CS498: Algorithmic Engineering

Lecture 17: Introduction to SAT and Z3

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 10 – 03/24/2026

Outline

Where We Are in the Course

Part I (Lec. 1–9):

- LP, IP, B&B.
- Row generation, lazy cuts.
- **Key:** discrete $\Rightarrow$ exact solvers.

Where We Are in the Course

Part I (Lec. 1–9):

- LP, IP, B&B.
- Row generation, lazy cuts.
- **Key:** discrete $\Rightarrow$ exact solvers.

Part II (Lec. 10–16):

- GD, PyTorch, constrained opt.
- Lagrangian, KKT.
- SA, GA.
- **Key:** continuous, convex, and non-convex.

Where We Are in the Course

Part I (Lec. 1–9):

- LP, IP, B&B.
- Row generation, lazy cuts.
- **Key:** discrete $\Rightarrow$ exact solvers.

Part II (Lec. 10–16):

- GD, PyTorch, constrained opt.
- Lagrangian, KKT.
- SA, GA.
- **Key:** continuous, convex, and non-convex.

Part III (Lec. 17–21):

- SAT, SMT.
- Z3 solver.
- **Key:** logical constraints $\Rightarrow$ feasibility.

Where We Are in the Course

Part I (Lec. 1–9):

- LP, IP, B&B.
- Row generation, lazy cuts.
- **Key:** discrete $\Rightarrow$ exact solvers.

Part II (Lec. 10–16):

- GD, PyTorch, constrained opt.
- Lagrangian, KKT.
- SA, GA.
- **Key:** continuous, convex, and non-convex.

Part III (Lec. 17–21):

- SAT, SMT.
- Z3 solver.
- **Key:** logical constraints $\Rightarrow$ feasibility.

Today: Part III begins. A new way of thinking about problems.

The Shift: From Optimization to Satisfiability

In Parts I and II, we always had an **objective function**:

- Minimize cost, maximize profit, minimize loss.

The Shift: From Optimization to Satisfiability

In Parts I and II, we always had an **objective function**:

- Minimize cost, maximize profit, minimize loss.
- The solver searched for the *best* feasible solution.

The Shift: From Optimization to Satisfiability

In Parts I and II, we always had an **objective function**:

- Minimize cost, maximize profit, minimize loss.
- The solver searched for the *best* feasible solution.

Today, we drop the objective entirely.

The Shift: From Optimization to Satisfiability

In Parts I and II, we always had an **objective function**:

- Minimize cost, maximize profit, minimize loss.
- The solver searched for the *best* feasible solution.

Today, we drop the objective entirely.

The new question is simpler:

$\boxed{\text{Is there } \textit{any} \text{ assignment that satisfies ALL the constraints?}}$

The Shift: From Optimization to Satisfiability

In Parts I and II, we always had an **objective function**:

- Minimize cost, maximize profit, minimize loss.
- The solver searched for the *best* feasible solution.

Today, we drop the objective entirely.

The new question is simpler:

$$\boxed{\text{Is there } \textit{any} \text{ assignment that satisfies ALL the constraints?}}$$

No “best.” No “minimize.” Just: *does a solution exist?*

The Shift: From Optimization to Satisfiability

In Parts I and II, we always had an **objective function**:

- Minimize cost, maximize profit, minimize loss.
- The solver searched for the *best* feasible solution.

Today, we drop the objective entirely.

The new question is simpler:

$\boxed{\text{Is there } \textit{any} \text{ assignment that satisfies ALL the constraints?}}$

No “best.” No “minimize.” Just: *does a solution exist?*

This is the world of **satisfiability**.

Why Should We Care?

Many real problems are naturally about feasibility, not optimization:

Why Should We Care?

Many real problems are naturally about feasibility, not optimization:

- **Hardware verification:** “Can this circuit ever produce the wrong output?”

Why Should We Care?

Many real problems are naturally about feasibility, not optimization:

- **Hardware verification:** “Can this circuit ever produce the wrong output?”
- **Software testing:** “Is there an input that crashes this function?”

Why Should We Care?

Many real problems are naturally about feasibility, not optimization:

- **Hardware verification:** "Can this circuit ever produce the wrong output?"
- **Software testing:** "Is there an input that crashes this function?"
- **Scheduling:** "Can we assign all courses to rooms and times with no conflicts?"

Why Should We Care?

Many real problems are naturally about feasibility, not optimization:

- **Hardware verification:** “Can this circuit ever produce the wrong output?”
- **Software testing:** “Is there an input that crashes this function?”
- **Scheduling:** “Can we assign all courses to rooms and times with no conflicts?”
- **Puzzle solving:** “Does this Sudoku have a solution?”

Why Should We Care?

Many real problems are naturally about feasibility, not optimization:

- **Hardware verification:** “Can this circuit ever produce the wrong output?”
- **Software testing:** “Is there an input that crashes this function?”
- **Scheduling:** “Can we assign all courses to rooms and times with no conflicts?”
- **Puzzle solving:** “Does this Sudoku have a solution?”
- **Proof Verification:** “Is there a counterexample to this theorem?”

Why Should We Care?

Many real problems are naturally about feasibility, not optimization:

- **Hardware verification:** “Can this circuit ever produce the wrong output?”
- **Software testing:** “Is there an input that crashes this function?”
- **Scheduling:** “Can we assign all courses to rooms and times with no conflicts?”
- **Puzzle solving:** “Does this Sudoku have a solution?”
- **Proof Verification:** “Is there a counterexample to this theorem?”

Observation: in all of these, the answer is yes or no. If yes, we want a witness. If no, we want a proof. There is nothing to optimize.

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

1. If Alice goes, then Bob does **not** go.

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

1. If Alice goes, then Bob does **not** go.
2. If Bob does not go, then Carol **must** go.

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

1. If Alice goes, then Bob does **not** go.
2. If Bob does not go, then Carol **must** go.
3. Alice or Bob must go (at least one).

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

1. If Alice goes, then Bob does **not** go.
2. If Bob does not go, then Carol **must** go.
3. Alice or Bob must go (at least one).
4. Carol and Alice cannot **both** go.

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

1. If Alice goes, then Bob does **not** go.
2. If Bob does not go, then Carol **must** go.
3. Alice or Bob must go (at least one).
4. Carol and Alice cannot **both** go.

The question is: can we find an assignment (who goes, who stays) that satisfies all four rules simultaneously?

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

1. If Alice goes, then Bob does **not** go.
2. If Bob does not go, then Carol **must** go.
3. Alice or Bob must go (at least one).
4. Carol and Alice cannot **both** go.

The question is: can we find an assignment (who goes, who stays) that satisfies all four rules simultaneously?

Try it by hand. It is doable with 3 variables and 4 constraints. But what if there were 1,000 friends and 10,000 rules?

A Motivating Puzzle

Three friends are deciding whether to go to a party. Their preferences:

1. If Alice goes, then Bob does **not** go.
2. If Bob does not go, then Carol **must** go.
3. Alice or Bob must go (at least one).
4. Carol and Alice cannot **both** go.

The question is: can we find an assignment (who goes, who stays) that satisfies all four rules simultaneously?

Try it by hand. It is doable with 3 variables and 4 constraints. But what if there were 1,000 friends and 10,000 rules?

We need a systematic framework. That framework is **propositional logic** and **SAT**.

Propositional Logic: The Basics

A **propositional variable** is either True or False. Nothing else.

Propositional Logic: The Basics

A **propositional variable** is either True or False. Nothing else.

We combine variables using **logical connectives**:

Propositional Logic: The Basics

A **propositional variable** is either True or False. Nothing else.

We combine variables using **logical connectives**:

Name	**Symbol**	**Meaning**
AND	$\wedge$	Both must be true
OR	$\vee$	At least one must be true
NOT	$\neg$	Flip the value
Implication	$\rightarrow$	“If . . . then . . . ”

Propositional Logic: The Basics

A **propositional variable** is either True or False. Nothing else.

We combine variables using **logical connectives**:

Name	**Symbol**	**Meaning**
AND	$\wedge$	Both must be true
OR	$\vee$	At least one must be true
NOT	$\neg$	Flip the value
Implication	$\rightarrow$	"If . . . then . . . "

Example: let a = "Alice goes," b = "Bob goes," c = "Carol goes."

Propositional Logic: The Basics

A **propositional variable** is either True or False. Nothing else.

We combine variables using **logical connectives**:

Name	**Symbol**	**Meaning**
AND	$\wedge$	Both must be true
OR	$\vee$	At least one must be true
NOT	$\neg$	Flip the value
Implication	$\rightarrow$	"If . . . then . . . "

Example: let a = "Alice goes," b = "Bob goes," c = "Carol goes."
"If Alice goes, then Bob does not go" becomes:

$$a \rightarrow \neg b$$

Implication Is Just a Disguised OR

$$\boxed{p \to q \;\equiv\; \neg p \lor q}$$

Implication Is Just a Disguised OR

$$\boxed{p \to q \;\equiv\; \neg p \vee q}$$

In words: “if p then q” is the same as “either p is false, or q is true.”

Implication Is Just a Disguised OR

$$\boxed{p \rightarrow q \;\equiv\; \neg p \vee q}$$

In words: "if *p* then *q*" is the same as "either *p* is false, or *q* is true."

Let's translate all four party rules:

Implication Is Just a Disguised OR

$$\boxed{p \to q \;\equiv\; \neg p \vee q}$$

In words: "if p then q" is the same as "either p is false, or q is true."

Let's translate all four party rules:

1. $a \to \neg b$ becomes $\neg a \vee \neg b$

Implication Is Just a Disguised OR

$$\boxed{p \to q \;\equiv\; \neg p \vee q}$$

In words: “if p then q” is the same as “either p is false, or q is true.”

Let’s translate all four party rules:

1. $a \to \neg b$ becomes $\neg a \vee \neg b$
2. $\neg b \to c$ becomes $(\neg\neg b) \vee c \equiv b \vee c$

Implication Is Just a Disguised OR

$$\boxed{p \to q \;\equiv\; \neg p \vee q}$$

In words: “if p then q” is the same as “either p is false, or q is true.”

Let’s translate all four party rules:

1. $a \to \neg b$ becomes $\neg a \vee \neg b$
2. $\neg b \to c$ becomes $(\neg\neg b) \vee c \equiv b \vee c$
3. “Alice or Bob”: $a \vee b$

Implication Is Just a Disguised OR

$$\boxed{p \to q \;\equiv\; \neg p \lor q}$$

In words: "if p then q" is the same as "either p is false, or q is true."

Let's translate all four party rules:

1. $a \to \neg b$ becomes $\neg a \lor \neg b$
2. $\neg b \to c$ becomes $(\neg\neg b) \lor c \equiv b \lor c$
3. "Alice or Bob": $a \lor b$
4. "Not both Alice and Carol": $\neg a \lor \neg c$

Implication Is Just a Disguised OR

$$\boxed{p \to q \;\equiv\; \neg p \vee q}$$

In words: "if p then q" is the same as "either p is false, or q is true."

Let's translate all four party rules:

1. $a \to \neg b$ becomes $\neg a \vee \neg b$
2. $\neg b \to c$ becomes $(\neg\neg b) \vee c \equiv b \vee c$
3. "Alice or Bob": $a \vee b$
4. "Not both Alice and Carol": $\neg a \vee \neg c$

Notice: every rule became a disjunction (OR of literals). This is not a coincidence.

CNF: Conjunctive Normal Form

Definitions

A **literal** is a variable or its negation: a, $\neg b$, c.

A **clause** is a disjunction (OR) of literals: $(\neg a \vee \neg b)$.

A formula in **CNF** is a conjunction (AND) of clauses.

CNF: Conjunctive Normal Form

Definitions

A **literal** is a variable or its negation: a, $\neg b$, c.

A **clause** is a disjunction (OR) of literals: $(\neg a \vee \neg b)$.

A formula in **CNF** is a conjunction (AND) of clauses.

In words: CNF is an AND of ORs.

CNF: Conjunctive Normal Form

Definitions

A **literal** is a variable or its negation: a, $\neg b$, c.

A **clause** is a disjunction (OR) of literals: $(\neg a \vee \neg b)$.

A formula in **CNF** is a conjunction (AND) of clauses.

In words: CNF is an AND of ORs.

Our party puzzle in CNF:

$$\underbrace{(\neg a \vee \neg b)}_{\text{rule 1}} \wedge \underbrace{(b \vee c)}_{\text{rule 2}} \wedge \underbrace{(a \vee b)}_{\text{rule 3}} \wedge \underbrace{(\neg a \vee \neg c)}_{\text{rule 4}}$$

CNF: Conjunctive Normal Form

Definitions

A **literal** is a variable or its negation: a, $\neg b$, c.

A **clause** is a disjunction (OR) of literals: $(\neg a \vee \neg b)$.

A formula in **CNF** is a conjunction (AND) of clauses.

In words: CNF is an AND of ORs.

Our party puzzle in CNF:

$$\underbrace{(\neg a \vee \neg b)}_{\text{rule 1}} \wedge \underbrace{(b \vee c)}_{\text{rule 2}} \wedge \underbrace{(a \vee b)}_{\text{rule 3}} \wedge \underbrace{(\neg a \vee \neg c)}_{\text{rule 4}}$$

Key fact: every propositional formula can be converted to an equivalent CNF formula. So CNF is the “universal input format” for SAT solvers.

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b) \wedge (\neg a \vee \neg c)$.

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b) \wedge (\neg a \vee \neg c)$.

Let's try all assignments systematically:

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \lor \neg b) \land (b \lor c) \land (a \lor b) \land (\neg a \lor \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \lor \neg b$	$b \lor c$	$a \lor b$	$\neg a \lor \neg c$	**All?**
T	T	T	F	T	T	F	No

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \lor \neg b) \land (b \lor c) \land (a \lor b) \land (\neg a \lor \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \lor \neg b$	$b \lor c$	$a \lor b$	$\neg a \lor \neg c$	**All?**
T	T	T	F	T	T	F	No
T	T	F	F	T	T	T	No

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b) \wedge (\neg a \vee \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \vee \neg b$	$b \vee c$	$a \vee b$	$\neg a \vee \neg c$	**All?**
T	T	T	F	T	T	F	No
T	T	F	F	T	T	T	No
T	F	T	T	T	T	F	No

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \lor \neg b) \land (b \lor c) \land (a \lor b) \land (\neg a \lor \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \lor \neg b$	$b \lor c$	$a \lor b$	$\neg a \lor \neg c$	**All?**
T	T	T	F	T	T	F	No
T	T	F	F	T	T	T	No
T	F	T	T	T	T	F	No
T	F	F	T	F	T	T	No

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b) \wedge (\neg a \vee \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \vee \neg b$	$b \vee c$	$a \vee b$	$\neg a \vee \neg c$	**All?**
T	T	T	F	T	T	F	No
T	T	F	F	T	T	T	No
T	F	T	T	T	T	F	No
T	F	F	T	F	T	T	No
F	T	T	T	T	T	T	**Yes!**

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b) \wedge (\neg a \vee \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \vee \neg b$	$b \vee c$	$a \vee b$	$\neg a \vee \neg c$	**All?**
T	T	T	F	T	T	F	No
T	T	F	F	T	T	T	No
T	F	T	T	T	T	F	No
T	F	F	T	F	T	T	No
F	T	T	T	T	T	T	**Yes!**
F	T	F	T	T	T	T	**Yes!**

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b) \wedge (\neg a \vee \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \vee \neg b$	$b \vee c$	$a \vee b$	$\neg a \vee \neg c$	**All?**
T	T	T	F	T	T	F	No
T	T	F	F	T	T	T	No
T	F	T	T	T	T	F	No
T	F	F	T	F	T	T	No
F	T	T	T	T	T	T	**Yes!**
F	T	F	T	T	T	T	**Yes!**
F	F	T	T	T	F	T	No
F	F	F	T	F	F	T	No

Solving the Party Puzzle by Hand

Our CNF formula: $(\neg a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b) \wedge (\neg a \vee \neg c)$.

Let's try all assignments systematically:

a	b	c	$\neg a \vee \neg b$	$b \vee c$	$a \vee b$	$\neg a \vee \neg c$	**All?**
T	T	T	F	T	T	F	No
T	T	F	F	T	T	T	No
T	F	T	T	T	T	F	No
T	F	F	T	F	T	T	No
F	T	T	T	T	T	T	**Yes!**
F	T	F	T	T	T	T	**Yes!**
F	F	T	T	T	F	T	No
F	F	F	T	F	F	T	No

Two satisfying assignments. Both require: Alice stays home, Bob goes. Carol can go or not.

The SAT Problem

Boolean Satisfiability (SAT)

Input: a CNF formula with n variables and m clauses.
Question: is there a True/False assignment to all n variables that makes every clause true?
Output: SAT + a satisfying assignment, or UNSAT.

The SAT Problem

Boolean Satisfiability (SAT)

Input: a CNF formula with n variables and m clauses.
Question: is there a True/False assignment to all n variables that makes every clause true?
Output: SAT + a satisfying assignment, or UNSAT.

We just solved it by brute force: enumerate all 2^n assignments.

The SAT Problem

Boolean Satisfiability (SAT)

Input: a CNF formula with n variables and m clauses.
Question: is there a True/False assignment to all n variables that makes every clause true?
Output: SAT + a satisfying assignment, or UNSAT.

We just solved it by brute force: enumerate all 2^n assignments.

But: $n = 3$ was fine. $n = 100$ means $2^{100} \approx 10^{30}$ assignments. The sun burns out before you finish.

The SAT Problem

Boolean Satisfiability (SAT)

Input: a CNF formula with n variables and m clauses.
Question: is there a True/False assignment to all n variables that makes every clause true?
Output: SAT + a satisfying assignment, or UNSAT.

We just solved it by brute force: enumerate all 2^n assignments.

But: $n = 3$ was fine. $n = 100$ means $2^{100} \approx 10^{30}$ assignments. The sun burns out before you finish.

The question is: can we do better than brute force?

SAT Is NP-Complete (But Don't Panic)

You learned this in CS 374: SAT was the **first** problem proven NP-complete (Cook-Levin, 1971).

SAT Is NP-Complete (But Don't Panic)

You learned this in CS 374: SAT was the **first** problem proven NP-complete (Cook-Levin, 1971).

In words: every problem in NP can be reduced to SAT. If you could solve SAT in polynomial time, you could solve every NP problem in polynomial time.

SAT Is NP-Complete (But Don't Panic)

You learned this in CS 374: SAT was the **first** problem proven NP-complete (Cook-Levin, 1971).

In words: every problem in NP can be reduced to SAT. If you could solve SAT in polynomial time, you could solve every NP problem in polynomial time.

But: theory vs. practice

NP-completeness is a **worst-case** statement. It says there exist hard instances. It does *not* say that every instance is hard.

SAT Is NP-Complete (But Don't Panic)

You learned this in CS 374: SAT was the **first** problem proven NP-complete (Cook-Levin, 1971).

In words: every problem in NP can be reduced to SAT. If you could solve SAT in polynomial time, you could solve every NP problem in polynomial time.

But: theory vs. practice

NP-completeness is a **worst-case** statement. It says there exist hard instances. It does *not* say that every instance is hard.

Modern SAT solvers routinely handle formulas with **millions** of variables and **tens of millions** of clauses in seconds.

SAT Is NP-Complete (But Don't Panic)

You learned this in CS 374: SAT was the **first** problem proven NP-complete (Cook-Levin, 1971).

In words: every problem in NP can be reduced to SAT. If you could solve SAT in polynomial time, you could solve every NP problem in polynomial time.

But: theory vs. practice

NP-completeness is a **worst-case** statement. It says there exist hard instances. It does *not* say that every instance is hard.

Modern SAT solvers routinely handle formulas with **millions** of variables and **tens of millions** of clauses in seconds.

The gap between worst-case theory and practical performance is *enormous*. Understanding why is one of the great mysteries of computer science. We will peek under the hood in Lecture 18.

SAT vs. IP: A Quick Comparison

You might be thinking: “this looks a lot like integer programming.”

SAT vs. IP: A Quick Comparison

You might be thinking: “this looks a lot like integer programming.”

You’d be right. Let’s compare:

SAT vs. IP: A Quick Comparison

You might be thinking: “this looks a lot like integer programming.”

You’d be right. Let’s compare:

	IP (Part I)	**SAT (Part III)**
Variables	$x_i \in \{0, 1\}$ or $\mathbb{Z}$	$x_i \in \{\mathrm{T}, \mathrm{F}\}$
Constraints	Linear inequalities	Clauses (OR of literals)
Objective	Minimize $c^\top x$	None (feasibility only)
Solver	Gurobi (B&B + LP relax.)	SAT solver (DPLL + CDCL)

SAT vs. IP: A Quick Comparison

You might be thinking: “this looks a lot like integer programming.”

You’d be right. Let’s compare:

	IP (Part I)	SAT (Part III)
Variables	$x_i \in \{0, 1\}$ or $\mathbb{Z}$	$x_i \in \{T, F\}$
Constraints	Linear inequalities	Clauses (OR of literals)
Objective	Minimize $c^\top x$	None (feasibility only)
Solver	Gurobi (B&B + LP relax.)	SAT solver (DPLL + CDCL)

In fact, any SAT instance can be encoded as a 0-1 IP, and vice versa. But SAT solvers exploit the boolean structure in ways that generic IP solvers cannot.

SAT vs. IP: A Quick Comparison

You might be thinking: “this looks a lot like integer programming.”

You’d be right. Let’s compare:

	IP (Part I)	**SAT (Part III)**
Variables	$x_i \in \{0, 1\}$ or $\mathbb{Z}$	$x_i \in \{\mathrm{T}, \mathrm{F}\}$
Constraints	Linear inequalities	Clauses (OR of literals)
Objective	Minimize $c^\top x$	None (feasibility only)
Solver	Gurobi (B&B + LP relax.)	SAT solver (DPLL + CDCL)

In fact, any SAT instance can be encoded as a 0-1 IP, and vice versa. But SAT solvers exploit the boolean structure in ways that generic IP solvers cannot.

Takeaway: SAT is not “better” or “worse” than IP. It is a different language. The right tool depends on what your constraints look like.

What Is Z3?

Z3 is a **satisfiability solver** developed by Microsoft Research.

What Is Z3?

Z3 is a **satisfiability solver** developed by Microsoft Research.

- Handles pure SAT (boolean variables, clauses).

What Is Z3?

Z3 is a **satisfiability solver** developed by Microsoft Research.

- Handles pure SAT (boolean variables, clauses).
- Also handles **SMT**: SAT + integers, reals, arrays, bit-vectors, and more. (We will cover SMT in Lectures 19-21.)

What Is Z3?

Z3 is a **satisfiability solver** developed by Microsoft Research.

- Handles pure SAT (boolean variables, clauses).
- Also handles **SMT**: SAT + integers, reals, arrays, bit-vectors, and more. (We will cover SMT in Lectures 19-21.)
- Has a clean Python API. `pip install z3-solver` (or `brew install z3`).

What Is Z3?

Z3 is a **satisfiability solver** developed by Microsoft Research.

- Handles pure SAT (boolean variables, clauses).
- Also handles **SMT**: SAT + integers, reals, arrays, bit-vectors, and more. (We will cover SMT in Lectures 19-21.)
- Has a clean Python API. `pip install z3-solver` (or `brew install z3`).
- Free, open-source, used in industry (security, verification, compilers).

What Is Z3?

Z3 is a **satisfiability solver** developed by Microsoft Research.

- Handles pure SAT (boolean variables, clauses).
- Also handles **SMT**: SAT + integers, reals, arrays, bit-vectors, and more. (We will cover SMT in Lectures 19-21.)
- Has a clean Python API. `pip install z3-solver` (or `brew install z3`).
- Free, open-source, used in industry (security, verification, compilers).

You declare variables and constraints. Z3 searches for a satisfying assignment. You never write a search algorithm.

Z3 Basics: The Party Puzzle

Let's solve our party puzzle with Z3.

```
from z3 import *                                  #New!!

a = Bool('a')  # Alice goes?
b = Bool('b')  # Bob goes?
c = Bool('c')  # Carol goes?
```

Z3 Basics: The Party Puzzle

Let's solve our party puzzle with Z3.

```
from z3 import *                              #New!!

a = Bool('a')  # Alice goes?
b = Bool('b')  # Bob goes?
c = Bool('c')  # Carol goes?

s = Solver()
s.add(Implies(a, Not(b)))     # rule 1: if Alice then not Bob
s.add(Implies(Not(b), c))     # rule 2: if not Bob then Carol
s.add(Or(a, b))               # rule 3: Alice or Bob
s.add(Or(Not(a), Not(c)))     # rule 4: not both Alice and Carol
```

Z3 Basics: The Party Puzzle

Let's solve our party puzzle with Z3.

```
from z3 import *                            #New!!

a = Bool('a')  # Alice goes?
b = Bool('b')  # Bob goes?
c = Bool('c')  # Carol goes?

s = Solver()
s.add(Implies(a, Not(b)))     # rule 1: if Alice then not Bob
s.add(Implies(Not(b), c))     # rule 2: if not Bob then Carol
s.add(Or(a, b))               # rule 3: Alice or Bob
s.add(Or(Not(a), Not(c)))     # rule 4: not both Alice and Carol

print(s.check())              # sat
print(s.model())              # [a = False, b = True, c = False]
```

Reading Z3's Output

Let's unpack what just happened:

Reading Z3's Output

Let's unpack what just happened:

- `s.check()` returns one of three values:
  - `sat`: a satisfying assignment exists.
  - `unsat`: no assignment can satisfy all constraints.
  - `unknown`: Z3 could not determine (**very** rare for pure SAT).

Reading Z3's Output

Let's unpack what just happened:

- `s.check()` returns one of three values:
  - `sat`: a satisfying assignment exists.
  - `unsat`: no assignment can satisfy all constraints.
  - `unknown`: Z3 could not determine (**very** rare for pure SAT).
- `s.model()` returns the actual assignment (the "witness").

Reading Z3's Output

Let's unpack what just happened:

- `s.check()` returns one of three values:
  - `sat`: a satisfying assignment exists.
  - `unsat`: no assignment can satisfy all constraints.
  - `unknown`: Z3 could not determine (**very** rare for pure SAT).
- `s.model()` returns the actual assignment (the "witness").
- We can query individual values: `s.model()[a]` returns `False`.

Reading Z3's Output

Let's unpack what just happened:

- `s.check()` returns one of three values:
    - `sat`: a satisfying assignment exists.
    - `unsat`: no assignment can satisfy all constraints.
    - `unknown`: Z3 could not determine (**very** rare for pure SAT).
- `s.model()` returns the actual assignment (the "witness").
- We can query individual values: `s.model()[a]` returns `False`.

Key insight: compare this to Gurobi's workflow. In Gurobi: `m.optimize()`, then `x.X` to read values. In Z3: `s.check()`, then `s.model()` to read values.

What If It’s UNSAT?

Add a fifth rule: “Bob must not go.”

```
from z3 import *
set_param(proof=True) #Generate proof for unsatisfiability!

s = Solver()
s.add(Implies(a, Not(b)))      # rule 1
s.add(Implies(Not(b), c))     # rule 2
s.add(Or(a, b))               # rule 3
s.add(Or(Not(a), Not(c)))     # rule 4
s.add(Not(b))                 # rule 5: Bob does NOT go     #New
```

What If It's UNSAT?

Add a fifth rule: "Bob must not go."

```python
from z3 import *
set_param(proof=True) #Generate proof for unsatisfiability!

s = Solver()
s.add(Implies(a, Not(b)))      # rule 1
s.add(Implies(Not(b), c))     # rule 2
s.add(Or(a, b))               # rule 3
s.add(Or(Not(a), Not(c)))     # rule 4
s.add(Not(b))                 # rule 5: Bob does NOT go     #New

print(s.check())   # unsat
if s.check() == unsat:
    print(s.proof())
```

What If It’s UNSAT?

Add a fifth rule: “Bob must not go.”

```
from z3 import *
set_param(proof=True) #Generate proof for unsatisfiability!

s = Solver()
s.add(Implies(a, Not(b)))     # rule 1
s.add(Implies(Not(b), c))    # rule 2
s.add(Or(a, b))              # rule 3
s.add(Or(Not(a), Not(c)))    # rule 4
s.add(Not(b))                # rule 5: Bob does NOT go     #New
print(s.check())   # unsat
if s.check() == unsat:
    print(s.proof())
```

Why? Rule 3 says $a \lor b$. Rule 5 says $\neg b$. So a must be true. But then rule 1 forces $\neg b$ (already satisfied) and rule 2 forces c. Now rule 4 says $\neg a \lor \neg c$, but a and c are both true. Contradiction.

Sudoku: The Poster Child for SAT

Rules of Sudoku:

Sudoku: The Poster Child for SAT

Rules of Sudoku:

1. Fill a 9×9 grid with digits 1–9.

Sudoku: The Poster Child for SAT

Rules of Sudoku:

1. Fill a 9×9 grid with digits 1–9.
2. Each **row** contains all digits 1–9 (no repeats).

Sudoku: The Poster Child for SAT

Rules of Sudoku:

1. Fill a 9×9 grid with digits 1–9.
2. Each **row** contains all digits 1–9 (no repeats).
3. Each **column** contains all digits 1–9.

Sudoku: The Poster Child for SAT

Rules of Sudoku:

1. Fill a 9×9 grid with digits 1–9.
2. Each **row** contains all digits 1–9 (no repeats).
3. Each **column** contains all digits 1–9.
4. Each 3×3 **box** contains all digits 1–9.

Sudoku: The Poster Child for SAT

Rules of Sudoku:

1. Fill a 9×9 grid with digits 1–9.
2. Each **row** contains all digits 1–9 (no repeats).
3. Each **column** contains all digits 1–9.
4. Each 3×3 **box** contains all digits 1–9.
5. Some cells are pre-filled (clues).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku: The Poster Child for SAT

Rules of Sudoku:

1. Fill a 9×9 grid with digits 1–9.
2. Each **row** contains all digits 1–9 (no repeats).
3. Each **column** contains all digits 1–9.
4. Each 3×3 **box** contains all digits 1–9.
5. Some cells are pre-filled (clues).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

No objective function. Pure feasibility. This is a SAT problem in disguise.

Encoding Sudoku in CNF

Variables: for each cell (i, j) and value $v \in \{1, \ldots, 9\}$, a **Boolean** variable:

$$x_{i,j,v} = \begin{cases} \text{True} & \text{if cell } (i,j) \text{ holds value } v \\ \text{False} & \text{otherwise} \end{cases}$$

Encoding Sudoku in CNF

Variables: for each cell (i, j) and value $v \in \{1, \ldots, 9\}$, a **Boolean** variable:

$$x_{i,j,v} = \begin{cases} \text{True} & \text{if cell } (i,j) \text{ holds value } v \\ \text{False} & \text{otherwise} \end{cases}$$

That is $9 \times 9 \times 9 = 729$ Boolean variables. Now we need clauses:

Encoding Sudoku in CNF

Variables: for each cell (i, j) and value $v \in \{1, \ldots, 9\}$, a **Boolean** variable:

$$x_{i,j,v} = \begin{cases} \text{True} & \text{if cell } (i, j) \text{ holds value } v \\ \text{False} & \text{otherwise} \end{cases}$$

That is $9 \times 9 \times 9 = 729$ Boolean variables. Now we need clauses:

- **Exactly one value per cell:**
  - At least one: $(x_{i,j,1} \vee x_{i,j,2} \vee \cdots \vee x_{i,j,9})$ for all i, j.
  - At most one: $(\neg x_{i,j,v_1} \vee \neg x_{i,j,v_2})$ for every pair $v_1 \neq v_2$

Encoding Sudoku in CNF

Variables: for each cell (i, j) and value $v \in \{1, \ldots, 9\}$, a **Boolean** variable:

$$x_{i,j,v} = \begin{cases} \text{True} & \text{if cell } (i,j) \text{ holds value } v \\ \text{False} & \text{otherwise} \end{cases}$$

That is $9 \times 9 \times 9 = 729$ Boolean variables. Now we need clauses:

- **Exactly one value per cell:**
  - At least one: $(x_{i,j,1} \vee x_{i,j,2} \vee \cdots \vee x_{i,j,9})$ for all i, j.
  - At most one: $(\neg x_{i,j,v_1} \vee \neg x_{i,j,v_2})$ for every pair $v_1 \neq v_2$
- **Each value once per row, column, and 3×3 box:** same “exactly one” pattern.

Encoding Sudoku in CNF

Variables: for each cell (i, j) and value $v \in \{1, \dots, 9\}$, a **Boolean** variable:

$$x_{i,j,v} = \begin{cases} \text{True} & \text{if cell } (i,j) \text{ holds value } v \\ \text{False} & \text{otherwise} \end{cases}$$

That is $9 \times 9 \times 9 = 729$ Boolean variables. Now we need clauses:

- **Exactly one value per cell:**
  - At least one: $(x_{i,j,1} \vee x_{i,j,2} \vee \cdots \vee x_{i,j,9})$ for all i, j.
  - At most one: $(\neg x_{i,j,v_1} \vee \neg x_{i,j,v_2})$ for every pair $v_1 \neq v_2$
- **Each value once per row, column, and 3×3 box:** same “exactly one” pattern.
- **Clues:** if cell (i, j) is pre-filled with v, add the unit clause $(x_{i,j,v})$.

Encoding Sudoku in CNF

Variables: for each cell (i, j) and value $v \in \{1, \ldots, 9\}$, a **Boolean** variable:

$$x_{i,j,v} = \begin{cases} \text{True} & \text{if cell } (i,j) \text{ holds value } v \\ \text{False} & \text{otherwise} \end{cases}$$

That is $9 \times 9 \times 9 = 729$ Boolean variables. Now we need clauses:

- **Exactly one value per cell:**
  - At least one: $(x_{i,j,1} \vee x_{i,j,2} \vee \cdots \vee x_{i,j,9})$ for all i, j.
  - At most one: $(\neg x_{i,j,v_1} \vee \neg x_{i,j,v_2})$ for every pair $v_1 \neq v_2$
- **Each value once per row, column, and 3×3 box:** same "exactly one" pattern.
- **Clues:** if cell (i, j) is pre-filled with v, add the unit clause $(x_{i,j,v})$.

Every constraint is an OR of literals. This is **pure CNF**: ∼12,000 clauses, each with at most 9 literals.

Sudoku in Z3: Variables and the “Exactly One” Pattern

```
from z3 import *

# x[i][j][v] = True iff cell (i,j) holds value v+1
X = [[[Bool(f'x_{i}_{j}_{v}') for v in range(9)]
      for j in range(9)] for i in range(9)]
s = Solver()
```

Sudoku in Z3: Variables and the “Exactly One” Pattern

```python
from z3 import *

# x[i][j][v] = True iff cell (i,j) holds value v+1
X = [[[Bool(f'x_{i}_{j}_{v}') for v in range(9)]
      for j in range(9)] for i in range(9)]
s = Solver()
def exactly_one(bools):
    """Encode 'exactly one is True' as pure CNF clauses."""
    s.add(Or(bools))                          # at-least-one clause
    for a in range(len(bools)):               # pairwise at-most-one
        for b in range(a+1, len(bools)):
            s.add(Or(Not(bools[a]), Not(bools[b])))
```

Sudoku in Z3: Variables and the “Exactly One” Pattern

```python
from z3 import *

# x[i][j][v] = True iff cell (i,j) holds value v+1
X = [[[Bool(f'x_{i}_{j}_{v}') for v in range(9)]
      for j in range(9)] for i in range(9)]
s = Solver()
def exactly_one(bools):
    """Encode 'exactly one is True' as pure CNF clauses."""
    s.add(Or(bools))                            # at-least-one clause
    for a in range(len(bools)):                 # pairwise at-most-one
        for b in range(a+1, len(bools)):
            s.add(Or(Not(bools[a]), Not(bools[b])))
# Each cell holds exactly one value
for i in range(9):
    for j in range(9):
        exactly_one([X[i][j][v] for v in range(9)])
```

Sudoku in Z3: Rows, Columns, Boxes, and Clues

```python
# Each value appears exactly once per row
for i in range(9):
    for v in range(9):
        exactly_one([X[i][j][v] for j in range(9)])

# Each value appears exactly once per column
for j in range(9):
    for v in range(9):
        exactly_one([X[i][j][v] for i in range(9)])
```

Sudoku in Z3: Rows, Columns, Boxes, and Clues

```
# Each value appears exactly once per row
for i in range(9):
    for v in range(9):
        exactly_one([X[i][j][v] for j in range(9)])

# Each value appears exactly once per column
for j in range(9):
    for v in range(9):
        exactly_one([X[i][j][v] for i in range(9)])

# Each value appears exactly once per 3x3 box
for bi in range(3):
    for bj in range(3):
        for v in range(9):
            box = [X[3*bi+di][3*bj+dj][v]
                   for di in range(3) for dj in range(3)]
            exactly_one(box)
```

Sudoku in Z3: Rows, Columns, Boxes, and Clues

```
# Each value appears exactly once per row
for i in range(9):
    for v in range(9):
        exactly_one([X[i][j][v] for j in range(9)])

# Each value appears exactly once per column
for j in range(9):
    for v in range(9):
        exactly_one([X[i][j][v] for i in range(9)])

# Each value appears exactly once per 3x3 box
for bi in range(3):
    for bj in range(3):
        for v in range(9):
            box = [X[3*bi+di][3*bj+dj][v]
                   for di in range(3) for dj in range(3)]
            exactly_one(box)

# Clues: assert the known value directly (unit clause)
clues = [(0,0,5),(0,1,3),(0,4,7),(1,0,6),(1,3,1),(1,4,9),(1,5,5), ...]
for (i, j, v) in clues:
    s.add(X[i][j][v-1])          # just one literal
```

Sudoku in Z3: Solving and Reading the Solution

```
if s.check() == sat:
    m = s.model()
    for i in range(9):
        row = [v+1 for j in range(9) for v in range(9)
               if is_true(m[X[i][j][v]])]
        print(' '.join(str(d) for d in row))
```

Sudoku in Z3: Solving and Reading the Solution

```python
if s.check() == sat:
    m = s.model()
    for i in range(9):
        row = [v+1 for j in range(9) for v in range(9)
               if is_true(m[X[i][j][v]])]
        print(' '.join(str(d) for d in row))
```

Output:

```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

Sudoku in Z3: Solving and Reading the Solution

```
if s.check() == sat:
    m = s.model()
    for i in range(9):
        row = [v+1 for j in range(9) for v in range(9)
               if is_true(m[X[i][j][v]])]
        print(' '.join(str(d) for d in row))
```

Output:

```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

Z3 solves this in around 46 milliseconds. We fed it ∼12,000 clauses over 729 Boolean variables. Trivial for a modern SAT solver.

The N-Queens Problem

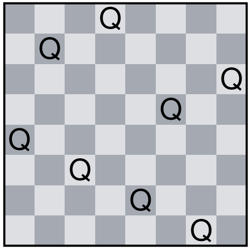
Place n queens on an $n \times n$ chessboard so that no two queens attack each other.

The N-Queens Problem

Place n queens on an $n \times n$ chessboard so that no two queens attack each other. Queens attack along rows, columns, and both diagonals. For $n = 8$:

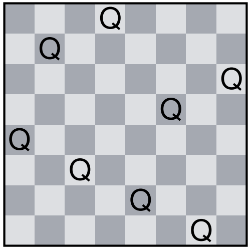
The N-Queens Problem

Place n queens on an $n \times n$ chessboard so that no two queens attack each other. Queens attack along rows, columns, and both diagonals. For $n = 8$:



The N-Queens Problem

Place n queens on an $n \times n$ chessboard so that no two queens attack each other. Queens attack along rows, columns, and both diagonals. For $n = 8$:



For $n = 8$, there are $\binom{64}{8} \approx 4.4$ billion ways to place 8 queens. Only 92 are valid.

N-Queens in Z3: Pure Boolean Encoding

$q_{i,j}$ = True iff there is a queen at row i, column j. That is $n^2 = 64$ Boolean variables.

```
from z3 import *
n = 8
Q = [[Bool(f'q_{i}_{j}') for j in range(n)] for i in range(n)]
s = Solver()
```

N-Queens in Z3: Pure Boolean Encoding

$q_{i,j}$ = True iff there is a queen at row i, column j. That is $n^2 = 64$ Boolean variables.

```
from z3 import *
n = 8
Q = [[Bool(f'q_{i}_{j}') for j in range(n)] for i in range(n)]
s = Solver()
for i in range(n):                                      # exactly one queen per row
    s.add(Or([Q[i][j] for j in range(n)]))              # at-least-one clause
    for j1 in range(n):                                 # pairwise at-most-one
        for j2 in range(j1+1, n):
            s.add(Or(Not(Q[i][j1]), Not(Q[i][j2])))
```

N-Queens in Z3: Pure Boolean Encoding

$q_{i,j}$ = True iff there is a queen at row i, column j. That is $n^2 = 64$ Boolean variables.

```python
from z3 import *
n = 8
Q = [[Bool(f'q_{i}_{j}') for j in range(n)] for i in range(n)]
s = Solver()
for i in range(n):                                  # exactly one queen per row
    s.add(Or([Q[i][j] for j in range(n)]))          # at-least-one clause
    for j1 in range(n):                             # pairwise at-most-one
        for j2 in range(j1+1, n):
            s.add(Or(Not(Q[i][j1]), Not(Q[i][j2])))
for j in range(n):                                  # at most one per column
    for i1 in range(n):
        for i2 in range(i1+1, n):
            s.add(Or(Not(Q[i1][j]), Not(Q[i2][j])))

for i1 in range(n):                                 #at most one per diagonal
    for j1 in range(n):
        for i2 in range(i1+1, n):
            for d in [j1+(i2-i1), j1-(i2-i1)]:     # both diags
                if 0 <= d < n:
                    s.add(Or(Not(Q[i1][j1]), Not(Q[i2][d])))
```

N-Queens: Solving

```python
print(s.check())    # sat
m = s.model()
for i in range(n):
    col = [j for j in range(n) if is_true(m[Q[i][j]])][0]
    print(f"Row {i}: column {col}")
# Row 0: column 3, Row 1: column 6, ...
```

N-Queens: Scaling Up

n	**Z3 solving time**
8	$<$ 0.01 seconds
20	$\sim$ 0.01 seconds
50	$\sim$ 1 seconds
100	$\sim$ 10 second

N-Queens: Scaling Up

n	**Z3 solving time**
8	$<$ 0.01 seconds
20	$\sim$ 0.01 seconds
50	$\sim$ 1 seconds
100	$\sim$ 10 second

Observation: for pure N-Queens, specialized algorithms (backtracking with constraint propagation) are faster.

N-Queens: Scaling Up

n	**Z3 solving time**
8	$<$ 0.01 seconds
20	$\sim$ 0.01 seconds
50	$\sim$ 1 seconds
100	$\sim$ 10 second

Observation: for pure N-Queens, specialized algorithms (backtracking with constraint propagation) are faster.

But: what if you wanted extra constraints? "Queens in rows 0 and 7 must be in adjacent columns." With Z3, you add two more lines. With a hand-written solver, you rewrite your solver.

N-Queens: Scaling Up

n	**Z3 solving time**
8	$<$ 0.01 seconds
20	$\sim$ 0.01 seconds
50	$\sim$ 1 seconds
100	$\sim$ 10 second

Observation: for pure N-Queens, specialized algorithms (backtracking with constraint propagation) are faster.

But: what if you wanted extra constraints? "Queens in rows 0 and 7 must be in adjacent columns." With Z3, you add two more lines. With a hand-written solver, you rewrite your solver.

Takeaway: Z3 trades raw speed for modeling flexibility. When constraints are complex and evolving, that tradeoff is worth it.

Finding ALL Solutions

Sometimes we don't just want *a* solution, we want *all* of them.

Finding ALL Solutions

Sometimes we don't just want *a* solution, we want *all* of them.

Z3 returns one model at a time. To enumerate, we add a **blocking clause** after each solution:

```
n = 5   # 5-queens (10 solutions exist)
Q = [[Bool(f'q_{i}_{j}') for j in range(n)] for i in range(n)]
s = Solver()
# ... (same row/column/diagonal constraints as before) ...
```

Finding ALL Solutions

Sometimes we don't just want *a* solution, we want *all* of them.

Z3 returns one model at a time. To enumerate, we add a **blocking clause** after each solution:

```
n = 5   # 5-queens (10 solutions exist)
Q = [[Bool(f'q_{i}_{j}') for j in range(n)] for i in range(n)]
s = Solver()
# ... (same row/column/diagonal constraints as before) ...
count = 0
while s.check() == sat:
    m = s.model()
    count += 1
    # Block this placement: negate at least one placed queen
    block = [Not(Q[i][j]) for i in range(n) for j in range(n)
             if is_true(m[Q[i][j]])]
    s.add(Or(block))                      # itself a clause!

print(f"Total solutions: {count}")
```

Finding ALL Solutions

Sometimes we don't just want *a* solution, we want *all* of them.

Z3 returns one model at a time. To enumerate, we add a **blocking clause** after each solution:

```python
n = 5   # 5-queens (10 solutions exist)
Q = [[Bool(f'q_{i}_{j}') for j in range(n)] for i in range(n)]
s = Solver()
# ... (same row/column/diagonal constraints as before) ...
count = 0
while s.check() == sat:
    m = s.model()
    count += 1
    # Block this placement: negate at least one placed queen
    block = [Not(Q[i][j]) for i in range(n) for j in range(n)
             if is_true(m[Q[i][j]])]
    s.add(Or(block))                      # itself a clause!

print(f"Total solutions: {count}")
```

Key idea: the blocking clause is itself pure CNF: an OR of negated literals. We tell Z3 "not that exact placement" and ask again.