Ersch. in: Advances in artificial intelligence : proceedings / KI 2011, 34th Annual German Conference on AI, Berlin, Germany, October 4 - 7, 2011. Joscha Bach ; Stefan Edelkamp (ed.). - Berlin [u.a.] : Springer, 2011. - S. 357-368. - (Lecture notes in computer science ; 7006). - ISBN 978-3-642-24454-4

http://dx.doi.org/10.1007/978-3-642-24455-1_35

Algorithmic Debugging to Support Cognitive Diagnosis in Tutoring Systems

Claus Zinn

University of Konstanz, Department of Computer and Information Science,
Box D188, 78457 Konstanz, Germany
claus.zinn@uni-konstanz.de

Abstract. Cognitive modelling in intelligent tutoring systems aims at identifying a learner's skills and knowledge from his answers to tutor questions and other observed behaviour. In this paper, we propose an innovative variant of Shapiro's algorithmic debugging technique whose application can be used to pin-point learners' erroneous behaviour in terms of an irreducible disagreement to the execution trace of an expert model. Our variant has two major benefits: in contrast to traditional approaches, it does not rely on an explicit encoding on mal-rules, and second, it induces a natural teacher-learner dialogue with no need for the prior scripting of individial turns or higher-level dialogue planning.

1 Introduction

The interpretation and diagnosis of student answers is one of the central issues to be addressed when building intelligent tutoring systems (ITSs). Depending on the approach followed, it relies on a variety of knowledge sources such as domain models (modelling the expertise for the machine tutor in a given domain), task models (supporting students' problem solving process), error models (anticipating the many kinds of typical learner errors), and individual student models (capturing given learners' knowledge, strengths and weaknesses).

The first tutoring systems modelled learners solely in terms of expert skills or the lack thereof. The *overlay approach* only requires an adequate representation and operationalisation of expertise in terms of factual knowledge units and procedural skills. It assumes that all differences between learner and expert behaviour can be reduced to the learner's lack of skill. The studies of Brown & Burton and others suggest however that student errors cannot be described in terms of absent expert knowledge only [2]. They argue that the representation of expert knowledge must be complemented by a *bug library* to account for typical or high-frequent student errors in a given domain, usually in terms of buggy variants of expert skills. An erroneous student answer can then be reproduced by finding a combination of expert and buggy skills that yields the same answer. The resulting deep-structure model pin-points a student's misconception and supports the generation of appropriate corrective or remedial feedback.

The quality of cognitive diagnosis depends on the right granularity of (expert and buggy) skill decomposition, which in turn profits from an in-depth analysis

of a large and representative number of student protocols. Clearly, cognitive diagnosis that uses bug libraries can only recognise the errors it knows about, and usually the amount of buggy knowledge easily surpasses the amount of expert knowledge. The (De)Buggy programs [2,3], *e.g.*, relied on a bug library of 120 primitive and compound bugs to model errors in multi-column subtraction. Given the computational complexity of the approach, the systems performed diagnosis off-line, following a complex process of eliminating error hypotheses.

Model tracing tutors tackle the complexity issue by inviting learners to provide their answers in a piecemeal fashion. It is thus no longer necessary to reproduce a student's line of reasoning from question to (final) answer; only the student's next step towards a solution is analysed, and immediate feedback is given. While tutoring systems such as the Lisp Tutor [7,4] and the Algebra Tutor [6] have been highly successful, they are also expensive to build. A time-consuming cognitive task analysis now goes hand in hand with user interface design that encourages or enforces students to deliver their solution step by step.

In this paper, we report a method for the automated identification of errors that only requires the student's *full* answer to a given problem and a logic program encoding the expert's problem solving. The method relies on an innovative use of *algorithmic debugging* to identify learner errors by the analysis of correct *(sic)* Prolog-based procedures, given the answers of an oracle – a role being played by the student. Compared to previous approaches to cognitive diagnosis, the method does not rely on bug libraries and has low computational complexity. It supports the analysis of learners committing multiple bugs by attacking bugs one after another, and it is immune to learners giving inconsistent replies. Moreover, the execution of the method supports the generation of tutorial interactions without requiring dialogue planning or scripting. In addition, we can give a mechanisation of the oracle, which relieves the student from answering any tutor question at all. Errors can thus be identified without dialogue intervention.

2 Background

In this section, we give a brief account on multi-column subtraction, typical errors in this domain, and Shapiro's original algorithmic debugging method.

2.1 Multi-column Subtraction

Fig. 1 gives an implementation of multi-column subtraction in Prolog. Sums are processed column by column, from right to left. The predicate `subtract/2` implements the recursion, and `process_column/2` gets a partial sum, processes its right-most column and takes care of borrowing (`increment/2`) and payback (`increment/2`) actions. A column is represented as 3-element term `(M, S, R)` representing minuend, subtrahend and result cell. The program code implements the *equal additions method*, also known as *Austrian method*. When the subtrahend `S` is greater than the minuend `M`, then `M` is increased by 10 (borrowing) before the difference between `M` and `S` is taken. To compensate, the `S` in the column left to the current one is then increased by one (payback).

```
subtract(Sum, Sum)       :- finished(Sum).
subtract(Sum, NewSum) :-
     process_column(Sum, Sum1),
     shift_left(Sum1, Sum2, ProcessedColumn),
     subtract(Sum2, SumFinal),
     append(SumFinal, [ProcessedColumn], NewSum).

process_column(Sum, NewSum) :-
     butlast(Sum, LastColumn),     allbutlast(Sum,RestSum),
     subtrahend(LastColumn, Sub), minuend(LastColumn, Min),
     Sub > Min,
     add_ten_to_minuend(LastColumn, LastColumn1),
     take_difference(LastColumn1, LastColumn2),
     butlast(RestSum, LastColumnRestSum), allbutlast(RestSum, RestSum1),
     increment(LastColumnRestSum, LastColumnRestSum1),
     append(RestSum1,[LastColumnRestSum1,LastColumn2],NewSum).
process_column(Sum, NewSum) :-
     butlast(Sum, LastColumn),     allbutlast(Sum,RestSum),
     subtrahend(LastColumn, Sub), minuend(LastColumn, Min),
     Sub =< Min,
     take_difference(LastColumn, LastColumn1),
     append(RestSum,[LastColumn1], NewSum).

shift_left( SumList, RestSumList, Item ) :-
     allbutlast(SumList, RestSumList), butlast(SumList, Item).

add_ten_to_minuend( (M,S,R),  (M10,S, R) ) :- irreducible, M10 is M+10.
increment(          (M,S,R),  (M,  S1,R) ) :- irreducible, S1 is S+1.
take_difference(    (M,S,_R), (M,  S, R1)) :- irreducible, R1 is M-S.

minuend( (M,_S,_R), M).  subtrahend( (_M,S,_R), S).   finished( [] ).
```

Fig. 1. Multi-column subtraction

2.2 Error Analysis in Multi-column Subtraction

Some student errors may be caused by a simple oversight (usually, students are able to correct such errors as soon as they see them), but others are *systematic errors* (those keep re-occurring again and again). It is the systematic errors that we aim at diagnosing as they indicate a student's wrong understanding about some subject matter. The small sample of errors given in Fig. 2 can be classified as *errors of omission* (forget to do something), see (b,c); *errors of commission* (doing the task incorrectly), see (a,e); and *sequence errors* (doing the task not in the right order), see (d). Rather than having pre-compiled explicit representations of buggy program fragments (following the bug library approach) to detect such errors, we would like to use the method of algorithmic debugging on the expert program to help identifying them.

```
  5 2 4
- 2 9 8
-------
= 3 7 4
```

(a) Always subtracting the smaller from the larger number

```
    3 8 2
-     3 5
---------
=     4 7
```

(b) Not finishing the task

```
    3 2
- 1 7
-----
= 2 5
```

(c) Forgot to payback

```
  1 1 2 3
-   4 9 0
---------
= 1 7 2 2
```

(d) Perform algorithm from left to right

```
    5 2 3 4
-   5 6 7
-----------
= 2 7 7 7
```

(e) Accumulating all paybacks to highest place value

Fig. 2. A small selection of errors in subtraction

2.3 Shapiro's Algorithmic Debugging

Shapiro's algorithmic debugging technique for logic programming prescribes a systematic manner to identify bugs in programs. In the top-down variant, the program is traversed from the goal clause downwards. At each step during the traversal of the program's AND/OR tree, the programmer is taking the role of the *oracle*, and answers whether the currently processed goal holds or not. If the oracle and the buggy program agree on a goal, then algorithmic debugging passes to the next goal on the goal stack. If the oracle and the buggy program disagree on the result of a goal, then this goal is inspected further. Eventually an *irreducible agreement* will be encountered, hence locating the program's clause where the buggy behaviour is originating from.

Shapiro's algorithmic debugging method extends, thus, a simple meta-interpreter for logic programs. During the meta-interpretation of the program, it generates oracle questions. Given the programmer's answer to a Prolog clause, it is either silently executed to quickly recur algorithmic debugging on the remaining clauses, or investigated further to identify the source of the bug.

In his thesis, Shapiro gives several variants or extensions to account for various types of erroneous program code, including procedures terminating with incorrect output and non-terminating procedures. He also gives algorithms for debugging a program top-down or bottom-up, and also for minimising the number of queries the oracle (*i.e.*, the programmer) needs to answer.

3 Algorithmic Debugging in Tutoring

Shapiro devised algorithmic debugging to systematically identify bugs in incorrect programs. Our Prolog code for multi-column subtraction in Fig. 1, however, presents the expert model, that is, a presumably correct program. Given that cognitive modelling seeks to reconstruct students' erroneous procedures by an analysis of their problem-solving behaviour, it is hard to see – at least at first sight – how algorithmic debugging might be applicable in this context. There is a

simple but almost magical trick, however. We can turn Shapiro's algorithm on its head: instead of having the oracle (the programmer) specifying how the assumed incorrect program should behave, we take the expert program to take the role of the buggy program, and the role of the oracle is filled by a student's potentially erroneous answers. An irreducible disagreement between program behaviour and given answer then helps indicating a student's potential misconception.

Our algorithm, especially adapted for tutoring, is given in Fig. 3. Our algorithm traverses a given program in a top-down manner. There are four cases. If a goal is a conjunction of goals, then the algorithm is called for each conjunct. If a goal is a simple goal, then we distinguish goals that can be discussed (`on_discussion_table/1`) from those that cannot and should not be discussed. In the latter case, we check whether the current goal is a built-in predicate (in which case it is called), or whether it is a user-defined goal (in which case, its body is subjected to algorithmic debugging). It is the second clause where the main part of algorithmic debugging takes place. Given a goal that can be discussed, we use a fail-safe approach to evaluate whether it succeeds or not; we then ask the oracle whether it agrees or disagrees with the fact that the goal succeeded (with some of its arguments potentially instantiated) or failed. When there is agreement on a goal, we regard the goal (and its subgoals) as processed, and continue with other goals on the stack resulting from recursion. If there is disagreement on a goal, we check whether we have identified an irreducible agreement (in which case we terminate the algorithm with the goal in question), or not (in which case we inspect the goal's body).

Note. Programs that model expert problem solving will often have some rather technical steps that should not be subjected to questioning. In our subtraction routine, these are, *e.g.*, the clauses `butlast/2` and `allbutlast/2`; while they are necessary to implement the recursion to process partial sums from right to left, they might be hard to grasp for students and perceived as too low-level. The clause `on_discussion_table_p( Goal )` will fail on these technical clauses.

4 Example

We reconsider two of the five examples of typical subtraction errors. The GUI-based *exercise sheet* that students will interact with is presented further below.

4.1 Single Error Tracking

The result in Fig. 2(c) constitutes a simple error and can be described as a single omission in the program code; the learner forgets to honor the payback operation, following the borrowing that happened in the first (right-most) column.

$$\begin{array}{rrr} & 3 & 2 \\ - & 1 & 7 \\ \hline = & 2 & 5 \end{array}$$

```
algorithmic_debugging( (Goal1, Goal2) ) :- !,
        algorithmic_debugging( Goal1 ),
        algorithmic_debugging( Goal2 ).

algorithmic_debugging( Goal ) :-
        on_discussion_table_p( Goal ), !,
        copy_term( Goal, CopyGoal ), eval_goal( Goal, SucceededP ),
        ask_oracle( Goal, SucceededP, oracleAnswer ),
        (
          oracleAnswer = no
          ->
          (
           clause( CopyGoal, Clause ),
           (
            ( Clause == true ; Clause = ( irreducible, _Subgoals) )
           ->
            throw( Goal )
           ;
            algorithmic_debugging( Clause )
           )
          )
          ;
           true
        ).

algorithmic_debugging( Goal ) :-
        predicate_property( Goal, built_in), !,
        call(Goal).

algorithmic_debugging( Goal ) :-
        clause( Goal, Clause),
        algorithmic_debugging( Clause ).

get_diagnosis( Goal, Diagnosis ) :-
        catch( algorithmic_debugging( Goal ),
               Diagnosis,
               ( Diagnosis = true
                   -> format('~w~n',     ['correct solution'])
                   ;  format('~w ~w~n', ['irreducible disagreement:',
                                          Diagnosis]) ) ).
```

Fig. 3. Algorithmic debugging for tutoring (top-down)

Given our implementation of algorithmic debugging, the following dialogue (see clause ask_oracle/3) between system and learner will identify the error:

```
get_diagnosis(subtract([(3,1,S1),(2,7,S2)],[(3,1,2),(12,7,5)],Diagnosis).

do you agree that the following goal holds:
      subtract([ (3,1,_G226), (2,7,_G235)],[ (3,2,1), (12,7,5)])
|: no.

do you agree that the following goal holds:
      process_column([(3,1,_G475), (2,7,_G484)],[(3,2,_G475), (12,7,5)])
|: no.

do you agree that the following goal holds:
      add_ten_to_minuend((2,7,_G652), (12,7,_G652))
|: yes.

do you agree that the following goal holds:
      take_difference((12,7,_G652), (12,7,5))
|: yes.

do you agree that the following goal holds:
      increment((3,1,_G643), (3,2,_G643))
|: no.
irreducible disagreement: increment((3,1,_G643), (3,2,_G643))
```

When the student corrects the subtrahend in the left-most column, and subsequently updates the corresponding difference, we obtain the correct solution. Re-running algorithmic debugging will have both sides agree on the top clause.

4.2 Tracking Multiple Errors

The method also works well with student answers containing multiple errors. In this case, errors are attacked and repaired one by one. After each correction, the algorithm is re-run. Having the method memoizing the student answers to prior questions will avoid re-asking some – but not all – for them in subsequent runs.

Reconsider the example given in Fig. 2(a).

$$\begin{array}{rrrr} & 5 & 2 & 4 \\ - & 2 & 9 & 8 \\ \hline = & 3 & 7 & 4 \end{array}$$

First Run.

```
do you agree that the following goal holds:
      subtract([ (5,2,_G226), (2,9,_G235), (4,8,_G244)],
               [ (5,3,2), (12,10,2), (14,8,6)])
|: no.
```

```
do you agree that the following goal holds:
       process_column([ (5,2,_G523), (2,9,_G532), (4,8,_G541)],
                      [ (5,2,_G523), (2,10,_G532), (14,8,6)])
|: no.

do you agree that the following goal holds:
       add_ten_to_minuend((4,8,_G808), (14,8,_G808))
|: no.
irreducible disagreement: add_ten_to_minuend((4,8,_G808), (14,8,_G808))
```

Let us assume that the student corrects himself by only adding ten to the minuend in the right-most column. We re-run algorithmic debugging.

Second Run. The second run will produce the following interaction:

```
do you agree that the following goal holds:
       subtract([ (5,2,_G226), (2,9,_G235), (4,8,_G244)],
                [ (5,3,2), (12,10,2), (14,8,6)])
|: no.

do you agree that the following goal holds:
       process_column([ (5,2,_G523), (2,9,_G532), (4,8,_G541)],
                      [ (5,2,_G523), (2,10,_G532), (14,8,6)])
|: no.

do you agree that the following goal holds:
       add_ten_to_minuend((4,8,_G808), (14,8,_G808))
|: yes.

do you agree that the following goal holds:
       take_difference((14,8,_G808), (14,8,6))
|: no.
irreducible disagreement: take_difference((14,8,_G808), (14,8,6))
```

With the student having corrected the difference sum in this column, we start the third run.

Third Run. When we omit Oracle questions already asked (except the one that indicated the irreducible disagreement from the last run), we get the following questions:

```
do you agree that the following goal holds:
       take_difference((14,8,_G808), (14,8,6))
|: yes.

do you agree that the following goal holds:
       increment((2,9,_G799), (2,10,_G799))
|: no.
irreducible disagreement: increment((2,9,_G799), (2,10,_G799))
```

Once the student realizes the increment error, we start anew. Either the student learned from the past three runs so that he corrects the other remaining errors

in the exercise sheet, or algorithmic debugging will yield similar results for the processing of the middle column.

The practicality of our approach has to address two issues: (i) how to map Prolog queries into questions that learners can easily understand; and (ii) how to optimise the question-answering process induced by algorithmic debugging.

5 Practical Use in Tutoring

5.1 Graphical User Interface – Exercise Sheet

We have implemented the following browser-based graphical user interface for students to tackle multi-column subtraction tasks (see Fig. 4). In addition to rows for minuend, subtrahend and result cells, it also consists of explicit representations for borrow and payback cells. When students click on a "B" ("P") cell, it automatically swaps it values to "10" ("1"). Clicking on a result cell "S" brings up a number pane with digits from 0-9. The GUI's representation of the subtraction task extends the one used in the Prolog program `subtract/2`, but the B_i can be easily combined with the cells for minuends (and the P_i with the subtrahends). Moreover, to indicate to learners which GUI cells requires their attention, an adaptation to the program given in Fig. 1 was necessary, namely, the introduction of a column counter that is being increased at each recursive call to `subtract/3`. During algorithmic debugging, we can thus enhance tutorial interaction by highlighting the corresponding cells or columns in the GUI; erroneous cells that correspond to irreducible disagreements are marked in red.

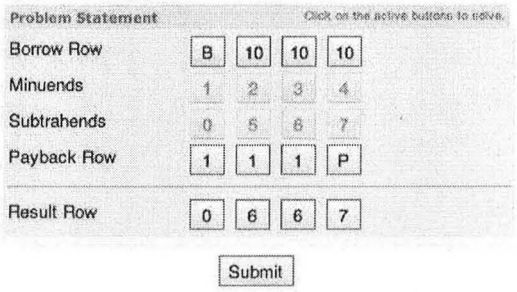

Fig. 4. GUI interaction (fully-functional prototype)

5.2 Implementing the Oracle

It may not be necessary nor pedagogically effective to forward all questions induced by algorithmic debugging to students. Moreover, some learners will hardly be patient enough to go through many iterations of system-learner interactions once they have already given an answer to a given problem. To address this issue, we have build an oracle that can answer questions about the subtraction

task: it takes a learner's *full* answer from the exercise sheet, and extracts from it those parts required to answer a given question. Fig. 5 depicts a code fragment to handle cases for the top clause `subtract/2` and the clause `increment/2`.

```
oracle( Step, YesNo ) :-
        get_student_answer_from_gui( Answer ), !,
        oracle_helper( Step, Answer, YesNo ).

oracle_helper( subtract(Col, _Input, Output), Answer, YesNo ) :-
        length(Answer, NumberColumns),
        Col = NumberColumns, !,
        (
          subsumes_term(Output, Answer) -> YesNo = yes ; YesNo = no
        ).

oracle_helper( subtract( Col, Input, Output ), Answer, YesNo ) :-
        allbutlast(Answer, AnswerRed),
        oracle_helper( subtract(Col, Input, Output), AnswerRed, YesNo ).

oracle_helper(increment(Col,_Input,(_M2, S2, R2)),Answer,YesNo ) :-
        length(Answer, NumberColumns),
        Col = NumberColumns, !,
        butlast( Answer, ( _MS, SS, _RS) ),
        ( S2 == SS -> YesNo = yes ; YesNo = no ).

oracle_helper( increment(Col,InputCol,OutputCol),Answer,YesNo ) :-
        allbutlast(Answer, AnswerRed),
        oracle_helper(increment(Col,InputCol,OutputCol),AnswerRed,YesNo).

?- get_diagnosis([(3,1,S1),(2,7,S2)], [(3,1,2), (12,7,5)], Diagnosis).
 ==> irreducible disagreement: increment(2, (3,1,_G3659), (3,2,_G3659))
```

Fig. 5. Implementing the Oracle (fragment for subtraction task)

The clause `oracle/2` is called with the current step executed by the expert model and outputs `yes` or `no`, depending on whether the learner agrees or disagrees with the expert step. The new argument `Col` marks the column currently processed. It is compared to the length of `Answer`, initially capturing the full sum given by the learner. When they are equal, we extract from `Answer` the relevant parts and check them for correctness with respect to the expert answer; otherwise, we recurse with the learner's answer reduced by the one column no longer of interest.

6 Discussion and Related Work

In Fig. 2, we have given a small selection of typical errors in subtraction. Our use of algorithmic debugging alone will surely fail to identify the full nature of

learners' erroneous behaviour, but algorithmic debugging in combination with other techniques will get closer to this (see Future Work).

Our adaptation and use of algorithmic debugging in tutoring is similar in spirit to model tracing. While our discussion assumed the learner to give a full answer to a given subtraction task, we can instrument our algorithm to provide next-step help in case where learners only submitted a partial sum. Note also that our approach does not require any representation of buggy skills. The interaction induced by algorithmic debugging compares learner actions to those of the expert model only – admittingly at the cost of providing less effective remedial feedback once a disagreement has been identified. On the positive side, our expert model is *just* an executable Prolog program. While it encodes skills that we want the learner to acquire, it contains no (other) tutoring-specific information. The power and simplicity of our approach is due to the meta-programming perspective.

There is only little research in the ITS community that builds upon logic programming techniques. Thirty years ago, Self described student modelling in terms of an induction problem, where student observations can be seen as the I/O behaviour of an unknown program that needs to be induced from such behaviour [8]. An induction logic programming approach to cognitive diagnosis has been taken by [5]. Here, expert knowledge is represented as a set of Prolog clauses, and Shapiro's Model Inference System (MIS) [9] is used to synthesise the student model from expert knowledge and student answers. Once the student model, a set of Prolog clauses, is constructed, Shapiro's Program Diagnosis System (PDS), based upon algorithmic debugging, is used to identify students' misconceptions, that is, the bugs in the MIS-constructed Prolog program.

In [1], Beller & Hoppe use a fail-safe meta-interpreter to identify student error. A Prolog program, modelling the expert knowledge for doing subtraction, is executed by instantiating its output parameter with the student answer. While standard Prolog interpretation would fail, a fail-safe meta-interpreter can recover from execution failure, and can also return an execution trace. Beller & Hoppe formulate *error patterns* which are then matched against the execution trace, and where each successful match is indicating a plausible student bug.

7 Conclusion and Future Work

There is good evidence that a sound methodology of cognitive diagnosis in intelligent tutoring can be realised in the framework of logic programming. Its declarative aspect, its view that program equals data, the substantial work in areas such as meta-level interpreters, partial evaluation, reasoning about programs, algorithmic debugging and inductive logic programming shows that there is ample potential to harness such techniques to support cognitive diagnosis in the context of intelligent tutoring systems. In this paper, we reported on our work to use algorithmic debugging to advance cognitive diagnosis in this direction.

In the future, we aim at systematically studying, adapting and combining various logic programming methods to effectively perform cognitive diagnosis in educational settings. We are currently working on a combination of algorithmic

debugging and program transformation. Once algorithmic debugging identifies an irreducible disagreement between expert behaviour and the learner, we induce code perturbations (*i.e.* bugs) into the expert program to make the disagreement disappear. After a series of applications of algorithmic debugging and code perturbations, we obtain a buggy procedure which models the learner's erroneous behaviour. In a related strand, we would like to adapt and use inductive program synthesis techniques; those, for instance, will be necessary to cover program transformations where novel elements need to be constructed. Once an arsenal of such techniques is developed, we seek to characterise the types of errors that can be diagnosed with each method or combination of methods, and to determine the effectiveness of remedial feedback based on such diagnosis.

Acknowledgements. The idea of turning Shapiro's method for algorithmic debugging on its head to support the diagnosis of student errors originated from Alan Smaill and Alan Bundy (both University of Edinburgh); personal communication (Note 1396).

Many thanks to the reviewers whose comments helped improve the paper.

References

1. Beller, S., Hoppe, U.: Deductive error reconstruction and classification in a logic programming framework. In: Brna, P., Ohlsson, S., Pain, H. (eds.) Proc. of the World Conference on Artificial Intelligence in Education, pp. 433–440 (1993)
2. Brown, J.S., Burton, R.R.: Diagnostic models for procedural bugs in basic mathematical skills. Cognitive Science 2, 155–192 (1978)
3. Burton, R.R.: Debuggy: Diagnosis of errors in basic mathematical skills. In: Sherman, D., Brown, J.S. (eds.) Intelligent Tutoring Systems. Academic Press, London (1982)
4. Corbett, A.T., Anderson, J.R., Patterson, E.J.: Problem compilation and tutoring flexibility in the lisp tutor. In: International Conference of Intelligent Tutoring Systems, Montreal (1988)
5. Kawai, K., Mizoguchi, R., Kakusho, O., Toyoda, J.: A framework for ICAI systems based on inductive inference and logic programming. New Generation Computing 5, 115–129 (1987)
6. Koedinger, K.R., Anderson, J.R., Hadley, W.H., Mark, M.A.: Intelligent tutoring goes to school in the big city. Journal of Artificial Intelligence in Education 8(1), 30–43 (1997)
7. Reiser, B.J., Anderson, J.R., Farrell, R.G.: Dynamic student modelling in an intelligent tutor for lisp programming. In: IJCAI 1985: Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 8–14. Morgan Kaufmann Publishers Inc., San Francisco (1985)
8. Self, J.: Student models and artificial intelligence. Computers & Education 3, 309–312 (1979)
9. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertations. MIT Press (1983); Thesis (Ph.D.) – Yale University (1982)