

CSE 417 Autumn 2025

Lecture 25: Using SAT solvers

Glenn Sun

HW8 releasing today

- Problem 15: Solving Killer Sudoku
- Problem 15X.1: Generating Sudoku puzzles
- Problem 15X.2: Writing a Sudoku solver in Java
- Problem 16: True/False/Unknown questions about P and NP

Encoding problems in SAT

SAT (Satisfiability)

literals: variables or their negation

$$a, \neg b, x, \neg x, y$$

clause: OR of literals

$$(a \vee \neg b), (x \vee \neg y \vee z)$$

conjunction normal form (CNF): AND of clauses

$$(a \vee \neg b) \wedge (x \vee \neg y \vee z)$$

SAT (Satisfiability)

Input: A CNF formula $f(x_1, \dots, x_n)$ (equivalently a set of clauses)

Goal: Does there exist x_1, \dots, x_n such that $f(x_1, \dots, x_n)$ is true?

SAT is NP-hard: We don't believe we can solve it quickly in general.

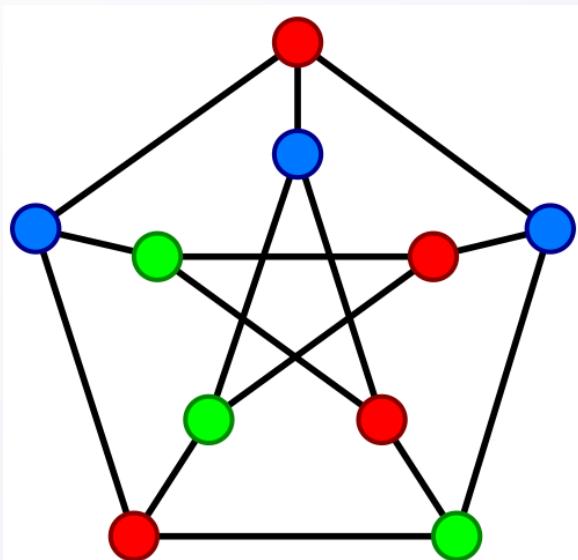
However, in the last ~20 years, we've gotten very good!

Real world problems with >1,000,000 variables/clauses are OK!

3-coloring

Input: An undirected graph with vertices V and edges E

Goal: Can you color the vertices using k colors so that no edge has the same color on both ends?



Coloring is an NP-hard problem,
even with only $k = 3$ colors.

3-coloring

Define variables r_v, g_v, b_v for every vertex v .

- Every vertex gets a color.

$(r_v \vee g_v \vee b_v)$ for every vertex v

- No vertex gets two colors.

$(\neg r_v \vee \neg g_v), (\neg g_v \vee \neg b_v), (\neg b_v \vee \neg r_v)$ for every vertex v

- No edge has the same color on its two vertices.

$(\neg r_u \vee \neg r_v), (\neg g_u \vee \neg g_v), (\neg b_u \vee \neg b_v)$ for every edge (u, v)

What SAT solvers are available?

Z3	Giant software package that provides a neat interface for SAT solvers and other solvers. Can use with Python, C++, Java, JavaScript, etc. Often default choice for real-world work.
CaDiCaL, Kissat, Glucose	Pure SAT solvers written in C++
Sat4J (on your homework)	Pure SAT solver written in Java

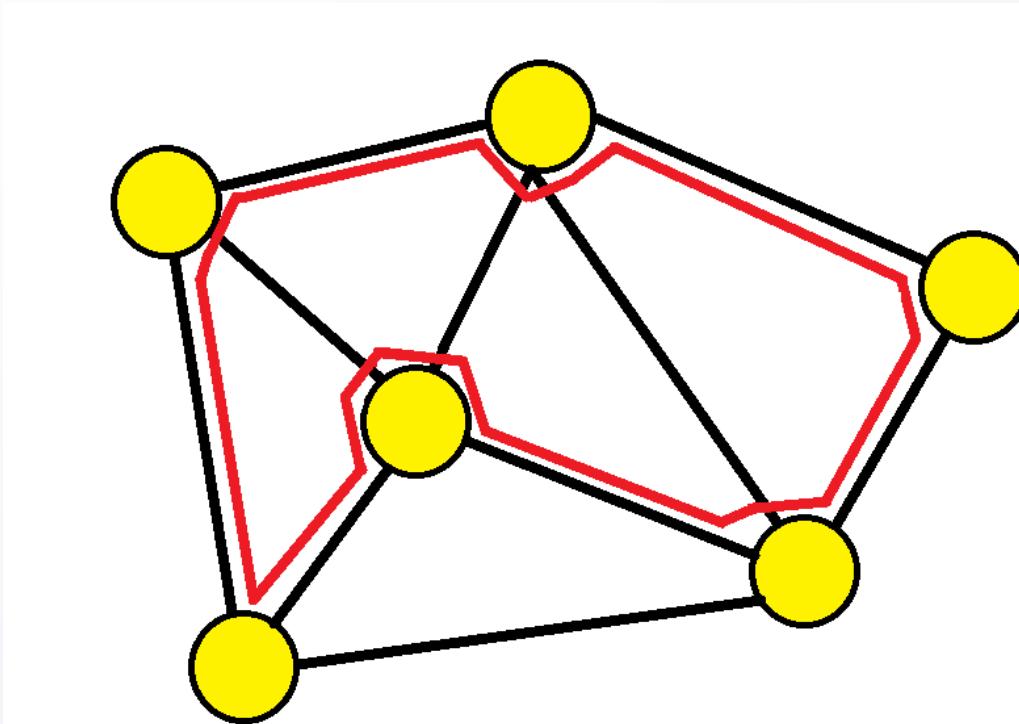
An example with Sat4J

```
public static int[] solve3Coloring(int n, List<int[]> E) {  
    ISolver s = SolverFactory.newDefault();  
    s.newVar(n*3);  
    for (int v=0; v<n; v++) {  
        int r=v*3+1, g=v*3+2, b=v*3+3;  
        s.addClause(new VecInt(new int[]{ r, g, b }));  
        s.addClause(new VecInt(new int[]{ -r, -g }));  
        s.addClause(new VecInt(new int[]{ -g, -b }));  
        s.addClause(new VecInt(new int[]{ -b, -r }));  
    }  
    // Similar things for the edge constraints  
    return s.isSatisfiable() ? s.model() : null;  
}
```

Hamiltonian path

Input: An undirected graph with vertices V and edges E

Goal: Is there a path that uses every vertex exactly once?



Also an NP-hard problem!

Hamiltonian path

Define $p_{v,i}$ to mean “vertex v is the i th vertex on the path”.

- Each vertex appears exactly once on the path.
- Each position on the path has exactly one vertex.
- If two vertices are adjacent on the path, then there is an edge between them.

“Exactly one” constraints

We just saw this with “every vertex gets exactly one color”!

To translate “exactly one of x_1, x_2, \dots, x_n ”:

- At least one of x_1, x_2, \dots, x_n :

$$(x_1 \vee x_2 \vee \cdots \vee x_n)$$

- No two of x_1, x_2, \dots, x_n :

$$(\neg x_i \vee \neg x_j) \text{ for every pair of } i \text{ and } j$$

Requires $O(n^2)$ constraints.

Hamiltonian path

Define $p_{v,i}$ to mean “vertex v is the i th vertex on the path”.

- Each vertex appears exactly once on the path.
“exactly one of $p_{v,1}, \dots, p_{v,n}$ ” for every vertex v
- Each position on the path has exactly one vertex.
“exactly one of $p_{1,i}, \dots, p_{n,i}$ ” for every position i

Hamiltonian path

Define $p_{v, i}$ to mean “vertex v is the i th vertex on the path”.

- If two vertices are adjacent on the path, then there is an edge between them.

$$(p_{u, i} \wedge p_{v, i+1}) \Rightarrow "(u, v) \text{ is an edge}"$$

Use contrapositive: whenever (u, v) is not an edge, include clause

$$(\neg p_{u, i} \vee \neg p_{v, i+1})$$

Defining helper variables

Witches puzzle

- Alice: Bob is a witch. I am not a witch.
- Bob: Carol is as honest as I am.
- Carol: The number of witches is even. Alice and I are not both truthtellers. I am not a witch.
- David: At least one of the witches is a truthteller.

Witches puzzle

From the reading, Alice's statement that "Bob is a witch. I am not a witch." can be written as:

$$\mathbf{ATrue} \Leftrightarrow (\mathbf{BWitch} \wedge \neg \mathbf{AWitch})$$

Which converts to the clauses:

$$(\mathbf{BWitch} \vee \neg \mathbf{ATrue})$$

$$(\neg \mathbf{AWitch} \vee \neg \mathbf{ATrue})$$

$$(\mathbf{ATrue} \vee \neg \mathbf{BWitch} \vee \mathbf{AWitch})$$

Witches puzzle

When Carol says, “The number of witches is even”, this converts to:

$$\text{EvenWitches} = \neg(\text{AWitch} \oplus \text{BWitch} \oplus \text{CWitch} \oplus \text{DWitch})$$

How would this convert to CNF?

Long XORs

The statement $x_1 \oplus x_2 \oplus \dots \oplus x_n$ takes 2^{n-1} clauses to convert!

$$x_1 \oplus x_2 = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

$$\begin{aligned} x_1 \oplus x_2 \oplus x_3 &= (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge \\ &\quad (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \end{aligned}$$

8 clauses for $x_1 \oplus x_2 \oplus x_3 \oplus x_4$, etc.

Tseitin transformations

To translate such Boolean sentences efficiently, introduce helper variables! For $x_1 \oplus x_2 \oplus \dots \oplus x_n$, let

- $z_2 \Leftrightarrow x_1 \oplus x_2$
 - $z_3 \Leftrightarrow z_2 \oplus x_3$
 - ...
 - $z_n \Leftrightarrow z_{n-1} \oplus x_n$
- Each $a \Leftrightarrow b \oplus c$ takes 4 clauses to convert to CNF (today's concept check).
- Use $n - 1$ new variables and represent this sentence in $O(n)$ clauses!

Tseitin transformations

For *any* boolean operation R (could be XOR, AND, OR, etc.),

$$a \Leftrightarrow b R c$$

takes at most 8 clauses to convert to CNF (since there are only 8 possible clauses at all with 3 variables).

Doing this is called a **Tseitin transformation**.

Program verification

What is program verification?

Input: A computer program written in some language and a formal specification

Goal: Does the program meet the spec for all inputs?

What is program verification?

```
division(int x, int y) {  
    int r = x;  
    int q = 0;
```

```
    while (r >= y) {  
        r = r - y;  
        q++;  
    }
```

```
    assert x == y * q + r;
```

```
    assert r >= 0 && r < Math.abs(y); ] spec
```

code

spec

Program verification is too hard

```
hello(int n) {  
    String hello = "Hello world!";  
  
    if (n is a counterexample to the  
        Collatz conjecture) {  
        hello = "Bye world";  
    }  
    assert hello = "Hello world!";
```

major open
problem in number
theory

Bounded model checking (BMC)

Two ideas to constrain the problem:

- Assume all data has finite, fixed size (e.g. ints have 32 bits, or maybe even 4 bits)
- Assume all loops terminate in a small number of iterations

Key assumption: Most software bugs can occur even when inputs are quite small!

Loop unrolling

```
division(int x, int y) {  
    int r = x;  
    int q = 0;  
  
    if (r >= y) { r = r - y; q++; }  
    if (r >= y) { r = r - y; q++; }  
    assume r < y;  
  
    assert x == y * q + r;  
    assert r >= 0 && r < Math.abs(y);
```

This means: all future assertions get “or $r \geq y$ ”

Single static assignment

```
division(int x, int y) {      division(int x, int y) {  
    int r = x;                  int r1 = x;  
    int q = 0;                  int q1 = 0;  
  
    if (r >= y) {              bool b1 = r1 >= y;  
        r = r - y;            int r2 = r1 - y;  
        q++;                  int q2 = q1 + 1;  
    }  
    int r3 = b1 ? r2 : r1;  
    int q3 = b1 ? q2 : q1;  
}
```

x ? y : z means “if x, then y, else z”



What we have now

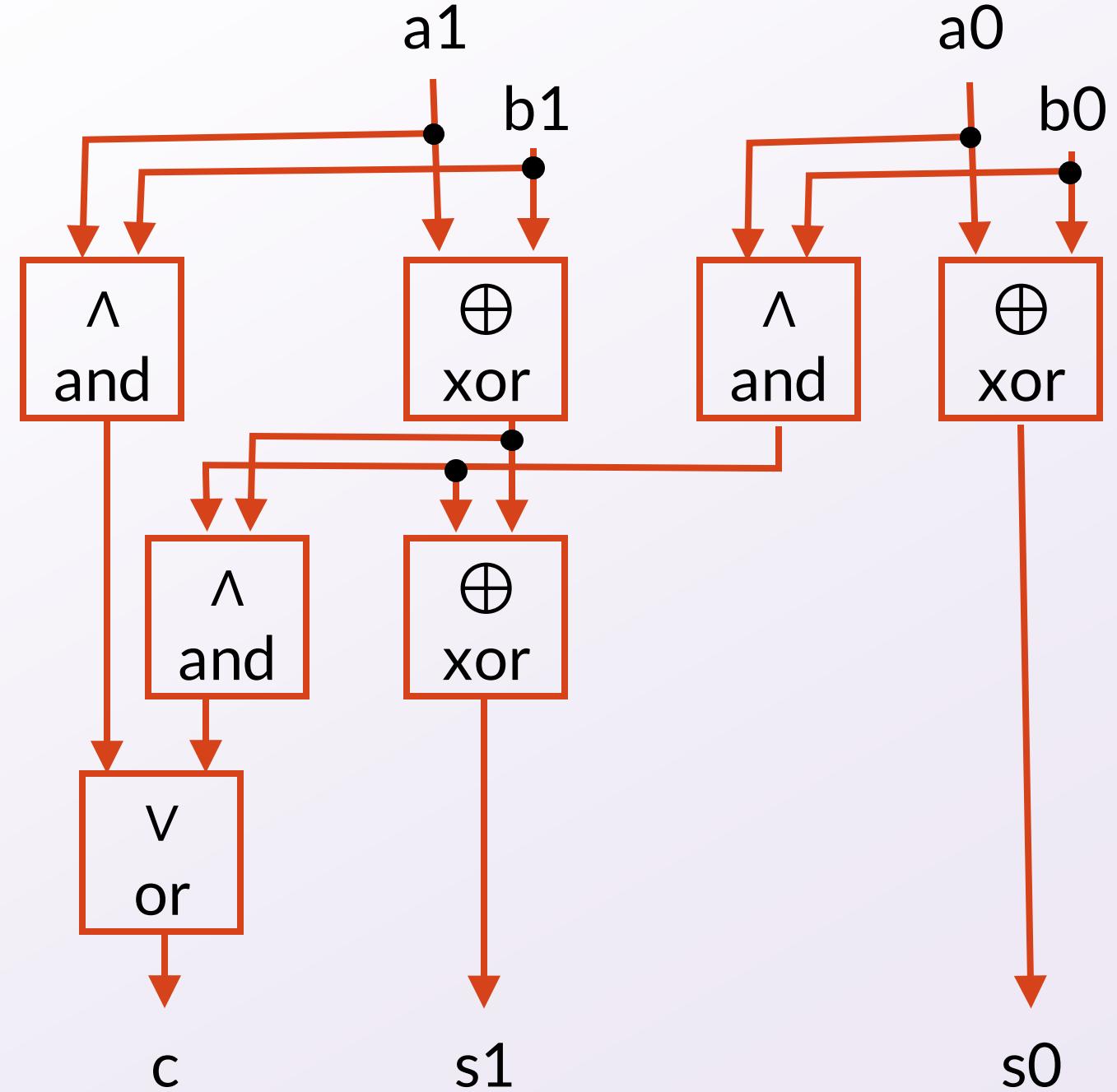
Our program now consists only of:

- Basic functions like `+`, `>`, and `_ ? _ : _` 
 - Assignment to variables 
 - “Assume” statements 
 - “Assert” statements
- Kind of like OR,
AND, XOR, etc?
- Kind of like Tseitin transformations,
except not just bools!
- CNF constraints!

Circuits

In a class on digital design (CSE 369/EE 271), you would learn how to implement all these basic functions with circuits!

$$\begin{array}{r} & \text{a1} & \text{a0} \\ + & \text{b1} & \text{b0} \\ \hline \text{c} & \text{s1} & \text{s0} \end{array}$$



Some things you can do with circuits

Addition of n -bit binary numbers	$O(n)$ gates
Multiplication of n -bit binary numbers	$O(n^2)$ gates with simple implementation, improvable
Comparison of n -bit numbers	$O(n)$ gates
If-then-else for n -bit numbers	$O(n)$ gates
Anything that you can compute on a computer in T time	at most $O(T \log T)$ gates

By Tseitin transformations: # new clauses \approx # new variables = # gates!

Complete workflow for BMC

1. Unroll loops to a fixed depth
2. Convert to single static assignment form
3. Build circuits to represent your code and assertions
4. **Convert circuits to clauses** via Tseitin transformation
5. Add a clause to say that **at least one assertion is false**
6. If a SAT solver says “satisfiable”, you have a counterexample!

Final reminders

HW6 (Greedy) resubmissions close tonight @ 11:59pm!

HW7 (Flows) due tonight @ 11:59pm!

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 214 if you're coming later

Nathan has online OH 12–1pm:

- <https://washington.zoom.us/my/nathanbrunelle>