ECE650
Normal Forms and DPLL SAT Algorithm

Vijay Ganesh

Overview

- Last lecture:
  - Two simple techniques for proving satisfiability and validity in propositional logic: truth tables and semantic argument
  - Neither very useful for practical automated reasoning
- This Lecture:
  - An algorithm called DPLL for determining satisfiability
  - Many SAT solvers used today based on DPLL (more precisely, conflict-driven clause-learning)
  - However, requires converting formulas to a respresentation called normal forms
- The plan: First talk about normal forms, then discuss DPLL

Normal Forms

- A normal form of a formula F is another formula F' such that F is equivalent to F', but F' obeys certain syntactic restrictions.
- There are three kinds of normal forms that are interesting in propositional logic:
  - Negation Normal Form (NNF)
  - Disjunctive Normal Form (DNF)
  - Conjunctive Normal Form (CNF)

Negation Normal Form (NNF)

Negation Normal Form requires two syntactic restrictions:

- The only logical connectives are $\neg, \wedge, \vee$ (i.e., no $\rightarrow$, $\leftrightarrow$)
- Negations appear only in literals
- i.e., negations not allowed inside $\wedge$, $\vee$, or any other $\neg$
- i.e., negations can only appear in front of variables
- Is formula $p \vee (\neg q \wedge (r \vee \neg s))$ in NNF? Yes!
- What about $p \vee (\neg q \wedge \neg(\neg r \wedge s))$? No!
- What about $p \vee (\neg q \wedge (\neg\neg r \vee \neg s))$? No!

Conversion to NNF I

- To make sure the only logical connectives are $\neg, \wedge, \vee$, need to eliminate $\rightarrow$ and $\leftrightarrow$
- How do we express $F_1 \rightarrow F_2$ using $\vee, \wedge, \neg$?

$$\boxed{F_1 \rightarrow F_2 \;\Leftrightarrow\; \neg F_1 \vee F_2}$$

- How do we express $F_1 \leftrightarrow F_2$ using only $\neg, \wedge . \vee$?

$$\boxed{F_1 \leftrightarrow F_2 \;\Leftrightarrow\; (\neg F_1 \vee F_2) \wedge (\neg F_2 \vee F_1)}$$

Conversion to NNF II

- Also need to ensure negations appear only in literals: push negations in
- Use DeMorgan's laws to distribute $\neg$ over $\wedge$ and $\vee$:

$$\neg(F_1 \wedge F_2) \Leftrightarrow \neg F_1 \vee \neg F_2$$

$$\neg(F_1 \vee F_2) \Leftrightarrow \neg F_1 \wedge \neg F_2$$

- We also disallow double negations:

$$\neg\neg F \Leftrightarrow F$$

NNF Example

Convert $F: \neg(p \rightarrow (p \wedge q))$ to NNF

$$
\begin{aligned}
F_1: &\quad \neg(\neg p \vee (p \wedge q)) \\
F_2: &\quad \neg\neg p \wedge \neg(p \wedge q) \\
F_3: &\quad \neg\neg p \wedge (\neg p \vee \neg q) \\
F_4: &\quad p \wedge (\neg p \vee \neg q)
\end{aligned}
$$

F_4 is equivalent to F and is in NNF

Vijay Ganesh 对所有人说: 下午 4:21

For clarification, second condition for a formula to be in nnf which is all negations must be appear in the graph of the formula only in the layer above the leaf nodes, is false and the correct way if negation appears in the formula then it must appear in layer above the leaf nodes, right? But, why?

Vijay Ganesh 对所有人说: 下午 4:22

The reason is the following: what if no negation appears in the formula. For example, (x v y). This formula also is in NNF.

Disjunctive Normal Form (DNF)

- A formula in disjunctive normal form is a disjunction of conjunction of literals.

$$\bigvee_i \bigwedge_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}$$

- i.e., $\vee$ can never appear inside $\wedge$ or $\neg$
- Called disjunctive normal form because disjuncts are at the outer level
- Each inner conjunction is called a clause
- Question: If a formula is in DNF, is it also in NNF? Yes

-a in DNF and NNF
a also in DNF and NNF

But not the other way around, if a formula is in NNF, it's not necessarily in DNF.
-a ^ (-b v c) would be a counter example. But (-a ^ -b) v c is NOT a counter example.

Conversion to DNF

- To convert formula to DNF, first convert it to NNF.
- Then, distribute $\wedge$ over $\vee$:

$$(F_1 \vee F_2) \wedge F_3 \quad \Leftrightarrow \quad (F_1 \wedge F_3) \vee (F_2 \wedge F_3)$$
$$F_1 \wedge (F_2 \vee F_3) \quad \Leftrightarrow \quad (F_1 \wedge F_2) \vee (F_1 \wedge F_3)$$

If want to check if this equivalence is true, can just use a truth table

Example

Convert $F: (q_1 \vee \neg\neg q_2) \wedge (\neg r_1 \rightarrow r_2)$ into DNF

$$
\begin{array}{lll}
F_1: & (q_1 \vee \neg\neg q_2) \wedge (\neg\neg r_1 \vee r_2) & \text{remove} \rightarrow \\
F_2: & (q_1 \vee q_2) \wedge (r_1 \vee r_2) & \text{in NNF} \\
F_3: & (q_1 \wedge (r_1 \vee r_2)) \vee (q_2 \wedge (r_1 \vee r_2)) & \text{dist} \\
F_4: & (q_1 \wedge r_1) \vee (q_1 \wedge r_2) \vee (q_2 \wedge r_1) \vee (q_2 \wedge r_2) & \text{dist}
\end{array}
$$

F_4 equivalent to F and is in DNF

Conjunction of literals is satisfiable if and only if it does NOT contain a pair of opposing literals. (Such as x1 ^ -x1)

DNF and Satisfiability

If can find a conjunctive clause that does not contain opposing literals, you are done. Can immediately decide that the formula is Satisfiable.

- Claim: If formula is in DNF, trivial to determine satisfiability. How?
- Since disjunction of clauses, formula is satisfied if any clause is satisifed.
- If there is any clause that neither contains $\perp$ nor a literal and is and its negation, then the formula is satisfiable.
- Idea: To determine satisfiability, convert formula to DNF and just do a syntactic check.

DNF and Blow-up in formula size

It is possible that there are certain formulas for which you are forced to take exponential space, if you take the approach of translation to DNF and then call the SAT solver (no matter heuristic you use, assuming P != NP).

Vijay Ganesh 对所有人说: 下午 3:49
Step 1: program/ckt to a SAT formula (NNF, DNF, CNF,...)

ckt stands for circuits

Vijay Ganesh 对所有人说: 下午 3:49
Step 2: Call the SAT solver on the input formula

Vijay Ganesh 对所有人说: 下午 3:50
Ideally, we want Step1 and Step 2 to be very efficient.

Vijay Ganesh 对所有人说: 下午 3:50
Unfortunately, DNF translation (one approach to solving the translation problem, i.e., step 1) is exponential space. So, you will never get to step 2, in the worst case.

- This idea is completely impractical. Why?
- Consider formula: $(F_1 \vee F_2) \wedge (F_3 \vee F_4)$
- In DNF:

$$(F_1 \wedge F_3) \vee (F_1 \wedge F_4) \vee (F_2 \wedge F_3) \vee (F_2 \wedge F_4)$$

- Every time we distribute, formula size doubles! Every time we do the distributivity law, the size of the formula doubles
- Moral: DNF conversion causes exponential blow-up in size!
- Checking satisfiability by converting to DNF is almost as bad as truth tables!

When NNF formulas are transformed into DNF formulas, then it is possible that the output formula may be exponentially larger than the input formula.

And in the worse case, you have to scan the whole formula to determine if the formula is Satisfiable.

Worst case 1st literal x and last literal ~x - not satisfiable. Computer has to traverse the entire formula

Vijay Ganesh 对所有人说： 下午 3:51

The term P refers to polynomial time. This is the class of problems for which there exist polynomial time deterministic algorithms.

Vijay Ganesh 对所有人说： 下午 3:51

We say an algorithm is polynomial time, if the runtime of the algorithm grows as a polynomial of the input size (n).

If an algorithm is linear time, it's great, it means you just read the input and could solve the problem.

In fact, most problems are hard and could take polynomial time in the worst case.

Vijay Ganesh 对所有人说： 下午 3:53

NP stands for non-deterministic polynomial time algorithm (TM). Turing Machine

Vijay Ganesh 对所有人说： 下午 3:54

We say a problem is in NP, if a solution to an instance of the problem can be checked in polynomial time. (However, it is possible that **finding the solution** may take exponential time.)

Vijay Ganesh 对所有人说： 下午 3:55

P != NP is mathematical way of saying that there exist problems for which solving is much harder than checking solution (in the worst case).

For example: Checking if a number is prime or not is easy compared to finding let's say nth prime number.

Another Example:

Vijay Ganesh 对所有人说: 下午 4:01
factoring problem: given a composite number, find its prime factors.

Vijay Ganesh 对所有人说: 下午 4:01
N = p * q, where N is composite and p,q are prime

Fan Zhang 对所有人说: 下午 4:01
the RSA problem

Vijay Ganesh 对所有人说: 下午 4:03
Solving factoring problem is believed to be hard (and we have lot of evidence that it is hard).

Vijay Ganesh 对所有人说: 下午 4:03
But checking whether given p, q are indeed factors of the composite number N is easy.

SAT problem is the following:

Vijay Ganesh 对所有人说: 下午 3:56
Given Boolean formulas in CNF (conjunctive normal form), decide whether they have solutions.

Conjunctive Normal Form (CNF)

E.g.: (-x1 v x2 v x3) ^ (x1 v -x2 v -x3)

- A formula in conjuctive normal form is a conjunction of disjunction of literals.

$$\bigwedge_i \bigvee_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}$$

Both -a and (a) are in CNF.

- i.e., $\wedge$ not allowed inside $\vee, \neg$.

(-a ^ b) v (-b ^c)^(-a ^ b) in NNF, but not DNF and CNF.
And this could be used as a counter example of
NNF = DNF U CNF

- Called conjunctive normal form because conjucts are at the outer level
- Each inner disjunction is called a clause
- Is formula in CNF also in NNF?

Hence, a formula can be both in DNF and CNF.
E.g.: -a and (a)

If there is an AND node in the graph of the formula, then it must at the top-level (root).

The layer right below is a set of OR nodes.

If a negation exists in the formula, then they occur in the third layer right below the (potential) OR layer.

Finally, the variables are at the leaf level.

With it defined in this way, the number of layers is limited to 4.

Conversion to CNF

Interleaving layer refers to a pair of OR followed by AND or AND followed by OR.

- To convert formula to CNF, first convert it to NNF.
- Then, distribute $\vee$ over $\wedge$:

$$(F_1 \wedge F_2) \vee F_3 \quad \Leftrightarrow \quad (F_1 \vee F_3) \wedge (F_2 \vee F_3)$$
$$F_1 \vee (F_2 \wedge F_3) \quad \Leftrightarrow \quad (F_1 \vee F_2) \wedge (F_1 \vee F_3)$$

Not exactly the same as the way of thinking in arithmetic,
since (x * y) + z != (x+z) * (y+z)

(x) is NNF, CNF, DNF, same with -x

CNF Conversion Example

Convert $F: (p \leftrightarrow (q \rightarrow r))$ into CNF

$F_1:$	$(p \rightarrow (q \rightarrow r)) \wedge ((q \rightarrow r) \rightarrow p)$	remove $\leftrightarrow$
$F_2:$	$(\neg p \vee (q \rightarrow r)) \wedge (\neg(q \rightarrow r) \vee p)$	remove $\rightarrow$
$F_3:$	$(\neg p \vee (\neg q \vee r)) \wedge (\neg(\neg q \vee r) \vee p)$	remove $\rightarrow$
$F_4:$	$(\neg p \vee \neg q \vee r) \wedge ((q \wedge \neg r) \vee p)$	De Morgan
$F_5:$	$(\neg p \vee \neg q \vee r) \wedge (q \vee p) \wedge (\neg r \vee p)$	Distribute $\vee$ over $\wedge$

F_5 is equivalent to F and is in CNF

(a v (b v c)) <=> ((a v b) v c) <=> (a v b v c)

DNF vs. CNF

- Fact: Unlike DNF, it is not trivial to determine satisfiability of formula in CNF.
- News: But almost all SAT solvers first convert formula to CNF before solving!

CNF is harder to solve than DNF.

But the translation to DNF is hard, although the solving is easy.

However, for CNF, the translation is hard, and the solving is also hard.

Why CNF?

CNF has 2 advantages:
1. There exists an algorithm to translate arbitrary code/circuit/formula into CNF in polynomial time.

2. We have built SAT solvers that take CNF as input and that can run in polynomial time for many many practical applications.

3. This does not mean that we have solved the P vs. NP question. Because, SAT solvers that take CNF formulas as input can run in exponential time for certain kind of formulas that are not derived from typical practical applications.

- Interesting Question: If it is just as expensive to convert formula to CNF as to DNF, why do solvers convert to CNF although it is much easier to determine satisfiability in DNF?

Vijay Ganesh 对所有人说: 下午 2:39
1. The translation/coversion from ckt into CNF can be done in linear time.

Vijay Ganesh 对所有人说: 下午 2:40
2. CNF enables deductions that are otherwise difficult to do. There is proof system called resolution that operations only on CNF and all modern SAT solvers implement resolution.

- Two reasons:

1. Possible to convert to equisatisfiable (not equivalent) CNF formula with only linear increase in size!

Logical equivalence: means they have the same truth table.

2. CNF makes it possible to perform interesting deductions (resolution)

The algorithm mentioned in 2 is called Tseitin method (algorithm).

2 important concepts: Equisatisfiability and Logical equivalence in terms two formula \phi(x1,...,xn) and \psi(x1,...,xn). Logical equisatisfiable is weaker than logical equivalence.

We say two formulas \phi and \psi are equisatisfiable iff the following is true: \phi is satisfiable iff \psi is satisfiable.

Vijay Ganesh 对所有人说: 下午 2:41
All satisfiable formulas are equisat with each other.

Vijay Ganesh 对所有人说: 下午 2:44
All UNSAT formulas are equisatisfiable with each other.

Vijay Ganesh 对所有人说： 下午 2:34
semantic argument method

Vijay Ganesh 对所有人说： 下午 2:35
Consider a formula F

Vijay Ganesh 对所有人说： 下午 2:35
Before applying the semantic argument method, we may not know that F is valid.

Vijay Ganesh 对所有人说： 下午 2:35
Step 1: Negate F

Vijay Ganesh 对所有人说： 下午 2:35
Step 1: Assume there exists a counterexample to F.

Vijay Ganesh 对所有人说： 下午 2:36
Subsequent steps: Apply the proof rules.

Vijay Ganesh 对所有人说： 下午 2:36
If every branch of the proof ends in false, then we know that the original formula F is valid.

Vijay Ganesh 对所有人说： 下午 2:37
if even one branch of the purported proof does not end in false (and no more rules can be applied), then we know that the formula is not valid.

Equisatisfiability

If we have (x) equiSAT (x v -x) then it means that: if (x) is SAT then (x v -x) is also SAT, and if (x v -x) is SAT then (x) is SAT.
However, (x) is NOT logically equivalent to (x v -x).

Means that (x v -x) is a satisfiable formula, and x then is also a satisfiable formula.

Also, x is equisat to -x.

We perform tseitin transformation on CNF (F) and get F' and use F' for our solver and if it is sat then F is sat.

(x AND -x) equiSAT False.

- Two formulas F and F' are equisatisfiable iff:

$$\boxed{F \text{ is satisfiable if and only if } F' \text{ is satisfiable}}$$

- Any satisfiable formula (e.g., p) is equisat as $\top$

The interpretation of F and F' can be independent.

- But clearly, p is not equivalent to $\top$! Why?

The goal is to convert F into CNF in polynomial time. So convert F into an equisatisfiable CNF formula F'. So, from the SAT solver, if F' has a solution, then F has a solution. If F' doesn't have a solution, then F doesn't have any solution.

- Equisatisfiability is a much weaker notion than equivalence.

Finding the equisatisfiable formula F' takes linear time.

- But useful if all we want to do is determine satisfiability.

The concept of Equisat simply states: Take all Boolean formulas, Boolean formulas = SAT box + UNSAT box (A formula is either SAT or it's not).

And it simply states if F has a solution, then F' has a solution. Also, if F' has a solution, then F has a solution.

Any 2 satisfiable literals are equisat.

The Plan

- To determine satisfiability of F, convert formula to equisatisfiable formula F' in CNF
- Use an algorithm (DPLL) to decide satisfiability of F'
- Since F' is equisatisfiable to F, F is satifiable iff algorithm decides F' is satisfiable
- Big question: How do we convert formula to equisatisfiable formula without causing exponential blow-up in size?

Tseitin's Transformation

Tseitin's transformation converts arbitrary formula F to equisatisfiable formula F' in CNF with only a linear increase in size.

Tseitin's Transformation Example 1

Root node of the graph for F is ->
Root node of left sub-graph is V
Root node of right sub-graph is ∧

Convert $F : (p \vee q) \rightarrow (p \wedge \neg r)$ to equisatisfiable CNF formula.

1. For each subformula, introduce new variables: p_1 for F, p_2 for $p \vee q$, p_3 for $p \wedge \neg r$, and p_4 for $\neg r$. New variable refers to variables that do not exist in the given formula F.

2. Stipulate equivalences and convert them to CNF:

Observe that every definition has at most 3 variables.
Because every operator in our logic is at most binary (AND, OR, IFF,...)

$$
\begin{array}{rl}
p_1 \leftrightarrow (p_2 \rightarrow p_3) & \Rightarrow F_1 : (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_2 \vee p_1) \wedge (\neg p_3 \vee p_1) \\
p_2 \leftrightarrow (p \vee q) & \Rightarrow F_2 : (\neg p_2 \vee p \vee q) \wedge (\neg p \vee p_2) \wedge (\neg q \vee p_2) \\
p_3 \leftrightarrow (p \wedge p_4) & \Rightarrow F_3 : (\neg p_3 \vee p) \wedge (\neg p_3 \vee p_4) \wedge (\neg p \vee \neg p_4 \vee p_3) \\
p_4 \leftrightarrow \neg r & \Rightarrow F_4 : (\neg p_4 \vee \neg r) \wedge (p_4 \vee r)
\end{array}
$$

By definition, binary operators have 2 inputs and 1 outpu

Only time we don't need a variable is when we reach the end of the leaf node of graph. And you only don't need new variables at the leaf nodes because the leaf nodes are already variables.

3. The formula Each formula F1, F2, F3, F4 are definitions for the new variables p1, p2, p3, p4

$$p_1 \wedge F_1 \wedge F_2 \wedge F_3 \wedge F_4$$

F <=>_SAT (F1 ^ F2 ^ F3 ^ F4)

is equisatisfiable to F and is in CNF. The reason why p1 is needed is explained in p26

Observe that each definition of a new variable is a constant-size formula. There can never be more than 3 variables per definition.

INtroduce new variables in the graph of F, where they are used to mark or label edges, such that:

Vijay Ganesh 对所有人说: 下午 2:50
the new variable can take the value of the underlying sub-graph G when the variables in G are assigned values.

Is the number of new variables = number of operator nodes in original formula?

Yes

for -(p ^ -r) we should assign three variables? p2 for -(p^ -r) and p3 p^-r) and p4 for -r?

Yes

Vijay Ganesh 对所有人说: 下午 3:06

Tseitin method (high-level):Step1: Introduce

Vijay Ganesh 对所有人说: 下午 3:07

new variables for every edge of operators in the graph of the input formula F

Vijay Ganesh 对所有人说: 下午 3:07

(Recursively)

Vijay Ganesh 对所有人说: 下午 3:07

Step2: Traverse graph again, and translate each definition into CNF and store it somewhere in memory.

Vijay Ganesh 对所有人说: 下午 3:08

Step 3: Take the conjunction of all CNFs of the definitions

Yes

Whenever F is SAT, G may not be SAT under the same assignment. Why?
The reason is G has extra/new variables in it. These variables do not appear in F.
New variables are dependent on the original variables.

in other words, whenever F is SAT -> G is SAT but the assignment need not be the same, and similarly for the converse?

basically, for an assignment I that satisfies F, there exists another assignment I' that satisfies G.

Vijay Ganesh 对所有人说: 下午 3:15
p1 -> (p2 -> p3) ^ (p2 -> p3) -> p1

Manan Raheja 对所有人说: 下午 3:15
yes

Vijay Ganesh 对所有人说: 下午 3:15
(-p1 v (p2->p3))

Manan Raheja 对所有人说: 下午 3:15
yes

Vijay Ganesh 对所有人说: 下午 3:16
(-p1 v -p2 v p3)

Manan Raheja 对所有人说: 下午 3:16
yes

Vijay Ganesh 对所有人说: 下午 3:16
((p2 -> p3) -> p1)

Vijay Ganesh 对所有人说: 下午 3:16
(-(p2 -> p3) v p1)

Vijay Ganesh 对所有人说: 下午 3:16
(-(-p2 v p3) v p1)

Vijay Ganesh 对所有人说: 下午 3:17
((p2 ^ -p3) v p1)

(p2 v p1) ^ (-p3 v p1)

Sheena 对所有人说: 下午 3:17
Why can't we directly expand the implication in F instead?

Vijay Ganesh 对所有人说: 下午 3:18
(-p1 v -p2 v p3) ^ (p2 v p1) ^ (-p3 v p1)

For the question, if have complicated formula, in the worst case, could have exponential blow-ups. That's the purpose of the introduction of variables p1, p2, p3.

Content around 3:40 talks about this again in relation with the assignment

F <=>_SAT p1 ^ F1 ^ F2 ^ F3 ^ F4
This is what the algorithm guarantees.
Whenever F is SAT, G= (p1 ^ F1 ^ F2 ^ F3 ^ F4) is SAT.
There exists an assignment to the new variables, such that whenever F is SAT, G is SAT.
Whenever G is SAT, we have to show that F is SAT.

Whenever F is UNSAT under an assignment, then G is also UNSAT.
A->B is the same -B -> -A
So F is not true under this assignment: (p=0, q=1, r=0)
However, when under this assignment, all of F1, F2, F3, and F4 are evaluated to true. But F is false. Hence, the p1 in the final formula is NEEDED.

Whenever F is UNSAT under an assignment, then G is also UNSAT under an assignment that extends the assignment to F with the new variables (and appropriate assignment to those new variables).
We say a transformation(F) -> G is satisfiability preserving if whenever F <=>_SAT G.

Hence: F <=>_SAT G (The symbol <=>_SAT just means equisat)

Does it mean that if every subgraph is SAT, then F is SAT ?

No

Given the fact that Tseitin transformation is satisifiability-preserving, and further the input formula F(x) is in fact logically equivalent to \exists new-x G(x,new-x), it follows that whenever G is SAT, F MUST be SAT.

And that mapping is given by Tseitin's rules

If our SAT solver shows G is SAT we can conclude that F is SAT. But But the assignments for which F is SAT - How can that be found?

You can simply read off the assignment to the old variables in G and you get a satisfying assignment for F. (Just ignore the new variables)

Old variables: Original variables from F
New variables: Variables we introduced during the transformation. In the example above, it's p1, p2, p3 and p4.

Tseitin's Transformation I

Whenever we use the term F is satisfiable, we mean that there exists an assignment to the variables of F that makes F true.

We say a formula F is unsatisfiable if it is false under all assignment to the variables of F.

- Step 1: Introduce a new variable p_G for every subformula G of F (unless G is already an atom). atom: variable or constant
- For instance, if $F = G_1 \wedge G_2$, introduce two variables p_{G_1} and p_{G_2} representing G_1 and G_2 respectively.
- p_{G_1} is said to be representative of G_1 and p_{G_2} is representative of G_2.

Tseitin's Transformation II

- Step 2: Consider each subformula

$$G : G_1 \circ G_2 \quad (\circ \text{ arbitrary boolean connective})$$

- Stipulate representative of G is equivalent to representative of $G_1 \circ G_2$

$$p_G \leftrightarrow p_{G_1} \circ p_{G_2}$$

- Step 3: Convert $p_G \leftrightarrow p_{G_1} \circ p_{G_2}$ to equivalent CNF (by converting to NNF and distributing $\vee$'s over $\wedge$'s).

- Observe: Since $p_G \leftrightarrow p_{G_1} \circ p_{G_2}$ contains at most three propositional variables and exactly two connectives, size of this formula in CNF is bound by a constant.

Tseitin's Transformation II

- Given original formula F, let p_F be its representative and let S_F be the set of all subformulas of F (including F itself).
- Then, introduce the formula

$$p_F \wedge \bigwedge_{G=(G_1 \circ G_2) \in S_F} CNF(p_g \leftrightarrow p_{g_1} \circ p_{g_2})$$

- Claim: This formula is equisatisfiable to F.
- The proof is by structural induction
- Formula is also in CNF because conjunction of CNF formulas is in CNF.

Tseitin's Transformation and Size

- Using this transformation, we converted F to an equisatisfiable CNF formula F'.
- What about the size of F'?

$$p_F \wedge \bigwedge_{G=(G_1 \circ G_2) \in S_F} CNF(p_g \leftrightarrow p_{g_1} \circ p_{g_2})$$

- $|S_F|$ is bound by the number of connectives in F.
- Each formula $CNF(p_g \leftrightarrow p_{g_1} \circ p_{g_2})$ has constant size.
- Thus, trasformation causes only linear increase in formula size.
- More precisely, the size of resulting formula is bound by $30n + 2$ where n is size of original formula

Tseitin's Transformation Example 2

Convert $F : (p \wedge q) \rightarrow (p \vee \neg r)$ to equisatisfiable CNF formula.

1. For each subformula, introduce new variables: p_1 for F, p_2 for $p \wedge q$, p_3 for $p \vee \neg r$, and p_4 for $\neg r$.

2. Stipulate equivalences and convert them to CNF:

$$\begin{array}{cc} p_1 \leftrightarrow (p_2 \rightarrow p_3) & \Rightarrow F_1 : (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_2 \vee p_1) \wedge (\neg p_3 \vee p_1) \\ p_2 \leftrightarrow (p \wedge q) & \Rightarrow F_2 : (\neg p_2 \vee p) \wedge (\neg p_2 \vee q) \wedge (\neg p \vee \neg q \vee p_2) \\ p_3 \leftrightarrow (p \vee p_4) & \Rightarrow F_3 : (\neg p_3 \vee p \vee p_4) \wedge (\neg p \vee p_3) \wedge (\neg p_4 \vee p_3) \\ p_4 \leftrightarrow \neg r & \Rightarrow F_4 : (\neg p_4 \vee \neg r) \wedge (p_4 \vee r) \end{array}$$

3. The formula

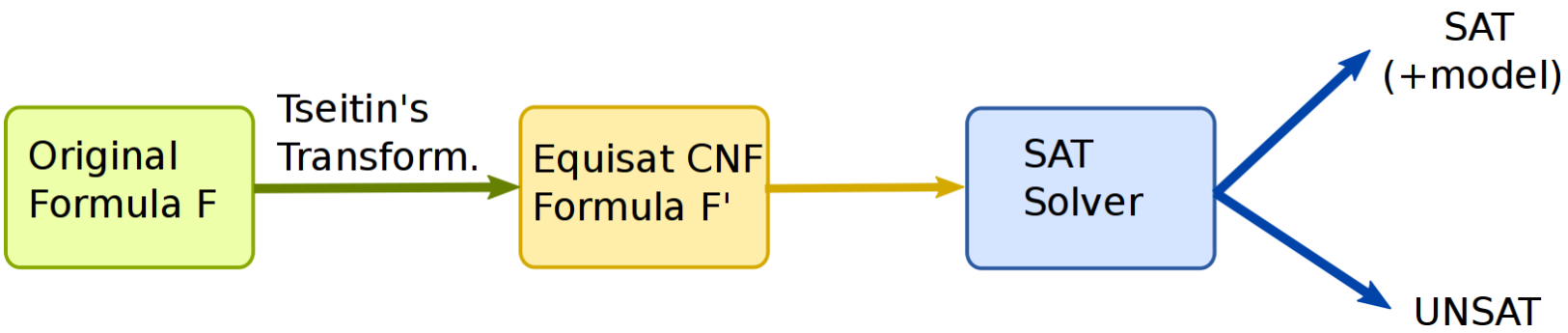
$$p_1 \wedge F_1 \wedge F_2 \wedge F_3 \wedge F_4$$

is equisatisfiable to F and is in CNF.

SAT Solvers

What is SAT solver?

A SAT solver is a computer program that when given a boolean formula \phi decides whether \phi is satisfiable in finite time (worst-case exponential time).



- Almost all SAT solvers today are based on an algorithm called DPLL (Davis-Putnam-Logemann-Loveland)

DPLL: Historical Perspective

- 1962: the original algorithm known as DP (Davis-Putnam)
  $\Rightarrow$ "simple" procedure for automated theorem proving

- Davis and Putnam hired two programmers, George Logemann and David Loveland, to implement their ideas on the IBM 704.
- Not all of their ideas worked out as planned $\Rightarrow$ refined algorithm to what is known today as DPLL

DPLL insight

DPLL SAT Solver is the foundation all industrial SAT solvers today.

- There are two distinct ways to approach the boolean satisfiability problem:
- Search
  - Find satisfying assignment in by searching through all possible assignments $\Rightarrow$ most basic incarnation: truth table!
- Deduction
  - Deduce new facts from set of known facts $\Rightarrow$ application of proof rules, semantic argument method
- DPLL combines search and deduction in a very effective way!

Deduction in DPLL

- Deductive principle underlying DPLL is propositional resolution /resolution/general resolution
- Resolution can only be applied to formulas in CNF
- SAT solvers convert formulas to CNF to be able to perform resolution

Propositional Resolution

C1: (l1 v l2...v lk v p)
C1: (-(l1 v l2 v ... v lk) -> p)

C2: ((p -> (l1' v l2' v ... v ln'))

(A -> B) (B -> C) |- (A -> C)

|- means conclude/derives

(-A v B) (-B v C) |- (-A v C)

B is called pivot literal as it appeared, in the second formula, positive in the first clause and negative in the second clause.

If p is true, then C1 is true. And in order for C2 to be true, one of the prime variables need to be true. Then, C3 must be true.

If p is false, then C2 is true. And in order for C1 to be true, one of the l variables need to be true. Then, C3 must be true as well.

- Consider two clauses in CNF:

$$C_1: \quad (l_1 \vee \ldots p \ldots \vee l_k) \qquad C_2: \quad (l_1' \vee \ldots \neg p \ldots \vee l_n')$$

In this example, p is the only pivot literal. And C3 can be formed by simply dropping p and combine the other.

- From these, we can deduce a new clause C_3, called resolvent:

(C1 ^ C2) => C3
C1: (-x1 v x2) ^ C2:(-x2 v x3)
C1: (x1->x2) ^ C2: (x2->x3)
C3: (x1->x3) is the same as C3:(-x1 v x3)

$$C_3: \quad (l_1 \vee \ldots \vee l_k \vee l_1' \vee \ldots\ldots \vee l_n')$$

C1: (-(l1 v l2 v ...v lk) -> p), C2: ((p -> (l1' v l2' v ... v ln'))
C3: (-(l1 v l2 v ... v lk) -> (l1' v l2' v ... v lk'))
C3: ((l1 v l2 v ... v lk) v (l1' v l2' v ... v lk'))
C3: (l1 v l2 v ... v lk v l1' v ... v lk')

- Correctness:
  - Suppose p is assigned $\top$: Since C_2 must be satisfied and since $\neg p$ is $\bot$, $(l_1' \vee \ldots\ldots \vee l_n')$ must be true.

Yes, you have to have a pivot literal. AND YOU CAN HAVE ONE AND ONLY ONE PIVOT LITERAL. Otherwise, it's not very useful

  - Suppose p is assigned $\bot$: Since C_1 must be satisfied and since p is $\bot$, $(l_1 \vee \ldots\ldots \vee l_k)$ must be true.

we can only resolve when we have pairs of P and -P?

  - Thus, C_3 must be true.

The only way that C2 can be true if at least one of l1', ...,ln' are true.
This immediately implies that C3 is true.
Case 2 (p is false): this case is symmetric.
You are given two clauses and C1 and C2. C1 has exactly one literal and is therefore called "unit clause".

Whenever C1 or C2 is false, we don't even care if C3 is true or false. We only care when C1 and C2 are both true. Then we have to prove C3 is true.

(C1 ^ C2) -> C3

Vijay Ganesh 对所有人说: 下午 3:09

The only way C1, C2 can be true is if p is true AND at least one of l1,...lk is true

Vijay Ganesh 对所有人说: 下午 3:10

But, then if the resolvent is C3: (l1 v...lk), then it is also true whenever C1 AND C2 are true.

C3 could be true even when C1 and C2 are false. Since (C1 ^ C2) => C3.
And if (C1 ^ C2) are true, then C3 must be true.
Contrapositively, if C3 is false, then we know that at least one of C1 or C2 is false.

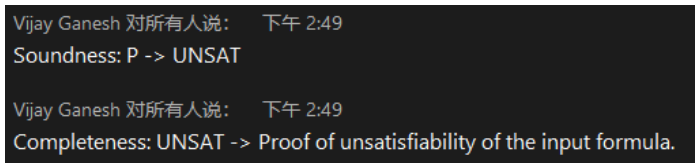
Resolution is complete. If input formula phi is UNSAT, then resolution could prove its UNSATisfiability.

If any formula generate false answer then it is UNSAT

\phi(x1,...,xn) is UNSAT. Then you can derive from \phi all possible clauses over the variables (x1,...xn), including () clause.

Unit Resolution

Completeness of resolution proof system: if the input Boolean formula is UNSAT, then the resolution proof system has a proof of unsatisfiability.



We say that a solver is terminating if it produces the correct result and halts for every input.

- DPLL uses a restricted form of resolution, known as unit resolution.
- Unit resolution is propositional resolution, but one of the clauses must be a unit clause (i.e., contains only one literal)
- $C_1 : p \qquad C_2 : \quad (l_1 \vee \ldots \neg p \ldots \vee l_n)$
- Resolvent: $(l_1 \vee \ldots \vee l_n)$

C1: (p) C2:(-p)
C3:(), empty clause

C1 ^ C2 |- C3
(p) ^ (-p) |- ()
False
() empty clause is the same thing
FALSE.

- Performing unit resolution on C_1 and C_2 is same as replacing p with true in the original clauses.
- In DPLL, all possible applications of unit resolution called Boolean Constraint Propagation (BCP).

Boolean Constraint Propagation (BCP) Example

BCP applies unit resolution repeatedly until either the formula is determined to be SAT or UNSAT or UNKNOWN.

How do we resolve (p) ^ (-p v q) ^ (r v -p v s)?
(q) ^ (r v s)

- Apply BCP to CNF formula: P must be set to true. There's no other way if you want to determine the satisfiability.
  Note: If the first clause is (-p), then p must be set to false.

$$F = \underline{(p)} \wedge (\underline{\neg p} \vee q) \wedge (r \vee \neg q \vee s)$$

Apply unit resolution to these first two clauses

- q
- $q \wedge (r \vee \neg q \vee s)$
- $(r \vee s)$
- No more unit resolution possible, so this is the result of BCP.

State of the solver is the input formula + all derived clauses
Hence, for this example, the final output is F ^ (q) ^ (r v s)

Basic DPLL

In addition to doing unit resolution, BCP also detects whether clauses are being satisfied by the partial assignment A.

For some formulas, DPLL could take exponential time.

BCP does essentially two things:
* Unit resolution and derive clauses as a consequence.
* It also detects which clauses are satisfied under A.

For this example: p v q

\phi' is exactly the same as \phi

Then it chooses a var in step 4 Let's say it chooses p, then it will call DPLL again, and assign p to true.

Then within the second call of DPLL, BCP realizes that the clause (p v q) is satisfied by the assignment A, and will return SAT in step 2.

A map from variable to value

```
bool DPLL(CNF φ, AssignMap A)
{
  1. φ' = BCP(φ,A)
  2. if(φ' = ⊤) then return SAT;
  3. else if(φ' = ⊥) then return UNSAT;
  4. p = choose_var(φ');
  5. if(DPLL(φ',A[p ↦ ⊤])) then return SAT;
  6. else return (DPLL(φ',A[p ↦ ⊥]));
}
```

We can't choose a variable that has an assignment

- Recursive procedure; input is formula in CNF
- Formula is $\top$ if no more clauses left
- Formula becomes $\bot$ if we derive $\bot$ due to unit resolution

BCP can be implemented as follows:
If a clause is satisfied under A, then drop it.
If a literal in a clause is falsified under A, then drop it.
For (p v q), A:{p->T}
BCP would drop p v q, and \phi^' is an empty formula.
If on the other hand, \phi^' contains the empty clause then that is equivalent FALSE.
f \phi' is the empty formula, then that is equivalent to TRUE.

So could I understand in this way: BCP is kind of deduction process and line 4-6 is the search process since DPLL combines search and deduction

Yes

And a follow-up question, if the BCP could not simplify the formula, then the time complexity of DPLL is the same with truth table method?

BCP will try to simplify formula with each new assignment, but yes, in the worst case, it's the same.

For: (p) (-p v q), BCP will assign A: {p->T} in the beginning.
It will drop p since it's true, and then it will detect falsified -p under A, and then formula becomes q, and will give new assignment A: {p->T, q->T}.

For: (p) (-p v q) (-q), BCP will assign A: {p->T}.
It then applies unit resolution, and we then have (q) (-q), and BCP will have A:{p->T, q->F}. Then, since -q is satisfied, the clause it dropped. But (q) is falsified and you only drop q, and leaves behind () empty clause. Which we know is false. Hence, we know this formula is unsatisfiable.

It is possible to have a family of formulas where the size of the formulas grows exponentially in the number of variables (n)

(x v y) (x v -y) (-x v y) (-y v -x) - 2 variables, 4 clauses
(x v y v z) (x v y v -z)...(-x v -y v -z) - 3 variables, 8 clauses
For these, each clause rules out an possible assignment for the formula to be true.

A DPLL is a sound and complete decision procedure, whereas BCP is NOT a complete decision procedure (because it can return UNKNOWN for certain inputs)
It checks every possible assignment in the worst case, and it makes sure that every possible assignment is covered. Also, in worst case it is replicating a truth table, and it goes through every possible assignments, hence it is complete.
Unit propagation or BCP can be described as unit resolution until saturation (by saturation we mean that all possible unit resolutions that can applied have been applied)

SHA1
Hash function, it take as input a message (512 bits, could be longer), and outputs a digest, say, 160 bits
Cryptographic hash functions: preimage resistance / inversion resistance
hash(msg) = d
It is infeasible for a polytime attacker to find a input m, given hash(m)=d. If you could do this, then the hash function is not inversion resistance.

DPLL SAT SOLVER ARCHITECTURE (1958)
THE BASIC BACKTRACKING SAT SOLVER

DPLL(Θ_{cnf}, assign) {

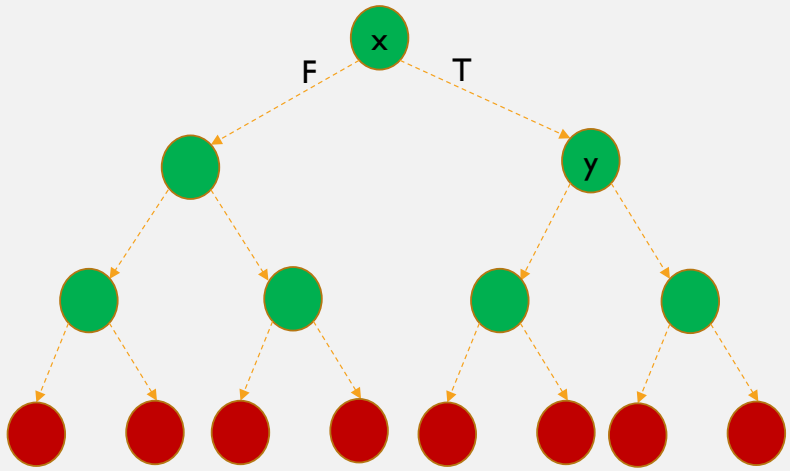
Propagate unit clauses;

if *"conflict"*: **return** FALSE;

if *"complete assign"*: **return** TRUE;

"pick decision variable x";

return
DPLL($\Theta_{cnf}\big|_{x=0}$, assign[x=0]) ||
DPLL($\Theta_{cnf}\big|_{x=1}$, assign[x=1]);
}



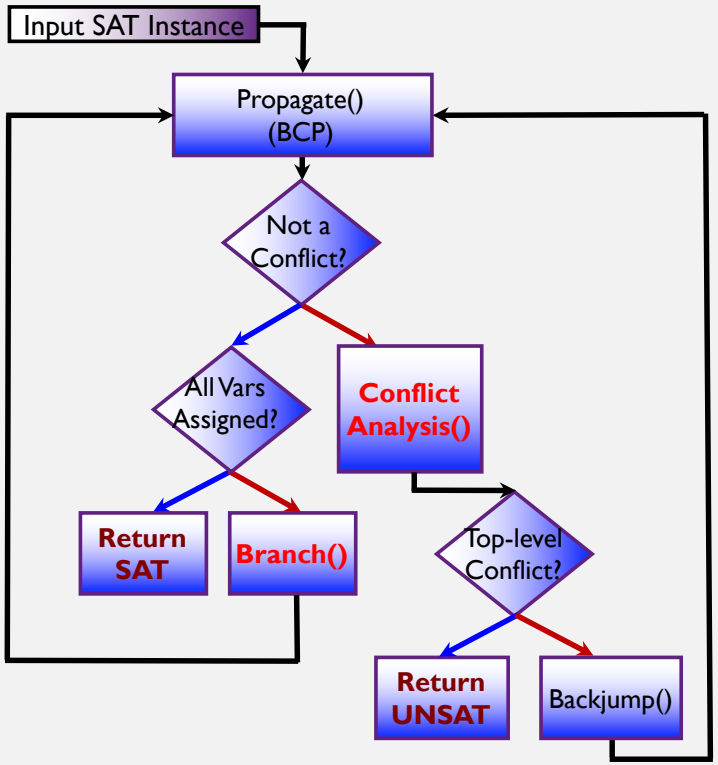
In the worst case, this tree can be exponential in the size of variables.

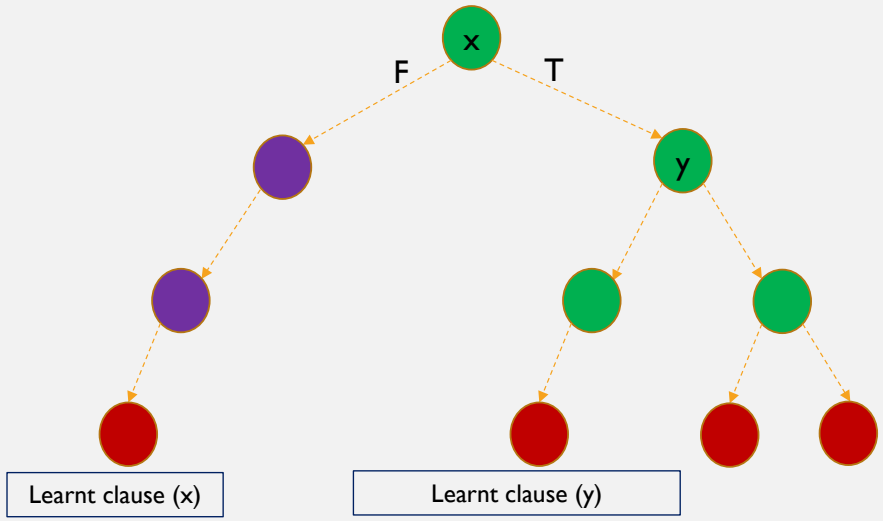
DPLL stands for Davis, Putnam, Logemann, and Loveland

Vijay Ganesh

11

MODERN CDCL SAT SOLVER ARCHITECTURE OVERVIEW





Vijay Ganesh

12

An Optimization: Pure Literal Propagation

- If variable p occurs only positively in the formula (i.e., no $\neg p$), p must be set to $\top$
- Similarly, if p occurs only negatively (i.e., only appears as $\neg p$), p must be set to $\bot$
- This is known as Pure Literal Propagation (PLP).

DPLL with Pure Literal Propagation

```
bool DPLL(CNF φ, AssignMap A)
{
  1. φ' = BCP(φ,A)
  2. φ'' = PLP(φ')
  3. if(φ'' = ⊤) then return SAT;
  4. else if(φ'' = ⊥) then return UNSAT;
  5. p = choose_var(φ'');
  6. if(DPLL(φ'',A[p ↦ ⊤])) then return SAT;
  7. else return (DPLL(φ'',A[p ↦ ⊥]));
}
```

Example

$F: (\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r) \wedge (p \vee \neg q \vee \neg r)$

- No BCP possible because no unit clause
- No PLP possible because there are no pure literals
- Choose variable q to branch on:

$$F[q \mapsto \top]: (r) \wedge (\neg r) \wedge (p \vee \neg r)$$

- Unit resolution using (r) and $(\neg r)$ deduces $\bot \Rightarrow$ backtrack

Example Cont.

$$F: \overset{\text{F}}{(\neg p} \vee q \vee r) \wedge (\overset{\text{T}}{\neg q} \vee r) \wedge (\overset{\text{T}}{\neg q} \vee \neg r) \wedge (p \vee \overset{\text{T}}{\neg q} \vee \neg r)$$

- Now, try $q = \bot$

$$F[q \mapsto \bot]: (\neg p \vee r)$$

- By PLP, set p to $\bot$ and r to $\top$
- $F[q \mapsto \bot, p \mapsto \bot, r \mapsto \top] : \top$
- Thus, F is satisfiable and the assignment $[q \mapsto \bot, p \mapsto \bot, r \mapsto \top]$ is a model (i.e., a satisfying interpretation) of F.

Summary

- Normals forms: NNF, DNF, CNF (will come up again)
- For every formula, there exists an equivalent formula in normal form
- But equivalence-preserving transformation to DNF and CNF causes exponential blowup
- However, Tseitin's transformation gives an equisatisfiable formula in CNF with only linear increase in size
- Almost all SAT solvers work on CNF formulas to perform BCP
- DPLL basis of most state-of-the-art SAT solvers

Next Lecture

- First-order logic
- Syntax, semantics, proof systems, and properties of first-order logic