Review

Last week

- Propositional Logic: Semantics
- Satisfiability and Tautologies
- Propositional Connectives and Boolean Functions
- Compactness

Outline

- Computability and Decidability
- Boolean Circuits
- Boolean Satisfiability (SAT)
- Binary Decision Diagrams (BDD's)

Computability

The important notion of *computability* relies on a formal model of computation.

Many formal models have been proposed:

1. General recursive functions defined by means of an equation calculus (Gödel-Herbrand-Kleene)
2. λ-definable functions (Church)
3. μ-recursive functions and partial recursive functions (Gödel-Kleene)
4. Functions computable by finite machines known as Turing machines (Turing)
5. Functions defined from canonical deduction systems (Post)
6. Functions given by certain algorithms over a finite alphabet (Markov)
7. Universal Register Machine-computable functions (Shepherdson-Sturgis)

Fundamental Result

All of these (and many other) models of computation are equivalent. That is, they give rise to the same class of functions.

Computability and Decidability

All of these models are equivalent to what can be achieved by a computer with any standard programming language, given arbitrary (but finite) time and memory.

Church's Thesis

A notion known as Church's thesis states that all models of computation are either equivalent to or less powerful than those just described.

We will accept Church's thesis and thus define a function to be *computable* if we can describe precisely (using any model of computation) how to compute it. Such a description will be called an *effective procedure*.

All of these models are equivalent to what can be achieved by a computer with any standard programming language, given arbitrary (but finite) time and memory.

Church's Thesis

A notion known as Church's thesis states that all models of computation are either equivalent to or less powerful than those just described.

We will accept Church's thesis and thus define a function to be *computable* if we can describe precisely (using any model of computation) how to compute it. Such a description will be called an *effective procedure*.

Decidability

Given a universal set U, a set $S \subseteq U$ is decidable if there exists a computable function $f : U \to \{\mathbf{F}, \mathbf{T}\}$ such that $f(x) = \mathbf{T}$ iff $x \in S$.

Decidability of W

Earlier, we presented an algorithm which, given any expression α determines whether the expression is well-formed. Thus, the set W of well-formed formulas is decidable.

Decidability

Some decidable sets

- For a given finite set of *wffs* Σ, the set of all *tautological consequences* of Σ (i.e. $\{\alpha \mid \Sigma \models \alpha\}$) is decidable.

  The truth table algorithm given earlier decides $\Sigma \models \alpha$.

- The set of tautologies is decidable.

  The set of tautologies is just the set of tautological consequences of the empty set.

Existence of undecidable sets

A simple argument shows the existence of undecidable sets of expressions: an algorithm is completely determined by its finite description. Thus, there are only countably many effective procedures. But there are uncountably many sets of expressions.

Why?

The set of expressions is countably infinite. Therefore, its power set is uncountable.

Semi-Decidability

Suppose we wish to determine whether $\Sigma \models \alpha$ where Σ is infinite. In general, this is not decidable.

But we can obtain a weaker result:

A set A is *semi-decidable* (or *effectively enumerable*) if there is an effective procedure which lists, in some order, every member of A.

Note that if A is infinite, then the procedure will never finish, but every member of A must appear in the list after some finite amount of time.

Theorem

A set A of expressions is effectively enumerable iff there is an effective procedure which, given any expression α, produces the answer "yes" iff $\alpha \in A$.

Proof

If A is effectively enumerable, then we simply enumerate its members and check each one to see if it is equivalent to α. If it is, we return "yes" and stop. Otherwise, we keep going. Thus, if $\alpha \in A$, the procedure produces "yes". If $\alpha \notin A$, the procedure runs forever.

Proof, continued

On the other hand, suppose that we have an effective procedure P which produces "yes" iff $\alpha \in A$. To produce an enumeration of A, we proceed as follows. First enumerate all expressions:

$$\epsilon_1, \epsilon_2, \epsilon_3, \ldots$$

Then proceed as follows.

- Break the procedure P into a finite number of "steps".
- Run P on ϵ_1 for 1 step.
- Run P on ϵ_1 for 2 steps, and then run P on ϵ_2 for 2 steps.
- ...
- Run P on each of $\epsilon_1, \ldots, \epsilon_n$ for n steps each
- ...

If at any time, the procedure P produces "yes", then we list the expression which produced "yes" and continue.

This procedure will eventually enumerate all members of A.

Semi-Decidability

Theorem

A set is decidable iff both it and its complement (with respect to a given universal set) are effectively enumerable.

Proof

Alternate between running the procedure for the set and the procedure for its complement. One of them will eventually produce “yes”.

Properties of decidable and semi-decidable sets

Decidable sets are closed under union, intersection, and complement.

Semi-decidable sets are closed under union and intersection.

Theorem

If Σ is an effectively enumerable set of *wffs*, then the set of tautological consequences of Σ is effectively enumerable.

Proof

Consider an enumeration of the elements of Σ:

$$\sigma_1, \sigma_2, \sigma_3, \ldots$$

By the compactness theorem, $\Sigma \models \alpha$ iff $\{\sigma_1, \ldots, \sigma_n\} \models \alpha$ for some n.

Hence, it is sufficient to successively test:

$$\begin{array}{lll} \emptyset & \models & \alpha \\ \{\sigma_1\} & \models & \alpha \\ \{\sigma_1, \sigma_2\} & \models & \alpha \\ \ldots & & \end{array}$$

If any of these conditions is met (each of which is decidable), the answer is “yes”.

Theorem

If Σ is an effectively enumerable set of *wffs*, then the set of tautological consequences of Σ is effectively enumerable.

Proof (continued)

This demonstrates that there is an effective procedure that, given any *wff* α, will output "yes" iff α is a tautological consequence of Σ.

Thus, the set of tautological consequences of Σ is effectively enumerable.

Boolean Satisfiability (SAT)

As we mentioned last time, Boolean satisfiability or *SAT* is widely useful for a variety of problems.

SAT was the first problem ever shown to be $\mathcal{NP}$-complete:

> S. A. Cook. The Complexity of Theorem Proving Procedures. *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, 151-158, 1971.

This means that:

- Unless $\mathcal{P} = \mathcal{NP}$, we will never find a polynomial algorithm to solve SAT.
- If we can nonetheless improve algorithms for SAT, there are many other problems that could benefit.

Converting to CNF

Given an arbitrary formula in propostitional logic, most algorithms for determining satisfiability first convert the formula into *conjunctive normal form (CNF)*.

Some definitions:

- A *literal* is a propositional variable or its negation
- A *clause* is a disjunction of one or more literals
- A formula is in *CNF* if it consists of a conjunction of clauses
- A propositional symbol occurs *positively* if it occurs unnegated in a clause.
- A propositional symbol occurs *negatively* if it occurs negated in a clause.

Converting to CNF

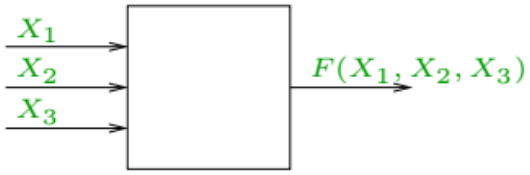
Examples

- Literals: P_i, $\neg P_i$
- Clauses: $(P_1 \vee \neg P_3 \vee P_5)$, $(P_2 \vee \neg P_2)$
- CNF: $(P_1 \vee \neg P_3) \wedge (\neg P_2 \vee P_3 \vee P_5)$
- In the above formula, P_1 occurs positively and P_2 occurs negatively

To provide intuition for how to convert to CNF, we first explore the connection between propositional formulas and Boolean circuits.

Boolean Gates

Consider an electrical device having n inputs and one output. Assume that to each input we apply a signal that is either $\mathbf{T}$ or $\mathbf{F}$, and that this uniquely determines whether the output is $\mathbf{T}$ or $\mathbf{F}$.

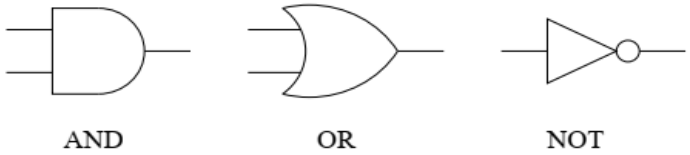


The behavior of such a device is described by a Boolean function:

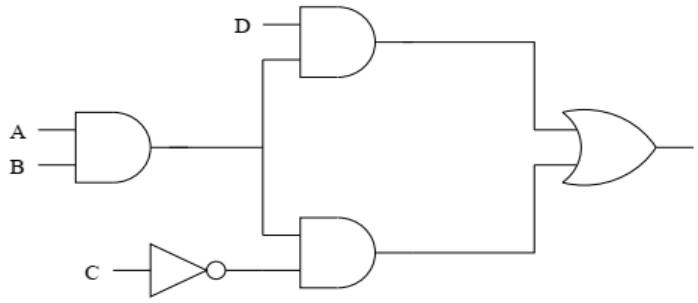
$$F(X_1, \ldots, X_n) = \text{the output signal given the input signals } X_1, \ldots, X_n.$$

We call such a device a *Boolean gate*.

The most common Boolean gates are *AND*, *OR*, and *NOT* gates.



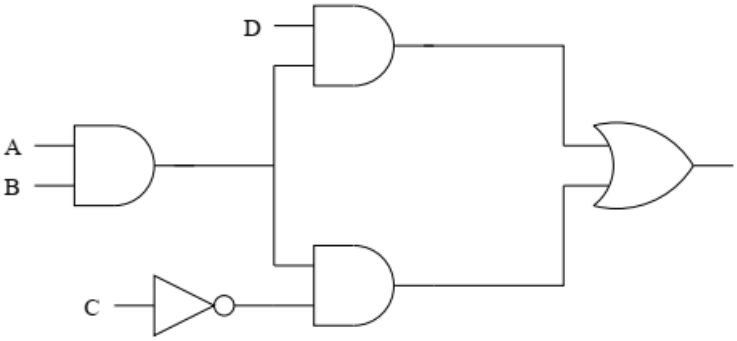
The inputs and outputs of Boolean gates can be connected together to form a *combinational Boolean circuit*.



A combinational Boolean circuit corresponds to a *directed acyclic graph* (DAG) whose leaves are *inputs* and each of whose nodes is labeled with the name of a Boolean gate. One or more of the nodes may be identified as outputs.

A common question with Boolean circuits is whether it is possible to set an output to true (e.g. when the output represents an *error* signal).

The inputs and outputs of Boolean gates can be connected together to form a *combinational Boolean circuit*.



There is a natural correspondence between Boolean circuits and formulas of propositional logic. The formula corresponding to the above circuit is:

$$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C).$$

A satisfying assignment for this formula gives the values that must be applied to the inputs of the circuit in order to set the output of the circuit to true.

In this lecture, we will refer to propositional symbols such as A, B, etc. as *propositional variables*.

Sharing Sub-Expressions

$$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C)$$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation: the formula $A \wedge B$ appears twice. For larger circuits, this kind of redundancy can result in an exponential blow-up in the size of the corresponding formula.

$$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C)$$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation: the formula $A \wedge B$ appears twice. For larger circuits, this kind of redundancy can result in an exponential blow-up in the size of the corresponding formula.

We can overcome this inefficiency by replacing the redundant sub-expression with a new place-holder variable. We then conjoin a new formula which says that the new variable is equivalent to the replaced expression:

$$((D \wedge E) \vee (E \wedge \neg C)) \wedge (E \leftrightarrow (A \wedge B))$$

$$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C)$$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation: the formula $A \wedge B$ appears twice. For larger circuits, this kind of redundancy can result in an exponential blow-up in the size of the corresponding formula.

We can overcome this inefficiency by replacing the redundant sub-expression with a new place-holder variable. We then conjoin a new formula which says that the new variable is equivalent to the replaced expression:

$$((D \wedge E) \vee (E \wedge \neg C)) \wedge (E \leftrightarrow (A \wedge B))$$

Note that the new formula is *not* tautologically equivalent to the original formula (why?).

But it *is* equisatisfiable (i.e. the original formula is satisfiable iff the new formula is satisfiable). Since we are only concerned with the *satisfiability* of the formula, this is sufficient.

Converting to CNF

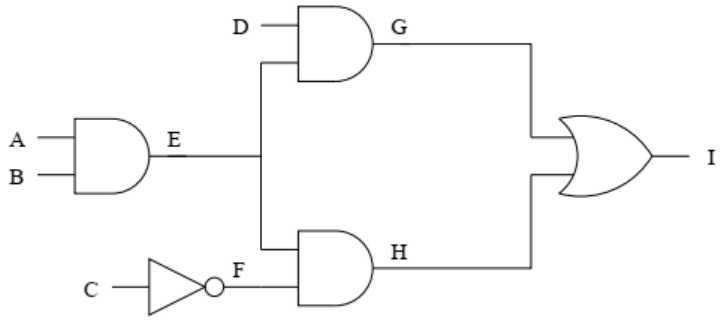
This same idea is behind a simple algorithm for converting any propositional formula (or an associated Boolean circuit) into an equisatisfiable formula in conjunctive normal form (CNF) in linear time and space. We will view the formula or circuit as a DAG.

1. Label each non-leaf node of the DAG with a new propositional variable.

2. Construct a conjunction of disjunctive clauses which relate the inputs of that node to its output (the new propositional variable)

3. The conjunction of all of these clauses together with a single clause consisting of the variable for the root node is satisfiable iff the original formula is satisfiable.

Converting to CNF: Example



$$(A \wedge B) \leftrightarrow E$$
$$((A \wedge B) \rightarrow E) \wedge (E \rightarrow (A \wedge B))$$
$$(\neg(A \wedge B) \vee E) \wedge (\neg E \vee (A \wedge B))$$
$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B)$$

$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge$$
$$(\neg C \vee \neg F) \wedge (C \vee F) \wedge$$
$$(\neg D \vee \neg E \vee G) \wedge (\neg G \vee D) \wedge (\neg G \vee E) \wedge$$
$$(\neg E \vee \neg F \vee H) \wedge (\neg H \vee E) \wedge (\neg H \vee F) \wedge$$
$$(G \vee H \vee \neg I) \wedge (I \vee \neg G) \wedge (I \vee \neg H) \wedge$$
$$(I)$$

CNF: Alternative Notations

$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge$$
$$(\neg C \vee \neg F) \wedge (C \vee F) \wedge$$
$$(\neg D \vee \neg E \vee G) \wedge (\neg G \vee D) \wedge (\neg G \vee E) \wedge$$
$$(\neg E \vee \neg F \vee H) \wedge (\neg H \vee E) \wedge (\neg H \vee F) \wedge$$
$$(G \vee H \vee \neg I) \wedge (I \vee \neg G) \wedge (I \vee \neg H) \wedge$$
$$(I)$$

$$(A' + B' + E)(E' + A)(E' + B)$$

$(C' + F')(C + F)$
$(D' + E' + G)(G' + D)(G' + E)$
$(E' + F' + H)(H' + E)(H' + F)$
$(G + H + I')(I + G')(I + H')$
(I)

CNF: Alternative Notations

DIMACS standard

Each variable is represented by a positive integer. A negative integer refers to the negation of the variable. Clauses are given as sequences of integers separated by spaces. A 0 terminates the clause.

$(A' + B' + E)(E' + A)(E' + B)$	-1 -2 5 0	-5 1 0	-5 2 0
$(C' + F')(C + F)$	-3 -6 0	3 6 0	
$(D' + E' + G)(G' + D)(G' + E)$	-4 -5 7 0	-7 4 0	-7 5 0
$(E' + F' + H)(H' + E)(H' + F)$	-5 -6 8 0	-8 5 0	-8 6 0
$(G + H + I')(I + G')(I + H')$	7 8 -9 0	9 -7 0	9 -8 0
(I)	9 0		

Davis-Putnam Algorithm

From now on, unless otherwise indicated, we assume formulas are in CNF, or, equivalently, that we have a set of clauses to check for satisfiability (i.e. the conjunction is implicit).

The first algorithm to try something more sophisticated than the truth-table method was the *Davis-Putnam (DP)* algorithm, published in 1960.

It is often confused with the later, more popular algorithm presented by Davis, Logemann, and Loveland in 1962, which we will refer to as *Davis-Putnam-Logemann-Loveland (DPLL)*.

We first consider the original DP algorithm.

There are three satisfiability-preserving transformations in DP.

- The 1-literal rule
- The affirmative-negative rule
- The rule for eliminating atomic formulas

The first two steps reduce the total number of literals in the formula.

The last step reduces the number of variables in the formula.

By repeatedly applying these rules, eventually we obtain a formula containing an empty clause, indicating unsatisfiability, or a formula with no clauses, indicating satisfiability.

The 1-literal rule

Also called *unit propagation*.

Suppose (p) is a unit clause (clause containing only one literal). Let $-p$ denote the negation of p where double negation is collapsed (i.e. $-\neg q \equiv q$).

- Remove all instances of $-p$ from clauses in the formula (shortening the corresponding clauses).
- Remove all clauses containing p (including the unit clause itself).

Davis-Putnam Algorithm

The affirmative-negative rule

Also called the *pure literal rule*.

If a literal appears *only positively* or *only negatively*, delete all clauses containing that literal.

Why does this preserve satisfiability?

Rule for eliminating atomic formulas

Also called the *resolution rule*.

- Choose a propositional symbol p which occurs positively in at least one clause and negatively in at least one other clause.
- Let P be the set of all clauses in which p occurs positively.
- Let N be the set of all clauses in which p occurs negatively.
- Replace the clauses in P and N with those obtained by resolution on p using all pairs of clauses from P and N.

For a single pair of clauses, $(p \vee l_1 \vee \cdots \vee l_m)$ and $(\neg p \vee k_1 \vee \cdots \vee k_n)$, *resolution on* p forms the new clause $(l_1 \vee \cdots \vee l_m \vee k_1 \vee \cdots \vee k_n)$.

DPLL Algorithm

In the worst case, the resolution rule can cause a quadratic expansion every time it is applied.

For large formulas, this can quickly exhaust the available memory.

The DPLL algorithm replaces resolution with a *splitting rule*.

- Choose a propositional symbol p occuring in the formula.
- Let Δ be the current set of clauses.
- Test the satisfiability of $\Delta \cup \{(p)\}$.
- If satisfiable, return *true*.
- Otherwise, return the result of testing $\Delta \cup \{(\neg p)\}$ for satisfiability.

Some Experimental Results

Problem	tautology	dptaut	dplltaut
prime 3	0.00	0.00	0.00
prime 4	0.02	0.06	0.04
prime 9	18.94	2.98	0.51
prime 10	11.40	3.03	0.96
prime 11	28.11	2.98	0.51
prime 16	>1 hour	out of memory	9.15
prime 17	>1 hour	out of memory	3.87
ramsey 3 3 5	0.03	0.06	0.02
ramsey 3 3 6	5.13	8.28	0.31
mk_adder_test 3 2	>>1 hour	6.50	7.34
mk_adder_test 4 2	>>1 hour	22.95	46.86
mk_adder_test 5 2	>>1 hour	44.83	170.98
mk_adder_test 5 3	>>1 hour	38.27	250.16
mk_adder_test 6 3	>>1 hour	out of memory	1186.4
mk_adder_test 7 3	>>1 hour	out of memory	3759.9

The DPLL algorithm is the basis for most modern SAT solvers.

We will look at DPLL in more detail, but first we consider one alternative algorithm.

Incomplete SAT: GSAT

```
Input: a set of clauses F, MAX-FLIPS, MAX-TRIES
Output: a satisfying truth assignment of F
        or ∅, if none found
for i := 1 to MAX-TRIES
  v := a randomly generated truth assignment
  for j := 1 to MAX-FLIPS
    if v satisfies F then return v
    p := a propositional variable such that a
         change in its truth assignment gives the
         largest increase in the total number of
         clauses of F that are satisfied by v
    v := v with the assignment to p reversed
  end for
end for
return ∅
```

Abstract DPLL

We now return to DPLL. To facilitate a deeper look at DPLL, we use a high-level framework called *Abstract DPLL*.

- Abstract DPLL uses *states* and *transitions* to model the progress of the algorithm.
- Most states are of the form $M \parallel F$, where
  - M is a *sequence of* annotated *literals* denoting a partial truth assignment, and
  - F is the CNF formula being checked, represented as a *set of clauses*.

- The *initial state* is $\emptyset \parallel F$, where F is to be checked for satisfiability.
- Transitions between states are defined by a set of *conditional transition rules*.

The *final state* is either:

- a special fail state: $fail$, if F is unsatisfiable, or
- $M \parallel G$, where G is a CNF formula equisatisfiable with the original formula F, and M satisfies G

We write $M \models C$ to mean that for every truth assignment v, $v(M) = true$ implies $v(C) = true$.

UnitProp :

$$M \parallel F, C \vee l \implies M\ l \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

PureLiteral :

$$M \parallel F \implies M\ l \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F \\ -l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \implies M\ l^{\mathrm{d}} \parallel F \quad \textbf{if} \begin{cases} \mathrm{l} \text{ or } \neg\mathrm{l} \text{ occurs in a clause of } F \\ \mathrm{l} \text{ is undefined in M} \end{cases}$$

Backtrack :

$$M\ l^{\mathrm{d}}\ N \parallel F, C \implies M \neg l \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathrm{d}}\ N \models \neg C \\ \mathrm{N} \text{ contains no decision literals} \end{cases}$$

Fail :

$$M \parallel F, C \implies fail \quad \textbf{if} \begin{cases} M \models \neg C \\ \mathrm{M} \text{ contains no decision literals} \end{cases}$$

$$\emptyset \parallel \quad 1 \vee \overline{2},\ \ \overline{1} \vee \overline{2},\ \ 2 \vee 3,\ \ \overline{3} \vee 2,\ \ 1 \vee 4$$

Example

$$
\begin{array}{rcll}
\emptyset & \| & 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{3}\vee2,\ 1\vee4 & \Longrightarrow \quad \text{(PureLiteral)} \\
4 & \| & 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{3}\vee2,\ 1\vee4 & \Longrightarrow \quad \text{(Decide)} \\
4\ 1^{\mathsf{d}} & \| & 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{3}\vee2,\ 1\vee4 & \Longrightarrow \quad \text{(UnitProp)} \\
4\ 1^{\mathsf{d}}\ \overline{2} & \| & 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{3}\vee2,\ 1\vee4 & \Longrightarrow \quad \text{(UnitProp)} \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 & \| & 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{3}\vee2,\ 1\vee4 & \Longrightarrow \quad \text{(Backtrack)} \\
4\ \overline{1} & \| & 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{3}\vee2,\ 1\vee4 & \Longrightarrow \quad \text{(UnitProp)} \\
4\ \overline{1}\ \overline{2}\ \overline{3} & \| & 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee3,\ \overline{3}\vee2,\ 1\vee4 & \Longrightarrow \quad \text{(Fail)} \\
\mathit{fail} & & &
\end{array}
$$

Result: *Unsatisfiable*

Abstract DPLL: Backjumping and Learning

The basic rules can be improved by replacing the Backtrack rule with the more powerful Backjump rule and adding a Learn rule:

Backjump :

$$
M\ l^{\mathsf{d}}\ N \,\|\, F, C \implies M\ l' \,\|\, F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that :} \\ F, C \models C' \vee l' \text{ and } M \models \neg C', \\ l' \text{ is undefined in M, and} \\ \mathrm{l}' \text{ or } \neg\mathrm{l}' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N \end{cases}
$$

Learn :

$$
M \,\|\, F \implies M \,\|\, F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F \\ F \models C \end{cases}
$$

The Backjump rule is best understood by introducing the notion of *implication graph*, a directed graph associated with a state $M \parallel F$ of Abstract DPLL:

- The vertices are the *variables* in M
- There is an edge from v_1 to v_2 if v_2 was assigned a value as the result of an application of UnitProp using a clause containing v_2.

When we reach a state in which $M \models \neg C$ for some $C \in F$, we add an extra *conflict* vertex and edges from each of the variables in C to the conflict vertex.

The clause to use for backjumping (called the *conflict clause*) is obtained from the resulting graph:

- We first cut the graph along edges in such a way that it separates the conflict vertex from all of the decision vertices.
- Then, every vertex with an outgoing edge that was cut is marked.
- For each literal l in M whose variable is marked, $-l$ is added to the conflict clause.

To avoid ever having the same conflict again, we can learn the conflict clause using the $learn$ rule.

$$
\begin{array}{rcll}
\emptyset & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow \quad \text{(Decide)} \\
1^{d} & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow \quad \text{(UnitProp)} \\
1^{d}\ 2 & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow \quad \text{(Decide)} \\
1^{d}\ 2\ 3^{d} & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow \quad \text{(Decide)} \\
1^{d}\ 2\ 3^{d}\ 5^{d} & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow \quad \text{(UnitProp)} \\
1^{d}\ 2\ 3^{d}\ 5^{d}\ \overline{6} & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow \quad \text{(Learn)} \\
1^{d}\ 2\ 3^{d}\ 5^{d}\ \overline{6} & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5} & \\
 & & \Longrightarrow \quad \text{(Backjump)} & \\
1^{d}\ 2\ \overline{5} & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5} & \\
 & & \Longrightarrow \quad \text{(Decide)} & \\
1^{d}\ 2\ \overline{5}\ 3^{d} & \parallel & \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5} &
\end{array}
$$

Result: *Satisfiable*

Two final rules also have to do with learning:

- If too many clauses are learned, performance suffers. It is useful to *forget* some clauses (typically those that have not participated in an application of UnitProp for a while).
- If we are stuck, we can *restart* by throwing away M. Since we have learned clauses, this means our efforts were not entirely wasted. Randomly restarting can improve performance dramatically.

Forget :

$$M \parallel F, C \implies M \parallel F \quad \textbf{if} \left\{ F \models C \right.$$

Restart :

$$M \parallel F \implies \emptyset \parallel F$$

Decision Heuristics

The rules do not give any strategy for *how* to pick a variable when applying Decide.

In practice, this is critical for performance.

There are many heuristics, but the most successful currently use very cheap heuristics to try to prefer variables that are frequently involved in conflicts.

Boolean Constraint Propagation

The most expensive part of a SAT solver is the part that checks for and applies instances of the UnitProp rule.

A key insight that can be used to speed this up is that as long as a clause has at least two unassigned literals, it cannot participate in an application of UnitProp.

For every clause, we assign two of its unassigned literals as the *watched* literals.

Every time a literal is assigned, only those clauses in which it is watched need to be checked for a possible triggering of the UnitProp rule.

For those clauses that are inspected, if UnitProp is not triggered, a new unassigned literal is chosen to be watched.

Other Considerations

Modern SAT solvers have a number of other tricks to speed things up:

- Highly tuned code
- Optimization for cache performance
- Preprocessing and clever CNF encodings
- Automatic tuning of program parameters

Modeling for SAT

Modeling

- Define a finite set of possibilities called *states*.
- Model states using (vectors of) propositional variables.
- Use propositional formulas to describe legal and illegal states.
- Construct a propositional formula describing the desired state.

Solving

- Translate the formula into CNF.
- If the formula is satisfiable, the satisfying assignment gives the desired state.
- If the formula is not satisfiable, the desired state does not exist.

Example: Graph Coloring

Problems involving *graph coloring* are important in both theoretical and applied computer science.

Recall that a *graph* consists of a set V of vertices and a set E of edges, where each edge is an *unordered* pair of *distinct* vertices.

A *complete graph* on n vertices is a graph with $|V| = n$ such that E contains all possible pairs of vertices.

A *complete graph* on n vertices is a graph with $|V| = n$ such that E contains all possible pairs of vertices.

How many edges are in a complete graph? $\frac{n(n-1)}{2}$

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color? **Answer:** $n = 2$

What if the number of colors is *2*? **Answer:** $n = 5$

What if the number of colors is *3*? This is a job for *SAT*

Example: Graph Coloring

- *Define a finite set of possibilities called states.*

  For this problem, each possible coloring is a state. There are $3^{|E|}$ possible states.

- *Model states using (vectors of) propositional variables.*

  A simple encoding uses two propositional variables for each edge. Since there are 4 possible combinations of values of two variables, this gives us a state space of $4^{|E|}$, which is larger than we need, but keeps the encoding simple.

- *Use propositional formulas to describe legal and illegal states.*

  Since the color of each edge is modeled with 2 variables, there are 4 possible colors. We can write a set of formulas which disallow the fourth color.

  For example, if e_1 and e_2 are the variables for edge e, we simply require $\neg(e_1 \wedge e_2)$.

Example: Graph Coloring

- *Construct a propositional formula describing the desired state.*

  The desired state is one in which there are no triangles of the same color. For each triangle made up of edges e, f, g, we require:
  $\neg((e_1 \leftrightarrow f_1) \wedge (f_1 \leftrightarrow g_1) \wedge (e_2 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow g_2))$.

- *Translate the formula into an equisatisfiable CNF formula.* This can be done using the CNF conversion algorithm described earlier.

- *If the formula is satisfiable, the satisfying assignment gives the desired state.*

  An actual coloring can be constructed by looking at the values of each variable given by the satisfying assignment.

Example: Graph Coloring

- *If the formula is not satisfiable, the desired state does not exist.*

  If the formula can be shown to be unsatisfiable, this is proof that there is no coloring.

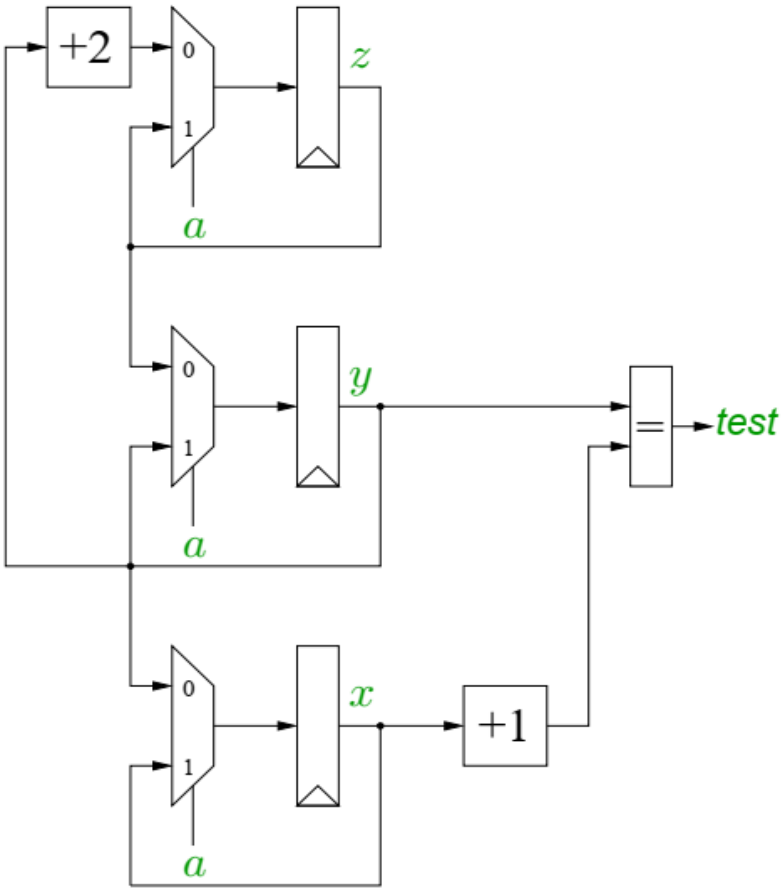
What if the number of colors is *3*?

Answer: $n = 16$

Modeling

Let us consider a circuit example.

Circuit Example



Modeling

One way to prove the property of the circuit is by induction.

The inductive step is essentially the following:

```
(y = x + 1 AND z = x + 2 AND
 x' = IF a THEN x ELSE y AND
 y' = IF a THEN y ELSE z AND
 z' = IF a THEN z ELSE y + 2) IMPLIES
 y' = x' + 1 AND z' = x' + 2
```

We can prove this formula by showing that the negation is unsatisfiable.

We can write this formula in propositional logic by using one propositional variable for each bit in the current and next states.

Assuming a bit-width of 2 for simplicity and skipping the details, we get the following formula:

$$
\begin{array}{l}
(z1 \leftrightarrow \neg x1) \wedge (z0 \leftrightarrow x0) \wedge \\
(y1 \leftrightarrow (x1 \oplus x0)) \wedge (y0 \leftrightarrow \neg x0) \wedge \\
(a \rightarrow ((xp1 \leftrightarrow x1) \wedge (xp0 \leftrightarrow x0))) \wedge \\
(\neg a \rightarrow ((xp1 \leftrightarrow y1) \wedge (xp0 \leftrightarrow y0))) \wedge \\
(a \rightarrow ((yp1 \leftrightarrow y1) \wedge (yp0 \leftrightarrow y0))) \wedge \\
(\neg a \rightarrow ((yp1 \leftrightarrow z1) \wedge (yp0 \leftrightarrow z0))) \wedge \\
(a \rightarrow ((zp1 \leftrightarrow z1) \wedge (zp0 \leftrightarrow z0))) \wedge \\
(\neg a \rightarrow ((zp1 \leftrightarrow \neg y1) \wedge (zp0 \leftrightarrow y0))) \wedge \\
(\neg(zp1 \leftrightarrow \neg xp1) \vee \neg(zp0 \leftrightarrow xp0) \vee \\
\neg(yp1 \leftrightarrow (xp1 \oplus xp0)) \vee (yp0 \leftrightarrow xp0))
\end{array}
$$

Modeling: Transition Systems

Often, we want to model a system as a *transition system*: a system with a set of states and a set of possible transitions between states.

Suppose Q is a set of states, $Q_0 \subseteq Q$ a set of initial states, and T a transition relation on states (i.e. $T \subseteq Q \times Q$).

Since Q is finite, we can find an m such that $2^m \geq |Q|$. We can then use m variables: $\vec{x} = [x_1, \ldots, x_m]$ to represent the states. These are called *state variables*.

To represent T, we need m additional variables, $\vec{y} = [y_1, \ldots, y_m]$, which we call *next-state variables*.

We can write formulas $F_{Q_0}(\vec{x})$ and $F_T(\vec{y})$ such that the solutions of $F_{Q_0}(\vec{x})$ correspond to initial states in Q_0 and the solutions of $F_T(\vec{x}, \vec{y})$ correspond to valid transitions in T.

Bounded Model Checking

Bounded Model Checking can be used to determine whether a state is reachable from the initial state in some bounded number of transitions.

To perofrm bounded model checking to a depth of n using SAT, we need n extra copies of the state variables and a set of states Q_P that we are trying to reach.

Let $\vec{x_0}, \ldots, \vec{x_n}$ be $n+1$ copies of the state variables. And let $F_{Q_P}(\vec{x})$ be a formula that is true for the states in Q_P.

Q_P is reachable in n steps iff the following formula is satisfiable:

$$F_{Q_0}(\vec{x_0}) \wedge F_T(\vec{x_0}, \vec{x_1}) \wedge \cdots \wedge F_T(\vec{x_{n-1}}, \vec{x}) \wedge F_{Q_P}(\vec{x_n}).$$