Erschienen in: IEEE 14th International Conference on Advanced Learning Technologies : ICALT 2014 ; proceedings / . - Piscataway, NJ : IEEE, 2014. - S. 52-53. - ISBN 978-1-4799-4038-7

A lean constraint-based system to support intelligent tutoring

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

***Abstract*—We present a lean, glass-box engine for constraint-based intelligent tutoring. It has a concise and elegant embedding in Prolog, and offers an easy-to-use but expressive constraint language. It can serve as a formal playground to investigate the nature of constraint-based tutoring.**

***Keywords*-cognitive diagnosis; constraint-based tutoring**

I. Introduction

The intelligent behaviour of a tutoring system (ITS) is usually implemented in one of the two prominent paradigms: production rules [1] or constraints [4]. To help clarify the pending controversy about the nature and potential of constraint-based tutoring [3], [2], we have implemented a lean and inspectable constraint engine in Prolog. Its expressive constraint language supports the specification of constraints to diagnose whether a learner action is correct or not, to give error-specific feedback on an incorrect step, and to generate hints on the next step. We propagate the use of four types of constraints to realize such feedback repertoire, which we demonstrate with an example model.

II. Constraint-based Modeling

A. Cognitive Diagnosis

Constraint-based tutoring diagnoses the correctness of learner input in terms of a *problem state*, given a set of constraints that test whether relevant aspects of the state are satisfied or not. Following [4], a constraint is a pair $< C_r, C_s >$, where C_r is the *relevance condition*, identifying "the class of problem states for which the constraint is relevant", and C_s the *satisfaction condition*, identifying "the class of (relevant) states in which the constraint is satisfied".

The following example constraint encodes the principle that only fractions with equal denominators can be added: *If the problem is $\frac{n_1}{d_1} + \frac{n_2}{d_2}$ and if $n = n_1 + n_2$, then it had better be the case that $d_1 = d_2$ (or else something is wrong).*

Each relevant, unsatisfied constraint signals an error.

B. Domain of Instruction: Adding Fractions

Given two fractions n_1/d_1 and n_2/d_2, compute their sum. When the two input fractions do not share a common denominator, it must be computed. Once the lowest common denominator d for d_1 and d_2 is determined, the numerators must be rewritten in terms of d_1, n_1 and d, and d_2, n_2 and d (yielding d_{11}, d_{22}, n_{11} and n_{22}, with $d_{11} = d_{22} = d$). The converted fractions are then added by adding their numerators. When the resulting fraction n/d is improper, it must be reduced to a proper one: the greatest common divisor g of n and d is computed, and a reduced fraction returned where n and d are both divided by g (yielding n_r and d_r). We capture the problem state as

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} =^c \frac{n_{11}}{d_{11}} + \frac{n_{22}}{d_{22}} = \frac{n}{d} =^r \frac{n_r}{d_r}.$$

C. A Prolog-based Constraint System

We define the problem state as a Prolog fact `current_state/2` with two argument terms: `given/1` encodes the list of givens and `sought/1` encodes a list of values for learners to determine. The fact `problem_context(_N1/_D1+_N2/_D2)` encodes the general task to be solved, here the addition of two arbitrarily given fractions.

Constraints are represented by 5-ary Prolog facts

constraint(*Name*, *State*, *Relevance*, *Satisfaction*, *Feedback*).

The first argument identifies the constraint with a name, for testing and debugging. Its second parameter is used for passing on the current problem state to the constraint. A constraint's third and fourth argument encode the relevance and satisfaction conditions as a Prolog goal structure. The last argument associates a feedback string with a list of state variables affected by the constraint.

```
constraint_engine(Satisfied, UnSatisfied) ←
    current_state(Given, Sought),
    findall( constraint(Name, Sat, Feedback),
             ( constraint(Name,
                          state(Given, Sought),
                          relevance_cond(Rel),
                          satisfaction_cond(Sat),
                          Feedback),
               call(Rel) ), RelConstraints),
    test_constraints( RelConstraints, Satisfied, UnSatisfied ).

test_constraints( [], [], [] ).
test_constraints( [ constraint(Name, Sat, _Fb)|OtherC ],
                  [Name|OtherSatisC], UnSatisC) ←
    call(Sat), !,
    test_constraints( OtherC, OtherSatisC, UnSatisC ).
test_constraints( [ constraint(Name, _Sat, Fb)|OtherC ],
                  SatisC,
                  [(Name, Fb)|OtherUnSatisC]) ←
    test_constraints( OtherC, SatisC, OtherUnSatisC ).
```

Figure 1. A Constraint Engine in Prolog.

A constraint engine examines all constraints, filters out those that are relevant for a given state, and then checks

whether the relevant ones are satisfiable or not. Fig. 1 depicts the full implementation of our constraint engine. The main clause `constraint_engine/2` returns as result all satisfiable relevant constraints and all non-satisfiable relevant constraints. First, the current problem state is retrieved. Then, the all-solutions predicate `findall/3` identifies all constraints that are relevant in the given state. Here, note the passing on of the values `Given` and `Sought` to `constraint/5`, and the use of `call/1` to execute the goal structure `Rel`. The relevant constraints are collected with their names, their satisfaction conditions, and their feedback terms. Then, `test_constraints/3` checks each relevant constraint whether it is satisfied (`call/1` succeeds), or not.

After each learner step, the tutoring system updates the world state, calls the constraint engine, and uses its output of (un-)satisfied constraints to construct the system's response.

D. Four Types of Constraints for Adding Fractions

A *state constraint* specifies certain conditions that must be satisfied by all correct solutions. Only all state constraints taken together test the learner solution for correctness in all relevant aspects. Fig. 2 depicts the previous example constraint. Its second argument is used to establish bindings

```
constraint(check_denom_when_n_equals_n11_and_n22,
    state(given([N1, D1, N2, D2]),
          sought([N11, D11, N22, D22, N, _D, _NR, _DR])),
    relevance_cond((problem_context(N1/D1 + N2/D2),
                    integers([N, N11, N22, D11, D22]),
                    N is N11 + N22)),
    satisfaction_cond((D11 == D22)),
    (['Only add the numerators', N11, 'and', N22,
      'when they share a common denominator'],
     [n, d11, d22])).

constraint(hint_find_lcd,
    state(given([N1, D1, N2, D2]),
          sought([N11, D11, N22, D22, N, D, NR, DR])),
    relevance_cond((problem_context(N1/D1 + N2/D2),
                    vars([N11, D11, N22, D22, N, D, NR, DR])),
    satisfaction_cond((fail)),
    (['Seek common denom of', D1, 'and', D2], [])).

constraint(remedial_sum_reduced_partially_nr,
    state(given([N1, D1, N2, D2]),
          sought([N11, D11, N22, D22, N, D, NR, _DR])),
    relevance_cond((problem_context(N1/D1 + N2/D2),
                    integers([N11, D11, N22, D22, N, D, NR]),
                    gcd(N, D, G), (\+ NR is N / G),
                    get_multiple(NR, N))),
    satisfaction_cond((fail)),
    (['Answer numerator only partially reduced.',
      'The GCD of', N, 'and', D, 'is', G, '.'], [nr])).
```

Figure 2. Three Constraints for Adding Fractions.

for the input variables `N1`, `D1`, `N2` and `D2`, and the output variables `N11`, `D11`, `N22`, `D22`, and `N`. Its relevancy condition specifies that (i) we are in a task context where fractions are being added; (ii) the relevant cells all have been given integer values; and (iii) the value of the n cell equals the sum of adding n_{11} and n_{22}. The constraint's satisfaction condition then checks whether the converted fractions share a common denominator, *i.e.*, whether d_{11} equals d_{22}. The fifth argument is used for constructing a feedback string.

Next-Step Help Constraints help learners that are stuck. All the learner's problem state advances are correct – all relevant state constraints are satisfied – but some values are missing. For learners stuck in the initial state, we add the constraint `hint_find_lcd`. It is relevant if none of the fields has a value. The constraint's feedback gives-away process-related information by eliciting the task's goal structure. The definition of next-step help constraints follows a pattern: their relevancy conditions test whether some `sought` cells are still variable, and they have a single satisfaction condition `fail`, which is bound to fail.

Path constraints (not shown here) check whether learners perform steps in the correct order. They are related to next-step help constraints because they also check for gaps.

Buggy Constraints check whether a problem state is incorrect *in a certain way*; they provide remediation specific to the nature of the error. Consider a state where the learner is only partially reducing a fraction (*e.g.*, the fraction $\frac{16}{24}$ is only reduced to $\frac{8}{12}$ rather than $\frac{2}{3}$). The constraint `remedial_sum_reduced_partially_nr` captures this situation.

Multiple errors will cause the constraint engine to return multiple unsatisfied constraints. When learners commit an erroneous partial solution, unsatisfied state constraints will be joined by unsatisfied next-step help and buggy constraints. Here, the tutoring system may want to attack erroneous behaviour in a configurable manner, *e.g.*, if a buggy constraint "fired", then address the captured learner's misconception first, before addressing next-step help and state constraints. To support such heuristics, we extend our constraint language by a sixth argument to specify the type of the constraint: *state*, *hint*, *path*, and *buggy*.

III. Conclusion

We have built an easy-to-use tutoring engine that supports four types of constraints. We invite researchers to use our software as a formal playground to further investigate the nature of constraint-based tutoring.

References

[1] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier. Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4(2):167–207, 1995.

[2] V. Kodaganallur, R. Weitz, and D. Rosenthal. A comparison of model-tracing and constraint-based intelligent tutoring paradigms. *Int. J. of Artif. Intell. in Educ.*, 15:117–144, 2005.

[3] A. Mitrovic, K. R. Koedinger, and B. Martin. A comparative analysis of cognitive tutoring and constraint-based modeling. In *Proc. of the Ninth Int'l Conf. on User Modeling*, pages 313–322. Springer, 2003.

[4] S. Ohlsson. Constraint-based student modeling. *Int J. of Artif. Intell. in Educ.*, 3(4):429–447, 1992.