CSC2512
Algorithms for Solving Propositional Theories

CSC2512: Satisfiability

Proof Systems.

- A **proof system** for a language **L** is a polynomial time algorithm **PC** s.t.
  - For all inputs **F**
    F $\in$ **L iff** there exists a string **P** s.t. **PC accepts input (F,P)**

- EXAMPLE
- **L** is the set of **unsatisfiable** CNF formulas. **F** is a sample CNF, and we want to test if **F** $\in$ L, i.e., if F is unsatifiable.
- **P** is a proof that **F** is UNSAT, this proof is valid if there is a proof-checking algorithm (verifier) **PC** that runs in time polynomial in the size of **P** and **F**
- The string **P** is a proof, e.g., a resolution refutation. But other proof systems exist that verify other type of proofs.

CSC2512: Satisfiability

Proof Systems.

- The **complexity** of a **proof system, PC** for a language **L** is a function

$$f(n) = \max_{F \in L, |F|=n} \min_{P : s.t. PC\ accepts(F,P)} |P|$$

- The smallest proof of any F that is accepted by the proof system. f(n) is how the maximum smallest proof grows as the length of F grows.

CSC2512: Satisfiability

Proof Systems.

- Given two proof systems PC_1 and PC_2 we say that PC_1 **p-simulates** PC_2 if there is a polynomially computable function f such that for any proof P_2 of PC_2 (i.e., a proof accepted by PC_2) $f(P_2)$ is a proof of PC_1.

- In other words any proof of PC_2 can be converted to a proof of PC_1 with at most a polynomial increase in size (if the size increased non-polynomially, f could not be computed in polynomial time).

CSC2512: Satisfiability

Resolution Proof system

- Resolution is a proof system. Given an unsatisfiable CNF **F** a proof **P** of **F** is a sequence of clauses as defined before.
- The proof system (or checker) can check that every step of **P** is a valid step that is:
  - Each clause in the sequence **P** is either a clause of **F** or is the result of a legal resolution step involving two previous clauses.
  - Clearly this check can be done in time polynomial in the length of **P**.
- However, the complexity of resolution is $2^{O(n)}$. That is, formulas exist of length n that require exponentially long proofs **P**.
  - The check is still polynomial in the length of **P** (but takes exponential time since **P** has exponential length)

CSC2512: Satisfiability

Resolution "Refinements"

- A number of special cases of resolution have been defined and studied empirically and theoretically.

- These special cases are called refinements, although they are actually restrictions of the general case not improvements.

- Each refinement forms a new proof system:

  For a refinement the proof checker will accept only resolution proofs of a certain structure.

CSC2512: Satisfiability

Resolution DAGS

Represent Resolution proofs as DAGs:

1. Arcs go from two clauses to the resolvant clause.
2. There is only one sink node which is the empty clause ()
3. Each clause of the input formula F has in-degree 0 (these are source nodes)
4. Every other clause has in-degree 2 (the two clauses that produced it via a resolution step
5. The arc pointing (A,B) from the two clauses (A,x) and (B,-x) are **labeled** with the literals x and –x. (Represents the literal that was removed during the resolution step).

CSC2512: Satisfiability

Resolution "Refinements"

1. **Negative Resolution**
   A resolution step R[(A,x), (-x, B)] = (A,B) is **negative** whenever B contains only negative literals (negated variables). Negative Resolution requires that all resolution steps be negative.
2. **Semantic Resolution**
   Given a truth assignment π to the variables of **F** a π–refutation of **F** is a resolution refutation such that when two clauses are resolved at least one of them must be **falsified** by π. A refutation of **F** is called **semantic** if it is a π–refutation for some truth assignment .
   – Negative resolutions are semantic for the truth assignment π that assigns every variable TRUE.

CSC2512: Satisfiability

Resolution "Refinements"

3. **Linear Resolution**
   Each refutation must have a linear underlying DAG:
   - The proof consists of a sequence of clauses $c_1, c_2, \ldots, c_m = ()$, such that either ci is a clause of F or ci is derived from ci-1 and cj for some j < i-1. That is, each resolved clause must use the previous clause in the sequence.

4. **Regular Resolution**
   In the refutation DAG each path from the sink empty clause node to a clause of F (source nodes) has the property that no variable appears more than once as an arc-label.

CSC2512: Satisfiability

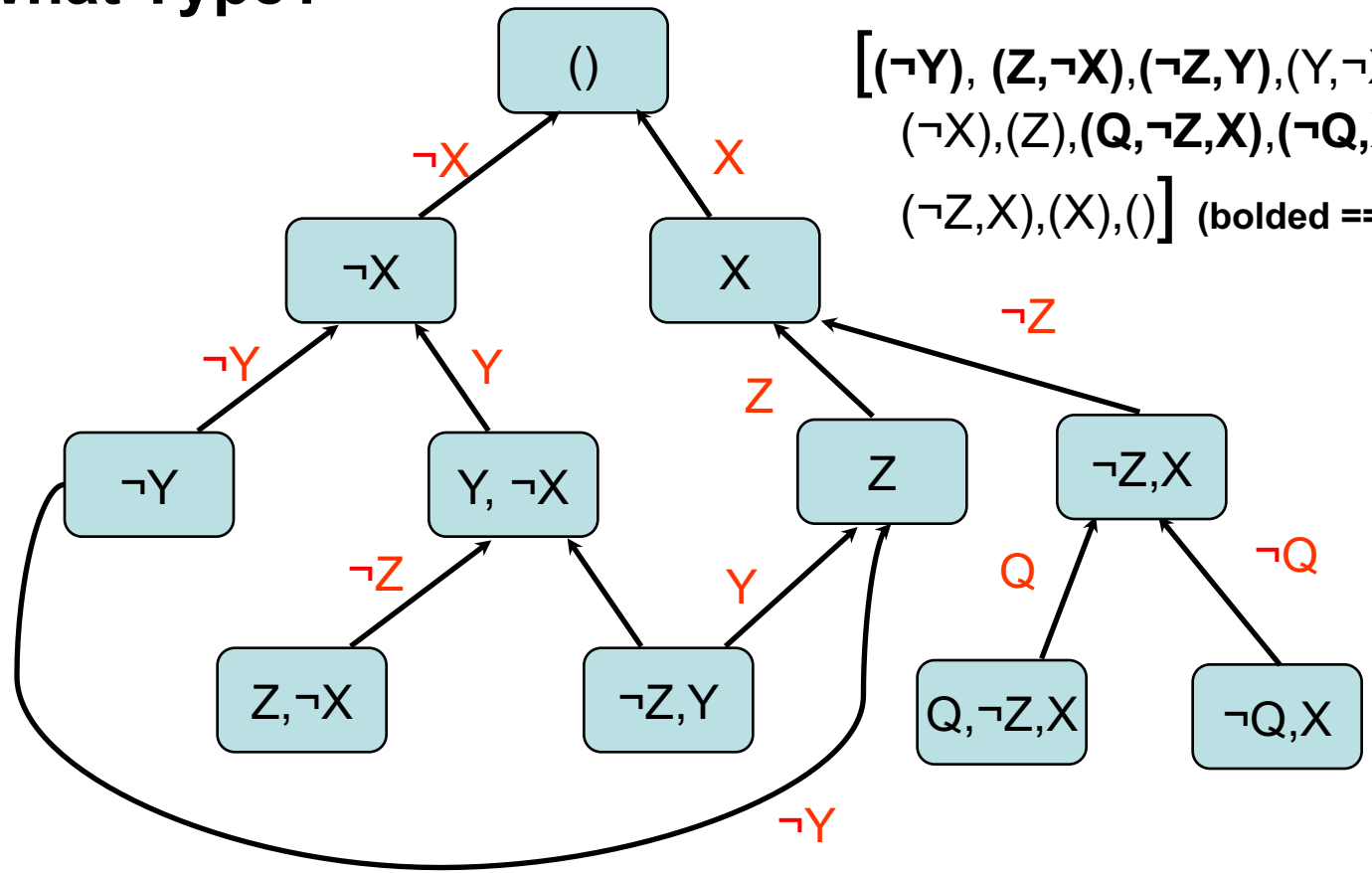
Resolution "Refinements"

5. **Ordered Resolution**
   In the DAG of each refutation the sequence of variables labeling each path from the source node to a sink node respects some total ordering of the variables.
6. **Tree-Like Resolution**
   In the DAG of each refutation is a tree once we remove the input clauses.

Each of these refinements of resolution is sound and complete. Soundness is straightforward (each proof is still an resolution proof). Completeness is typically done by showing that any resolution proof can be converted to the refinement form.

CSC2512: Satisfiability

What Type?



[**(¬Y)**, **(Z,¬X)**,**(¬Z,Y)**,(Y,¬X),
(¬X),(Z),**(Q,¬Z,X)**,**(¬Q,X)**,
(¬Z,X),(X),()] **(bolded == input clause)**

Tree-like, not ordered, regular, not linear (consider clause (z)), semantic?

CSC2512: Satisfiability

Known P-Simulation Results

	Neg	Sem	Lin	Order	Reg	Tree
Neg	Yes	No	No	No	No	Yes
Sem	Yes	Yes	No	No	No	Yes
Lin	Yes	Yes	Yes	Yes	Yes	Yes
Order	No	No	No	Yes	No	No
Reg	No	No	No	Yes	Yes	Yes
Tree	No	No	No	No	No	Yes

Cell (i,j) = does refinement of row i p-simulate refinement of column j
No means further that Refinement i requires on some formulas an exponentially long proof while Refinement j has polynomial sized proofs for the formula.

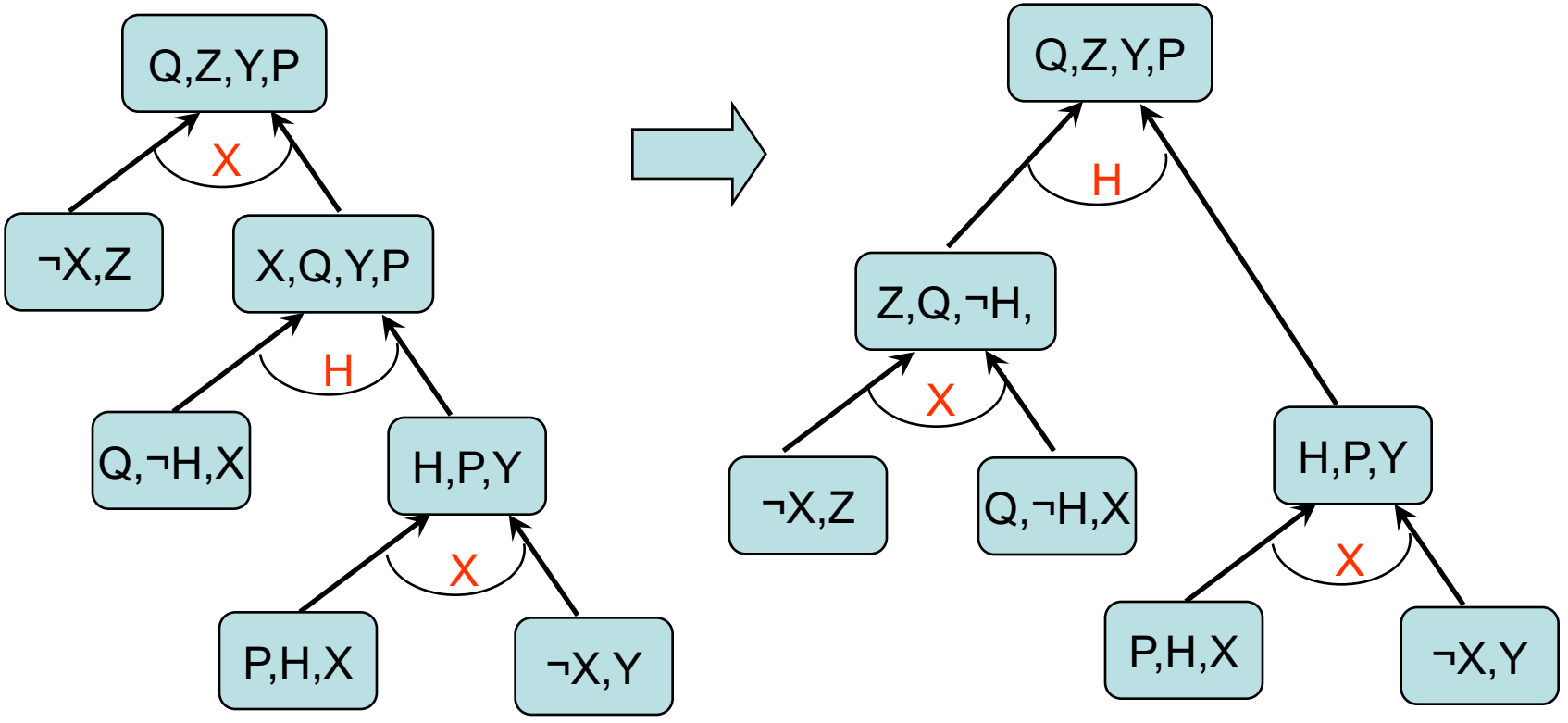
Source: "The Complexity of Resolution Refinements" by Buresh-Oppenheim and Pitassi

CSC2512: Satisfiability

Known P-Simulation Results

- Some key results:
- **Regular** is a generalization of Tree and Ordered:
  - Both Tree and Ordered proofs are regular proofs.
- **Tree** and **ordered** are very weak. They both require exponential sized proofs for formulas that other systems can prove with polynomial sized proofs.
- **Regular** also not that powerful

CSC2512: Satisfiability

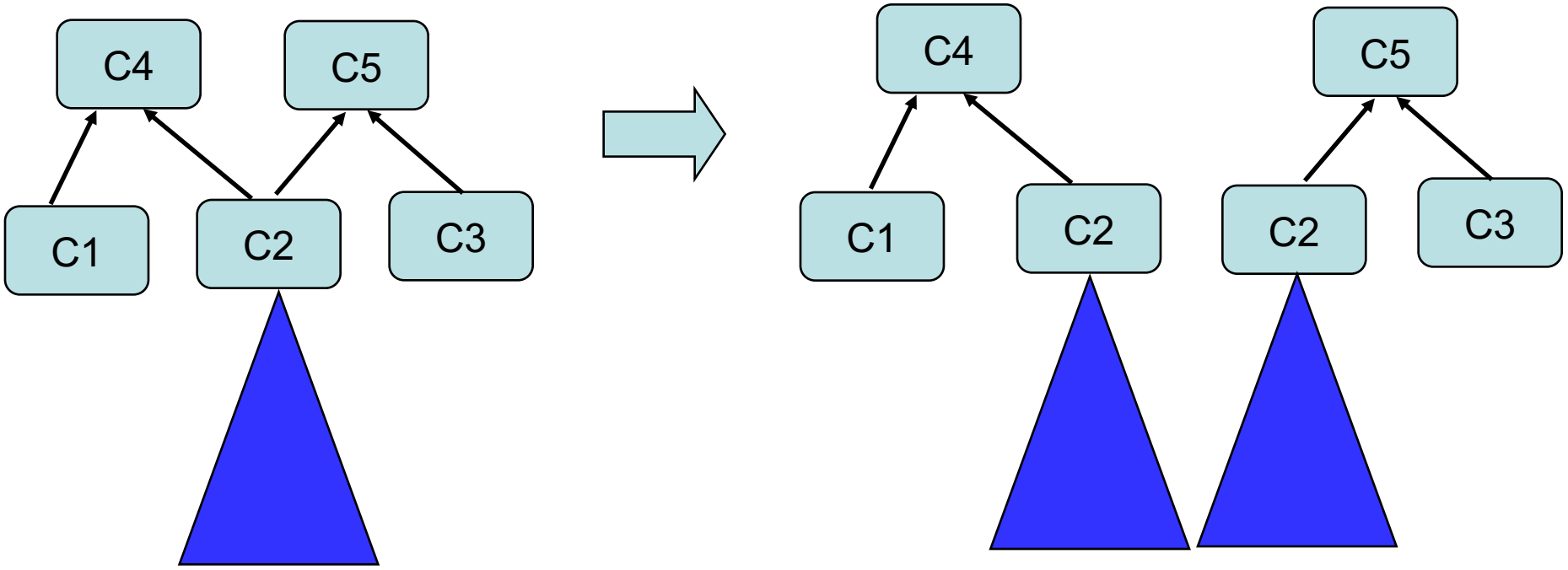


Not Regular

Regular

CSC2512: Satisfiability

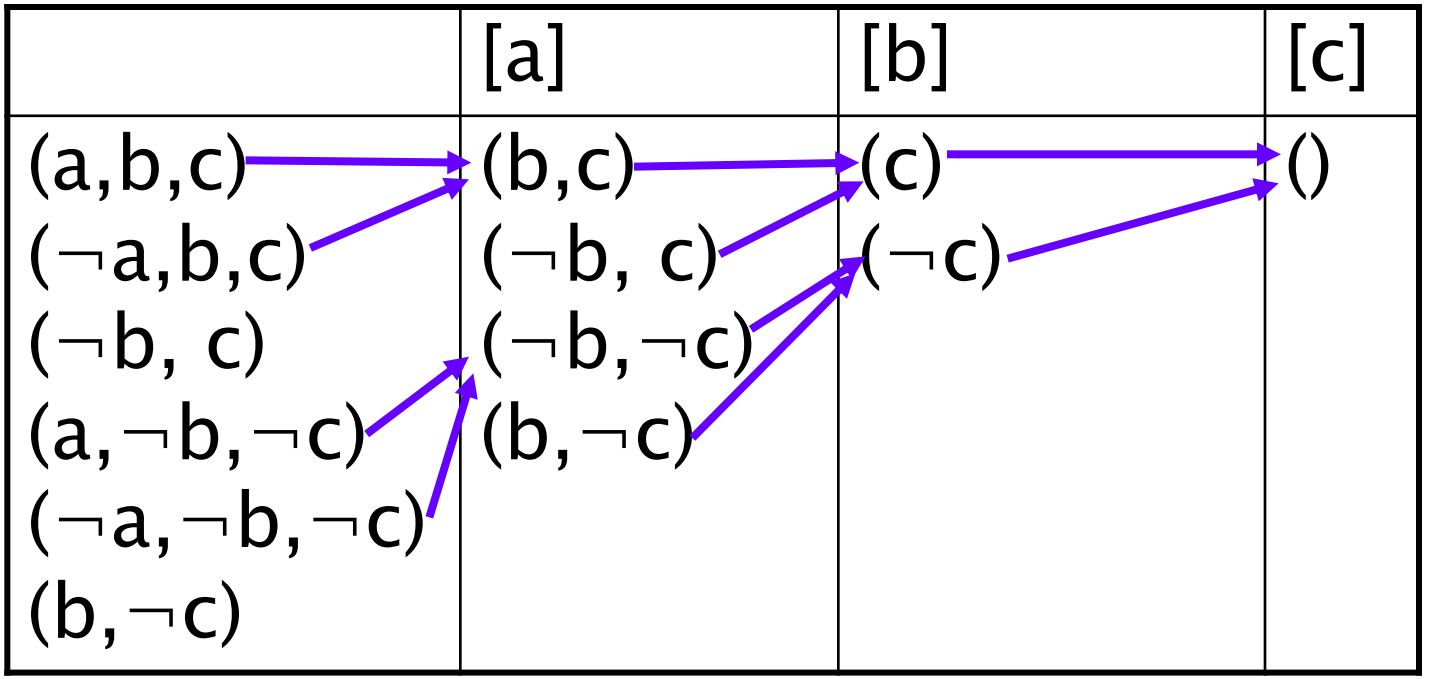
Tree Resolution



CSC2512: Satisfiability

DP produces ordered resolution proofs.

- **E**very DP run that yields the empty clause contains an ordered proof.



Potentially many redundant clauses are generated, but an ordered resolution is contained in these clauses.

CSC2512: Satisfiability

DP is not very effective at determining SAT.

1. Let **F** be a formula that requires an exponentially sized ordered refutation.

   What will be the run time of DP on **F**?

2. Also DP has high space complexity—the updated sets of clauses C become exponentially large.

Davis, Martin; Putnam, Hillary (1960). "A Computing Procedure for Quantification Theory". *Journal of the ACM* **7** (3): 201–215. doi:10.1145/321033.321034

CSC2512: Satisfiability

DPLL is also not very effective at determining SAT.

1. Let **F** be a formula that requires an exponentially sized tree refutation.

   What will be the run time of DPLL on **F**?

CSC2512: Satisfiability

DP generates ordered resolution proofs
DPLL generates tree resolution proofs

1. Any ordered resolution proof can be generated by some run of DP: just follow the same order of the variables
2. Any tree resolution proof can be generated by some run of DPLL: just branch on the variables in the same order as a pre-order traversal of the DAG (starting at the root) and mark each leaf with the input clause contained in the tree resolution.

CSC2512: Satisfiability

Modern SAT solvers

1. Based on DPLL
2. More efficient implementation methods.
3. Additional inference allowing them to move beyond tree resolution.

CSC2512: Satisfiability

DPLL(π, F)

If **F** is empty
  return (SAT,π) (π is a satisfying assignment)

If **F** contains an empty clause
  return UNSAT

else choose a variable **v** in **F** **//Want to pick variables**
//In unit clauses. How
//Implemented?

F' = F|$_v$ //How implemented? Copy , Modify/restore

(SAT?,π') = **DPLL(π + v, F')**

if SAT? == SAT return (SAT,π')

F' = F|$_{-v}$

return DPLL(π + -v, F')

CSC2512: Satisfiability

Unit Preference vs. Unit Propagation

E.g. (a, b) (-b, a, d) (-d, -b, e) (-d, -e, c, g) (-a)

DPLL would choose a:
a = True yields an empty clause
a = False yields (b) (-b, d) (-d, -b, e) (-d, -e, c, g)

Now have to choose b: b = False yields an empty clause
b = True yields (d) (-d, e) (-d, -e, c, g)

Now have to choose d: d = False yields an empty clause
d = True yields (e) (-e, c, g)

CSC2512: Satisfiability

Unit Preference vs. Unit Propagation

One Unit clause yields new unit clauses via reduction. DPLL would choose a sequence of the variables exploring the path that makes all units (and generated units) true (the other side always yields a false clause). Each new unit requires a new recursion of the algorithm.

CSC2512: Satisfiability

On large industrial problems this chaining of units might yield hundreds of new units. Need efficient way of doing it.

Don't implement as a sequence of choices of unit variables. Instead choose a literal **l** and then

1. **forced_lits = {l}**
2. While there exists a literal appearing in a unit clauses (**l'**) an not in **forced_lits**:
   1. **forced_lits = forced_lits U {l'}**
   2. set **l'** to **true** (i.e., satisfy the unit clause)

Each literal set to **true** can generate more more unit clauses. So repeat until no more units.

This process is called **Unit Propagation.**

CSC2512: Satisfiability

DPLL(π, F)

If **F** is empty
  return (SAT,π) (π is a satisfying assignment)

If **F** contains an empty clause
  return UNSAT

else choose a variable **v** in **F**

forced_lits = UP(F,v)

F' = $F|_{forced\_lits}$

(SAT?,π') = **DPLL(π + forced_lits, F')**

if SAT? == SAT return (SAT,π')

forced_lits = UP(F,-v)

F' = $F|_{forced\_lits}$

return DPLL(π + forced_lits, F')

CSC2512: Satisfiability

Recursion, how do we compute $F|_l$ and then restore F so that we can compute $F|_{-l}$?

Make a copy of F then reduce?

Large instances can require many Mbytes

Make changes then restore?

Unit propagation can force hundreds of variables forcing extensive changes.

CSC2512: Satisfiability

Modern technique:

Why do we need $F|_l$?

Only two reasons in the algorithm

Find units, and perform unit propagation.

Find empty clauses—but empty clauses must first become unit.

Can do these two things without ever explicitly computing $F|_l$?