Notes on Satisfiability-Based Problem Solving

Introduction

David Mitchell
mitchell@cs.sfu.ca
February 5, 2020

This is a preliminary draft. Use freely, but please do not re-distribute publicly without permission. Corrections and suggestions are welcome.

1 Introduction

The subject of these notes is an approach to declaratively specifying and solving computational search and optimization problems. A "computational search problem" here is defined by a collection of instances (inputs), and some collection of solutions (outputs) for each instance, with the relationship between instances and solutions given in mathematically precise terms. For example, a latin square of order n (for n a natural number) is an n by n matrix with elements from $\{1, 2, \ldots, n\}$, with the property that no element appears twice in any row or any column. Constructing latin squares is easy, but Latin Square Completion problem is not. This problem is defined by:

> Instance: $n \times n$ matrix A, with elements from $\{\perp, 1, 2, \ldots, n\}$,
> Solution: An order-n latin square B, such that $B[i, j] = A[i, j]$ for every i, j for which $A[i, j] \neq \perp$.

More practical examples include problems like timetabling, crew rostering, finding bugs in circuits or computer programs, etc.

By "declarative", we mean that to solve a problem you describe the properties of a solution, not a method for obtaining one. To solve an instance of the problem, we pass the instance together with the declarative problem specification to a "solver". The general problem solving scheme is illustrated in Figure 1.

A variety of communities have envisioned and constructed solvers for declarative problem solving, and there is no uniform terminology for the field. The general approach is called, in some places, "constraint-based problem solving", because the properties of a solution are expressed as a collection of constraints. We call the approach "Satisfiability-based Problem Solving", which emphasizes the fact that problem solving amounts to

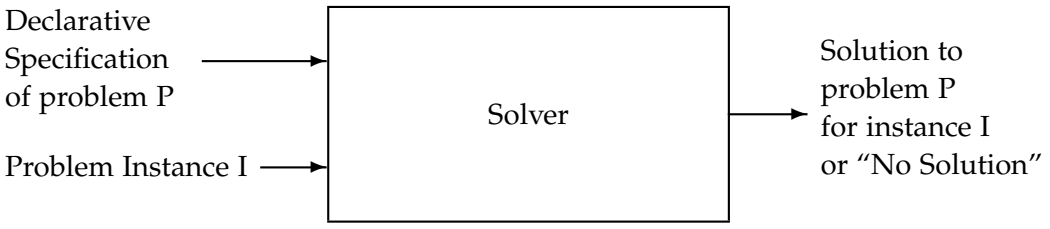

Figure 1: General Scheme for Specification-Based Solving

finding an object that satisfies those constraints. It also reflects our emphasis on formalization in mathematical logic. While practical declarative specification languages may not look a lot like formulas in a logic textbook, in effect that is primarily what they are.

We will be interested in the following very general questions:

1. What sorts of problems are candidates for this approach to problem solving?
2. What methods can we use to construct effective problem solving technology?
3. What is a suitable formalization of the relevant problems and methods?
4. What can we say about the power and limitations of the technology and underlying formalization, based on either theoretical analysis or empirical study.

Our main emphasis will be on an underlying formalization based on mathematical logic, and understanding key elements of current technology in terms of this formalization.

2 Formalization of Specification-Based Solving

We want a formalization of this scheme as an aid to analyzing the languages and systems involved, to have a precise notion of what it means for such a system to be correct, etc. We begin with decision problems: problems for which the "solution" is just a "yes" or "no" answer. The decision version of Latin Square Completion is:

> Instance: $n \times n$ matrix A, with elements from $\{\perp, 1, 2, \ldots, n\}$,
> Question: Is there an order-n latin square B, such that $B[i,j] = A[i,j]$ for every i, j for which $A[i,j] \neq \perp$.

To formalize this manner of specifying problems, we begin with a suitably generic notion of a "problem instance". An instance of Latin Square Completion consists of a matrix, with elements from a finite set. A matrix is just a finite function of a certain sort, which in turn is a finite relation of a certain sort.

Consider next the problem of Graph Colouring:

> Instance: A Graph $G = (V, E)$ and set K of "colours",
> Question: Is there a colouring of elements of V with colours from K such that no edge $(u, v) \in E$ has u and v with the same colour.

An instance of Graph Colouring consists of two finite sets (V and K) and a binary relation E over V.

The following is actual solver input in the language Essence, describing a particular instance of graph colouring.

```
letting V be new type enum {v1,v2,v3,v4}
letting K be new type enum {R,B,G}
letting E be rel {(v1,v3),(v1,v2),(v3,v4),(v4,v1)}
```

We will not go into language details here, but it is still fairly clear what this code does. It defines for the solver an instance of graph colouring, for a graph with four vertices, four edges, and three colours.

In general, the instances of a computational problem consist of a finite collection of finite relations over some finite sets. Such an object is called a finite structure. So, every decision problem P corresponds to a collection of finite structures – those with the desired property.

For a specification to describe this collection of structures, we need a suitable vocabulary, that is, we need names for each of the relations in the structures of interest. A bit more formally, a (finite) vocabulary τ is a tuple of symbols. Each symbol is a relation symbol or a function symbol, and has a specific arity. Relation and function symbols are used as names for relations and functions in our specifications. Function symbols of arity 0 are called constant symbols. For simplicity, we will talk mostly about relational vocabularies, which have no function symbols.

A structure $\mathcal{A}$ for a relational vocabulary $\tau = (R_1, \ldots, R_k)$ is a tuple $\mathcal{A} = (A, R_1^{\mathcal{A}}, \ldots R_k^{\mathcal{A}})$. Here, A is a set called the domain or universe of $\mathcal{A}$, and for each relation symbol R_i of arity r, $R_i^{\mathcal{A}} \subset A^r$ is a k-ary relation over A, called the interpretation or denotation of R_i in $\mathcal{A}$. We will be concerned mainly with finite structures, i.e., where A (and thus each relation also) is finite. We will write $\mathrm{Struc}[\tau]$ for the class of all finite τ-structures.

Now, generically, a problem definition for a decision problem P looks like this, for some particular vocabulary τ:

> Instance: Finite τ-structure $\mathcal{A}$,
> Question: Does $\mathcal{A}$ have property P.

So, problem P corresponds to a collection of finite structures for some vocabulary τ that have the desired property. The job of a specification for P is to precisely define this collection of structures. Here is an Essence specification of graph colouring:

```
given V: new type enum
given K: new type enum
given E: rel of (V x V)
find Col: rel of (V x K)
such that
    forall u:V exists c:K . Col(u,c)
such that
    forall u:V forall c1, c1:K . ((Col(u,c1) /\ (Col(u,c1)) -> c1=c2))
such that
    forall u, v:V forall c:K . (Edge(u,v) ->  !(Col(u,c) and Col(v,c)))
```

The first three lines tell tell us the vocabulary of an structure that is an instance. The fourth line tells the vocaulary for the solution. The remaining lines describe the properties that a solution must have: 1. Each vertex must get a colour; 2. No vertex gets two colours; 3. No edge has the same colour on both ends.

As with languages for other purposes, we would like our specification languages to be as human-readable as possible, and to have features that make specifying a variety of problems as natural as possible. More precise criteria include the following:

1. They define classes of structures;
2. They have a formal semantics, which allows us to reason precisely about correctness.
3. They have suitable expressive power.

The mathematical study of languages that define classes of structures is a part of mathematical logic called model theory. This suggests that our formal notion of specification could be formulas in some appropriate logic. Conveniently, using this formal approach does give us specification languages with these three properties.

If we now think of specifying a problem as, essentially, writing a formula ϕ in some suitable logic that is true on exactly the structures the with the desired property P, we

then we can generically write a formal definition as:

> Instance: τ-structure $\mathcal{A}$,
> Question: Does $\mathcal{A}$ satisfy ϕ?.

2.1 Search Problems

Just as a problem instance, generically, is a finite structure for some vocabulary, a solution for an instance is a finite structure for some other vocabulary. So formally, a search problem P is a binary relation $P \subset \text{Struc}[\tau] \times \text{Struc}[\sigma]$. The solving task is: Given a τ-structure $\mathcal{A}$, find a σ-structure $\mathcal{B}$ such that $(\mathcal{A}, \mathcal{B}) \in P$.

The structures $\mathcal{A}, \mathcal{B}$ of course are not arbitrary. In particular, in order to describe how $\mathcal{B}$ is related to $\mathcal{A}$ so as to constitute a solution, there must be some association between domain elements of $\mathcal{B}$ and those of $\mathcal{A}$.

Much of the time, we will make the simplifying assumption that in fact the domain of $\mathcal{B}$ is actually the same as that of $\mathcal{A}$: $B = A$. While this assumption would be a significant impediment in practical problem solving, it makes for a much simpler formalization. Under this simplifying assumption, then, a search problem P is formalized as

$$P \subset \{(\mathcal{A}, \mathcal{B}) | \mathcal{A} \in \text{Struc}[\tau],\ \mathcal{B} \in \text{Struc}[\sigma],\ A = B\}.$$

Now, a definition of search problem P looks like this:

> Instance: τ-structure $\mathcal{A} = (A, \tau^{\mathcal{A}})$,
> Find: σ-structure $\mathcal{B} = (B, \tau^{\mathcal{B}})$ such that $(\mathcal{A}, \mathcal{B}) \in P$.

But notice that, because $A = B$, we have that $\mathcal{B} = (B, \tau^{\mathcal{B}}) = (A, \tau^{\mathcal{B}})$. We can define another structure: $\mathcal{D} = (A, \tau^{\mathcal{A}}, \sigma^{\mathcal{B}})$, which consists of the domain $A = B$, the interpretation of symbols of τ given by $\mathcal{A}$, and the interpretation of symbols of σ given by $\mathcal{B}$. (We generally assume that τ and σ are disjoint. $\mathcal{D}$ is called an expansion of $\mathcal{A}$.) Now, we can write a formula ϕ, in some appropriate logic, that is true of a structure like $\mathcal{D}$ if and only if $\mathcal{D}$ is composed of an instance of P together with one of its solutions. This formula has the the property that

$$(A, \tau^{\mathcal{A}}, \sigma^{\mathcal{B}}) \models \phi \ \Leftrightarrow\ (\mathcal{A}, \mathcal{B}) \in P.$$

Finally, observe that, since $A = B$, we don't need the "Instance:" and "Find:" parts of a problem definition to talk about a complete structure at all, they just need to state the relevant vocabularies. (With the domain not explicitly given, it may be taken as the set of all elements appearing in the interpretations of τ.) So now a definition of a search problem has this simple form:

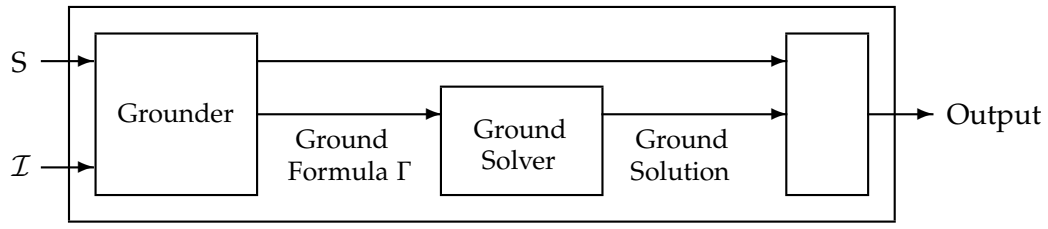

Figure 2: Typical Scheme for a Specification-Based Solver

Instance: τ,
Find: σ such that ϕ.

Example 1. *We continue with the Graph Colouring example. We have that* $\tau = (V, E, K)$ *and* $\sigma = (C)$. *Our problem specification looks like this:*

Instance: (V, E, K)*, where* E *is binary and* V, K *are unary,*
Find: C *such that* ϕ.

Here, ϕ *may be the conjunction of the following formulas of first order logic.*

1. $\forall x \, \forall y \, (Cxy \rightarrow (Vx \wedge Ky))$.
2. $\forall v \, (Vv \rightarrow \exists c \, (Kc \wedge Cvc))$.
3. $\forall u \, \forall v \, \forall c \, \forall d \, (Vu \wedge Vv \wedge Euv \wedge Cc \wedge Cd \wedge Cuc \wedge Cvd \rightarrow c \neq d)$.

The specification of Example 1 is perhaps not the most elegant or readable specification of the graph colouring problem, but it illustrates that we can now give a completely formal specification of the problem.

3 Implementation of Specification-Based Solving

Next we consider how we might go about building a suitable solver, that is, a piece of software that tries to construct solutions for problems based on a declarative specification, as in Figure 1. Problem-solving systems that fit this general scheme have been proposed many, many times, and implemented by a number of communities. While the terminologies, specification languages, and target application domains vary greatly, in almost all cases a similar general scheme is used to construct the solver. This scheme is illustrated in Figure 2.

The main elements of the scheme are, in our diagram, called the "grounder" and the "ground solver". The task carried out by the grounder is variously called "grounding", "flattening", "instantiation", etc. This task is to produce from the specification and instance a single expression with the property that solutions for the expression correspond to solutions for the problem instance. The language for this expression is said to be "ground", which essentially means it contains no quantifiers, and "flat", which means it does not have significant nesting of operators. So, this language is comparatively simple. The "ground solver" takes as input an expression in this ground or flat language, and searches for solutions. Examples of choices for this ground language include propsitional CNF formulas, sets linear inequalities, and other simple languages.

If the specification language is classical first order logic (FO), then the search problems that can be solved in the manner described in Section 2.1 are exactly those search problems for which the natural associated decision problem is in the complexity class NP. These include the NP-complete problems, which is a large class of problems (including many practical problems from many fields) that appear not to have general efficient solving algorithms, but for which a candidate solution can be efficiently verified. One such problem is that of checking satisfiability of a formula of propositional logic. The special case of this problem in which the formula is in a particular syntactic form, call conjunctive normal form (CNF), is called SAT. It happens that the a grounding algorithm that takes specifications in FO and produces a CNF formula is natural and efficient. It also happens that construction of efficient SAT solvers is an active and competitive area of work. Thus, this special case is both an elegant formal setting, and also a useful basic practical setting, to start with, even if ultimately we are interested in much richer languages.

Suppose we take as our specification the formula ϕ from Example 1, and the graph G with vertices $V = \{1, 2, 3\}$, edges $E = \{(1, 2), (2, 3)\}$, and colours $K = \{r, b\}$. A very simple grounding algorithm produces the propositional CNF formula $(1r \vee 1b) \wedge (2r \vee 2b) \wedge (3r \vee 3b) \wedge (\neg 1r \vee \neg 2r) \wedge (\neg 1b \vee \neg 2b) \wedge (\neg 3r \vee \neg 2r) \wedge (\neg 3b \vee \neg 2b)$. Here, the symbol $1r$ is a propositional atom that, if assigned true, will indicate that vertex 1 is coloured r. Each assignment of truth values to the atoms of this formula that makes the formula true identifies a proper colouring of the graph.

4 Remarks

The intention of these notes is to provide an introduction to the foundations of this area, a modest amount of exposure to practical problem solving, and an introduction to

several research questions regarding the effectiveness, potential, and limitations of the methods.

Very roughly speaking, this sort of approach to problem solving can be useful under a few different sets of circumstances. Here are a few:

1. A high-performance solver is needed for a computationally intensive problem, and engineering a problem-specific technology would be prohibitively expensive. This is frequently the case with problems for which there is no known polynomial time algorithm.
2. It's simply faster and easier to write a specification and apply a declarative solver than to find another way of solving a given problem, even if there are methods with better performance. This can be the case when the instances at hand are not too hard.
3. The exact nature of the problem keeps changing. In this case, revising a declarative specification and re-running the solver may not be hard, but re-engineering a high-performance problem-specific implementation would be impractical. This issue arises in many business, science, and other domains.

5 An Illustrative Example

Suppose you are an event organizer. At each event, there are several rooms (or several groups of people), and each room (or group) will be given a presentation by some specialist. For illustration, lets say we have 9 groups and 9 specialist presenters, and we want each group to get exactly one presentation by each specialist. We need nine timeslots, and can easily devise a schedule for presentations (which you might recognize as a 9-by-9 Latin Square), as follows:

Session:	1	2	3	4	5	6	7	8	9
Group 1 Speaker:	1	2	3	4	5	6	7	8	9
Group 2 Speaker:	2	3	4	5	6	7	8	9	2
Group 3 Speaker:	3	4	5	6	7	8	9	2	3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Group 9 Speaker:	9	1	2	3	4	5	6	7	8

It is easy to make an n-by-n Latin Square, for any natural number n, but what if there are constraints on the schedule? For example, you might need to have a certain presentation at a certain time (to accommodate certain speakers or guests), you might need to have

certain presentations in a particular order, etc. For simplicity, lets just say we have certain points in the schedule filled in ahead of time, like this:

Session:	?	?	4	?	?	?	?	?	9
Group 1 Speaker:	?	?	3	4	5	?	?	?	?
Group 2 Speaker:	5	6	?	?	?	7	8	9	3
Group 3 Speaker:	?	5	4	3	?	?	?	?	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Group 9 Speaker:	?	?	2	?	?	5	6	7	8

Now, constructing a solution is not so easy: The problem of deciding if a partially filled n-by-n table can be completely filled to make a Latin Square probably has no efficient algorithm. (More precisely, the problem is NP-complete. If you don't know what that means, we will visit the term later.)

How would you produce a program to solve this problem? What sort of algorithm would you use? And, for what values of n do you think it would produce solutions in reasonable time - meaning, fast enough to be useful in organizing the event?

Now, lets add one more twist. Suppose there will be coffee in three locations, after each 3 sessions. Groups 1-3 have coffee together, as do groups 4-6 and 7-9. To maximize discussion, the event sponsor wants guests in each coffee section who have heard each of the 9 speakers in one of the three sessions just before the break. A solution to this problem can be described as a 9-by-9 matrix satisfying the following constraints:

1. each cell contains one integer in $\{1, \ldots 9\}$;
2. each of the numbers in $\{1, \ldots 9\}$ appears in every row;
3. each of the numbers in $\{1, \ldots 9\}$ appears in every column;

Completing a partially-filled table to satisfy these constraints is a well-known puzzle. The problem generalizes to n-by-n tables for any n which is a perfect square, and this generalized version also is not known (or likely) to have an efficient algorithm.

Here are the results of a small experiment, using three different methods of solving this problem, for three different choices of n. (The puzzle boards were randomly generated, and the times reported are mean solving time for a number of instances.)

	Method 1	Method 2	Method 3
Time for 9-by-9 boards	0.7 sec	0 sec	0 sec
Time for 16-by-16 boards	204 sec	0.01 sec	0.07 sec
Time for 25-by-25 boards	> 600 sec	0.09 sec	0.3 sec

We are accustomed to the 9-by-9 puzzles in books and magazines not being very hard, but that is partly because they are chosen to be so. Many 9-by-9 puzzles are not easy, and as the table suggests, difficulty increases rapidly with size. 36-by-36 puzzles are not solvable by Method 1 in reasonable time, and often take substantial time (considering the size of the inputs!) for the other methods.

Here is a brief description of the three methods.

Method 1: A Hand-crafted search algorithm. A program, written in C++, which reads in the puzzle board (in a commonly-used file format), and then searches for a solution, based on a method found on the internet (along with an suggestiong that it is a good solution). It uses a back-tracking search, refined by a number of problem-specific techniques for reducing search time of naive back-tracking.

Method 2: Hand-crafted transformation to SAT. A program, written in C++, reads in the puzzle board, and generates a formula of propositional logic with the property that satisfying assignments for the formula correspond to solutions to the puzzle. This formula is then sent to a "SAT Solver" (a program that searches for satisfying assignments to formulas of propositional logic), which finds a solution.

Method 3: Specification-based Solver. A specification for the problem, written in a high-level language, for a general-purpose combinatorial problem solving tool. The specification and an instance are sent to the solver, which produces a solution. (Actually, in this case, the solver generates a formula of propositional logic and sends it to a SAT solver, but the user does not need to describe the transformation - it is automatic.) The specification involved is about half a page long, and a small program is needed to transform the instance to the solver format.

Method 1 is the sort of thing that most people try. The other two are the sorts of methods these notes are about, and we will describe them in detail later. It is worth saying something about the amount of programmer time or effort involved in the three methods. Method 1 requires the greatest programmer time, but produces the worst solver. Method 2 produces the fastest solver, but requires some knowledge beyond understanding the problem. One must know how to generate the formula, and indeed the formula used is not the most obvious, and some experimentation was involved in choosing it. The easiest solution to generate is Method 3, which requires only a few hours of work. Although it does not produce as fast a solver as Method 2, it is much faster than Method 1, and involved a lot less work.