MaVeriC – A Constraint-Based System for Web-Based Learning

Claus Zinn

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

Abstract. We present a constraint-based system for web-based learning. Its constraint engine has a concise and elegant embedding in the Prolog programming language, and it offers an easy-to-read and easy-to-write constraint language. We use its glass-box design as a formal playground to investigate the nature of constraint-based tutoring.

Keywords: Constraint-based tutoring, cognitive diagnosis, Prolog.

1 Introduction

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 [8], 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 constraint encodes, for instance, the domain 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).*

Constraint-based tutoring has gained traction [6]. It now competes with the most prominent programming paradigm for building intelligent tutors, the model-tracing tutors that are based on production rule systems [2]. But which approach is better suited to capture, monitor and address learners' (potentially erroneous) problem solving? Does it depend on the domain of instruction to be modeled? How about the costs of authoring and running task models?

In this paper, we aim at contributing to the pending controversy about the nature and potential of constraint-based tutoring [4,3,7]. For this, we have implemented a constraint-based system for basic mathematics. While its outer loop is very simple – learners can define arbitrary fraction addition tasks – it offers a fully-fledged inner loop 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. The system architecture follows the model-view-controller design pattern to enforce a separation of concerns between components. The paper focuses on the model, which offers a simple, concise but powerful constraint engine that is re-usable and runs independently from the other system components.

We use the constraint engine as a playground for complementing *state constraints* with three other types of constraints, which helps clarifying some of the pending issues in constraint-based tutoring.

2 A Constraint System

2.1 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). The problem state can be captured as

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

2.2 The Constraint Engine

Our system to define and process constraints is embedded in Prolog. The problem solving state is defined as a Prolog fact `current_state/2` with two argument terms: `given/1` encodes the givens as a list of values; `sought/1` encodes a list of values for learners to determine to solve a given problem, for instance,

$\mathbf{current\_state}(\ \mathbf{given}(\ [5,7,4,9]), \mathbf{sought}([45,63,28,63,N,D,NR,DR])).$

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 convenience, 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 feedback with the constraint. It consists of a feedback string and a list of state variables affected by the constraint.

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. It fits in 20 lines of Prolog code. The main clause `constraint_engine/2` returns as result all satisfiable relevant clauses and all non-satisfiable relevant clauses. First, the

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

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

Fig. 1. A Constraint Engine in Prolog

current problem state – as represented in the Prolog system – 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` processes relevant constraints. Its first clause tests for termination, and the subsequent definitions deal with the two cases of a constraint being satisfied (`call/1` succeeds) or not satisfied.

The following predicate drives constraint-based diagnosis:

```
get_diagnosis(Task, Solution, Diagnosis) ← set_current_state(Task, Solution),
      constraint_engine(Satisfied, UnSatisfied),
      construct_remedial_feedback(UnSatisfied, Diagnosis).
```

With the problem state set, the engine is called to determine all relevant constraints; the unsatisfied constraints are aggregated to construct the feedback.

2.3 State Constraints for Adding Fractions

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

```
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])).
```

Fig. 2. A State Constraint for Adding Fractions

Prolog input variables `N1`, `D1`, `N2` and `D2`, and the Prolog 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 (the content of the other cells is ignored); 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, and for supporting the GUI to highlight the corresponding cells.

2.4 Other Types of Constraints

An ITS must cope with and support learners that fail to advance a problem state, perform steps in the wrong order, or exhibit errors common to a domain of instruction. For these cases other types of constraints are necessary, see Fig. 3.

```
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␣denominator␣of', D1, 'and', D2], [])).
constraint(path_missing_intermediate,
   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]), integer(N), integer(D),
                  lcd(D1, D2, D), rewrite_numerator(N1, D1, D, N11),
                  rewrite_numerator(N2, D2, D, N22), N is N11 + N22)),
   satisfaction_cond((fail)),
   (['Correct␣result,␣but␣missing␣steps'], [N11, D11, N22, D22])).
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),
                  Fact is N / NR, Fact > 1)),
   satisfaction_cond((fail)),
   (['Answer numerator␣only␣partially␣reduced.', 'The␣GCD␣of', N, 'and', D,
     , 'is', G, '.', 'Divide', N, 'by', G, 'rather␣than', Fact, '.'], [nr])).
```

Fig. 3. Three Other Types of Constraints

Next-Step Help Constraints help learners that are stuck. All the learner's problem state advances are correct, but some values are missing. For learners stuck in the initial state, we add the constraint `hint_find_lcd`. It is only relevant if none of the fields has a value, *i.e.*, when all the terms in the `sought` list are variables. The constraint's feedback component gives-away process-related information. Feedback may include the elicitation of the task's goal structure, *e.g.*, by mentioning the next goal to be tackled, or by decomposing a goal into subgoals. The definition of help constraints follows a pattern: their relevancy conditions test whether some `sought` cells are variable; also, they have a single satisfaction condition `fail`, which is bound to fail.

Path Constraints check whether learners perform steps in the correct order. They are related to next-step help constraints because they also check for gaps. Consider the following world state, where the answer is correct but lacks the intermediate conversion steps: $5/7 + 4/9 =^c \Box/\Box + \Box/\Box = 73/63 =^r \Box/\Box$.

The constraint `path_missing_intermediate` captures this behaviour. Here, relevancy conditions not only check for empty cells, but also ensure that all given values are correct. Again, there is the single satisfaction condition `fail`.

Buggy Constraints address the wrongness of the situation. While the satisfaction conditions for state constraints enforce the correctness of values in the problem space (if a relevant state constraint is unsatisfiable, then *something* in the problem space is incorrect), the relevancy conditions of buggy constraints test whether a problem state is incorrect *in a certain way*; their feedback component provides remediation that is specific to the nature of the error. Consider a problem 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 buggy constraint `remedial_sum_reduced_partially_nr` captures this situation. Its relevancy conditions `gcd(N,D,G), (\+ NR is N/G)` check whether `NR` is *not* the result of dividing `N` by `G`. If `N` is a multiple of `NR`, and if `N` divided by `NR` is greater than 1, then the learner only partially reduced the non-proper fraction.

When learners commit multiple errors, more than a single relevant constraint will be unsatisfied. Moreover, when learners commit an erroneous partial solution, unsatisfied relevant state constraints will be joined by relevant unsatisfied next-step help constraints and buggy constraints. Here, the tutoring system must attack erroneous behaviour in a step-wise manner. Rather than presenting learners with the direct output of the constraint engine, the system selects the next best issue, following this heuristics: if a buggy constraint "fired", then address the captured learner's misconception first, before addressing next-step help, path, and state constraints; if a state constraint "fired", then omit hints resulting from next-step help and path constraints; otherwise give feedback from next-step help and path constraints. – We extend our constraint language by a sixth argument to specify the type of the constraint: *state*, *hint*, *path*, and *buggy*.

3 Discussion

The ASPIRE system is a constraint-based system that also offers an authoring environment for instructional designers [5]. Its architectural design is rather monolithic: its constraint engine, which is not available independently, seems to be tightly interwoven with the user interface as many of the ASPIRE constraints for fractions suggest. As there is no clear separation between the different components, the constraint model cannot be developed and debugged independently from the overall system, which in turn, raises development costs. In contrast, MaVeriC follows a modular design philosophy. Its constraint-based model can be designed, implemented and tested independently from the other system components, using an easy-to-use constraint language, and a standard Prolog shell with its built-in debugging capabilities. Only little Prolog expertise is required. With regard to modularization, we would like to see our constraint engine on par with the JESS production rule engine (see `http://herzberg.ca.sandia.gov`), which is used in the cognitive tutor authoring tools [1].

With regard to the nature of constraint-based tutoring, we propose four different types of constraints for modeling. *State constraints* check whether the learner's solution is correct and whether its parts are in a correct relation with one another. *Next-step constraints* check for an incomplete solution. They identify the missing parts and have feedback that hint learners toward filling the gaps. *Path constraints* capture "jumping to the conclusion" situations with intermediate steps missing; their feedback instructs learners to perform steps in a proper order. *Buggy constraints* check whether a given solution is incorrect in a certain way. They encode typical errors in a domain and give error-specific remediation to learners to correct the errors. – We invite researchers to use our glass-box and highly re-usable constraint engine as a formal playground to make their case and to further investigate the nature of constraint-based tutoring.

References

1. Aleven, V., McLaren, B., Koedinger, K.: Rapid development of computer-based tutors with the cognitive tutor authoring tools (CTAT). In: Proc. of the Int'l Conf. on Artificial Intelligence in Education, pp. 990–990. IOS Press (2005)
2. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: Lessons learned. Journal of the Learning Sciences 4(2), 167–207 (1995)
3. Kodaganallur, V., Weitz, R., Rosenthal, D.: A comparison of model-tracing and constraint-based intelligent tutoring paradigms. Int. J. of Artif. Intell. in Educ. 15, 117–144 (2005)
4. Mitrovic, A., Koedinger, K.R., Martin, B.: A comparative analysis of cognitive tutoring and constraint-based modeling. In: Brusilovsky, P., Corbett, A.T., de Rosis, F. (eds.) UM 2003. LNCS, vol. 2702, pp. 313–322. Springer, Heidelberg (2003)
5. Mitrovic, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., Holland, J., McGuigan, N.: Aspire: An authoring system and deployment environment for constraint-based tutors. Int. J. of Artif. Intell. in Educ. 19(2), 155–188 (2009)
6. Mitrović, A., Mayo, M., Suraweera, P., Martin, B.: Constraint-based tutors: A success story. In: Monostori, L., Váncza, J., Ali, M. (eds.) IEA/AIE 2001. LNCS (LNAI), vol. 2070, pp. 931–940. Springer, Heidelberg (2001)
7. Mitrovic, A., Ohlsson, S.: A critique of Kodaganallur, Weitz and Rosenthal, "A comparison of model-tracing and constraint-based intelligent tutoring paradigms". Int. J. of Artif. Intell. in Educ. 16(3), 277–289 (2006)
8. Ohlsson, S.: Constraint-based student modeling. Int J. of Artif. Intell. in Educ. 3(4), 429–447 (1992)