Plan for Today

- Finish discussion of practical SAT solvers (decision heuristics, implementation considerations)
- Applications of SAT: product configuration, hardware manufacturing
- Variations on the satisfiability problem (e.g., MaxSAT)

Recall: SAT Solver Architecture

- Decide
  - BCP
    - no conflict
    - conflict
      - Analyze Conflict
        - UNSAT
        - backtrack if d > 0

Decision Heuristics

- We’ll only talk about two:
  1. dynamic largest individual sum (DLIS)
  2. variable state independent decaying sum (VSIDS)

Dynamic Largest Individual Sum (DLIS)

- This heuristic chooses the literal that satisfies the largest number of currently unsatisfied clauses.
- A clause is unsatisfied if the clause does not evaluate to true under the current partial assignment.
- Example: \((x_1 \lor \neg x_2) \land (\neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3)\)
- What assignment would DLIS pick for this formula? (assuming no assignments so far)
- How is this heuristic dynamic?
- Thus, overhead can be high and must be implemented carefully to minimize bookkeeping

Variable State Independent Decaying Sum (VSIDS)

- Similar to DLIS, but tries to reduce overhead and favor literals involved in conflicts (i.e., conflict-driven)
- Count number of clauses in which the literal appears, but disregard if the clause it appears in is satisfied or not
- Specifically, initialize the score of each literal to the number of clauses in which literal appears
- Every time we add a conflict clause involving literal \(l\), increase the score of that literal by 1
- Much cheaper compared to DLIS because we don’t need to scan all clauses to figure out which ones are satisfied
Variable State Independent Decaying Sum (VSIDS), cont.

- Second aspect of VSIDS: To favor literals that appear in recent conflicts, periodically divide scores of all literals by $2 \Rightarrow$ decaying sum
- If a literal doesn’t appear in recent conflict, its score will decay over time
- On the other hand, if literal appears in recent conflict, its score will be increased, so its score won’t decay as much
- Thus, the VSIDS heuristic favors literals that appear in recent conflicts
- Introduced in the CHAFF SAT solver from Princeton, written by undergrads!

Implementation Considerations for Practical SAT Solving

- To build competitive SAT solvers, it is important to minimize overhead of implementing Decide, BCP, and Analyze Conflict
- Very important because SAT solver might be searching through hundreds of thousands of assignments!
- We’ll talk about two issues:
  1. number of conflict clauses
  2. trick to perform BCP fast: watch literals

Conflict Clauses

- Recall: After analyzing conflict, we add new conflict clause to our clause database
- Advantage: Conflict clauses quickly block bad assignments and prevent future mistakes
- Disadvantage: More clauses = more overhead
  $\Rightarrow$ Tradeoff between conflict prevention and minimizing overhead

Conflict Clauses, cont.

- For this reason, many SAT solvers do not keep all the conflict clauses they derive
- For example, they put a limit on the number of conflict clauses they derive
- Typically, keep most recent conflict clauses since they are most relevant to current part of search space
- Important fact: Can guarantee termination of algorithm even if we do not keep all conflict clauses!

Implementing BCP

- Implementing BCP efficiently is very important because SAT solvers spend a lot of time doing BCP
- Naive implementation of BCP: Requires scanning all currently unsatisfied clauses
- But industrial instances of boolean SAT problems contain hundreds of thousands of clauses
- Thus, scanning all unsatisfied clauses too expensive!
- A more intelligent implementation: Keep mapping from each literal to all clauses in which each literal appears (because we perform unit resolution after each variable assignment)
- But this is still very expensive because typically each literals appears in many clauses

The Trick: Watch Literals

- Modern SAT solvers use a much more clever trick to perform BCP fast: watch literals
- Observe: Ultimate purpose of BCP is to figure out which variable assignments imply which others
- Question: If we are performing unit resolution between $l$ and clause $c = (\neg l \lor l_1 \lor \ldots \lor l_k)$, under what condition will a new assignment be implied?
- Answer:
  - Idea: Since a clause will not imply new variable assignment unless it has only two literals left, we only need to look at clauses that have two unassigned literals!
Watch Literals

- To efficiently detect clauses with at most two unassigned literals, select two unassigned literals in each unsatisfied clause as watch literals.
- Invariant: Watch literals are always unassigned!
- To maintain invariant: If a watch literal is assigned a truth value and clause has other unassigned literals, choose any unassigned literal in clause to be new watch literal.

Watch Literals, cont.

- Question: Given this invariant, if we make assignment \( l \), which clauses can imply new variable assignments?
- Answer:
  - If \( \neg l \) does not appear, we can’t perform unit resolution
  - If \( \neg l \) appears but is not a watch literal, then clause has more than two unassigned literals \( \Rightarrow \) won’t imply new assignment!
  - Watch literal trick makes BCP much faster because much fewer clauses contain negation of current literal as a watch literal!
  - Yielded huge improvement in SAT solver performance!

SAT Solving Landscape Today

- The kind of SAT solvers we discussed are called CDCL-based SAT solvers (conflict-driven clause learning).
- Current CDCL based solvers able to solve problems with hundreds of thousands or even millions of variables.
- SAT solvers routinely solve very large problems, but possible to create very small instances that take very long!

Encoding Pigeon Hole Problem in Propositional Logic

- 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?

\[
\begin{align*}
\text{Pigeon Hole Problem, cont.} & \quad \text{More concise of writing this:} \\
& \quad \bigwedge_{0 \leq k < n} \left( \bigvee_{0 \leq l < n-1} p_{k,l} \right) \\
& \quad \text{We also need to state that multiple pigeons cannot be placed into same hole:} \\
& \quad \bigwedge_{k \neq l} \bigwedge_{1 \leq j \leq n-1} \neg p_{k,j} \lor \neg p_{l,j} \\
& \quad \text{With } n > 25, \text{ this formula cannot be solved by competitive SAT solvers!} \\
& \quad \text{Problem: Conflict clauses talk about specific holes/pigeons, but problem is symmetric!} \\
& \quad \Rightarrow \text{Research on symmetry breaking}
\end{align*}
\]
SAT Solving Landscape

- Even though SAT is NP-complete, SAT solvers work well in practice
- Active research community: annual SAT solving competitions (check out satcompetition.org)
- Since all NP-complete problems can be reduced to SAT, we can solve many other difficult problems using SAT
- People reduce many real-world problems to SAT all the time and use off-the-shelf SAT solvers to solve them!

Practical Applications of SAT

- Applications of SAT solvers: automated hardware/software testing, product configuration, package management, computational biology, cryptanalysis, particle physics, ...
- 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 692 options!
  - Leather interior, built-in GPS, seat heating, thermotronic comfort air conditioning, high-capacity battery, ...

Lots of Options = Lots of Dependencies

- 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
- Encoded dependencies between configurations as a propositional formula $\psi$
- Encode user-selected options as propositional formula $\phi$
- Use SAT solver to check if $\psi \land \phi$ 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 $\psi \land \phi$ 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

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: Consider CNF formula $(a \lor b) \land \neg a \land \neg b$.
- The maximum number of clauses that can be satisfied by any assignment is 2.

Partial MaxSAT

- Yet another generalization over 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 clauses 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?