Constraint Logic Programming

1 Introduction

Constraint logic programming (CLP) is an extension of logic programming that bakes in additional logic theories as language primitives. Examples of these theories include: finite domains, integer linear arithmetic, real arithmetic, etc. Recall that we made logic programming (LP) tractable by restricting ourselves to a fragment of first-order logic; CLP is a way to regain some expressive power. In practical terms, CLP is LP plus the ability to symbolically reason about numeric constraints (we could also add theories about other domains, such as strings, sets, and bit-vectors, but most CLP languages only use numeric domains). Another way to think of "regular" LP is as a kind of CLP where the only theories are uninterpreted functions (i.e., numbers and ground terms) and equality (i.e., unification); "real" CLP then adds additional theories to those two. We can think of a CLP engine as two pieces: the LP engine plus a *constraint store* that reasons about these additional theories. The LP engine works much like we've already seen in `miniprolog` but slightly modified to interact with the constraint store, which acts as a black box *constraint solver* for dealing with constraints involving the additional theories.

As a simple example to make these ideas more concrete, consider the following scheduling problem: we want to assign seven meetings to days of the week with the following constraints: (1) certain meetings can't be on the same day; and (2) one particular meeting has to come after all other meetings. We will use CLP with the finite domains theory. We can encode this problem by using one variable for each meeting (we'll call them x_1 through x_7) and the finite domain of values for those variables is the integer interval [0..4] (where 0 stands for Monday, 1 for Tuesday, ..., 4 for Friday). If any two meetings x_a and x_b cannot be on the same day, then we add the constraint $x_a \neq x_b$. If meeting x_1 must be the last meeting, then we add the constraints $x_1 > x_2$, $x_1 > x_3$, ..., $x_1 > x_7$. Finally, we ask the CLP engine to compute a satisfying solution; the resulting values of $x_1, \ldots, x_7$ represents a valid schedule (and if there are no resulting values because the problem was unsatisfiable, then there is no valid schedule). In this case, the entire problem is solved by the constraint store; in more complex problems there would be interactions between the LP part of the engine and the constraint store part of the engine.

There are certain characteristics that make a problem particularly suited to solving using CLP. While any Turing-complete programming language can solve any problem that CLP can solve, problems that have these characteristics tend to have relatively simple, elegant, and efficient solutions in CLP and to have complex, messy, and inefficient solutions in other languages. The characteristics are:

- No general, efficient algorithms exist (e.g., the problem is NP-complete). Enumerating all solutions is impractical, instead we need to use some form of efficient search.
- The problem specification has a dynamic component: it should be easy to change programs rapidly to adapt.
- The problem requires some sort of decision procedure. These decision procedures can often be encoded in mathematical formulae and handled by special-purpose solvers.

Examples of problems that have these characteristics include planning, scheduling, resource allocation, logistics, circuit design and verification, finite state machines, financial decision making, transportation, spatial databases, etc.

2 Finite Domains

There is a huge body of work on efficiently implementing various theories. For this class, we will restrict ourselves to a fairly naïve implementation of finite domains. In this theory we add to the language a finite set of possible values that can be assigned to variables, usually represented as integers (of course, these integers can stand for elements of any finite set). The basic problem is this: we are **given** (1) a set of variables over some finite domain (in our case, integers within some bounded range); and (2) a set of constraints on those variables (in our case, the constraints are equalities, disequalities, and inequalities). Our **goal** is to find values from that domain for all of the variables such that all of the constraints are

satisfied. Here we will discuss at a high level the basic ideas of implementing a constraint solver for finite domains. Later handouts will discuss specifics of implementing the constraint solver and integrating it with the LP engine to arrive at the final CLP engine.

2.1 First Try (Intractable)

The simplest approach to this problem is also the worst in terms of performance. We can simply try all possible assignments of values to variables until we find an assignment that satisfies the constraints. In pseudocode, it would look like this (where the variables in question are x_1 through x_n):

```
for each possible value of x_1
  for each possible value of x_2
    ⋮
    for each possible value of x_n
      if the values of x_1...x_n satisfy the constraints, we're done
      else continue
```

This brute-force approach is exponentially expensive with respect to the number of variables we need to find values for. We need to find a better method.

2.2 Second Try (Better)

If we think about the above approach for a bit, we can see that we're doing a lot of wasted work. In particular, if we've made a choice for variable x_k that violates the constraints, then no matter what values we choose for $x_{k+1} \ldots x_n$ the constraints are still going to be violated. Consider the example from Section 1. We would start with assigning the value 0 (i.e., Monday) to variable x_1; however, the constraints specify that x_1 must be the last meeting, i.e., that all of the other variables must be less than x_1. No matter what values we choose for $x_2 \ldots x_7$, the constraints will be violated. Thus, everything we do in the nested loops that iterate through all possible values of $x_2 \ldots x_7$ is wasted work. The key optimization is to detect bad choices as early as possible, in order to avoid doing all of that wasted work. Our solution is to sanity check variables *as* we assign them values, rather than waiting until *after* we've assigned all of the variables values and then checking them at the very end:

```
for each possible value of x_1
  for each possible value of x_2 consistent with x_1
    ⋮
    for each possible value of x_n consistent with x_1...x_{n-1}
      success, return the solution
```

In the example from Section 1, after choosing the value 0 for x_1 we would iterate through all of the possible values for x_2 trying to find one that is consistent with $x_1 = 0$ (i.e., that doesn't violate any of the constraints). Since there is no such value, we'll give up and move on to the next possible value for x_1 instead of wasting time iterating through all possible values of the rest of the variables.

2.3 Final Version (Even Better)

The optimized version still has problems. Consider again the example from Section 1, but suppose we changed the constraints to state only that $x_1 > x_7$ rather than stating the x_1 must be greater than *all* of the variables. Then the optimized algorithm would still iterate through all possible values of $x_2 \ldots x_6$ before giving up, again doing a lot of wasted work. The problem is that we only do sanity checks against our *past* choices, not our *future* choices. We need to be able to immediately figure out that the choice $x_1 = 0$ is inconsistent with any possible choice for x_7, thus avoiding that wasted work. The solution is called *constraint propagation*. At the moment we choose a value for some variable x_k, we look at all of the relevant constraints that mention x_k and consider what effect that choice has on the other variables involved in those constraints. In particular, we can cross of possible values for those variables that are inconsistent with the choice we made for x_k (restricting the set of possible values for a variable is called *narrowing*). If any of the variables are narrowed to the point that there are no possible values left, then we know that we made a bad choice for x_k. In pseudocode:

```
for each variable x, initialize values(x) to the set of all possible values
for each v ∈ values(x_1)
  narrow values(x_1) to v
  for each other variable x_k, narrow values(x_k) based on the new values(x_1)
  if no variable was narrowed to the empty set then
    for each v ∈ values(x_2)
      narrow values(x_2) to v
      for each other variable x_k, narrow values(x_k) based on the new values(x_2)
      if no variable was narrowed to the empty set then
        ⋮
        for each v ∈ values(x_n)
        narrow values(x_n) to v
        for each other variable x_k, narrow values(x_k) based on the new values(x_n)
        if no variable was narrowed to the empty set then
          success, return the solution
        else undo all changes from this iteration and continue
      else undo all changes from this iteration and continue
  else undo all changes from this iteration and continue
return failure
```

Notice that we're effectively describing a search procedure, much like the one that you're implementing for `miniprolog`. There are some similarities between the implementation of this constraint solver and the implementation of `miniprolog`. We will see these in more detail in later handouts that give specifics on the constraint solver implementation.