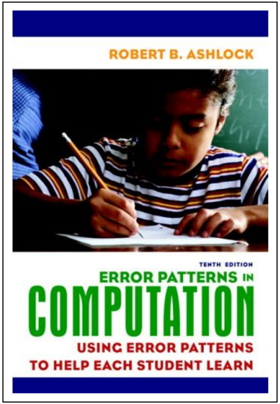
Heuristic Search Over Program Transformations

WFLP-2013
September 13, 2013

Claus Zinn
FB Informatik und Informationswissenschaft
Universität Konstanz
Email: claus.zinn@uni-konstanz.de
WWW: `http://www.inf.uni-konstanz.de/~zinn`
Funded by DFG, Ref. ZI 1322/2-1

Motivation

- application of LP techniques to tutoring
- students may adopt erroneous procedures and misconceptions
- good human instructors can make thoughtful analyses of their students' work and in doing so, discover patterns in errors made
- good human instructors use student error patterns to gain more specific knowledge of students' understanding; this informs their future instruction.
- goal: build program to replicate diagnostic competence of teachers for typical errors, and that can reconstruct learner's erroneous procedure from observation
- use program as part of an ITS for learners, but also for teacher training



Typical Subtraction Errors

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

(a) correct solution

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

(b) smaller-from-larger

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

(c) stops-borrow-at-zero

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

(d) borrow-across-zero

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

(e) borrow-from-zero

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

(f) borrow-no-decrement

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

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

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

(h) always-borrow-left

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

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

Typical Subtraction Errors

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

(a) correct solution

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

(b) smaller-from-larger

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

(c) stops-borrow-at-zero

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

(d) borrow-across-zero

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

(e) borrow-from-zero

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

(f) borrow-no-decrement

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

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

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

(h) always-borrow-left

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

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

- either student has learnt incorrect procedure
- or he knows correct procedure, but cannot execute a step
  - encounters *impasse*, *e.g.*, how to borrow from zero?
  - makes *repair action*, *e.g.*, skip the borrowing step in this case.

Overview

Prior Work

- encoded expert knowledge as Prolog program(s)
- developed variant of algorithmic debugging to localise learner's bug wrt. expert procedure (KI-11)
- perturbated expert program to reproduce learner's erroneous procedure (LOPSTR-12)
  - interative process interleaving algorithmic debugging with program transformation
  - but transformation costly, vast search space conquered (mostly) in blind manner

Overview

Prior Work

- encoded expert knowledge as Prolog program(s)
- developed variant of algorithmic debugging to localise learner's bug wrt. expert procedure (KI-11)
- perturbated expert program to reproduce learner's erroneous procedure (LOPSTR-12)
  - interative process interleaving algorithmic debugging with program transformation
  - but transformation costly, vast search space conquered (mostly) in blind manner

Current Work

- extending algorithmic debugging wrt. (dis-)agreements
- exploiting extension as program similarity metric
- using metric to inform search

Subtraction in Prolog (AM):

	3	2	*minuends*
-	1	7	*subtrahends*
	_	_	*results*

```
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).
```

Subtraction in Prolog (AM):

	3	2	*minuends*
-	1	7	*subtrahends*
	_	_	*results*

```
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.
```

E. Shapiro, Algorithmic Debugging, MIT Press, 1982

- meta-interpreter using divide and conquer to descend computation trees
- *semi-automatic* debugging technique to localise bugs
- based on the answers of an *oracle* – the programmer – to a series of *questions generated automatically* by algorithmic debugger
- answers provide debugger with information about the *correctness of some (sub-)computations* of given program
- uses them to *guide bug search* until portion of code responsible for buggy behaviour is identified: "irreducible disagreement"

E. Shapiro, Algorithmic Debugging, MIT Press, 1982

- meta-interpreter using divide and conquer to descend computation trees
- *semi-automatic* debugging technique to localise bugs
- based on the answers of an *oracle* – the programmer – to a series of *questions generated automatically* by algorithmic debugger
- answers provide debugger with information about the *correctness of some (sub-)computations* of given program
- uses them to *guide bug search* until portion of code responsible for buggy behaviour is identified: “irreducible disagreement”

At first sight not applicable in tutoring context, but

- turning Shapiro’s idea on its head!
- take expert program as buggy program, and oracle answers as student answers
- disagreement between program and oracle identifies learner error
- moreover, oracle can be mechanised, all student answers “read” from solution
- moreover, Oracle also returns nature of disagreement: *missing*, *incorrect*, and *superfluous*

⇒ this variant of algorithmic debugging locates errors in student problem solving

Example

	3	~~2~~ 12
-	1	7
=	2	5

Example

	3	~~2~~ 12
-	1	7
=	2	5

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

Example

$$\begin{array}{ccc} & 3 & \cancel{2}\ 12 \\ - & 1 & 7 \\ \hline = & 2 & 5 \end{array}$$

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

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

Example

$$\begin{array}{cccc} & 3 & \cancel{2}\ 12 \\ - & 1 & 7 \\ \hline = & 2 & 5 \end{array}$$

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

Q: do you agree that the following goal holds:

```
      subtract([ (3,1,_G226), (2,7,_G235)],[ (3,2,1), (12,7,5)])
|: no.
```

Q: do you agree that the following goal holds:

```
      process_column([(3,1,_G475), (2,7,_G484)],[(3,2,_G475), (12,7,5)])
|: no.
```

Example

$$
\begin{array}{ccc}
 & 3 & \cancel{2}\ 12 \\
- & 1 & 7 \\
\hline
= & 2 & 5
\end{array}
$$

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

```
Q: do you agree that the following goal holds:
        subtract([ (3,1,_G226), (2,7,_G235)],[ (3,2,1), (12,7,5)])
   |: no.
Q: do you agree that the following goal holds:
        process_column([(3,1,_G475), (2,7,_G484)],[(3,2,_G475), (12,7,5)])
   |: no.
Q: do you agree that the following goal holds:
        add_ten_to_minuend((2,7,_G652), (12,7,_G652))
   |: yes.
```

Example

	3	~~2~~ 12
-	1	7
=	2	5

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

```
Q: do you agree that the following goal holds:
       subtract([ (3,1,_G226), (2,7,_G235)],[ (3,2,1), (12,7,5)])
   |: no.
Q: do you agree that the following goal holds:
       process_column([(3,1,_G475), (2,7,_G484)],[(3,2,_G475), (12,7,5)])
   |: no.
Q: do you agree that the following goal holds:
       add_ten_to_minuend((2,7,_G652), (12,7,_G652))
   |: yes.
Q: 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))
```

Perturbations

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

Perturbations

```
function ReconstructErroneousProcedure(Program, Problem, Solution)
    (Disagr, Cause) ← AlgorithmicDebugging(Program, Problem, Solution)
    if Disagr = nil then
        return Program
    else
        NewProgram ← 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
```

Example:

	5	2	4
-	2	9	8
=	3	7	4

- First irreducible disagreement at

  `add_ten_to_minuend(3, (4,8,_G808), (14,8,_G808))`

  delete subgoal `add_ten_to_minuend/3` from first clause of `process_column/3`

Example:

	5	2	4
-	2	9	8
=	3	7	4

- First irreducible disagreement at

  `add_ten_to_minuend(3, (4,8,_G808), (14,8,_G808))`

  delete subgoal `add_ten_to_minuend/3` from first clause of `process_column/3`
- Given modified program, next irreducible disagreement (with cause “missing”) at:

  `increment(2, (2,9,_G799), (2,10,_G799))`

  delete subgoal `increment/3` from the first clause of `process_column/3`

Example:

	5	2	4
-	2	9	8
=	3	7	4

- First irreducible disagreement at

  `add_ten_to_minuend(3, (4,8,_G808), (14,8,_G808))`

  delete subgoal `add_ten_to_minuend/3` from first clause of `process_column/3`
- Given modified program, next irreducible disagreement (with cause "missing") at:

  `increment(2, (2,9,_G799), (2,10,_G799))`

  delete subgoal `increment/3` from the first clause of `process_column/3`
- Next irreducible disagreement (with cause "incorrect") at:

  `take_difference(3, (4,8,_G808), (4,8,-4))`

  Mere deletion of `take_difference/3` not possible, other perturbation required
  shadow existing clause with

```
take_difference( 3, (4, 8, _R),  (4, 8, 4)) :- irreducible.
```

Example:

	5	2	4
-	2	9	8
=	3	7	4

- First irreducible disagreement at

  `add_ten_to_minuend(3, (4,8,_G808), (14,8,_G808))`

  delete subgoal `add_ten_to_minuend/3` from first clause of `process_column/3`
- Given modified program, next irreducible disagreement (with cause "missing") at:

  `increment(2, (2,9,_G799), (2,10,_G799))`

  delete subgoal `increment/3` from the first clause of `process_column/3`
- Next irreducible disagreement (with cause "incorrect") at:

  `take_difference(3, (4,8,_G808), (4,8,-4))`

  Mere deletion of `take_difference/3` not possible, other perturbation required
  shadow existing clause with

  ```
  take_difference( 3, (4, 8, _R),  (4, 8, 4)) :- irreducible.
  ```
- Next irreducible disagreement (with cause "incorrect") at:

  `take_difference(2, (2,9,_G808), (2,9,-7))`

  shadow existing clause with

  ```
  take_difference( _CC, (2, 9, _R),  (2, 9, 7)) :- irreducible.
  ```

  ⇒ New program reproduces learner's result.

Heuristics For Program Perturbations

- “often”, algorithmic debugging correctly indicates the clause that required manipulation
- skipping a step can often be reproduced by deleting the respective call to the clause in question in the expert program
- never delete a clause that ran successfully at an earlier problem solving stage $\Rightarrow$ shadow clause with specialised instance (derivable from algorithmic debugging)
- shadowing is also a good heuristics for irreducible disagreements with cause “incorrect” *and as fallback*.

Heuristics For Program Perturbations

- "often", algorithmic debugging correctly indicates the clause that required manipulation
- skipping a step can often be reproduced by deleting the respective call to the clause in question in the expert program
- never delete a clause that ran successfully at an earlier problem solving stage $\Rightarrow$ shadow clause with specialised instance (derivable from algorithmic debugging)
- shadowing is also a good heuristics for irreducible disagreements with cause "incorrect" *and as fallback*.

but better heuristics needed

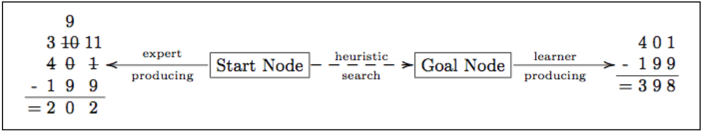
- which action to choose from, *e.g.* in a more search-global context
- same action can be applied at different program locations, *e.g.*, delete one or many subgoals in the clause indicated by algorithmic debugging?
- other mutation operators will be added
- transformation cost should be taken into account

New approach defines program distance measure to inform search...

```
NumberAgreements ← 0, NumberDisagreements ← 0
Problem ← current task to be solved, Solution ← learner input to task
Goal ← top-clause of routine, with input Problem and output Solution
procedure ALGORITHMICDEBUGGING(Goal)
    if Goal is conjunction of goals (Goal1, Goal2) then
        ← algorithmicDebugging(Goal1)
        ← algorithmicDebugging(Goal2)
    end if
    if Goal is system predicate then
        ← call(Goal)
    end if
    if Goal is not on the list of goals to be discussed with learners then
        Body ← getClauseSubgoals(Goal)
        ← algorithmicDebugging(Body)
    end if
    if Goal is on the list of goals to be discussed with learners then
        SystemResult ← call(Goal)
        OracleResult ← oracle(Goal)
        if results agree on Goal then
            Weight ← computeWeight(Goal)    ▷ compute # of skills in proof tree
            NumberAgreements ← NumberAgreements + Weight
        else
            if Goal is leaf node (or marked as irreducible) then
                NumberDisagreements ← NumberDisagreements + 1
            else
                Body ← getClauseSubgoals(Goal)
                ← algorithmicDebugging(Body)
            end if
        end if
    end if
end procedure
Score ← NumberDisagreements − NumberAgreements
```

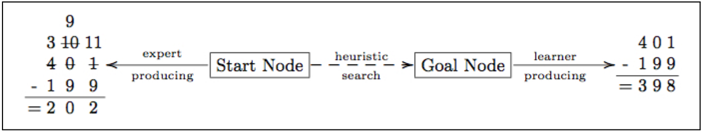
Idea: Consider problems in terms of *heuristic* search

- replace blind-search over program transformations with heuristic search



Idea: Consider problems in terms of *heuristic* search

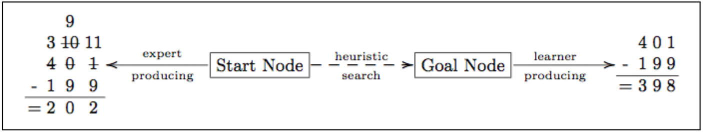
- replace blind-search over program transformations with heuristic search



- each state in the search tree is represented by the tuple
  - `Algorithm`: the program to be transformed
  - `IrreducibleDisagreement`: the first irreducible disagreement with learner behaviour
  - `Path`: a sequence of transformation actions applied so far

Idea: Consider problems in terms of *heuristic* search

- replace blind-search over program transformations with heuristic search



- each state in the search tree is represented by the tuple
  - `Algorithm`: the program to be transformed
  - `IrreducibleDisagreement`: the first irreducible disagreement with learner behaviour
  - `Path`: a sequence of transformation actions applied so far
- each state n has heuristic measure $f(n) = g(n) + h(n)$
  - $g(n)$ is cost function
    - `ShadowClause`: 5 (expensive)
    - `DeleteSubgoalsOfClause`: 1 for each subgoal deleted, extra penalty if applicable.
    - `DeleteCallToClause`: 1
    - `SwapClauseArguments`: 1
  - $h(n)$ measures program distance in terms of agreement score

Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

Error: first irreducible disagreement at `add_ten_to_minuend/3`.

Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

  Error: first irreducible disagreement at `add_ten_to_minuend/3`.

- Scope of action:
  - n_1 deletion of call to `add_ten_to_minuend/3` in first program clause `process_column/3`
  - n_2 addition of irreducible disagreement (learner's view) `add_ten_to_minuend(3,1,1):-irreducible.`
  - n_3 deletion of subgoals from definition of predicate `add_ten_to_minuend/3`: delete subgoal `M10 is M + 10`

Example

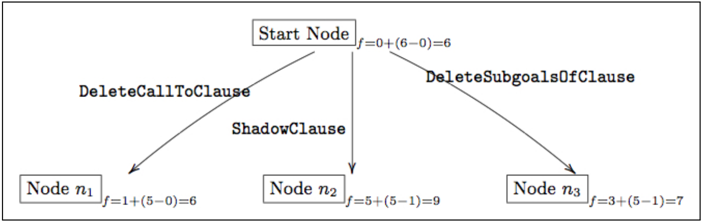
- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

  Error: first irreducible disagreement at `add_ten_to_minuend/3`.

- Scope of action:
  - n_1 deletion of call to `add_ten_to_minuend/3` in first program clause `process_column/3`
  - n_2 addition of irreducible disagreement (learner's view) `add_ten_to_minuend(3,1,1):-irreducible.`
  - n_3 deletion of subgoals from definition of predicate `add_ten_to_minuend/3`: delete subgoal `M10 is M + 10`

- search space:



Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

  Error: first irreducible disagreement at `decrement/3`.

Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

  Error: first irreducible disagreement at `decrement/3`.

- Scope of action:

  n_{11} delete call to `decrement/3` in first clause of `process_column/3`.
  $f(n_{11}) = (1+1) + (2-1) = 3$

  n_{12} delete one/more subgoals in any of the two clause definitions for `decrement/3`, *e.g.*, in first clause:

  - delete subgoals `NM1 is NM-1`, `NM is M+10` and `decrement(CurrentColumn1, RestSum, NewRestSum)`.
    $f(n_{12(a)}) = (1+3) + (4-1) = 7$
  - delete the two goals `NM is M + 10` and `NM1 is NM-1`:
    $f(n_{12(b)}) = (1+2) + 4 - 1 = 6.$
  - delete single goal `NM1 is NM-1`:
    $f(n_{12(c)}) = (1+1) + 5 - 0 = 7$
  - delete recursive call to decrement/3:
    $f(n_{12(d)}) = (1+1) + 3 - 1 = 4$

  n_{13} add disagreement clause to program

  ```
  decrement(2,[(4,1,S1),(0,9,S2)],[(4,1,3),(0,9,9)]):-irreducible.
  ```

  $f(n_{13}) = (1+5) + 4 - 1 = 9.$

Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

  Error: first irreducible disagreement at `decrement/3`.

- Scope of action:

  n_{11} delete call to `decrement/3` in first clause of `process_column/3`.
  $f(n_{11}) = (1+1) + (2-1) = 3$

  n_{12} delete one/more subgoals in any of the two clause definitions for `decrement/3`, *e.g.*, in first clause:

  - delete subgoals `NM1 is NM-1`, `NM is M+10` and `decrement(CurrentColumn1, RestSum, NewRestSum)`.
    $f(n_{12(a)}) = (1+3) + (4-1) = 7$
  - delete the two goals `NM is M + 10` and `NM1 is NM-1`:
    $f(n_{12(b)}) = (1+2) + 4 - 1 = 6.$
  - delete single goal `NM1 is NM-1`:
    $f(n_{12(c)}) = (1+1) + 5 - 0 = 7$
  - delete recursive call to decrement/3:
    $f(n_{12(d)}) = (1+1) + 3 - 1 = 4$

  n_{13} add disagreement clause to program

  ```
  decrement(2,[(4,1,S1),(0,9,S2)],[(4,1,3),(0,9,9)]):-irreducible.
  ```

  $f(n_{13}) = (1+5) + 4 - 1 = 9.$

- n_{11} has lowest overall estimate; since not goal node, continue search...

Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

Error: first irreducible disagreement at `take_difference/4`

Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

  Error: first irreducible disagreement at `take_difference/4`

- scope of action:

  n_{111} delete call to `take_difference/4` in 1st or 2nd clause of `process_column/3`; not fruitful

  n_{112} delete single subgoal in definition of `take_difference/4`; produces incorrect cells.

  n_{113} insert clause `take_difference(3,1,9,8)`; removes disagreement in current column, but not in others

  n_{114} swap arguments of `take_difference/4` in 1st or 2nd clause of `process_column/3`;

⇒ n_{114} yields program with zero disagreements

Example

- `smaller-from-larger`:

	4	0	1
-	1	9	9
=	3	9	8

  Error: first irreducible disagreement at `take_difference/4`

- scope of action:

  n_{111} delete call to `take_difference/4` in 1st or 2nd clause of `process_column/3`; not fruitful

  n_{112} delete single subgoal in definition of `take_difference/4`; produces incorrect cells.

  n_{113} insert clause `take_difference(3,1,9,8);` removes disagreement in current column, but not in others

  n_{114} swap arguments of `take_difference/4` in 1st or 2nd clause of `process_column/3`;

⇒ n_{114} yields program with zero disagreements

- winning path:

  1. deletion of call to clause `add_ten_to_minuend/3` (line 15)
  2. deletion of call to clause `decrement/3` (line 17)
  3. swapping of arguments in `take_difference/4` (line 18)

Discussion

- for top-five bugs, new method capable of reproducing the "preferred" perturbations, using same path
- costlier goal nodes were also found
- $g(n)$ also important, because it discourages use of certain ops
  - method also capable of reproducing programs for the other errors, but with task-specific `ShadowClause` perturbations
  - `ShadowClause` is fall-back and guarantees success

Discussion

- for top-five bugs, new method capable of reproducing the “preferred” perturbations, using same path
- costlier goal nodes were also found
- $g(n)$ also important, because it discourages use of certain ops
  - method also capable of reproducing programs for the other errors, but with task-specific `ShadowClause` perturbations
  - `ShadowClause` is fall-back and guarantees success

- add more mutation operators [Toaldo and Vergilio, 2006],
- investigate their interaction with our existing ones,
- fine-tune cost function, and study effect
- goal to (mostly) “un-employ” the costly `ShadowClause` operator
- need to conduct thorough experimental evaluation in terms of computational benefits of using informed search

Related Work: Program Testing

Program Testing

- rests on *competent programmer hypothesis* (deMillo 1978)
  - programmers create programs that are *close* to being correct
  - if program is buggy, it differs from correct program only by combination of simple errors
  - programmers have rough idea of kinds of errors that are likely to occur, and they are capable of examining their programs in detail (and fix them)
  - *coupling effect*: test cases that detect simple types of faults are sensitive enough to detect more complex types of faults

Related Work: Program Testing

Program Testing

- rests on *competent programmer hypothesis* (deMillo 1978)
  - programmers create programs that are *close* to being correct
  - if program is buggy, it differs from correct program only by combination of simple errors
  - programmers have rough idea of kinds of errors that are likely to occur, and they are capable of examining their programs in detail (and fix them)
  - *coupling effect*: test cases that detect simple types of faults are sensitive enough to detect more complex types of faults
- analogy to VanLehn's theory of impasses and repairs
  - learners exhibit problem solving behaviour that is often close to being correct
  - if their "program" is buggy, it differs from the expert behaviour only by a combination of simple errors
  - teachers have rough idea of the kind of errors learners are likely to make (and learners might be aware of their repairs, too), and learners are capable of correcting their mistakes (either themselves or with teacher support)
  - complex errors can be described in terms of simpler ones

Related Work: Program Testing

Mutation Testing

- identifies test suite deficiencies
- can increase programmer's confidence in the tests' fault detection power
- mutated variant p' of program p created to evaluate test suite designed for p on p'
- if behaviour between p and p' on test t is different, mutant p' is *dead*; test suite "good enough" wrt. mutation
- if behaviours equal, then either p and p' are equivalent, or test set not good enough.

  programmer must examine equivalence; if negative, test suite must be extended to cover the critical test

Related Work: Program Testing

Mutation Testing

- identifies test suite deficiencies
- can increase programmer's confidence in the tests' fault detection power
- mutated variant p' of program p created to evaluate test suite designed for p on p'
- if behaviour between p and p' on test t is different, mutant p' is *dead*; test suite "good enough" wrt. mutation
- if behaviours equal, then either p and p' are equivalent, or test set not good enough.

  programmer must examine equivalence; if negative, test suite must be extended to cover the critical test

Our Approach

- if given program unable to reproduce a learner's solution, we create set of mutants
- if one of them reproduces the learner's solution, it passes test, and we are done
- otherwise, we choose the best mutant, given f, and continue perturbating
- originality due to systematic search for mutations measuring distance between mutants wrt. a given input/output

Conclusion Work

- proposed method to automatically transform initial Prolog program into another program capable of producing a given input/output behaviour
- now supported by heuristic function that measures program distance
- application context where test-debug-repair cycle can be mechanised
  - because of reference model
  - many learner errors can be captured and reproduced by combination of simple, syntactically-driven program transformation actions

Conclusion Work

- proposed method to automatically transform initial Prolog program into another program capable of producing a given input/output behaviour
- now supported by heuristic function that measures program distance
- application context where test-debug-repair cycle can be mechanised
  - because of reference model
  - many learner errors can be captured and reproduced by combination of simple, syntactically-driven program transformation actions

Future Work

- practical employment for pupils and teachers
- revisit mechanisation of Oracle, using program specifications
- adapt other LP techniques to support diagnosis engine
- in the long term, build programming tutor!