Heuristic Search over Program Transformations

Claus Zinn(✉)

Department of Computer Science, University of Konstanz, Konstanz, Germany
claus.zinn@uni-konstanz.de

Abstract. In prior work, we have developed a method for the automatic reconstruction of buggy Prolog programs from correct programs to model learners' incorrect reasoning in a tutoring context. The method combines an innovative variant of algorithmic debugging with program transformations. Algorithmic debugging is used to indicate a learner's error and its type; this informs a program transformation that "repairs" the expert program into a buggy variant that is closer at replicating a learner's behaviour. In this paper, we improve our method by using heuristic search. To search the space of program transformations, we estimate the distance between programs. Instead of only returning the first irreducible disagreement between program and Oracle, the algorithmic debugger now traverses the entire program. In the process, all irreducible agreements and disagreements are counted to compute the distance metrics, which also includes the cost of transformations. Overall, the heuristic approach offers a significant improvement to our existing blind method.

1 Introduction

Typically, programs have bugs. We are interested in runtime bugs where the program terminates with output that the programmer judges incorrect. In these cases, Shapiro's algorithmic debugging technique can be used to pinpoint the location of the error. A dialogue between the debugger and the programmer unfolds until the meta-interpretation of the program reaches a statement that captures the cause of disagreement between the program's actual behaviour and the programmer's intent of how the program should behave. Once the bug has been located, it is the programmer's task to repair the program, and then, to start another test-debugging-repair cycle. Let us make the following assumption: there exists an Oracle that relieves the programmer from answering any of the questions during the debugging cycle; the Oracle "knows" the programmer's intent for each and every piece of code. With the mechanisation of the Oracle to locate the program's bugs, we now seek to automate the programmer's task to repair the bug, and thus, to fully automate the test-debug-repair cycle.

In the tutoring context, Oracles can be mechanised: for a given domain of instruction, there is always a reference model that defines expert problem solving behaviour. Moreover, a learner's problem solving behaviour is judged with regard to this model; a learner commits a mistake whenever the learner deviates from the expert problem solving path. Algorithmic debugging can be used to identify the location of learners' erroneous behaviour. For this, we have to turn Shapiro's method on its head: we take the expert program to take the role of the buggy program, and the learner to take the role of the programmer, that is, the Oracle. As in the traditional method, any disagreement between the two parties indicates the location of the bug. Moreover, we can relieve the learner from answering Oracle questions. Answers to all questions can be reconstructed from the learner's answer to a given problem, using the expert model [11].

With the ability to locate a learner's error, we now seek to "repair" the expert program (assumed buggy) in such a way that it reproduces the learner's erroneous (assumed expert) behaviour. The resulting program acts as symbolic artifact of a deep diagnosis of a learner's problem solving process; it can be used to inform effective remediation, helping learners to realize and correct their mistakes. Ideally, repair operators shall mirror typical learner errors. This is feasible indeed. There is a small set of error types, and many of them can be formally described in a domain-independent manner.

With the identification of an error's location, and a small, effective set of mutation operators for program repair, we strive to fully automate the test-debug-repair cycle in the tutoring context. Our approach is applicable for a wider context, given the specification of an ideal program and a theory of error.

Main contributions. To address an important issue in intelligent tutoring, the deep diagnosis of learner input, we cast the problem of automatically deriving one (erroneous) program from another (expert) program as a heuristic search problem. We define a metric that quantifies the distance of two given programs with regard to an input/output pair. We define a number of domain-independent code perturbation operators whose execution transforms a given program into its mutated variant. Most mutation operators encode typical actions that learners perform when encountering an impasse during problem solving. We show the effectiveness of our approach for the most frequent learner bugs in the domain of multi-column subtraction. Erroneous procedures are automatically derived to reproduce these errors. This work extends and generalises our previous work in this area [11,12] with regard to the heuristic search approach, which is novel.

Overview. Section 2 gives a very brief review on student errors in tutoring. It presents multi-column subtraction as domain of instruction and gives an encoding of the expert model in Prolog. For each of the top-eight learner errors in this domain, we demonstrate how the expert model needs to be perturbated to reproduce them. We show that most perturbations are based on a small but effective set of mutation operators. Also, we briefly review our existing method of error diagnosis in the tutoring context. In Sect. 3, we improve and generalise our method. The problem of deriving one program from another is cast in terms of a heuristic search problem. We introduce a distance metrics between programs that is based on algorithmic debugging, and use a best-first search algorithm to illustrate and evaluate the effectiveness of our approach. Section 4 discusses our approach and relates it to existing work. Section 5 concludes with future work.

2 Background

2.1 Human Error in Tutoring

When learning something new, one is bound to make mistakes. Effective teaching depends on deep cognitive analyses to diagnose learners' problem solving paths, and subsequently to repair the incorrect parts. Good teachers are thus capable to reconstruct students' erroneous procedures and use this information to inform their remediation. In the area of elementary school mathematics, our chosen tutoring domain, the seminal works of Brown and Burton [2,3], O'Shea and Young [10], and VanLehn [9], among others, extensively studied the subtraction errors of large populations of pupils. Their research included a computational account of errors by manually constructing cognitive models that reproduced learners' most frequent errors. The main insight of this research is that student errors are seldom random. There are two main causes. The first cause is that student errors may result from *correctly* executing an erroneous procedure; for some reasons, the erroneous rather than the expert procedure has been acquired. The second cause is based on VanLehn's theory of *impasses* and *repairs*. Following VanLehn, learners "know" the correct procedure, but face difficulties executing it. They "treat the impasse as a problem, solve it, and continue executing the procedure" [9, p. 42]. The repair strategies to address an impasse are known to be common across student populations and domains. Typical repairs include executing only the steps known to the learner and to skip all other steps, or to adapt the situation to prevent the impasse from happening.

2.2 Expert Model for Multi-column Subtraction

Figure 1 depicts the entire cognitive model for multi-column subtraction using the decomposition method. The Prolog code represents a subtraction problem as a list of column terms `(M, S, R)` consisting of a minuend `M`, a subtrahend `S`, and a result cell `R`. The main predicate `subtract/2` determines the number of columns and passes its arguments to `mc_subtract/3`.[1] This predicate processes columns from right to left until all columns have been processed and the recursion terminates. The predicate `process_column/3` receives a partial sum, and processes its right-most column (extracted by `last/2`). There are two cases. Either the column's subtrahend is larger than its minuend, when a borrowing operation is required, or the subtrahend is not larger than the minuend, in which case we can subtract the former from the latter (calling `take_difference/4`). In the first case, we add ten to the minuend (`add_ten_to_minuend/3`) by borrowing from the left (calling `decrement/3`). The decrement operation also consists of two clauses, with the second clause being the easier case. Here, the minuend of the column left to the current column is not zero, so we simply reduce the minuend by one. If the minuend is zero, we need to borrow again, and hence `decrement/3` is called recursively. When we return from recursion, we first add ten to the minuend, and then reduce it by one.

[1] The argument `CurrentColumn` is passed onto most other predicates; it is only used to help automating the Oracle.

```
subtract(PartialSum, Sum) ←
            length(PartialSum, LSum),
            mc_subtract(LSum, PartialSum, Sum).

mc_subtract(_, [], []).
mc_subtract(CurrentColumn, Sum, NewSum) ←
            process_column(CurrentColumn, Sum, Sum1),
            shift_left(CurrentColumn, Sum1, Sum2, ProcessedColumn),
            CurrentColumn1 is CurrentColumn − 1,
            mc_subtract(CurrentColumn1, Sum2, SumFinal),
            append(SumFinal, [ProcessedColumn], NewSum).

process_column(CurrentColumn, Sum, NewSum) ←
            last(Sum, LastColumn), allbutlast(Sum, RestSum),
            minuend(LastColumn, M), subtrahend(LastColumn, S),
            S > M, !,
            add_ten_to_minuend(CurrentColumn, M, M10),
            CurrentColumn1 is CurrentColumn − 1,
            decrement(CurrentColumn1, RestSum, NewRestSum),
            take_difference(CurrentColumn, M10, S, R),
            append(NewRestSum, [(M10, S, R)], NewSum).

process_column(CurrentColumn, Sum, NewSum) ←
            last(Sum, LastColumn), allbutlast(Sum, RestSum),
            minuend(LastColumn, M), subtrahend(LastColumn, S),
            % S =< M,
            take_difference(CurrentColumn, M, S, R),
            append(RestSum, [(M, S, R)], NewSum).

shift_left( _CurrentColumn, SumList, RestSumList, Item ) ←
            allbutlast(SumList, RestSumList), last(SumList, Item).

decrement(CurrentColumn, Sum, NewSum ) ←
            irreducible,
            last( Sum, (M, S, R) ), allbutlast( Sum, RestSum),
            M == 0, !,
            CurrentColumn1 is CurrentColumn − 1,
            decrement(CurrentColumn1, RestSum, NewRestSum ),
            NM is M + 10,
            NM1 is NM − 1,
            append( NewRestSum, [(NM1, S, R)], NewSum),

decrement(CurrentColumn, Sum, NewSum) ←
            irreducible,
            last( Sum, (M, S, R) ), allbutlast( Sum, RestSum),
            % \+ (M == 0),
            M1 is M − 1,
            append( RestSum, [(M1, S, R)], NewSum ).

add_ten_to_minuend( _CC, M, M10) ← irreducible, M10 is M + 10.
take_difference(_CC, M, S, R) ← irreducible, R is M − S.

minuend( (M, _S, _R), M).
subtrahend( (_M, S, _R), S).

allbutlast([], []).
allbutlast([_H], []).
allbutlast([H1|[H2|T]], [H1|T1]) ← allbutlast([H2|T], T1).

irreducible.
```

Fig. 1. The decomposition method for subtraction in prolog

$$\begin{array}{cccc} & & 9 & \\ & 3 & \cancel{10} & 11 \\ & \cancel{4} & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 2 & 0 & 2 \end{array}$$

(a) correct solution

$$\begin{array}{cccc} & 4 & 0 & 1 \\ - & 1 & 9 & 9 \\ \hline = & 3 & 9 & 8 \end{array}$$

(b) smaller-from-larger

$$\begin{array}{cccc} & 3 & 10 & 11 \\ & \cancel{4} & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 2 & 1 & 2 \end{array}$$

(c) stops-borrow-at-zero

$$\begin{array}{cccc} & 2 & & \\ & \cancel{3} & 10 & 11 \\ & \cancel{4} & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 1 & 1 & 2 \end{array}$$

(d) borrow-across-zero

$$\begin{array}{cccc} & & 9 & 11 \\ & 4 & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 3 & 0 & 2 \end{array}$$

(e) borrow-from-zero

$$\begin{array}{cccc} & & 10 & 11 \\ & 4 & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 3 & 1 & 2 \end{array}$$

(f) borrow-no-decrement

$$\begin{array}{cccc} & & & 11 \\ & 4 & 0 & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 3 & 9 & 2 \end{array}$$

(g) stops-borrow-at-zero diff-0-N=N

$$\begin{array}{cccc} & 2 & & \\ & \cancel{3} & 11 & 11 \\ & \cancel{4} & \cancel{1} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 1 & 2 & 2 \end{array}$$

(h) always-borrow-left

$$\begin{array}{cccc} & 3 & & 11 \\ & \cancel{4} & 0 & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 2 & 9 & 2 \end{array}$$

(i) borrow-across-zero diff-0-N=N

Fig. 2. A correct solution, and the top-eight bugs sets, see [9, p. 195].

2.3 Buggy Sets in Multi-column Subtraction

Figure 2(a) depicts the correct solution to the subtraction problem 401 – 199, the Fig. 2(b)–(i) show how the top-eight bug sets from the DEBUGGY study [9, p. 195, p. 235] manifest themselves in the same task. All erroneous answers are rooted in learners' difficulty to borrow: the errors in Fig. 2(b) and (f) result from the learners' more general impasse "does not know how to borrow", and the errors in Fig. 2(c)–(e) results from the learners' more specific impasse "does not know how to borrow from zero". All other errors, but Fig. 2(h), are variations of the previous error types. Figure 2(h) is better explained by the incorrect acquisition of knowledge rather than within the impasse-repair theory.

We now describe how the expert procedure given in Fig. 1 needs to be "repaired" to reproduce each of the top-eight bugs.

smaller-from-larger: *the student does not borrow, but in each column subtracts the smaller digit from the larger one* [9, p. 228]. The impasse "learner does not know how to borrow" is overcome by not letting borrowing to happen. The expert model is perturbated at the level of `process_column/3`. In its first clause, we delete the calls to `add_ten_to_minuend/3` (line 15) and `decrement/3` (line 17). As a consequence, we replace all remaining occurrences of `M10` and `NewRestSum` with `M` and `RestSum`, respectively. Moreover, we swap the arguments for `M` and `S` when taking differences (line 18).

borrow-no-decrement: *when borrowing, the student adds ten correctly, but does not change any column to the left* [9, p. 223]. The learner addresses the impasse "does not know how to borrow" with a partial skipping of steps. In the first clause of `process_column/3`, the subgoal `decrement/3` (line 17) is deleted; the remaining occurrence of `NewRestSum` is then replaced by `RestSum` (line 19).

stops-borrow-at-zero: *instead of borrowing across a zero, the student adds ten to the column he is doing, but does not change any column to the left* [9, p. 229]. The impasse "learner does not know how to borrow from zero" is overcome by not performing *complete* borrowing when the minuend in question is zero. The recursive call to `decrement/3` (line 33) and the goals producing `NM1` and `NM` (lines 34, 35) are removed, and the remaining occurrence of `NM1` replaced by `M` (line 36).

borrow-across-zero: *when borrowing across a 0, the student skips over the 0 to borrow from the next column. If this causes him to have to borrow twice, he decrements the same number both times* [9, p. 114, p. 221]. Same impasse, different repair. The clauses that produce `NM1` and `NM` (lines 34, 35) are removed; the remaining occurrence of `NM1` in `append/3` replaced by `M` (line 36).

borrow-from-zero: *instead of borrowing across a zero, the student changes the zero to nine, but does not continue borrowing from the column to the left* [9, p. 223]. Same impasse, yet another repair: the assignments `NM` and `NM1` stay in place, but the recursive call to `decrement/3` (line 33) is deleted; the occurrence of `NewRestSum` is replaced by `RestSum` (line 36).

stops-borrow-at-zero diff0-N=N: *when the student encounters a column of the form* $0 - N$, *he does not borrow, but instead writes* N *as the answer, possibly combined with* `stops-borrow-at-zero`. For `diff-0-N=N`, we shadow the existing clause for taking differences with `take_difference(M, S, R):- M == 0, R = S.` To ensure that no borrowing operation is performed in case the minuend is zero, the first clause of `process_column/3` is modified. The constraint S>M (line 14) is complemented with `\+ (M == 0)`; line 23 is changed to `(S =< M) ; (M == 0)`.

always-borrow-left: *the student borrows from the left-most digit instead of borrowing from the digit immediately to the left* [9, p. 225]. This error is best explained by the incorrect acquisition of knowledge rather than within the impasse-repair theory. To reproduce it, we shadow the existing clauses for `decrement/3` with `decrement([(M,S,R)|OtherC], [(M1,S,R)|OtherC]) :- !, M1 is M - 1.`

borrow-across-zero diff-0-N=N: *see above.* With both errors already been dealt with, we combine the respective perturbations to reproduce this error.

Summary. All error types except `always-borrow-left` require the deletion of one or more subgoals, with a tidying-up phase for their input and output arguments. For `smaller-than-larger`, the swapping of arguments was necessary. For `always-borrow-left`, we shadowed the existing clauses for `decrement/3` with a new clause. While the top five errors can be reproduced by syntactic means, the last three errors seem to require elements whose construction will be hard to mechanise.

```
function ReconstructErroneousProcedure($Program, Problem, Solution$)
    $(Disagr, Cause) \leftarrow AlgorithmicDebugging(Program, Problem, Solution)$
    if $Disagr = nil$ then
        return $Program$
    else
        $NewProgram \leftarrow$ Perturbation($Program, Disagr, Cause$)
        ReconstructErroneousProcedure($NewProgram, Problem, Solution$)
    end if
end function

function Perturbation($Program, Clause, Cause$)
    return chooseOneOf(Cause)
        DeleteCallToClause($Program, Clause$)
        DeleteSubgoalsOfClause($Program, Clause$)
        SwapClauseArguments($Program, Clause$)
        ShadowClause($Program, Clause$)
end function
```

Fig. 3. Pseudo-code: compute variant of *Program* to reproduce a learner's *Solution*.

2.4 Existing Method

In [12], we have presented a method that interleaves algorithmic debugging with program transformations for the automatic reconstruction of learners' erroneous procedure, see Fig. 3. The function `ReconstructErroneousProcedure/3` is recursively called until a program is obtained that reproduces learner behaviour, in which case there are no further disagreements. Note that multiple perturbations may be required to reproduce single bugs, and that multiple bugs are tackled by iterative applications of algorithmic debugging and code perturbation.

The irreducible disagreement resulting from the algorithmic debugging phase locates the code pieces where perturbations must take place; its cause determines the kind of perturbation. The function `Perturbation/3` can invoke various kinds of transformations: the deletion of a call to the clause in question, or the deletion of one of its subgoals, or the shadowing of the clause in question by a more specialized instance, or the swapping of the clause' arguments. These perturbations reflect the repair strategies learners use when encountering an impasse.

Our algorithm for clause call deletion, e.g., traverses a given program until it identifies a clause whose body contains a call to the clause `Clause` in question; once identified, it removes `Clause` from the body and replaces all occurrences of its output argument by its input argument in the adjacent subgoals as well as in the clause's head, if present. Then, the modified program is returned.

There are many choice points as an action can materialise in many different ways. Our original method uses Prolog's built-in depth-first mechanism to *blindly* search the space of program transformations. Our new method uses a heuristics to make *informed* decisions during search.

3 Heuristic Search over Program Transformations

The problem of automatically reconstructing a Prolog program to model a learner's incorrect reasoning can be cast as a heuristic search problem. The initial state holds a Prolog program that solves arbitrary multi-column subtraction tasks in an expert manner. The goal state holds the program's perturbated variant whose execution reproduces the learner's erroneous behaviour. For each state s, a successor state s' can be obtained by the application of a single perturbation operator op_i. We seek a sequence of perturbation actions $op_1, op_2, ...op_n$ to define a path between start and goal state, with minimal costs.

3.1 Heuristic Function

Best-first search depends on a heuristic function to evaluate a node's distance to the goal node. For this, we extend our variant of algorithmic debugging. A heuristic score could be obtained, e.g., by counting the number of agreements until the first irreducible disagreement is found; however, when errors occur early in the problem solving process, this simple scoring performs poorly. Modifying the algorithmic debugger to always traverse the entire program and count all irreducible agreements and disagreements during traversal yields a better score.

Figure 4 depicts the algorithmic debugger in pseudo-code; it extends a simple meta-interpreter. Before start, both counters are initialised, and the references set for `Goal`, `Problem`, `Solution` to hold the top-level goal, the task to be solved and the learner's `Solution` to the task, respectively. There are four main cases. The meta-interpreter encounters either (i) a conjunction of goals, (ii) a goal that is a system predicate, (iii) a goal that does not need to be inspected, or (iv) a goal that needs to be inspected. For (i), algorithmic debugging is called recursively on each of the goals of the conjunctions; for (ii), the goal is called; and for (iii), we obtain the goal's body and ask the meta-interpreter to inspect it. The interesting aspect is case (iv) for goals marked relevant. Here, the goal is evaluated by both the expert program (using `call/1`) and the Oracle. The Oracle retrieves the learner's solution for the given `Problem` and reconstructs from it the learner's answer to the goal under discussion. Now, there are two cases. If system and learner agree on the goal's result, then the goal's weight is determined and added to the number of agreements; if they disagree, the goal must be inspected further to identify the exact location of the disagreement. If the goal is a leaf node, the irreducible disagreement has been identified and the disagreement counter is incremented by one; otherwise, the goal's body is retrieved and subjected to algorithmic debugging. The heuristic score is obtained by subtracting the number of agreements from the number of disagreements.

3.2 Best-First Search: Guiding Program Transformations with A^*

Typically, a search method maintains two lists of states: an *open list* of all states still to be investigated for the goal property, and a *closed list* for all states that were already checked for the goal property but where the check failed. Among

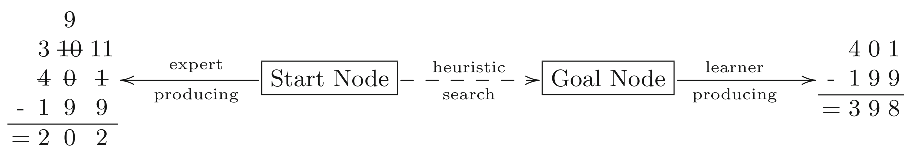
```
1: $NumberAgreements \leftarrow 0$, $NumberDisagreements \leftarrow 0$
2: $Problem \leftarrow$ current task to be solved, $Solution \leftarrow$ learner input to task
3: $Goal \leftarrow$ top-clause of routine, with input $Problem$ and output $Solution$
4: procedure algorithmicDebugging($Goal$)
5:     if $Goal$ is conjunction of goals ($Goal1, Goal2$) then
6:         $\leftarrow$ algorithmicDebugging($Goal1$)
7:         $\leftarrow$ algorithmicDebugging($Goal2$)
8:     end if
9:     if $Goal$ is system predicate then
10:        $\leftarrow$ call($Goal$)
11:    end if
12:    if $Goal$ is not on the list of goals to be discussed with learners then
13:        $Body \leftarrow$ getClauseSubgoals($Goal$)
14:        $\leftarrow$ algorithmicDebugging($Body$)
15:    end if
16:    if $Goal$ is on the list of goals to be discussed with learners then
17:        $SystemResult \leftarrow$ call($Goal$)
18:        $OracleResult \leftarrow$ oracle($Goal$)
19:        if results agree on $Goal$ then
20:            $Weight \leftarrow$ computeWeight($Goal$)    ▷ compute # of skills in proof tree
21:            $NumberAgreements \leftarrow NumberAgreements + Weight$
22:        else
23:            if $Goal$ is leaf node (or marked as irreducible) then
24:                $NumberDisagreements \leftarrow NumberDisagreements + 1$
25:            else
26:                $Body \leftarrow$ getClauseSubgoals($Goal$)
27:                $\leftarrow$ algorithmicDebugging($Body$)
28:            end if
29:        end if
30:    end if
31: end procedure
32: $Score \leftarrow NumberDisagreements - NumberAgreements$
```

Fig. 4. Pseudo-code: top-down traversal, keeping track of (dis-)agreements.

all the open states, *greedy* best-first search always selects the most promising candidate, i.e., the candidate that is most likely the closest to a given goal state. Our approach also takes into account the cost of program transformations. With the heuristic function defined as $f(n) = g(n) + h(n)$, we thus implement the A^*-algorithm. The cost function $g(n)$ returns the cost of producing state n. The function $h(n)$ estimates the distance between the program in state n and the goal state; it is described as `Score` in Fig. 4.

Representation. Each state in the search tree is represented by the term `(Algorithm, IrreducibleDisagreement, Path)`, encoding a reified version of a Prolog program, the *first* irreducible agreement between program and learner behavior, and the path of prior perturbation actions to reach the current state. Each state n is also associated with a numerical value $f(n)$ that quantifies its production cost as well as the `Algorithm`'s distance to the algorithm of the goal state. A successor state of a given state results from applying a perturbation action. The action obtains a Prolog program, performs some sort of mutation, and returns a modified program. – We discuss our approach by example.

Initialisation. The start node holds the expert program (see Fig. 1) that produces the correct solution. Sought is a mutated variant of the expert program to produce the learner's erroneous solution, here the error `smaller-from-larger`:



Best-first search starts with initialising a heap data structure. For this, the start node's distance to the goal node is estimated, using the algorithm given in Fig. 4. There is no single agreement between expert program solving behaviour and learner behaviour, i.e., no single subtraction cell has been filled out the same way. There are six disagreements, yielding a heuristic score of $6 - 0 = 6$.

To inform the generation of the node's children, the first of the six irreducible disagreements – `add_ten_to_minuend(3, 1, 1)` – (1 instead of 11) is attached to the node's second component. The third component is initialised with the empty path `[]` (cost 0). The node and its estimate is then added to the empty heap.

Checking for Goal State. A state is a goal state when its associated program passes algorithmic debugging with zero disagreements. In this case, best-first search terminates with the goal state, returning the node's algorithm and its path, i.e., a list of actions that were applied to reach the goal state. Here, the initial node, with a non-zero number of disagreements is not the goal node.

Generation of Successor Nodes. If a given state is not the goal state, the state's successors are computed. Given the state's algorithm and the first irreducible disagreement that indicates the location of the "error", Prolog is asked to `findall` applicable perturbation actions, see Fig. 3. For the initial state, we obtain:

- n_1 `DeleteCallToClause/2`: deletion of the call to `add_ten_to_minuend/3` in the first program clause `process_column/3` (line 15).
- n_2 `ShadowClause/2`: addition of the irreducible disagreement (learner's view) `add_ten_to_minuend(3, 1, 1) :- irreducible.` to the program.
- n_3 `DeleteSubgoalsOfClause/2`: deletion of subgoals from the definition of the predicate `add_ten_to_minuend/3`. As the goal `irreducible/0` cannot be deleted as it is needed by the Oracle, the only permissible action is to delete the subgoal `M10 is M + 10`, and to replace `M10` by `M` in the clause' header.

To add a successor node to the heap, the existing path is extended with the respective action taken. Also, for each node's algorithm, its first irreducible disagreement with the learner must be identified, and the distance to the goal node must be determined. For all successor nodes, we get the irreducible disagreement `decrement(2,[(4,1,S1), (0,9,S2)],[(3,1,S1), (9,9,S2)])`.

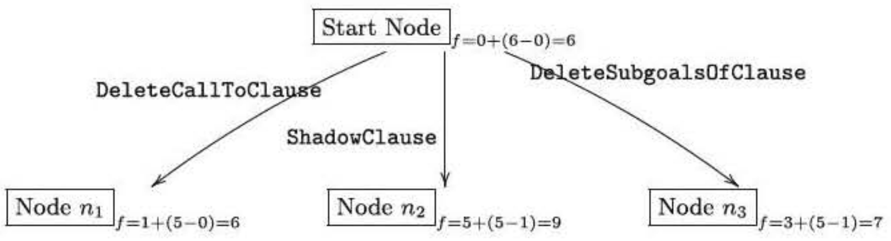


Fig. 5. Best-first search over program transformations: first expansion.

Now, consider the nodes' (dis-) agreement scores. Each of the nodes has five disagreements, one less than in the parent node; the second and third child now also feature one agreement. In node n_2, there is an agreement with the new clause `add_ten_to_minuend/3` added; in node n_3, there is an agreement with the perturbated clause `add_ten_to_minuend/3`.

Action Cost. Some program transformations are better than others. Consider the `ShadowClause` action, yielding mutations that are specific to a given input/output pair. The resulting program will, thus, reproduce the learner's error only for the given subtraction task, not for other input. The action's lack of generality is acknowledged by giving it a high cost, namely 5. Hence, the action will only be used when more general and less costly actions are not applicable.

The action `DeleteSubgoalsOfClause` can delete more than a single subgoal from the predicate indicated by the disagreement. This adds a notion of focus to the perturbation and mirrors the fact that learners often address an impasse with skipping one or more steps of the skill in question. Its cost is defined by the number of subgoals deleted; a penalty is added, however, when the resulting body is left with the single subgoal `irreducible`. The mutations performed by `DeleteCallToClause` and `SwapClauseArguments` have a cost of 1.

Score and Continued Search. Figure 5 depicts the scores obtained. Best-first search selects the child with the lowest valuation, n_1. Its irreducible disagreement on the `decrement/3` operation in column 2 can be addressed by eight different repairs: the deletion of the call to `decrement/3` in the first clause of `process_column/3` (line 17), the addition of the disagreement clause (the learner's view) to the program, and the removal of one or more subgoals in any of the two clause definitions for `decrement/3` (6 possible repairs). It shows that the deletion of line 17 yields the algorithm with the lowest overall estimate; the algorithm's irreducible disagreement at `take_difference(3,1,9,8)` (8 *vs.* -8) lets best-first search determine the final perturbation action, which is `SwapClauseArguments/2` to swap the arguments of `take_difference/4` in the first clause of `process_column/3`.

3.3 Evaluation

We have tested our new approach against the eight most frequent bugs of VanLehn's study (Fig. 2). For this, we have implemented the following three search

Table 1. Evaluation: blind-search *vs.* cost-based search *vs.* A^*

Error	Blind	Cost-based	A^*
smaller-from-larger	33.882K-N	2.385K-Y	659K-Y
borrow-no-decrement	425K-N	425K-Y	425K-Y
stops-borrow-at-zero	406K-N	640K-N	407K-Y
borrow-across-zero	444K-N	1.126K-Y	446K-Y
borrow-from-zero	457K-N	510K-Y	457K-Y
stops-borrow-at-zero. diff0-N=N	5.054K-N	108.832K-C	2.786K-C
always-borrow-left	111K-N	2.831K-C	111K-C
borrow-across-zero. diff0-N=N	5.053K-N	203.125K-C	1.874K-C

methods: blind (depth-first) search where each node n is associated with the value $f(n) = 1$, cost-based search where each node is associated with its construction cost $f(n) = g(n)$, and A^*-search where each node is associated with its construction cost and its estimated distance to the goal node: $f(n) = g(n)+h(n)$.

Table 1 compares the three search methods using the two metrics *performance* and *quality of solution*. Performance is measured in terms of inferences required to obtain the first goal node (as computed by SWI-Prolog's `time/1` predicate). The quality of solution is measured using the perturbations described in Sect. 2.3 as gold standard. Inference numbers are either annotated with "Y" (the gold standard has been reproduced automatically), "C" (the reproduction is close to the gold standard), and "N" (no reproduction).

All three search methods have access to the same arsenal of actions, which includes the action `ShadowClause`. This perturbation acts as a fallback mechanism and ensures that all search terminates with a program mutation whose execution reproduces the learner's erroneous answer to a given subtraction task. If a `ShadowClause` action has been applied, the resulting mutation is task-specific; it usually fails to reproduce a learner's consistent erroneous behaviour across other tasks. The explanatory power of the resulting mutation is rather limited.

In blind search, all perturbation actions have equal cost. Therefore, blind search often yields programs that result from applying `ShadowClause` perturbations. As Table 1 shows, blind search often terminates with less inferences than the other two methods, but at the cost of low-quality solutions. None of the typical errors were reproduced faithfully.

Cost-based search and A^*-search offer a vast improvement to blind-search. Here, `ShadowClause` transformations are only chosen when no other transformations are available. While cost-based search reproduces four of the top-five errors, A^* manages to get all five reproductions right. For `stops-borrow-at-zero`, cost-based search constructs a buggy variant of the expert program by deleting only a single subgoal in the first definition of `decrement/3` (line 35); A^*-search performs three deletions in this clause, effectively rendering it into a null operation. While both variants reproduce the learner error, their dynamics is different: the first program forces `process_column/3` to enter its second clause for processing the middle column, while A^* forces `process_column/3` into its first clause. Clearly, A^* returns a more faithful reproduction of the given error.

In terms of inferences, A^* has equal or better performance than the other two methods, while returning equal or better solutions. The benefit of A^* is dramatic for `smaller-from-larger`, where blind search delivers a low-quality path of length 5, and cost-based search and A^* a high-quality path of length 3.

For the last three errors, we can only obtain solutions that are close to our gold standard. This is due to the current lack of inductive capabilities in the test framework. The perturbations to reproduce the error `stops-borrow-at-zero, diff0-N=N` follow, by and large, the perturbations performed for `stops -borrow-at-zero`. The presence of the error `diff0-N=N`, however, implies that no decrement operation is necessary for the given task 401 − 199. Rather than just making `decrement/3` a null operation (as in `stops-borrow-at-zero`), the perturbations for `stops-borrow-at-zero, diff0-N=N` delete the call to `decrement/3` in `process_column/3` altogether. In addition, two task-specific clauses are added for the `diff0-N=N` case. Similar remarks apply to `borrow_across_zero_diff0_N_eq_N`. We find the performance gain in both of the cases significant.

4 Related Work

4.1 Program Testing

Our research has an interesting link to program testing and the design and reliability of test data [4]. The theory of program testing rests on the *competent programmer hypothesis*, which states that programmers "create programs that are *close* to being correct" [4]. In other words, if a program is buggy, then it differs from the correct program only by a combination of simple errors. Moreover, programmers have a rough idea of the kind of error that are likely to occur, and they have the ability to examine their programs in detail. Program testing is also thought to be aided by the *coupling effect*: test cases that detect simple types of faults are sensitive enough to detect more complex types of faults. The analogy to VanLehn's theory of impasses and repairs is striking. When learners encounter an impasse in executing a correct procedure, they address the impasse by a local repair, which often can be explained in terms of simple errors. Also, teachers have a rough idea of the kind of errors learners are likely to make (and learners might be aware of their repairs, too). Good teachers are able to reconstruct the erroneous procedure a learner is executing, and learners are able to correct their mistakes either themselves or under teacher supervision.

In program testing, the technique of *mutation testing* aims at identifying deficiencies in test suites, and to increase the programmer's confidence in the tests' fault detection power. A mutated variant p' of a program p is created only to evaluate the test suite designed for p on p'. If the behaviour between p and p' on test t is different, then the mutant p' is said to be dead, and the test suite "good enough" wrt. the mutation. If they are equal, then p and p' are equivalent, or the test set is not good enough. In this case, the programs' equivalence must be examined by the programmer; if they are not equivalent, the test suite must be extended to cover the critical test. This relates to our approach. When a given program is unable to reproduce a learner's solution, we create a set of perturbated variants, or mutants. If one of them reproduces the learner's solution, it passes the test, and we are done. Otherwise, we choose the best mutant, given the heuristic function f, and continue with the perturbations. The originality of our approach is due to our systematic search for mutations and the use of f to measure the distance between mutants wrt. a given input/output.

In [6], Kilperäinen & Mannila describe a general method for producing complete sets of test data for simple Prolog programs. Their method is based on the competent programmer hypothesis, and works by mutating list processing programs with a small class of suitable modifications. In [8], the authors give a wide range of mutation operators for Prolog. At the clause level, they have operators for the removal of subgoals, for changing the order of subgoals, and for the insertion, removal, or permutation of cuts. At the operator level, they propose mutations that change one arithmetic or relational operator by another one. Moreover, they propose mutations that act on Prolog variables or constants, e.g., the changing of one variable into another variable, an anonymous variable, or a constant, or the changing of one constant into another one. All mutations are syntactic, and aim at capturing typical programmer errors. So far, our approach makes use of a subset of the aforementioned mutation operators. It is surprising that the top-five bugs, accounting for nearly 50 % of all learner errors, can be explained by learners skipping steps, i.e., mostly in terms of clause deletions.

4.2 Intelligent Tutoring Systems

In the intelligent tutoring community, most system designers follow a rule-based approach to implement *interactive exercises* that help students learn. The ACT* architecture is both theoretical embedding and practical implementation basis for ITSs such as the LISP Tutor or the PAT algebra tutor, see the overview [1]. In the ACT* approach, a set of production rules models the skills to be acquired by the learner. To capture erroneous learner behaviour, expert rules are complemented by buggy rules. The rule engine's step-wise interpretation of the rule system allows the tracing of learner actions in terms of the model. Learner actions are on-path when reproducible by the execution of expert rules, or off-path when explainable in terms of buggy production rules, or when no sequence of rules can be found. Positive and remedial feedback, which is attached to rules, can be generated to support learners' problem solving.

Tutors built upon production rule systems have two major drawbacks: they have high authoring costs, and they need to keep learners close to the correct solution path to tame the combinatorial explosion of the (forward reasoning) rule engine. We focus on the first aspect. While rule-based systems offer an adequate formalism to represent the *logic* of a given domain using a *set* of rules, it seems to be much harder to encode a domain's *control* aspect. The hierarchical aspect of the domain algorithm can only be modeled in terms of goal structures that reside in the rule system's *working memory* and which must be explicitly maintained and manipulated using the rules' pre- and postconditions. The ACT*-like encoding of control creates rules that depend on each other, and hence, makes the authoring and managing of large rule bases a costly undertaking.

In our approach, logic and control are encoded using Prolog, where goal structures are automatically taken care of. Moreover, our tracing of learner actions does not require an *a priori* encoding of buggy rules; buggy program variants are generated on the fly, using a clever variant of algorithmic debugging, which compares expert with learner behaviour, and program transformation techniques that are based on well-defined perturbation operators. In rule-based systems, there is no representation that encodes the difference between an expert and a buggy rule, and hence, little support for modeling learners' repair strategies.

The Icarus cognitive architecture [7] addresses some of the drawbacks of rule-based systems. Inspired by Prolog, it allows rules (*skills*) to explicitly mention sub-skills (i.e., other rules) without making indirect references to them through the working memory. Nevertheless, Icarus retains the overall flavour of a production system by following a recognize-act-cycle. A more radical approach to separate logic (*domain-specific rules*) from control (*strategic guidance*) is proposed by Heeren *et al.* [5]. They separate: (i) information about the domain (e.g., the subtraction matrix and its place-value system), (ii) rules for manipulating expressions in this domain (e.g., performing a complete borrow-payback operation, or taking the difference in a column), (iii) a strategy for solving the exercise (e.g., performing subtraction from right to the left), and (iv) buggy knowledge for modeling both incorrect expression manipulations and incorrect strategies. In their approach, a strategy language is defined that has rules as smallest building blocks, and which controls their combination using rule sequencing, rule choice, and a recursion mechanism. Using the language, a strategy can be defined as a context-free grammar. With the tracing of learner actions reduced to a parsing problem, Heeren *et al.* define a strategy recognizer that is able to compute several types of feedback to support learners when incrementally solving interactive exercises. In this approach, the strategy recognizer should be capable of coping with learner errors that result from strategic (skipping the step) repairs.

5 Conclusion and Future Work

In this paper, we propose a method to automatically transform an initial Prolog program into another program capable of producing a given input/output behaviour. The method depends on a heuristic function that estimates the distance between programs. An experimental evaluation demonstrated the benefits of using heuristic search when compared to the blind (depth-first) search we have used in [12]. It shows that the test-debug-repair cycle can be mechanised in the tutoring context. Here, there is always a reference model to encode ideal behaviour; moreover, many learner errors can be captured and reproduced by a combination of simple, syntactically-driven program transformation actions.

In the near future, we would like to include more mutation operators (see [8]), investigate their interaction with our existing ones, fine-tune the cost function, and study whether erroneous procedures can be obtained that better reflect learners' incorrect reasoning. Ideally, the new operators can be used to "un-employ" the costly `ShadowClause` operator, whose primary purpose is to serve as a fall-back action when all other actions fail. Moreover, we are currently working on a web-based interface for multi-column subtraction tasks that we want to give to learners, and where we plan an evaluation in terms of pedagogical benefits.

In the long term, we would like to take on another domain of instruction to underline the generality of our approach. The domain of learning programming in Prolog is particularly interesting. In the subtraction domain discussed in this paper, we are systematically modifying an expert program into a buggy program to model a learner's erroneous behaviour. In the "learning Prolog domain", we can re-use our program distance measure in a more traditional sense. When learners do specify an executable Prolog program, we compare its behaviour with the prescribed expert program, identify their (dis-)agreement score, and then repair the learner's program, step by step, to become the expert program.

Acknowledgments. The research was funded by the DFG (ZI 1322/2-1).

References

1. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: lessons learned. J. Learn. Sci. **4**(2), 167–207 (1995)
2. Brown, J.S., Burton, R.R.: Diagnostic models for procedural bugs in basic mathematical skills. Cogn. Sci. **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. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**(4), 34–41 (1978)
5. Heeren, B., Jeuring, J., Gerdes, A.: Specifying rewrite strategies for interactive exercises. Math. Comput. Sci. **3**(3), 349–370 (2010)
6. Kilperäinen, P., Mannila, H.: Generation of test cases for simple prolog programs. Acta Cybern. **9**(3), 235–246 (1990)
7. Langley, P., Cummings, K.: Hierarchical skills and cognitive architectures. In: 26th Annual Conference of the Cognitive Science Society, pp. 779–784 (2004)
8. Toaldo, J.R., Vergilio, S.R.: Applying mutation testing in prolog programs. http://www.lbd.dcc.ufmg.br/colecoes/wtf/2006/st2_1.pdf
9. VanLehn, K.: Mind bugs: The Origins of Procedural Misconceptions. MIT Press, Cambridge (1990)
10. Young, R.M., O'Shea, T.: Errors in children's subtraction. Cogn. Sci. **5**(2), 153–177 (1981)
11. Zinn, C.: Algorithmic debugging to support cognitive diagnosis in tutoring systems. In: Bach, J., Edelkamp, S. (eds.) KI 2011. LNCS (LNAI), vol. 7006, pp. 357–368. Springer, Heidelberg (2011)
12. Zinn, C.: Program analysis and manipulation to reproduce learners' erroneous reasoning. In: Albert, E. (ed.) LOPSTR 2012. LNCS, vol. 7844, pp. 228–243. Springer, Heidelberg (2013)