University of Strathclyde Science

CS208 (Semester 1) Topic 9 : Automating Logic

Dr. Robert Atkey

Computer & Information Sciences

University of Strathclyde Science

Automating Logic, Part 1

Automating Logic

University of Strathclyde Science

What to Automate?

Given a formula P, we can ask:

1. Does P have a proof?
2. Is P valid?
3. Is P satisfiable?
4. For a model $\mathcal{M}$, do we have $\mathcal{M} \models \mathrm{P}$?
5. Can we generate $\mathcal{M}$, such that $\mathcal{M} \models \mathrm{P}$?
6. How many models does P have?

Given the expressiveness of Predicate Logic, this covers a large range of questions.

The Bad News

Lots of things are undecidable:

1. Validity in Predicate Logic
2. Entailment in Predicate Logic
3. Checking models of Predicate Logic
4. Model generation for Predicate Logic

Undecidable: No program that can perfectly say “yes” or “no”.

More Bad News

Or theoretically intractable:

1. Validity / Satisfiability checking in Propositional Logic
2. Synthesis of finite models
3. Checking of finite models

Intractable: there is no (known) program to solve it that runs in better than $O(2^n)$ on inputs of size n.

The Good News

University of Strathclyde Science

For undecidable problems, there are good *semi-decision* procedures. Semi-decision: says "yes" exactly when the problem is solved. No guarantees otherwise.

For intractable problems, there are heuristics that solve common cases quickly.

University of Strathclyde Science

Many Algorithms

1. Proof Search
   Try all possible proof rules, under some strategy
2. Resolution Provers
   A proof system specialised to proof search
3. Specific tools for sub-languages of Predicate Logic
   E.g., “Horn” clause provers
4. SAT / SMT Solvers
   SATisfiabilty solvers
   SATisfiabilty Modulo Theory solvers

University of Strathclyde Science

Many Algorithms

1. Proof Search
   Try all possible proof rules, under some strategy
2. Resolution Provers
   A proof system specialised to proof search
3. Specific tools for sub-languages of Predicate Logic
   E.g., “Horn” clause provers
4. SAT / SMT Solvers
   SATisfiabilty solvers
   SATisfiabilty Modulo Theory solvers

We will look at SAT / SMT solvers here.

SAT / SMT

SAT solvers work on Propositional Logic.

SMT solvers work on a *quantifier-free* fragment of Predicate Logic.

Lots of industrial strength tools: Z3, CVC5, Yices, ...

Used by (e.g.) Amazon Web Services to check access control rules, Microsoft to verify software, ...

University of Strathclyde Science

Automating Logic, Part 2

SAT Solvers

SAT solvers

SATisfiability solvers.

The problem they solve:

- Given a formula P (in *conjunctive normal form*), find a valuation ν that makes it T and return SAT(ν), or if there is no such valuation, return UNSAT.

Solving SAT

University of Strathclyde Science

- In the worst case, there are 2^n cases to check, where n is the number of atomic propositions.
  - Checking each case is quick ... but there are a lot of cases.
- This is the archetypal NP problem:
  - If we knew the answer, it would be easy to check
    (**P**olynomial time)
  - But there are exponentially many to check
    (**N**ondeterminism)
- It is unknown if there is a better way. Does P = NP?

Encoding Problems into SAT

In general, if we want to prove that P is valid, then it suffices to show that $\neg$P is not satisifable:

1. Take P
2. Put $\neg$P into a SAT solver:
   - 2.1 if $\neg$P is satisfiable, then we have a counter example to P
   - 2.2 if $\neg$P is not satisfiable, then P is valid

Also, there are *many* problems that can be encoded as Propositional Logic formulas.

But SAT is useful: Solving Problems

University of Strathclyde Science

1. Package installations

   (satisfying valuation = good package installation)

2. Solving Sudoku

   (satisfying valuation = correct solution)

3. Solving Resource allocations

   (satisfying valuation = feasible resource allocation)

SAT is Useful: Finding Bugs

University of Strathclyde Science

1. Finding faults in systems

   (satisfying valuation = path to a bad state)

2. Finding flaws in Access Control rules

   (satisfying valuation = unexpectedly permitted request)

3. Verifying hardware

   (satisfying valuation = counterexample to correctness)

An alluring proposition

Instead of writing custom solvers for all these problems, we:

1. translate into propositional logic; and
2. use an off the shelf SAT solver.

Solving the problem in practice

Despite the 2^n worst case time, practical SAT solvers are possible:

1. Solvers don't blindly check all cases:
   - Use the formula to guide the search;
   - Analyse dead ends to avoid finding them more than once;
   - Very efficient data structures.
2. Human-made problems tend to be quite regular.
3. Modern SAT solvers can handle
   - 10s of thousands of variables
   - millions of clauses
4. Practical tools for solving real-world problems.

University of Strathclyde Science

Input for SAT solvers

SAT solvers take input in *Conjunctive Normal Form* (CNF):

$$\begin{aligned} & (\neg a \vee \neg b \vee \neg c) \\ \wedge \quad & (\neg b \vee \neg c \vee \neg d) \\ \wedge \quad & (\neg a \vee \neg b \vee c) \\ \wedge \quad & b \end{aligned}$$

1. Entire formula is a conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_n$
2. where each *clause* $C_i = L_{i,1} \vee L_{i,2} \vee \cdots \vee L_{i,k}$
3. where each *literal* $L_{i,j} = x_{i,j}$ or $L_{i,j} = \neg x_{i,j}$

Every formula can be put into CNF (later)

Conjunctive Normal Form

The restriction to CNF may seem like a massive restriction.

Every Propositional Logic formula can be translated into CNF.

1. Slow way: “multiply out the brackets”
   Resulting formula might be exponentially larger
2. Fast way: “Tseytin translation”
   Resulting formulas is at most 3 times larger

We’ll just assume this can be done for now.

University of Strathclyde Science

A SAT Solver's job

Given clauses that look like:

$$\begin{aligned} & (\neg a \vee \neg b \vee \neg c) \\ \wedge \quad & (\neg b \vee \neg c \vee \neg d) \\ \wedge \quad & (\neg a \vee \neg b \vee c) \\ \wedge \quad & b \end{aligned}$$

To find a valuation v for the $a, \ldots$ such that at least one literal in every clause is true.

Returns either: $\mathsf{SAT}(v)$ or UNSAT.

Basic idea of the algorithm

1. The clauses $C_1, \ldots, C_n$ to be satisfied are fixed;
2. The state is a partial valuation (next slide);
3. At each step we pick a way to modify the current partial valuation by choosing from a collection of rules;
4. Algorithm terminates when either a satisfying valuation is constructed, or it is clear that this is not possible.

This is known as the *DPLL Algorithm*.

Partial Valuations

To describe what a SAT solver does, we need *partial valuations*.

A **partial valuation** $\nu^?$ is a:

- *sequence* of assignments to atoms; with each one marked
  1. decision point, if we guessed this value.
  2. forced, if we were forced to have this value.

Examples: $\nu_1^? = [a :_d T, b :_d F, c :_f T]$

$\nu_2^? = [a :_f F, b :_d F]$

Differences with Valuations

1. The order matters

   (we keep track of what decisions we make during the search)

2. Not all atoms need an assignment

   (we want to represent partial solutions during the search)

3. We mark decision points and forced decisions.

Notation

We write

$$\nu_1^?, a :_d x, \nu_2^?$$

for a partial valuation with $a :_d x$ somewhere in the middle.

We write

$$\text{decisionfree}(\nu^?)$$

if none of the assignments in $\nu^?$ are marked d

(i.e., all decisions in $\nu^?$ are forced)

1. Initialisation

We start with the *empty partial valuation* $\nu^? = []$.

(We make no commitments)

We must extend this guess to a valuation that satisfies all the clauses.

2. Guessing

If there is an atom a in the clauses that is not in the current partial valuation $\nu^?$, then we can make a guess. We pick one of:

$$\nu^?, a :_d \mathsf{T} \qquad \text{or} \qquad \nu^?, a :_d \mathsf{F}$$

(Note: we have marked this as a decision point)

3. Success

If the current $v^?$ makes all the clauses true (for all i, $[\![C_i]\!]v^? = T$), then stop with $SAT(v^?)$.

Example

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

(Need at least one cyan in every clause)

Sequence of (lucky) guesses

1. $\square$

Example

$$(\checkmark\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\checkmark\neg a \vee \neg b \vee c) \wedge b$$

(Need at least one cyan in every clause)

Sequence of (lucky) guesses

1. []
2. $[a :_d F]$

Example

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \neg c) \wedge (\overset{\times}{\neg b} \vee \neg c \vee \neg d) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee c) \wedge \overset{\checkmark}{b}$$

(Need at least one cyan in every clause)

Sequence of (lucky) guesses

1. []
2. $[a :_d F]$
3. $[a :_d F, b :_d T]$

Example

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c} \vee \neg d) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{c}) \wedge \overset{\checkmark}{b}$$

(Need at least one cyan in every clause)

Sequence of (lucky) guesses

1. $[]$
2. $[a :_d F]$
3. $[a :_d F, b :_d T]$
4. $[a :_d F, b :_d T, c :_d F]$

Example

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c} \vee \overset{\checkmark}{\neg d}) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{c}) \wedge \overset{\checkmark}{b}$$

(Need at least one cyan in every clause)

Sequence of (lucky) guesses

1. []
2. $[a :_d F]$
3. $[a :_d F, b :_d T]$
4. $[a :_d F, b :_d T, c :_d F]$
5. $[a :_d F, b :_d T, c :_d F, d :_d F]$, a satisfying valuation.

University of Strathclyde Science

But we can't program "luck"!

4. Backtracking

If we have a partial valuation:

$$\nu_1^?, a :_d x, \nu_2^?$$

and $\mathrm{decisionfree}(\nu_2^?)$ (so $a : x$ was our most recent guess).

Then we backtrack (throw away $\nu_2^?$) and change our mind:

$$\nu_1^?, a :_f \neg x$$

marking the assignment as forced.

5. Failure

If all decisions are forced ($\mathrm{decisionfree}(\nu^?)$), and there is at least one clause C_i such that $[\![C]\!]\nu^? = F$, then return UNSAT.

$$(\neg a \lor \neg b \lor \neg c) \land (\neg b \lor \neg c \lor \neg d) \land (\neg a \lor \neg b \lor c) \land b$$

1. □

$$(\overset{\times}{\neg a} \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\overset{\times}{\neg a} \vee \neg b \vee c) \wedge b$$

1. $[]$
2. $[a :_d \top]$

$$(\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \neg c) \wedge (\overset{\times}{\neg b} \vee \neg c \vee \neg d) \wedge (\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee c) \wedge \overset{\checkmark}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$

$$(\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\times}{\neg c} \vee \neg d) \wedge (\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*

$$(\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c} \vee \neg d) \wedge (\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*
5. $[a :_d T, b :_d T, c :_f F]$ *clause 3 failed, backtrack...*

$$(\overset{\times}{\neg a} \vee \overset{\checkmark}{\neg b} \vee \neg c) \wedge (\overset{\checkmark}{\neg b} \vee \neg c \vee \neg d) \wedge (\overset{\times}{\neg a} \vee \overset{\checkmark}{\neg b} \vee c) \wedge \overset{\times}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*
5. $[a :_d T, b :_d T, c :_f F]$ *clause 3 failed, backtrack...*
6. $[a :_d T, b :_f F]$ *clause 4 failed, backtrack...*

$$(\underset{\checkmark}{\neg a} \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\underset{\checkmark}{\neg a} \vee \neg b \vee c) \wedge b$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*
5. $[a :_d T, b :_d T, c :_f F]$ *clause 3 failed, backtrack...*
6. $[a :_d T, b :_f F]$ *clause 4 failed, backtrack...*
7. $[a :_f F]$

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \neg c) \wedge (\overset{\times}{\neg b} \vee \neg c \vee \neg d) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee c) \wedge \overset{\checkmark}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*
5. $[a :_d T, b :_d T, c :_f F]$ *clause 3 failed, backtrack...*
6. $[a :_d T, b :_f F]$ *clause 4 failed, backtrack...*
7. $[a :_f F]$
8. $[a :_f F, b :_d T]$

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\times}{\neg c} \vee \neg d) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*
5. $[a :_d T, b :_d T, c :_f F]$ *clause 3 failed, backtrack...*
6. $[a :_d T, b :_f F]$ *clause 4 failed, backtrack...*
7. $[a :_f F]$
8. $[a :_f F, b :_d T]$
9. $[a :_f F, b :_d T, c :_d T]$

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\times}{\neg c} \vee \overset{\times}{\neg d}) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*
5. $[a :_d T, b :_d T, c :_f F]$ *clause 3 failed, backtrack...*
6. $[a :_d T, b :_f F]$ *clause 4 failed, backtrack...*
7. $[a :_f F]$
8. $[a :_f F, b :_d T]$
9. $[a :_f F, b :_d T, c :_d T]$
10. $[a :_f F, b :_d T, c :_d T, d :_d T]$ *clause 2 failed, backtrack*

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\times}{\neg c} \vee \overset{\checkmark}{\neg d}) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$
2. $[a :_d T]$
3. $[a :_d T, b :_d T]$
4. $[a :_d T, b :_d T, c :_d T]$ *clause 1 failed, backtrack...*
5. $[a :_d T, b :_d T, c :_f F]$ *clause 3 failed, backtrack...*
6. $[a :_d T, b :_f F]$ *clause 4 failed, backtrack...*
7. $[a :_f F]$
8. $[a :_f F, b :_d T]$
9. $[a :_f F, b :_d T, c :_d T]$
10. $[a :_f F, b :_d T, c :_d T, d :_d T]$ *clause 2 failed, backtrack*
11. $[a :_f F, b :_d T, c :_d T, d :_d F]$ SAT

Summary

University of Strathclyde Science

1. SAT solvers are tools that find satisfying valuations for formulas in CNF.
2. Having a SAT solver enables solving of problems modelled using logic.
3. The core algorithm is a backtracking search.

University of Strathclyde Science

Automating Logic, Part 3

Faster SAT by Unit Propagation

Backtracking is Oblivious

The example:

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

Backtracking tries the atoms in some order.

But we can see immediately that b must be true.

Other forced assignments occur during the search.

Making the Search less naive

If we are in a situation like:

$$(\overset{\times}{\boxed{\neg b}} \vee \overset{\times}{\boxed{\neg c}} \vee \neg d)$$

then if the current valuation is to succeed in any way, it must be the case that $d : F$.

(because we need at least one literal in every clause to be true.)

Using this, we can make the search a little less naive.

6. Unit Propagation Step

(a) If there is a clause $C \vee a$and $[\![C]\!]\nu^? = F$, then we extend $\nu^?$ to:

$$\nu^?, a :_f T$$

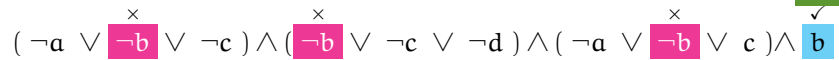
(b) If there is a clause $C \vee \neg a$ and $[\![C]\!]\nu^? = F$, then we extend $\nu^?$ to:

$$\nu^?, a :_f F$$

(Note: the a needn't necessarily appear at the end of the clause)

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee \neg b \vee c) \wedge b$$

1. $\square$ *do unit propagation...*



1. [] *do unit propagation...*
2. [b :$_f$ T]

$$(\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \neg c) \wedge (\overset{\times}{\neg b} \vee \neg c \vee \neg d) \wedge (\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee c) \wedge \overset{\checkmark}{b}$$

1. $[]$ *do unit propagation...*
2. $[b :_f T]$
3. $[b :_f T, a :_d T]$ *do unit propagation...*

$$(\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\checkmark}{\neg c} \vee \neg d) \wedge (\overset{\times}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$ *do unit propagation...*
2. $[b :_f T]$
3. $[b :_f T, a :_d T]$ *do unit propagation...*
4. $[b :_f T, a :_d T, c :_f F]$ *clause 3 failed, backtrack...*

University of Strathclyde Science

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \neg c) \wedge (\overset{\times}{\neg b} \vee \neg c \vee \neg d) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee c) \wedge \overset{\checkmark}{b}$$

1. $[]$ *do unit propagation...*
2. $[b :_f T]$
3. $[b :_f T, a :_d T]$ *do unit propagation...*
4. $[b :_f T, a :_d T, c :_f F]$ *clause 3 failed, backtrack...*
5. $[b :_f T, a :_f F]$

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\times}{\neg c} \vee \neg d) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$ *do unit propagation...*
2. $[b :_f T]$
3. $[b :_f T, a :_d T]$ *do unit propagation...*
4. $[b :_f T, a :_d T, c :_f F]$ *clause 3 failed, backtrack...*
5. $[b :_f T, a :_f F]$
6. $[b :_f T, a :_f F, c :_d T]$ *do unit propagation...*

University of Strathclyde Science

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\times}{\neg c} \vee \overset{\checkmark}{\neg d}) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$ *do unit propagation...*
2. $[b :_f T]$
3. $[b :_f T, a :_d T]$ *do unit propagation...*
4. $[b :_f T, a :_d T, c :_f F]$ *clause 3 failed, backtrack...*
5. $[b :_f T, a :_f F]$
6. $[b :_f T, a :_f F, c :_d T]$ *do unit propagation...*
7. $[b :_f T, a :_f F, c :_d T, d :_f F]$ SAT

University of Strathclyde Science

$$(\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\times}{\neg c}) \wedge (\overset{\times}{\neg b} \vee \overset{\times}{\neg c} \vee \overset{\checkmark}{\neg d}) \wedge (\overset{\checkmark}{\neg a} \vee \overset{\times}{\neg b} \vee \overset{\checkmark}{c}) \wedge \overset{\checkmark}{b}$$

1. $[]$ *do unit propagation...*
2. $[b :_f T]$
3. $[b :_f T, a :_d T]$ *do unit propagation...*
4. $[b :_f T, a :_d T, c :_f F]$ *clause 3 failed, backtrack...*
5. $[b :_f T, a :_f F]$
6. $[b :_f T, a :_f F, c :_d T]$ *do unit propagation...*
7. $[b :_f T, a :_f F, c :_d T, d :_f F]$ SAT

One backtrack vs. four without unit propagation.

2-SAT

If every clause has at most two literals, UP means less backtracking:

$$
\begin{array}{ll}
(\neg \mathrm{libD}_1 \vee \neg \mathrm{libD}_2) & \wedge\ (\neg \mathrm{libC}_1 \vee \neg \mathrm{libC}_2) \\
\wedge\ (\neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_2) & \wedge\ (\neg \mathrm{progA}_1 \vee \mathrm{libC}_1) \\
\wedge\ (\neg \mathrm{progA}_2 \vee \mathrm{libC}_2) & \wedge\ (\neg \mathrm{libC}_1 \vee \mathrm{libD}_2) \\
\wedge\ (\neg \mathrm{libC}_2 \vee \mathrm{libD}_2) & \wedge\ (\mathrm{progA}_1 \vee \mathrm{progA}_2)
\end{array}
$$

□

2-SAT

If every clause has at most two literals, UP means less backtracking:

$$(\neg \mathrm{libD}_1 \vee \neg \mathrm{libD}_2) \wedge (\neg \mathrm{libC}_1 \vee \neg \mathrm{libC}_2)$$

$$\wedge\ (\neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_2) \wedge (\neg \mathrm{progA}_1 \vee \mathrm{libC}_1)$$

$$\wedge\ (\neg \mathrm{progA}_2 \vee \mathrm{libC}_2) \wedge (\neg \mathrm{libC}_1 \vee \mathrm{libD}_2)$$

$$\wedge\ (\neg \mathrm{libC}_2 \vee \mathrm{libD}_2) \wedge (\mathrm{progA}_1 \vee \mathrm{progA}_2)$$

$$[\mathrm{progA}_1 :_{\mathrm{d}} \top]$$

2-SAT

If every clause has at most two literals, UP means less backtracking:

$$(\neg \mathrm{libD}_1 \vee \neg \mathrm{libD}_2) \wedge (\neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_2) \wedge (\neg \mathrm{progA}_2 \vee \mathrm{libC}_2) \wedge (\neg \mathrm{libC}_2 \vee \mathrm{libD}_2)$$

$$\wedge (\neg \mathrm{libC}_1 \vee \neg \mathrm{libC}_2) \wedge (\neg \mathrm{progA}_1 \vee \mathrm{libC}_1) \wedge (\neg \mathrm{libC}_1 \vee \mathrm{libD}_2) \wedge (\mathrm{progA}_1 \vee \mathrm{progA}_2)$$

$$[\mathrm{progA}_1 :_{\mathrm{d}} \mathsf{T}, \mathrm{progA}_2 :_{\mathrm{f}} \mathsf{F}]$$

2-SAT

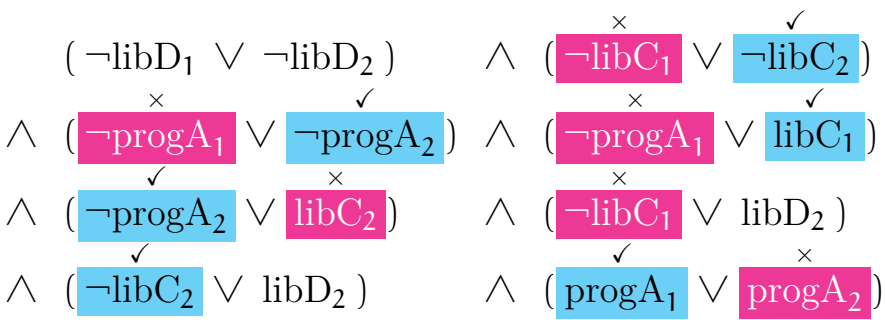
If every clause has at most two literals, UP means less backtracking:

$$
\begin{array}{ll}
(\neg \mathrm{libD}_1 \vee \neg \mathrm{libD}_2) & \wedge\ (\overset{\times}{\neg \mathrm{libC}_1} \vee \neg \mathrm{libC}_2) \\
\wedge\ (\overset{\times}{\neg \mathrm{progA}_1} \vee \overset{\checkmark}{\neg \mathrm{progA}_2}) & \wedge\ (\overset{\times}{\neg \mathrm{progA}_1} \vee \overset{\checkmark}{\mathrm{libC}_1}) \\
\wedge\ (\overset{\checkmark}{\neg \mathrm{progA}_2} \vee \mathrm{libC}_2) & \wedge\ (\overset{\times}{\neg \mathrm{libC}_1} \vee \mathrm{libD}_2) \\
\wedge\ (\neg \mathrm{libC}_2 \vee \mathrm{libD}_2) & \wedge\ (\overset{\checkmark}{\mathrm{progA}_1} \vee \overset{\times}{\mathrm{progA}_2})
\end{array}
$$

$$[\mathrm{progA}_1 :_{\mathsf{d}} \mathsf{T}, \mathrm{progA}_2 :_{\mathsf{f}} \mathsf{F}, \mathrm{libC}_1 :_{\mathsf{f}} \mathsf{T}]$$

2-SAT

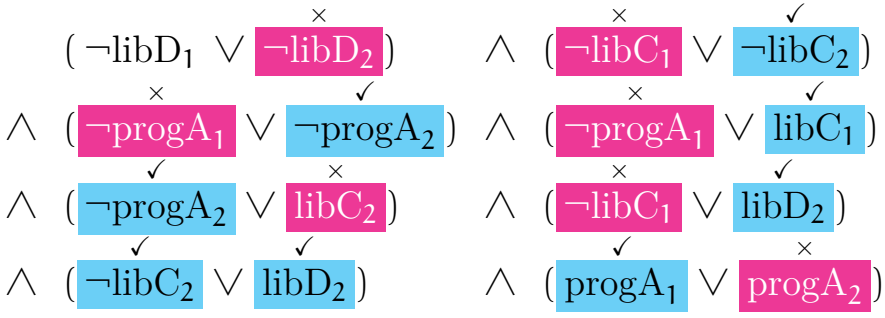
If every clause has at most two literals, UP means less backtracking:



$[\mathrm{progA}_1 :_\mathsf{d} \mathsf{T}, \mathrm{progA}_2 :_\mathsf{f} \mathsf{F}, \mathrm{libC}_1 :_\mathsf{f} \mathsf{T}, \mathrm{libC}_2 :_\mathsf{f} \mathsf{F}]$

2-SAT

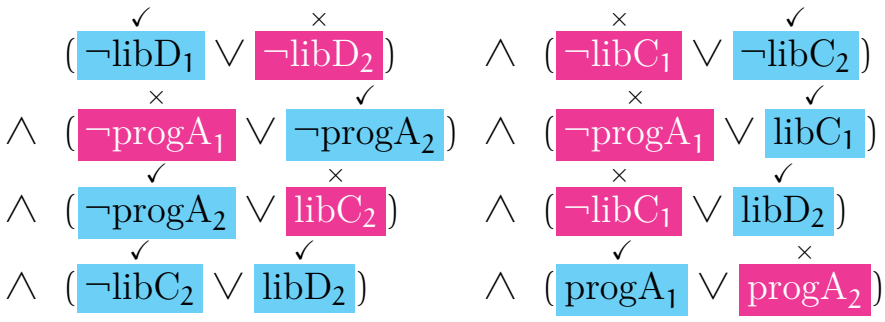
If every clause has at most two literals, UP means less backtracking:



$$[\mathrm{progA}_1 :_\mathsf{d} \mathsf{T}, \mathrm{progA}_2 :_\mathsf{f} \mathsf{F}, \mathrm{libC}_1 :_\mathsf{f} \mathsf{T}, \mathrm{libC}_2 :_\mathsf{f} \mathsf{F}, \mathrm{libD}_2 :_\mathsf{f} \mathsf{T}]$$

2-SAT

If every clause has at most two literals, UP means less backtracking:



$[\mathrm{progA}_1 :_\mathrm{d} \mathsf{T}, \mathrm{progA}_2 :_\mathrm{f} \mathsf{F}, \mathrm{libC}_1 :_\mathrm{f} \mathsf{T}, \mathrm{libC}_2 :_\mathrm{f} \mathsf{F}, \mathrm{libD}_2 :_\mathrm{f} \mathsf{T}, \mathrm{libD}_1 :_\mathrm{f} \mathsf{F}]$

2-SAT

If every clause has at most two literals,

- UP means at most one backtrack
- Means that we can solve the problem in polynomial time
- So for the n-SAT problem:
  - If $n \leq 2$, there is a fast polynomial time algorithm
  - If $n \geq 3$, no known general fast algorithm

Summary of the Rules 1

DecideTrue	$v^? \implies v^?, a :_d \mathsf{T}$		*if* a *is not assigned in* $v^?$
DecideFalse	$v^? \implies v^?, a :_d \mathsf{F}$		*if* a *is not assigned in* $v^?$
Success	$v^? \implies \mathsf{SAT}(v^?)$		*if* $v^?$ *makes all the clauses true.*

Summary of the Rules 2

BackTrack	$\nu_1^?, a :_d x, \nu_2^?$	$\Longrightarrow$	$\nu_1^?, a :_f \neg x$	*if $\nu_2^?$ is decision free*
Fail	$\nu^?$	$\Longrightarrow$	UNSAT	*if $\nu^?$ is decision free, and makes at least one clause false.*

Summary of the Rules 3

University of Strathclyde Science

$$\textsc{UnitPropTrue} \quad v^? \implies v^?, a :_f T \qquad \text{if there is a clause } C \vee a \text{ and } [\![C]\!](v^?) = F$$

$$\textsc{UnitPropFalse} \quad v^? \implies v^?, a :_f F \qquad \text{if there is a clause } C \vee \neg a \text{ and } [\![C]\!](v^?) = F$$

Real SAT solvers

Use very efficient data structures. (Key is very fast unit propagation)

Use heuristics to guide the search:

- Which atom to try next? (not just $a, b, c, \ldots$)
- Whether to try T or F first?

Incorporate additional rules:

- Non-chronological backjumping

  (skip several decision points by analysing conflicts)
- Clause learning to avoid doing the same work over again.
- "CDCL" (Conflict Driven Clause Learning)
- Random walk between possible valuations "WalkSAT".

Summary

- Unit Propagation speeds up SAT Solving

  (by using the structure of the problem)
- This makes 2-SAT very fast
- Real SAT Solvers are very sophisticated.

University of Strathclyde Science

Automating Logic, Part 4

Conversion to CNF

Conjunctive Normal Form (CNF)

$$
\begin{aligned}
& (\neg a \vee \neg b \vee \neg c) \\
\wedge \quad & (\neg b \vee \neg c \vee \neg d) \\
\wedge \quad & (\neg a \vee \neg b \vee c) \\
\wedge \quad & b
\end{aligned}
$$

1. Entire formula is a conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_n$
2. where each *clause* $C_i = L_{i,1} \vee L_{i,2} \vee \cdots \vee L_{i,k}$
3. where each *literal* $L_{i,j} = x_{i,j}$ or $L_{i,j} = \neg x_{i,j}$

Disjunctive Normal Form (DNF)

Disjunctive Normal Form (DNF) is similar, but swaps $\wedge$ and $\vee$.

$$\begin{array}{ll} & (\neg a \wedge \neg b \wedge \neg c) \\ \vee & (\neg b \wedge \neg c \wedge \neg d) \\ \vee & (\neg a \wedge \neg b \wedge c) \\ \vee & b \end{array}$$

1. Entire formula is a *dis*junction $D_1 \vee D_2 \vee \cdots \vee D_n$
2. where each *disjunct* $D_i = L_{i,1} \wedge L_{i,2} \wedge \cdots \wedge L_{i,k}$
3. where each *literal* $L_{i,j} = x_{i,j}$ or $L_{i,j} = \neg x_{i,j}$

Normal Forms and Satisfiability

University of Strathclyde Science

CNF
Each clause is a *constraint* and all constraints must be satisfied.

DNF
At least one of the disjuncts must be satisfied.

Exercise: How would you write a SAT Solver for formulas in DNF? Why don't we do this instead of CNF?

Conversion to CNF

Not every formula is in CNF, e.g.,

$$(A \wedge B) \rightarrow (B \wedge A)$$

What if we want to use a SAT solver to determine satisfiability?

Two ways to convert a formula to CNF that is "the same":

- "Multiplying out"
- Tseytin transformation

First we need to define what we mean by "the same".

Equivalent Formulas

Define two formulas P and Q to be *equivalent*, written

$$P \equiv Q$$

exactly when, for all valuations ν,

$$[\![P]\!]\nu = [\![Q]\!]\nu$$

Equivalent to both $P \models Q$ and $Q \models P$ being valid

Simplifying Implication

$$A \rightarrow B \equiv \neg A \vee B$$

valuation			P	Q
A	B	$\neg A$	$A \rightarrow B$	$\neg A \vee B$
F	F	T	T	T
F	T	T	T	T
T	F	F	F	F
T	T	F	T	T

Double Negation

Negating twice is the same as doing nothing:

$$A \equiv \neg\neg A$$

valuation		P	Q
A	$\neg A$	A	$\neg\neg A$
F	T	F	F
T	F	T	T

de Morgan's laws

Negation swaps $\wedge$ and $\vee$:

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

valuation					P	Q
A	B	$\neg A$	$\neg B$	$A \wedge B$	$\neg(A \wedge B)$	$\neg A \vee \neg B$
F	F	T	T	F	T	T
F	T	T	F	F	T	T
T	F	F	T	F	T	T
T	T	F	F	T	F	F

Similar for $\neg(A \vee B) \equiv \neg A \wedge \neg B$

Negation Normal Form (NNF)

Using the equivalences:

$$
\begin{aligned}
A \rightarrow B &\equiv \neg A \vee B \\
A &\equiv \neg\neg A \\
\neg(A \wedge B) &\equiv \neg A \vee \neg B \\
\neg(A \vee B) &\equiv \neg A \wedge \neg B
\end{aligned}
$$

We can *rewrite* any formula into an equivalent one with

1. No implications ($\rightarrow$s)
2. All negation signs on the atomic propositions

Example

$$
\begin{aligned}
& (a \wedge (a \rightarrow b)) \rightarrow c \\
\equiv\ & \neg(a \wedge (a \rightarrow b)) \vee c && \textit{converted } \rightarrow \\
\equiv\ & \neg(a \wedge (\neg a \vee b)) \vee c && \textit{converted } \rightarrow \\
\equiv\ & \neg a \vee \neg(\neg a \vee b) \vee c && \textit{converted } \wedge \textit{ to } \vee \\
\equiv\ & \neg a \vee (\neg\neg a \wedge \neg b) \vee c && \textit{converted } \vee \textit{ to } \wedge \\
\equiv\ & \neg a \vee (a \wedge \neg b) \vee c && \textit{converted double negation}
\end{aligned}
$$

Now in NNF, but not CNF.

"Push" $\vee$s into $\wedge$s

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

valuation						P	Q
A	B	C	$B \wedge C$	$A \vee B$	$A \vee C$	$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$
F	F	F	F	F	F	F	F
F	F	T	F	F	T	F	F
F	T	F	F	T	F	F	F
F	T	T	T	T	T	T	T
T	F	F	F	T	T	T	T
T	F	T	F	T	T	T	T
T	T	F	F	T	T	T	T
T	T	T	T	T	T	T	T

Conversion to CNF

$$
\begin{aligned}
& \neg a \vee (a \wedge \neg b) \vee c \\
\equiv & \qquad \textit{multiply out} \\
& \neg a \vee ((a \vee c) \wedge (\neg b \vee c)) \\
\equiv & \qquad \textit{multiply out} \\
& (\neg a \vee a \vee c) \wedge (\neg a \vee \neg b \vee c)
\end{aligned}
$$

Now in CNF.

(Can further simplify to: $(\neg a \vee \neg b \vee c)$)

Exponential Blowup

If we convert $(a \wedge b \wedge c) \vee (d \wedge e \wedge f) \vee (g \wedge h \wedge i)$ to CNF, we get:

$$(a \vee d \vee g) \wedge (a \vee d \vee h) \wedge (a \vee d \vee i) \wedge (a \vee e \vee g) \wedge (a \vee e \vee h) \wedge$$
$$(a \vee e \vee i) \wedge (a \vee f \vee g) \wedge (a \vee f \vee h) \wedge (a \vee f \vee i) \wedge (b \vee d \vee g) \wedge$$
$$(b \vee d \vee h) \wedge (b \vee d \vee i) \wedge (b \vee e \vee g) \wedge (b \vee e \vee h) \wedge (b \vee e \vee i) \wedge$$
$$(b \vee f \vee g) \wedge (b \vee f \vee h) \wedge (b \vee f \vee i) \wedge (c \vee d \vee g) \wedge (c \vee d \vee h) \wedge$$
$$(c \vee d \vee i) \wedge (c \vee e \vee g) \wedge (c \vee e \vee h) \wedge (c \vee e \vee i) \wedge (c \vee f \vee g) \wedge$$
$$(c \vee f \vee h) \wedge (c \vee f \vee i)$$

which has 27 clauses.

Summary

- SAT Solvers take their input in CNF
- Some problems are naturally in CNF
- Conversion by “multiplying out” can generate huge formulas
- We need something better

University of Strathclyde Science

Automating Logic, Part 5

Tseytin Transformation

Tseytin Transformation

The Tseytin transformation converts a formula into CNF with at most 3 times as many clauses as connectives in the original formula (versus potentially exponential for multiplying out the brackets).

1. Convert the formula into equations
   One connective $\rightsquigarrow$ one equation
2. Convert each equation into clauses
   One equation $\rightsquigarrow$ 2-3 clauses

Result is not equivalent, but *equisatisfiable*.

1. Name subformulas

Take the formula and name all the non-atomic subformulas.

Example:

$$\neg(a \wedge (\neg a \vee b)) \vee c$$

becomes:

$$\begin{aligned} x_1 &= x_2 \vee c \\ x_2 &= \neg x_3 \\ x_3 &= a \wedge x_4 \\ x_4 &= x_5 \vee b \\ x_5 &= \neg a \end{aligned}$$

2. Converting Equations to Clauses

Given an equation like $x = y \wedge z$, we want some clauses that are true for every valuation that satisfies the equation.

2. Converting Equations to Clauses

Given an equation like $x = y \wedge z$, we want some clauses that are true for every valuation that satisfies the equation.

Derive by conversion to CNF:

$$
\begin{aligned}
& x = y \wedge z \\
\equiv\ & (x \rightarrow (y \wedge z)) \wedge ((y \wedge z) \rightarrow x) \\
\equiv\ & (\neg x \vee (y \wedge z)) \wedge (\neg(y \wedge z) \vee x) \\
\equiv\ & (\neg x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z \vee x)
\end{aligned}
$$

2. Equations to Clauses

Take each equation $x = y \square z$ and turn it into clauses:

1. If $x = y \wedge z$, add

$$(\neg x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z \vee x)$$

2. If $x = y \vee z$, add

$$(y \vee z \vee \neg x) \wedge (\neg y \vee x) \wedge (\neg z \vee x)$$

3. If $x = \neg y$, add

$$(\neg y \vee \neg x) \wedge (y \vee x)$$

3. Assert the top level variable

If x is the name of the whole formula, add a clause with just x:

$$
\begin{aligned}
& \textit{equation 1} \\
\wedge\ & \textit{equation 2} \\
\wedge\ & \ldots \\
\wedge\ & x
\end{aligned}
$$

This asserts that our original formula must be true.

Example: $\neg(A \wedge B) \vee (B \wedge A)$

1. Name the subformulas:

$$x_1 = x_2 \vee x_4 \qquad x_2 = \neg x_3$$
$$x_3 = A \wedge B \qquad x_4 = B \wedge A$$

Example: $\neg(A \wedge B) \vee (B \wedge A)$

1. Name the subformulas:

$$x_1 = x_2 \vee x_4 \qquad x_2 = \neg x_3$$
$$x_3 = A \wedge B \qquad x_4 = B \wedge A$$

2+3. Generate clauses: (One line per equation)

$$\begin{aligned}
& (x_2 \vee x_4 \vee \neg x_1) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_4 \vee x_1) \\
\wedge\ & (\neg x_3 \vee \neg x_2) \wedge (x_3 \vee x_2) \\
\wedge\ & (\neg A \vee \neg B \vee x_3) \wedge (A \vee \neg x_3) \wedge (B \vee \neg x_3) \\
\wedge\ & (\neg B \vee \neg A \vee x_4) \wedge (B \vee \neg x_4) \wedge (A \vee \neg x_4) \\
\wedge\ & x_1
\end{aligned}$$

Efficiency

University of Strathclyde Science

In small examples, we get many clauses.

But we *always* get $\leq 3n$ clauses, where n number of connectives.

Multiplying out can result in exponential number of clauses.

Can also optimise (see the tutorial questions).

Not Equivalent!

The formulas generated by the Tseytin transformation are **not** equivalent to the original, because they have extra atomic propositions.

Example

If the original formula is

$$\neg A$$

the Tseytin transformed version is: (assuming we don't optimise)

$$(\neg A \vee \neg x) \wedge (A \vee x) \wedge x$$

Then $\{A : F, x : F\}$ satisfies the original, but not the transformed formula.

Equisatisfiable

If we write $\mathsf{Tseytin}(P)$ for the Tseytin translation of P, then:

1. If there exists a valuation ν_1 such that $[\![P]\!]\nu_1 = \mathsf{T}$, then there exists a valuation ν_2 such that $[\![\mathsf{Tseytin}(P)]\!]\nu_2 = \mathsf{T}$;
2. If there exists a valuation ν such that $[\![\mathsf{Tseytin}(P)]\!]\nu = \mathsf{T}$, then the valuation $\nu' = \nu$ without the additional x_is makes $[\![P]\!]\nu' = \mathsf{T}$.

This is called "equisatisfiability".

Example

$v = \{A : \mathsf{F}\}$ satisfies $\neg A$

The corresponding satisfying valuation for

$$(\neg A \vee \neg x) \wedge (A \vee x) \wedge x$$

is $\{A : \mathsf{F}, x : \mathsf{T}\}$.

A corresponding satisfying assignment always exists for the Tseytin transformation, because it is built from equations.

Summary

- Tseytin transformation converts formulas to CNF
- Generates $\leq 3n$ clauses, where n is the number of connectives
- Avoids exponential blowup
- Can be further optimised
- Result is *equisatisfiable*