CS389L: Automated Logical Reasoning

Lecture 4: Applications of SAT

Işıl Dillig

Plan for Today

- Some challenges for current SAT solvers
- Applications of SAT: product configuration, hardware manufacturing
- Variations on the satisfiability problem (e.g., MaxSAT)

SAT Solving Landscape Today

- Current CDCL based solvers able to solve problems with hundred thousands or even millions of variables
- Check out satcompetition.org!
- SAT solvers routinely solve very large problems, but 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 Pigeonhole Problem in Propositional Logic

- 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 do we say i'th pigeon must be placed in at least one hole?
- Given we have n pigeons, how do we say every pigeon must be placed in one hole?

$$\begin{array}{l} p_{1,1} \lor p_{1,2} \lor \ldots p_{1,n-2} \lor p_{1,n-1} \\ \land\ p_{2,1} \lor p_{2,2} \lor \ldots p_{2,n-2} \lor p_{2,n-1} \\ \vdots \\ \land\ p_{n,1} \lor p_{n,2} \lor \ldots p_{n,n-2} \lor p_{n,n-1} \end{array}$$

Pigeon Hole Problem, cont.

- More concise 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:

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

- With $n > 25$, this formula cannot be solved by competitive SAT solvers!
- Problem: Conflict clauses talk about specific holes/pigeons, but problem is symmetric!
- $\Rightarrow$ Research on symmetry breaking

Why So Much Work on SAT solvers?

- Many interesting and computationally difficult problems can be reduced to SAT
- Boolean satisfiability is one of the most basic of NP-complete problems
- Idea: Write one really good SAT solver and reduce all other NP-complete problems to SAT

Review of NP-Completeness

A problem A is NP-complete if:

- In complexity class NP:
-
- Also NP-hard:
- This poly-time transformation to A is called a reduction

Complexity Class of Validity

- Question: Is checking validity in propositional logic also NP-complete?
- Answer:
- Checking validity in propositional logic is co-NP complete

Review of co-NP Completeness

A problem A is co-NP-complete if:

- In complexity class co-NP: its complement is in NP
- complement $=$ decision problem resulting from swapping the yes/no answers
- Another way of saying a problem is in co-NP: we can verify a counterexample in polynomial time
- Checking validity is in co-NP because we can verify that an interpretation is falsifying in polynomial time.
- Also co-NP hard: There is a polynomial time transformation from any problem in co-NP to A (holds for validity)

Practical Applications of SAT

- Applications of SAT solvers: automated testing of circuits, product configuration, package management, computational biology, cryptanalysis, particle physics, solving many graph problems . . .
- We will look at two example applications:
  - Product configuration
  - Automatic test pattern generation for hardware

Applications of SAT in Product Configuration

- Motivation: Some products, such as cars, are highly customizable

- For example, Mercedes C class has a total of >650 options!
- Leather interior, seat heating, thermotronic comfort air conditioning, high-capacity battery, ventilated seats, heated steering wheel, 64-color LED ambient lighting, blind spot assist...

Lots of Options = Lots of Dependecies

- But there may be intricate dependencies between these configurations
- Example: "Thermotronic comfort air conditioning requires high-capacity battery except when combined with gasoline engines of 3.2 liter capacity"
- Customers may not be aware of all these dependencies, so they may choose inconsistent configuration options

Using SAT Solvers to Check Configurations

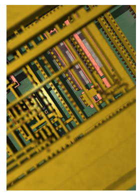
- Since there are too many configurations and too many dependencies, it is not feasible to have a human check them!
- Idea: Use SAT solver to check if the user picks consistent configuration options
- Encode the dependencies between configurations as a propositional formula ψ
- Encode user-selected options as propositional formula ϕ
- Use SAT solver to check if $\psi \wedge \phi$ is satisfiable
- If yes, then chosen configuration is fine

Example: Encoding Dependencies as Boolean Formulas

- Recall the dependency: "Thermotronic comfort air conditioning requires high-capacity battery except when combined with gasoline engines of 3.2 liter capacity"
- Introduce propositional variable for different options
  t = thermotronic comfort air conditioning
  b = high-capacity battery
  g = gasoline engine with 3.2 liter capacity
- Consistency of configuration requires:
- If user chooses comfort AC, small battery, but not the 3.2lt. engine, user configuration encoded as:
- Since $\psi \wedge \phi$ unsat, user must pick different configuration

Another Application of SAT Solvers: ATPG

- Another industrial application of SAT solvers: testing integrated circuits
- When manufacturing an integrated circuit, many things can go wrong: complex process involving photolithography, etching, dicing ...
- One common problem: component in circuit stuck at fault (i.e., output of the component is 0 or 1 regardless of input)
- Automatic test pattern generation (ATPG) tries to construct inputs to check for a particular component being stuck at fault

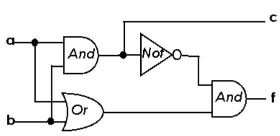


ATPG using SAT

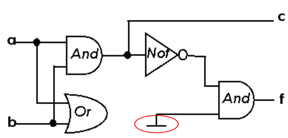
- To formulate ATPG using boolean satisfiability, we consider two variations of the circuit.
- The first one, "the good circuit", represents the circuit without any stuck-at-fault components.
- The second one, "the faulty circuit", represents the circuit with a particular component stuck at fault.

Good vs. Faulty Circuit

- Good circuit:



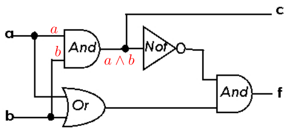
- Faulty circuit:



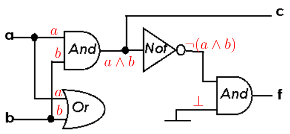
- Here, the OR component is stuck at 0.

Circuit as Propositional Formula

- Now, represent both the good and faulty circuit using propositional formulas F_G and F_F.
- Good circuit:



- Faulty circuit:



Finding an Input to Detect Fault

- To detect if manufactured circuit is faulty, we need an input for which the outputs of the good and faulty circuits differ.
- But such an input must be a satisfying assignment to the formula:
$$(F_G \land \neg F_F) \lor (\neg F_G \land F_F)$$
- Thus, to detect if manufactured circuit is stuck at fault, test on inputs that are sat assignments to above formula
- **Moral:** Boolean satisfiability useful for finding hardware defects!

Variations on the Boolean Satisfiability Problem

- So far, we considered the basic boolean satisfiability problem: Given a propositional formula F, is F satisfiable?
- There are also some common variations of SAT: Maximum Satisfiability (MaxSAT), Partial MaxSAT, Weighted MaxSAT, min unsat core, ...

Maximum Satisfiability (MaxSAT)

- The MaxSAT problem: Given formula F in CNF, find assignment maximizing the number of satisfied clauses of F.
- Observe: 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.
- Example: What is a solution for the MaxSAT problem $(a \lor b) \land \neg a \land \neg b$?
- Question: How could MaxSAT be useful for product configuration?

Partial MaxSAT

- 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
- Partial MaxSAT problem: 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 satisfied soft clauses

More on Partial MaxSAT

- Observe: Both regular SAT and MaxSAT are special cases of partial MaxSAT
- In normal SAT, all clauses are hard clauses
- In MaxSAT, all claues are implicitly soft clauses
- In this sense, Partial MaxSAT is a generalization over both SAT and MaxSAT

Partial Weighted MaxSAT

- There is even one more generalization over Partial MaxSAT: Partial Weighted MaxSAT
- In addition to being hard and soft, clauses also have weights (e.g., indicating their importance)
- Partial Weighted MaxSAT problem: Find assignment maximizing the sum of weights of satisfied soft clauses
- Partial MaxSAT is an instance of partial weighted MaxSAT where all clauses have equal weight

An Application of Partial MaxSAT

- Software package installation: Suppose you want to install software package A, but it has some dependencies
- For example, suppose A requires B but it is not compatible with package C
- B in turn requires D; E is not compatible with F
- Furthermore, some of these packages may already be installed on your computer (e.g., package C)
- You want to know (i) if it is possible to install package A, and (ii) if not, which software should you uninstall to install A?
- How can we formulate this partial MaxSAT?

A Funny Story...

Lisbon Wedding:
Seating arrangements using MaxSAT

Ruben Martins
Carnegie Mellon University
rubenm@cs.cmu.edu

Justine Sherry
Carnegie Mellon University
sherry@cs.cmu.edu

***Abstract*—Having a perfect seating arrangement for weddings is not an easy task. Can Alice sit next to Bob? Can we ensure that Charles and his ex-girlfriend Eve not be seated together? Meeting such constraints is classically one of the most difficult tasks in planning a wedding – and guests will not accept 'it's NP-complete!' as an excuse for poor seating arrangements. We discuss how MaxSAT can provide the optimal seating arrangement for a perfect wedding, saving brides and grooms (including the authors) from hours of struggle.**

I. INTRODUCTION

This benchmark description describes the encoding used for the wedding seating arrangement for our wedding in Lisbon. We needed to seat our guests according to a long list of constraints. For example, members of the same family should sit together; friends who went to school together should sit together; individuals with a history of conflict should be seated apart; etc. We wanted to maximize the happiness of our guests and what better way to do that than to encode the problem into MaxSAT! MaxSAT was an ideal solution for our own wedding: i) it saved us tens of hours, ii) it was stress free, and iii) in the rare case that a guest complained about their seating arrangement, we just blamed the algorithm![1]

$$\forall_{p \in P} \sum_{t \in T} p_t = 1$$

- Each table will have at most u guests:

$$\forall_{t \in T} \sum_{p \in P} p_t \leq u$$

- Each table will have at least l guests:

$$\forall_{t \in T} \sum_{p \in P} p_t \geq l$$

Since some guests may have disagreements with each other we also included some exclusion constraints that guarantee that guests which have conflicts with each other are not seated in the same table. For every pair of guests p and p' that have a conflict with each other we include the following constraints that guarantee that they will not seat together:

$$\forall_{t \in T} (p_t + p'_t \leq 1)$$

To enforce that if a person p is seated at table t then t