Automated Exercises for Constraint Programming

Johannes Waldmann (HTWK Leipzig)

`http://www.imn.htwk-leipzig.de/~waldmann/talk/14/wlp/auto/`

September 16, 2014

Automated Exercises - Why?

- lecture without homework exercises is useless
- homework that is not discussed/graded is useless
- no money to pay teaching assistants for grading
- not enough time to discuss everything in class
- automated and real-time grading helps student to understand the basics
- and frees class time for more interesting discussion

(Declarative) Constr. Programming

- constraint *program* (w.r.t. structure S) is formula F in predicate logic
- constraint *solver* answers the question $F \in \text{Theory}(S)$
- in particular, if F is of shape $\exists x_1, \ldots, x_n : M$, by giving a satisfying assignment

aspects for teaching:

- (syntax and semantics of predicate logic)
- model application problems by constraints
- explain how solvers work

Kroening, Strichman: *Decision Procedures* (Springer, 2008)

Exercise for Propositional SAT

(to show the general idea in a very straigthforward case)

- exercise
  - instance: satisfiable formula F (in CNF)
  - solution: a satisfying assignment
- generator
  - will produce random satisfiable F
  - with given number of variables and clauses

doing the exercise, student will

- (learn semantics of propositional formulas)
- appreciate the “hard work” that the SAT solvers do

UNSAT proofs by resolution

exercise:

- instance: unsatisfiable F in CNF
- solution: a resolution proof of the empty clause

actually,

- proof is DAG, represented as list of nodes
- root nodes are clauses from F
- each internal node is resolution step

A general model for automated exercises

doing the exercise, student has to make choices
general description (analogy)

- exercise: non-deterministic algorithm
- student: acts as oracle
- automated grader: acts as verifier

how does this fit the teaching objetives?

- if the subject is NP, then very well (obviously)
- if the subject is a deterministic algorithm (as used in constraint solvers), then what?

Exercises for Decision Methods

- present the (invariants of the) algorithm via inference rules,
  as in (e.g.) Apt: Principles of Constraint Programming
- most often these rules are non-deterministic in a natural way
  this allows to apply our exercise model
- concrete algorithm corresponds to specific strategy in rule applications
- strategy is ignored in verifying the solutions
- but can be enforced implicitly (using wrong strategy takes too many steps)

Exercise for solving FD constraints

via tree search, state is given by stack of domain assignments (variable $\mapsto$ subset of domain)

- Decide: for variable, pick value, push others
- Solved, Backtrack, Inconsistent

$$\frac{x \in D}{x = a \mid x \in D \setminus \{a\}} \text{ for } a \in D$$

```
Stack [listToFM [(x, [0,1,2,3]), (y, [0,3])] ]

    == Decide x 1 ==>

Stack [listToFM [(x, [  1    ]), (y, [0,3])]
      ,listToFM [(x, [0,  2,3]), (y, [0,3])] ]
```

Ex. for FD: Arc Consistency

```
( P , mkSet
  [ [ 0, 0, 0 ], [ 0, 1, 1 ], [ 0, 2, 2 ]
  , [ 0, 3, 3 ], [ 1, 0, 1 ], [ 1, 1, 2 ]
  , [ 1, 2, 3 ], [ 2, 0, 2 ], [ 2, 1, 3 ]
  , [ 3, 0, 3 ] ] )

current : Stack [ listToFM [ ( x, [ 0 ] )
                , ( y, [ 0, 1, 2, 3 ] ) ]]
step : Arc_Consistency_Deduction
        { atoms = [ P ( x, x, y ) ]
       , variable = y, restrict_to = [ 1 ] }
these elements cannot be excluded
  from the domain of the variable, because the
  given assignment is a model for the atoms:
    [ ( 0, listToFM [ ( x, 0 ), ( y, 0 ) ] ) ]
```

FD constraints (Exercise design)

if constraint is unsat, then . . .

- student has to produce a full search tree
- could be done by Decide/Backtrack only, but is impractical
- enforces the usage of arc consistency deductions

if constraint is sat, then . . .

- student could guess a solution
- and then just enter the corresponding Decide-steps (and avoid arc consistency considerations)
- Decide must always uses lowest value

Exercise for DPLL with CDCL

Davis, Putnam, Logeman, Loveland, solves SAT

plain DPLL: just like FD tree search,
unit propagation $\approx$ 1-consistency.

Conflict Driven Clause Learning, Backjumping
in case of conflict:

- learn a new clause R (the conflict “reason”, must be inferrable from clauses used to obtain current assignment)
- jump back (and use R for unit propagation)

student choices: what to learn, where to jump

DPLL Exercise Generator

naive approach:

- since DPLL is complete method, it can be applied to *any* formula
- drawback: solutions (proof traces) vary greatly in length

fair approach:

- generate formula
- find (shortest) proof trace (implement backtracking solver)
- choose formula where proof trace length is reasonable
- drawback: source code contains solver, students may exploit this

SAT and DPLL *modulo Theory*

Syntax: F in CNF where clauses may contain

- Boolean literals and
- theory literals, e.g., $\neg(2x + 3 > 4y)$

state of search given by partial assigment ($=$ set of literals) σ

two kinds of conflicts:

- Boolean conflict (F contains clause where all literals are false in σ)
- Theory conflict (theory literals from σ are inconsistent)

example:

- Theory of linear inequalities (over $\mathbb{Q}$)
- Solver: Fourier-Motzkin

Conclusion, Discussion

- exercises for constraint programming
- automated generation of instances, grading of solutions
- use exercises (anonymously) at `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=199`
- make our own autotool installation (run it in a VM, `https://github.com/marcellussiegburg/autobuildtool`)