SAT Solving

CS560: Reasoning About Programs

Roopsha Samanta

PURDUE UNIVERSITY

Partly based on slides by Isil Dillig, Aarti Gupta and Emina Torlak

Roadmap

Previously

- PL and normal forms

Today

- DPLL algorithm for SAT solving
- One challenge for current SAT solvers
- Variations of the satisfiability problem (e.g., MaxSAT)

Review: Conjunctive Normal Form (CNF)

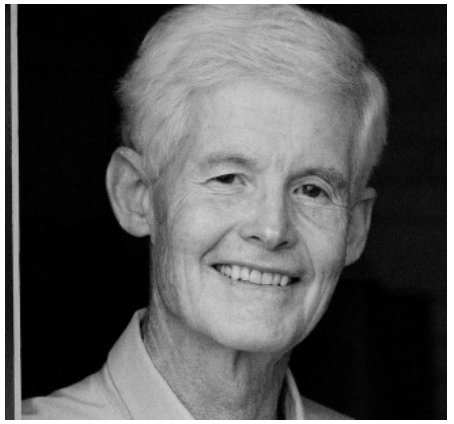
Atom	$\top$, $\bot$, propositional variables
Literal	Atom | $\neg$Atom
Clause	Literal $\vee$ Clause
Formula	Clause $\wedge$ Formula

Conjunction of disjunction of literals

The Boolean Satisfiability problem

A bit of history

Cook



Levin

Karp

The SAT problem

For F in CNF, exists $I : I \vDash F$?

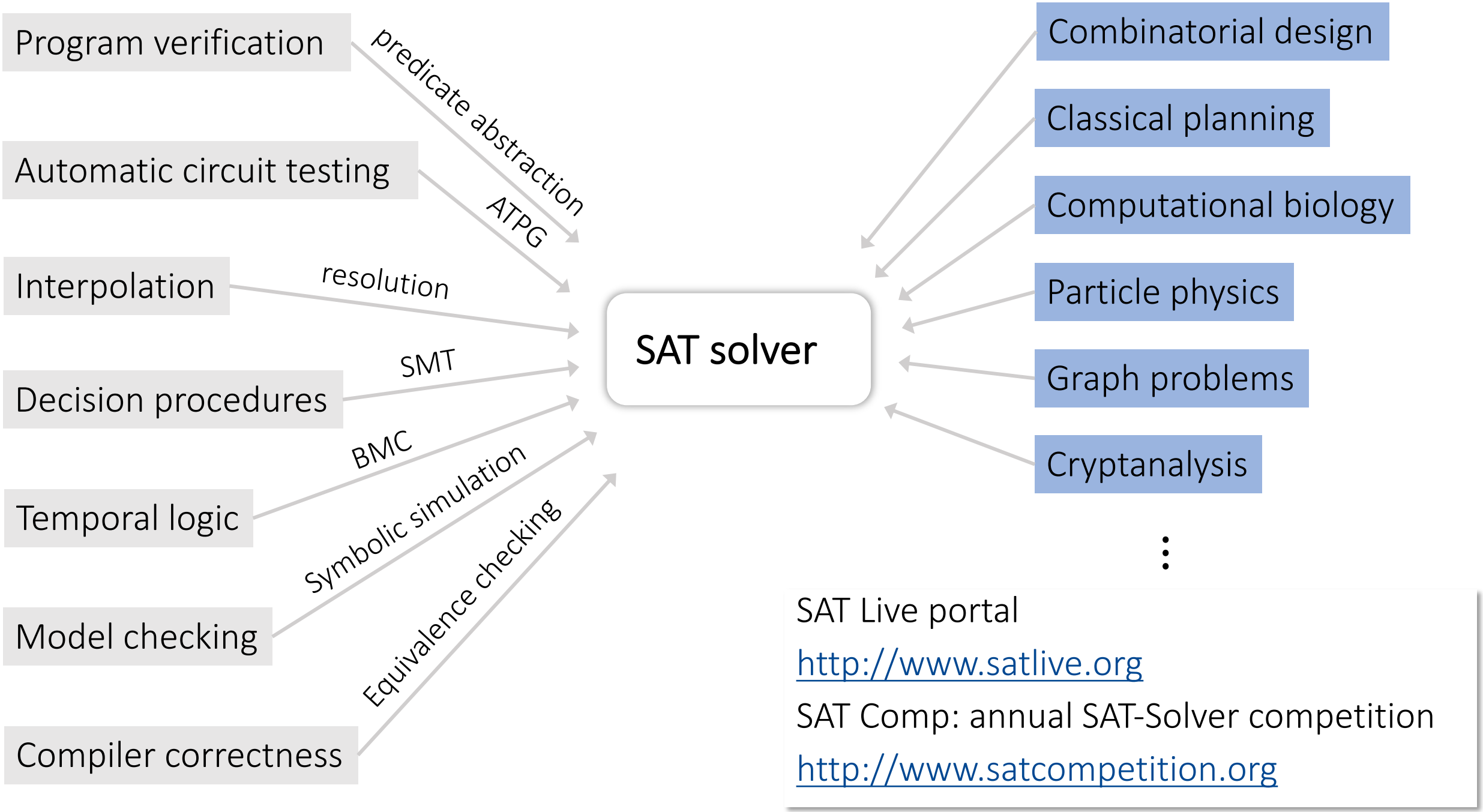
First NP-complete problem!

Cook-Levin Theorem:

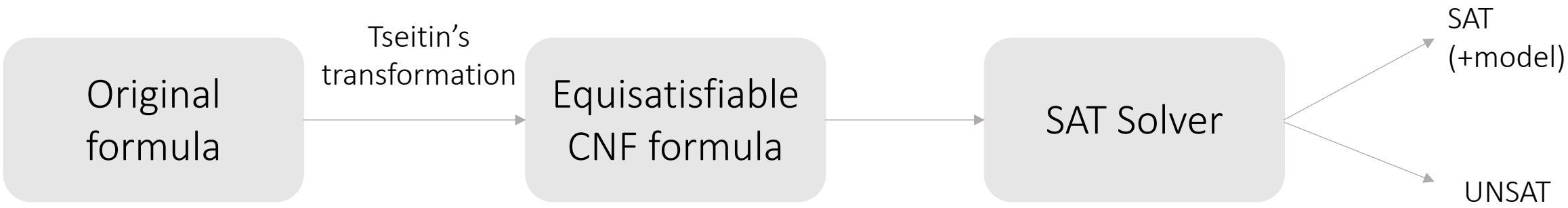
SAT is NP-complete

Cook, *The complexity of theorem proving procedures*, 1971

Karp, *Reducibility among combinatorial problems*, 1972


Program verification
Automatic circuit testing
Interpolation
Decision procedures
Temporal logic
Model checking
Compiler correctness
predicate abstraction
ATPG
resolution
SMT
BMC
Symbolic simulation
Equivalence checking
SAT solver
Combinatorial design
Classical planning
Computational biology
Particle physics
Graph problems
Cryptanalysis
SAT Live portal
http://www.satlive.org
SAT Comp: annual SAT-Solver competition
http://www.satcompetition.org

A Modern SAT Solver



Almost all SAT solvers today are based on DPLL (Davis-Putnam-Logemann-Loveland)

DPLL: A bit of history

1962: the original algorithm known as DP (Davis-Putnam)

⇒ “simple” procedure for automated theorem proving

Davis and Putnam hired two programmers, Logemann and Loveland, to implement their ideas on the IBM 704.

Not all of the original ideas worked out as planned

⇒ refined algorithm is what is known today as **DPLL**

DPLL insight

Two distinct approaches for the Boolean satisfiability problem

- **Search**
  - Find satisfying assignment by searching through all possible assignments
  - Example: truth table
- **Deduction**
  - Deduce new facts from set of known facts, i.e, application of proof rules
  - Example: semantic argument method
- DPLL combines search and deduction in a very effective way!

- Deductive principle underlying DPLL is **propositional resolution**
- Resolution can only be applied to formulas in CNF
- SAT solvers convert formulas to CNF to be able to perform resolution

Propositional Resolution

Consider two clauses in CNF:

$$C_1 : (l_1 \vee \ldots p \ldots \vee\ l_k)$$

$$C_2 : (l_1' \vee \ldots \neg p \ldots \vee\ l_n')$$

We can deduce a new clause C_3, called resolvent:

$$C_3 : (l_1 \vee \ldots \vee\ l_k \vee\ l_1' \vee \ldots\ldots \vee\ l_n')$$

Correctness:

1. If p is assigned T : since C_1 is SAT and since $\neg p$ is $\bot$, $(l_1' \vee \ldots\ldots \vee\ l_n')$ must be true
2. If p is assigned $\bot$: since C_2 is SAT and since p is $\bot$, $(l_1 \vee \ldots\ldots \vee\ l_k)$ must be true
3. Thus, C_3 must be true

Unit Resolution

Consider two clauses in CNF:

$$C_1 : \ p$$

$$C_2 : \ (l_1 \vee \ldots \neg p \ldots \vee l_n)$$

We can deduce a new resolvent:

$$C_3 : \ (l_1 \vee \ldots \vee \ l_n)$$

Unit clause: literal

- DPLL uses **unit resolution**
- **Boolean Constraint Propagation**: all possible applications of unit resolution on input

Basic DPLL

// returns SAT if CNF formula F is satisfiable; //
otherwise returns UNSAT

```
DPLL(F)
   G = BCP(F)
   if (G = ⊤) then return SAT
   else if (G = ⊥) then return UNSAT
   p = choose_var(G)
   if (DPLL(G[p ↦ ⊤])) then return SAT;
   else return (DPLL(G[p ↦ ⊥]));
```

Boolean constraint propagation

Decision heuristics

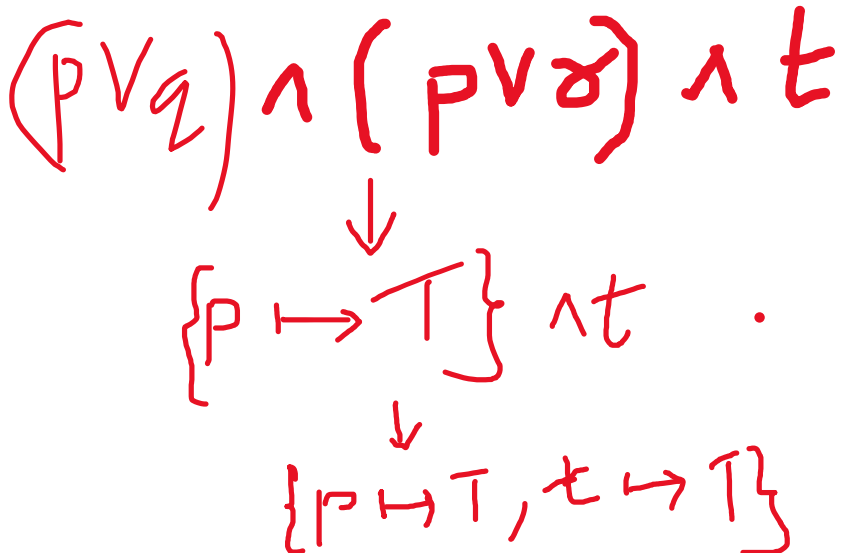
Backtracking

DPLL with Pure Literal Propagation

// returns SAT if CNF formula F is satisfiable; //
otherwise returns UNSAT

```
DPLL(F)
   H = BCP(F)
   G = PLP(H)
   if (G = ⊤) then return SAT
   else if (G = ⊥) then return UNSAT
   p = choose_var(G)
   if (DPLL(G[p ↦ ⊤])) then return SAT;
   else return (DPLL(G[p ↦ ⊥]));
```

Pure Literal Propagation
If variable p occurs only positively
p must be set to T
If p occurs only negatively,
p must be set to ⊥



$F : (\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r) \wedge (p \vee \neg q \vee \neg \gamma)$

choose q to branch on

$F[q \mapsto \top] : r \wedge \neg r \wedge (p \vee \neg \gamma)$

$\equiv$

$\bot$

$F[q \mapsto \bot] : (\neg p \vee r)$

PLP : $[p \mapsto \bot, \gamma \mapsto \top, q \mapsto \bot]$ Model

Beyond DPLL

Learning conflict clauses that summarize conflicts and augmenting F with them

Non-chronological backtracking to earlier decision levels based on cause of conflict

Decision heuristics choose the next literal to add to the current partial assignment based on the state of the search.

Conflict-Driven Clause Learning (CDCL)

SAT solving landscape today

- CDCL based solvers routinely solve problems with hundred of thousands or even millions of variables
- But still possible to create very small instances that take very long!

Not every small SAT problem is easy

- An example: the pigeonhole problem
- Is it possible to place n pigeons into m holes?
- Obvious for humans!
- But turns out to be very difficult to solve for SAT solvers!

Encoding the Pigeon hole problem in PL

Let's encode this for $m = n - 1$.

- Let $p_{i,j}$ stand for "pigeon i placed in j'th hole"
- Given we have $n - 1$ holes, how to say i'th pigeon must be placed in some hole?
- Given we have n pigeons, how to say every pigeon must be placed in some hole?

$$p_{1,1} \vee p_{1,2} \vee \ldots p_{1,n-2} \vee p_{1,n-1}$$
$$\wedge\ p_{2,1} \vee p_{2,2} \vee \ldots p_{2,n-2} \vee p_{2,n-1}$$

$$\vdots$$

$$\wedge\ p_{n,1} \vee p_{n,2} \vee \ldots p_{n,n-2} \vee p_{n,n-1}$$

Pigeon hole problem, cont.

- More concise way of writing this:

$$\bigwedge_{0 \le k < n} \left(\bigvee_{0 \le l < n-1} p_{k,l} \right)$$

- We also need to state that multiple pigeons cannot be placed into same hole:

$$\bigwedge_{k} \bigwedge_{i} \bigwedge_{j \neq i} \neg p_{ik} \vee \neg p_{jk}$$

$\neg (p_{ik} \wedge p_{jk})$

- With $n > 25$, this formula cannot be solved by competitive SAT solvers!

- Problem: Conflict clauses talk about specific holes/pigeons, but problem is symmetric!
- Research on *symmetry breaking*

Variations of the Boolean Satisfiability problem

Maximum Satisfiability (MaxSAT)

Given CNF formula F, find assignment maximizing the number of satisfied clauses of F

- If F is satisfiable, the solution to the MaxSAT problem is the number of clauses in F.
- If F is unsatisfiable, we want to find a maximum subset of F's clauses whose conjunction is satisfiable.

Partial MaxSAT

Given CNF formula F where each clause is marked as **hard** or **soft**, find an assignment that satisfies all hard clauses and maximizes the number of satisfied soft clauses

- Similar to MaxSAT, but we distinguish between two kinds of clauses
- **Hard clauses**: clauses that must be satisfied
- **Soft clauses**: clauses that we would like to, but do not have to, satisfy

- In normal SAT, all clauses are implicitly hard clauses
- In MaxSAT, all clauses are implicitly soft clauses
- In this sense, Partial MaxSAT is a generalization over both SAT and MaxSAT

Partial Weighted MaxSAT

Given CNF formula F where each clause is marked as **hard** or **soft** and is assigned a **weight**, find an assignment that satisfies all hard clauses and maximizes the sum of the weights of satisfied soft clauses

Partial MaxSAT is an instance of partial weighted MaxSAT where all clauses have equal weight

Unsatisfiable Cores

Given CNF formula F, an unsatisfiable core is an inconsistent subset of the clauses of F.

An unsatisfiable core is minimal if dropping *any* one of its clauses makes it satisfiable.

Helpful for fault localization and program repair!

$\neg s \wedge \neg\gamma \vee s \wedge \gamma \wedge s$

Unsat cores

1. $\neg s, s$

2. $\neg\gamma \vee s, \gamma, \neg s$

3. $\neg s, \neg\gamma \vee s, \gamma, s$ ✗ Not minimal

Summary

Today

- DPLL algorithm for SAT solving
- One challenge for current SAT solvers
- Variations of the satisfiability problem (e.g., MaxSAT)

Next

- First-order logic