CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 24: Satisfiability Modulo Theories and Cook–Levin

Harvard SEAS - Fall 2024 *2024-11-26*

1 Announcements

Recommended Reading:

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

Announcements:

- PS9 due Wed 12/4
- No section this week
- Final exam Thu 12/19, 9am. Cheat sheet allowed; details TBA.
- Practice exams to be released and review sessions and OH to be announced 12/4.
- Late revision videos will be accepted until 12/11 at half credit.
- CS1200 merch ideas due Sun 12/1.
- Today's material is mostly for fun/culture; you do not need to be able to work with it at a technical level, but it may help reinforce some of the earlier material in the course.
- Q evaluations open; please fill out.
- PS7 feedback: median 9hrs, 75th percentile 10hrs. Resolution problem a bit tedious, 3d complete matching input format ambiguous.
- SRE6 feedback: 89% of senders and 93% of receivers found it useful. Visualizations very helpful; DCE students could use ability to share screen.
- Happy Thanksgiving!

2 Program Verification and Analysis

Buggy software is a pervasive problem in computer science. A dream (especially for TFs in CS courses!) is that we could just write a mathematical specification of what a program P should do, such as the following (for the correctness of searching a sorted array), and then automatically verify that a program meets the specification:

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

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. Ideally, V either produces an easily verifiable *proof* that P satisfies Spec or that P fails to satisfy Spec. The latter (a proof that P fails to satisfy Spec) often consists of a *counterexample*, i.e. an input (e.g. (A, ℓ, u, K)) on which P fails to meet its specification.

Q: Why is the dream not achievable?

A: The unsolvability of the Halting Problem tells us that there are no such tools V that work in general, even for a very simple specification like "halts and produces an output on a fixed input x". Indeed, Rice's Theorem says that verifying *any* non-trivial semantic property of programs is unsolvable.

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:

- They apply to only a limited class P of programs (e.g. ones annotated by programmers with appropriate assertions, ones without unbounded loops, ones that only use certain data types and operations).
- They apply only to a limited class of specifications (e.g. checking that specific assertions fail as above, type-safety, etc.).
- They can find some bugs, but may fail to find bugs even on incorrect programs.
- They can sometimes find a proof of satisfying a specification, but may fail to find proofs even for correct programs.

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 Modulo 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 an example.

The Theory of Naturals. $\mathcal{D} = \mathbb{N}$, with the following predicates and their negations.

- $p_+(x, y, z) = [x + y = z]$,
- $p_\times(x, y, z) = [x \times y = z]$, and
- $p_c(x) = [x = c]$ for each $c \in \mathbb{N}$

An example instance of a CSP Modulo the Theory of Naturals is the following:

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

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

A: $x^4 + y^4 = z^4$. By Fermat's Last Theorem, this equation has no solution. Generalizing this idea, we can see the Diophantine Equations (an unsolvable problem) reduces to CSPs Modulo the Theory of Naturals, so the latter is also unsolvable.

Let's turn to some solvable examples:

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

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

CSPs Modulo The Theory of Disjuntions is just another way of describing SAT.

The Theory of Bitvectors of length w. Here the domain is $\mathcal{D}_w = \{0, 1, \ldots, 2^w - 1\} \equiv \{0, 1\}^w$, with the following predicates and their negations

- $p_{\texttt{op}}(x, y, z) = [x \texttt{ op } y = z]$ for whatever word operations `op` we want to allow (e.g. saturation arithmetic or modular arithmetic, possible bitwise XOR/AND/OR, etc.). We'll use the word operations that we used to define the Word-RAM Model.
- $p_c(x) = [x = c]$ for each $c \in \mathcal{D}$.

Q: Why is this theory solvable?

A: Since $\mathcal{D}_w$ is finite, we can do exhaustive search over assignments $\alpha \in (\mathcal{D}_w)^n$.

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.

The Theory of Difference Arithmetic: $\mathcal{D} = \mathbb{Q}$, with predicates

- $p_c(x, y) = [x - y \leq c]$ for each $c \in \mathbb{Q}$.

Here, even though the domain $\mathcal{D}$ is infinite, it is a solvable theory and in fact can be solved in polynomial time. Solving CSPs Modulo the Theory of Difference Arithmetic reduces to single-source shortest paths in directed graphs that can have both positive and negative edge weights. If you take CS1240, you'll learn the Bellman-Ford algorithm for solving this latter problem, and only very recently (2022) was a nearly linear time algorithm found (https://arxiv.org/abs/2203.03456).

The general definition of CSPs Modulo a Theory $\mathcal{T} = (\mathcal{D}, \mathcal{P})$ is as follows:

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 $\beta \in \mathcal{D}^n$ such that $$P_0(\beta) = P_1(\beta) = \cdots = P_{m-1}(\beta) = 1.$$ if such an assignment exists.

Computational Problem ConstraintSatisfactionInT

4 Satisfiability Modulo Theories

In *Satisfiability* Modulo Theories, we combine SAT over Boolean variables with predicates from a theory, 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: Subset Sum! The Boolean variables x_i determine which of the input variables v_i we include in our subset, and the s_i's represent a running sum. We see from this example that this extra flexibility makes it much easier to encode NP-complete problems as SMT instances rather than just as SAT instances. (The latter would require breaking down each number v_i into the binary representation of its numerators and denominators and writing Boolean formulas to express the addition.)

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:

$$([s_0 = 0]) \wedge (\neg x_0 \vee [s_1 = s_0 + 1]) \wedge (\neg x_1 \vee [s_2 = s_1 + 3]) \wedge (\neg x_2 \vee [s_3 = s_2 + 9])$$
$$\wedge (x_0 \vee [s_1 = s_0]) \wedge (x_1 \vee [s_2 = s_1]) \wedge (x_2 \vee [s_3 = s_2] \wedge ([s_3 = 10]),$$

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).

Formally, the SMT problem for theory $\mathcal{T} = (\mathcal{D}, \mathcal{P})$ is described as follows:

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 SatisfiabilityModuloT

For an illustration of how a solution to an SMT instance evaluates to true or false, see Figure 1.

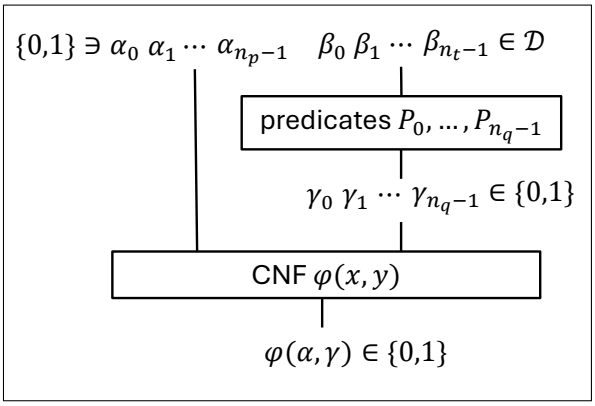

Figure 1: Evaluating a solution (α, β) to an SMT instance $(\varphi, P_0, \ldots, P_{n_q-1})$

Example: In our example above, the CNF formula and predicates are as follows:

$$\varphi(x_0, x_1, x_2, y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7)$$
$$= (y_0) \wedge (\neg x_0 \vee y_1) \wedge (\neg x_1 \vee y_2) \wedge (\neg x_2 \vee y_3) \wedge (x_0 \vee y_4) \wedge (x_1 \vee y_5) \wedge (x_2 \vee y_6) \wedge (y_7),$$

and

$$\begin{aligned}
P_0(s_0, s_1, s_2, s_3) &= [s_0 = 0] \\
P_1(s_0, s_1, s_2, s_3) &= [s_1 = s_0 + 1] \\
P_2(s_0, s_1, s_2, s_3) &= [s_2 = s_1 + 3] \\
P_3(s_0, s_1, s_2, s_3) &= [s_3 = s_2 + 9] \\
P_4(s_0, s_1, s_2, s_3) &= [s_1 = s_0] \\
P_5(s_0, s_1, s_2, s_3) &= [s_2 = s_1] \\
P_6(s_0, s_1, s_2, s_3) &= [s_3 = s_2] \\
P_7(s_0, s_1, s_2, s_3) &= [s_3 = 10]
\end{aligned}$$

Observe that $(x_0, x_1, x_2) = (1, 0, 1)$ and $(s_0, s_1, s_2, s_3) = (0, 1, 1, 10)$ gives a satisfying assignment to the SMT instance.

Interestingly, the formula φ is a 2-SAT formula. So even though both 2-SAT and CSPs Modulo the Theory of Difference Arithmetic are in $\mathsf{P}_{\mathsf{search}}$, by combining them this way, we can express $\mathsf{NP}_{\mathsf{search}}$-hard problems like SubsetSum.

Theorem 4.1. *For every theory $\mathcal{T}$, Satisfiability Modulo $\mathcal{T} = (\mathcal{D}, \mathcal{P})$ is solvable iff Constraint Satisfaction Modulo $\mathcal{T}' = (\mathcal{D}, \mathcal{P} \cup \neg\mathcal{P})$ is solvable, where $\neg\mathcal{P}$ includes the negations of all predicates in $\mathcal{P}$.*

Proof sketch.
$\Rightarrow$ Immediate because conjunctions of literals are a special case of CNF formulas.
$\Leftarrow$ We give a (exponential-time) reduction from Satisfiability Modulo $\mathcal{T}$ to Constraint Satisfaction Modulo $\mathcal{T}'$:

Input : An Satisfiability Modulo $\mathcal{T} = (\mathcal{D}, \mathcal{P})$ instance $\varphi(x_0, \ldots, x_{n_p-1}, y_0, \ldots, y_{n_q-1})$, $z = (z_0, \ldots, z_{n_t-1})$, $P_0(z), \ldots, P_{n_q-1}(z)$
Output : $\alpha \in \{0,1\}^{n_p}$, $\beta \in \mathcal{D}^{n_q}$ s.t. $\varphi(\alpha, P_0(\beta), \ldots, P_{n_q-1}(\beta)) = 1$, or $\perp$ if none exist.

```
foreach α ∈ {0,1}^{n_p}, γ ∈ {0,1}^{n_q} do
    if φ(α, γ) = 1 then
        foreach i = 0 to n_q-1 do
            if γ_i = 1 then Q_i = P_i;
            else Q_i = ¬P_i;
        β = ConstraintSatisfaction(z, Q_0, ..., Q_{n_q-1});
        if β ≠ ⊥ then return (α, β);
return ⊥
```

Algorithm 1: Exhaustive-Search SMT Solver

□

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/. Instead of doing exhaustive search, these practical SMT Solvers use a SAT solver to find a satisfying assignment (α, γ) to φ, and then apply a constraint satisfaction solver.

If the constraint satisfaction solver determines there is no corresponding assignment to the theory variables, we can use its output to construct a new clause to add to φ. For example, we can add a clause of length n_q over the auxiliary variables y ruling out the specific assignment γ, but sometimes the constraint solver can provide more information that allows for constructing a shorter clause that rules out more assignments.

5 Using SMT Solvers for Program Analysis

Let's see an example of how SMT Solvers (for "Satisfiability Modulo Theories"), are used for finding bugs in programs. Consider the following program for Binary Search:

```
1  BinarySearch(A, ℓ, u, k)
   Input   : Integers 0 ≤ ℓ ≤ u, a sorted array A of length at least u, a key k
   Output  : yes if k ∈ {A[ℓ], A[ℓ + 1], ..., A[u − 1]}, no otherwise
2  while ℓ < u do
3  |  m = (ℓ + u)/2;
4  |  if A[m] = k then
5  |  |  return yes
6  |  else if A[m] > k then
7  |  |  ℓ = ℓ; u = m
8  |  else ℓ = m + 1; u = u;
9  |  assert 0 ≤ ℓ ≤ u;
10 return no
```

This looks like a correct implementation of binary search, but to be sure we have added an assert command in Line 9 to make sure that we got all of our arithmetic right and maintain the invariant that $0 \leq \ell \leq u$. Now, to test for bugs, we could run the program on many different inputs and see if the assert command ever fails. But there are infinitely many choices for the inputs, and even for the pair (ℓ, u) of bounds there are roughly 2^{64} choices if they are 32-bit words, which is infeasible to exhaustively enumerate.

SMT Solvers allow us to more efficiently search for inputs that would violate a condition (or reach a certain state) in a program. 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.* (The approach can be generalized to any finite number of loop iterations, with a corresponding blow-up in the size of the SMT instance.) To do this, we will have the following variables in our SMT formula:

- x_i for $i = 2, \ldots, 10$: propositional variable representing whether or not we execute line i in the first iteration of the loop.
- x'_i for $i = 2, \ldots, 10$: 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 9 is reached in the first iteration of the loop.
- ℓ'', u'': integer variables representing the values of ℓ, u if and when Line 9 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. $(0 \le \ell) \wedge (\ell \le u)$
2. $((m \le m') \to (a \le a'))$ $\wedge ((m \ge m') \to (a \ge a'))$
3. (x_2)
4. $(x_2 \wedge (\ell < u)) \leftrightarrow x_3$
5. $(x_2 \wedge \neg(\ell < u)) \leftrightarrow x_{10}$
6. $x_3 \to ((m = (\ell + u)/2) \wedge x_4)$
7. $(x_4 \wedge (a = k)) \leftrightarrow x_5$
8. $(x_4 \wedge \neg(a = k)) \leftrightarrow x_6$
9. $x_5 \to \neg x_f$
10. $(x_6 \wedge (a > k)) \leftrightarrow x_7$
11. $(x_6 \wedge \neg(a > k)) \leftrightarrow x_8$
12. $x_7 \to ((u' = m) \wedge (\ell' = \ell) \wedge x_9)$
13. $x_8 \to ((\ell' = m + 1) \wedge (u' = u) \wedge x_9)$
14. $x_7 \vee x_8 \leftrightarrow x_9$
15. $(x_9 \wedge \neg((0 \le \ell') \wedge (\ell' \le u')) \to x_f$
16. $(x_9 \wedge (0 \le \ell') \wedge (\ell' \le u')) \leftrightarrow x'_2$
17. $x_{10} \to \neg x_f$
18. $(x'_2 \wedge (\ell' < u')) \leftrightarrow x'_3$
19. $(x'_2 \wedge \neg(\ell' < u')) \leftrightarrow x'_{10}$
20. $x'_3 \to ((m' = (\ell' + u')/2) \wedge x'_4)$
21. $(x'_4 \wedge (a' = k)) \leftrightarrow x'_5$
22. $(x'_4 \wedge \neg(a' = k)) \leftrightarrow x'_6$
23. $x'_5 \to \neg x_f$
24. $(x'_6 \wedge (a' > k)) \leftrightarrow x'_7$
25. $(x'_6 \wedge \neg(a' > k)) \leftrightarrow x'_8$
26. $x'_7 \to ((u'' = m') \wedge (\ell'' = \ell') \wedge x'_9)$
27. $x'_8 \to ((\ell'' = m' + 1) \wedge (u'' = u') \wedge x'_9)$
28. $x'_7 \vee x'_8 \leftrightarrow x'_9$
29. $(x'_9 \wedge \neg((0 \le \ell'') \wedge (\ell'' \le u'')) \to x_f$
30. $(x'_9 \wedge (0 \le \ell'') \wedge (\ell'' \le u'')) \to \neg x_f$
31. $x'_{10} \to \neg x_f$
32. x_f

Each of these constraints can be turned into a small CNF formula whose variables are either propositional variables or propositions asserting (in)equalities involving the theory variables, like $(m = (\ell + u)/2)$ or $\neg(\ell' \le u'')$. (Recall that every boolean function on k variables can be written as a k-CNF. Each of the constraints above involves at most 4 clauses.) Taking the AND of all of these CNFs yields a larger CNF that is part of our SMT instance φ.

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 10 is a correct instantiation of Binary Search over the natural numbers.

However, if we implement Algorithm 10 in C 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. $m = (\ell+u)/2 = ((2^{31}+2^{31}+2) \bmod 2^{32})/2 = 1$
4. $a = 0$, $k = 1$
5. $\ell' = \ell = 2^{31}$
6. $u' = m + 1 = 2$

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

6 The Cook–Levin Theorem

Now we'll see how the ideas used for the Binary Search 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 WordRAMSatisfiability

Many debugging problems can easily be reduced to WordRAMSatisfiability, 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 WordRAMSatisfiability will tell us whether there is an input y of length m that causes an arithmetic overflow within t steps.

Theorem 6.1. *WordRAMSatisfiability can be reduced to Satisfiability Modulo the Theory of Bitvectors. Given an instance (P, x, w, m, t) of WordRAMSatisfiability, 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 WordRAMSatisfiability instance (P, x, w, m, t), we construct our SMT instance very similarly to the binary search example:

- Propositional variables: $u_{i,j}$ for $i = 0, \ldots, \ell$ and $j = 0, \ldots, t$, meant to capture whether the computation of P on (x, y) is at line i at time j.
- Theory variables: $\texttt{var}_{i,j}$ for $i = 0, \ldots, v-1$ and $j = 0, \ldots, t$, S_j for $j = 0, \ldots, t$, and $M_j[i]$ for $i = 0, \ldots, n+t-1$ and $j = 0, \ldots, t$ corresponding to the value of P's v variables, the amount of memory allocated, and the values in the first $n+t$ memory locations of P at time j. (Note that in t time steps, P can use at most $n + t$ memory locations, since it can issue at most t MALLOC commands.)
- Constraints: initialization of the variables, memory locations, and line number of P at time 0; updates from time t to time $t + 1$; and requirement that P halts in t steps with output 1

□

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

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

Proof sketch. Let P be the polynomial-time verifier for Π. By using bignum arithmetic, we may assume without loss of generality that P takes its input (x, y) as an array of $N + m$ bits (rather than words) where $m = N^{O(1)}$ is the bitlength of solutions to Π. Then the minimum non-crashing wordsize is $w = O(\log(N + m)) = O(\log N)$ by the fact that P runs in polynomial time and all the numbers in the input are in $\{0, 1\}$. Setting $t = N^{O(1)}$ to be the running time of P, our reduction is $x \mapsto (P, x, w, m, t)$.

□

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. Replace each w-bit theory variable with w Boolean variables, and each constraint $p(x, y, z)$ on w-bit words with the brute-force CNF of size at most $2^{3w} = (2^w)^{O(1)}$. □

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.*