha)$ and $f_{diff}(\alpha)$ be the functions that give the activation for the Easy, Medium and Difficult level respectively.

The characteristics of the classification functions will differ per difficulty level. For instance: f_{diff} must be monotonically decreasing since if the average knowledge level for the exercise is low, the student will find the exercise difficult to solve. Thus, if α is close to 0, the result of the function must be close to 1, and vice versa. The opposite should hold for f_{easy}: if a student has a high knowledge level, the probability of an Easy classification must be high. The function f_{med} will resemble a concave parabola or a bell curve, as it will have low values at both ends of the range of α, but the probability will be high somewhere in between. See figure 5.4 for an example of these functions.

$f(\alpha)$ can be defined as:

$$f(\alpha) = max(f_{easy}(\alpha), f_{med}(\alpha), f_{diff}(\alpha)) \tag{5.3}$$

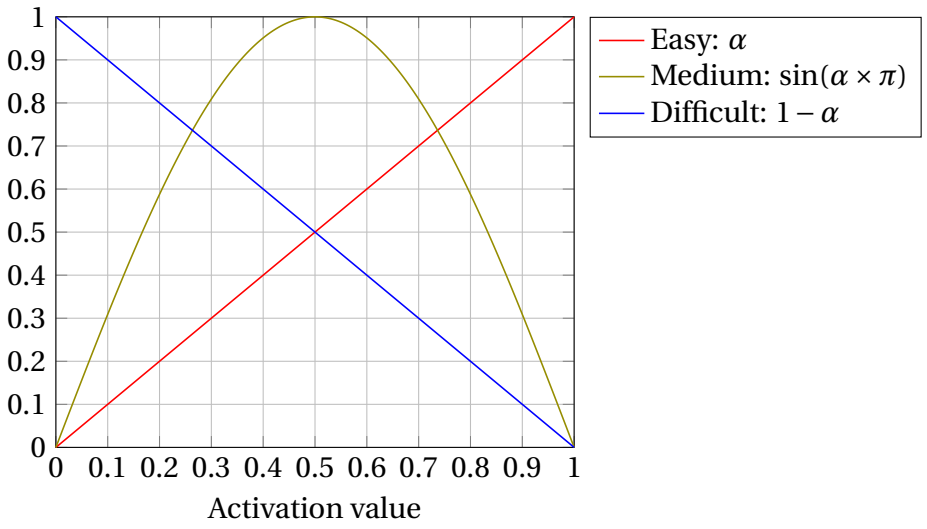


Figure 5.4: Example of the functions in the classification variable. The x-axis denotes the activation value. An exercise will be classified as the class associated with the function that has the highest value given the activation value.

When we have defined these functions, we can classify the exercise according to the following function:

$$class(\alpha) = \begin{cases} f(\alpha) = f_{easy}(\alpha) & = \text{Easy} \\ f(\alpha) = f_{med}(\alpha) & = \text{Medium} \\ f(\alpha) = f_{diff}(\alpha) & = \text{Difficult} \end{cases} \tag{5.4}$$

For now, we will assign an equal weight for each knowledge rule, but we will only use each rule once. If the default derivation of an exercise uses the same rule twice, we will not use the probability of that rule twice. That is: we will create a set of rules from all rules used in a derivation and calculate the value for α based on that set of rules.

We use the set of rules because we do not use information about whether the rule is applied to a formula containing nested sub-formulas or atomic propositions (for example). That is: we assume that each application of the same rule in the same exercise with the same knowledge level is equally difficult. This design choice can be questioned: if an exercise consists of several applications of a very simple rule, but it requires that a very hard rule must be applied once somewhere, we can argue about how difficult the student will find the exercise. We have decided to ignore this question, but we think that this is an interesting question for future research.

5.4. STUDENT MODEL UPDATE

For each step, the student can either provide a correct answer on the first try or an incorrect answer. LogEx only allows submissions that lead to a correct derivation. The default derivation acts as an answer model, but other sequences are also allowed as long as they lead to a correct end result. Such a sequence is called a 'strategy'. Thus, a correct answer in this context is any step that leads to a correct result.

To keep the model simple, we will not model the different strategies and the probabilities of the student correctly applying the whole strategy, but we will focus on individual

steps taken by the student.

Whenever a student provides a correct answer, the estimation of the knowledge level will be updated. When a student provides an incorrect step, the system indicates that the student entered an incorrect answer, and the student must try again. To keep things simple, we assume that students always pick the rule they meant to pick; accidental selection of an incorrect rule will not be accounted for.

A student can also give up on the exercise, which will lead to an unfinished sequence of steps. However, we do not care if an exercise is finished or not, since all steps taken by the student will influence the knowledge levels.

5.4.1. Update evidence

The evidence variables will be used to model the probability that the student correctly applies a rule in a step. Since the student interacts with the system, we can estimate the probability based on these interactions.

When we update the evidence probabilities with each step, the first couple of applications of a rule will influence the probabilities for that rule too much. For the second application, the probability of a correct outcome will either be 1 or 0, depending on the outcome of the first application. To mitigate this, we will only update the probabilities for the evidence variables after the student has applied a rule five times. A drawback of this is that the probabilities for rules which are not used often will be less accurate since they are not updated frequently.

We will only take the last twenty applications of a rule into account and discard any previous applications to avoid the influence of the history and all the errors made by the student while learning the rule, as we think it is unfair to judge a student on these errors. The probabilities for the evidence variables are thus calculated using a sliding window of the last five to twenty applications of the corresponding rules, depending on how often the rule is applied.

A note on the cut-off values (five and twenty): they are chosen by us and are not supported by any study. We felt that they are fair for most of the rules. During our experiment, we found that not all rules are used often enough to update the probability for the rule. We will leave it to further research to find more adequate numbers.

5.4.2. Update knowledge level

Before we update the probabilities of the knowledge variable, we first need to calculate the following probabilities from the conditional probability table of the corresponding variable.

$$p_r^n(k) = p_r^n(k \wedge e) + p_r^n(k \wedge \neg e) \tag{5.5}$$
$$p_r^n(\neg k) = 1 - p_r^n(k) \tag{5.6}$$
$$p_r^n(s) = p_r^n(\neg e \mid k) \tag{5.7}$$
$$p_r^n(g) = p_r^n(e \mid \neg k) \tag{5.8}$$

We can update the probabilities of the knowledge variable by applying these update rules, which are based on the update rule of the Bayesian Knowledge Tracing from [Corbett and Anderson, 1994], and use the probabilities as defined in 5.3.3.

$$p_r^{n+1}(k \mid e) = \begin{cases} p_r^n(k \mid e) \times (1 - p_r^n(s)) + p_r^n(\neg k \mid e) \times p(t) & \text{if correct step} \\ p_r^n(k \mid e) & \text{if incorrect step} \end{cases} \tag{5.9}$$

$$p_r^{n+1}(\neg k \mid \neg e) = \begin{cases} p_r^n(\neg k \mid \neg e) \times (1 - (p_r^n(g))) + p_r^n(k \mid \neg e) \times p(f) & \text{if incorrect step} \\ p_r^n(\neg k \mid \neg e) & \text{if correct step} \end{cases} \tag{5.10}$$

$$p_r^{n+1}(\neg k \mid e) = 1 - p_r^{n+1}(k \mid e) \tag{5.11}$$

$$p_r^{n+1}(k \mid \neg e) = 1 - p_r^{n+1}(\neg k \mid \neg e) \tag{5.12}$$

Equations 5.9 and 5.10 update the probability that the student knows the rule only for the conditioning case of the corresponding evidence. For instance: if the student applies a rule correctly, only the probability for the condition of correct evidence is updated. The new probability is the sum of the probability that the student knows the rule and didn't slip plus the probability that the student didn't know the rule but transitioned to knowing the rule.

In the case that the student didn't apply the correct rule, the probability for the corresponding case is updated. The new probability of not knowing the rule is the sum of the probability that the student didn't know the rule and didn't guess correctly, plus the probability that the student knew the rule but forgot it.

Equations 5.11 and 5.12 show the update of the corresponding other probability by calculating the complement of the updated probability.

5.5. SUMMARY

We asked three domain experts for characteristics that they used when judging the difficulty of an exercise for propositional logic. Most of the characteristics are not suitable for our purpose of the student model, as discussed in section 5.1. The general theme however is that how the rewrite rules should be applied is what makes an exercise harder. We decided to use the rules as the base for the student model since this can be tracked with each step that the students take.

As described in section 2.1.2, the choice to make an overlay model and use the rewrite rules is not a new idea. However, it will create a model that is intuitive, fairly simple to implement, and easy to explain to users and researchers.

5.5.1. MODEL DESIGN

Millán et al. [2010] describes a framework of questions and decisions that need to be taken when constructing a student model. We chose to use the Bayesian knowledge tracking model by Corbett and Anderson [1994] and alter it slightly by using two nodes to track each rule: an evidence variable capturing the interactions of the student, and a knowledge variable to estimate the knowledge level, and by adding a probability for 'forgetting' a learned rule.

The knowledge variables are influenced by the evidence variables. This makes sense when looking at how the system receives the information about a student: the system can only deduce something about a student when that student takes a step when working on an exercise.

The evidence variable for a rewrite rule is updated every five steps that a student takes using that rule. This is to avoid the system from overestimating the probabilities in the first few steps. The updates for the knowledge variables are done for each step a student takes. The conditional probabilities are updated according to the evidence the student provides: if the student takes an incorrect step, the conditional probabilities for this case are updated. The probabilities in the knowledge variables can directly be translated as confidence levels of the model on whether or not the student knows the rule and can correctly apply the rule.

The goal of the model is to classify the formulas into difficulty levels in LogEx. The classification is done in a node in the model that takes the knowledge variables as parents and, based on their average knowledge level and a function for each difficulty level, the outcome. The functions should have specific qualifications: for the 'Easy' difficulty level, the function should be monotonically increasing, since a higher knowledge level will lower the difficulty. The function for the 'Difficult' classification should be the opposite since a lower knowledge level will increase the difficulty, and the function for a 'Medium' classification needs to be a bell-shaped curve or concave curve. We have chosen to use functions instead of conditional probability tables since the conditional probability tables would become too large to calculate from a practical point of view.

As mentioned earlier, the structure of the model corresponds to the original Bayesian knowledge tracing model. This model is applied in many applications and is well understood. Our model changes it slightly by separating the evidence and the knowledge tracing nodes. As far as we know, this is not done before. However, we think that the distinction between evidence and knowledge levels of rules is beneficial since the probability of a slip is now captured in the conditional probabilities of the knowledge variable.

6

EXPERIMENTS AND RESULTS

We used LogEx in a live setting between the 26th of October and the 7th of November 2021 to record the students' interactions and get feedback on the difficulty of the exercises they worked on.

Our initial intention was to gather enough data to train student models for each student who used LogEx during this period. Then we could use the student model to generate a new exercise formula when the student requested a new formula with a certain difficulty level. Unfortunately, we did not get enough feedback to do this. Since we needed to adapt our approach, we used the data we could gather and did a brute-force search to find an optimal initialization of the parameters in the model.

This chapter describes the setting of the data-gathering session and the simulation. We will also report the performance of the model on the classifications of the formulas used in the live environment, and the students' feedback on the formulas.

6.1. DATA GATHERING

6.1.1. STUDENT PRIVACY

The student data was gathered with the privacy protection of the student in mind. However, to create a student model, we need to identify the students' interactions. As a consequence, we needed to know which exercise was done by which student and what the steps were that the students took while using LogEx.

In order to ensure the students' privacy but still identify the actions of students, we used a system that assigned a number to each student without us knowing the specific assignments of the numbers. All student who participated in the data collection session was given a unique number and a unique URL to access LogEx. The assignment of the unique numbers and URLs to the students was not stored. This allowed us to store the relevant information and associate this with a specific anonymous number, and track the steps of a student without the need to store personal data.

6.1.2. SETUP AND MODIFICATIONS TO LOGEX

The experiment aimed to verify if the difficulty classification worked as intended. Therefore, we needed feedback from the students to compare with the classification of the student model. Unfortunately, although LogEx has been used in live settings before, we did not

have data from previous sessions as this was not asked before, and no feedback is stored in the database of LogEx. As a such, we needed to gather new data before we could find the correct values for the parameters in the model.

LogEx contains three types of exercises: the DNF and CNF exercises, which we are interested in, and logical equivalence exercises. The interactions on the logical equivalence exercises were recorded but not used in the student models.

For each type of exercise, the students had a selection of several predefined exercises that they could work on, as well as the option to select a difficulty level. LogEx would then generate a formula that could be rewritten in a number of steps corresponding to the selected difficulty level. As mentioned earlier, we did not use the student models to classify and generate the formulas.

We added a feedback form to the LogEx web UI where the student could provide feedback on the exercise that she had just made. This form was shown when a student had finished a CNF or DNF exercise and only contained the question of how the student rated the exercise, as shown in figure 6.1. The feedback was then stored in the database of LogEx.

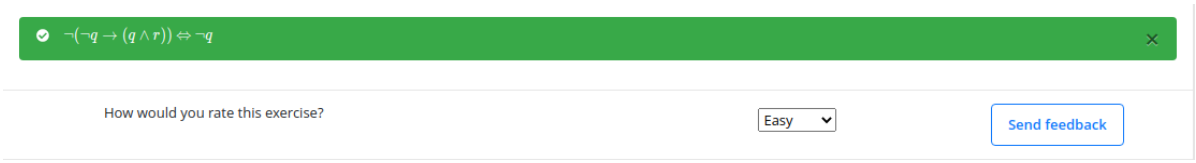


Figure 6.1: The feedback form was shown after a student finished an exercise.

The students used LogEx in an uncontrolled environment. They were given access to LogEx and were asked to use the system in their own time. We had no control over the mental state (level of concentration, tiredness, distracting factors) of the students, nor on the environment they used LogEx in.

LogEx logged the following interactions between the student and LogEx to the database:

- Request for a new exercise
- The diagnosis of a submitted rule
- A request for the next possible step (as hint)
- The check if the student is done
- A request for an example solution
- The feedback of the student, when provided by the student

6.1.3. Feedback results

In total, five students provided answers to 45 exercises. Table 6.1 shows a confusion matrix for the exercises for which the student have provided feedback.

Although 45 data points are not enough to make any solid claims, table 6.1 shows that, for this session at least, the perception of the student does not align well with the classification that LogEx gives to the exercises they have tried to solve.

Table 6.1: Confusion matrix for the selected difficulty and the feedback from the students. For the predefined exercises, the assigned difficulty level is taken into account as input difficulty.

		Feedback		
		Easy	Medium	Difficult
Selected difficulty	Easy	10	2	3
	Medium	8	4	8
	Difficult	3	1	6

We believe that one major factor for this difference could be attributed to the perception of the difficulty levels by the students and designers of LogEx. The difficulty level is calculated in LogEx by checking if the formula doesn't contain too many equivalences, doesn't require too many steps, and by counting the number of steps needed to rewrite the formula, regardless of what rules need to be applied in those steps. The boundaries between the difficulty levels are static.

However, when a student increases her knowledge, the exercises considered difficult in the past might now be regarded as easy because of the increase in knowledge. The designers of LogEx will assign a difficulty level to an exercise with the overall required level of the course in mind, while the student will answer the question with her own knowledge level as a frame of reference. As such, the difficulty level of an exercise as defined by the static rules in LogEx might not correspond with the perception of the student.

Another (related) factor is that an exercise might contain rewrite rules which are considered easy by a specific student that provided feedback, but in general, the rule is considered to be of medium or difficult level or vice versa. The designers of LogEx can not account for this variation in students and thus will try to make a classification that will align with most students. This discrepancy is one of the reasons why it is beneficial to implement a student model that can tailor to these kinds of variations.

6.2. Simulation

6.2.1. Classification formulas

After the feedback was gathered, a simulation was run where the student models were created based on the answers from the students. In the simulation, several values were applied to each of the probability parameters of the model. The values ranged from 0.1 to 1, with steps of 0.1. The classification functions (see equation 6.1) were kept the same for all student models.

$$\begin{aligned} f_{easy}(a) &= a \\ f_{med}(a) &= \sin(a \times \pi) \\ f_{diff}(a) &= 1 - a \end{aligned} \tag{6.1}$$

Figure 5.4 shows a plot with the curves of these functions. One thing to note is that the curve for the Medium class is dominating the other functions in a large portion of the domain. This means that most activation values will lead to a Medium classification.

For each formula that a student gave feedback on and for each combination of parameter values, a new student model was generated, and the formula was classified using that model. The student models were built using the steps that the student took in previous exercises. The decision to make a new student model for each exercise was taken to make the implementation somewhat easier.

6.2.2. CNF AND DNF ARE TREATED EQUALLY

To keep the complexity of the model low, the student models did not take into account whether the exercise was a DNF or CNF exercise. Although this would be possible, differentiating between the type of exercises would mean that we had fewer updates to the student model and less feedback.

A more profound reason for not differentiating between DNF and CNF exercises in the student model is that the student model is based on the rewriting rules and the ability of the student to apply them correctly. The classification was done using the default derivation of the exercise generated by LogEx. The default derivation depends on the type of exercise since, usually, there are different steps used to write a formula to CNF or DNF. However, a rule is applied on a step level (in each step, one or more rules must be used) and to a certain structure of the formula. Which rule must be applied in a step might differ between the different types of exercises, but that is not relevant to the student model.

6.2.3. PENALTY FUNCTION

The classification was then compared to the feedback that the student gave for the exercise, and a penalty was assigned according to table 6.2. Summing over the absolute values of the penalties gives an overview of which parameters give the least errors overall.

Table 6.2: The penalties assigned to the classification of the student model. We use negative scores to keep the information about the difference of classification in the scores. This might be useful for analytical purposes. However, we have not used this and could have used positive penalties instead.

Classification	Student feedback	Penalty
Easy	Easy	0
Easy	Medium	-1
Easy	Difficult	-2
Medium	Easy	+1
Medium	Medium	0
Medium	Difficult	-1
Difficult	Easy	+2
Difficult	Medium	+1
Difficult	Difficult	0

Because of the lack of data points, we did not train the model with a training set and validation set of data. Instead, to find the best possible values to initialize the student models with, we performed a brute-force search for the parameter values and the associated total error on the classifications with the values. Table 6.3 shows the top 10 combinations of initialization values with the lowest sum of absolute penalties.

Table 6.3: Top 10 combinations with the lowest absolute penalty over 45 exercises.

$p_r^0(t)$	$p(f)$	$p_r^0(e)$	$p_r^0(k \mid e)$	$p_r^0(\neg k \mid \neg e)$	Penalty
0.9	0.1	0.4	0.9	0.3	30
0.9	0.2	0.4	1	0.3	30
0.8	0.1	0.4	1	0.4	31
0.8	0.2	0.4	1	0.4	31
0.9	0.1	0.4	1	0.3	31
0.9	0.2	0.4	0.9	0.3	31
0.9	0.5	0.4	0.9	0.3	31
0.9	0.5	0.4	1	0.3	31
1	0.1	0.4	1	0.3	31
0.7	0.1	0.4	1	0.3	32

6.3. Performance of the models

Since the classification of the exercises is a multiclass classification, we show a confusion matrix to get an intuitive overview of the performance. As mentioned previously, we only have 45 data points, some of which are about the same exercise (but from different students). This is mainly because the updates of the probabilities for the evidence variables happen after five applications of the associated rule. As such, the results will not provide large confidence, and these results will be overfitted to the available data.

Although each exercise contains multiple rule applications, most students only applied a subset of the rules frequently, and only a few of the rules were applied often enough to get updates. Table 6.4 shows the number of times a rule has been applied in a step. These numbers are aggregated for the CNF and DNF exercises and for the students. This means, for example, that the probabilities for the Commutativity rule are not updated in most of the student models, as this rule has only been applied 18 times in all exercises made by all students.

Compared with students' feedback

Table 6.5 shows the confusion matrix of the classifications compared with the student's feedback. Most exercises are classified with Medium difficulty by the student models, while the students classify only 7 of the exercises as Medium. As such, the model's accuracy is very low: the model is correct in one-third of all the classifications.

As discussed earlier, all the knowledge variables are initialized with the same probability, and all the evidence variables are initialized with the same probability.

However, table 6.6 shows that is better to initialize the rules with a specific probability per rule, as not all rules are applied correctly with the same probability. For example, the Commutativity rule has been applied 18 times, but it has been applied incorrectly every time it was applied during the experiment session. The rule for double negation or the equivalence rule were applied correctly in most cases. Although we should not instantiate the Commutativity rule with a probability of 0, this shows that not all rules are equally difficult.

The functions in the classification node could also be changed to influence the intersections of these functions. This will alter the points where a classification changes, given

Table 6.4: The count of the application of a specific rule in a single step.

Rule name	Number of steps
De Morgan	99
Distribution	78
Implication	72
Double negation	64
T-rule complement	30
T-rule conjunction	26
F-rule disjunction	24
T-rule disjunction	23
F-rule complement	21
F-rule conjunction	20
Absorption	20
Idempotency	20
Commutativity	18
Equivalence	17
Association	17
Not true	9
Not false	5
Total:	563

Table 6.5: Confusion matrix for the classification of the student models and the feedback from the students. For the predefined exercises, the assigned difficulty level is taken into account as input difficulty.

		Classified difficulty		
		Easy	Medium	Difficult
Feedback	Easy	7	14	0
	Medium	0	7	0
	Difficult	2	12	3

a certain input value. If you change these intersection points, the probability of a classification will change.

Compared with requested level

Table 6.7 shows the confusion matrix of the classifications compared with the level of the exercise as selected by the student. Not surprisingly, the accuracy is somewhat better here. This is mainly because most generated exercises were classified as Medium by LogEx .

Compared with the expert classification

We have used the student model from the student who submitted the most feedback to classify the formulas that were given to the domain experts and compared them with the classification of the domain experts. Since the domain experts classified the formulas on a scale from 1 to 10, we translated those classifications by assigning a score of 3 or lower to Easy, 4 to 7 to Medium, and 8 or higher to Difficult.

Table 6.6: All evidence nodes are initialized with the same probability for a correct application of the rule associated with the node. As can be seen in this table, this is not a valid assumption. It would make sense to initialize the probabilities with an average score, and let the updates of the model personalize these probabilities.

Rule	Correct	Total	Percentage
Absorption	12	20	60.00
Association	9	17	52.94
Commutativity	0	18	0.00
De Morgan	48	99	48.48
Distribution	27	78	34.62
Double negation	57	64	89.06
Equivalence	13	17	76.47
F-rule complement	12	21	57.14
F-rule conjunction	9	20	45.00
F-rule disjunction	19	26	73.08
Idempotency	13	20	65.00
Implication-elimination	57	72	79.17
Not false	2	5	40.00
Not true	6	9	66.67
T-rule complement	19	30	63.33
T-rule conjunction	11	24	45.83
T-rule disjunction	10	23	43.48

Table 6.7: Confusion matrix for the classification of the student models and the difficulty according to the difficulty calculation in LogEx.

		Classified difficulty		
		Easy	Medium	Difficult
Original difficulty	Easy	4	11	0
	Medium	3	17	0
	Difficult	2	5	3

Table 6.8 shows that the student model classified most of the formulas as Medium again. Note that we have 45 formulas (three sets of 15 formulas), but 90 classifications here since each formula is classified as both an DNF and CNF exercise. Note as well that the set of formulas given to the domain experts is not the same as the formulas that were given to the students in the exercises.

6.4. Summary

We held a session with students to get feedback on the exercises in LogEx. We added a form to LogEx which was shown after a student finished an exercise, where the students could provide their judgment on the difficulty level of the exercise. We brute-force searched for the best parameter values to initialize the student model by comparing the difference of the classification of the student model with the feedback of the student. We assigned a

Table 6.8: Confusion matrix for the classification of the student models and the classification of the domain experts.

		Classified difficulty		
		Easy	Medium	Difficult
Expert's difficulty	Easy	15	14	1
	Medium	4	50	0
	Difficult	0	6	0

penalty to the differences: a difference of 2 levels resulted in more penalty points (2) than a difference of 1 level (1).

This allowed us to pick the optimal parameters and classify the exercises for which we received the student's feedback, using the student model. The students provided feedback for only 45 exercises. This does not allow us to make any definitive claims.

Since not a lot of exercises were made during the session and not so many steps were taken, the updates of the student model were not frequent enough. This resulted in too many Medium classifications by the student models, compared to what the feedback was of the students. Out of 45 exercises, the students found 21 Easy, 7 Medium and 17 Difficult, but the student models classified 9 as Easy, 33 as Medium and 3 as Difficult. The same pattern is visible when comparing with the chosen difficulty level when starting the exercise. Here, 15 exercises were requested as Easy, 20 as Medium and 10 as Difficult.

Another reason for the large amount of Medium classifications is the fact that the classification formulas used in the experiment are favouring the Medium class. This should be changed in future exeriments.

We asked the domain experts to classifiy 45 randomly generated exercises, and compared their judgment on these exercises with the classification of the student model of the student that provided the most feedback. We see that the classifications of the student model are more in line with the classifications of the domain experts, but the model is still classifying too many exercises as Medium. The exercises were classified for both DNF and CNF, which resulted in 90 classifications. The domain experts judged that 30 were Easy, 54 were Medium and 6 were classified as Difficult. The student model classified 19 as Easy, 70 as Medium and 1 as Difficult.

To say anything about the performance, the student model must be used in longer sessions, and the students must provide more feedback, so we can track the updates in the models better.

7

Conclusion, Threats to validity and future research

7.1. Threats to validity

7.1.1. Student must pick the correct rule to continue

As explained in section 5.4, LogEx will allow a student only to continue when it provides a correct answer. Whenever a student provides a wrong step, the student must try again. This means that if a student finishes an exercise and provided all answers by herself, for each wrong step taken by the student, a correct step has been taken as well. Although we do not think that this is a problem as we assume that a student learns from this, it might be good to account for this. In our model, the update for the probabilities will be proportional to the number of times the student took that step. This might not necessarily be correct. A student also can let LogEx take the next step. In that case, there will be no update for the correct step, but the student might have learned a bit, without any trace of this in our model.

7.1.2. Lack of participating students

We only held one experiment session, and our results are not generalizable. As discussed earlier, there were not a lot of students participating in the live session, and we did not retrieve a lot of feedback. This makes it impossible to draw any conclusions about whether the student model is capable of correctly predicting the difficulty level, and we can only give an indication of how well it works.

We also had our experiment with a homogenous group of students, as the students were all participating in the same course on propositional logic.

7.1.3. No knowledge about the students

We had no knowledge about the students participating in the experiment. This might influence our results in two different ways:

- We have no knowledge about the prior knowledge of the students. We can not tell if, for example, one student had several courses on propositional logic, and other students did not. The prior knowledge will influence the feedback of the student, and therefore the student model. However, since the group of students that participated

in the experiment is taken from an introduction course in proportional logic, we assumed no prior knowledge.

- We have limited knowledge about how the student uses LogEx. We can see all interactions of the student with LogEx, including if the student asks for help, but we can not see the help that the student gets from external sources. This also might influence the experience of the student and the feedback. If the student gets help from another student and applies the rules correctly because of the help from other students, then our model will predict that the student has good knowledge of the corresponding rules. However, we can question if that is really the case. However, this is a general issue with interaction-based models, and not specific for our model.

7.2. Research conclusion

We investigated the use of a student model in a live ITS for propositional logic and tried to automatically classify the difficulty levels of exercises for this ITS. Our initial problem was to find characteristics that can be used in the student model and track the knowledge level of the students. We described the implementation details of the model in LogEx and we used the model to classify the exercises that were done by students in an experiment.

7.2.1. Q1: Model characteristics

We have chosen to use a Bayesian Knowledge Tracing model in an overlay model, but in chapter 2 we have summarized more different structures of models that can be used to implement a student model. Since we use an overlay structure, our model is fairly simple. It can be extended by modeling more characteristics as proposed by the domain experts (see 7.3.3), but this will add to the complexity of the model. Since our model only contains dynamic variables, the student model can be improved by adding a mix of static and dynamic variables, as described in section 2.1.1.

7.2.2. Q2: Student model implementation

We have defined a model that can estimate the individual knowledge level per rule for the students using LogEx. There are situations where we want to improve on the granularity of the model, as we expect that a more complex model will work better to distinguish certain situations. For instance, we think that for certain rules, applying a rule on a subformula is easier than applying a rule on a compound formula.

We also have made some assumptions that can be improved upon. In our classification node, we take the set of all rules to calculate the average activation value of an exercise. This will cancel out multiple applications of the same rule in an exercise. If an exercise consists of applying the same rule multiple times, we can imagine that the student will find the exercise more difficult, or easier, depending on whether or not she finds that particular rule more difficult or easier.

The classification node uses functions that determine the classification of the exercise, based on the average activation for each rule. This is not entirely how normal BKT models work, but we think that it works relatively well and is easy to understand. Furthermore, it is easy to tweak the functions for each class, so that the total classification function is altered.

7.2.3. Q3: Performance

We have not received enough data to find any definitive conclusion on how well the model works, and we need to use the model for a longer period, or with more students to see the actual performance. For now, we can see that the model works in the sense that it updates the probabilities, and can distinguish between the different classes. We suggest that in future research a couple of parameters should be tweaked to get better results:

- The classification functions for the classification node need to be more equal. That is, they should divide the domain more evenly, or divide them differently.

- In our model, the initial probabilities for students to know a rule are the same for all the rules. The aggregated values in table 6.6 show that there is a variation in the percentage of correct applications per rule. This can be directly translated to the initial probabilities. A remark to this is that the figures in table 6.6 show all applications of the rules, and not only the initial applications of a rule. If we have more data, we can further refine these numbers so that they only take the initial applications into account. Since we only have five students in our data set, we found that showing the percentages of the initial applications did not add too much information.

- Currently we keep the same amount of applications in memory for all the rules. We think that it might be worthwhile to check if for certain rules we can keep a longer list of applications in memory, so we can get more refined probabilities.

7.3. Future research

7.3.1. Buggy rules

Next to the rewrite rules, LogEx contains a library of buggy rewrite rules. These buggy rules are used to recognize the wrong applications of a rule and to help the student pick the correct rule. Our model does not take this into account, but these buggy rules might be used to make a more balanced judgment on the knowledge level of the student. For instance, we might update the rules that are closely related to a buggy rule.

7.3.2. Model functions

Our model used very simple functions for the classification variable. As discussed in section 6.3 the functions can be altered to influence the probability of a certain classification. The update functions for the knowledge variables could also be changed, as well as the fixed probabilities for forgets and transition from not knowing to knowing.

7.3.3. Expand the levels of knowledge nodes

The model can be extended with more levels for the knowledge nodes by grouping the rules into layers. This will probably complicate the model a bit, but this will make it possible to define probabilities for similar rewrite rules.

Another way to extend the model is to add more characteristics to the knowledge level nodes and to model the characteristics as described by the domain experts together with the knowledge levels as currently defined. As an example, the model does not take into account whether a rule is applied on a (sub)formula that contains more subformula, or on a

(sub)formula that purely consists of atomic propositions. This might influence the knowledge level, presuming that applying a rewrite rule on a formula containing subformulas is harder than applying the rule on formulas containing only atomic propositions.

7.3.4. Grouping of rewrite rules

As we discussed in section 5.3.4, the complexity of the model can be reduced by grouping rewrite rules in a single variable. When the grouped variable is used, the updates of the probabilities in that variable will, in general, be more frequent than when the rules are not grouped. We expect that such a model will be more accurate in predicting the difficulty of exercises that use rules that aren't often used.

7.3.5. Change the activation calculation in the classification variable

As discussed in section 5.3.5, we do not take into account if a rule is applied more than once in an exercise. This means that each rule contributes in the same proportion to the activation values for the difficulty levels in the classification variable.

There are several ways to change this:

- Don't take the set of rules into account but calculate the activation based on the list of rules for an exercise.
- Define weights for all the rules, as some rules are easier than others. An easy rule will contribute more to an Easy classification and a hard rule will contribute more to a Difficult classification.
- As a variant of the previous method: calculate the weights dynamically based on the knowledge levels of the rules that must be applied. This means that each student will have their own weights, in contrast to the previous method where the weights are static and general for all students.

7.3.6. Use the student models to generate exercises

The initial idea of our study was to measure if the student models could be used to generate exercises so that the student would find the exercises challenging enough and would be motivated to use LogEx to increase their knowledge level. However, we did not use the model as such, and only used the classification 'after the fact'.

One application of using the model online to generate exercises is that we think it would be interesting to look into the motivation of students using LogEx where the exercises are generated using the current static rules and compare the motivation of students interacting with a version of LogEx where the exercises are generated based on the student models.

BIBLIOGRAPHY

Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. 2013. Automatically generating problems and solutions for natural deduction. In *In IJCAI*. 1968–1975. 5, 7

Steven P. Allscheid and Douglas F. Cellar. 1996. An interactive approach to work motivation: The effects of competition, rewards, and goal difficulty on task performance. *Journal of Business and Psychology* 11, 2 (1996), 219–237. ISBN: 0889-3268 Publisher: Springer. 2

Erik Andersen, Sumit Gulwani, and Zoran Popovic. 2013. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. Association for Computing Machinery, New York, NY, USA, 773–782. https://doi.org/10.1145/2470654.2470764 5

Konstantina Chrysafiadi and Maria Virvou. 2013. Student modeling approaches: A literature review for the last decade. *Expert Systems with Applications* 40, 11 (Sept. 2013), 4715–4729. https://doi.org/10.1016/j.eswa.2013.02.007 3, 4

Allan Collins and Ryszard Michalski. 1989. The Logic of Plausible Reasoning: A Core Theory. *Cognitive Science* 13, 1 (1989), 1–49. https://doi.org/10.1207/s15516709cog1301_1 5

Albert T. Corbett and John R. Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction* 4, 4 (Dec. 1994), 253–278. https://doi.org/10.1007/BF01099821 iii, 5, 6, 11, 17, 22, 23, 27, 28

Ilja Cornelisz and Chris van Klaveren. 2018. Student engagement with computerized practising: Ability, task value, and difficulty perceptions. *Journal of Computer Assisted Learning* 34, 6 (2018), 828–842. https://doi.org/10.1111/jcal.12292 2

Ludmila Dostálová and Jaroslav Lang. 2007. ORGANON — The Web Tutor for Basic Logic Courses. *Logic Journal of the IGPL* 15, 4 (Aug. 2007), 305–311. https://doi.org/10.1093/jigpal/jzm021 2, 7

Arno Ehle, Norbert Hundeshagen, and Martin Lange. 2018. The Sequent Calculus Trainer with Automated Reasoning - Helping Students to Find Proofs. *Electronic Proceedings in Theoretical Computer Science* 267 (March 2018), 19–37. https://doi.org/10.4204/EPTCS.267.2 8

Jacob Feldman. 2003. A catalog of Boolean concepts. *Journal of Mathematical Psychology* 47, 1 (Feb. 2003), 75–89. https://doi.org/10.1016/S0022-2496(02)00025-1 9

Cristiano Galafassi, Fabiane F. P. Galafassi, Eliseo B. Reategui, and Rosa M. Vicari. 2020. EvoLogic: Intelligent Tutoring System to Teach Logic. In *Intelligent Systems (Lecture Notes in Computer Science)*, Ricardo Cerri and Ronaldo C. Prati (Eds.). Springer International Publishing, Cham, 110–121. https://doi.org/10.1007/978-3-030-61377-8_8 7

Fabiane F. P. Galafassi, Cristiano Galafassi, Rosa Maria Vicari, and João Carlos Gluz. 2019. Identifying Knowledge from the Application of Natural Deduction Rules in Propositional Logic. In *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection (Lecture Notes in Computer Science)*, Yves Demazeau, Eric Matson, Juan Manuel Corchado, and Fernando De la Prieta (Eds.). Springer International Publishing, Cham, 66–77. https://doi.org/10.1007/978-3-030-24209-1_6 6

João Carlos Gluz, Fabiane Penteado, Marcel Mossmann, Lucas Gomes, and Rosa Vicari. 2014. A Student Model for Teaching Natural Deduction Based on a Prover That Mimics Student Reasoning. In *Intelligent Tutoring Systems (Lecture Notes in Computer Science)*, Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha Crosby, and Kitty Panourgia (Eds.). Springer International Publishing, Cham, 482–489. https://doi.org/10.1007/978-3-319-07221-0_60 2, 6

Guangbing Yang, Kinshuk, and Sabine Graf. 2010. A practical student model for a location-aware and context-sensitive Personalized Adaptive Learning System. In *2010 International Conference on Technology for Education*. IEEE, Mumbai, India, 130–133. https://doi.org/10.1109/T4E.2010.5550048 4

Z. Jeremić, J. Jovanović, and D. Gašević. 2012. Student modeling and assessment in intelligent tutoring of software patterns. *Expert Systems with Applications* 39, 1 (Jan. 2012), 210–222. https://doi.org/10.1016/j.eswa.2011.07.010 4

Josje Lodder, Bastiaan Heeren, and Johan Jeuring. 2016. A domain reasoner for propositional logic. *Journal of Universal Computer Science* 22, 8 (Jan. 2016), 1097–1122. 6, 8

Dino Mandrioli. 1982. On teaching theoretical foundations of computer science. *ACM SIGACT News* 14, 3 (July 1982), 36–53. https://doi.org/10.1145/990511.990516 8

Eva Millán, Tomasz Loboda, and Jose Luis Pérez-de-la Cruz. 2010. Bayesian networks for student model engineering. *Computers & Education* 55, 4 (Dec. 2010), 1663–1683. https://doi.org/10.1016/j.compedu.2010.07.010 5, 11, 14, 18, 19, 28

Antonija Mitrovic, Kenneth R. Koedinger, and Brent Martin. 2003. A comparative analysis of cognitive tutoring and constraint-based modeling. In *User Modeling 2003: 9th International Conference, UM 2003 Johnstown, PA, USA, June 22–26, 2003 Proceedings 9*. Springer, 313–322. 5

Behrooz Mostafavi and Tiffany Barnes. 2017. Evolution of an Intelligent Deductive Logic Tutor Using Data-Driven Elements. *International Journal of Artificial Intelligence in Education* 27, 1 (March 2017), 5–36. https://doi.org/10.1007/s40593-016-0112-1 6

Behrooz Mostafavi, Michael Eagle, and Tiffany Barnes. 2015. Towards data-driven mastery learning. In *Proceedings of the Fifth International Conference on Learning Analytics And Knowledge - LAK'15*. ACM Press, Poughkeepsie, New York, 270–274. https://doi.org/10.1145/2723576.2723622 2, 6

Eleanor O'Rourke, Eric Butler, Armando Díaz Tolentino, and Zoran Popović. 2019. Automatic Generation of Problems and Explanations for an Intelligent Algebra Tutor. In *Artificial Intelligence in Education (Lecture Notes in Computer Science)*, Seiji Isotani, Eva Mil-

lán, Amy Ogan, Peter Hastings, Bruce McLaren, and Rose Luckin (Eds.). Springer International Publishing, Cham, 383–395. https://doi.org/10.1007/978-3-030-23204-7_32 5

Judea Pearl. 1985. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, CA, USA*. 15–17. 13, 14

Hans van de Pol. 2010. *Berekenen moeilijkheidsgraaf van algebraïsche opgaven*. Bachelor thesis. Open Universiteit of the Netherlands. 9

Ferran Prados, Imma Boada, Josep Soler, and Jordi Poch. 2005. Automatic generation and correction of technical exercises. In *International conference on engineering and computer education: Icece*, Vol. 5. http://acme.udg.es/articles/ICECE2005.pdf 5

Rein Prank. 2014. A tool for evaluating solution economy of algebraic transformations. *Journal of Symbolic Computation* 61-62 (Feb. 2014), 100–115. https://doi.org/10.1016/j.jsc.2013.10.014 6

Sriram Rajamani, Sumit Gulwani, and Sriram Rajamani. 2012. Automatically Generating Algebra Problems. In *AAAI*. Microsoft Research. https://www.microsoft.com/en-us/research/publication/automatically-generating-algebra-problems/ 5

Rod Rivers. 1989. Embedded user models —where next? *Interacting with Computers* 1, 1 (April 1989), 13–30. https://doi.org/10.1016/0953-5438(89)90004-0 4

Stuart Russell and Peter Norvig. 2014. *Artificial Intelligence: A Modern Approach* (third ed.). Pearson Education Limited. 14

Raymund Sison and Masamichi Shimura. 1998. Student Modeling and Machine Learning. *International Journal of Artificial Intelligence in Education* 9 (1998), 128–158. https://telearn.archives-ouvertes.fr/hal-00257111 3

Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games (FDG '12)*. Association for Computing Machinery, New York, NY, USA, 156–163. https://doi.org/10.1145/2282338.2282370 5

David R. Surma. 2012. Rolling the dice on theory: one department's decision to strengthen the theoretical computing aspect of their curriculum. *Journal of Computing Sciences in Colleges* 28, 1 (Oct. 2012), 74–80. 8

Kurt VanLehn. 2006. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education* 16, 3 (Jan. 2006), 227–265. https://content.iospress.com/articles/international-journal-of-artificial-intelligence-in-education/jai16-3-02 Publisher: IOS Press. 1, 3

Maria Virvou and Katerina Kabassi. 2002. F-SMILE: An intelligent multi-agent learning environment. In *Proceedings of 2002 IEEE International Conference on Advanced Learning Technologies-ICALT*. Citeseer, 144–149. 5

Laura Zavala and Benito Mendoza. 2018. On the Use of Semantic-Based AIG to Automatically Generate Programming Exercises. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, Baltimore Maryland USA, 14–19. https://doi.org/10.1145/3159450.3159608 5

A

REQUEST LOG RECORD

This is a brief description of the attributes in the requests database table.

1. service: The service that was called
2. source: The environment that made the request
3. requestinfo: additional information used by a service
4. userid: the ID of the user that made the request
5. sessionid: each exercise session started by a student receives a unique identifier
6. taskid: the formula that needs to be proved equivalent
7. time: the timestamp of the request
8. responsetime: the time the service took to handle the request
9. errormsg: the error message that the service returns, when applicable
10. serviceinfo: the result of the service call. This might also show the applied rule.
11. ruleid: the rule that has been applied in the request
12. input: a JSON object containing the input parameters of the service
13. output: a JSON object containing information about the result of the service call. This includes the result of the service call.

B

Complexity scores for formula derivations to DNF and CNF

Table B.1: The complexity score based on the number of steps to rewrite a formula into DNF or CNF

Number of rules	Complexity
0 - 4	1
5, 6	2
7, 8	3
9, 10	4
11, 12	5
13, 14	6
15, 16	7
> 16	8

Table B.2: The complexity score based on the complexity of the minimal equivalent formula.

Number of rules	Complexity
0 - 4	1
5 - 7	2
8 - 10	3
11 - 13	4
14 - 16	5
17 - 19	6
20 - 22	7
> 22	8

Table B.3: The complexity score of the rewrite rules in LogEx.

Rule name	Number of steps
De Morgan	2
Distribution	1
Implication	2
Double negation	1
T-rule complement	1
T-rule conjunction	1
F-rule disjunction	1
T-rule disjunction	1
F-rule complement	1
F-rule conjunction	1
Absorption	1
Idempotency	1
Commutativity	1
Equivalence	1
Association	1
Not true	1
Not false	1

C

CODE SNIPPETS

This appendix contains interesting Haskell code snippets used in the student model. Readers that are interested in the entire body of code can contact the authors or the Open Universiteit.

Listing C.1: Update of the knowledge variable

```
-- | Update the prior probabilities of an @KnowledgeVariable@ given a step
updateKnowledgeVar :: (Ord a, Fractional a) => Variable a -> Step b -> Variable a
updateKnowledgeVar var@(KnowledgeVariable _ cpt parent probTransition probForget) step = var{cpt=cpt', parent=parent'}
    where
        -- | the probabililty that the student knows the rule applied in the step: p(know = True)
        probKnow    = prior (var, 1.0)
        probNotKnow = 1.0 - probKnow

        -- | the probability of the student making a mistake while knowing: p(evidence = False | know = True)
        probSlip  = posterior (parent, 0.0) (var, 1.0)

        -- | the probability of the student answering correctly while not knowing: p(evidence = True | know = False)
        probGuess = posterior (parent, 1.0) (var, 0.0)

        -- | the probability of the student knowing the rule given that she answers correctly: p(know = True | evidence = True)
        probKnowEv          = posterior (var, 1.0) (parent, 1.0)

        -- | the probability of the student not knowing given that she answers incorrectly: p(know = False | evidence = False)
        probNotKnowFalseEv  = posterior (var, 0.0) (parent, 0.0)

        probKnowEv'         = updateKnowProb step
        probNotKnowFalseEv' = updateNotKnowProb step

        cpt' = updateProbability   0.0 (1.0 - probKnowEv')         [(parent, 1.0)] -- p(know = False | evidence = True)
               $ updateProbability 1.0 probKnowEv'                 [(parent, 1.0)] -- p(know = True | evidence = True)
               $ updateProbability 0.0 probNotKnowFalseEv'         [(parent, 0.0)] -- p(know = False | evidence = False)
               $ updateProbability 1.0 (1.0 - probNotKnowFalseEv') [(parent, 0.0)] -- p(know = True | evidence = False)
                 cpt

        parent' = update parent step

        -- | Calculates p_(i+1)(know = True | evidence = True) based on:
        --     - p(know = True | evidence = True) * (1.0 - probSlip): The probability of the student
        --       knowing the rule given that she answers correctly and she doesn't make a mistake
        --     - and p(know = False | evidence = True) * probTransition: the probability that the student
        --       doesn't know the rule but answers correctly and transitions to knowing the rule
        updateKnowProb :: Step a -> Probability
        updateKnowProb step | correct step = probKnowEv * (1.0 - probSlip) + (1.0 - probKnowEv) * probTransition
                            | otherwise    = posterior (var, 1.0) (parent, 1.0)
        -- | Calculates p_(i+1)(know = False | evidence = False) based on:
        --     - p(know = False | evidence = False) * (1.0 - probGuess): The probability of the student
        --       not knowing the rule given that she answers incorrectly and she doesn't guess correctly
        --     - and p(know = True | evidence = False) * probForget: the probability that the student
        --       knows the rule but answers incorrectly and transitions to not knowing the rule
        updateNotKnowProb :: Step a -> Probability
        updateNotKnowProb step | not.correct $ step = probNotKnowFalseEv * (1.0 - probGuess) + (1.0 - probNotKnowFalseEv) * probForget
                               | otherwise          = posterior (var, 0.0) (parent, 0.0)

updateKnowledgeVar var _ = var
```

Listing C.2: Update of the evidence variable

```
-- | Update the prior probabilities of an @EvidenceVariable@ given a step
updateEvidenceVar :: (Ord a, Fractional a) => Variable a -> Step b -> Variable a
updateEvidenceVar var@(EvidenceVariable _ cpt applied) step = var{cpt=cpt', applied=applied'}
    where
        -- | add the step and retain the last 20 results
        applied'= take 20 $ correct step : applied

        -- Update the probabilities if there are 5 more pieces of evidence
        probCorrect = if length applied' `rem` 5 == 0
                      then (fromIntegral . length . filter id $ applied') / (fromIntegral . length $ applied')
                      else prior (var, 1.0)
        probs = Map.fromList [(0.0, 1.0-probCorrect), (1.0, probCorrect)]
        cpt' = [newCPTRow probs]

updateEvidenceVar var _ = var
```

Listing C.3: Retrieval of the prior and posteriors in the student model

```
-- ------------------------------------------
--      Assignment
-- ------------------------------------------
-- | An @Assignment@ is a fixation of the
--   value of a @Variable@
type Assignment a = (Variable a, a)
type Condition a = Assignment a -- This is only to not get confused and more of a semantic thing ...

-- | Gives the prior for a single condition: p(var=value)
prior :: (Ord a, Fractional a) => Assignment a -> Probability
prior (EvidenceVariable _ cpt _, val) = getProbability val . head $ cpt
prior (KnowledgeVariable _ cpt parent _ _, val) = probKnowEv * probEv + probKnowEv' * probEv'
    -- p(k=T) = p(k=T & e=T) + p(k=T & e=F) = p(k=T | e=T)p(e=T) + p(k=T|e=F)p(e=F)
    where
        val' = 1.0 - val
        rowEv = selectCPTRow [(parent, val)] cpt
        rowEv' = selectCPTRow [(parent, val')] cpt

        -- | p(know = val | evidence = val)
        probKnowEv = getProbability val rowEv

        -- | p(evidence = val)
        probEv = prior (parent, val)

        -- | p(know = val | evidence = not val)
        probKnowEv' = getProbability val rowEv'

        -- | p(evidence = not val)
        probEv' = prior (parent, val')

posterior :: (Ord a, Fractional a) => Assignment a -> Condition a -> Probability
-- | Evidence variables are only influenced by their child variables. That is: p(E=T|K=T) = p(E=T)p(K=T) if
--   K is *not* a child of E, because of the markov blanket.
posterior q@(EvidenceVariable {}, val) cnd@(KnowledgeVariable {}, val') =
    let
        ev = fst q
    in (prior q * posterior cnd q) / (prior cnd + prior (not cnd))
    where
        not as = Bifunctor.second (1.0 -) as

posterior as@(KnowledgeVariable _ cpt parent _ _, val) cnd@(EvidenceVariable {}, val') =
    getProbability val $ selectCPTRow [(parent, val')] cpt
-- ^ The probability p(K=T|E=T) is defined in the CPT for the variable K if E is the parent variable of K.
--   So, any assigment of K, given that E is the parent, can be taken directly from the CPT

posterior as cnd = prior as * prior cnd
-- ^ If both variables have the same 'level', then the variables are
--   independent due to the structure of the network.
```

Listing C.4: Datastructures used in the student model

```
-- ------------------------------------------------
--       Variable
-- ------------------------------------------------

-- | A @Variable@ models a random variable in the bayesian network
--   and holds a conditional probability table with the probability
--   distribution of the variable, as well as a list of pointers
--   to the parent variables
data Variable a
    = KnowledgeVariable {
        -- | A label to identify the variable. This is taken from the rule
        label :: Id
        -- | The conditional probability table for the variable
        , cpt :: CPT a
        -- | A parent variables
        , parent :: Variable a
        , transition :: Probability
        , forget :: Probability
    }
    | EvidenceVariable {
        -- | A label to identify the variable. This is taken from the rule
        label :: Id
        -- | The conditional probability table for the variable
        , cpt :: CPT a
        -- | A list of boolean values indicating if the applications of the rule for which
        --   this variable is where correct
        , applied :: [Bool]
    }

data Step a = Step {
    appliedRule :: Id
    , correct :: Bool
 }

type StudentHistory a = Map.Map Id [Step a]

data Student a = Student {
    -- | Student id
    sId :: Id
    -- | A map of id for rewriting rules to steps
    , history :: StudentHistory a
}

data StudentModel a = Model {
    student :: Student a
    , graph :: BayesianNetwork Probability
    , parameters :: Parameters
}

data Parameters = Parameters {
    probInitEvidence :: Probability,
    probInitKnowledge :: (Probability, Probability),
    probTransition :: Probability,
    probForget :: Probability,
    fEasy :: Double -> Double,
    fMed :: Double -> Double,
    fDiff :: Double -> Double
}

data DAG a = DAG {
    nodeSet :: [Node a],
    edgeSet :: [Edge a]
}

type BayesianNetwork a = DAG (Variable a)

newtype Node v = Node v deriving (Generic)
data Edge a = DirectedEdge {
    parent :: Node a
    , child :: Node a
} deriving Eq
```