Finite-Domain Constraint Solver

Kyle Dewey and Ben Hardekopf

1 Introduction

The constraint solver maintains a set of symbolic constraints on variables (note that 'variables' to the constraint solver are 'symbolic placeholders' to the `mini-prolog-fd` interpreter). It has an API that allows the `mini-prolog-fd` interpreter to insert a new constraint and to query for satisfying values. The remaining sections of this document describe: the constraint language used by the solver (Section 2); the data structures used to represent the constraints (Section 3); the API to the solver (Section 4); and the algorithm for determining satisfiability (Section 5).

2 Constraint Language

The constraint solver understands constraints from the following grammar:

$$x \in \textit{Variable} \qquad n \in \mathbb{Z}$$
$$c \in \textit{Constraint} ::= x \bowtie n \mid n \bowtie x \mid x_1 \bowtie x_2$$
$$\bowtie \in \textit{RelationalOp} ::= < \mid \leq \mid > \mid \geq \mid = \mid \neq$$

Note that this grammar is specific to the solver; in particular, the variables here correspond to symbolic placeholders in the `mini-prolog-fd` interpreter and the relational operators here correspond to the symbolic relational operators in the `mini-prolog-fd` interpreter.

3 Data Structures

An *interval* is a pair of numbers (n_1, n_2) such that $n_1 \leq n_2$. The possible values of a variable will be represented as a set of intervals (essentially, a compressed form of the set of all possible values for that variable). The solver will consist of two main data structures:

- `values` : maps each variable to a set of intervals representing the possible values of that variable.
- `constraints` : maps each variable to a set of constraints involving that variable.

As explained in Section 4, `contraints` will only map to constraints involving two variables; constraints involving one variable and one number will be represented by restricting the appropriate set of intervals in `values`.

4 Solver API

This section explains the behavior of each API method.

4.1 Insert New Constraint

The `mini-prolog-fd` interpreter will attempt to add constraints to the constraint store. Whenever a new constraint is added, the solver creates a new instance of itself that incorporates that constraint. Before returning the new instance, it must determine whether it is satisfiable. If it is, then the new instance is returned; otherwise, `None` is returned.

If the new constraint involves one variable x and one number n, then the solver computes the possible range of values for x based on the new constraint and uses `propagate` (described in Section 5) to determine if that range of values is consistent. If so, the new solver instance uses the mapping computed by `propagate` for its new `values`.

If the new constraint involves two variables x_1 and x_2, then the new solver instance adds the constraint to `constraints` and determines satisfiability using the `solve(values, [x1,x2])` function (also described in Section 5).

Caveat. If a new constraint is added that (1) involves only variables and not numbers, and (2) both variables involved in the constraint currently have a single satisfying solution, then instead of inserting the constraint and determining satisfiability the solver just tests whether the current satisfying values also satisfy this new constraint. If so then the current solver is returned (without adding the new constraint, because it is already satisfied); if not then `None` is returned.

4.2 Query for Satisfying Values

When the `mini-prolog-fd` interpreter is executing an **fd_labeling**(x) expression, it needs to query the constraint solver for satisfying values for a set of variables. To answer the query for a list of variables `xs`, the solver calls `solve(values, xs)`, where `solve` is described in Section 5. The solver returns a mapping from variables to satisfying values based on the result from `solve`.

5 SAT Algorithm

This code is used to find satisfying values for all of the variables in the given list `xs` (if possible). If we want satisfying solutions for all of the variables in the constraint store, we would call `solve()` passing in `values` and a list of all the variables in the domain of `values`. It will either return `None`, meaning the constraints are unsatisfiable, or `Some(cv)` where `cv` maps each variable in `xs` to a single number.

Solve. The main satisfiability solver function. Takes a mapping from variables to possible values and a list of variables of interest that it will attempt to find satisfying solutions for.

```
define solve(currValues, xs) {
  let x = xs.head, tl = xs.tail
  for each value v in currValues(x):
    let mutableCV be a mutable copy of currValues
    propagate(mutableCV, x, [(v,v)])
    if mutableCV is not empty:
      let currValuesN be an immutable copy of mutableCV
      if tl is not empty then:
        match solve(currValuesN, tl) with one of the following:
          case None ⇒ continue to next iteration
          case Some(cv) ⇒ return Some(cv)
      else return Some(currValuesN)
  end for
  return None
}
```

Propagate. The constraint propagator. Takes a mutable mapping from variables to possible values, a variable of interest, and a set of intervals representing possible values for that variable. Propagates the new information to determine whether those values are consistent with all of the constraints.

```
define propagate(mutableCV, x, ranges) {
  update mutableCV(x) to be the intersection of mutableCV(x) and ranges
  if mutableCV(x) is now empty, make mutableCV empty
  else if mutableCV(x) changed:
    for each constraint c in constraints(x):
      let x2 be the other variable in c besides x
      let r2 = narrow(x2, mutableCV(x2), c, mutableCV(x))
      propagate(mutableCV, x2, r2)
}
```

Narrow. Narrowing variable values. Takes a variable being narrowed x_1, the current possible values for that variable `rangesX1`, a constraint `c` that will determine the type of narrowing, and a set of possible values `rangesX2` for the second variable involved in the constraint. Returns a new range of values for x_1.

```
define narrow(x1, rangesX1, c, rangesX2) {
  match c with one of the following:
    case x1 = x2 | x2 = x1 ⇒
      return rangesX1 intersect rangesX2

    case x1 ≠ x2 | x2 ≠ x1 ⇒
      if rangesX2 is a singleton number n, return rangesX1 - n
      else return rangesX1

    case x1 < x2 | x2 > x1 ⇒
      return rangesX1 - any value v ≥ the maximum value in rangesX2

    case x1 ≤ x2 | x2 ≥ x1 ⇒
      return rangesX1 - any value v > the maximum value in rangesX2

    case x1 > x2 | x2 < x1 ⇒
      return rangesX1 - any value v ≤ the minimum value in rangesX2

    case x1 ≥ x2 | x2 ≤ x1 ⇒
      return rangesX1 - any value v < the minimum value in rangesX2
}
```