Boolean Satisfiability
Lecture 5: The Resolution Proof System

Edward A. Hirsch*

January 29, 2024

Lecture 5

In this lecture we

- Define resolution proofs (refutations).
- Relate treelike resolution to DPLL.
- Discuss the size-width relation for treelike resolution.

Contents

1 Resolution as a proof system **2**
1.1 Proofs of unsatisfiability 2
1.2 Using resolvents for proving unsatisfiability 3
1.3 Daglike vs treelike proofs 3

2 Decision trees vs treelike tesolution **5**

3 Versions of resolution and DPLL **6**
3.1 DPLL vs treelike resolution 6
3.2 Versions of resolution vs proof search 7
3.3 Clause width vs proof size 7

4 Takeaway **9**

Historical notes and further reading **9**

*Ariel University, http://edwardahirsch.github.io/edwardahirsch

1 Resolution as a proof system

1.1 Proofs of unsatisfiability

It is easy to prove that a formula is satisfiable — just provide a satisfying assignment, it is very easy to check that this is a correct proof. (By the way, the same happens with every problem in **NP**.)

However, how can we prove that a formula is *un*satisfiable? Recall from Lecture 1 that **UNSAT** is **co-NP**-complete, and so we cannot hope for short (polynomial-size) proofs for every formula unless **NP** = **co-NP** (think why!). However, we can have exponential-size proofs for all formulas and shorter proofs for some formulas (hopefully, the ones we are interested in in practice, see again Lecture 1). How?

We already know that we can solve **SAT** using a DPLL approach. The resulting tree can serve as the proof of unsatisfiability: for example, to prove that

$$(x \vee \overline{y}) \wedge (y \vee z) \wedge (y \vee \overline{z}) \wedge (\overline{y} \vee u) \wedge (x \vee z) \wedge (\overline{x} \vee \overline{u})$$

is unsatisfiable we can use the following tree:

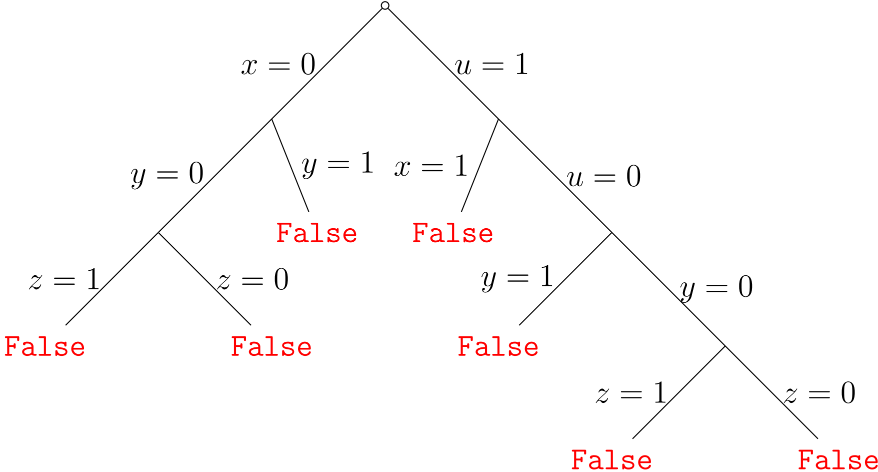


Figure 1: Example of a decision tree proving unsatisfiability

The tree given in the example corresponds to a very simple version of DPLL. We call such a tree a **decision tree**: at each node we query the value of one variable and branch. The leafs are labelled by either trivially `False` formulas (as in this example) or by trivially `True` formulas (these leafs correspond to [possibly incomplete] satisfying assignments).

Why is it called "decision tree"? Because starting at the root, you make decisions where to go (to $x = 0$ or to $x = 1$?), and at the end (in the leaf) you get your answer corresponding to these decisions: `False` or `True`. One can describe any Boolean function by a decision tree instead of a truth table.

1.2 Using resolvents for proving unsatisfiability

Recall from Lecture 2 the definition of a resolvent:

Definition 1 (Resolvent). Given $x \vee C$ and $\overline{x} \vee D$ s.t. $C \cap \overline{D} = \emptyset$, their **resolvent** (by x) is $C \vee D$. Denote the resolvent by $\mathcal{R}(C, D)$.

Recall that $\mathcal{R}(C, D)$ is implied by $C \wedge D$, that is, every assignment that satisfies $F \wedge C \wedge D$ must satisfy also $F \wedge C \wedge D \wedge \mathcal{R}(C, D)$. Thus one can add resolvents to the formula, which is sometimes depict by the following rule of logical inference:

$$\frac{x \vee C' \qquad \overline{x} \vee D'}{C' \vee D'} \quad (\text{if } C \cap \overline{D} = \emptyset)$$

This is called the **resolution rule**.

Now we can prove that $F = C_1 \wedge C_2 \wedge \ldots C_m$ is unsatisfiable by refuting it using this rule:

- Assume that there is an assignment that satisfies the clauses $C_1, \ldots, C_m$.
- Derive new clauses that are also true under this assignment:
  - Assume that we have already constructed t clauses $C_1, \ldots, C_t$.
    Take two resolvable clauses C_r, C_s and construct $C_{t+1} = \mathcal{R}(C_r, C_s)$.
- Continue deriving new clauses this way until we derive the empty clause (that is, `False`, which contradicts our assumption that we derive clauses that are true under some assignment).

At each step of this refutation we use the clauses that we derived at previous steps (including the clauses of the original formula). This refutation is called a **resolution proof**.

Note that we do not give any algorithm here: the choice of (C_r, C_s) is nondeterministic.
However, if we are given such a proof, we can easily verify it.

Remark 1. Sometimes they also use an additional derivation rule: the *weakening* rule

$$\frac{C}{C \vee D} \quad (\text{if } C \cap \overline{D} = \emptyset)$$

It allows to add literals to the previously derived clauses. It can be always eliminated at no cost (an exercise — see HW#2).

1.3 Daglike vs treelike proofs

Let us arrange our proof in a tree, where the clauses of the original formula are the leaves, and the contradiction is at the root; we are constructing this tree from the leaves to the root.

For example, for the formula

$$(x \vee \overline{y}) \wedge (y \vee z) \wedge (y \vee \overline{z}) \wedge (\overline{y} \vee u) \wedge (x \vee z) \wedge (\overline{x} \vee \overline{u})$$

our tree could be

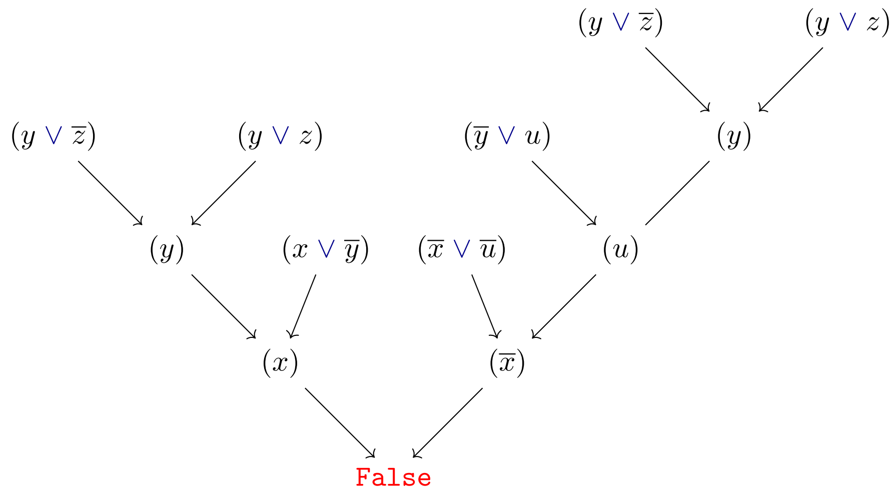


Figure 2: Example of a treelike resolution proof

Note that in this tree we derived the clause (y) twice: if we want to draw a tree, we have to derive every clause as many times as we use it. Such resolution proofs are called **treelike**.

Without this restriction, we could provide a shorter proof reusing already derived clauses as many times as we want. Such proofs are called **daglike**, as the corresponding graph is a dag and not a tree (see Figure 1.3). By default, the proofs are daglike.

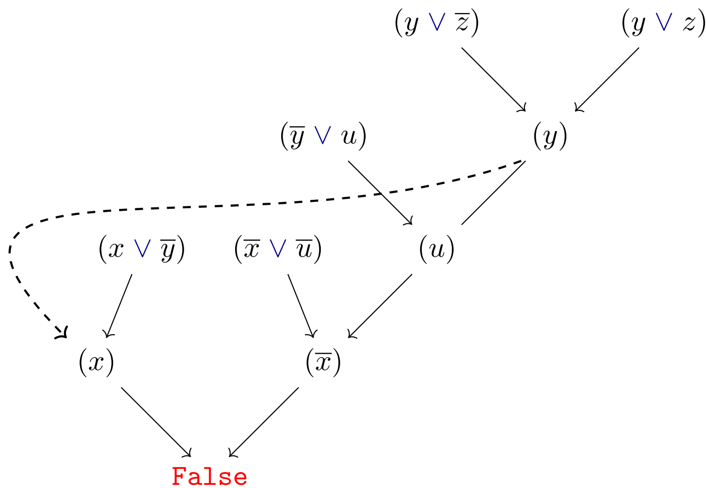


Figure 3: Example of a daglike resolution proof

Disclaimer: The fact that in these examples all derived clauses are unit clauses is a pure coincidence.

2 Decision trees vs treelike tesolution

Continuing the same example, observe that if we turn the treelike resolution picture upside down, we get a decision tree, and vice versa:

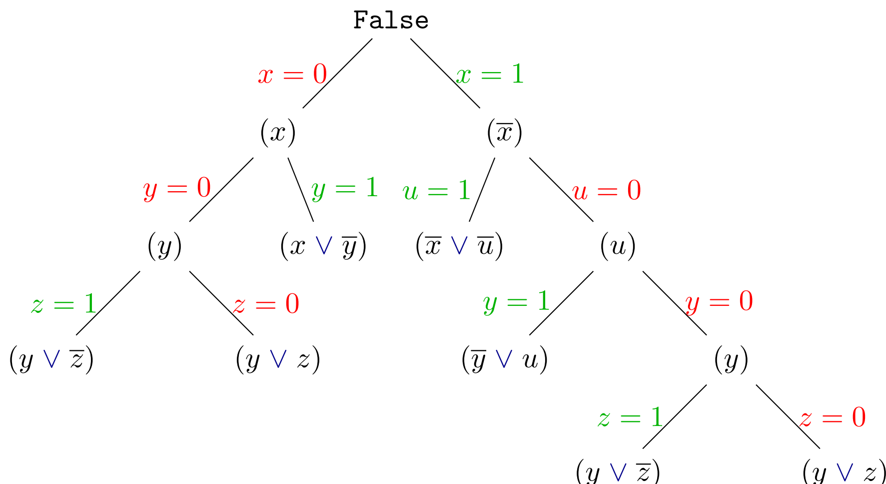


Figure 4: Treelike resolution vs decision tree: an example

Observe also that for every node, the clause written in this node is falsified by the assignment written on the path from the root to this node.

This is always the case, as we prove in the following theorem.

Theorem 1 (Treelike resolution is equivalent to decision trees)**.** *For every unsatisfiable formula, the number of nodes in the smallest treelike resolution tree refuting it equals the number of nodes in the smallest decision tree for it.*

Proof. 1. To transform a treelike resolution proof into a decision tree, turn the tree upside down and decide on the variable by which the resolvent was taken.

To prove that all the leaves will be indeed marked by `False`, proceed inductively using the following obvious lemma:

> **Lemma:** if $\mathcal{R}(x \vee C', \overline{x} \vee D')$ is falsified by A, then
> $(x \vee C')$ is falsified by $A \cup \{x = 0\}$ and $(\overline{x} \vee D')$ is falsified by $A \cup \{x = 1\}$.

As the last resolvent $\texttt{False} = \mathcal{R}(x, \overline{x})$ (for some variable x) appearing in the root is falsified by every assignment, applying the lemma downwards brings us all the way down to the leaves, and all these clauses are falsified by the corresponding assignments of the decision tree. (Look at Figure 2 and perform this induction on an example of a treelike resolution proof.)

CAUTION! We assumed here that we do not resolve by the same variable twice when deriving a clause (otherwise we would decide on the same variable twice in the same path of the decision tree). Such proofs are called *regular*, but in the treelike case it does not matter — HW#2 exercise!

2. To transform a decision tree into a treelike resolution proof proceed inductively.

Start from the leaves. The decision tree marks each leaf by `False`, and that means that one of the clauses of the input formula is falsified (otherwise how the conjunction of them could be false?). Mark the leaves by these falsified clauses and continue.

At each step take two already marked nodes that have a common unmarked parent (notice that they always exist if anything is still unmarked). Then mark this parent by the resolvent of the clauses marking these nodes (thus performing a resolution step) or just by one of these clauses (thus performing no step). To justify that we can always do this, use the following lemma:

Lemma. Consider the node corresponding to the assignment $A = [x_1 \leftarrow a_1, \ldots, x_{k-1} \leftarrow a_{k-1}]$, suppose the decision tree asks here for the decision on the variable x_k, and we have already marked the two children of this node by clauses C and D falsified by $A \cup \{x_k = 0\}$ and $A \cup \{x_k = 1\}$, respectively. Then either C and D are resolvable, and their resolvent is falsified by A, or one of them is falsified by A by itself.

Proof. Since these clauses are falsified, all their literals are false under the corresponding assignment (in particular, they are assigned by it: all the respective decisions have been already taken). If one of them does not contain the corresponding literal x_k (resp., $\overline{x_k}$), then this clause is falsified already by A.

Otherwise they contain x_k and $\overline{x_k}$, respectively, and they do not contain any other contrary pair of the literals since otherwise one of them would not be falsified. Thus they are resolvable, and it is easy to see that their resolvent is falsified by A (otherwise one of C, D would not be falsified by the respective A's extension).

falsified $\mathcal{R}(\mathbf{C},\mathbf{D}) = (C' \vee D')$

$x_k = 1$ $\quad$ $x_k = 0$

falsified $\quad$ falsified

$\mathbf{C}=(C' \vee \overline{x_k})$ $\quad$ $\mathbf{D}=(D' \vee x_k)$

□

3 Versions of resolution and DPLL

3.1 DPLL vs treelike resolution

While decision trees are a particular case of DPLL, in the other direction this is not necessarily true. However, most common DPLL tricks are still covered by the same paradigm and these DPLL trees can be transformed into treelike resolution proofs as well:

- Unit clause elimination: One can consider it as a splitting by the corresponding unit clause with one branch stopping immediately (thus we blow up the tree size at most twice).
- Pure literal elimination: Since the clauses removed by it could not be falsified in the future (unless we decide on it and take a wrong decision — it is obvious that we can always avoid deciding on a variable that is not present in the formula anymore), we do not need to form a new node of our decision tree, just proceed as usual.

- Subsumption $C \subseteq D$: Again, proceed as usual. Dropping D could potentially prevent us from stopping at some point, however, in fact it could not: if after the future decisions D is falsified, then so does C.

- See HW#2.

3.2 Versions of resolution vs proof search

How to find a proof? One could just form all the possible resolvents repeatedly, but this could take enormous time and space even if there exists a very short proof.

Let us follow the intuition: The stronger the system, the harder to find the proof. So let us restrict our search to a subset of proofs.

- Treelike resolution: as we know from the equivalence to the decision trees, to find a treelike proof, we only need to choose variables for decision — but how can we choose them so that we find a small proof? We know certain trick though, as we learned from studying upper bounds on the running time of DPLL algorithms.

- One can limit the search to **regular** proofs: the ones where we do not resolve twice on the same variable x while deriving a clause (look at the whole subtree or subgraph deriving this clause). In fact, in the case of treelike proofs it does not make the proofs larger (exercise – HW #2). In the case of daglike proofs it is known that it can do.

- **Ordered resolution**: Order the variables. We can limit our search by always resolving on the maximal literal in a clause. It is known that it can make proofs exponentially large. But can it always prove UNSAT? Yes, it can (exercise – see HW #2).

We study a different way to limit our search in the next subsection.

3.3 Clause width vs proof size

The clause **width** is the number of literals in the clause. If all clauses in the proof are narrow (they have small width), we call the proof narrow. Define the **width** of a resolution refutation as the width of the widest clause in it.

Searching for narrow proofs is much faster as there are only $O(n^c)$ clauses if the maximum width is c. Eli Ben-Sasson and Avi Wigderson show in their seminal paper *"Short Proofs are Narrow"* [BSW] that short proofs can always be made narrow.

Theorem 2. [1] *Assume we have a length-s treelike resolution refutation of a k-CNF F. Then F has a treelike resolution refutation of width at most $k + \log_2 s$.*

It follows that if there is a polynomial-size treelike proof, we can find it in time $|F|^{O(\log |F|)}$, already much better than the default 2^n.

[1] The statements formulated in these lecture notes are numerated differently from the corresponding statements formulated in the lecture.

Theorem 3. *Assume we have a length-s daglike resolution refutation of a k-CNF F. Then F has a daglike resolution refutation of width at most $k + O(\sqrt{n \log_2 s})$.*

Also not bad in certain cases.

Let us now prove Theorem 2 and prove the daglike version later in the course.

Our strategy is to split the task into refuting two simpler formulas and then reconstruct the proof of the original formula from them.

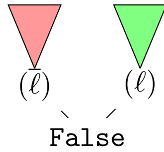
Lemma 1. *Let F be in k-CNF, and ℓ be a literal.*
Assume that $F[\ell \leftarrow 0]$ be refutable within width $w - 1$ with proof (*).
Assume that $F[\ell \leftarrow 1]$ be refutable within width w with proof (**).
Then F is refutable within width $\max(w, k)$.

Proof. Why (*) is not a refutation of F? This is because we dropped ℓ from the original clauses when we substituted ℓ by 0. Let us add ℓ back to the clauses of the derivation (*) and perform all its steps again. Note that we can still make this derivation, however, some of the clauses in it will have ℓ in addition. If the last clause is `False`, we are already done with proving the lemma conclusion. Otherwise this clause is just (ℓ).

Therefore, we have just derived (ℓ) from F within width w. Let us continue the proof: derive the clauses of $F[\ell \leftarrow 1]$. How? Just resolve (ℓ) with every clause of F that contains $\overline{\ell}$: it has exactly the same effect as the substitution $\ell \leftarrow 1$. Thus we get all the clauses of $F[\ell \leftarrow 1]$ and can use (**) to complete the refutation. □

Proof of Theorem 2. Let us prove by the induction on b and n that a proof of size at most 2^b can be transformed into a proof of width at most $k + b$.

The last step $\mathcal{R}(\ell, \overline{\ell}) =$ `False` divides the proof into the derivation of (ℓ) and the derivation of $(\overline{\ell})$:



One of these derivations is smaller than 2^{b-1}; assume w.l.o.g. this is the derivation of (ℓ). If we substitute ℓ by 0, this derivations turns into a refutation of $F[\ell \leftarrow 0]$. The induction hypothesis (both b and n are decreased by 1) then says that there is a refutation of this formula within width $k + b - 1$.

The derivation of $(\overline{\ell})$ can be turned into a refutation of $F[\ell \leftarrow 1]$ by substituting ℓ to 1. The induction hypothesis (n is decreased by 1) provides then a refutation of width at most $k + b$.

Now combine these two refutations using Lemma 1 to prove the induction hypothesis.

To apply the statement we have just proved, assume now that we have a length-s treelike resolution refutation of a k-CNF F. Take $b = \log_2 s$. The conclusion of the theorem follows. □

4 Takeaway

- We introduced resolution proofs of unsatisfiability.
- Short proofs of k-CNFs can be made narrow (and thus we can search for them relatively efficiently).
- We related treelike resolution to DPLL algorithms: it appears that most DPLL solvers are bounded to treelike resolution refutations only!

In the next lecture we will discuss CDCL, which is an extension of DPLL that overcomes the latter barrier: it derives new resolvents in the course of constructing a branching tree!

Historical notes and further reading

The resolution proof system has been defined by Martin Davis and Hilary Putnam; it was further generalized to first-order theorem proving by John Alan Robinson. All this took place in the 1960s. In fact, a similar approach was developed by Archie Blake in 1937 in his PhD thesis.

Further historical notes can be found, for example, in the 7th chapter of the 2nd edition of *Handbook of Satisfiability* (2021): "Proof Complexity and SAT Solving" by Samuel R. Buss and Jakob Nordström. However, reading any external sources is not needed for understanding these lecture notes.

References

[BSW] Eli Ben-Sasson, Avi Wigderson:
Short proofs are narrow — resolution made simple.
J. ACM 48(2): 149-169 (2001)