CS208 (Semester 1) Week 2 : Logical Modelling I

Dr. Robert Atkey

Computer & Information Sciences

Logical Modelling I, Part 1

Package Installations

The Problem

1. We have a collection of packages

$$\text{progA} \qquad \text{progB} \qquad \text{libC} \qquad \text{libD} \qquad \cdots$$

The Problem

1. We have a collection of packages

$$\text{progA} \qquad \text{progB} \qquad \text{libC} \qquad \text{libD} \qquad \cdots$$

2. Each package has several versions: $\text{progA}_1, \text{progA}_2, \ldots$

The Problem

1. We have a collection of packages

$$\mathrm{progA} \qquad \mathrm{progB} \qquad \mathrm{libC} \qquad \mathrm{libD} \qquad \cdots$$

2. Each package has several versions: progA_1, progA_2, ...
3. Only *one* version of a package may be installed at a time

   installing two copies would overwrite each others's files

The Problem

1. We have a collection of packages

$$\mathrm{progA} \qquad \mathrm{progB} \qquad \mathrm{libC} \qquad \mathrm{libD} \qquad \cdots$$

2. Each package has several versions: $\mathrm{progA}_1, \mathrm{progA}_2, \ldots$
3. Only *one* version of a package may be installed at a time

   installing two copies would overwrite each others's files

4. Packages have dependencies: progA_1 depends: $\mathrm{libC}_1, \mathrm{libD}_2$

The Problem

1. We have a collection of packages

$$\mathrm{progA} \qquad \mathrm{progB} \qquad \mathrm{libC} \qquad \mathrm{libD} \qquad \cdots$$

2. Each package has several versions: $\mathrm{progA}_1, \mathrm{progA}_2, \ldots$
3. Only *one* version of a package may be installed at a time

   installing two copies would overwrite each others's files

4. Packages have dependencies: progA_1 depends: $\mathrm{libC}_1, \mathrm{libD}_2$
5. The user wants some packages installed.

Key Idea

1. Each package/version pair is an atomic proposition

$$\mathrm{progA}_1, \mathrm{progA}_2, \mathrm{progA}_3, \mathrm{libC}_1, \mathrm{libC}_2, \cdots$$

2. A valuation v represents a set of installed packages:
   - $v(\mathrm{progA}_1) = \mathsf{T}$ means progA_1 is installed;
   - $v(\mathrm{progA}_1) = \mathsf{F}$ means progA_1 is not installed.

Remember: a valuation is an assignment of T or F to every atomic proposition.

Example Valuations / Installations

$$v = \{\mathrm{progA}_1 : \mathsf{F}, \mathrm{progB}_1 : \mathsf{F}, \cdots : \mathsf{F}\}$$

Nothing is installed.

Example Valuations / Installations

$$v = \{\mathrm{progA}_1 : \mathsf{F}, \mathrm{progB}_1 : \mathsf{F}, \cdots : \mathsf{F}\}$$

Nothing is installed.

$$v = \{\mathrm{progA}_1 : \mathsf{T}, \mathrm{progB}_1 : \mathsf{T}, \cdots : \mathsf{F}\}$$

progA_1 and progB_1 are installed, and nothing else is.

Example Valuations / Installations

$$v = \{\mathrm{progA}_1 : \mathsf{T}, \mathrm{libC}_1 : \mathsf{T}, \cdots : \mathsf{F}\}$$

progA_1 and libC_1 are installed, and nothing else is.

Example Valuations / Installations

$$v = \{\mathrm{progA}_1 : \mathsf{T}, \mathrm{libC}_1 : \mathsf{T}, \cdots : \mathsf{F}\}$$

progA_1 and libC_1 are installed, and nothing else is.

$$v = \{\mathrm{progA}_1 : \mathsf{T}, \mathrm{progA}_2 : \mathsf{T}, \cdots : \mathsf{F}\}$$

progA_1 and progA_2 are installed, and nothing else is.

Adding Constraints

This valuation:

$$v = \{\mathrm{progA}_1 : \mathsf{T}, \mathrm{progA}_2 : \mathsf{T}, \cdots : \mathsf{F}\}$$

says we should install two versions of progA, which is impossible.

Adding Constraints

This valuation:

$$v = \{\mathrm{progA}_1 : \mathsf{T}, \mathrm{progA}_2 : \mathsf{T}, \cdots : \mathsf{F}\}$$

says we should install two versions of progA, which is impossible.

So not all valuations are sensible! We must *constrain* to the sensible valuations by writing down some formulas.

Adding Constraints

This valuation:

$$v = \{\mathrm{progA}_1 : \mathsf{T}, \mathrm{progA}_2 : \mathsf{T}, \cdots : \mathsf{F}\}$$

says we should install two versions of progA, which is impossible.

So not all valuations are sensible! We must *constrain* to the sensible valuations by writing down some formulas.

The formulas we write down to do this are called *constraints*.

Encoding incompatibility

Requirement: one only version of each package may be installed.

Encoding incompatibility

Requirement: one only version of each package may be installed.

For each package p and versions i, j, where $i < j$, we assume:

$$\neg p_i \vee \neg p_j$$

Exercise: why does this cover all the cases?

Encoding incompatibility

Requirement: one only version of each package may be installed.

For each package p and versions i, j, where $i < j$, we assume:

$$\neg p_i \vee \neg p_j$$

Exercise: why does this cover all the cases?

Example

Constraint: never install two versions of progA.

$$\neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_2, \neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_3, \neg \mathrm{progA}_2 \vee \neg \mathrm{progA}_3$$

Understanding the Constraint

Why does $\neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_2$ work?

progA_1	progA_2	$\neg \mathrm{progA}_1$	$\neg \mathrm{progA}_2$	$\neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_2$
F	F	T	T	T
T	F	F	T	T
F	T	T	F	T
T	T	F	F	F

The last line, where both are installed, is the case we want to disallow, and it is the only one assigned F.

Incompatibility Constraints

We have a collection of constraints:

For each package p and versions i, j, where $i < j$: $\neg p_i \vee \neg p_j$

Take all these constraints, $\wedge$ them together, and call it Incompat.

$$\textsc{Incompat} = (\neg progA_1 \vee \neg progA_2) \wedge (\neg progA_1 \vee \neg progA_3) \wedge \cdots$$

Filtering Valuations

Before: all valuations (installations) v
Now: only valuations such that $[\![\textsc{Incompat}]\!]v = \mathsf{T}$

Pay-off: We have a way of removing the nonsense valuations that allow multiple versions of the same package to be installed.

Encoding Dependencies

Requirement: Packages depend on other packages:

$$\begin{aligned} \mathrm{progA}_1 \quad &\textsf{depends}: \quad \mathrm{libC}_1, \mathrm{libD}_2 \\ \mathrm{progA}_2 \quad &\textsf{depends}: \quad \mathrm{libC}_2, \mathrm{libD}_2 \end{aligned}$$

Encoding Dependencies

University of Strathclyde Science

Requirement: Packages depend on other packages:

$$\begin{array}{lll} \mathrm{progA}_1 & \textsf{depends}: & \mathrm{libC}_1, \mathrm{libD}_2 \\ \mathrm{progA}_2 & \textsf{depends}: & \mathrm{libC}_2, \mathrm{libD}_2 \end{array}$$

As Formulas

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

Dependency Constraints

Dependency Constraints

For each package-version p_i with dependency q_j: $p_i \to q_j$.

Exercise: why is this the correct thing?

Dependency Constraints

For each package-version p_i with dependency q_j: $p_i \to q_j$.

Exercise: why is this the correct thing?

Gather these up as $\textsc{Dep}$:

$$\textsc{Dep} = (\mathrm{progA}_1 \to \mathrm{libC}_1) \land (\mathrm{progA}_1 \to \mathrm{libD}_1) \land \cdots$$

Understanding the Constraint

How to understand $\mathrm{progA}_1 \to \mathrm{libC}_1$?

progA_1	libC_1	$\mathrm{progA}_1 \to \mathrm{libC}_1$
F	F	T
T	F	F
F	T	T
T	T	T

The second last line, where progA_1 is installed, but its dependency libC_1 is not, is the case we want to disallow, and it is the only one assigned F.

Putting together the constraints

Original idea: valuations represent installations.

Putting together the constraints

Original idea: valuations represent installations.

Problem: Mutually incompatible packages can be installed.

Putting together the constraints

Original idea: valuations represent installations.

Problem: Mutually incompatible packages can be installed.

Solution: Impose the constraints INCOMPAT.

Putting together the constraints

Original idea: valuations represent installations.

Problem: Mutually incompatible packages can be installed.
Solution: Impose the constraints INCOMPAT.

Problem: Packages could be installed without their dependencies.

Putting together the constraints

Original idea: valuations represent installations.

Problem: Mutually incompatible packages can be installed.
Solution: Impose the constraints INCOMPAT.

Problem: Packages could be installed without their dependencies.
Solution: Impose the constraints DEP.

Putting together the constraints

Original idea: valuations represent installations.

Problem: Mutually incompatible packages can be installed.
Solution: Impose the constraints $\textsc{Incompat}$.

Problem: Packages could be installed without their dependencies.
Solution: Impose the constraints $\textsc{Dep}$.

In summary

Now we have,

$$[\![\textsc{Incompat} \wedge \textsc{Dep}]\!]v = \mathsf{T}$$

exactly when the valuation v is a sensible selection of packages.

Relating to Satisfiability

P is *satisfiable* if there exists a valuation v with $[\![P]\!]v = \mathrm{T}$.

Relating to Satisfiability

P is *satisfiable* if there exists a valuation v with $[\![P]\!]v = T$.

For the package installation problem:

1. If the formula $\textsc{Incompat} \wedge \textsc{Dep}$
   is satisfiable, then there is least one possible installation.
2. If the formula $\textsc{Incompat} \wedge \textsc{Dep} \wedge \mathrm{progA}_1$
   is satisfiable then progA_1 is installable (with its dependencies)
3. if $\textsc{Incompat} \wedge \textsc{Dep} \wedge (\mathrm{progA}_1 \vee \mathrm{progA}_2 \vee \mathrm{progA}_3)$
   is satisfiable then some version of progA is installable.

Example 1

Assume one version of each package: INCOMPAT is empty.

$$\textsc{Dep} \;=\; (\mathrm{progA}_1 \rightarrow \mathrm{libC}_1) \wedge (\mathrm{libC}_1 \rightarrow \mathrm{libD}_1) \wedge (\mathrm{libC}_1 \rightarrow \mathrm{libE}_1)$$

Example 1

Assume one version of each package: INCOMPAT is empty.

$$\textsc{Dep} \;=\; (\mathrm{progA}_1 \to \mathrm{libC}_1) \land (\mathrm{libC}_1 \to \mathrm{libD}_1) \land (\mathrm{libC}_1 \to \mathrm{libE}_1)$$

We would like to install progA_1.

Example 1

Assume one version of each package: $\textsc{Incompat}$ is empty.

$$\textsc{Dep} \;=\; (\mathrm{progA}_1 \to \mathrm{libC}_1) \land (\mathrm{libC}_1 \to \mathrm{libD}_1) \land (\mathrm{libC}_1 \to \mathrm{libE}_1)$$

We would like to install progA_1.

As a formula: Is this formula satisfiable?

$$\textsc{Incompat} \land \textsc{Dep} \land \mathrm{progA}_1$$

Example 1

Assume one version of each package: $\textsc{Incompat}$ is empty.

$$\textsc{Dep} \;=\; (\mathrm{progA}_1 \to \mathrm{libC}_1) \land (\mathrm{libC}_1 \to \mathrm{libD}_1) \land (\mathrm{libC}_1 \to \mathrm{libE}_1)$$

We would like to install progA_1.

As a formula: Is this formula satisfiable?

$$\textsc{Incompat} \land \textsc{Dep} \land \mathrm{progA}_1$$

Yes:

$$\{\mathrm{progA}_1 : \mathsf{T}, \mathrm{libC}_1 : \mathsf{T}, \mathrm{libD}_1 : \mathsf{T}, \mathrm{libE}_1 : \mathsf{T}\}$$

(Install everything)

Example 2

Assume two versions of libE:

$$\textsc{Incompat} = \neg \mathrm{libE}_1 \vee \neg \mathrm{libE}_2$$

Add a dependency:

$$\begin{aligned} \textsc{Dep} &= (\mathrm{progA}_1 \rightarrow \mathrm{libC}_1) \wedge (\mathrm{libC}_1 \rightarrow \mathrm{libD}_1) \wedge (\mathrm{libC}_1 \rightarrow \mathrm{libE}_1) \\ &\wedge (\mathrm{libD}_1 \rightarrow \mathrm{libE}_2) \end{aligned}$$

Example 2

Assume two versions of libE:

$$\textsc{Incompat} = \neg \mathrm{libE}_1 \vee \neg \mathrm{libE}_2$$

Add a dependency:

$$\begin{aligned} \textsc{Dep} \;&=\; (\mathrm{progA}_1 \to \mathrm{libC}_1) \wedge (\mathrm{libC}_1 \to \mathrm{libD}_1) \wedge (\mathrm{libC}_1 \to \mathrm{libE}_1) \\ &\wedge\; (\mathrm{libD}_1 \to \mathrm{libE}_2) \end{aligned}$$

As a formula: Is this formula satisfiable? $\textsc{Incompat} \wedge \textsc{Dep} \wedge \mathrm{progA}_1$

Example 2

Assume two versions of libE:

$$\textsc{Incompat} = \neg \mathrm{libE}_1 \vee \neg \mathrm{libE}_2$$

Add a dependency:

$$\begin{aligned} \textsc{Dep} \quad = & \quad (\mathrm{progA}_1 \rightarrow \mathrm{libC}_1) \wedge (\mathrm{libC}_1 \rightarrow \mathrm{libD}_1) \wedge (\mathrm{libC}_1 \rightarrow \mathrm{libE}_1) \\ \wedge & \quad (\mathrm{libD}_1 \rightarrow \mathrm{libE}_2) \end{aligned}$$

As a formula: Is this formula satisfiable? $\textsc{Incompat} \wedge \textsc{Dep} \wedge \mathrm{progA}_1$

No! $\textsc{Incompat} \wedge \mathrm{progA}_1$ force both libE_1 and libE_2 to be T, but this is disallowed by the $\textsc{Incompat}$ constraint. *"diamond dependency"*

Summary

- Package installations solved via Logical Modelling
- Valuations are installations
- Impose constraints to match requirments
- Satisfying valuations = viable installations

Logical Modelling I, Part 2

SAT Solving

SAT solvers

SATisfiability solvers.

The problem they solve:

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

Solving SAT

- 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?

But SAT is useful: Solving Problems

1. Package installations (last lecture)

   (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

(Recall: $P_1 \to P_2 \to Q$ is valid if $\neg(P_1 \to P_2 \to Q)$ is not satisfiable)

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.

Input for SAT solvers

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

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

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

For the package installation problems, we already have CNF:

$$
\left.\begin{array}{rl}
 & (\neg \mathrm{libD}_1 \vee \neg \mathrm{libD}_2) \\
\wedge & (\neg \mathrm{libC}_1 \vee \neg \mathrm{libC}_2) \\
\wedge & (\neg \mathrm{progA}_1 \vee \neg \mathrm{progA}_2)
\end{array}\right\} \quad \textsc{Incompat}
$$

$$
\left.\begin{array}{rl}
\wedge & (\neg \mathrm{progA}_1 \vee \mathrm{libC}_1) \\
\wedge & (\neg \mathrm{progA}_2 \vee \mathrm{libC}_2) \\
\wedge & (\neg \mathrm{libC}_1 \vee \mathrm{libD}_2) \\
\wedge & (\neg \mathrm{libC}_2 \vee \mathrm{libD}_2)
\end{array}\right\} \quad \textsc{Dep}
$$

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

A SAT Solver's job

Given clauses that look like:

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

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** $v^?$ 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: $v_1^? = [\mathsf{a} :_{\mathsf{d}} \mathsf{T}, \mathsf{b} :_{\mathsf{d}} \mathsf{F}, \mathsf{c} :_{\mathsf{f}} \mathsf{T}]$

$v_2^? = [\mathsf{a} :_{\mathsf{f}} \mathsf{F}, \mathsf{b} :_{\mathsf{d}} \mathsf{F}]$

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

$$v_1^?, a :_d x, v_2^?$$

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

We write

$$\mathrm{decisionfree}(v^?)$$

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

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

1. Initialisation

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

(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 $v^?$, then we can make a guess. We pick one of:

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

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

3. Success

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

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 green in every clause)

Sequence of (lucky) guesses

1. ▯

Example

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

(Need at least one green in every clause)

Sequence of (lucky) guesses

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

Example

$$(\overset{\checkmark}{\neg a} \lor \overset{\times}{\neg b} \lor \neg c) \land (\overset{\times}{\neg b} \lor \neg c \lor \neg d) \land (\overset{\checkmark}{\neg a} \lor \overset{\times}{\neg b} \lor c) \land \overset{\checkmark}{b}$$

(Need at least one green in every clause)

Sequence of (lucky) guesses

1. $[]$
2. $[a :_d \mathsf{F}]$
3. $[a :_d \mathsf{F}, b :_d \mathsf{T}]$

Example

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

(Need at least one green 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

$$(\overset{\checkmark}{\neg a} \lor \overset{\times}{\neg b} \lor \overset{\checkmark}{\neg c}) \land (\overset{\times}{\neg b} \lor \overset{\checkmark}{\neg c} \lor \overset{\checkmark}{\neg d}) \land (\overset{\checkmark}{\neg a} \lor \overset{\times}{\neg b} \lor \overset{\times}{c}) \land \overset{\checkmark}{b}$$

(Need at least one green 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.

But we can’t program “luck”!

4. Backtracking

If we have a partial valuation:

$$v_1^?, a :_d x, v_2^?$$

and $\mathrm{decisionfree}(v_2^?)$ (so $a : x$ was our most recent guess).

Then we backtrack (throw away $v_2^?$) and change our mind:

$$v_1^?, a :_f \neg x$$

marking the assignment as forced.

5. Failure

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

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

1. $[]$
2. $[a :_d \mathsf{T}]$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. $[]$
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

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.

Logical Modelling I, 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 $[\![C]\!]v^? = \mathsf{F}$, then we extend $v^?$ to:

$$v^?, a :_f \mathsf{T}$$

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

$$v^?, a :_f \mathsf{F}$$

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

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

1. ▯ *do unit propagation...*

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

1. $[]$ *do unit propagation...*
2. $[b :_f \mathsf{T}]$

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

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. $[b :_f \mathsf{T}]$
3. $[b :_f \mathsf{T}, a :_d \mathsf{T}]$ *do unit propagation...*
4. $[b :_f \mathsf{T}, a :_d \mathsf{T}, c :_f \mathsf{F}]$ *clause 3 failed, backtrack...*

University of Strathclyde Science

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

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

University of Strathclyde Science

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

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:

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

▯

2-SAT

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

$$
\begin{array}{lll}
& (\neg \mathrm{libD}_1 \lor \neg \mathrm{libD}_2) & \land\ (\neg \mathrm{libC}_1 \lor \neg \mathrm{libC}_2) \\
\land & (\overset{\times}{\boxed{\neg \mathrm{progA}_1}} \lor \neg \mathrm{progA}_2) & \land\ (\overset{\times}{\boxed{\neg \mathrm{progA}_1}} \lor \mathrm{libC}_1) \\
\land & (\neg \mathrm{progA}_2 \lor \mathrm{libC}_2) & \land\ (\neg \mathrm{libC}_1 \lor \mathrm{libD}_2) \\
\land & (\neg \mathrm{libC}_2 \lor \mathrm{libD}_2) & \land\ (\overset{\checkmark}{\boxed{\mathrm{progA}_1}} \lor \mathrm{progA}_2)
\end{array}
$$

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

2-SAT

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

$$
\begin{array}{lll}
 & (\neg \mathrm{libD}_1 \lor \neg \mathrm{libD}_2) & \land\ (\neg \mathrm{libC}_1 \lor \neg \mathrm{libC}_2) \\
\land & (\overset{\times}{\neg \mathrm{progA}_1} \lor \overset{\checkmark}{\neg \mathrm{progA}_2}) & \land\ (\overset{\times}{\neg \mathrm{progA}_1} \lor \mathrm{libC}_1) \\
\land & (\overset{\checkmark}{\neg \mathrm{progA}_2} \lor \mathrm{libC}_2) & \land\ (\neg \mathrm{libC}_1 \lor \mathrm{libD}_2) \\
\land & (\neg \mathrm{libC}_2 \lor \mathrm{libD}_2) & \land\ (\overset{\checkmark}{\mathrm{progA}_1} \lor \overset{\times}{\mathrm{progA}_2})
\end{array}
$$

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

2-SAT

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

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

$$[\mathrm{progA}_1 :_d \mathsf{T}, \mathrm{progA}_2 :_f \mathsf{F}, \mathrm{libC}_1 :_f \mathsf{T}]$$

2-SAT

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

$$
\begin{array}{llll}
 & (\neg \mathrm{libD}_1 \lor \neg \mathrm{libD}_2) & \land & (\overset{\times}{\neg \mathrm{libC}_1} \lor \overset{\checkmark}{\neg \mathrm{libC}_2}) \\
\land & (\overset{\times}{\neg \mathrm{progA}_1} \lor \overset{\checkmark}{\neg \mathrm{progA}_2}) & \land & (\overset{\times}{\neg \mathrm{progA}_1} \lor \overset{\checkmark}{\mathrm{libC}_1}) \\
\land & (\overset{\checkmark}{\neg \mathrm{progA}_2} \lor \overset{\times}{\mathrm{libC}_2}) & \land & (\overset{\times}{\neg \mathrm{libC}_1} \lor \mathrm{libD}_2) \\
\land & (\overset{\checkmark}{\neg \mathrm{libC}_2} \lor \mathrm{libD}_2) & \land & (\overset{\checkmark}{\mathrm{progA}_1} \lor \overset{\times}{\mathrm{progA}_2})
\end{array}
$$

$$[\mathrm{progA}_1 :_d \mathsf{T}, \mathrm{progA}_2 :_f \mathsf{F}, \mathrm{libC}_1 :_f \mathsf{T}, \mathrm{libC}_2 :_f \mathsf{F}]$$

2-SAT

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

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

$$[\mathrm{progA}_1 :_d \mathsf{T}, \mathrm{progA}_2 :_f \mathsf{F}, \mathrm{libC}_1 :_f \mathsf{T}, \mathrm{libC}_2 :_f \mathsf{F}, \mathrm{libD}_2 :_f \mathsf{T}]$$

University of Strathclyde Science

2-SAT

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

$$
\begin{array}{rlcl}
 & (\overset{\checkmark}{\neg \mathrm{libD}_1} \lor \overset{\times}{\neg \mathrm{libD}_2}) & \land & (\overset{\times}{\neg \mathrm{libC}_1} \lor \overset{\checkmark}{\neg \mathrm{libC}_2}) \\
\land & (\overset{\times}{\neg \mathrm{progA}_1} \lor \overset{\checkmark}{\neg \mathrm{progA}_2}) & \land & (\overset{\times}{\neg \mathrm{progA}_1} \lor \overset{\checkmark}{\mathrm{libC}_1}) \\
\land & (\overset{\checkmark}{\neg \mathrm{progA}_2} \lor \overset{\times}{\mathrm{libC}_2}) & \land & (\overset{\times}{\neg \mathrm{libC}_1} \lor \overset{\checkmark}{\mathrm{libD}_2}) \\
\land & (\overset{\checkmark}{\neg \mathrm{libC}_2} \lor \overset{\checkmark}{\mathrm{libD}_2}) & \land & (\overset{\checkmark}{\mathrm{progA}_1} \lor \overset{\times}{\mathrm{progA}_2})
\end{array}
$$

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

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

Summary of the Rules 2

Backtrack	$v_1^?, a :_d x, v_2^?$	$\Longrightarrow$	$v_1^?, a :_f \neg x$	*if $v_2^?$ is decision free*
Fail	$v^?$	$\Longrightarrow$	UNSAT	*if $v^?$ is decision free, and makes at least one clause false.*

Summary of the Rules 3

University of Strathclyde Science

$$\textsc{UnitPropTrue} \quad v^{?} \implies v^{?}, a :_{f} \mathsf{T} \qquad \textit{if there is a clause } C \lor a \textit{ and } [\![C]\!](v^{?}) = \mathsf{F}$$

$$\textsc{UnitPropFalse} \quad v^{?} \implies v^{?}, a :_{f} \mathsf{F} \qquad \textit{if there is a clause } C \lor \neg a \textit{ and } [\![C]\!](v^{?}) = \mathsf{F}$$

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".

Further Reading

A blog post with a Python implementation:

Understanding SAT by Implementing a Simple SAT Solver in Python
Sahand Saba

`https://sahandsaba.com/understanding-sat-by-implementing-a-simple-sat-solver-in-python.html`

Another blog post with more formalism:

A Primer on Boolean Satisfiability
Emina Torlak

`https://homes.cs.washington.edu/~emina/blog/2017-06-23-a-primer-on-sat.html`

See also the links at the end for lots more detail.

More Further Reading

For more breadth and detail than you could possibly imagine:

The Art of Computer Programming: 7.2.2.2 Satisfiability

Draft: Volume 4B, Pre-fascicle 6A

Donald E. Knuth

`https://cs.stanford.edu/~knuth/fasc6a.ps.gz`

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.