COSC410 Logic for AI

Introduction to SAT solvers

Richard A. O'Keefe

12 May 2016

Key Topics

- propositional formulas
- seeing formulas as trees
- clausal form
- the SAT problem and its complexity
- getting to clausal form (two ways)
- the DPLL procedure

Propositional Formulas

A propositional formula ϕ can be

- an atomic sentence π such as *p*
- or a compound formula $\psi \leftrightarrow \chi$, $\psi \rightarrow \chi$, $\psi \vee \chi$, $\psi \wedge \chi$, or $\neg\chi$, where ψ and χ are smaller formulas.

Not a string but a tree

- When we look at a written formula, we see a string of symbols.
- We have to use precedence rules and parentheses to disambiguate it.
- But a compound formula has a principal connective and one or more parts.
- We're dealing with a *tree*.

Why do we care?

Recursive definition by cases.

$$
\begin{array}{rcl}
V[\pi]\rho & = & \rho(\pi) \\
V[\psi \leftrightarrow \chi]\rho & = & V[\psi]\rho {\leftrightarrow} V[\chi]\rho \\
V[\psi \rightarrow \chi]\rho & = & V[\psi]\rho {\rightarrow} V[\chi]\rho \\
V[\psi \vee \chi]\rho & = & V[\psi]\rho {\vee} V[\chi]\rho \\
V[\psi \wedge \chi]\rho & = & V[\psi]\rho {\wedge} V[\chi]\rho \\
V[\neg\chi]\rho & = & \neg V[\chi]\rho
\end{array}
$$

where the connectives on the left connect trees and the connectives on the right are applied to truth values, and ρ maps atomic sentences to truth values.

Why this is meaningful

Induction over trees works like induction over natural numbers.

$$
\begin{aligned}
S[\pi] &= 1 \\
S[\psi \leftrightarrow \chi] &= 1 + S[\psi] + S[\chi] \\
S[\psi \rightarrow \chi] &= 1 + S[\psi] + S[\chi] \\
S[\psi \vee \chi] &= 1 + S[\psi] + S[\chi] \\
S[\psi \wedge \chi] &= 1 + S[\psi] + S[\chi] \\
S[\neg \chi] &= 1 + S[\chi]
\end{aligned}
$$

Recursion on formulas drive sizes (S) down.

Proofs by cases and induction

- Just as we can define functions on formulas recursively, so we can give inductive proofs about formulas.
- Having more ways to build a tree makes a language more convenient to *use*, but harder to reason about.
- A simpler *language* may mean more complex *formulas*

We're already missing handy connectives

- $\psi \oplus \chi = \neg(\psi \leftrightarrow \chi)$
- if τ then ψ else χ
- $\langle \alpha\beta\gamma \rangle$, the median or majority function
- $n^{=}(\psi, \chi, \dots) =$ exactly n of ψ, χ, …are true.
  $\psi \wedge \chi = 2^{=}(\psi, \chi)$.
- $n^{\geq}(\psi, \chi, \dots) =$ at least n of ψ, χ, …are true.
  $\psi \vee \chi = 1^{\geq}(\psi, \chi)$.

Simplicity can go too far, or can it?

Anything you can express with the standard propositional connectives or the ones introduced on the previous slide can be expressed using just *one* connective.

- NAND: $\psi \tilde{\wedge} \chi = \neg(\psi \wedge \chi)$
- NOR: $\psi \tilde{\vee} \chi = \neg(\psi \vee \chi)$
- ITE: if π then ψ else χ (but needs $\top$ and $\bot$ at leaves)

NAND and NOR are handy for making electronic circuits. If-then-else is one effective approach to SAT solving. Not good for people, though!

Clausal form

- A sentence in clausal form is the conjunction ($\wedge$) of a set of *clauses*.
- A clause is the disjunction ($\vee$) of a set of *literals*.
- A literal is either an atom (π) or the negation of an atom ($\neg\pi$).

Example: $((\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r))$.

Suppressing the connectives

- Since we know the top level connectives are all $\wedge$ we don't need to write them, just write the sentence as a set.
- Since we know the mid level connectives are all $\vee$ we don't need to write them, just write the clause as a set.

Example: $\{\{\neg p, q\}, \{\neg q, r\}, \{\neg r\}\}$.

Valuations

- A *valuation* is a function from atomic sentences to truth values.
- Because there are only two truth values, we can think of a valuation as a partition of the set of atomic sentences into $\mathcal{T}$ (the ones mapped to $\top$) and $\mathcal{F}$ (the ones mapped to $\perp$)
- The second approach is handy for *partial* valuations where some atomic sentences have been assigned values and others may not. We shall meet partial valuations later.
- A *model* of a formula is a valuation making the formula true.

SAT

Given a propositional formula in clausal form,

- **Decision** problem: does it have a model?
- **Search** problem: find a model or determine that there isn't one.
- **Optimisation** problem: given a price function on valuations, find a cheapest model (if there is one).

Strictly speaking, SAT is the decision problem.

SAT complexity

- If a formula has n variables, there are 2^n valuations, which we can generate straightforwardly.
- If the formula has m operators, checking a valuation takes $O(m)$ time, so we can try all possible valuations in $O(m.2^n)$ time.
- Can we do better than this?
- In theory, no. Cook proved that SAT is NP-complete. (Checking is easy, searching is hard. Lots of problems can be converted to SAT, which shows they're hard.)
- Yet SAT-solvers are quite practical these days.

k-SAT

- k-SAT is the problem where every clause has k literals.
- 2-SAT can be solved in linear time.
- 3-SAT is as hard as SAT.

Getting to clausal form

- The basic idea is to *rewrite* a formula from one form to another.
- We use logical theorems $\alpha \equiv \beta$ (to preserve meaning)
- and *orient* them $\alpha \Rightarrow \beta$ to convert a "more complex" formula to strictly "less complex" one (so that rewriting terminates and "simplifies")

Four stages

1. Eliminate equivalence.
2. Eliminate implication.
3. Move negation down to the leaves.
4. Move conjunction above disjunction.

Each of these stages has its own notion of "complexity".

Eliminate equivalence

- Complexity of a formula: the number of $\leftrightarrow$ connectives in it.
- Rewrite rule: $\psi \leftrightarrow \chi \Rightarrow (\psi \wedge \chi) \vee (\neg\psi \wedge \neg\chi)$.
- Apply this rule bottom-up, so that we know ψ and χ do not contain $\leftrightarrow$.

Eliminate implication

- Complexity of a formula: the number of $\rightarrow$ connectives in it.
- Rewrite rule: $\psi \rightarrow \chi \Rightarrow (\neg\psi) \vee \chi$
- This rule does not duplicate ψ or χ so the proof of termination works top-down or bottom-up.

Move negation down to the leaves

- Complexity: the number of nodes covered by negations
- Rules:
  - $\neg(\psi \vee \chi) \Rightarrow (\neg\psi) \wedge (\neg\chi)$
  - $\neg(\psi \wedge \chi) \Rightarrow (\neg\psi) \vee (\neg\chi)$
  - $\neg(\neg\chi) \Rightarrow \chi$

Move conjunction above disjunction

- Complexity: weighted sum of or-over-and violations.
- Rules:
  - $\tau \vee (\psi \wedge \chi) \Rightarrow (\tau \vee \psi) \wedge (\tau \vee \chi)$
  - $(\psi \wedge \chi) \vee \tau \Rightarrow (\tau \vee \psi) \wedge (\tau \vee \chi)$
- What if both rules are applicable? E.g., $(a \wedge b) \vee (c \wedge d)$.
- You get the same end result either way. (Try it.)

Oops

- This process shows that we *can* get from any formula to clausal form by a simple mechanisable procedure.
- But "Eliminate equivalence" *duplicates* subformulas. Hello power of two! Eliminating $\oplus$ or $\langle \ldots \rangle$ or if-then-else or $n^{=}$ or $n^{\geq}$ this way would also duplicate subformulas.
- Worse still, applying the distribution laws *also* duplicates a subformula. Hello exponential growth!

Fighting the monster

- Cook's theorem tells us that we can't do better than exponential time in the worst case.
- But we don't have to go out of our way to cause trouble for ourselves.
- Avoiding an exponential blowup in the size of the formula is a good idea.
- The easiest way to do that introduces new atomic sentences.

The Tseitin Transformation

- Key idea: introduce a new atom for each node in the tree.
- Each node now relates at most 3 atoms, and its semantics can be represented by a small set of clauses.
- Glue those clause sets together with $\wedge$ and you're done.
- But since the original formula and the transformed one have different sets of atoms, they have different sets of valuations.

We mostly don't care

- **Decision:** the transformed formula has a model if and only if the original formula has a mode.
- **Search:** any model for the transformed formula yields a model for the original, by dropping the new atoms.
- **Optimisation:** if the price of a valuation ignores the new atoms, the transformed and original formulas have the same cheapest models.
- See `Tseytin_transformation` in Wikipedia.

Tseitin: the rules

$$
\begin{aligned}
C = \neg A \quad & \Rightarrow \{\{\neg A, \neg C\}, \{A, C\}\} \\
C = A \wedge B \quad & \Rightarrow \{\{\neg A, \neg B, C\}, \{A, \neg C\}, \{B, \neg C\}\} \\
C = A \vee B \quad & \Rightarrow \{\{A, B, \neg C\}, \{\neg A, C\}, \{\neg B, C\}\} \\
C = A \rightarrow B \quad & \Rightarrow \{\{\neg A, B, \neg C\}, \{A, C\}, \{\neg B, C\}\} \\
C = A \leftrightarrow B \quad & \Rightarrow \{\{A, B, C\}, \{A, \neg B, \neg C\}, \\
& \qquad \{\neg A, B, \neg C\}, \{\neg A, \neg B, C\}\}
\end{aligned}
$$

Exhaustive search

- A simple way to solve any problem with a finite number of discrete variables:
  - Generate each combination and
  - check if it works.
- Simple, yes. Efficient, no. Adequate to show that the problem *can* be solved. Always seek something better.
- Called "generate-and-test". Rule of thumb: push tests back into generator.

Seeking a path through a state space

```
— Simple depth first search
procedure DFS(state, solved?, report, children)
    if solved?(state) then
        report(state)
    else
        for successor ∈ children(state) do
            DFS(successor, solved?, report, children)
```

Backtrack programming

- Given a problem with n variables, where each variable x_i has a finite domain D_i, and there is a system of constraints,
- we seek a solution by calling solve($\{\}$, $\{1..n\}$, $\{j \mapsto D_j, \ldots\}$), where
- solve(Known, Unknown, Domains) $=$
  - if Unknown is empty, report Known.
  - remove some j from Unknown.
  - $\langle$try values for $j\rangle$
  - *undo* remove j
- If no solution is reported, there is none.

$\langle$try values for $j\rangle$

- for each v in Domains[j] in some order,
  - add $j \mapsto v$ to Known.
  - if the constraints are not yet violated
    - remove now impossible values from Domains
    - solve(Known$'$, Unknown$'$, Domains$'$)
    - *undo* remove now impossible values
  - *undo* add $j \mapsto v$

Room for heuristics

- Which variable j do you pick? (Try the one with the smallest domain, called the "fail-fast" heuristic.)
- What order do you try the values v?
- How much work do you put into ensuring that constraints are not obviously unsatisfiable?
- Do you learn anything from unsuccessful searches? (Modern SAT solvers do.)
- Key point: the *undo* steps may be unfamiliar. They let us explore large problems by making small, incremental changes and undoing them, instead of copying.

Backtracking is used for

- pretty much any constraint satisfaction or combinatorial optimisation problem, at least as a starting point.
- graph colouring, parsing mildly ambiguous grammars, Sudoku and similar puzzles, finding paths, planning, finding assignments of people to tasks or resources to people, course selection, ...

Incremental simplification (*alias* forward checking, *alias* constraint propagation) usually helps.

Davis-Putnam-Logemann-Loveland Procedure

- Variables: atomic sentences
- Their domains: $\{\bot, \top\}$
- Constraints: the clauses
- Known: a partial valuation
- Unknown: as-yet-unknown atomic sentences
- Seeking: a complete valuation consistent with the clauses

At each DPLL step

- **unit propagation**. Unit clauses have one literal. "Propagation" is substituting known information in clauses (or other constraints) and simplifying. For example, $\{x\}$ and $\{\{x, \neg y\}, \{\neg x, y\}\}$ simplifies to $\{\{y\}\}$. The first clause drops out. The second clause simplifies: if x is true, the only way to make $\neg x \vee y$ true is for y to be true. Now we can substitute y and simplify…
- **pure literal elimination**. If some clause has π but no clause has $\neg\pi$, make π true and delete all clauses containing π. (Similarly $\neg\pi$.)

and then

After unit propagation and pure literal elimination,

- **search**: pick some variable π
- make π true, simplify, and recur.
- *undo* those changes.
- make π false, simplify, and recur.
- *undo* those changes.

Finally *undo* changes made by unit propagation and pure literal elimination.

Improvements to DPLL

- Clever data structure design to make changing and undoing changes easy.
- Typical backtracking heuristics: choose the most constrained variable, try first the value that will let you eliminate most clauses.
- Learn from failure: non-chronological backtracking and conflict-driven clause learning.
- Human care in formulating a problem, *e.g.*, exploit symmetry.