Automated Reasoning in Generating Exam Sheets for Automated Deduction

Petra Hozzová, Laura Kovács, and Jakob Rath

TU Wien, Austria

Abstract. Amid the COVID-19 pandemic, distance teaching became the default in world-wide higher education, urging teachers and researchers to revise course materials into a more accessible online content for a diverse audience. Probably one of the hardest challenges in this new form of education came with online assessments of course performance, especially organizing and grading online written exams. In this short paper we describe the online assesment setting we organized for our master's level course "Automated Deduction" at TU Wien. The algorithmic and rigorous logical reasoning developed within our course calls for exam sheets focused on problem solving and deductive proofs; as such exam sheets using test grids are not a viable solution for written exams within our course. We report on the automated reasoning framework we developed for generating individual online exam sheets for students enrolled in the course. We believe that the toolchain of automated reasoning tools we have developed for holding online written exams could be beneficial not only for other distance learning platforms, but also to researchers in automated reasoning by providing our community with a large set of randomly generated benchmarks in SAT/SMT solving and first-order theorem proving.

1 Motivation

online exams due to pandemic. we want to avoid collaboration between students during exam (or make it at least a bit harder), so each student gets their own exam sheet. etc...

2 todo

Challenge: problem instances should be different but of similar difficulty to make sure the exam is fair to students.

3 Method 1: Varying Templates and Fixed Patterns

3.1 Satisfiability Modulo Theories (SMT)

relatively easy: provided template, do small random perturbations that don't change the solution

3.2 Ground Superposition

todo

4 Method 2: Full Random Generation

more sophisticated: full random generation, filter out "too hard"/"too easy" instances. Note that we don't need very efficient implementation of filters since the instances are very small. so we can use naive satisfiability tests or model counting.

4.1 Boolean Satisfiability (SAT)

Example 1 (SAT). Hello Consider the formula:

$$(r \wedge \neg(q \to p)) \vee (q \leftrightarrow \neg(p \to q))$$

(a) Which atoms are pure in the above formula?
(b) Compute a clausal normal form C of the above formula by applying the CNF transformation algorithm with naming and optimization based on polarities of subformulas;
(c) Decide the satisfiability of the computed CNF formula C by applying the DPLL method to C. If C is satisfiable, give an interpretation which satisfies it. □

describe filters, and why they were chosen (aim for a challenging problem, but still solvable by hand).

problems: when restricting too much, the resulting formulas may end up boring. e.g. SAT formula with exactly one model, or no model

4.2 Non-Ground Superposition and Redundancy

todo

5 Implementation

Various programs/scripts tied together by a control script. Running the control script generates all problems and puts them together, resulting in n different exams in PDF format.

Latex template: leave holes for formulas/signatures that will be inserted using the command "\input".

Problem templates: (todo: describe impl of smt and groundsup)

For the full random generation of the SAT and redundancy problems, we were initially inspired by Haskell's QuickCheck library. We used QuickCheck for the first prototype, but realized that we need backtracking in the generator.

cite? Claessen, Koen and Hughes, John (2000). "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" (PDF). Proceedings of the International Conference on Functional Programming (ICFP), ACM SIGPLAN.

Added mtl-style typeclass 'MonadChoose' with a primitive operation 'choose' for choosing an element from a list of choices. Our generator implementations are generic over the monad, constrained by this typeclass.

We used two concrete implementations to evaluate generators: 1. 'RandomChoiceT', a monad transformer that implements 'choose' as uniform random choice with backtracking support (can be thought of as 'ListT' where 'choose' is like the normal monadic bind but shuffles the list with a random permutation first). This is what we actually use to generate random exams. 2. a standard list monad (or similar) to enumerate the sample space. This second evaluation method lets us ensure that the sample space is sufficiently large.

6 Conclusion