CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 24: Satisfiability Modulo Theories

Harvard SEAS - Fall 2025 *2025-11-25*

1 Announcements

- Final exam: Friday 12/12 9am-12pm, Science Center B.
- Review sessions: 12/5 12-3pm (content review, going through algorithms), 12/7 1-3pm (proof strategies for reductions)
- Final exam cheat sheet: One *handwritten* page, double-sided. (Rationale: educational value of actively summarizing the course content yourself.)
- Madhu Sudan P vs. NP lecture highly recommended: Wed 12/3 5p-6pm, Science Center D.
- My last OH for the term: Mon 12/1 5pm-5:45pm (note earlier start time).

Recommended Reading:

- Hesterberg–Vadhan 27
- De Moura & Bjørner: `https://cacm.acm.org/magazines/2011/9/122785-satisfiability-modulo-theories/abstract`

2 Program Verification and Analysis

A dream is that we could just write a mathematical specification of what a program P should do, such as the following , and then automatically verify that a program meets the specification:

$$\text{Spec} : \forall A, \ell, u, K \ (0 \leq \ell \leq u, \forall i \ A[i-1] \leq A[i]) \rightarrow (P(A, \ell, u, K) = \texttt{yes} \leftrightarrow \exists i \in [\ell, u] \ A[i] = K)\,.$$

More precisely, our dream is an algorithm V that given a specification Spec and a program P, $V(\text{Spec}, P)$ will always tell us whether or not P satisfies Spec.

Q: Why is the dream not achievable?

A: Like with the NP-complete problems, this does not mean we completely give up on building useful software tools to analyze and verify programs. But it does mean that all such tools must have one or more of the following limitations:

-
-
-
-

Nevertheless, the tools for program verification have grown in power and sophistication over the years, and today we will study one of the most powerful approaches, known as Satisfiability Modulo Theories.

3 Constraint Satisfaction In Theories

Constraint satisfaction problems (CSPs) are a general framework for defining computational problems, where the instances are a set of constraints over a finite set of variables $z_0, \ldots, z_{n-1}$ and we ask to find an assignment to the variables that satisfies all of the constraints, or declare that the set of constraints is unsatisfiable. SAT is an example, where the constraints allowed are *disjunctions* of literals and we ask whether there is a *Boolean* assignment to the variables that satisfies all of the constraints. Here we consider a much more general formulation where (a) the variables can take values in an arbitrary domain $\mathcal{D}$ and (b) the constraints can come from an arbitrary collection $\mathcal{P}$.

Definition 3.1. A *theory* $\mathcal{T}$ consists of a domain $\mathcal{D}$ and a collection $\mathcal{P}$ of predicates $p : \mathcal{D}^k \to \{0,1\}$, $k \geq 0$.

Let's see some examples:

Definition 3.2. The *Theory of Naturals* is a theory with domain $\mathcal{D} = \mathbb{N}$, with the following predicates and their negations.

-
-
-

Now an example instance of Constraint Satisfaction in the Theory of Naturals is the following:

$$\begin{array}{ccc} x \neq 0, & x_2 = x \times x, & x_4 = x_2 \times x_2, \\ y \neq 0, & y_2 = y \times y, & y_4 = y_2 \times y_2, \\ z \neq 0, & z_2 = z \times z, & z_4 = z_2 \times z_2, \\ & x_4 + y_4 = z_4 & \end{array}$$

Q: What Diophantine Equation does the above look for solutions to?

A:

Let's abstract how we derive a constraint-satisfaction computational problem for a theory.

Definition 3.3. For a theory $\mathcal{T} = (\mathcal{D}, \mathcal{P})$, CONSTRAINT SATISFACTION IN $\mathcal{T}$ is the following computational problem:

> **Input:** A sequence $z = (z_0, \ldots, z_{n-1})$ of n *theory variables*, and a sequence of *theory predicates* $P_0(z), \ldots, P_{m-1}(z)$, each of which is obtained by applying a predicate $p \in \mathcal{P}$ to a sequence $(z_{i_0}, \ldots, z_{i_{k-1}})$ of theory variables.
>
> **Output:** An assignment

Computational Problem CONSTRAINT SATISFACTION IN $\mathcal{T}$

Generalizing the above Fermat example, it is possible to prove that DIOPHANTINE EQUATIONS $\leq$ CONSTRAINT SATISFACTION IN THE THEORY OF NATURALS and thus we have:

Theorem 3.4. CONSTRAINT SATISFACTION IN THE THEORY OF NATURALS *is unsolvable.*

Let's turn to some solvable examples:

Definition 3.5. The *Theory of Disjunctions* is the theory $(\mathcal{D}, \mathcal{P})$ with $\mathcal{D} = \{0, 1\}$, with the predicate family $\mathcal{P} = \{p_{k,\ell} : k, \ell \in \mathbb{N}\}$, where

$$p_{k,\ell}(x_0, \ldots, x_{k+\ell-1}) = [x_0 \vee \cdots \vee x_{k-1} \vee \neg x_k \vee \cdots \vee \neg x_{k+\ell-1}].$$

Notice that CONSTRAINT SATISFACTION IN THE THEORY OF DISJUNCTIONS is equivalent to SAT.

Definition 3.6. The *Theory of Bitvectors of length w* is the theory $(\mathcal{D}_w, \mathcal{P})$ with $\mathcal{D}_w = \{0, 1, \ldots, 2^w - 1\} \equiv \{0, 1\}^w$, with the following predicates and their negations

-

-

Q: Why is this theory solvable?

A:

As we see in this example, sometimes the domain and/or the predicates in the theory have one or more size parameters, like the word size w, which may affect the complexity of solving the problem.

Definition 3.7. The *Theory of Difference Arithmetic* is the theory $(\mathcal{D}, \mathcal{P})$ with predicates

-

Here, even though the domain $\mathcal{D}$ is infinite, it is a solvable theory and in fact can be solved in polynomial time. Specifically, Constraint Satisfaction in the Theory of Difference reduces to single-source shortest paths in directed graphs that can have both positive and negative edge weights, which can be solved by the Bellman–Ford algorithm (which we have not covered, but which you can learn about in CS1240). Only very recently (2022) was a nearly linear time algorithm found (https://arxiv.org/abs/2203.03456).

4 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) provides us with even richer computational problems that combine SAT(with Boolean variables) with Constraint Satisfaction in $\mathcal{T}$, allowing us to have a system of constraints like the following over the Theory of Difference Arithmetic, where $x_0, \ldots, x_n$ are propositional (i.e. Boolean) variables, $s_0, \ldots, s_n$ are rational variables, and $v_0, \ldots, v_{n-1}, t \in \mathbb{Q}$ are constants given in the input:

- $s_0 = 0$,
- $x_i \to s_{i+1} = s_i + v_i$ for $i = 0, \ldots, n-1$,
- $\neg x_i \to s_{i+1} = s_i$ for $i = 0, \ldots, n-1$,
- $s_n = t$

Q: What familiar problem do the above constraints encode?

A:

In general, an SMT instance is given by CNF formula where some of the variables are replaced by predicates over theory variables. For example, the Subset Sum constraints as above with $n = 3$, $(v_0, v_1, v_2) = (1, 3, 9)$ and $t = 10$ can be written as the following CNF formula over propositional variables and predicates of theory variables:

Note that we have replaced the implications $a \to b$ with the equivalent disjunction $\neg a \vee b$. The predicates above are not exactly in the Theory of Difference Arithmetic as specified above, which only has inequalities, but equalities can be expressed using two inequalities. And equality of a theory variable s to a constant c can be expressed by introducing a global variable z to represent 0 and requiring that $s - z = c$. (It can be shown that if there is a solution, then there is also one where $z = 0$ by shifting all theory variables by a constant).

In general, SMT refers to computational problems of the following type.

Definition 4.1. For a theory $\mathcal{T} = (\mathcal{D}, \mathcal{P})$, Satisfiability Modulo $\mathcal{T}$ is the following computational problem:

Input: A CNF formula $\varphi(x_0, \ldots, x_{n_p-1}, y_0, \ldots, y_{n_q-1})$ on n_p *propositional variables* and n_q *auxiliary variables*, a sequence $z = (z_0, \ldots, z_{n_t-1})$ of n_t *theory variables*, and a sequence of n_q *theory predicates* $P_0(z), \ldots, P_{n_q-1}(z)$, each of which is obtained by applying a predicate $p \in \mathcal{P}$ to a sequence $(z_{i_0}, \ldots, z_{i_{k-1}})$ of theory variables.

Output: Assignments $\alpha \in \{0,1\}^{n_p}$, $\beta \in \mathcal{D}^{n_t}$ such that

$$\varphi(\alpha, P_0(\beta), \ldots, P_{n_q-1}(\beta)) = 1,$$

if such assignments exist.

Computational Problem Satisfiability Modulo $\mathcal{T}$

For an illustration of how a solution to an SMT instance evaluates to true or false, see here:

Theorem 4.2. *For every theory $\mathcal{T} = (\mathcal{P}, \mathcal{D})$,* Satisfiability Modulo $\mathcal{T}$ *is solvable iff* Constraint Satisfaction in $\mathcal{T}'$ *is solvable for $\mathcal{T}' = (\mathcal{D}, \mathcal{P} \cup \neg\mathcal{P})$ is solvable, where $\neg\mathcal{P}$ includes the negations of all predicates in $\mathcal{P}$.*

Proof sketch.
$\Rightarrow$

$\Leftarrow$ We give a (exponential-time) reduction from Satisfiability Modulo $\mathcal{T}$ to Constraint Satisfaction in $\mathcal{T}'$.

$\square$

Like SAT Solvers, there has been an enormous amount of engineering effort put into designing highly optimized SMT Solvers, which perform well on many useful real-world instances, even when the theory $\mathcal{T}$ is unsolvable. You can play around with one such solver, Z3, at `https://microsoft.github.io/z3guide/`.

5 Using SMT Solvers for Program Analysis

Let's see an example of how SMT Solvers are used for finding bugs in programs. Consider the following program for Binary Search:

```
  BinarySearch($A, \ell, u, k$):
  Input  : Integers $0 \le \ell \le u$, a sorted array $A$ of length at least $u$, a
           key $k$
  Output : yes if $k \in \{A[\ell], A[\ell+1], \dots, A[u-1]\}$, no otherwise
0 while $\ell < u$ do
1     $m = (\ell + u)/2$;
2     if $A[m] = k$ then
3         return yes
4     else if $A[m] > k$ then
5         $\ell = \ell$; $u = m$
6     else $\ell = m + 1$; $u = u$;
7     assert $0 \le \ell \le u$;
8 return no
```

Algorithm 5.1: `BinarySearch()`

This looks like a correct implementation of binary search, but to be sure we have added an assert command in Line 7 to make sure that we got all of our arithmetic right and maintain the invariant that $0 \le \ell \le u$.

For our `BinarySearch` program here, we will consider whether there are inputs that make the assertion fail *within the first two iterations of the loop.* To do this, we will have the following variables in our SMT formula:

- x_i for $i = 2, \dots, 10$: propositional variable representing whether or not we execute line i in the first iteration of the loop.
- x'_i for $i = 2, \dots, 10$: propositional variable representing whether or not we execute line i in the second iteration of the loop.
- x_f: propositional variable representing whether the assertion fails during the first two iterations of the loop. (So x_f will be false if the program either halts with an output during the first two iterations or reaches the end of the second iteration without the assertion failing.)
- ℓ, u, k: integer variables representing the input values for ℓ, u, k
- ℓ', u': integer variables representing the values of ℓ, u if and when Line 7 is reached in the first iteration of the loop.
- ℓ'', u'': integer variables representing the values of ℓ, u if and when Line 7 is reached in the second iteration of the loop.
- m, m'; integer variables representing the values assigned to m in the first and second iterations of the loop.
- a, a': representing values of $A[m]$ and $A[m']$.

We then construct our formula as the conjunction of the following constraints, corresponding to the input preconditions (Constraint 1–2), the control flow and assignments made by the program (Constraints 3–31)), and asking for the assertion to fail (Constraint 32).

1. $(0 \leq \ell) \land (\ell \leq u)$
2. $((m \leq m') \to (a \leq a'))$
   $\land ((m \geq m') \to (a \geq a'))$
3. (x_2)
4. $(x_2 \land (\ell < u)) \leftrightarrow x_3$
5. $(x_2 \land \neg(\ell < u)) \leftrightarrow x_{10}$
6. $x_3 \to ((m = (\ell + u)/2) \land x_4)$
7. $(x_4 \land (a = k)) \leftrightarrow x_5$
8. $(x_4 \land \neg(a = k)) \leftrightarrow x_6$
9. $x_5 \to \neg x_f$
10. $(x_6 \land (a > k)) \leftrightarrow x_7$
11. $(x_6 \land \neg(a > k)) \leftrightarrow x_8$
12. $x_7 \to ((u' = m) \land (\ell' = \ell) \land x_9)$
13. $x_8 \to ((\ell' = m + 1) \land (u' = u) \land x_9)$
14. $x_7 \lor x_8 \leftrightarrow x_9$
15. $(x_9 \land \neg((0 \leq \ell') \land (\ell' \leq u')) \to x_f$
16. $(x_9 \land (0 \leq \ell') \land (\ell' \leq u')) \leftrightarrow x_2'$
17. $x_{10} \to \neg x_f$
18. $(x_2' \land (\ell' < u')) \leftrightarrow x_3'$
19. $(x_2' \land \neg(\ell' < u')) \leftrightarrow x_{10}'$
20. $x_3' \to ((m' = (\ell' + u')/2) \land x_4')$
21. $(x_4' \land (a' = k)) \leftrightarrow x_5'$
22. $(x_4' \land \neg(a' = k)) \leftrightarrow x_6'$
23. $x_5' \to \neg x_f$
24. $(x_6' \land (a' > k)) \leftrightarrow x_7'$
25. $(x_6' \land \neg(a' > k)) \leftrightarrow x_8'$
26. $x_7' \to ((u'' = m') \land (\ell'' = \ell') \land x_9')$
27. $x_8' \to ((\ell'' = m' + 1) \land (u'' = u') \land x_9')$
28. $x_7' \lor x_8' \leftrightarrow x_9'$
29. $(x_9' \land \neg((0 \leq \ell'') \land (\ell'' \leq u'')) \to x_f$
30. $(x_9' \land (0 \leq \ell'') \land (\ell'' \leq u'')) \to \neg x_f$
31. $x_{10}' \to \neg x_f$
32. x_f

To apply an SMT Solver, however, we need to select a "theory" that tells us the domain that the theory variables range over and how to interpret the operations and (in)equality symbols. If we use The Theory of Natural Numbers, then we will find out that φ is *unsatisfiable*, because Algorithm 8 is a correct instantiation of Binary Search over the natural numbers.

However, if we implement Algorithm 8 in the C programming language using the `unsigned int` type, then we should not use the theory of natural numbers, but use the Theory of Bitvectors *with modular arithmetic*, because C `unsigned int`'s are 32-bit words, taking values in the range $\{0, 1, 2, \ldots, 2^{32} - 1\}$ with modular arithmetic. And in this case, an SMT Solver will find that the formula is *satisfiable*! One satisfying assignment will have:

1. $\ell = 2^{31}$
2. $u = 2^{31} + 2$
3. $m = (\ell + u)/2 =$ __________
4. $a = 0,\ k = 1$
5. $\ell' = \ell = 2^{31}$
6. $u' = m + 1 =$ __________

This violates the assertion that $\ell' \leq u'$ — a genuine bug in our implementation of binary search!

6 The Cook–Levin Theorem

Now we'll see how the ideas used for the `BinarySearch` example above can be generalized to perform automatic program analysis of an *arbitrary* Word-RAM program via Satisfiability Modulo the Theory of Bitvectors. We will then use that to give a proof sketch of the Cook–Levin Theorem.

We consider the following very general problem:

Input: A Word RAM program P, an array of natural numbers $x = (x_0, \ldots, x_{n-1})$ and parameters $w, m, t \in \mathbb{N}$.

Output: An array $y = (y_0, \ldots, y_{m-1})$ of natural numbers y such that:

1. $P[w]$ halts without crashing on input (x, y) within t steps, and
2. $P[w](x, y) = 1$,

if such a y exists.

Computational Problem Word-RAM Satisfiability

Many debugging problems can easily be reduced to Word-RAM Satisfiability, if we fix bounds on the input length and the running time that we care about. For example, if we take $n = 0$ and modify P to output 1 only when an arithmetic overflow occurs, then Word-RAM Satisfiability will tell us whether there is an input y of length m that causes an arithmetic overflow within t steps.

Theorem 6.1. Word-RAM Satisfiability *can be reduced to* Satisfiability Modulo the Theory of Bitvectors. *Given an instance* (P, x, w, m, t) *of* Word-RAM Satisfiability*, the (mapping) reduction produces an* SMT *instance with the same word size parameter* w *and runs in time polynomial in* $|P|$*,* $|x|$*,* m*, and* t*.*

Proof sketch. Given a Word-RAM Satisfiability instance (P, x, w, m, t), we construct our SMT instance very similarly to the binary search example:

- Propositional variables:

- Theory variables:

- Constraints:

□

Now let's use this to prove the Cook–Levin Theorem.

Theorem 6.2. *For every problem* $\Pi \in \mathsf{NP}_{\mathsf{search}}$ $\Pi \leq_p$ Word-RAM Satisfiability. *Specifically, an instance x of Π of length N maps to an instance of* Word-RAM Satisfiability *with* $n = N$, $m = N^{O(1)}$, $t = N^{O(1)}$, *and* $w = O(\log N)$.

Proof sketch.

□

Theorem 6.3. Satisfiability Modulo the Theory of Bitvectors *reduces to* SAT *in time polynomial in the length of the input and* 2^w.

There is a more efficient reduction that has complexity polynomial in w, but we don't need it because Theorem 6.2 gives us word length $w = O(\log N)$.

Proof sketch.

□

Combining Theorems 6.2, 6.1, and 6.3 shows that SAT is $\mathsf{NP}_{\mathsf{search}}$-hard. Thus, we have completed the proof of:

Theorem 6.4 (Cook–Levin Theorem)**.** SAT *is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*