Student Modelling
Using a
Genetic Algorithm

Edo Plantinga
Student number s1007017
13 May 2003

External research project at the
School of Information Technologies
University of Sydney, Australia

Supervisors:
Dr. Josiah Poon (University of Sydney)
Dr. Rineke Verbrugge (University of Groningen)
Dr. Niels Taatgen (University of Groningen)

Artificial Intelligence
University of Groningen, The Netherlands

Table of contents

Acknowledgements

A number of people have helped me complete this thesis. First of all, I would like to thank my supervisor in Australia, Dr. Josiah Poon, for all the help, support and time he gave me during my time at the University of Sydney. This thesis would definitely not have been possible without him. At the University of Groningen, The Netherlands, I would like to thank my supervisors Dr. Niels Taatgen and Dr. Rineke Verbrugge for all their help and feedback, and Geertje Zwarts and Judith Grob for proofreading this thesis. I would like to thank my parents for supporting me during my stay in Sydney. Finally, I would especially like to thank Claire Yeo, for making my stay in Sydney so enjoyable (I couldn't 'of done it without you).

Abstract

The focus of the research described in this thesis is on modelling a student's knowledge and skills in an Intelligent Tutoring System (ITS) using a genetic algorithm (GA). ITSs are computer-based teaching systems that adapt the material that is taught to the level of the student. This means that the system will present the student hints, exercises, questions, etc. that are expected to be most beneficial for this particular student. In order to do this it is necessary to keep track of a student model: a set of beliefs about the knowledge and skills of a student.

The key observation that inspired this research was that students often give answers that can be interpreted in several ways. It is not always possible to translate an observation of the student to a direct update of the student model because of this ambiguity. Human tutors do not seem to have a lot of difficulties with this problem: apparently their beliefs about a student can be updated flexibly enough to avoid serious misconceptions.

The technique that we propose in this thesis is a novel way of handling this problem of ambiguity. Rather than choosing one interpretation of an observation and discarding the other possible interpretations (or making them less likely), we chose to implement a system in which all possible interpretations are considered simultaneously. The student models that have interpreted the observations correctly will generally be able to make better predictions about the next answer of the student. Every student model has a fitness parameter to signify how good the predictions of this student model have been in the past. The fitter student models are more likely to be retained as acceptable hypotheses of the student's knowledge and skills, whereas the less fit student models are more likely to be discarded. In this way student models that can interpret the observations well will evolve. Because of the similarities with the biological process of natural selection this approach is referred to as a genetic algorithm approach.

We have implemented a simplified version of the proposed algorithm to gain an insight into the principles of how the algorithm functions. We have tested this simplified version using artificial students. These are simulations of human students whose knowledge changes on the basis of the material that they practise. As a consequence, the way that they solve problems also changes. We have tested to what extent our algorithm was able to track these changes in knowledge and therefore in observable behaviour. We observed that the algorithm could only find an accurate model of our artificial students in the more simplified test cases. The algorithm in its current form is not robust enough for more complex (and therefore more realistic) test cases.

1 Introduction and overview

1.1 The context of this thesis

This thesis describes the research project that I have done for the final stage of Artificial Intelligence at the University of Groningen, The Netherlands. The research described in this thesis was conducted at the School of Information Technologies at the University of Sydney, Australia. The supervisors of this project were Dr. Josiah Poon at the University of Sydney and Dr. Niels Taatgen and Dr. Rineke Verbrugge at the University of Groningen.

1.2 Student modelling and Intelligent Tutoring Systems

Intelligent Tutoring Systems (ITSs) are computer programs written for educational purposes that use information that was deduced from the student actions to personalize the program in accordance with the student's needs. This means that such a system presents information and exercises to a student that is expected to be most beneficial for this particular student. Such a customized learning environment ensures that the student can learn more efficiently, mainly because the student is always practicing at the appropriate level (i.e. the student is less likely to get stuck on an overly difficult problem and is less likely to waste time with practising an overly simple problem). The adaptation to the student is considered to be the most important reason why human tutors are so effective (Bloom 1984), and ITSs are aimed at achieving a similarly effective teaching style.

To be able to customize the learning material these systems keep track of a set of beliefs about the student's skills, knowledge and characteristics. This set of beliefs is referred to as a *student model*. Keeping track of a correct student model is one of the most difficult tasks in designing an ITS. An ITS only has access to a limited amount of observations from which it can draw its conclusions about the student's capabilities. Whereas a human tutor can infer a range of characteristics about the student's attitudes, background, knowledge, etc. either from direct observation, from experience or from observations in the past, ITSs usually only have access to information about direct interactions with the system (essentially no more than the keyboard inputs). Furthermore, the system needs to be flexible: sometimes the system should be able to revise its conclusions about the student's capabilities since none of the conclusions can be drawn with absolute certainty. Any student modelling technique in an ITS therefore needs to incorporate some kind of uncertainty management.

1.3 Research summary

The research that is described in this thesis focuses on the updating of the student model. Since often the observations of the student behaviour can be interpreted in more than one way, it is not always possible to make a direct translation from observation to update of the student model. To tackle this problem, we suggest a novel approach that uses a genetic algorithm (GA) to draw several parallel conclusions from observations of the student's interactions with the program. Each different conclusion results in a different hypothetical student model that therefore corresponds with one way of explaining the observations. The student models that interpret the observations well will be able to make correct predictions about the student behaviour and are retained as realistic models of the student.

Exactly how each student model interprets the observations and adjusts its beliefs about the student is specified in a set of rules called the *update rules*. Some student models will be able to make better predictions about the student's actions than others, that is, the update rules that govern these student models will be more effective than others. These student models and their corresponding update rules are selected to be the basis of the next generation of hypothetical student models. In this next generation the parameters of the update rules will be slightly modified to test whether there is a set of update rules that has an even better predictive capability. In this way it is hoped that an effective way of updating the student model can be found for every individual student.

The original intention was to test the proposed algorithm in a realistic situation. To this purpose the core of an ITS for teaching high school level students the technique of mathematical differentiation was developed. Due to time constraints this system could not be fully implemented. Instead we decided to test the basic principles using a simplified testing algorithm that used artificial students with certain predefined behaviours. An advantage of this approach is that although the test conditions are not as realistic as they can be, a better statistical analysis of the results is possible. This gives a better insight in how well the principles of the algorithm work than a classroom test in which the interpretation of the results is much more difficult.

The approach of updating the student model that is described here is quite novel, so the main focus of this research is on the feasibility of this approach and how it compares to and complements other techniques used for updating the student model.

1.4 Thesis overview

Conceptually, this thesis is divided in three parts. In the first part (chapters 2 to 4) the fields of Intelligent Tutoring Systems, student modelling and genetic algorithms are described to give some background information about the research areas that are relevant for this research project.

In the second part (chapters 5 to 7) the research project is described. First the proposed algorithm is described, and the motivations are given why we were interested in trying out this technique for modelling the student. After that a description is given of how we evaluated the basic functioning of the algorithm by implementing a simplified version of the algorithm and testing it with artificial students. Finally, the results of the series of tests are presented.

In the last part (chapter 8) the conclusions are presented. An evaluation of the practical feasibility of the algorithm is given. After that a number of theoretical explanations for the results are given and the research questions that we have posed are answered. Subsequently some suggestions for further research are given. Finally the motivations that inspired the research are re-examined, to assess to what extent our expectations and motivations corresponded with the results that we obtained. [1,2]

[1] In the remainder of this thesis students and tutors are referred to as 'he'. This should be interpreted as referring to both male and female students and tutors.

[2] New definitions shall be introduced in this thesis in *italic* script. An overview of these terms can be found in the appendices.

2 Intelligent Tutoring Systems

In this section I shall first give an overview of the research area of ITSs and explain why we are interested in this form of computer-based education. After that I shall describe how these systems are generally structured and indicate what kind of ITSs we can distinguish. Finally I shall discuss to what extent these systems are used in practice.

2.1 Introduction

Ever since personal computers became widely available, there has been an interest in the use of computers as tools for educational purposes. By using computer software for teaching, ways of interaction are possible that cannot be achieved by using textbooks or by teaching in classrooms. The earliest attempts at *Computer Assisted Instruction* (CAI), starting in the 1960s, have not always been successful. The initial expectations were high: CAI would educate school children and students in a new way that would never be boring and would explore an interesting new use of computers. These early attempts at introducing computers in the classroom did more to uncover the problems in this field than actually produce usable programs (Wenger 1987). The costs of developing CAI programs were underestimated, the inherent attractiveness of the medium was overestimated and providing responsive teaching turned out to be much more complex than expected (Kimball 1982). This same kind of initial over-optimism about a new technology and its influence on education could be seen with the introduction of radio and television (Hoban 1946) and more recently with the advent of the internet. Often the CAI programs consisted of a basic algorithm that analysed the multiple-choice answer of the student and offered the next question based on this answer. These script-based programs have been criticized in those days as 'a very expensive substitute for a book' (Kemeny 1972).

Advances in the field of artificial intelligence techniques in the late seventies, most notably in knowledge representation and uncertainty management, have helped to give a new impulse to the field. To signify the difference with the older CAI programs, the term Intelligent Tutoring System (ITS) was coined (although originally the term Intelligent Computer Assisted Instruction was sometimes used to indicate the same research area). ITSs aim to give *individualized* instruction. ITSs are not only focused at the material that is taught, but also at the students and their personal needs. Advances in the understanding of how to encode expert knowledge, student knowledge and instructional principles have allowed the design of more sophisticated programs. This way an ITS can adopt a teaching style that is closer to one-on-one teaching (Lesgold 1987).

2.2 Advantages of Intelligent Tutoring Systems

After the phase of CAI systems the initial unrealistically high expectations cooled down somewhat and the goals of ITSs were set to more realistic levels. In this section I shall

look at some of the (potential) advantages of these systems, to give an impression of why we are interested in computer-based education.

The key feature of an ITS is its adaptability to the level and needs of the student. This is one of the most important reasons why human tutors are so effective. In a study by Bloom, classroom teaching in a class with 30 students was compared to one-on-one tutoring (Bloom 1984). It was found that students in the tutoring condition performed with a mean of two standard deviations higher than the students that only received education in the classroom situation. The better performance of the students that were tutored privately was an effect of the multiple styles of interaction between the student and the tutor. A combination of non-verbal cues, discussion, questioning, feedback, support and correction helped to teach the students in a much more versatile way and the teaching styles were tailored exactly to the student's needs. The obvious problems with private tutoring are the availability of tutors and the costs involved. It is not feasible to provide every student with a private tutor all the time, and this is why the potential of ITSs is so interesting. The main goal of ITSs is to mimic some of the interaction styles of human tutors that are so effective without the high costs and the problem of limited availability of tutors.

Numerous articles have identified advantages of ITSs. These include:

- Availability of direct feedback. There is psychological evidence that the most effective pedagogical action that can be taken in case the student makes a mistake is to correct the mistake instantaneously. For the student it is easier to localize and analyse the mental state that led to the mistake and to identify bugs in their knowledge if the mistake is pointed out straightaway. An additional advantage of offering direct feedback is that the frustration that occurs when a student gets stuck due to a lack of knowledge is reduced (Lewis, Milson et al. 1987; Kulik and Kulik 1988).
- Availability of instant help. This advantage is directly related to the one pointed out in the previous paragraph. The student does not waste time with being stuck and can take the initiative to improve his knowledge in order to overcome the difficulty at hand.
- Possibility of evaluating many students at the same time. This is a particularly time saving feature from the perspective of the teacher.
- Possibility of storing large databases of questions. If a student needs to practise a particular topic more often than the average student, there is no practical limit of the amount of questions about this topic that can be stored (although naturally the time needed to design questions can be a limiting factor). In a textbook, on the other hand, a compromise needs to be made between brevity and clarity (Millan, Pérez-de-la-Cruz et al. 2000).
- Possibility of storing commonly made mistakes. By storing mistakes that students are likely to make in a database, it is possible to give directed feedback about how to prevent this kind of mistake the next time. This strategy has been applied successfully in a logic tutor (Lesta and Yacef 2002).

Some more system-specific advantages include:

- Overcoming fear of formal reasoning. Fung and O'Shea have pointed out that a major difficulty for students in mastering the knowledge of formal domains is that they are

intimidated by the unfamiliar and complex formal notations. By allowing them to experiment freely with the constraints and the possibilities of the concepts that are taught the basis for a deeper understanding is laid. The students can manipulate and investigate relatively complex concepts at a stage where their own limited expertise would make this difficult in a traditional context (Fung and O'Shea 1993). Heffernan and Koedinger have done research in order to discover why students have difficulties with symbolization (i.e. translating descriptions about a particular situation into mathematical formulas). They discovered that the 'articulation in the "foreign" language of "algebra" ' caused the students the most problems (Heffernan and Koedinger 2002).

- Incorporation of multimedia elements. Some concepts are easier to explain visually, for example the movement of molecules in a gas. Multimedia simulations offer the possibility to experiment freely without causing unsafe or undesirable situations in the real world (Shapiro and Eckroth 1987).
- Deployment at any place and any time. Since the advent of the Internet new possibilities are opened up that allow students to study at any location and whenever they want to study. Students are not bound to a physical location anymore.
- Teaching in domains that have few experts. For some domains there are only a few people who are sufficiently informed about the material that needs to be taught. This was the reason to develop an ITS in the field of quantum information processing (Aïmeur, Brassard et al. 2002).

The goal of ITSs to improve the effectiveness of teaching methods by creating a personalized teaching environment has often been achieved. In many research papers a positive influence on the performance of the students and a reduction in the time that students need to study to achieve similar results as their peers have been measured (Shute, Glaser et al. 1989; Mark and Greer 1991; Koedinger, Anderson et al. 1997). Also the effects on the motivation of the students are often positive. In the research by Nicaud, Bouhineau et al. the Aplusix-Editor for forming algebraic expressions was tried in the classroom. They described that 'for many students, what was usually opaque in algebra regained interest. Some of them, who generally didn't listen, all of a sudden began to ask questions. From passive, they became active.' (Nicaud, Bouhineau et al. 2002).

2.3 Components of Intelligent Tutoring Systems

ITSs usually consist of several components, that all implement a certain part of the functionality of the program. Figure 2.1 gives a possible generalization of these components (Beck, Stern et al. 1996). Although some ITSs may interact in slightly different ways than depicted in this figure, most systems have similar components. In the next sections I shall give a quick summary of these components.

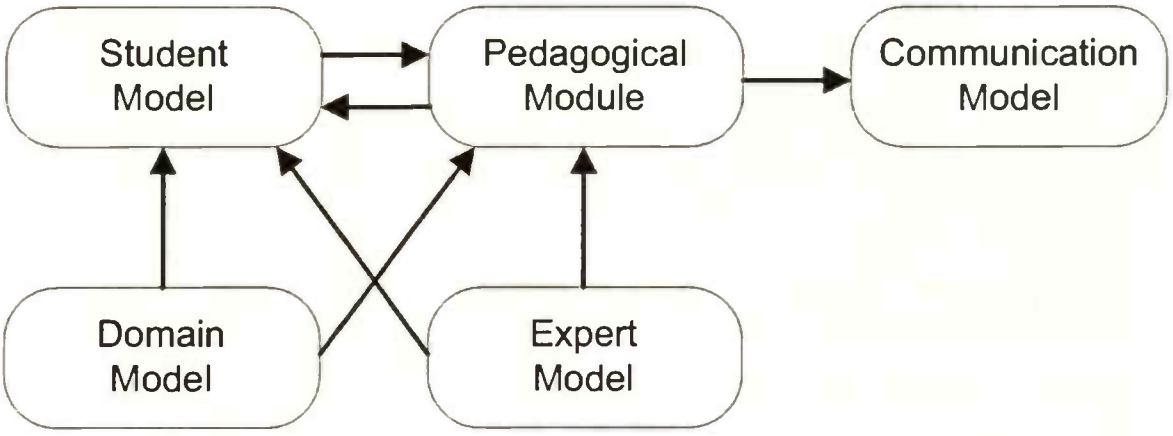


Figure 2-1: General components of Intelligent Tutoring Systems (Beck, Stern et al. 1996). The arrows indicate the information flow between components.

The student model

The student model consists of a set of variables that describes certain aspects of the student's knowledge and skills. The student model forms a very important part of most ITSs. Without a student model, the material that is offered to the student cannot be personalized. Furthermore, if the beliefs about the student's knowledge and skills are incorrect, the system will not be able to present the material to the student that is most useful for the student at that moment.

The student model is usually formed by getting feedback about the way the student performs (e.g. whether the student has made a mistake, how long it took to answer the question, etc.). It is regularly updated in this way, usually after every problem that is presented to the student. Another source of information for the student model can be a direct questioning of the student.

Since the student model is the main focus of the research project described in this thesis, I will elaborate on this topic in chapter 3.

The pedagogical module

The pedagogical module is used to model the teaching process and it incorporates knowledge about teaching techniques and strategies. It uses the information stored in the student model to adjust the presentation and selection of the material that is taught. For example, the pedagogical module may influence which topics and problems should be taught next, considering the topics with which the student is familiar. The pedagogical module may also make decisions about the level of the problem to generate next, which hints to present, what feedback to give and what information to hide or present (Mayo and Mitrovic 2000).

The pedagogical module should select problems that are neither too complex nor too simple. The appropriate complexity of a problem is one that falls in the *zone of proximal development*. This is 'the distance between the actual development level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or collaboration of more capable peers.' (Vigotsky 1978). This means that a problem needs to be far enough above the level of the student to be challenging, but close enough to the level of the student to not discourage the student.

Domain model

The domain model holds all the domain-specific knowledge about the domain that is taught. It is quite difficult to effectively store this knowledge so that it can easily be accessed by the other components. In a textbook it is often sufficient to describe the different concepts in a logical order in the hope that the students will be able to see how the concepts relate to each other. To facilitate this process, usually a number of problems are posed for the student to solve. However, for an ITS the requirements are much higher: it has to present personalized feedback and exercises to the student. These requirements imply that a lot of meta-knowledge of the domain is necessary (besides the knowledge of the domain itself). I shall give some examples:

- To assess the level of the student, it is necessary to know whether the sum that he just solved was difficult or easy.
- If the student has difficulties with a topic, it is interesting to see whether he has mastered the topics that are required for understanding this concept (the so-called pre-topics).
- In order to give useful hints it is necessary to know what kind of hints are available: some students may prefer abstract hints whereas other students benefit more from more concrete examples.

The distinction between pedagogical knowledge and domain knowledge is not always very clear. The domain model contains a lot of meta-knowledge that is pedagogical by nature. The pedagogical module, however, is aimed at making *decisions* about what to teach next, whereas the domain knowledge is more descriptive. For example, a pedagogical module may specify that topic X can be taught when pre-topics A, B and C have all been mastered for over 80 percent. The domain knowledge component on the other hand describes what the pre-topics for every topic are.

Communication model

The communication model determines how the information that the pedagogical module decides to present can be effectively communicated to the student. There are many different ways to communicate with students. Some ITSs offer only very basic forms of interaction, for example by asking multiple-choice questions. However, some research projects have focused almost entirely on this component. Some examples are dialogue-based systems (Paraskakis 2002), systems that can adapt the user interface to the level of the student (Cooper 1988) and simulation-based ITSs (Shapiro and Eckroth 1987).

Expert model

The expert model represents the domain knowledge similar to how a domain expert would. This way the solution of the student can be compared to the solution of a domain expert. Usually the expert model is a runnable model that can actually solve the problems.

There is an overlap between the domain knowledge and the expert model: There is a close interaction between these two modules: the concepts, procedural rules, meta-rules by which the concepts are used and the heuristics all need to be engineered in such a way that this interaction can take place (Woolf 1988).

2.4 Knowledge-based tutoring systems and cognitive tutoring systems

One way to classify ITSs, is to distinguish between systems that focus on teaching procedural skills and systems that focus on teaching concepts. Systems that mainly teach procedural skills are referred to as *cognitive tutoring systems*. They teach a particular skill (for example a mathematical skill) and compare the skills of the student with the skills of an expert in the domain. Often this type of system has a runnable expert model that is constructed after analysing the domain knowledge using techniques from cognitive psychology.

Systems that focus mainly on teaching concepts are called *knowledge-based tutoring systems*. They are usually more difficult to implement, since the process of learning concepts is more difficult to model and less understood than learning procedural knowledge. These systems require a much bigger knowledge base than cognitive tutors. Usually this type of system places more emphasis on the communication of the knowledge to the student (Beck, Stern et al. 1996).

Usually it is not possible to make a clear distinction between cognitive tutoring systems and knowledge-based tutoring systems, since most systems contain features of both types of system. There are many different teaching styles that ITSs can adopt. For example, for rote-learning tasks such as topography and augmenting student's vocabularies in a foreign language, a simple system that keeps track of the student's knowledge may be sufficient. However, more interesting styles of interaction are also possible. To mimic the teaching style of a human tutor, dialogue based systems have been developed (Paraskakis 2002). For some domains, such as learning how to operate a machine or learning concepts in physics, a simulation environment is more appropriate. This allows students to freely experiment and even simulate conditions that are not even possible in the real world. In game environments the students are encouraged to apply their knowledge in a playful way (Lesgold 1987). In short, the teaching styles of ITSs are as diverse as the teaching styles of human tutors.

In general it can be said that most ITSs focus on the more formal domains, such as programming and mathematics, and therefore the cognitive tutors are most popular. The reasons for this are twofold. Firstly, most researchers that do research in this area have a technical background. In practice, this leads to a distinct preference for the more formal domains such as programming and mathematics. The second reason is that it is much easier to represent all the knowledge that is necessary for these formal domains on a computer. The system can directly evaluate the answer of the student, since for formal domains the answer is often either correct or incorrect. It is straightforward to represent the domain knowledge of a formal domain on a computer, since it is essentially a machine based on logic. For humans, formal reasoning does not come so naturally, since most of the human reasoning processes tend to be more heuristics-based. To have interesting interactions with a knowledge-based tutor on the other hand, often some kind of dialogue-based system is necessary. Naturally such a system is much more difficult to implement, because not all the possible interactions with the students can be foreseen.

2.5 Use of ITSs in the classroom

Considering the years of research into ITSs, the use of ITSs in the classroom is rather limited. The main reason for this is that the design and implementation of an ITS is very costly. The development of an ITS for even only a small domain has proven to be quite difficult. Anderson, Corbett et al. have concluded that it takes about 10 hours to construct one production rule (an particular type of elementary rule that defines a part of the domain knowledge) (Anderson, Corbett et al. 1995). Woolf and Cunningham have estimated that for the development of an ITS it takes more than 200 hours to develop one hour of instruction (Woolf and Cunningham 1987). There are several reasons for this.

First of all, there is the inherent complexity of the systems. The field of ITSs lies at the intersection of three research fields: computer science, cognitive psychology and educational research. This makes it a challenging field to research, since keeping up with the developments in just one of these fields is already quite demanding. Often the researchers have a certain preference for their own area of expertise, which can be deduced from the fact that a large part of the ITSs that are developed cover the domain of computer programming. Nicaud, Delozanne et al. have argued that one of the problems with the field of ITSs applied to the domain of algebra is the separation between the fields of cognitive psychology and computer science on the one side and the field of algebra education on the other side (Nicaud, Delozanne et al. 2002). Self rightly points out that if we want to model the interactions between the student and the tutor completely, 'the student modelling problem expands – from computational questions, to representational differences, through plan recognition, mental models, episodic memory to individual differences – to encompass, it would seem, almost all of cognitive science.' (Self 1990).

Secondly, the re-use of software components has proven to be quite difficult. Although all the systems implement the components that were mentioned in section 2.3 in some way, they are often interwoven and the interactions between the different parts of the system differ from system to system. Therefore it is difficult to design a general authoring tool that can function as a backbone for any system. Considering the variety of systems that are implemented, this is hardly surprising. Authoring tools such as REDEEM (Ainsworth and Grimmshaw 2002), WEAR (Virvou and Moundridou 2000) and GET-BITS (Devedzic and Jerinic 1997) only allow limited freedom in the design of an ITS. The result is that most systems are built from scratch. Devedzic, Radovic et al. have argued that the use of re-usable and upgradeable software components that can be programmed in different languages are the way to go, but so far no working architecture has been developed (Devedzic, Radovic et al. 1998).

The conclusion that can be drawn here is that the transparency and simplicity of the techniques that are used in an ITS is of crucial importance to the economic feasibility of the system. Without an easily adaptable framework the development costs of an ITS are simply too high for practical use. Also, we should not expect to build the 'perfect' system. The techniques that are necessary to model only the basic interactions between the system and the student are already rather complex.

3 Student modelling

In this chapter I shall give a short overview of the field of student modelling. Firstly, I shall discuss some issues that need to be addressed when designing the framework for a student model. Secondly I shall discuss some of the more common techniques. Finally, I shall give some evaluation criteria for student modelling techniques, to give an insight into what aspects are important for a practically usable technique.

3.1 How to construct the student model

A student model contains information about the knowledge of the student. There are several ways of obtaining this knowledge. Some often-used methods are given here:

First of all, there is the most direct method: the user interview. Especially for simple features this technique is quite effective. For example, before the student is asked to use a particular grammar rule, we can ask whether the student is familiar with this rule.

Secondly, it is possible to describe directly how the student model should be updated after a particular observation has been made. The rules that govern this type of updating can be quite simple. For example, if the student makes three consecutive mistakes with irregular verbs, we can conclude that the student's knowledge of irregular verbs is poor.

Thirdly, we can use inference procedures to draw new conclusions. For example, if the student has a low score on addition and subtraction skills, we can conclude that his elementary calculation skills are poor.

Finally we can make assumptions about the student's knowledge by using *stereotype student models*. These are standard models that make certain assumptions about the student based on what group he belongs to. For example, if the student is a beginner in studying French, it is reasonable to assume that he has some knowledge of regular verbs, no knowledge of irregular verbs, etc. (Pohl 1996). Stereotype student models can be loaded from a file by simple if-then rules that are triggered based on a particular observation. Stereotypes are especially useful for initializing the student model. Naturally stereotype student models cannot be realistic for all students, but it is expected that they are realistic for most students (Kay 2000).

3.2 Representational issues

One of the most important issues in the domain of student modelling is how to represent all the different types of knowledge. Before all the components of an ITS can exchange information, the data needs to be abstracted in some way. Some decisions need to be made about how to do this. One common mistake in the design of student models is that the student model is made more elaborate than is necessary. If the pedagogical module only makes decisions based on whether the student is a 'good' or 'mediocre' student, than that is all that needs to be modelled by the student model (Self 1990). Usually fine-grained student models are needed for decisions that have a short-term effect, such as

which exercise to select next. Large-grained models are used for decisions that have a long-term effect, such as deciding on which course to study next (Martin and VanLehn 1993).

Another representational issue is how to model what the student knows in comparison with an expert. We can make a distinction between overlay and buggy models. In *overlay student models* the knowledge of the student is represented as a subset of the total domain knowledge that is modelled. A drawback of this approach is that it does not acknowledge that students can have beliefs that are outside the set of beliefs that is modelled in the domain. In *buggy student models* the student's incorrect beliefs are also modelled. This can be advantageous for the system, because some of the mistakes of the students can be interpreted better if they are seen in the light of common misconceptions.

3.3 Common student modelling techniques

In this section I shall describe some of the more popular techniques that have been used for student modelling.

Bayesian networks

One of the most common ways of representing the student's knowledge is using *Bayesian networks* (also called *belief networks*). Every node in these networks represents a certain part of knowledge that a student can have. The nodes are connected by arrows that depict how the factors influence each other. For example, the student's knowledge of formulas can influence his understanding of graphs. Because a probabilistic value is assigned to these relationships, Bayesian models are relatively insensitive to noise and do not have the difficulties with reasoning about an uncertain domain that logical reasoning systems have (Russell and Norvig 1995).

The specification of causal relationships between factors makes it possible to *predict* outcomes that depend on particular causes (*predictive inference* or *causal inference*) and also to *interpret* observed outcomes as evidence concerning the variables that caused them (*diagnostic inference*). For example, if a student is good at solving addition exercises, it is predicted that he will correctly solve a simple addition sum. Conversely, if we observe that he solved a difficult addition sum, we can take this as evidence that he is good at solving addition exercises.

Since all the factors are connected by probability values, these networks are updated according to the rules of probability theory (Jameson 1996). The name Bayesian network is derived from *Bayes' rule*, which is used to calculate unknown probabilities from known probabilities. This rule is used repeatedly for updating the network (Russell and Norvig 1995).

There are several difficulties in applying the Bayesian approach to student modelling. A disadvantage of Bayesian networks is that they only provide a snapshot of the student's knowledge. No learner styles are modelled and it is not possible to revisit domain areas that were previously very difficult for the student. Also the computational complexity can be a problem (Reye 1996). Finally, the knowledge acquisition is difficult. All the initial probability values need to be specified for a network. For large networks an astronomical number of probabilities need to be specified (Murray 1998).

ACT-R based systems

To keep track of the student's cognitive capabilities the ACT-R (Adaptive Control of Thought, Rational) theory can be used. This technique has been applied successfully in practice: several ITSs using this approach have been tested in classroom situations (Anderson, Corbett et al. 1995). ACT-R based tutoring systems are cognitive tutors and therefore focus on formal domains such as LISP (a programming language), geometry and algebra. A central part of ACT-R based tutors is a runnable expert model that is capable of solving the problems that are given to the student in a similar way to how the student is expected to solve them.

The theory makes a distinction between declarative knowledge and procedural knowledge. *Declarative knowledge* is the kind of factual knowledge that we are aware of and can describe to others (however, this does not imply knowing how to use this knowledge). *Procedural knowledge* on the other hand, is knowledge that we display in our behaviour but we are not conscious of. Examples of procedural knowledge are mental manipulations (such as subtracting two digits) or knowing how to manipulate physical entities (such as accessing a computer application). Procedural knowledge is used to solve problems on the basis of our declarative knowledge (Anderson and Lebiere 1998). ACT-R theory provides a framework for implementing cognitive skills by specifying *production rules* (the procedural knowledge) and facts (declarative knowledge). Several parameters can be set for elementary operations such as retrieving a fact from memory or comparing two facts.

An important research goal of the ACT-R theory is to model the cognitive skills of humans. Although the theory has been quite successful in achieving this goal, for the application to ITSs the theory has been criticized as not being flexible enough to let the student explore different solution paths. Every step that the student makes is compared to the step that an expert would make, which makes it impossible for the student to combine several primitive operators and apply them at once. This kind of rigidity also makes it difficult for more experienced students to explore the problem space (Mitrovic 2000).

Other techniques

Several other techniques have been used for student modelling, although they are not used as often as ACT-R-based techniques and Bayesian techniques. Some of these are:

- Dempster-Shafer theory of evidence. This technique is designed to distinguish between uncertainty and ignorance. Instead of directly computing the probability of a proposition, the probability that the evidence supports the proposition is calculated. For example, when a system observes that a student makes five mistake adding two numbers, the *belief mass* (degree of belief in a proposition) that a student is not good at adding numbers is larger than when the system observes only one mistake. In other words, a measure of the reliability of the evidence is given, which gives this theory an intuitive appeal. A disadvantage of this technique is that it cannot be used to make predictions: it is only designed to support diagnostic inference (Russell and Norvig 1995; Jameson 1996).
- Fuzzy logic. This technique is used for reasoning with vaguely defined propositions. This kind of proposition is typical for natural concepts. For example, if we say that a student is 'good' at maths, we expect him to do 'quite' well at a 'simple' problem (Jameson 1996). The reasoning processes that take place in systems that use fuzzy

logic are specified in a similar way. However, at some stage these fuzzy concepts are translated into certainty values, which means that this technique is really more a description of uncertainty about the meaning of the linguistic terms used than a real uncertainty management tool (Russell and Norvig 1995).

- Machine learning. Machine learning techniques (such as *neural networks* (Beck and Woolf 1998) and *decision tree learning* (Chiu and Webb 1997), see appendix) all have in common that usually little knowledge engineering is necessary to construct these systems. The data processing is seen as a black box that connects inputs to the desired outputs. This can be both an advantage and a disadvantage. These systems are more flexible when coping with unexpected situations or student behaviours, since it is not necessary to model all situations in advance. On the other hand, they are less insightful if we want to know why a system behaves the way it does (Pohl 1996).
  The technique that is proposed in this thesis can be classified as a machine learning technique.

3.4 Criteria for practically usable student models

To give an insight into what qualities of a student modelling technique are important, Jameson has composed a list of criteria by which the practical usability of a technique can be evaluated (Jameson 1996). Unfortunately, it is not possible to compare different classes of techniques used for student modelling to each other directly. Too much depends on the specific implementation of a technique. The criteria mentioned here do not specify how accurate a student modelling technique is. Rather, they aim to evaluate whether a technique can be used satisfactorily in the conditions in which research and application typically take place. This gives a good insight into the difficulties that need to be overcome when designing an ITS with a student model.

Jameson's criteria for techniques used for student modelling are:

- Where will the numbers that are needed for building the student model come from? Many research projects have focused on how to elicit quantified knowledge from domain experts (for example probability assessments). The quantifying of all parameters is often not an easy task. Some parameters are set intuitively by the programmer. Self argues that it is better to make an educated guess about a particular parameter than not to include the parameter at all (Self 1990).
- How much effort does it cost to implement the system? A technique will be more successful if it is easy to program or if an easily adaptable shell can be implemented.
- To what extent will the system have to be improved through trial and error? It is important to specify how each parameter influences the behaviour of the system. If the system then behaves unexpectedly, the designer should not merely adjust some parameters, but rather reassess the initial assumptions.
- Will the inference methods of the system be efficient enough to permit acceptably fast system responses? In the fields of Bayesian networks this is a relevant issue, especially if the model itself is quite complex.
- To what extent will the inferences made by the system be similar to the inferences made by humans in a similar situation? This question is especially important if the

technique is aimed at simulating the student's reasoning rather than at managing uncertainty about the student's knowledge.

- To what extent will it be possible to explain the inferences to users and other people who want to understand them? For the user it can be insightful to view the structure of a domain in a Bayesian network, for example. Opening up the student model to be viewed by the student can often be beneficial to the student. According to Goodman, Soller et al. 'reflective activities encourage students to analyse their performance, contrast their actions to those of others, abstract the actions they used in similar situations and compare their actions to those of novices and experts.'(Goodman, Soller et al. 1998).
- To what extent can the conclusions drawn by the system be justified? This issue is particularly important in case conclusions of a system have important consequences, such as in assessments of a student's suitability to follow a course.
- How effectively will it be possible to communicate the lessons learned in the design and testing of the system to other designers of user modelling systems? If a technique is formulated using a system-specific framework rather than using a well-known uncertainty paradigm, it is hard for other researchers to gain an insight into how the technique works. This makes it difficult to compare the technique to other techniques.

4 Genetic algorithms

Since the research that is presented in this thesis uses a genetic algorithm for student modelling, I shall give a general overview of this field in this chapter. In the introduction I shall first give an explanation of how genetic algorithms work and in the subsequent chapters I shall discuss some topics that are particularly relevant for this research project more in depth.

Four articles that give an overview of the field of genetic algorithms were used for this chapter (Beasley, Bull et al. 1993; Beasley, Bull et al. 1993; Whitley 1994; Busetti 2000). These sources all give a good but similar overview of the field, so for increased readability I shall only refer to them here.

4.1 Introduction

Genetic algorithms (GAs) are used for search and optimization problems in which there is no path that guarantees an acceptably quick way to get to the goal. GAs are inspired by the evolution theory that was first formulated by Charles Darwin in '*The Origin of Species*' in 1849. In nature different individuals compete for resources such as food and water. The individuals that are better adapted to their environment will have a higher chance of surviving and reproducing. The children will have the genes of both of the parents, and will therefore also exhibit some of the traits of both parents. Sometimes a child will be better adapted to its environment than both the father and the mother. This child therefore has a better chance of surviving and reproducing and in this way species can evolve to be more and more adapted to their environment. The processes of natural selection and survival of the fittest involved in the evolution of biological organisms are mimicked by genetic algorithms (GAs) in order to evolve a good solution to a problem. This technique was first suggested by Holland (Holland 1975).

The terms that are used for GAs are similar to the terms used for biological evolution. [3] GAs start by trying a number of possible *solutions* simultaneously. The *individuals* (i.e. the possible solutions to a problem) are selected for reproduction on the basis of a *fitness function*, which ensures that the solutions closer to the desired answer are more likely to survive. The algorithm iteratively selects the best solutions to form a basis for the next set of possible solutions. In this way it is hoped that an acceptable solution will be found in a reasonable time span, without having to explore too much of the *search space* (i.e. all the possible solutions). A typical example of a problem that can be solved with a GA is that of designing a bridge that has a good strength-to-weight ratio. To solve such a problem the combination of for example the number of arches, building material and beam thickness can be systematically varied until a satisfying design is acquired.

[3] The reader is reminded that the definitions of the terms introduced here in *italic* can be found in the appendix.

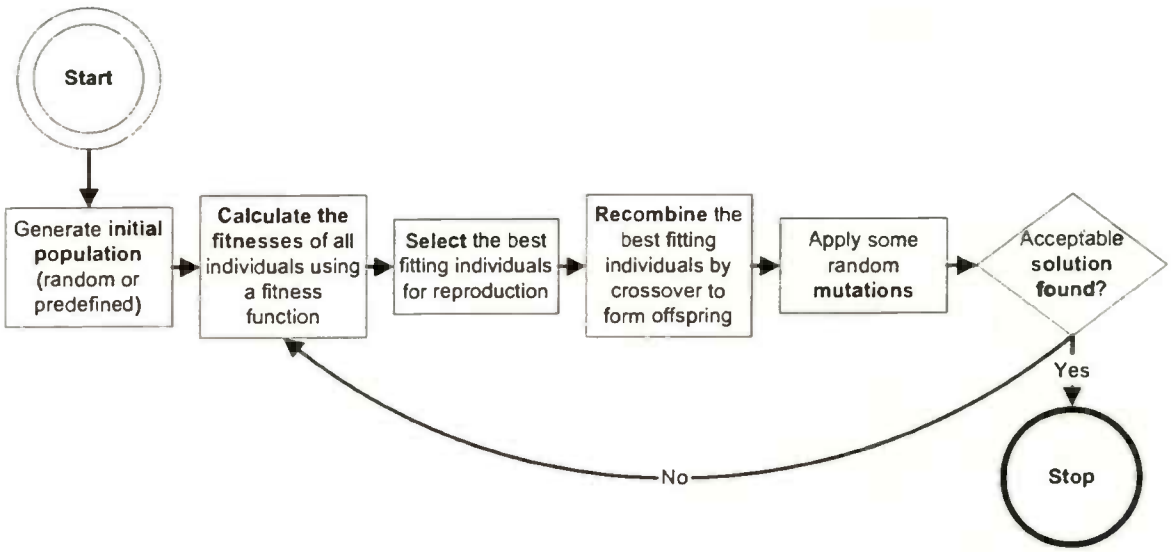


Figure 4-1: General structure of genetic algorithms.

Figure 4-1 gives the general structure of genetic algorithms. I shall now describe one iteration of the algorithm to give a better understanding of how GAs function. After that I shall discuss the algorithm more in depth in the following sections.

The group of solutions (or *population*) that is evaluated in one iteration is called a *generation*. Usually the initial generation consists of random individuals, but if there are individuals in the search space that are expected to quickly lead to a good solution, it can be sensible to start with a population that also contains these individuals. A *chromosome* is the representation of a solution that consists of the parameters of a solution.[4] The parameters that together form a chromosome are often referred to as *genes*. An example of a gene is the thickness of a beam in a bridge design. For each solution a fitness score is calculated using a *fitness function* to signify how good the solution is. A set number of best fitting individuals (the *parents*) is selected to *reproduce*. The selected individuals reproduce by *crossover* (i.e. *recombine* the genes of the *parents* to produce the *offspring*). After that a number of genes are *mutated* at random to introduce new gene values. Sometimes two parents will produce a '*super fit*' individual, whose fitness is greater than that of either parent. This is the principle that brings the algorithm closer to an acceptable solution. The worst fitting individuals do not reproduce and *die out*, whereas the better fitting individuals are likely to make it to the next generation and form the basis for another generation of solutions. If at this stage an acceptably good solution has been found, the algorithm terminates, otherwise the process repeats itself.

GAs are effective algorithms mainly because of their robustness and ability to deal with a wide range of problems. GAs are classified as *weak search methods*, because they make relatively little assumptions about the domain to which they are applied. They are not guaranteed to find the optimal solution to a problem, but they can find an acceptable solution to a problem acceptably quickly. For some problems specialized techniques exist

[4] It is important to note that the terms individual, chromosome and solution will be used intermittently, but refer to the same concept: the parameterized representation of a solution. The term individual is used when the relation between solutions is most important (for example when we are comparing parents and children). The term chromosome is used in the context of recombining solutions. The term solution is used when we want to stress the practical meaning of the representation (i.e. a possible solution to a problem).

which are quicker and more accurate; GAs are intended to solve problems where no such techniques exist.

4.2 Coding

The first thing to do when implementing a GA is to devise a suitable coding for the problem. A potential solution consists of chromosome parameters that can be set. It is up to the program designer to come up with an efficient representation of the solutions. In the bridge design example, it can be decided to only use simple rules of thumb and in this way simplify both the representation of the genes and the fitness function. Alternatively a lot of parameters like different choices of materials with their properties, triangular constructions etc. can be represented in each solution. This would dramatically increase the search space, but might deliver more accurate and unforeseen solutions.

The set of parameters that form a chromosome is often referred to as the *genotype*. For example, all the parameters that specify the design of a bridge (number of arches, thickness of beams, etc.) form the genotype. The resulting organism (in our example the constructed bridge with all its physical properties) is referred to as the *phenotype*. The fitness of an individual depends on the performance phenotype.

4.3 The fitness function

Closely related to the problem coding is the design of the fitness function. It should accurately indicate how good a particular solution is, so that no inappropriate selections are made in the selection phase. In other words, the calculated fitness value should be *proportional* to the utility of the solution. The fitness function basically tries to minimize or maximize some function $F(X_1,X_2,\ldots,X_M)$. In GA problems it is not possible to optimize each parameter independently, since the parameters influence each other. For the simple problem of function optimization, the fitness value is simply the value of the function. If the exact fitness is hard to calculate, a heuristic fitness function can be used to save processor time. If it takes the same time to evaluate 10 approximate fitnesses as to evaluate one exact fitness, it is often more economical to choose the first option.

4.4 Parent selection

In the selection phase the fittest individuals are selected for reproduction. An often-used method is *remainder stochastic mapping*. In this method a relative fitness value ($fitness_i$ / $fitness_{average}$) is calculated for every individual *i*. Of each individual a number of copies equal to the truncated relative fitness value (note that this may be zero) is placed in an intermediary population (called the *mating pool*) that is filled with the parents of the next generation. The decimal part of the relative fitness value indicates the *chance* of an individual to be selected for the mating pool (again). For example, an individual with $fitness_i$ / $fitness_{average}$ = 2.36 will be placed in the mating pool twice with a .36 chance of being placed in the mating pool once more. An individual with $fitness_i$ / $fitness_{average}$ = .12 has a .12 chance to be placed in the mating pool. This way the fitter individuals will survive and individuals will reproduce proportionally to their fitness.

There are quite a few other selection methods; all have in common that the fitter individuals have a higher chance of producing offspring. The difference between these methods lies mainly in the *selection pressure*, i.e. how much the fitter individuals are favoured over the less fit individuals. An interesting technique of reducing the selection pressure is to use a *fitness ranking* scheme in which the fitness of an individual is remapped to its fitness rank in the population. This effectively compresses the fitness range so that extremely fit individuals do not have a disproportionately high number of individuals that are used as the basis for the next generation.

4.5 Crossover

The reproduction of the individuals ensures that every child has properties of both the parents. This is usually done by single point crossover: a random point in a chromosome is selected and both parent chromosomes are split at that point into a head and a tail segment. After that the tail segments are swapped to produce the two children. For example, in bridge design, an arch bridge and a suspension bridge could be combined to form a bridge that uses both arches and steel cables.

Crossover is usually used with a likelihood between .6 and 1.0. If crossover is not used, the parents are directly copied into the next generation, giving the next generation the chance to keep the unaltered version of some (hopefully strong) individuals. This ensures that interesting parts of the search space are likely to be exploited further.

4.6 Mutation

Mutation is used for randomly introducing new gene values. This way the search space will be explored outside the search space that is represented by the genes of the parents. This prevents the overlooking of solutions, by introducing a small amount of random search. The mutation rate is part of the trade-off between the exploration and exploitation of the algorithm. The exploitation of points in the search space means that gene values of good solutions are re-used and recombined to find better solutions. The exploration refers to randomly trying new points in the search space in the hope that an interesting solution can be found.

4.7 Convergence

The constant selection of the best individuals ensures that all the chromosomes in a generation will converge increasingly for each iteration (i.e. all the chromosomes will become more and more similar). If the GA has been correctly implemented and tuned, the fittest individual will represent a point in the search space that is close to the global maximum (or minimum, if so desired).

A problem with GAs is that sometimes a few highly fit individuals start to dominate the population quite quickly. This happens if the selection pressure is too high. This phenomenon is called *premature convergence*. If this happens only a small part of the search space is explored, since the recombination of nearly identical chromosomes does not produce very new chromosomes.

The opposite phenomenon is called *slow finishing* and is the consequence of an overly low selection pressure. In this case the individuals will not be able to locate the global maximum in a reasonable time because the fitness function does not sufficiently push the GA towards the maximum.

4.8 Epistasis

Epistasis refers to the interaction between genes in which the expression of one gene can influence the fitness of an individual depending on what gene values are present elsewhere. In nature we see the same phenomenon. For example, if a bat has the correct genes for making high frequency sounds but not the correct genes for hearing these sounds properly, it is still not able to locate its prey by echolocation. This means that the effects of the genes that govern the making of sounds are masked if the genes for hearing these sounds are not present. In all GA problems there is some form of epistasis, since otherwise all the gene values could simply be optimized independently. If the epistasis is too high however, a small change in a chromosome can lead to large and unpredictable variations is fitness. This means that fit parents do not have a higher chance of producing a fit child than unfit parents. This renders a GA useless. A solution to problems with high epistasis can be to code the problem in a different way. Usually this involves designing a more complex representation of the problem for which more genes per chromosome are necessary. This increased complexity and increase in search space is compensated by the fact that the algorithm can find a good solution more easily because it is not as likely to be lead away from the most promising part of the search space.

5 The proposed algorithm

The focus of this research project is to design an algorithm that can manage uncertainties about the knowledge of the student that are the consequence of ambiguities in the answer of the student. This is done by using a number of concurrent hypothetical student models that each represent a possible interpretation of the answers of the student. In this section I shall first give a brief summary of the Intelligent Tutoring System that forms the context for which the proposed algorithm is intended. After that I shall present the motivations for trying this approach and describe how exactly the proposed algorithm works. Some more expected advantages are given and finally the research questions that were used to guide the research are posed.

5.1 Context

The idea for the technique that is proposed in this chapter for keeping track of a student model originally arose from an Intelligent Tutoring System (ITS) that was partly developed as a short-term project at the faculty of Artificial Intelligence at the University of Groningen (The Netherlands). In this project the focus was on generating problems and their solutions for the domain of mathematical differentiation. Several criteria could be set to manage the complexity and the topics used for the problems. Although the algorithm that is described here has been designed to work for student modelling in any ITS, this particular context influenced some of the specific design choices and therefore provides a useful context for making a concrete example student model. Another purpose of fully designing the system was to give an insight in the difficulties of the field of ITS and to provide a basis on which further research can be done. In this paragraph only a short description will be presented. Due to time constraints, the program could not be fully implemented and it was decided to follow a more basic proof-of-concept strategy to test the performance of the algorithm.

The domain of the ITS that forms the context for this algorithm, mathematical differentiation, is basically a technique to algebraically transform a formula into another formula called its derivative. A short overview of the topic can be found in the appendix. Since several steps are necessary to transform a formula into its derivative, it was decided to first let the student try and solve the problem without any help, and only provide a step-by-step explanation at the moment the system detects that a mistake was made. The student is then taken by the hand and led to the correct solution, in the hope that he will be able to perform all the steps autonomously in a later stage. For the student this means that he will not spend more time than necessary on problems that are not difficult for him. It will allow him to perform some of the steps in his head without the need to make all the steps explicit. If, however, the student makes a mistake then apparently he needs more guidance and each step leading to the correct answer is explained interactively. An advantage from the student modelling point of view is that the system can pinpoint which steps are difficult for the student, by observing the time spent, the mistakes made, and the hints requested for each step. This approach has been called the poor man's eye tracker

(an eye tracker is a device used to track at which part of the screen the student is looking), because it gives an indication about what the student is thinking about at each moment (Martin and VanLehn 1993).

A screen shot from the demonstration version of the system can be seen in Figure 5-1. In this example the student could not find the correct derivative of (2 sin x^2), and therefore this screen was presented to the student to explain how the correct answer can be deduced. In the screenshot the student has solved the problem by filling in all the input fields. It can be seen that the way to the correct answer can be found by an iterative process, which is essentially the skill that is taught. The input fields are filled in from left to right, and from top to bottom. For every line the student needs to indicate which rule needs to be used next to simplify the formula. He then needs to decide whether it is necessary to use the chain rule. On the basis of this information the student needs to recall what the general format for the rule is. Finally, by specifying what the parameters (*c*, *n*, and/or *U*) are for this general format, the formula can be simplified. For each input field it is recorded how long the student takes to fill it in, whether a hint was requested and whether a mistake was made. The data structure that stores these values is called a set of *student answer features.* These are a summary of the observations that were made about the student. From these student answer features deductions can be made about the student knowledge, and how this knowledge changes over time. This data is stored in the student model. The next problem will be generated on the basis of this student model, and in this way the program can adapt to the level of the student. The next problem that is generated can be tailored to the student's need, by selecting the appropriate level and focussing on the type of problem that is expected to be most beneficial to the student.

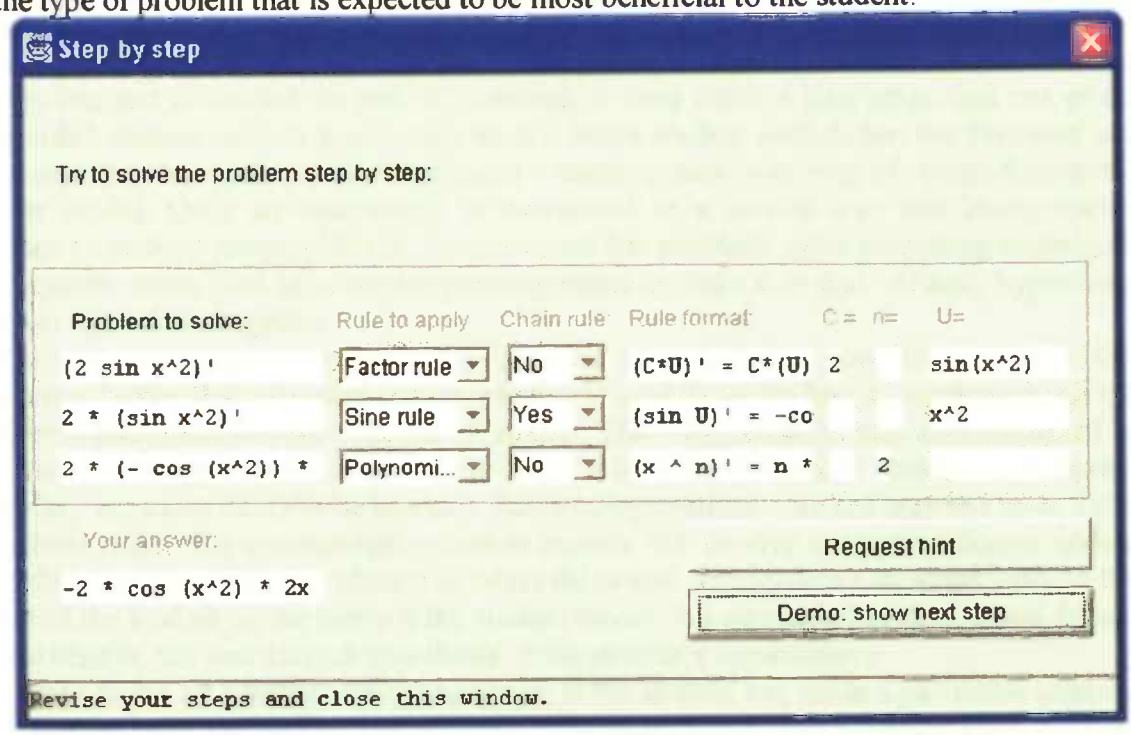


Figure 5-1 A screen shot of the step-by-step explanation of the demonstration version of the ITS that was used as a guide to design the algorithm. The student has to find the derivative of (2 sin x^2). For each transformation the student needs to specify which rule needs to be applied, whether the chain rule is applicable, what the general format of the rule is and how this general format is applied to this particular problem.

5.2 Motivation

The system described above provided the context for the research that is described in this thesis. The focus of the research, however, is on modelling the student. In our approach, we made use of a genetic algorithm to model the student. This is quite an unusual approach, and in this subsection I shall give the reason why we were interested in exploring this way of student modelling.

The key observation that inspired this approach is that it is not always possible to interpret the actions of the student in an unambiguous way. Often several conclusions can be made from a single observation. According to Jameson this is a problem that frequently arises in student modelling (Jameson 1996). Such situations can arise on a number of occasions. For example, a student makes a certain mistake that can be explained by difficulties with two different concepts. How does one adjust the student model in such a situation? Or if a student reads a text for a long time, does that mean that he now knows the topic well or does that mean that he has a lot of difficulties with this subject or even that he was simply distracted? If a student seems to know a particular concept, does that mean that he will also know a different concept that is very similar? If a student hasn't practised for a week, does that mean he has forgotten some topics? If so, to what extent has he forgotten them? We would like to be able to find a way to handle these ambiguities.

Since the only source for building a student model are the observations of student actions, it is possible to make several deductions about the student's knowledge after several observations (i.e. there can be several possible student models). One way of dealing with this ambiguity is to make a decision about which student model is the most likely one and to discard the rest. If, however, it turns out at a later stage that one of the discarded student models would have been a better student model than the favoured one (because the observations were interpreted wrongly), there is no way of reintroducing this better model. Once an observation is interpreted in a certain way this interpretation cannot be undone retrospectively. To overcome this problem, most modelling techniques incorporate some kind of uncertainty management to make sure that unlikely hypotheses are not excluded altogether.

The genetic algorithm approach is intended as a different way to overcome this problem. Rather than selecting one of the student models as the best interpretation, all the different interpretations are retained in parallel. The interpretations that are considered to be the best fitting at some stage are rewarded with a higher fitness value, which means that they are more likely to be retained. Some interpretations may not turn out to be valid at a later stage. The corresponding student models will receive a negative fitness update and therefore have a higher chance of being discarded. The system can adapt itself to the level of the student on the basis of the student model that currently has the highest fitness value (that is, the best current hypothesis of the student's capabilities).

Human tutors do a similar thing implicitly: if the student has made a particular mistake that suggests that the student has problems with either one of two concepts (say, concept A and concept B), both possible explanations are considered a possibility until the student proves to grasp either concept A or concept B. The alternative explanation is then used as the best hypothesis at that moment, and the tutor selects the next problem accordingly.

5.3 The algorithm

Now that the basic context of the system has been explained, it is possible to describe how exactly the proposed algorithm works. Since the algorithm described here is fairly complex and not all the modules that were necessary for the algorithm to work could be finished in the time that was available, the decision was made to implement only a simplified version of the algorithm (as described in chapter 6). This way it is possible to find out to what extent the basic principle works, and to see if the simplified algorithm could be extended to the complete algorithm that is described here.
A schematic overview of the algorithm is given in Figure 5-2. I shall use this figure as a guideline to explain the algorithm in more detail. Some genetic algorithm terminology will be used (see chapter 4), for an overview of the terms used please refer to the appendix.

Process 1: Ask the student for his level of expertise.

Before the iterative part of the algorithm starts, the student is asked for some basic details to make sure that the algorithm is not initialized completely at random. By setting some expectations about the student's knowledge an appropriate student model can be determined more quickly. The student can choose between a beginner, intermediate, expert or revision level, so that the first problems that are presented are more likely to be at an appropriate level for the student. It is also possible to let the student specify at a more detailed level what topics he is familiar with.

Process 2: Initialize a generation of stereotype student models.

A generation (in the genetic algorithm sense) of stereotype student models is loaded from file based on the level that the student selected. The stereotype student models in this case are student models that are likely to be close to a model that is realistic for the student: if for example the student indicates that he is a beginner, this will be reflected in modest expectations about the student's capabilities. This can be compared to expectations that a human tutor may have on the basis of the level that a student claims to have.

Stereotype student models can be generated by storing snapshots of the student models of previous students in a database. This is an efficient way of filling in the blanks of a student model: if a student shows particular characteristics in the first couple of problems he or she makes, a student model that incorporates these characteristics is selected automatically. At the same times some expectations are set for other characteristics of the student (Pohl 1996).

For our domain the student model consisted of predictions about the answers of the student: how long it takes him to solve a problem, whether he makes a mistake and how likely it is that he requests a hint. In other domains different representations may be used.

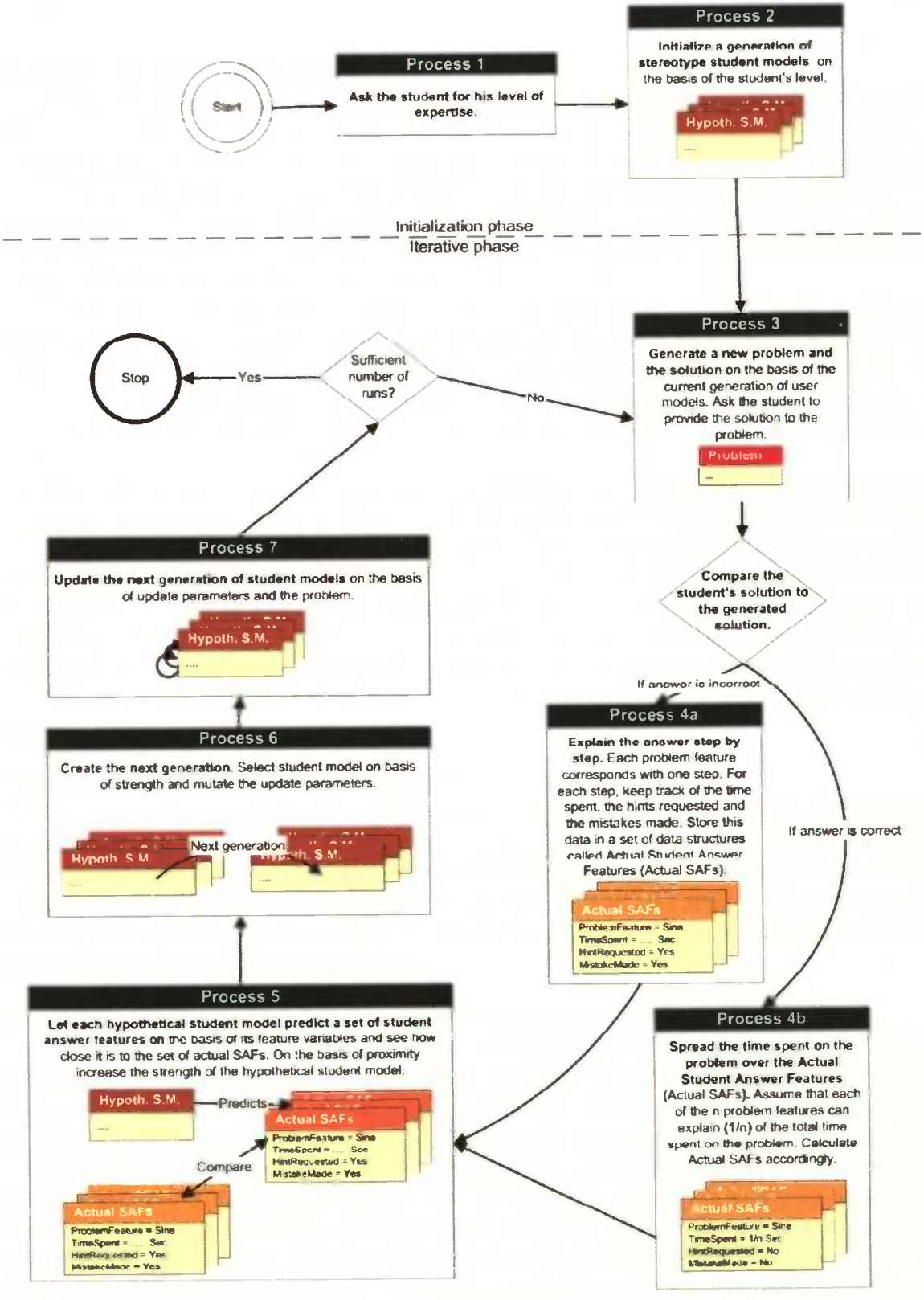


Figure 5-2 Schematic description of the proposed algorithm.

Process 3: Generate a new problem and the solution.

The next problem is generated on the basis of the current generation of student models. Since there are several concurrent hypothetical student models and the pedagogical module needs to make a decision based on a single student model, it is necessary to calculate a summarized student model that is representative of the entire generation of student models (with a bias to the fitter student models). It is possible to simply use the fittest student model or to take an average of a number of the fittest student models (by simply averaging all the values of these models).

The pedagogical module then needs to make a decision about what to teach next based on this summarized student model. This is not a trivial task. The pedagogical module needs to make decisions about the level and complexity of the next problem, when to introduce a new topic, which topic to introduce next, when to revise, what hints to show, etc. (Beck and Woolf 1998). Quite a lot of knowledge engineering is necessary for this step.

Once a decision is made about what the features of the next problem should be it is necessary to generate the next problem with these features or read a problem with these features from a database. In any case the problem, the path to the solution, the appropriate hints and the correct solution need to be obtained. The student is then asked to provide the solution to the problem, and the student's solution and the correct solution are compared. If the student's solution is incorrect, the path to the correct solution is explained as described in process 4b. If the student's solution is correct, the student models are updated as described in processes 4b to 7 and the next problem is presented to the student.

Process 4a: Explain the answer step by step.

If the student is not able to solve the problem correctly, the correct path to the solution is explained step by step. This is the process that is shown in Figure 5-1. For each step the characteristics of the student's answer are stored in a data structure called the *Actual Student Answer Features* (Actual SAFs). More specifically, this data structure stores the time that the student spends on a step, the hints requested for this step and whether or not the student made a mistake for this step.

Process 4b: Spread the time spent on the problem over the Actual SAFs.

If the student is able to solve the problem without any mistakes, the student is simply offered the next problem that is slightly more complex. To update the student models, the Actual SAFs still need to be calculated. A problem that arises here is that of assignment-of-credit (Sleeman and Brown 1982): if a number of elementary steps are needed to solve a problem, how do we assign the total time that was spent to these elementary steps? We have chosen to use the simplest solution to this problem: suppose a particular generated problem contains n different problem types (meaning that n identifiable topics are contained in this problem). We then assume that every problem type can explain $(1/n)^{th}$ of the total time spent on the problem. Problem types that occur more than once are assigned credit proportionately. The Actual SAFs for each problem type are calculated accordingly.

Note that the assignment-of-credit problem is avoided by using the step-by-step explanation in process 4a.

Process 5: Let each hypothetical student model predict a set of student answer features.

Every student model has certain expectations about how the student will perform on a problem with certain problem types. More specifically, each student model can predict a set of SAFs that describe the predictions of how a student will perform for each step of the path to the solution. These *Predicted SAFs* are similar in structure to the Actual SAFs, and the distance between the two can be calculated to give an indication of how accurate each student model is in predicting the behaviour of the student. This distance can be calculated by simply adding all the differences between the Predicted SAFs and the Actual SAFs. For Boolean values (whether the answer was correct for example) a scaling factor is necessary. The better the prediction of the student model, the more the student model will increase in fitness. Therefore the fitness value of a student model indicates how well the student model has been able to predict the student behaviour.

Process 6: Create the next generation of student models.

In this process the genetic algorithm part of the cycle starts. This step is to ensure that the best predicting student models survive and that the poorly predicting student models die out. The fittest (best predicting) student models have a higher chance of being copied into the next generation more than once. The less fit student models are more likely not to be copied into the next generation at all. This ensures that the more reasonable student models (namely the ones that have made good predictions in the past) are maintained as good hypotheses about the student's knowledge.

Some student models make better predictions about the student's answers than others because some student models are updated more effectively on the basis the student's previous answers. In other words, every student models interprets the observations in a different way. This way, every student model has a different hypothesis of the knowledge and skills of the student. How a student model interprets the observations is governed by *update rules*. For example, an update rule might be that if the student has improved problems based on the sine rule, problems based on the cosine rule will be easier for the student as well. To quantify how the update rules update the student model *update parameters* are used. The general idea of the algorithm is that several rules are made by the designer of the software that specify how a student model can be updated on the basis of observations. The designer effectively gives some suggestions about where in the search space to look for a good student model. The algorithm is then able to find which of these rules prove to be effective for updating the student models and what the update parameter settings for these rules should be. In this way it is possible to let the algorithm discover in which way the knowledge of the student changes on the basis of practice. The advantage is that the rules by which the knowledge changes are meaningful to the designer of the program. This is not the case in many machine learning techniques that are used for student modelling.

Note that there is a difference between update parameters and parameters of the genetic algorithm. The update parameters indicate how the student models are updated and the parameters of the genetic algorithm are used to modify how the algorithm itself works, for example how eager the algorithm is to explore new search space.

From the preceding paragraphs it is made clear that each student model consists of three parts. First of all, there are the variables that describe the knowledge that a student has at some stage. These variables form the basis for predicting the SAFs. These variables shall

be called *student model features* or simply *features* from now on. Secondly there are the update parameters, which are used together with the corresponding update rules to update the knowledge variables on the basis of the observations about the student. Finally, there is a variable for the fitness of a student model, which reflects how well this model has been in the past at predicting the student's answer.

Process 7: Update the next generation of student models.

After creating the next generation of student models, it is necessary to update the student models on the basis of the update rules. An example of such an update rule is that the time spent on a sine rule becomes smaller if the cosine rule is practised, because of the similarity between the two rules. The fitter student models have more children in the next generation than the less fit student models, but because the update parameters were randomly mutated slightly for each child, no two student models will interpret the observations in exactly the same way. This effectively means that the children will be similar to the parents, but will start interpreting the observations in a slightly different way so that every child will explore a slightly different part of the search space of student models. In this way it is hoped that some of the children will be more effective in interpreting the observations than their parents.

After the next generation of student models is updated, it is necessary to assess whether the student has practised enough. This decision is made by the pedagogical module and depends on the goals that the student is expected to achieve. If the student has practised enough the algorithm terminates, otherwise the next problem is presented to the student.

5.4 Potential advantages

In subsection 5.2 the basic motivation for doing this research was given. Now that the system is described in more detail, I shall give some more specific potential advantages that motivated doing this research project:

- The strategy may be useful for plan recognition problems. In plan recognition problems it is often difficult to deduce what the student is planning to do, because different plans may have overlapping components. Both possible plans that the student is following may be retained in memory until further evidence is found that can only be interpreted as an action fitting only one of the plans.
- The student model does not only describe a snapshot of the student's knowledge at a particular moment, but it also describes how this knowledge has changed in the past. It therefore provides extra information that can be used in the ITS, for example to adapt the system to the individual student's learning style. This adaptation is not possible in Bayesian networks, because with that technique only static knowledge is modelled (Reye 1996).
- If no previous information about the student is available, several prototype student models can be tried at the same time. Since no definite commitment to any student model is made, the best fitting hypothetical student models will be automatically selected, ignoring the incorrect hypotheses.
- Machine learning techniques that have been used usually do not give an insight into why particular connections between data have been generated. This approach offers a

more knowledge-based strategy, in which machine learning techniques are used to find parameters of update rules that are meaningful to humans.

- The update rules are very flexible. Rules could be made to apply to all the features at the same time (for example if a student just logs in, it can be assumed that some knowledge is lost because of lack of practice), they could be specialized for a group of features (for example if a student gets better at a particular topic, it may influence related topics) or they could be specialized for a single feature (for example if the student practises on a particular problem, it can be assumed that the student gets better at solving this type of problem).
- An XML based language could be used to specify the update rules, the observations and the student model so that it is simple to make a general framework that could be used for other domains as well.
- Genetic algorithms are relatively insensitive to noise. This is important, because user actions may not always be consistent with the actual knowledge and capabilities that the student has.
- Genetic algorithms are adaptive by nature. Since the student knowledge changes continuously and the evaluation function is dependent on the student performance, it is possible to continuously track changes in student knowledge.
- Genetic algorithms are known to be robust: The exact settings of the different parameters of the algorithm itself (such as mutation rates) do not usually have a big influence on the overall performance of the algorithm (Beasley, Bull et al. 1993). This kind of robustness is very desirable since the data of the observations of the student is quite noisy.
- Genetic algorithms are easy to implement and easy to adapt once a framework has been made. This makes it easy to use this approach in other teaching domains.

5.5 Research questions

The following questions have been composed to guide the research:

1. Is it possible to generate an accurate student model using the proposed technique based on a genetic algorithm?
2. To what extent can this approach be generalized to other teaching domains?
3. What are the strengths and weaknesses of this approach, and how do these compare to the strengths and weaknesses of other approaches?
4. Are there ways in which this approach can complement other approaches?

Ad. 1: Since this is a novel approach, naturally the first question that we are interested in is the feasibility of the approach: does the algorithm behave as expected? How robust is it? Does it need a lot of fine-tuning? Are the generated problems reasonable considering the level of the student? It should be stressed here that the research is intended mainly as an exploration, to offer a different approach to the well-trodden paths of student modelling, in the hope that it may give a different point of view into the problems encountered in this domain.

Ad. 2: Considering the expenses involved in designing and implementing an ITS, it is of crucial importance that the approach is not too focused on the domain at hand: this

would mean that a lot of reprogramming needs to be done to use the same technique on a different domain, which is undesirable. It should be possible for any student modelling technique to be used for other domains than it was originally designed for, since otherwise the effort for designing the system would be too costly. Ideally, when an ITS is implemented for a new domain only little programming knowledge, code hacking and knowledge engineering is necessary, so essentially the same framework can be used with different parameters. This consideration has been important in the design of both the algorithm and the ITS.

Ad. 3: In the field of student modelling often new techniques or alterations to techniques are suggested on the basis of shortcomings in other techniques or to overcome a particular problem in a domain. The ensuing papers that are written usually point out the strong points of the proposed approach, without comparing the technique to existing ones that have often already proven their merits. This question is aimed at avoiding this situation: it is about how the research fits in to the previous research done, by comparing the merits and shortcomings of different approaches.

Ad. 4: This question is a logical consequence of question 3: since every student modelling approach has its own strengths and weaknesses, is it possible to combine the suggested approach with any existing approach to use the strengths of both and overcome (some of) the weaknesses of both?

6 Evaluation of the algorithm

In this section I shall explain how we have evaluated the proposed algorithm by simplifying it and testing it with artificial students. First I shall give a description of how student modelling techniques are usually tested, then I shall describe why we chose to test the algorithm with artificial students and finally I shall describe the simplified algorithm that we used for evaluating the proposed algorithm.

6.1 Testing effectiveness of student modelling

To test the effectiveness of a student modelling technique, usually an indirect method is used with which the capabilities of the students are tested before and after they have used the system. This data is then compared to a control group that followed instruction of the same topic in the conventional way. Often it is found that using the system at hand helps the students to master topics much faster compared to conventional methods (Shute, Glaser et al. 1989; Mark and Greer 1991; Koedinger, Anderson et al. 1997). This result is often implicitly ascribed to the student modelling technique that was used for making the system (after all, if that is what the research is about, that is what is to be tested). It is, however, not unthinkable that most of the improvements in the student's capabilities may be ascribed to the interactions with the system itself, and not so much to the *adaptations* (based on the user model) made by the system. Research in which a computer-based adaptable teaching system is compared to a similar non-adaptable computer-based system (to exclude influences that are not ascribed to the student model) is quite rare.

6.2 Testing with artificial students

In our case it was not feasible to test the student model on students altogether, due to time constraints. To be able to test the student modelling capabilities of the algorithm a separate program was implemented. This way it was possible to focus only on the student modelling, while excluding the factors mentioned in the previous paragraph.

The interactions between the algorithm and a human student were simulated by implementing *artificial students*. This testing technique is quite common in the context of ITSs, especially with systems that use machine learning. The idea is that if it is possible to track the changes of knowledge of an artificial student using this approach, it may also be possible to track the changes of knowledge of a real student. An artificial student is no more than a student model that changes on the basis of the problems that are posed to the student. The student model of an artificial student therefore represents the *'true' student model*: the knowledge states that a student actually has (as opposed to the hypothetical student models that predict the student's actions). For each artificial student different behaviours are defined by specifying how its 'true' student model changes in reaction to problems that are offered. The change in knowledge and therefore in the behaviours of the artificial student is defined using update rules that are similar to the update rules of

the hypothetical student models. The big advantage is that a statistical analysis of the functioning of the algorithm is possible. We can compare the student model features of the 'true' student model and the hypothetical student models. Also it is possible to compare the update parameters of the 'true' student model to the update parameters of the hypothetical student models. We expect to find that a student model is able to make correct predictions because it has found a correct way of updating its values. More concretely, if a student model is accurate at describing the knowledge of the artificial student (which is the basis for the predictions made by a student model), does this mean that the update parameters were similar to the update parameters that were used for the artificial student?

The update rules as they are implemented serve only as examples of how student knowledge might change. The framework itself is tested and not so much the specific update rules. Before this framework can actually be used, some observations about how the knowledge of human students changes need to be made and implemented in the framework. The basic claim is that there is some sort of logic (in the colloquial sense) to how the student's knowledge changes on the basis of problems with which the student practices. We will not make any claims about what exactly this logic is; the tests are aimed at proving whether the framework itself could be valid and whether it could predict the expected student model. In other words, the research is aimed at a proof-of-concept: if the simple test cases with our defined logic work, it is still not yet proven whether the actual algorithm will work in a classroom situation. Problems that are encountered with this proof-of-concept strategy may provide insights into challenges that need to be overcome.

6.3 The algorithm used for testing

In order to test the principles of the algorithm proposed in the previous chapter, a number of simplifications were made. This was done for two reasons. First of all, the algorithm needs several modules that were not yet implemented. A second reason is that the more modules play a part in the algorithm, the more difficult it becomes to analyse the data, because several explanations can be given to explain one phenomenon. Once the simplified algorithm is tested, it is possible to interpret the data and suggest in which way the algorithm could be extended.

The most significant and time saving simplification of the algorithm has been one of scale. By decreasing the number of types of problems that can be generated, the student model can be much smaller. This in turn simplifies the coding of the update rules, since no general framework is needed anymore. Furthermore it was decided to leave out the pedagogical module that makes intelligent decisions about what type of problem to generate on the basis of the current student model. Instead, the problems are generated at random. This eradicates the interaction between the algorithm and the pedagogical module, which potentially makes the data more difficult to interpret.

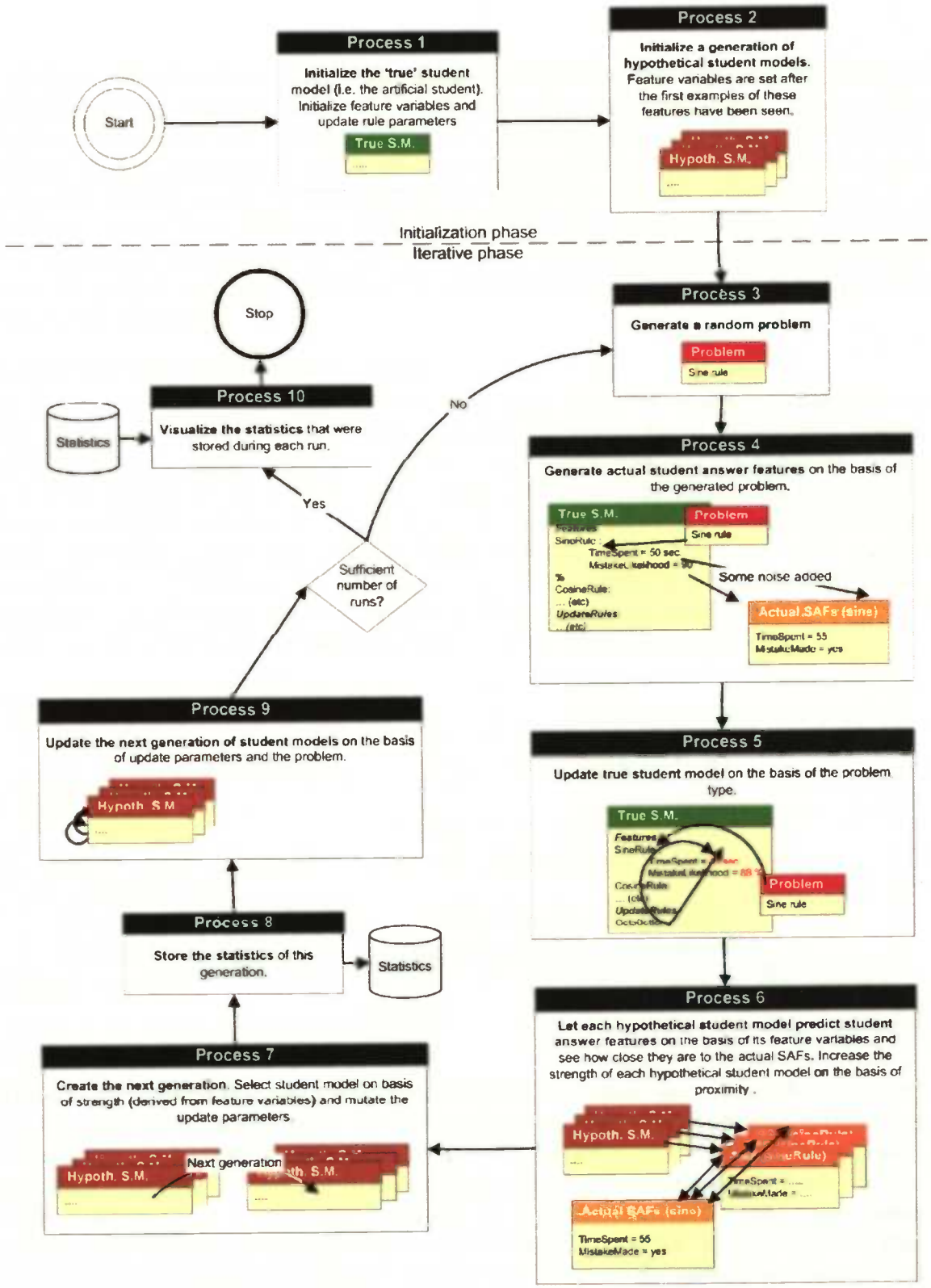


Figure 6-1. The algorithm used for testing.

The most straightforward way of describing the algorithm that was used for testing is again graphically. Figure 6-1 depicts the most important processes of the algorithm. This diagram provides a good starting point for describing the algorithm. In the black 'process' boxes a short description of each sub-process is given. The coloured boxes depict instances of objects (in the programming sense) that are used in the program. I s

hall now give a more elaborate description of these processes and objects in order to give a better understanding of the algorithm. Several of these processes have been varied in some ways to test some properties of the algorithm. These variations shall be described in the results chapter.

Process 1: Initialize the true student model.

Before the iterative part of the algorithm can start, the 'true' student model must be initialized. This is the student model that describes the knowledge of the artificial student and how this knowledge changes on the basis of the problems that are presented to the student. This is essentially the same data structure as the data structure of the hypothetical student models (note that student models are always hypothetical, the term hypothetical student model is used to stress that a different student model from the 'true' student model is intended). There are some differences in the way that the 'true' student model and a hypothetical student model are updated. These will be described later.

The 'true' student model is a data structure consisting of the following values:

Sine rule:	
TimeSpent =	... (*value in the interval [0,100]*)
MistakeMadeLikelihood =	... (*value in the interval [0,100]*)
Cosine rule:	
TimeSpent =	... (*value in the interval [0,100]*)
MistakeMadeLikelihood =	... (*value in the interval [0,100]*)
Natural logarithm rule:	
TimeSpent =	... (*value in the interval [0,100]*)
MistakeMadeLikelihood =	... (*value in the interval [0,100]*)
Update parameters =	[..., ..., ..., ...] *(four values in the interval [0,100])*

Table 6-1: Structure of the 'true' student model.

The sine rule, cosine rule and natural logarithm rule are three types of problems that can be generated. A problem type, in our domain, corresponds with the name of a differentiation rule that can be used to solve a problem (as a simplification only elementary problems are offered to the student; normally it would be necessary to use several rules to solve a problem). Naturally, in a realistic student model many more problem types than the three implemented here would be possible. The timeSpent value indicates how long a student will need to solve a problem of the corresponding problem type, and the mistakeMadeLikelihood value indicates how likely it is that the student will make a mistake while solving the corresponding problem type. These values are referred to as student model features or simply as features as was mentioned in the previous chapter.

The update parameters are parameters that influence how the student model is updated after a problem of a particular type is presented to the student. This reflects the changing knowledge of the student. A description of how exactly these update parameters influence the updating of the student model will be given later on in process 5.

Process 2: Initialize a generation of hypothetical student models.

A hypothetical student model consists of almost the same values as the true student model, with the only difference being that the hypothetical student model also has a value for its fitness. We usually used 100 student models to fill one generation.

Sine rule:	
TimeSpent =	... (*value in the interval [0,100]*)
MistakeMadeLikelihood =	... (*value in the interval [0,100]*)
Cosine rule:	
TimeSpent =	... (*value in the interval [0,100]*)
MistakeMadeLikelihood =	... (*value in the interval [0,100]*)
Natural logarithm rule:	
TimeSpent =	... (*value in the interval [0,100]*)
MistakeMadeLikelihood =	... (*value in the interval [0,100]*)
Update parameters =	[..., ..., ..., ...] *(four values in the interval [0,100])*
Fitness =	... *(float value)*

Table 6-2: Structure of a student model.

This fitness value is used to indicate how successful the hypothetical student model has been in the past in predicting how the student will perform in solving a problem.

Since the hypothetical student models do not initially have a way to make a good guess about the update parameters, these are initialized at a random value.

The features, however, can be set the moment that the student has tried to solve the first problem of a particular problem type. If, for example, the student has made a mistake the first time he has encountered a sine rule and took 20 seconds to fill in an answer, the hypothetical student model features are set to values similar to the observed values. Since the observed values are noisy values (the student will sometimes take a little longer to solve a problem than other times), each student model initializes its features *around* the observed values. The student models that have made the correct guess will become fitter, whereas the student models that have made a worse guess become less fit (and as a consequence will more likely die out).

Process 3: Generate a random problem.

As mentioned before, a problem is generated completely at random. The problem object in this algorithm is simplified in a number of ways: it does not actually generate a problem or its answer. A problem consists of only one feature (its problem type) and the problem has no level. It is quite straightforward to implement a more realistic problem object from this starting point, but in our testing case the simplified object sufficed.

Process 4: Generate actual student answer features.

The actual student answer features (actual SAFs) are generated on the basis of the generated problem and on the basis of the 'true' student model of the artificial student. A student answer feature is the data that describes how the student performs on a problem: whether a mistake was made and how long the student takes to fill in the answer. The actual SAFs therefore describe the observed behaviour of the artificial student. Because a student will never solve the problem in an exactly predictable way (the student might be distracted for example), some noise is added to these actual SAFs.

Process 5: Update the true student model.

Here the 'true' student model is updated on the basis of the problem type of the problem that was just presented to the student. As a simplification of the algorithm it does not matter whether a student solves the problem correctly or not. It is assumed that the student will always learn a little bit by seeing the problem.

How the student model changes exactly, is defined by the update rules and the update parameters. The update rules (and corresponding update parameters) that were implemented for this version of the algorithm were:

1. TimeSpent decreases: If the student has worked on a problem, it will take *updateParameter[1]* seconds less time the next time the student needs to solve this type of problem.
2. MistakeMadeLikelihood decreases: If the student has worked on a problem, it will be *updateParameter[2]* less likely that a mistake is made the next time the student needs to solve this type of problem.
3. Forget rules (I): Every time the student does not work on a problem (i.e. is working on another problem), the student tends to forget the rules needed for solving this type of problem a little due to interference, and the timeSpent value for this problem increases by *updateParameter[3]*.
4. Forget rules (II): Similar to forget 1, but the mistakeMadeLikelihood increases by *updateParameter[4]*.

Update parameters 1 and 2 are based on the basic observation that a student will improve with practice. Update parameters 3 and 4 are based loosely on the data-ageing effects described in (Webb and Kuzmycz) and on ACT-R based models (Anderson and Lebiere 1998). The algorithm is intended solely as a proof-of-concept, however, and any rules that have a stronger foundation in empirical research can be implemented. These rules were selected for two principal reasons. The first reason is simplicity. For the first test case, the simplest rules were selected to test whether the algorithm could realize the expected behaviour. The second reason is that we wanted to test whether the algorithm works if there are rules that affect just one feature at the time and rules that affect more than one feature at the time.

Process 6: Each hypothetical student model predicts student answer features.

At this moment each hypothetical student model predicts student answer features (SAFs) on the basis of the values that it currently holds. These SAFs are then compared to the actual SAFs that were generated by the 'true' student model of the artificial student, and the distance between the two sets of SAFs is calculated. On the basis of the distance the fitness is updated: the smaller the distance and thus the more correct the prediction of the answer that the artificial student gives, the fitter the student model

becomes. This score depends on only one feature of the student model (depending on the problem type at hand), and therefore the fitnesses are updated cumulatively. This way the predictions of the student model in the past also influence fitness of the current student model. This ensures that a student model is not immediately discarded the moment that it makes an incorrect prediction. This would be undesirable, because a student model consists of several sets of student model features (one for each problem type that can be encountered) and the SAFs are based on only one set of these student model features. It is not necessary to completely discard an entire student model just because the beliefs for one problem type are wrong.

Process 7: Create the next generation.

In this process the next generation of hypothetical student models is created. For this algorithm no cross-over mechanism was used: the exploration of the search space is done purely by a mutation mechanism (the GA term for this is naïve evolution). The reason for this is that there is a relation between the update parameters and the features of a student model. Some update parameters influence several features. If the cross-over mechanism creates children by randomly selecting the parameters and the features from the parents and copying them to the parameters and features of the children, this relation is lost. An alternative cross-over method in which the features and parameters of both parents are averaged to retain some of the relation, would lead to a quick convergence, and is therefore also unsuitable.

Only the update parameters and not the feature values are mutated. The feature values are only changed via the update parameters. This is done because we are trying to find a way to update the student model in a correct way. The update parameters can be thought of as the genotype that influences the phenotype (i.e. the feature values). The phenotype is used to select the individuals, and the genotype is the part of the data structure that is modified by the genetic algorithm.

The mutation mechanism uses the creep operator (Beasley, Bull et al. 1993). This means that rather than resetting the update parameters completely at random, the update parameters are increased or decreased by a small value. This is again to preserve the relation between the update parameters and the feature values. The update parameters of the child will look similar to the update parameters of the parents, meaning that each individual (in the genetic algorithm sense) explores the search space in the vicinity of the search space that the parents have explored.

Process 8: Store the statistics.

To gain an insight in how the algorithm performs, it is necessary to store some statistics of every generation. For each generation the distances between the features and parameters of the 'true' student model and the hypothetical student models are calculated. More specifically, the following values are calculated and stored:

- Average feature distance: The feature distance between the 'true' student model and a hypothetical student model is the sum of *all* the distances between the student model features. This average feature distance, then, is the average of all the distances between 'true' student model and hypothetical student models in a generation. This value gives an indication of how well the student models can predict how the student will perform on a question.

- Best feature distance: The lowest feature distance that was measured in a generation.
- Fittest feature distance: The feature distance between the fittest student model and the 'true' student model. This distance can be compared with the average feature distance and the best feature distance: if the algorithm most often assigns the highest fitness to a student model with a feature distance between the average and the best feature distance, the average feature distance should become smaller.
- Average parameter distance: Similar to the average feature distance, but the distance measured is between the update parameters. This, together with the average feature distance, gives an indication of the relation between the genotype and the phenotype. We are interested in the question whether it is possible to find the correct genotype (i.e. how the student model is updated on the basis of the update parameters) by ensuring that the phenotype (i.e. the feature variables) of the hypothetical student model comes closer to the phenotype of the 'true' student model. Since in real life only the student answer features that are influenced by the 'true' student model can be observed, measuring the 'true' student model can only be done indirectly.
- Fittest parameter distance: The parameter distance between the fittest student model and the 'true' student model. This value can give a similar insight as the average parameter distance: If the fittest *feature* distance is lower than the average feature distance, does this mean that this student model also has a lower parameter distance? In other words, if the phenotype matches better than average, does this indicate that the genotype matches better than average?
- Average fitness: the average fitness of the student models in a generation. The interpretation of this value depends on what function is used for updating the fitnesses.

Not all of these statistics are presented in the results chapter; some were only used to verify the correctness of the algorithm and to gain a better insight into how the algorithm works.

Process 9: Update the next generation of student models.

After the next generation has been created, it is necessary to update the student model features of the newly created student models on the basis of the update parameters. This process is similar to updating the 'true' student model (process 5).

Process 10: Visualize the statistics.

After a specified number of iterations (we used between 400 and 800) of the algorithm have been executed, the behaviour of the algorithm can be analysed by visualizing the stored statistics. To achieve a smoother graph the entire algorithm can be run several times and the statistics averaged. This is a better way to gain an insight in the trends, without the distractions of local fluctuations caused by the chance processes in the algorithm.

7 Results

In this chapter the results of testing the algorithm shall be presented. To avoid any unintentional behaviour caused by an inaccurate implementation of the algorithm, all the modules of the algorithm have been thoroughly tested and the algorithm has been run satisfactorily with a simplified updating function.

The testing of any genetic algorithm cannot be seen as a direct goal-oriented process: often minor modifications are made to try out different hypotheses about the observed behaviour. These modifications may then lead to new observations that need to be explained, and so forth. Furthermore, the parameter settings of the genetic algorithm often interact, which makes it difficult to explain with a reasonable amount of certainty why particular phenomena can be observed. Sometimes the increasing of one parameter (for example the mutation likelihood) can result in the opposite effect as decreasing another parameter (for example how much the gene values can be decreased). If both parameters are increased simultaneously, no effect is measured. Therefore, a favourable outcome can only be explained by the combination of all the parameters settings that were used for a run.

To categorize the test data, the subsections in this chapter contain the tests based on the most significant insights and hypotheses.

7.1 Test setups

To evaluate the algorithm a succession of test cases were designed. These test cases begin with the simplest case, and become more complex with every new test case. The intention is to see how robust the algorithm is by gradually increasing the realism of the test situation. The algorithm and the student model used are both simplified versions of an algorithm and student model as they would be used in a realistic situation. Therefore if the tests have a satisfactory result, it does not necessary mean that the algorithm will show satisfactory behaviour in a realistic environment, but it does give a good indication of the behaviour. On the other hand, the algorithm is unlikely to function in a realistic environment if the test cases fail to show the expected behaviour.

The three test cases were designed:

1. Test the algorithm with a static artificial student. A static artificial student can be implemented by setting the update parameters to zero. This means that the student model features will be constant over time. Since a static student model is the simplest student model to be mimicked by the hypothetical student models, this set-up can be used to test the different parameter settings of the genetic algorithm itself. The update parameters of the hypothetical student models should evolve to zero after a number of runs. This test case is intended to gain insight in the basic behaviours of the algorithm.
2. Test the algorithm with a dynamic artificial student. In this case the values of the update parameters of the artificial student will be set to values larger than zero, to mimic the changing knowledge of a human student. By tracking the update

parameters of the hypothetical student models and comparing them to the update parameters of the 'true' student model, it is possible to see how well the hypothetical student models can track the changing knowledge.

3. Test the algorithm in a more realistic setting in which the update rules of the 'true' student model are somewhat different from the update rules of the hypothetical student models. In a real life situation it is not possible to know what the exact processes are that underlie the changing of knowledge. This setting is to try out whether the algorithm is robust enough to model the knowledge change of the artificial student fairly accurately in a situation in which some processes of this knowledge change (represented by the update parameters) are unknown. For example, suppose the practising of the sine rule improves the skills of the cosine rule. If this phenomenon is not modelled in the update rules, is the algorithm still capable of tracking the knowledge change reasonably well?

At the end of the first series of tests, it turned out that it was already quite difficult to model the knowledge of a static student model. After that the second test round was conducted with a dynamic artificial student. The results of this round were not very satisfactory. The third case mentioned above has not been tested, since there is no reason to expect that the results would have been favourable after the simpler second case failed to prove very successful.

7.2 Interpretation of the statistics

In process 10 of Figure 6-1, the results are visualized to be able to interpret how well the algorithm performs. We are interested particularly in two values: the average feature distance and the average parameter distance. As was mentioned before, the average feature distance indicates how well a *generation* of student models can predict how the artificial student will solve a problem. If most of the student models can correctly predict how the artificial student solves a problem, then the average feature distance will be small. Since the algorithm is aimed at selecting better predicting student models, we expect the average feature distance to become smaller over time. Note that in the algorithm the student models are selected on the predictions that they make about the answers of the artificial student, and not on the distance between the features, since in a real life situation there is no access to the real knowledge of the student (which is what the 'true' student model represents). In other words, in the algorithm the feature distance is measured indirectly via the predictions that a student model makes, but for the evaluation of the algorithm the feature distance is measured directly.

The average parameter distance indicates how the hypothetical student models are updated compared to how the 'true' student model is updated. If the average parameter distance is small, this indicates that most student models have been updated in a similar way to the way the 'true' student model has been updated. This means that similar update rules have been used and therefore that the hypothetical student models have been able to track how the knowledge of the artificial student changes.

What we are hoping to find, is that the reason why the hypothetical student models are able to do better and better predictions is that the hypothetical update rules have found a correct way of updating the knowledge on the basis of observations. More concretely, we

are hoping to find a feature distance that decreases over time because the parameter distance decreases over time.

7.3 Test case 1: Modelling a static artificial student

The testing of the algorithm using a static artificial student is considered to be the easiest case to model. Naturally no practical conclusions can be drawn from this test case since the modelling of static student behaviour is a trivial case: after all, the aim of the algorithm is to model how the knowledge of the student *changes*. The intention is to experiment with the different parameter settings of the genetic algorithm itself and to explain the basic observed phenomena.

7.3.1 Test 1.1: Tackling premature convergence

An initial problem with the proposed algorithm was that after a small number of iterations (typically between 5 and 10), the entire population of student models consisted of individuals stemming from only one common ancestor. As we recall this is called premature convergence. This is undesirable, since this means that the later generations only explore a small area of the search space. During the first runs of the algorithm it is beneficial to explore as much of the search space as possible before narrowing the search down to only a small number of areas that look promising.

Premature convergence is a consequence of the fact that better fitting individuals have a higher chance of reproducing. This means that the genes of the fittest individual will most likely have a higher incidence in the next generation. If the individuals based on these genes produce highly fit individuals (which is likely), the process repeats itself and the incidence of these fit genes stemming from the same ancestor becomes higher still. Even the mere process of iteratively and randomly selecting and copying individuals (without the bias to pick fitter individuals) is enough to make this process happen: once an individual is not selected to be copied into the next generation its genotype is lost forever. This process is referred to as *genetic drift*.

A major cause of premature convergence that is specific to this particular algorithm is that there is no crossover operator. Normally when two parents are crossed over to form a child, the genes of the child consist of a combination of the genes of both parents. This means that the child will explore a new area of the search space that is different from the areas that are explored by the parents. Cross-over is not a sensible operator in this algorithm (because it destroys the relation between the update parameters and the student model features, as was mentioned earlier), and therefore only the creep mutation method is used. This method ensures that the search space of a child will always be close to that of the parent. The process of selecting individuals on the basis of their fitness ranking ensures that some gene combinations die out, because the less fit individuals have a low chance of reproducing.

Solution

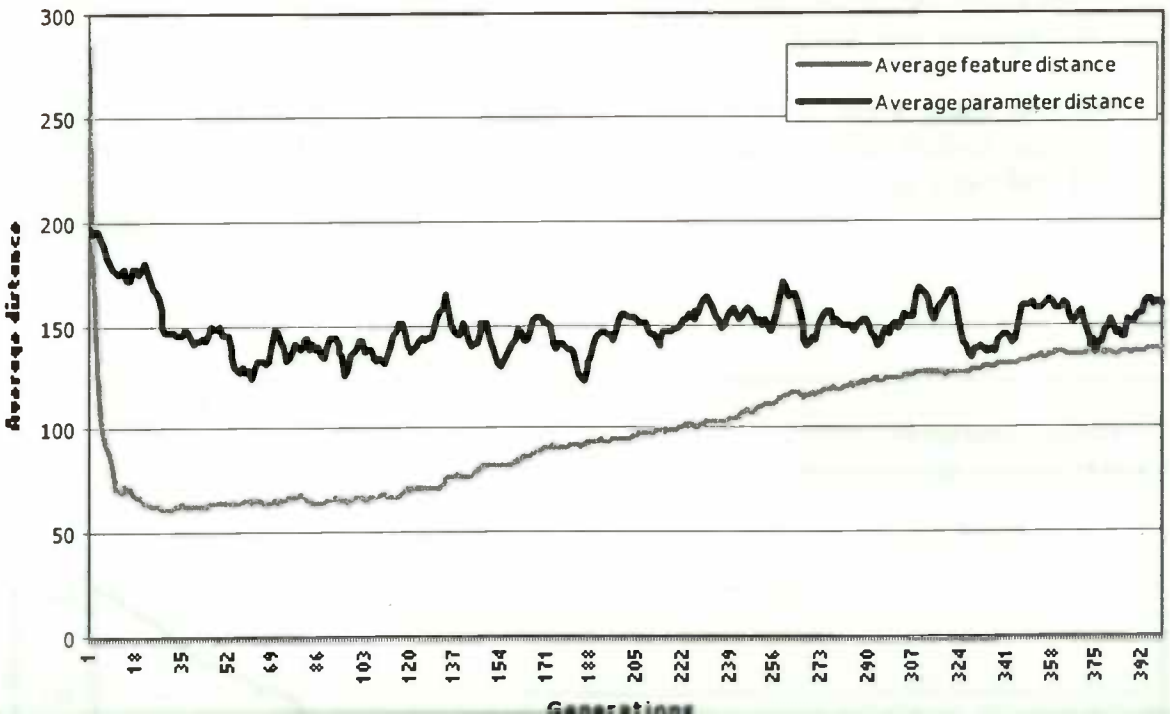
We tried to solve this problem in two ways. Firstly we used a selection mechanism based on fitness ranking. This means that the chance of being selected into the next generation is not proportional to the fitness but proportional to the fitness rank. This

effectively compresses the range of fitnesses and prevents the genes of one super-fit individual from spreading excessively over the population. A second solution to the problem was implemented by using steady-state replacement of individuals: most individuals are copied into the next generation directly, and only a few are replaced by new individuals. Beasley, Bull et al. have argued that this may be a better model of what happens in nature, where offspring and parents are alive concurrently (Beasley, Bull et al. 1993).

Another effect of these two measures is that the selection pressure is lowered, which means that individuals that do not make very accurate predictions in the beginning are given the benefit of the doubt. This is desirable in our case, because we do not want to discard any individuals based on one bad prediction. Only after several bad predictions an individual should be discarded.

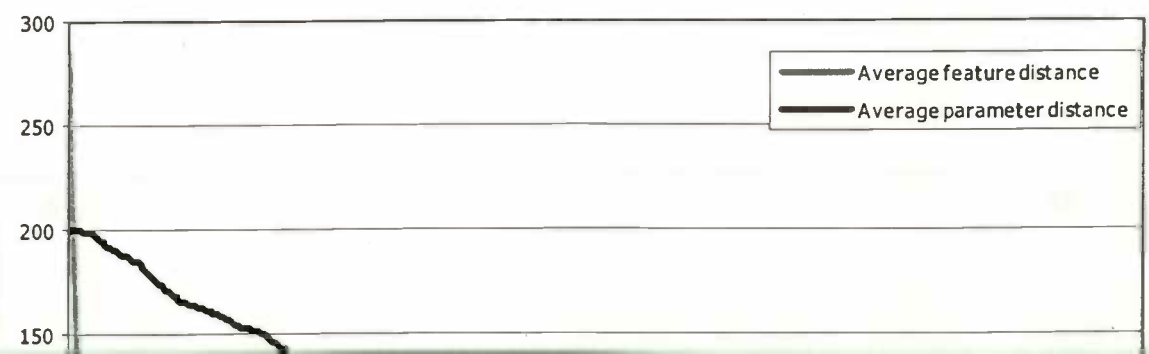
The effect of increasing the percentage of the population that remains unchanged can be found in the table below. It can be seen that a high steady state percentage leads to a slower convergence. Note that it is not necessary to prevent convergence altogether: if there is one student model that consistently makes correct predictions this is not a problem.

Steady state percentage:	**Number of generations before diversity falls below 10:**
20	± 6
90	± 30
	± 57


Average feature distance
Average parameter distance
Average distance
Generations
300
250
200
150
100
50
0
1
18
35
52
69
86
103
120
137
154
171
188
205
222
239
256
273
290
307
324
341
358
375
392

higher chance of being selected for the next generation. It turns out that these student models have, on average, a better way of updating their values, which is why the parameter distance becomes smaller too. This observation corresponds with what we were hoping to find, although a generation of reasonable student models are found only after many problems are solved by the student. Attempts to find parameter settings that get the same results quicker are described in the next subsection.

In both of the graphs mentioned above there is a steep drop in the average feature distance in the beginning. This is related to the fact that the values of the student models are set after the first observed answers of the artificial student, as was described in process 2 of the previous chapter.



settings is changed. Several times it seemed like the algorithm was behaving erratically because of a bug in the code. After closer inspection it nearly always turned out that the algorithm was merely behaving in an unforeseen way. The search process for the settings for the algorithm that produced the expected results was therefore based on several hypotheses that were tested with mixed results. In this section I shall give one of these hypotheses that was tested to illustrate the research process.

Since the average parameter distance in the graph in Figure 7-2 goes towards zero quite slowly, the mutation parameters of the algorithm were varied somewhat to investigate whether it was possible to find a quicker way of finding student models whose update parameters are closer to the 'true' student model. Since the mutation parameters influence how much the algorithm explores the search space in relation to how much the algorithm tends to stick with the current solutions, it is interesting to look at different settings of these parameters to see if it is possible to accelerate the process of finding good student models.

There are two different parameters that can be set to influence the way in which mutations take place. First of all, the parameter called mutationLikelihood indicates the likelihood that an update parameter is mutated. The parameter called mutateMaxCreep indicates what the maximum value is with which an update parameter can be incremented (note that an increment can be either positive or negative). The term creep refers to the fact that the mutation value is set incrementally rather than entirely at random. In short, for each update parameter there is a mutationLikelihood chance that it is incremented by a random number in the interval of [-mutateMaxCreep, mutateMaxCreep].

Several combinations of these two parameters were tried. In Figure 7-2 the mutationLikelihood was set at 30 and the mutateMaxCreep was set at 40. In Figure 7-3 both parameters were set at 10. It can be seen that the parameter distance in this case did not tend to get smaller more quickly compared to the previous settings, possibly because there was not enough exploration of the search space in this case.

The original values for these parameters proved to be better than the parameters tried later on. It is, however, not unthinkable that with the later parameter settings in combination with for example a lower steady state percentage (because this also influences how eager the algorithm is to explore new search space) good results might have been obtained. This is a good illustration of the task at hand: several hypotheses to improve the performance are tested and only occasionally a right set of parameters is found. In the absence of a more sophisticated way of automating the search for good parameter settings, luck is unfortunately a factor that has to be dealt with.

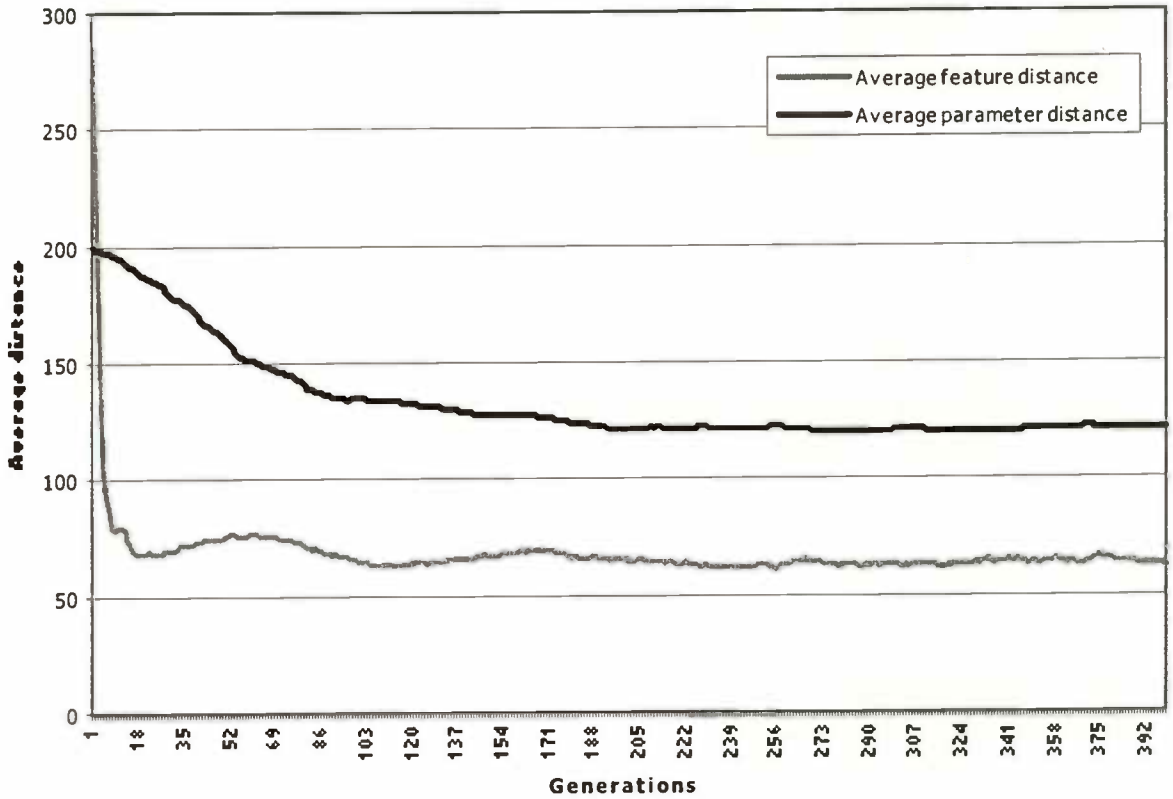


Figure 7-3. The average of 25 runs with mutateMaxCreep set at 10 and the mutationLikelihood set at 10.

Test 1.3: Trying out different update functions

Originally the student models could only get a positive fitness update. A problem that arose using this strategy, was that student models that made good predictions in the beginning, but made bad predictions in the last few generations, were still fitter than student models that made bad predictions in the beginning but made good predictions in the last few generations. Clearly this was undesirable. Several methods were tried to alleviate this problem:

1. The scores for the student models were scaled to the interval of [-1,1], where the worst scoring student model got a –1 fitness update, and the best scoring student model got a +1 update. The other student models got a value in between that is proportional to their score. This fitness update function was used in the results that have been presented so far.
2. Another strategy for fitness updating was tried in which the fitness of an individual purely depended on the score of the current feature, and the scores in the past were discarded. Since the student models with a higher fitness have a higher chance of being copied into the next generation, the nature of the algorithm itself ensured that there was still a memory effect. This fitness update function has been tested with little success.

3. The third fitness updating function that was tried combines the properties of the two mentioned above. In this function a parameter λ was introduced to indicate how much of the current fitness value is based on the predictions in the past and how much is based on the current prediction. The fitnesss are updated according to the following formula:
   NewFitness = λ * currentPrediction + (1- λ) * previousFitness.
   In this formula currentPrediction indicates how well the student's answer was predicted. It is scaled to the interval of [0, 1]. The parameter λ can be set to a value between 0 and 1, where a value closer to 0 indicates that it is very inert to the current prediction and a value closer to 1 indicates that it is very reactive.

In Figure 7-4 it can be seen that the third fitness updating function did not perform as well as the first function that was tried. For this function λ was set to a value of .2, meaning that the algorithm was set to behave quite conservatively. This was the best result that was obtained with this update function.

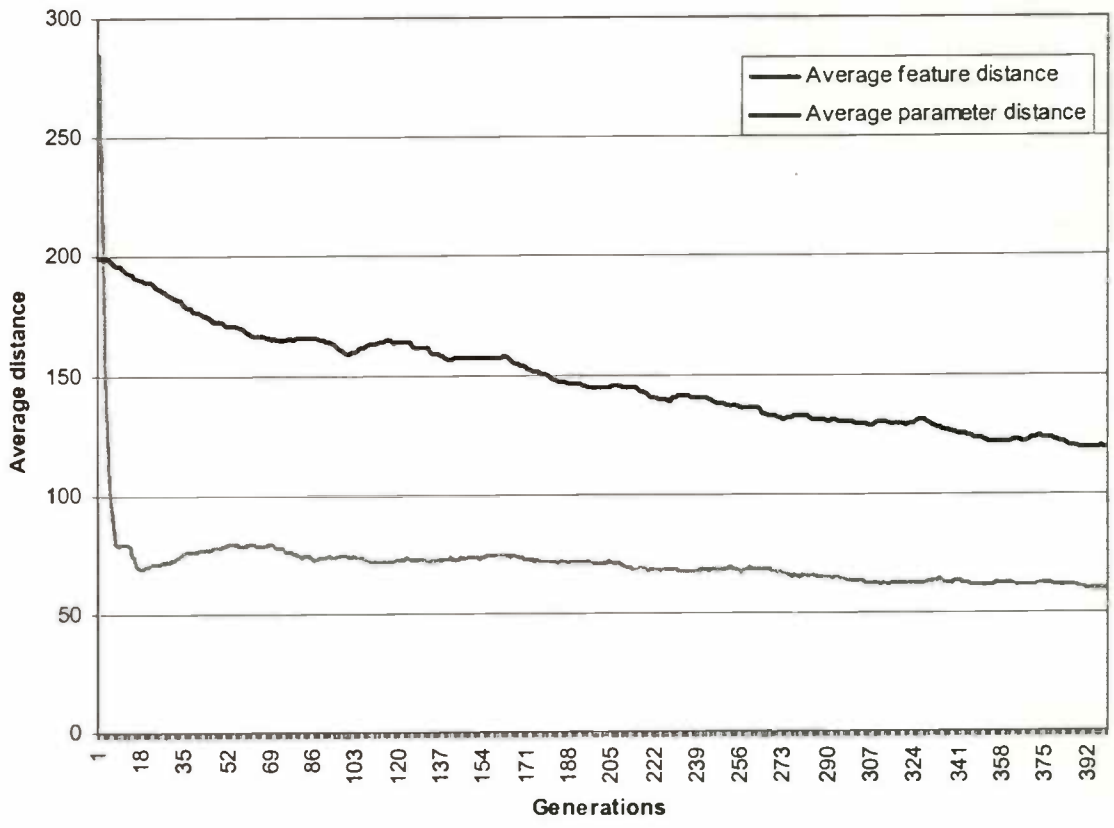


Figure 7-4. The performance of the algorithm with fitness update function alternative 3 (see above text). The parameter λ was set to .2.

7.4 Test case 2: Modelling a dynamic artificial student

After the first round of tests a set of parameters was found that indicated it could be possible to get an idea of how the student knowledge is updated on the basis of observations of the answers that the student gives. However, for the first round of tests a static student model was used. What we had investigated so far, is whether it was possible for the algorithm to discover that the student knowledge did not change at all (i.e. the update parameters were set to zero). This was the simplest case to model for the algorithm. Although it had no practical implications, it was necessary to investigate the behaviour of the algorithm in this simple case before moving on to a more realistic scenario where the student knowledge actually changes over time. The performance of the algorithm in this simple case was more or less as expected (provided the correct genetic algorithm parameters were set), although the algorithm was very slow to find update parameters that were reasonably good.

The next thing that we were interested in knowing is whether it was also possible to find reasonable update parameters when the student model changes over time. This case was one step closer to a realistic situation. Obviously, this case was more difficult to model, since the criteria about what is a good model of the student knowledge change over time.

Several tests runs were conducted for this test case. The correct genetic algorithm parameters were sought in a similar fashion as in the first test case. This time no acceptable parameter settings could be found (the best results that were obtained can be seen in Figure 7-5. The average over 25 runs of the algorithm in the test case of modelling a dynamic artificial student. It can be seen that the average feature distance tends to increase over time.. All the settings that were tried resulted in random fluctuations of the average parameter distance, meaning that the algorithm could not track how the knowledge of the artificial student changed over time.

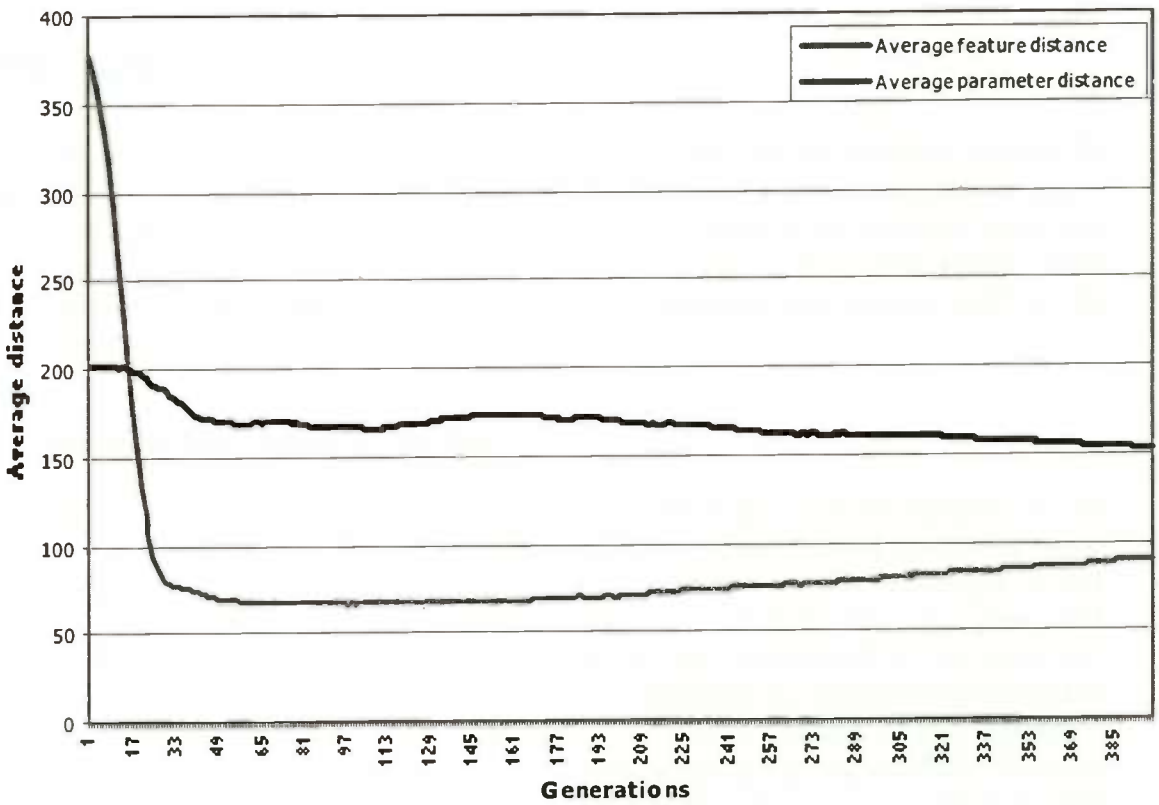


Figure 7-5. The average over 25 runs of the algorithm in the test case of modelling a dynamic artificial student. It can be seen that the average feature distance tends to increase over time.

7.5 Test case 3: A more realistic setting

We decided against conducting the third series of tests. After the unsuccessful results of the second series of tests, it was not realistic to expect that a more complex testing environment would show the expected results.

8 Conclusions

In this chapter I shall discuss the results that were described in the previous chapter. In the next subsection a general interpretation of the results and some suggestions about what the difficulties were are given. After that I shall refer back to the research questions and discuss to what extent they have been be answered. I shall then offer some suggestions for further research, and finally I shall re-examine the original motivations that inspired this research to see what the difficulties were.

8.1 Practical evaluation of the results

Before evaluating the results that we have obtained, I shall give a short summary of the test setups that we have used. We have designed a technique for student modelling using a genetic algorithm (described in section 5.3). The main idea of this technique is that ambiguous interpretations of observations about the student can be processed simultaneously. By discarding interpretations that do not correspond to the observed behaviour of the student later on, a correct way of interpreting the observations about the student can be found. This idea was tested using a simplified version of this algorithm (described in section 6.3). We tested this simplified algorithm using artificial students that model how the knowledge of a hypothetical student changes over time. We evaluated how well the algorithm could track these changes. We designed three test cases to evaluate the algorithm. For every test case the complexity (and therefore the realism) of the setup was increased.

In the first series of tests we researched how the algorithm behaved if an artificial student with static knowledge was used. What we saw was that if the parameters of the genetic algorithm were set correctly, a clear trend could be seen: the average parameter distance became smaller over time. This indicated that the way that most student models were updated was similar to how the 'true' student model was updated. This was the result that we were looking for. However, there were two observations with these results that were problematic.

Firstly, the algorithm did not seem to be very robust. The tolerance for different genetic algorithm parameters was not as high as we were hoping for. Acceptable results were only found after a laborious search for the correct parameters. The moment that one of the parameters was changed the behaviour of the algorithm deteriorated quite radically. Another hint at the lack of robustness was that for single runs, the results were sometimes quite different from run to run, solely due to the fact that different random seeds were used. This is not directly apparent from the presented graphs because they represent the average of several runs of the algorithm. Clearly this inconsistent behaviour is not what we were hoping for. Apart from the effort that it takes to set the parameters, it is not possible to finetune these parameters in practice, because then we do not have such a direct access to the knowledge of the students as we have with our artificial students.

Secondly, the algorithm tended to take a long time before a reasonable set of student models was found. For example, in Figure 7-2 the average parameter distance did not fall

below 100 until after the student had solved 165 problems. Considering the fact that the initial (random) distance was 200, this was not a very good result. Since only three different problem types were possible, the algorithm could only find moderately accurate update parameters after seeing the student solve an average 55 problems per problem type. Obviously one would want to be able to model the student much quicker than that. In a more realistic situation it may be possible that the student is offered perhaps 15 different types of problems. By the time that an acceptable set of student models is found in such a case, the student has solved so many problems that he hardly needs to practice anymore. Furthermore, the framework is intended to contain many different update rules corresponding with the different ways of interpreting the observations of the student's answers. It is to be expected that the algorithm will take even longer to find a reasonable set of student models in such a more complex case, if an acceptable set can be found at all.

In the second series of tests we tested with an artificial student whose knowledge changed over time. As can be deduced from the previous paragraphs, the expectations for this test case were not very high. Indeed, no set of parameters was found under which the algorithm performed satisfactorily. In the light of the lack of robustness in the simplest test case this was to be expected. We have decided to forgo the third round of tests altogether, since after the outcome of the second round of test this was not a hypothesis worth testing.

To summarize, the first round of tests suggested that the basic idea of the algorithm did make some sense, because some of the results corresponded with our expectations. For a slightly more realistic case the algorithm did not prove to be efficient and robust enough. Considering the fact that the algorithm that was implemented was already a simplification of a real life version, we can conclude that the algorithm proposed in this thesis is not suitable for student modelling in a practical application.

8.2 Explanations for the results

We have concluded that the algorithm in its current form did not perform the way we expected. I shall give some possible explanations for why this is the case.

The most important reason for the practical failing of the algorithm was the lack of robustness. Considering that normally in a genetic algorithm the parameter settings are not that critical (Busetti 2000), this came as quite a surprise. If the algorithm is to work in a more practical case, it is not possible to finetune all the genetic algorithm parameters to fit a particular (artificial) student as we have done in our test case. In a practical application we do not have access to the 'true' student model, and therefore no direct comparison can be made between the 'true' student model and the hypothetical student models (as we have done in our test cases).

It is difficult to say why exactly the algorithm was not robust. As was mentioned a lot of the research related to genetic algorithms is empirical by nature (Beasley, Bull et al. 1993) and it is not always possible to explain exactly why certain phenomena occur. Whitely (Whitley 1994) states that 'What may seem like simple changes in the algorithm, often result in surprising kinds of emergent behaviour'. As was stressed before, this results in a trial and error process in which many hypotheses have to be tested before some new insights are gained. Under these circumstances it is hard to make any

conclusive statements about the algorithm, but I shall give a number of explanations for the poor performance that were quite clear.

A big difference with most other genetic algorithms is that no crossover operator was used. This has been described in literature before (Beasley, Bull et al. 1993; Whitley 1994), and Schaffer *et al* claim that this *naive evolution* (i.e. based only on selection and mutation) can perform a powerful hill-climbing search. Furthermore, examples in nature show that asexual reproduction can evolve sophisticated creatures without crossover. We opted for the use of naive evolution because of the relation between the update parameters (the genotype) and the student model features (the phenotype). Normally a direct relation between the genotype and the phenotype exists: the phenotype is calculated directly from the genotype. The fitness function then selects the fittest individuals on the basis of properties of the phenotype. In such a case it makes sense to combine the genes of two individuals to try if this results in a fitter individual. In our case, however, phenotype could not be calculated directly from the genotype, since the update parameters kept influencing the student model features after each observation of the student. In other words, the way that the phenotype looked was not only influenced by the current genotype, but also by the way the genotype and the phenotype looked in the past. This relation would have been destroyed if we had randomly combined two genotypes.

The problem that we encountered once we came to the insight that the genotype – phenotype relation is not as direct as in traditional genetic algorithms, was how to explore the search space for the correct update parameters. The drawback of a mutation mechanism was the same as of a crossover mechanism in this case. By randomly varying the update parameters, the relation between the update parameters and the student model features is again lost. Since an exploration of the search space was of course necessary, we compromised by choosing an incremental mutation method, the so-called creep method. In this method the update parameters were *incremented* by a random number, so that the genotype – phenotype relation is somewhat disturbed rather than completely lost. This compromise unfortunately turned out to be sufficient for the simplest test case only. Normally the crossover mechanism is responsible for combining two good solutions so that a new and promising part of the search space is explored. Without a crossover mechanism the algorithm is essentially reduced to a search around known points in the search space. Obviously this search method does not exploit the full potential of genetic algorithms.

Another factor that probably reduced the effectiveness of the algorithm is the high interaction between the different genes (i.e. the update parameters). The term *epistasis* refers to the fact that the influence of a gene on the fitness of an individual depends on what gene values are present elsewhere. If a gene suppresses the effect of another gene it is said to be epistatic. All the representations of problems that are solved using genetic algorithms have some epistasis, since otherwise the problem would be easier to solve using alternative techniques. If the epistasis is very high, however, a genetic algorithm is not effective (Beasley, Bull et al. 1993). The way that the update rules and the corresponding parameters are defined ensures that the epistasis in our algorithm is intrinsically high: the update rules all increment or decrement a particular feature of the student model. This means that an increase caused by one update rule can be fully compensated by a decrease caused by another update rule. This effect will be even more

noticeable if more update rules are defined, because this will lead to even more possible ways in which update rules can compensate each other's effect.

In Figure 7-2 we can see that the average feature distance never reached zero. An explanation for this has to do with the representation of the student knowledge that we chose to use. The likelihood that an artificial student made a mistake was expressed as a percentage in both the 'true' student model and the hypothetical student models, but in the (predicted) answer that an artificial student gave this likelihood was converted to a Boolean value to indicate whether a mistake was made or not. A hypothetical student model could therefore hold the same value for the likelihood that the student would make a mistake as the 'true' student model, but still (by chance) bet on the wrong horse. A related reason for imperfect predictions is the fact that some noise was added to the answer of the artificial student to simulate human variance in the answers that are given (no human student would always solve a problem in exactly the same time). These two observations resulted in the fact that the 'perfect' hypothetical student model (namely the one identical to the 'true' student model) did not always get the highest fitness update. This is just a result of the nature of the domain; a human tutor would not be able to make the correct prediction in all cases either.

8.3 Answers to the research questions

In section 5.5 a number of research questions was posed to guide the research. I shall now answer these questions:

1. Is it possible to generate an accurate student model using the proposed technique based on a genetic algorithm?
   As was discussed in the previous three subsections, we have not been able to obtain satisfying test results. Considering the theoretical shortcomings that were discussed it is not likely that the proposed technique can lead to better results, and therefore we must conclude that it is unlikely that it is possible to generate an accurate student model using this technique.
2. To what extent can this approach be generalized to other teaching domains?
   The algorithm was intended to be very flexible by allowing the designer to specify update rules to his best knowledge. We tried to make a strict distinction between the knowledge-based (domain dependent) part of the algorithm and the generic framework. Implementing the algorithm in a different domain would be a case of changing the update rules. However, in the light of the answer to the first question this question is not very relevant anymore.
3. What are the strengths and weaknesses of this approach, and how do these compare to the strengths and weaknesses of other approaches?
   Again, in the light of the answer to the first question this question is not very relevant anymore.
4. Are there ways in which this approach can complement other approaches?
   Several student modelling techniques could benefit from using a genetic algorithm to set the parameters for these techniques. Some suggestions are made in the next section.

8.4 Further research

Although the algorithm that we proposed did perform as we expected, there are some issues that would be interesting to explore further.

First of all, the biggest challenge would be to find a representation in which the problem of not being able to perform crossover would be solved. In the representation that we used, it was not possible to use crossover because of the 'historic' relation between the update parameters (the genotype) and the student model features (the phenotype). If we can find a representation that takes the 'historic' changes into account (by keeping track of all the changes in the past), it may be possible to use crossover after all, and in this way make the algorithm more robust. A practical problem with this approach is probably the time it takes to calculate the values for the child from the values of the parents. The program already took quite long to execute (approximately one minute) if we wanted to use the average statistics of several runs. Retracing all the update calculations that were made in the past would greatly augment the computational complexity of the program, since many crossover operations are necessary for every new generation of student models.

A second issue that could be explored further is the combining of genetic algorithms with other approaches. This approach is already quite common in setting the parameters for neural networks (which are also used for student modelling). It may also be useful to apply a genetic algorithm technique to ACT-R based programs. In ACT-R it is necessary to set some parameters experimentally, which is a task that could be automated by using a genetic algorithm. Also it would be interesting to look at how this genetic algorithm approach could be combined with Bayesian networks to incorporate different possible interpretations of observations by letting each interpretation result in a different Bayesian network. Again a factor that needs to be taken into account is the computational complexity: if many calculations are necessary for every new generation, the execution of the algorithm can quickly become prohibitively slow.

8.5 Comments about the original motivations

Considering the fact that the empirical findings of this research project mostly did not correspond to our expectations, it is wise to re-examine the original motivations that inspired this research project.

The main source of inspiration for this proposed technique for student modelling has been the flexibility and seemingly effortless ability to adapt to the student that we see with human tutors. When a human tutor first starts interacting with a student, he has certain expectations about the student's abilities based on background information about the student that is available to him. The tutor then starts exploring what topics are difficult for the student by iteratively observing the student's actions and behaviour, adjusting his beliefs about the student's knowledge and customizing the material accordingly. However, the student's actions cannot always be interpreted unambiguously: if a student makes a particular mistake, there can be many reasons for it. For example, a student may not have studied the topic properly, the parent concept needed for the problem may not be clear, the student may be confused with a different topic, it may just have been a slip of concentration, etc. Also some students may learn quicker and in a

different way than other students. Different interpretations of the observations lead to different beliefs about the student and yet a human tutor does not seem to have a lot of difficulties testing different hypotheses about the student's capabilities. It is this flexibility with interpreting the student's actions that we wanted to model. To mimic this kind of reasoning of the tutor we designed the genetic algorithm in which multiple student models are retained in parallel and in which the best model is the one that is in correspondence with most observations.

After we finished the basic design of the algorithm we were confident that this was an interesting path to explore. The original intention was to finish the complete implementation of the ITS, so that we could test our algorithm in a real life situation. Unfortunately this turned out to be too time consuming and at a later stage we decided to test the basic assumptions that we made while designing the algorithm. We had to make many simplifications to be able to finish the algorithm in a reasonable time and to be able to analyse the data in a meaningful way.

We discovered that modelling the behaviour of a human tutor that we just described was not as straightforward as we expected it to be. The update rules were difficult to design in a naturally plausible way. Because the student models that we designed were a lot simpler than the mental models that a human tutor may have of a student, the update rules tended to have a little bit of an artificial feel. If a student is constantly distracted, for example, a human tutor will automatically incorporate this observation in his mental model of the student's knowledge, and ignore the fact that the student takes a long time to solve a problem. For a computer such an observation is impossible to make, and even if it could make this observation an adaptation of the student model and the pedagogical module is necessary before this extra information can be processed. Although the simplicity of the update rules is not necessarily a problem because we merely tried to test the framework, there are not many domains where a more natural set of update rules can easily be implemented. The human reasoning process that we tried to model is quite fuzzy and not so easily translated to a reasoning process that can be implemented on a computer.

Another factor that is very difficult to model is the knowledge that a human tutor has acquired from experience. For example, a human tutor has a lot of implicit knowledge of how students learn in general and of what problems students have encountered in the past. A human tutor will gradually learn from experience. If a student makes a particular mistake, the tutor might be reminded of a previous student that made a similar mistake and recall what the problem was in that case. For a computer this kind of matching is very difficult to implement, not in the least because of the representational flexibility that is necessary to match two cases.

We can conclude with the fact that although the outcomes of the research were not as we were expecting, the only way of finding out about the difficulties and challenges that were mentioned above, is to actually implement the proposed system and see what the bottlenecks are. The technique proposed in this thesis has to our knowledge not been described in literature before and we did not have a very clear idea as to where the research would lead. Implementing the basic algorithm gave us valuable insights that would not have been possible to obtain otherwise.

9 Bibliography

Aïmeur, E., G. Brassard, et al. (2002). CLARISSE: A machine learning tool to initialize student models. In: G. Goos, J. Hartmanis and J. van Leeuwen (Eds.). *Intelligent Tutoring Systems - 6th International Conference*. Berlin, Springer: 718-728.

Ainsworth, S. and S. Grimmshaw (2002). Are ITSs created with the REDEEM authoring tool more effective than "dumb" courseware? In: G. Goos, J. Hartmanis and J. van Leeuwen (Eds.). *Intelligent Tutoring Systems - 6th International Conference*. Berlin, Springer: 883-892.

Anderson, J., A. T. Corbett, et al. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences* 4(2): 167-207.

Anderson, J. and C. Lebiere (1998). *The Atomic Components of Thought*. Mahwah, NJ: Lawrence Erlbaum Associates.

Beasley, D., D. R. Bull, et al. (1993). An overview of genetic algorithms: Part 1, fundamentals. *University Computing* 15(2): 58-69.

Beasley, D., D. R. Bull, et al. (1993). An overview of genetic algorithms: Part 2, research topics. *University Computing* 15(4): 170-181.

Beck, J. E., M. Stern, et al. (1996). Applications of AI in education, *ACM Crossroads Student Magazine*. Available from: http://www.acm.org/crossroads/xrds3-1/aied.html.

Beck, J. E. and B. P. Woolf (1998). Using a learning agent with a student model. In: B. P. Goettl, H. M. Halff, C. L. Redfield and V. J. Shute (Eds.). *Intelligent Tutoring Systems : 4th International Conference, ITS '98*. Berlin, Springer: 6-15.

Bloom, B. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher* 13: 4-15.

Busetti, F. (2000). Genetic algorithms overview. Available from: http://www.citeseer.nj.nec.com/464346.html.

Chiu, B. and B. Webb (1997). Using C4.5 as an induction engine for agent modelling: An experiment of optimisation. *Sixth User Modeling Conference, Workshop on Machine Learning for User Modeling*. Available from: http://www.citeseer.nj.nec.com/87139.html.

Cooper, M. (1988). Interfaces that adapt to the user. In: J. Self (Ed.). *Artificial Intelligence and Human Learning*. London, Chapmann and Hall: 300-309.

Devedzic, V. and L. Jerinic (1997). Knowledge representation for intelligent tutoring systems: The GET-BITS Model. In: B. Du Boulay and R. Mizoguchi (Eds.). *Artificial Intelligence in Education*. Amsterdam, IOS Press: 63-70.

Devedzic, V., D. Radovic, et al. (1998). On the notion of components for intelligent tutoring systems. In: B. P. Goettl, H. M. Halff, C. L. Redfield and V. J. Shute (Eds.). *Intelligent Tutoring Systems : 4th International Conference, ITS '98*. Berlin, Springer: 735-744.

Fung, P. and T. O'Shea (1993). Fear of formal reasoning. In: P. Brna, S. Ohlsson and H. Pain-Lewins (Eds.). *ArtificialIntelligence in Education 1993 : Proceedings of AI-ED 93*. Charlottesville, AACE: 201-208.

Goodman, B., A. Soller, et al. (1998). Encouraging student reflection and articulation using a learning companion. *International Journal of Artificial Intelligence in Education* 9(1998): 237-255.

Heffernan, N. T. and K. R. Koedinger (2002). An intelligent tutoring system incorporating a model of an experienced human tutor. In: G. Goos, J. Hartmanis and J. van Leeuwen (Eds.). *Intelligent Tutoring Systems 6th International Conference, ITS 2002*. Berlin, Springer: 596-609.

Hoban, C. F. (1946). *Movies That Teach*. New York Dryden Press.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*, University of Michigan Press.

Jameson, A. (1996). Numerical uncertainty management in user and student modeling: an overview of systems and issues. *User Modeling and User-Adapted Interaction* 5(3-4): 193-251.

Jordan, D. W. and P. Smith (1994). *Mathematical Techniques: an Introduction for the Engineering, Physical and Mathematical Sciences*. Great Britain, Butler & Tanner Ltd.

Kay, J. (2000). Stereotypes, student models and scrutability. In: B. P. Goettl, H. M. Halff, C. L. Redfield and V. J. Shute (Eds.). *Intelligent Tutoring Systems 5th International Conference, ITS 2000*. Berlin, Springer: 19-30.

Kemeny, J. G. (1972). *Man and the Computer*. New York, Schribner's.

Kimball, R. (1982). A self-improving tutor for symbolic integration. In: D. Sleeman and J. S. Brown. New York (Eds.). *Intelligent Tutoring Systems*. Academic Press Inc.: 287-307.

Koedinger, K. R., J. Anderson, et al. (1997). Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education* 8: 30-43.

Kulik, J. and C. Kulik (1988). Timing of feedback and verbal learning. *Review of Educational Research* 58: 79-97.

Lesgold, A. (1987). Education applications. In: *Encyclopedia of Artificial Intelligence.* New York, Wiley: 267-272.

Lesta, L. and K. Yacef (2002). An intelligent teaching assistant system for logic. In: G. Goos, J. Hartmanis and J. van Leeuwen (Eds.). *Intelligent Tutoring Systems 6th International Conference, ITS 2002.* Berlin, Springer: 421-431.

Lewis, M. W., R. Milson, et al. (1987). The teacher's apprentice: Desiging authoring systems for high school mathematics. In: G. P. Kearsly (Ed.). *Artificial Intelligence and Instruction - Applications and Methods.* Boston, Addison-Wesley: 269-301.

Mark, M. A. and J. E. Greer (1991). The VCR tutor: Evaluating instructional effectiveness. *13th Annual Conference of the Cognitive Science Society.* Hillsdale, NJ, Lawrence Erlbaum Associates: 564-569.

Martin, J. D. and K. VanLehn (1993). Olae: Progress towards a multi-activity, Bayesian student modeler. In: P. Brna, S. Ohlsson and H. Pain-Lewins (Eds.). *Artificial Intelligence in Education 1993 : Proceedings of AI-ED 93.* Charlottesville, AACE: 410-417.

Mayo, M. and A. Mitrovic (2000). Using a probabilistic student model to control problem difficulty. In: B. P. Goettl, H. M. Halff, C. L. Redfield and V. J. Shute (Eds.). *Intelligent Tutoring Systems 5th International Conference, ITS 2000.* Springer, Verlag: 524-533.

Millan, E., J. L. Pérez-de-la-Cruz, et al. (2000). Adaptive bayesian networks for multilevel student modelling. In: G. Gauthier and C. Frasson (Eds.). *Intelligent Tutoring Systems 5th international conference, ITS 2000.* Berlin, Springer: 534-543.

Mitrovic, A. (2000). SINT - a symbolic integration tutor. In: G. Gauthier and C. Frasson (Eds.). *Intelligent Tutoring Systems. 5th International Conference, ITS 2000.* Berlin, Springer: 587-595.

Murray, W. R. (1998). A practical approach to Bayesian student modeling. In: B. P. Goettl, H. M. Halff, C. L. Redfield and V. J. Shute (Eds.). *Intelligent tutoring systems : 4th International Conference, ITS '98.* Berlin, Springer: 424-433.

Nicaud, J.-F., D. Bouhineau, et al. (2002). The Aplusix-Editor: A new kind of software for the learning of algebra. In: G. Goos, J. Hartmanis and J. van Leeuwen (Eds.). *Intelligent Tutoring Systems 6th International Conference, ITS 2002.* Berlin, Springer: 178-187.

Nicaud, J.-F., E. Delozanne, et al. (2002). Editorial. *Sciences et Techniques Éducatives* 9(1-2): 7-11.

Paraskakis, I. (2002). TeLoDe: Towards creating an intelligent computer algebra system. In: G. Goos, J. Hartmanis and J. van Leeuwen (Eds.). *Intelligent Tutoring Systems 6th International Conference, ITS 2002*. Berlin, Springer: 410-420.

Pohl, W. (1996). Learning about the user - user modeling and machine learning. In: V. Moustakis and J. Hermann (Eds.). *Proc. ICNL '96 Workshop - Machine learning meets Human-Computer Interaction*. 29-40.

Reye (1996). A belief net backbone for student modelling. In: A. M. Lesgold, C. Frasson and G. Gauthier (Eds.). *Intelligent Tutoring Systems: Third International Conference ITS '96*. Berlin, Springer: 596-595.

Russell, S. J. and P. Norvig (1995). *Artificial Intelligence : a Modern Approach*. Englewood Cliffs, N.J., Prentice Hall.

Self, J. (1990). Bypassing the intractable problem of student modelling. In: C. Frasson and G. Gauthier (Eds.). *Intelligent Tutoring Systems: at the Crossroads of Artificial Intelligence and Education,*. Norwood, N.J., Ablex: 107-123.

Shapiro, S. C. and D. Eckroth (1987). *Encyclopedia of Artificial Intelligence*. New York, Wiley.

Shute, V., R. Glaser, et al. (1989). Inference and discovery in an exploratory laboratory. In: P. Ackerman and R. Glaser (Eds.). *Learning and Individual Differences*. San Francisco, Freeman: 279-326.

Sleeman, D. and J. S. Brown (1982). *Intelligent Tutoring Systems*. Orlando, Academic Press Inc.

Vigotsky, L. S. (1978). *The Development of Higher Psychological Processes*. Cambridge, MA, Harvard University Press.

Virvou, M. and M. Moundridou (2000). Modelling the instructor in a web-based authoring tool for algebra-related ITSs. In: G. Gauthier and C. Frasson (Eds.). *Intelligent Tutoring Systems 5th International Conference, ITS 2000*. Berlin, Springer: 635-644.

Webb, G. I. and M. Kuzmycz (1998). Evaluation of data ageing: A technique for discounting old data during student modeling. In: B.P. Goettl, H.M. Halff, C.L. Redfield and V.J. Shute (Eds.). *Proceedings of the Fourth International Conference on Intelligent Tutoring Systems*. Berlin, Springer: 384-393.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems : Computational and Cognitive Approaches to the Communication of Knowledge*. Los Altos, Calif., Morgan Kaufmann Publishers.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing* 4: 65-85.

Woolf, B. P. (1988). Representing complex knowledge in an intelligent machine tutor. In: J. Self (Ed.). *Artificial Intelligence and Human Learning*. New York, Chapman and Hall Ltd.: 3-27.

Woolf, B. P. and P. A. Cunningham (1987). Multiple knowledge sources in intelligent teaching systems. *IEEE Expert* (2): 41-54.

Appendices

Appendix A: Glossary of terms introduced in this paper

'True' student model	A model of the knowledge of the artificial student. This knowledge is structured in a similar way as the hypothetical student models.
Actual student answer feature (Actual SAF)	A variable that describes a feature of an answer that the (artificial) student gives. See predicted student answer feature.
Artificial student	An artificial student is intended to mimic a human student and is used for testing the algorithm. An artificial student's changing knowledge will result in changing answers given by this artificial student.
Average feature distance	The average distance between the features of the 'true' student model and the features of the hypothetical student models.
Average parameter distance	The average distance between the update parameters of the 'true' student model and the update parameters of the hypothetical student models.
Genetic algorithm parameters	Parameters that influence the way the genetic algorithm behaves, for example the mutation likelihood or the steady state percentage.
Hypothetical student model	A hypothetical student model (also simply called a student model) is a model that intends to describe certain aspects of the knowledge and skills that a student has. In our test algorithm the knowledge is represented by values called Student Answer Features that try to predict how a student will solve a problem.
Predicted student answer feature (predicted SAF)	A prediction of a feature of an answer that a student is expected to give. This prediction is made on the basis of a student model. One answer has several features.
Problem type	Description of what topic is necessary to master in order to be able to solve a problem.

Student answer feature (SAF)	A variable that describes a particular (predicted) feature of the student's answer, for example how long a student takes to find a solution to a problem. See Actual student answer feature, predicted student answer feature.
Student model feature	Knowledge variable of a student model: indicates how well a student can solve a particular problem type.
Update parameter	An update parameter is used to quantify an update rule.
Update rule	An update rule specifies how an observation about a student's answer is translated into an update of a student model.

Appendix B: Glossary of terms in the field of Intelligent Tutoring Systems

Cognitive tutor	ITS that focuses on teaching procedural skills.
Communication model	Component of an ITS that decides how the information is presented to the student (by adjusting the user interface and interaction style).
Computer Assisted Instruction (CAI)	Computer-based tutoring system. This type of system does not keep track of the student's progress and therefore cannot adapt to the individual level to the student. This type of system is the predecessor of Intelligent Tutoring Systems.
Domain model	Component of an ITS that holds all the information about the domain that is taught.
Expert model	Component of an ITS that models the knowledge of a domain expert. Usually a runnable model that can answer questions in a similar fashion as a domain expert.
Intelligent Tutoring Systems (ITS)	Computer-based tutoring system that keeps track of the knowledge and skills of the student and adapts itself to the level of the student.
Knowledge-based tutoring system	ITS that focuses on teaching concepts.
Pedagogical module	Component of an ITS consisting of a data structure that models the teaching process and incorporates knowledge about teaching techniques and strategies. Decides on what problems, hints, questions, etc. to present to the student.
Student model	Component of an ITS consisting of a set of variables that describe certain aspects of the student's knowledge and skills.

Appendix C: Glossary of terms in the field of student modelling

ACT-R theory	Theory used to model cognitive processed. Applied to ITSs to implement a runnable expert model.
Bayes' rule	Rule used in probability theory to calculate unknown probabilities from known probabilities.
Bayesian network	Acyclic network for modelling domain knowledge that uses probability values to indicate how the nodes influence each other. Also called belief network.
Belief mass	Term used in the Dempster-Shafer theory of evidence to indicate the amount of belief in a proposition.
Belief network	See Bayesian network
Buggy student model	Student model in which the knowledge of the student is represented as a subset of the total domain knowledge and also models the incorrect knowledge that a student may have.
Causal inference	See predictive inference.
Decision tree learning	Technique to output a Boolean decision based on a number of attributes. Can be used to predict whether students will give a correct answer.
Declarative knowledge	Factual knowledge that we are aware of.
Dempster-Shafer theory of evidence	Theory used for student modelling that focuses on how much evidence has been seen to support propositions.
Diagnostic inference	Inference in Bayesian networks to update the probability values between nodes.
Fuzzy logic	Technique for reasoning with vaguely defined propositions.
Machine learning	Automatically deriving a model for data from large datasets without relying on heuristics or knowledge engineering.
Neural network	Interconnected network inspired by the human brain, used for automatically interpreting large datasets.
Overlay student model	Student model in which the knowledge of the student is represented as a subset of the total domain knowledge.

Predictive inference	Inference in a Bayesian network to predict the answer of the student based on the probability values between nodes in the network. Also called causal inference.
Procedural knowledge	Knowledge that we display in our behaviour that we are not aware of.
Production rule	Representation of procedural knowledge in ACT-R based systems.
Stereotype student model	Student model that is selected on the basis of a few criteria to quickly initialize the student model. For example, on the basis of whether the student has selected beginner or expert level, expectations about the student's knowledge are set.

Appendix D: Glossary of terms in the field of genetic algorithms

Child	The individual that is created from two other individuals.
Chromosome	The parametric representation of a solution, consisting of genes. See individual, solution.
Creep mutation	Mutation of a gene where a small randomly generated amount is added or subtracted.
Crossover	The process of combining two parent chromosomes to form a child chromosome.
Die out	The discarding of a solution that is very far from the desired solution.
Epistasis	Interaction between genes in a chromosome. Occurs when one gene influences the expression of an other gene in the chromosome.
Fitness	A calculated value to indicate how close a solution is to the desired answer.
Fitness function	A formula used to calculate the fitness of a chromosome on the basis of the genes.
Fitness ranking	Technique where the fitness rank (i.e. fittest individual gets highest rank, second fittest gets second rank, etc.) is considered when selecting individuals for reproduction, rather than the absolute fitness value. This decreases the selection pressure.
Fitness score	Fitness value that is calculated for an individual on the basis of the fitness function.
Gene	Basic building block of a solution: one parameter of a solution.
Generation	A population of individuals that is considered in one iteration of the algorithm.
Genetic drift	Phenomenon where one gene becomes predominant in a generation due to the process of iteratively selecting individuals.
Genotype	The set of parameters that specify a solution

Individual	Another word for solution, used in the context of reproduction. Individuals are often referred to as parents, children, offspring, etc. to refer to the relation it has to other individuals. See chromosome, solution.
Mating pool	Population of individuals that are selected for reproduction.
Mutation	The random changing of a gene.
Offspring	All the children of a generation, i.e. the new generation for the next iteration of the algorithm.
Parent	Individual that is recombined with another individual to form a child.
Phenotype	The appearance of a solution as is specified in the genotype.
Population	Set of individuals.
Premature convergence	Phenomenon where the algorithm tends to focus on one small part of the search space to quickly. Solutions in other parts of the search space may be overlooked. Opposite of slow finishing.
Recombine	Combining two individuals to form a child individual.
Reproduce	See recombine.
Search space	All the possible solutions. We are only interested in the best solutions.
Selection pressure	Likelihood that the less fit individuals die out. If the selection pressure is high, only the fitter few individuals will survive.
Slow finishing	Phenomenon where the algorithm tends to focus on too many solutions and therefore is not able to find a good solution. Opposite of premature convergence.
Solution	One possible way of problem parameter setting. This term does not refer to the desired answer, but to a hypothesis that is to be tested for closeness to the desired answer. See individual, chromosome.
Steady state	Form of genetic algorithm in which only a few individuals in every generation die out.
Super fit	An individual is said to be super fit if its fitness value is greater than that of either parent.

Weak search method	Search method in which relatively few assumptions about the domain are made. A genetic algorithm is such a search method.
Remainder stochastic mapping	Selection method in which the individuals are selected for reproduction on the basis of the relative fitness value.

Appendix E: Basics of mathematical differentiation

The following text is intended for revision purposes only: It does not claim try to explain the mathematical theory.

The formal definition of the derivative of the function $f(x)$ at the point x_0 is given by:

$$\frac{df}{dx}(x_0) = f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

This is an indication of the steepness of a function in a point.
To obtain a graph of the steepness in all the points in a graph, an algebraic method called differentiation can be used. This technique can be used to transform a formula into another formula that is called its derivative.

These are some of the simpler rules for differentiation that have been implemented in the demonstration version of the ITS that is included with this thesis (u and v stand for a formula dependent on x):

Rule name:	**F (x)**	**f '(x)**
Constant rule	u*c	c*u'
Factor rule	c*u	c*u'
Polynomial rule	x^n	n*x^(n-1)
E power rule	e^x	e^x
Sine rule	sin(x)	cos(x)
Cosine rule	cos(x)	-sin(x)
Natural logarithm rule	ln(x)	1/x
Chain rule	u(v)	u' (v) * v'

The main difficulty of finding the derivative of a formula is the correct order of application of the different rules.
A simple example of how to find the derivative of a formula:
Question:
Give the derivative of sin(x^2)
Answer:

(sin(x^2))'	= cos(x^2)*(x^2)'	(← Using the sine rule and the chain rule)
	= cos(x^2)*2x	(← Using the polynomial rule)

Source: (Jordan and Smith 1994)

Appendix F: Javadoc description of the testing program

In this subsection the Javadoc description is given of the program that was implemented to test the algorithm. Javadoc is an automatically generated summary that is composed of the code comments that describe the objects and their associated methods.

gastudentmodeltest

Class GAStudentModelTest

public class **GAStudentModelTest**

Title: Student modelling using a genetic algorithm

Description: Proof of concept. This program investigates whether it is possible to model a student using a (simplified) genetic algorithm.

Constructor Detail

GAStudentModelTest

public **GAStudentModelTest()**

Constructor. Initializes the genetic algorithm, the trueStudentModel and the first problem.

Method Detail

initializeStudentModel

private StudentModel **initializeStudentModel()**

Used to set the values of the 'true' student model.

Returns:

An initialized student model

---

doMultipleRuns

public void **doMultipleRuns()**

Method to test the genetic algorithm: the statistics of several runs are gathered and averaged. These statistics are written to the screen, ready to be copied into excel for visualizing.

The basic algorithm:

Initialization: Create the genetic algorithm object.

Repeat _numberOfGenerations times:

- Generate a new problem.
- Let the true student model generate a student answer feature.
- Update the true student model.
- Create the next generation on the basis of the generated student answer feature.

To make the graph of the data smoother and less influenced by the chance values, this basic algorithm is repeated a number of times and the statistics are averaged.

---

gastudentmodeltest.problem

Class Problem

public class **Problem**

Description: Simple problem generator

Constructor Detail

Problem

public **Problem**(int randomSeed)

Constructor.

Parameters:

randomSeed - An integer to initialize the private random field in order to generate predictable pseudorandom series of problems

---

Method Detail

generateNextProblem

public void **generateNextProblem**()

Generates a new problem type. In our model this consists of an integer between 0 and 2 to represent the sine rule, cosine rule or natural logarithm rule.

---

gastudentmodeltest.student

Class GeneticAlgorithm

public class **GeneticAlgorithm**

The genetic algorithm and additional functions.

Constructor Detail

GeneticAlgorithm

public **GeneticAlgorithm**(int mutationLikelihood,
int generationSize,
int randomSeed,
int penaltyForMistakeMismatch,
int noiseFactor,
int steadyStateNumber,
int mutateMaxCreep)

Constructor. Sets all the parameters, and initializes the first generation of student models with random parameter values and feature values of 0. These feature values are set at the moment the first corresponding student answer feature has been observed.

Parameters:

`mutationLikelihood` - The likelihood that an update parameter will be mutated.

`generationSize` - The number of student models in a generation

`randomSeed` - Integer used to initialize a private random object, in order to get predictable pseudo random numbers.

`penaltyForMistakeMismatch` - Used in the calculation of the distance between two student answer features: if a student model has incorrectly predicted whether a mistake was made, the value for the distance is increased by this penalty.

`noiseFactor` - Indicates how much noise will be added to the student answer feature created on the basis of the 'true' student model.

`steadyStateNumber` - Indicates how many individuals in a generation are copied into the next generation without being mutated. This influences how quickly the algorithm converges.

`mutateMaxCreep` - The creep mutation method is used: mutations are not random, but a random positive or negative value with a an average standard deviation of mutateMaxCreep is added to the (Jordan and Smith 1994)al value. In other words, the mutation is incremental.

Method Detail

createNextGeneration

public void **createNextGeneration**(StudentAnswerFeature actualSAF)

Creates the next generation. First the fitnesses are updated on the basis of actualSAF. Then the fittest _steadyStateNumber student models are copied directly in the next generation and updated. The rest of the student models that are copied in the next generation are selected on the basis of their rank (the higher the rank, the higher the chance of being selected) and mutated and updated. Finally the next generation is copied into the current generation and the currentgeneration is sorted.

Parameters:

`actualSAF` - The student answer feature of the 'true' student model.

---

performMutation

private void **performMutation**(StudentModel sm)

Every parameter in a student model has a _mutationLikelihood chance of being mutated. The creep mutation method is used: mutations are not random, but a random positive or negative value with a an average standard deviation of mutateMaxCreep is added to the original value.

Parameters:

`sm` - Student model that is mutated

---

selectParentByRank

private int **selectParentByRank**()

Selects a student model from the current generation. The higher the rank of a student model (based on the fitness value), the higher the chance that it is selected.

Returns:

An integer representing the rank of a parent.

---

updateFitnesses

public void **updateFitnesses**(StudentAnswerFeature actualSAF)

For each studentModel: Calculate the distance between the actualSAF and predictedSAF (i.e. the studentAnswerFeatures that are expected on the basis of the values of the hypothetical student model). Only calculate the distance for the active gene (based on the problem), otherwise a mistake in one gene may have a disproportunately big influence. The best scoring student model will receive a fitness update of +1, the worst will receive a fitness update of -1, the other student models will receive an update between -1 and +1.

Parameters:

`actualSAF` - The student answer feature generated by the 'true' student model.

---

getPopulationStatistics

public PopulationStatistics **getPopulationStatistics**(StudentModel trueSM)

To measure how well the algorithm is performing, it is necessary to calculate a number of values to give an insight into what the distances are between the features and parameters of the true student model and the hypothetical student model. avgFeatureDistance: The featuredistance between two student models is the sum of the differences of ALL the features of the model. The average feature distance is the average of all the distances (to the 'true' student model) in a generation. Note that the fitness is updated on the basis of the distance of ONE feature (i.e. the feature that is relevant to the current problem). fittestFeatureDistance: This is the feature distance between the fittest student model of a generation and the 'true' student model.

bestFeatureDist: This is the smallest feature distance that was measured in a generation. This is not necessarily the feature distance of the fittest.

avgParameterDistance: similar to avgFeatureDistance, but the distance measured is between the parameters. This gives an indication of how wel the genotype matches if the phenotype matches. fittestParameterDistance: similar to fittestFeatureDistance, but the distance measured is between the parameters.

averageFitness: The average fitness of the student models in a generation.

Parameters:

`trueSM` - The 'true' student model.

Returns:

Data structure containing avgFeatureDistance, fittestFeatureDistance, bestFeatureDist, avgParameterDistance,fittestParameterDistance, averageFitness

---

debugInsertTrueStudentModel

public void **debugInsertTrueStudentModel**(StudentModel trueStudentModel,
int index)

Inserts a copy of the 'true' student model into the generation of the genetic algorithm. This method is used to see whether a perfect student model will actually stay the fittest and stay unmodified. This may not be the case due to variance in the answer of the student.

Parameters:

`trueStudentModel` - The 'true' student model.

`index` - The place in the generation at which the 'true' student model is to be inserted.

---

gastudentmodeltest.student

Class PopulationStatistics

public class **PopulationStatistics**

Description: The statistics give an idea how close the hypothetical student models are to the 'true' student model.

Constructor Detail

PopulationStatistics

public **PopulationStatistics**(float avgFeatureDistance,
float fittestFeatureDistance,
float bestFeatDist,
float avgParameterDistance,
float fittestParameterDistance,
float averageFitness)

Constructor. To measure how well the algorithm is performing, it is necessary to calculate a number of values to give an insight into what the distances are between the features and parameters of the true student model and the hypothetical student model. This class contains the data structure for these values.

Parameters:

`avgFeatureDistance` - The feature distance between two student models is the sum of the differences of ALL the features of the model. The average feature distance is the average of all the distances (to the 'true' student model) in a generation. Note that the fitness is updated on the basis of the distance of ONE feature (i.e. the feature that is relevant to the current problem).

`fittestFeatureDistance` - This is the feature distance between the fittest student model of a generation and the 'true' student model.

`bestFeatDist` - This is the smallest feature distance that was measured in a generation. This is not necessarily the feature distance of the fittest.

`avgParameterDistance` - Similar to avgFeatureDistance, but the distance measured is between the parameters. This gives an indication of how wel the genotype matches if the phenotype matches.

`fittestParameterDistance` - Similar to fittestFeatureDistance, but the distance measured is between the parameters.

`averageFitness` - The average fitness of the student models in a generation.

Method Detail

addStatistics

public void **addStatistics**(gastudentmodeltest.student.PopulationStatistics otherStats)

Adds the values of another *populationstatistics* to this object. Used for calculating average statistics over multiple runs.

Parameters:

`otherStats` - The other statistics object whose values need to be added.

---

divideBy

public void **divideBy**(int divideBy)

Divides all fields by a number. Used for calculating the average distance.

Parameters:

`divideBy` - The number by which all the values of the object need to be divided.

---

gastudentmodeltest.student

Class StudentAnswerFeature

public class **StudentAnswerFeature**

A StudentAnswerFeature is a data structure that indicates how long a student needed to finish a problem and whether a mistake was made.

Constructor Detail

StudentAnswerFeature

public **StudentAnswerFeature**(int problemType,
float timeSpent,
boolean mistakeMade)

Constructor.

Parameters:

`problemType` - Integer that defines what the problem type of the problem was.

`timeSpent` - How much time was spent solving the problem.

`mistakeMade` - Whether or not a mistake was made.

Method Detail

getDistance

public float **getDistance**(gastudentmodeltest.student.StudentAnswerFeature otherSAF,
int penaltyForMistakeMismatch)

Returns a measure for how close this SAF is to another SAF by comparing the timeSpent values. If the booleans for mistake made do not match, penaltyForMistakeMismatch is added to the distance. Used for updating fitnesses of GA.

Parameters:

`otherSAF` - The other student answer feature.

`penaltyForMistakeMismatch` - Positive integer value described above.

Returns:

A measure for the distance between the two SAFs.

---

gastudentmodeltest.student

Class StudentModel

public class **StudentModel**
implements Comparable, Cloneable

Description: data structure representing a student model.

A student model consists of three student model features, four update parameters and a fitness value. The student model features keep track of how the student performs on the three possible problem types: the sine rule, the cosine rule and the natural logarithm rule. Each of these student model features store the time spent and the likelihood that a mistake is made on the rule. The update parameters are used together with the update rules to indicate how the student model changes. The fitness gives an indication of how well the student model has predicted the student answer features in the past.

Constructor Detail

StudentModel

public **StudentModel**(int randomSeed,
int noiseFactor)

The constructor creates an empty student model.

Parameters:

`randomSeed` - Integer used to initialize a private random object, in order to get predictable pseudo random numbers.

`noiseFactor` - Defines how noisy the student answer feature created on the basis of the true student model is.

Method Detail

calculateStudentAnswerFeature

public StudentAnswerFeature **calculateStudentAnswerFeature**(int problemType)

Calculates the studentAnswerFeature that this studentModel predicts. This is a data structure very similar to the student model features with the main difference that a boolean variable is calculated to predict whether the answer was correct or incorrect.

Parameters:

`problemType` - The type of problem that was generated

Returns:

A student answer feature

---

calculateTrueStudentAnswerFeature

public StudentAnswerFeature **calculateTrueStudentAnswerFeature**(Problem problem)

Method similar to calculateStudentAnswerFeature. Difference with the prediction of the 'true' student model: some noise (based on noiseFactor) is introduced by adding a random number to the variables. The mean of this generated number is 0 and the standard deviation is noiseFactor.

Parameters:

`problem` - The problem that was generated.

Returns:

A student answer feature based on this (noisy) student model.

---

updateTrue

public void **updateTrue**(Problem problem)

Updates the true student model features on the basis of the current problem and the updaterules / parameters. This is where the knowledge about the update rules is located. For a more generic approach, this could be implemented in XML. This update function is designed for the 'true' student model, which is the student model of the artificial student. The update function specifies how the knowledge of the artificial student changes over time.

Parameters:

`problem` - The problem that was generated.

---

update

public void **update**(StudentAnswerFeature saf)

Updates the student model features on the basis of the current problem and the updaterules / parameters. This is where the knowledge about the update rules is located. For a more generic approach, this could be implemented in XML. the first time that the studentAnswerFeature of a particular problem has been seen, the studentMODELfeature corresponding to the problem type of the studentAnswerFeature is initialized to the observed value. Note that this is not necessarily the correct value, due to noise added to the studentAnswerFeature. The mistakeMade value of the studentAnswerFeature is a boolean, but the studentModelFeature stores an likelihood value. If the observed boolean is true, the studentAnswerFeature mistake likelihood is set to a random value around 75 percent, otherwise it is set at a random value around 25 percent.

Parameters:

`saf` - The student answer feature on the basis of which the updating takes place.

---

calculateFeatureDistance

public float **calculateFeatureDistance**(gastudentmodeltest.student.StudentModel otherSM)

Calculates the total distance of the features of this student model to the features of another student model. This method is used for the calculation of the distance between the 'true' student model and the hypothetical student model.

Parameters:

`otherSM` - The student model with which this one is compared.

Returns:

A value indicating the feature distance between the two student models

---

calculateParameterDistance

public float **calculateParameterDistance**(gastudentmodeltest.student.StudentModel otherSM)

Calculates the total distance of the parameters of this student model to the parameters of another student model. This method is used for calculation of the distance between the 'true' student model and the hypothetical student model.

Parameters:

`otherSM` - The student model which this one is compared with.

Returns:

A value indicating the parameter distance between the two student models

---

compareTo

public int **compareTo**(Object o)

Defines which StudentModel is 'better' on the basis of the fitness. This is used to sort the student models by fitness: fitter are higher ranked Used by the genetic algorithm.

Parameters:

`o` - Student model to compare with

Returns:

1 if the other student model is fitter, -1 if this student model is fitter.

---

gastudentmodeltest.student

Class StudentModelFeature

public class **StudentModelFeature**
implements Cloneable

Description: describes one feature of a student model

Constructor Detail

StudentModelFeature

public **StudentModelFeature**(int problemType,
float timeSpent,
float mistakeMadeLikelihood)

Constructor

Parameters:

`problemType` - An integer representing the problemType: sine rule, cosine rule or natural logarithm rule

`timeSpent` - Value indicating how much time is spent solving this type of problem

`mistakeMadeLikelihood` - Value indicating how likely it is that a mistake is made solving this type of problem

---

Method Detail

getDistance

public float **getDistance**(gastudentmodeltest.student.StudentModelFeature otherSMF)

Compares this student model feature to another and calculates a measure for the distance. Used for statistics for GA.

Parameters:

`otherSMF` - The student model feature that this student model feature is being compared with.

Returns:
A measure for the distance

---

increaseTimeSpent

public void **increaseTimeSpent**(float update)
Used for incremental mutation.
Parameters:
`update` - The amount to increase the timeSpent with (can be negative).

---

increaseMistakeMadeLikelihood

public void **increaseMistakeMadeLikelihood**(float update)
Used for incremental mutation.
Parameters:
`update` - The amount to increase the mistakeMadeLikelihood with (can be negative).

---

forceToInterval

private float **forceToInterval**(float value)
Method used to ensure that a variable falls in the interval between 0-100
Parameters:
`value` - The value for which is checked whether it falls in the interval [0,100]
Returns:
0 if value < 0; 100 if value > 100; the original value if 0 =< value =< 100.