CSC2512
Algorithms for Solving Propositional Theories

CSC2512: Propositional Reasoning Lecture 1

Instructor: Fahiem Bacchus

- Office D.L. Pratt, Room 398B
- Office Hours: Friday 3pm to 4pm
- fbacchus@cs.toronto.edu (please put CSC2512 in the subject)
- www.cs.toronto.edu/~fbacchus

Meetings:

- Tuesdays 2:00pm to 4:00pm. AB 114

- Course Website: www.cs.toronto.edu/~fbacchus/csc2512/

CSC2512: Propositional Reasoning

Evaluation:

- 25% Assignments (two).
- 25% Class participation. Paper summaries, discussion, etc.
- 50% Project
  - You can work individually or in teams of 2-3. The project will involve applying ideas from the course to solve some problem from your own area of research, or some further exploration of a topic in the course.

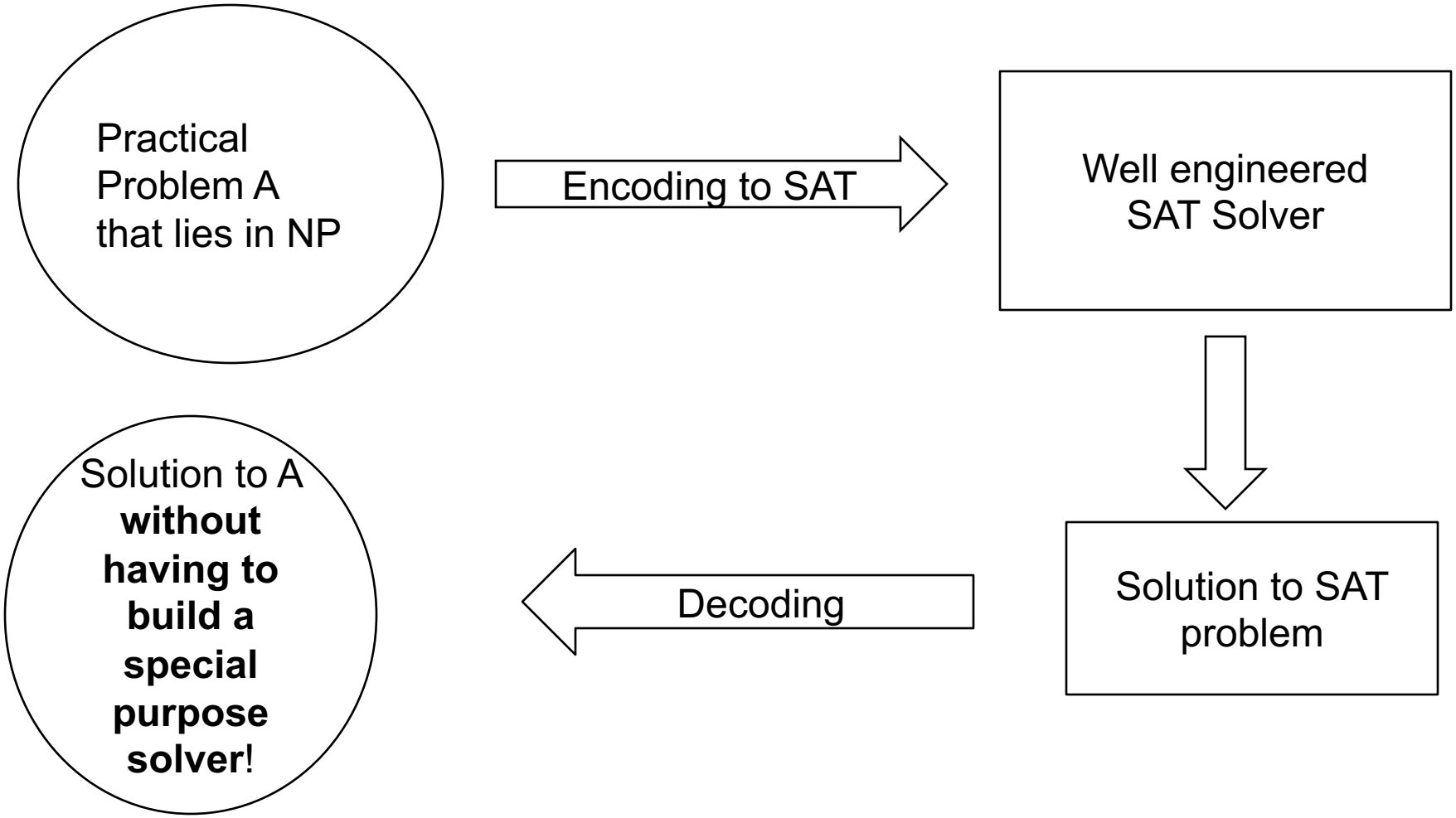
CSC2512: Propositional Reasoning

Overview:

- We will study various problems in propositional reasoning.
- These problems are complete for various complexity classes.
- The aim is to develop effective algorithms for solving these problems.
  - By effective we mean effective on a range of useful practical problems.

- Why?
  - If we can build effective solvers for a complete problem then any other problem within that problem class then then be solved by the “simple” device of encoding.

CSC2512: Propositional Reasoning



CSC2512: Propositional Reasoning

- Can this approach be successful?
- Evidence with modern SAT solvers indicate that in fact this approach can sometimes offer **significant performance improvements** over developing problem specific software.

- In this course we will look at various complete problems and solvers for these problems.
- The most natural complete problems are **propositional reasoning problems**. (Logic plays a fundamental role of logic in computer science).
- Propositional reasoning problems include the NP complete SAT problem and other problems that go beyond NP

CSC2512: Propositional Reasoning

Prerequisites:

- The course should be assessable to any CS or ECE graduate student.
  - Familiarity with programming, data structures and algorithms
  - Basics of propositional logic.

CSC2512: Propositional Reasoning

Types of Complete Problems we will examine (we will cover as many of these as we have time to):

- **Satisfiability**.
  - SAT, which is testing satisfiability over propositional theories
  - Many problems in scheduling, test generation, verification, etc. can be naturally encoded in SAT
- **SMT**
  - SAT problems with additional formulas over decidable first-order theories.
- **MAXSAT**
  - Optimization version of SAT. Compute for FP^{NP} the class of functions computable by a polynomial number of calls to an NP oracle. Also hard to approximate---APX complete (so does not admit a polytime approximation scheme).
  - Many important practical optimization problems can be encoded in MAXSAT.

CSC2512: Propositional Reasoning

Types of Complete Problems we will examine:

- **QBF (Quantified Boolean Formulas)**
  - PSPACE complete.
  - Offers compact encodings for many problems whose SAT encoding would be too large.
- **#SAT**
  - Count the number of satisfying models.
  - Probabilistic reasoning over discrete probability spaces can be reduced to #SAT.

CSC2512: Propositional Reasoning

Style of Course

- I will present lectures to provide needed background and then we will read some research papers to cover more recent topics.

- We will discuss these papers together. Your participation in these discussions will constitute the course participation component of your mark.

CSC2512: Propositional Logic

Syntax of propositional Logic:

- Set of Boolean variables (True/False propositional variables)
- Set of logical connectives, typically including
  - Negation $\neg$
  - Conjunction (AND) $\wedge$
  - Disjunction (OR) $\vee$
- Sometimes,
  - Exclusive-or $\oplus$
  - Implication $\rightarrow$
  - Not And (NAND), Not Or (NOR)
  - Etc.

CSC2512: Propositional Logic

Syntax of propositional Logic:

- p when p is a Boolean variable
- ¬f when f is a formula
- $f_1 \wedge f_2$ when f_1 and f_2 are formulas
- $f_1 \vee f_2$ when f_1 and f_2 are formulas

CSC2512: Propositional Logic

Semantics of propositional logic.

Truth Assignments (Models)

1. Truth assignment π: map each propositional variable to True/False (0,1)

$$p_i \rightarrow \{0, 1\}$$

2. Extend to formulas:

$\pi(\neg f) = 1$ if $\pi(f) = 0$
$= 0$ if $\pi(f) = 1$

$\pi(f_1 \wedge f_2) = 1$ if both $\pi(f_1) = 1$ and $\pi(f_2) = 1$
$= 0$ otherwise

$\pi(f_1 \vee f_2) = 1$ if $\pi(f_1) = 1$ or $\pi(f_2) = 1$
$= 0$ otherwise

CSC2512: Propositional Logic

Satisfiability: Given a formula F is there a truth assignment π such that $\pi(\mathbf{F}) = 1$?

With n propositional variables we have 2^n possible truth assignments. Computationally hard in general to find a satisfying assignment from this large number of truth assignments.

Abbreviations: $f_1 \rightarrow f_2 = \neg f_1 \vee f_2$

$f_1 \equiv f_2 = (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$

$f_1 \oplus f_2 = (f_1 \wedge \neg f_2) \wedge (\neg f_1 \wedge f_2)$

$f_1 \oplus f_2 \oplus f_3 \oplus f_4 \oplus f_5 \equiv ((((f_1 \oplus f_2) \oplus f_3) \oplus f_4) \oplus f_5)$

ODD number of the f_i are true.

CSC2512: Propositional Logic

Propositional Formula F

F is satisfiable if there exists a truth assignment π such that $\pi(F) = 1$. Unsatisfiable otherwise.
Sometimes written $\pi \models$ **F**

F is valid (a tautology) if for all truth assignments π we have $\pi \models$ **F**

F is unsatisfiable if and only if $\neg$F is **valid**.

f is a logical consequence of F if for all truth assignment π such that $\pi \models$ **F we have that** $\pi \models$ **f**

CSC2512: CNF

Modern SAT solvers work with **F** expressed in Conjunctive Normal Form (CNF)

CNF: a conjunction of clauses, each of which is a disjunction of literals, each of which is either a propositional variable or the negation of a propositional variable.

$(p_1 \vee \neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_5) \wedge (p_2 \vee \neg p_6) \wedge (p_4 \vee p_5) \wedge (\neg p_3)$

We typically write this in abbreviated form:

$(p_1 , \neg p_2, p_3)(p_2, \neg p_5)(p_2, \neg p_6)(p_4, p_5)(\neg p_3)$

CSC2512: CNF

1. A clause with clashing literals in it is true under any truth assignment. Such clauses are called **tautological.** Such clauses can be removed from the CNF

2. Duplicate literals are irrelevant (the disjunction of two 0 or two 1 is still 0 or 1).

3. We say that a clause c is **subsumed by** another clause c' if c' is a **subset** of c when viewed as being a set of literals.
   - All clauses must be satisfied
   - Any truth assignment that satisfies c' must also satisfy c
   - c can be removed from the CNF without changing the set of models that make it true. **(a preprocessing reduction)**

CSC2512: CNF

Notes:

4. Each clause serves to eliminate some set of truth assignments (i.e., these truth assignments cannot be models of the CNF.
   - E.g., (a, b, $\neg$c) eliminates all truth assignments π such that $\pi(a) = 0$, $\pi(b) = 0$, and $\pi(c) = 1$
   - Shorter clauses eliminate more truth assignments
5. No truth assignment satisfies the empty clause '()'
   - must satisfy at least one of the literals in the clause and there are none.
6. Validity is easy to detect
   Only the CNF containing no clauses is valid.
7. Satisfiability is hard to detect (as hard as an arbitrary propositional formula).

CSC2512: DNF

Disjunctive Normal Form, the dual of CNF.

DNF: a disjunction of terms, each of which is a conjunction of literals, each of which is either a propositional variable or the negation of a propositional variable.

$(p_1 \wedge \neg p_2 \wedge p_3) \vee (p_2 \wedge \neg p_5) \vee (p_2 \wedge \neg p_6) \vee (p_4 \wedge p_5) \vee (\neg p_3)$

An often used abbreviated form:

$[p_1 , \neg p_2, p_3] [p_2, \neg p_5] [p_2, \neg p_6] [p_4, p_5] [\neg p_3]$

CSC2512: DNF

1. A term with clashing literals in it is false under any truth assignment. Such terms are **unsatisfiable** and can be removed from the DNF
2. Duplicate literals are irrelevant (the conjunction of two 0s or two 1s is still 0 or 1).
3. A term t is **subsumed by** another term t' if t' is a **subset** of t when viewed as being a set of literals. t' expressed a weaker condition and thus in a disjunction we don't need the stronger condition t.
   - If we satisfy t we also satisfy t'
   - Only one of t or t' need be satisfied when satisfying the DNF
   - So we can remove t: any truth assignment satisfying t also satisfies t' so we loose nothing by only requiring t' to be satisfied.

CSC2512: DNF

4. Each term serves to include some set of truth assignments (i.e., these truth assignments must be models of the DNF.
   - E.g., [a, b, $\neg$c] includes all truth assignments π such that π(a) = 1, π(b) =1, and π(c) = 0
5. By convention an empty term is valid (satisfied by all truth assignments)

   All literals in the term can be made true as there are none.
6. Satisfiability is easy to detect. Any DNF containing at least one term is SAT.
   The empty DNF (no terms) is unsatisfiable.

7. Validity is hard to detect (as hard as an arbitrary propositional formula).

CSC2512: CNF

CNF

a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	
1	1	0	0
1	1	1	

(a)

(-a, c)

Determining if there is a one anywhere (satisfiable) for **F** becomes combinatorial as each clause makes a different set of truth assignments unsatisfying.

CSC2512: DNF

DNF

a	b	c	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	
1	1	0	1
1	1	1	

[a]

[-a, c]

Determining if there is a zero (not valid) anywhere for **F** becomes combinatorial as each term makes a different set of truth assignments satisfying

CSC2512: Formula Transformations

Negation Formal Form

NNF: negations only apply to propositional variables (push all negations in using DeMorgan's laws

$$\neg(f_1 \wedge f_2) = (\neg f_1 \vee \neg f_2)$$
$$\neg(f_1 \vee f_2) = (\neg f_1 \wedge \neg f_2)$$

Transformation to NNF preserves satisfiability.

If NNF(F) is F converted to negation normal form then

F is satisfiable if and only if NNF(F) is satisfiable

More strongly NNF preserves models

$\pi \models$ **F if and only if** $\pi \models$ **NNF(F)**

CSC2512: Satisfiability

Converting to CNF.

If we have an arbitrary propositional formula **f** how do we convert it to a CNF so that a SAT solver can be used?

- Could **multiply** out the formula to obtain a conjunction of disjunctions.

  $(a \wedge b) \vee (c \wedge d)$
  $(a \vee (c \wedge d)) \wedge (b \vee (c \wedge d))$
  $(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d))$

- This leads to an CNF that can be exponentially larger than **F** (but perhaps useful in limited contexts?)
- Multiplying out preserves models

CSC2512: Satisfiability

Converting to CNF in time poly in the size (length) of **f**.

We introduce new variables (Tseitin 1970)

First convert to NNF (negation normal form)

CSC2512: Satisfiability

ToCNF(f)

if f is a literal

return (f, {})

if f = $f_1 \wedge f_2$

$(g_1,\{F_1\})$ = ToCNF(f_1)

$(g_2,\{F_2\})$ = ToCNF(f_2)

Let g be a new propositional variable

return $(g,\{F_1\} \cup \{F_2\} \cup \{(-g, g_1), (-g, g_2), (-g_1, -g_2, g)\})$

if f = $f_1 \vee f_2$

$(g_1,\{F_1\})$ = ToCNF(f_1)

$(g_2,\{F_2\})$ = ToCNF(f_2)

Let g be a new propositional variable

Return$(g,\{F_1\} \cup \{F_2\} \cup \{(-g, g_1, g_2), (-g_1, g), (-g_2, g)\})$

CSC2512: Satisfiability

Notes:

1. The top level returns (g, **F**)
   - **F** is a set of clauses that capture the condition that g (a propositional variable) is equivalent to the original formula **f**
   - To test the satisfiability of **f** we add the unit clause (g) to **F** and test whether or not the CNF {g U F} is satisfiable.
   - How can we test **validity?**
2. All newly introduced variables are forced (must have a single value) under any assignment to the original variables.
3. **F** is always **satisfiable** and has 2^n satisfying models (where the original formula f has n variables).
4. F ∧ (g) has the same number of models as **f**
5. F ∧ (¬g) has the same number of models as **¬f**
6. So ToCNF does not preserve satisfiability unless we also add the unit clause (g)

CSC2512: Satisfiability

Example

$(a \wedge b) \vee (c \wedge d)$

ToCNF($(a \wedge b) \vee (c \wedge d)$)

ToCNF($(a \wedge b)$)

$= (g_1, \{(-g_1, a), (-g_1, b), (-a, -b, g_1)\})$

ToCNF($(c \wedge d)$)

$= (g_2, \{(-g_2, c), (-g_2, d), (-c, -d, g_2)\})$

$= (g, \{(-g_1, a), (-g_1, b), (-a, -b, g_1),$
$(-g_2, c), (-g_2, d), (-c, -d, g_2),$
$(-g, g_1, g_2), (-g_1, g), (-g_2, g)\})$

a = 1, b = 1, c = 1, d = 1 → g1 = 1 & g2 = 1 & g = 1

a = 1, b = 0, c = 1, d = 0 → g1 = 0 & g2 = 0 → g = 0

CSC2512: Satisfiability

There are other translations to CNF.

A Structure-preserving Clause Form Translation
David A. Plaisted Steven Greenbaum

If we only want to test satisfiability, we don't need to encode $g_i \equiv f_i$ for every subformula f_i.

Plaisted & Greenbaum keep track of the "polarity" of the subformula f_i. Dependent on the polarity they show that encoding $g_i \rightarrow f_i$ or $g_i \leftarrow f_i$ is sufficient to test satisfiability.

This results in fewer clauses.

Some preprocessing techniques automatically convert the longer Tseitin encoding to the shorter Plaisted Greenbaum encoding.

CSC2512: Satisfiability

Encodings: CNF is for most application not a natural language for expressing a problem. Various domains have different "standard" languages.

- Automated Planning: STRIPS or ADL actions specified with first-order variables
- Hardware: Circuits
- Software: Various specification languages (logics with extensions).

Specialized techniques have also been developed to encode problems expressed in these languages in CNF. The encoding can have a tremendous impact on how easy it is to solve the CNF.

CSC2512: Satisfiability

Reasoning with CNF

CNF is used in modern SAT solvers mainly because there is a very simple reasoning rule that can be efficiently implemented.

Definition: **Resolution**
From two clauses (A, x) and (B, -x) (where A and B are sets of literals), the **resolvant** is the new clause:

R[c,c'] = (A,B) with all duplicate literals removed.

- Typically we require A and B not to contain conflicting literals (i.e., v and –v for any variable v). If they do then (A, B) will be a tautological clause. Under this requirement, along with the restriction that there are no duplicate literals, the resolvant, if defined, is unique.

CSC2512: Satisfiability

Resolution is **sound**: Any truth assignment that satisfies c and c' must satisfy R[c,c'] (if c and c' are resolvable).

That is resolution generates **logical consequences.**

if $\pi \models c \wedge c'$ **then** $\pi \models R[c,c']$

CSC2512: Satisfiability

Definition: **Resolution Proof of a clause c_n from a set of clauses C**

A sequence of clauses $c_1, c_2, \ldots, c_n$ such that:

Each c_i is either

1. A member of the set of input clauses C
2. or was derived by a resolution step from two prior clauses in the sequence c_j and c_k (j, k < i)

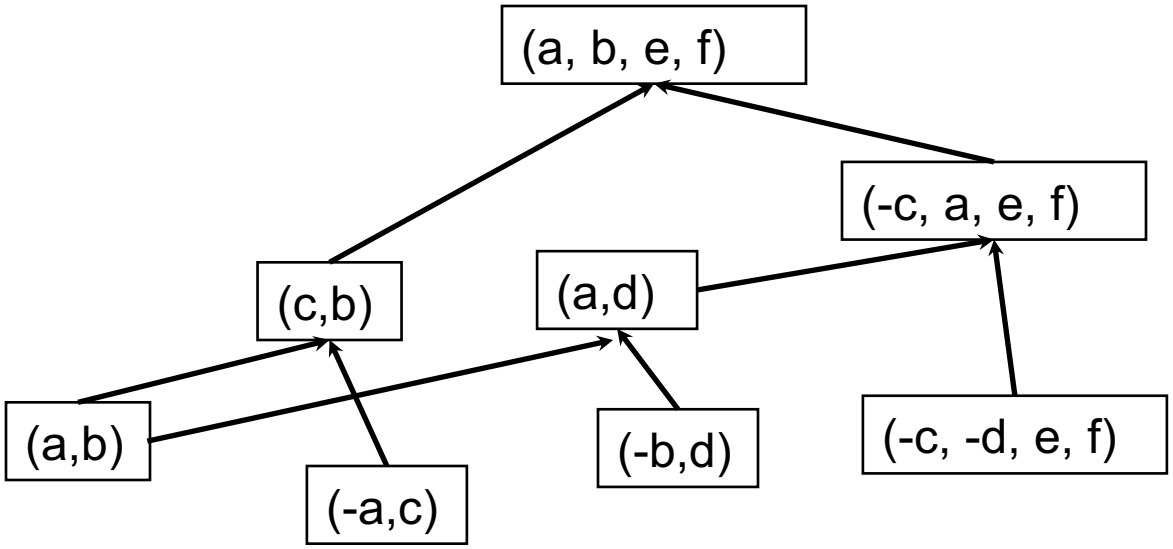
The sequence can also be represented as a DAG (directed acyclic graph).

CSC2512: Satisfiability

C = {(a, b) (-a, c) (-b, d) (-c, -d, e, f)}

There is a resolution proof of (a, b, e, f) from C:

(a, b), (-a, c), (c, b), (-b, d), (a, d), (-c, -d, e, f) (-c, a, e, f) (b, a, e, f)



CSC2512: Satisfiability

Definition: **Resolution Refutation** of a set of clauses C is a resolution proof of the empty clause '()' from C.

From soundness, any truth assignment satisfying C must satisfy the empty clause, but no truth assignment satisfies the empty clause ➔ Proves that C is unsatisfiable

CSC2512: Satisfiability

Resolution is **Refutation Complete:**

If C is unsatisfiable there exists a resolution refutation of C.

Two computational difficulties

a) Finding a resolution refutation
b) Size of the refutation

Resolution is a relatively weak proof system. Well known families of CNFs whose shortest resolution proofs grow exponentially in size.

- Pigeon Hole Principle PHP first such problem shown to require exponential sized resolution proofs
- Other proof systems are known to have short proofs of PHP

CSC2512: Satisfiability

Two computational difficulties

a) Finding a resolution refutation
b) Size of the refutation

Even if a short proof exists finding it might be hard

1. Notion of automatizability from proof theory.
2. For general resolution we can find a proof (given that one exists) in time $2^{O(n\log n * \log S)}$ where n is the number of variables and S is the size of the shortest proof.
3. Unfortunately not a very practical method but does indicate that theoretically resolution proofs can be found in exponential time—some proof systems are not automatizable.

CSC2512: Satisfiability

Davis Putnam (DP) [1960s] gave a procedure for determining satisfiability of a CNF formula. The procedure employs resolution in a systematic way so that if a resolution refutation is not found, none exists.

Ordered Resolution to test the set of clauses C:

1. Pick an ordering of the variables i[1], i[2], … i[n]
2. $C_0 = C$
3. For j = 1 to n
    1. Let p = i[j] (the j'th variable to be eliminated)
    2. Let X = {all clauses of C_{j-1} that contain p}
       Y = {all clauses of C_{j-1} that contain –p}
    3. Apply resolution to all pairs of clauses from X and Y to obtain a set of new clauses R.
    4. $C_j = C_{j-1} – X – Y + R$
    5. If C_j contains {} return UNSAT
4. Return SAT

CSC2512: Satisfiability

Example:

C_0 = (a, -b), (-a, b), (-b, c), (a, c), (a, -c), (-b, -c)

[a] X = (a, -b), (a,c), (a, -c) Y = (-a, b),
R = (b, c), (b, -c)

C_1 = (-b, c), (-b, -c), (b, c), (b, -c)

[b] X = (b,c) (b, -c) Y = (-b, c) (-b, -c)
R = (c) (-c)

C_2 = (c) (-c)

[c] X = (c) Y = (-c)
R = ()

C_3 = ()

CSC2512: Satisfiability

Example:

C_0 = (a, -b), (-a, b), (-b, c), (a, c), (a, -c),

[a] X = (a, -b), (a,c), (a, -c) Y = (-a, b),
R = (b, c), (b, -c)

C_1 = (-b, c), (b, c), (b, -c)

[b] X = (b,c) (b, -c) Y = (-b, c)
R = (c)

C_2 = (c)

[c] X = (c) Y = {}
R = {}

C_3 = {}

CSC2512: Satisfiability

Correctness:

C_i is satisfiable if and only if C_{i-1} is.

Note: Correctness is sufficient to see that the algorithm is sound:

If this is true then when j = n either C_n is the empty set of clauses or it contains the empty clause. (All variables have been removed). Thus we can immediately determine if C_n is satisfiable or not. Working back to the prior clause sets C_i, we see that this determines whether or not the original clause set C_0 is satisfiable.

CSC2512: Satisfiability

Correctness:

Let $C_j = C_{j-1} – X – Y + R$ we want to prove that C_j if and only if C_{j-1} is.

Proof: At stage j, say that C_j is SAT and let π be a satisfying truth assignment.

Find all clauses of C_{j-1} not satisfied by π (note, π does not assign a value to p since C_j does not contain p)

Claim: all of these unsatisfied clauses contain p in only one polarity. If this is true then we simply extend π to make that polarity of p true, and thus satisfy all of the clauses of C_{j-1}.

CSC2512: Satisfiability

Say that this is not true, so there is c_1=(p,A) and c_2 = (-p, B) unsatisfied by π. That is, no literal in A and no literal in B is made true by π.

However, we have that (A,B) is in C_j, and satisfied by π, i.e., at least one literal from either A or B is made true, thus one of c_1 or c_2 must be satisfied by π--**contradiction.**

Hence, all clauses of C_{j-1} not satisfied by π must contain p in the same polarity and so C_j is satisfiable by π extended with the right assignment to p.

- **For (A,B) to be in C_j, c1 and c2 must be resolvable, i.e, not a tautology. Why must this be the case?**
- **Why can we have a clause c unsatisfied by π that does not contain p in either polarity?**

CSC2512: Satisfiability

Correctness:

Let $C_j = C_{j-1} – X – Y + R$

Proof:

In the other direction:

if C_{j-1} is satisfiable so is C_j: C_j contains only the clauses of C_{j-1} and resolvants of clauses of C_{j-1}.

CSC2512: Satisfiability

Davis Putnam (DP)

Ordered Resolution to test the set of clauses C:

1. Pick an ordering of the variables i[1], i[2], … i[n]
2. $C_0 = C$
3. For j = 1 to n
    1. Let p = i[j] (the j'th variable)
    2. Let X = {all clauses of C_{j-1} that contain i[j]}
       Y = {all clauses of C_{j-1} that contain –i[j]}
    3. Apply resolution to all pairs of clauses from X and Y to obtain a set of new clauses R.
    4. $C_j = C_{j-1} – X – Y + R$ //CORRECTNESS. New C_j is satisfiable //if and only if old C_{j-1} is.
    5. If C_j contains {} return UNSAT //C_j is UNSAT
4. Return SAT //At this stage C_j = the empty set of clauses // ➔ a satisfiable set of clauses

CSC2512: Satisfiability

DP produces resolution proofs.

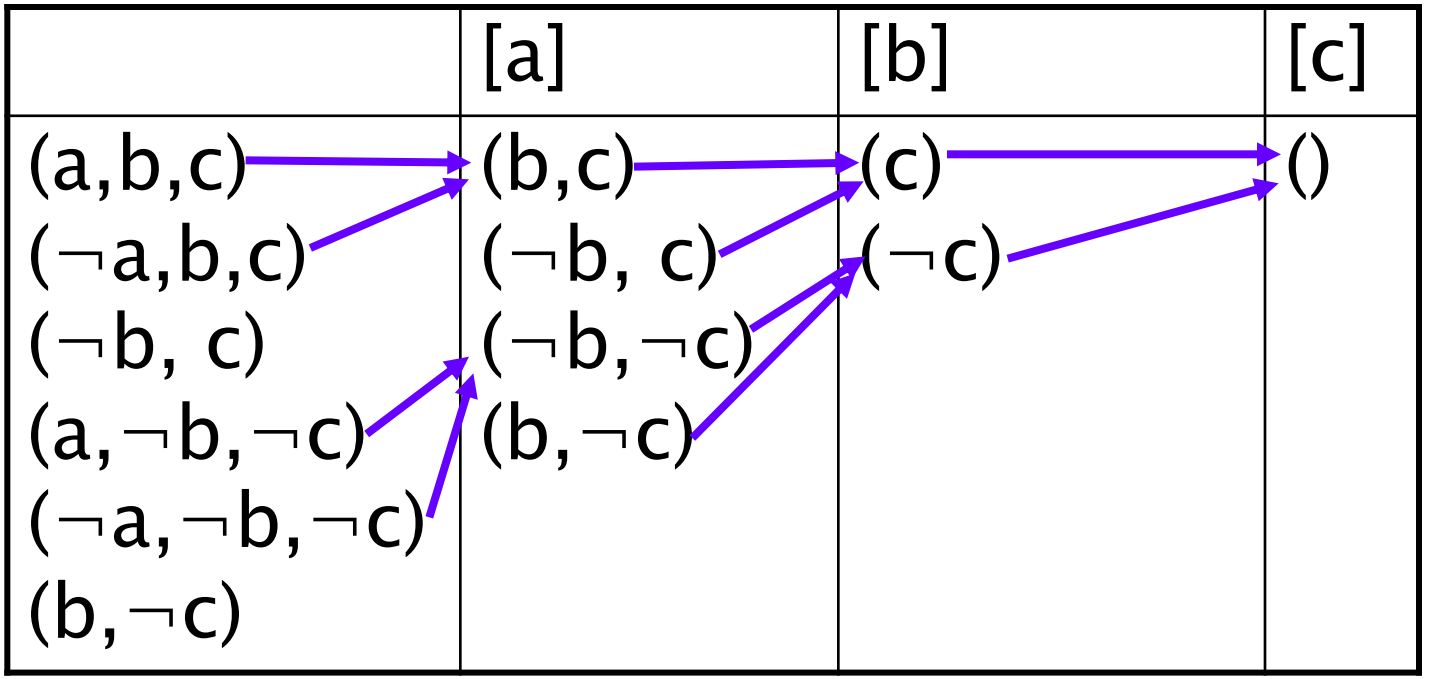
- DP was developed prior to resolution, but every DP run that yields the empty clause contains an resolution proof.

	[a]	[b]	[c]
(a,b,c) (¬a,b,c) (¬b, c) (a,¬b,¬c) (¬a,¬b,¬c) (b,¬c)	(b,c) (¬b, c) (¬b,¬c) (b,¬c)	(c) (¬c)	()

CSC2512: Satisfiability

DP produces resolution proofs.

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



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

CSC2512: Satisfiability

Two years later Davis, Logemann and Loveland developed a new procedure for testing SAT (also predating resolution).

This procedure required only linear space. The algorithm became known as DPLL (although Putnam didn't play a role).

DPLL was a backtracking search algorithm (backtracking originated much earlier)

Davis, Martin; Logemann, George, and Loveland, Donald (1962). "A Machine Program for Theorem Proving". *Communications of the ACM* **5** (7): 394–397. doi:10.1145/368273.368557.

CSC2512: Satisfiability

DPLL(π, F) // Initially F is the input formula.
π is an empty set of literals (truth assignment)

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** //Prefer a **v** appearing
  //in a unit clause if one exists

  F' = F|$_v$ //Reduce F

  (SAT?,π') = **DPLL(π + v, F')**

    if SAT? == SAT return (SAT,π')

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

  return DPLL(π + -v, F')

CSC2512: Satisfiability

Reduction:

$F|_l$ F reduced by literal l

Remove all clauses of F that contain l
Remove –l from all remaining clauses.

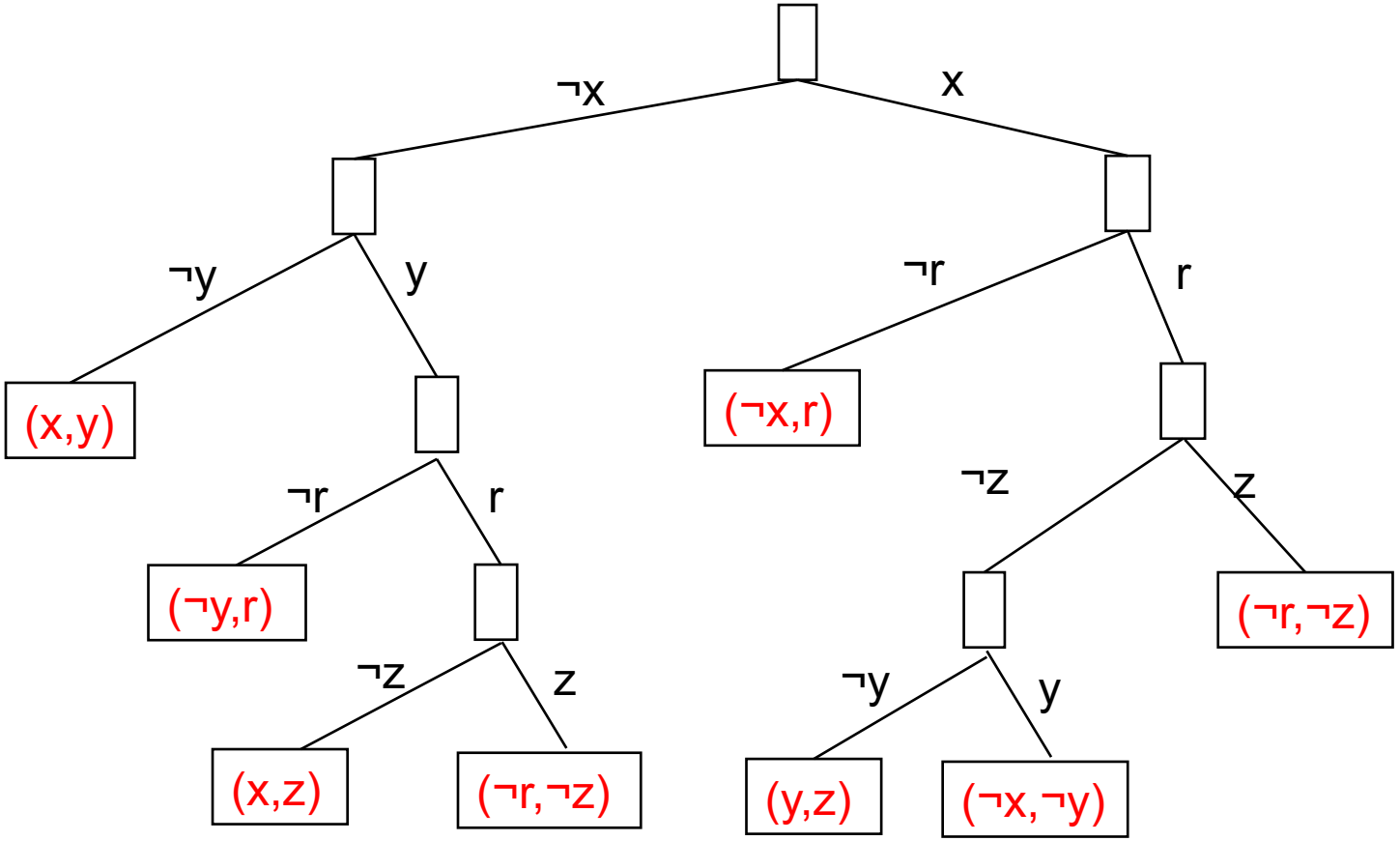
$\{(a, b, -d), (d, c, e), (g, h, e)\}|_{-d}$
$= \{(c, e), (g, h, e)\}$

Note $(F|_a)|_b = (F|_b)|_a$, so we write $F|_{a,b}$

DPLL

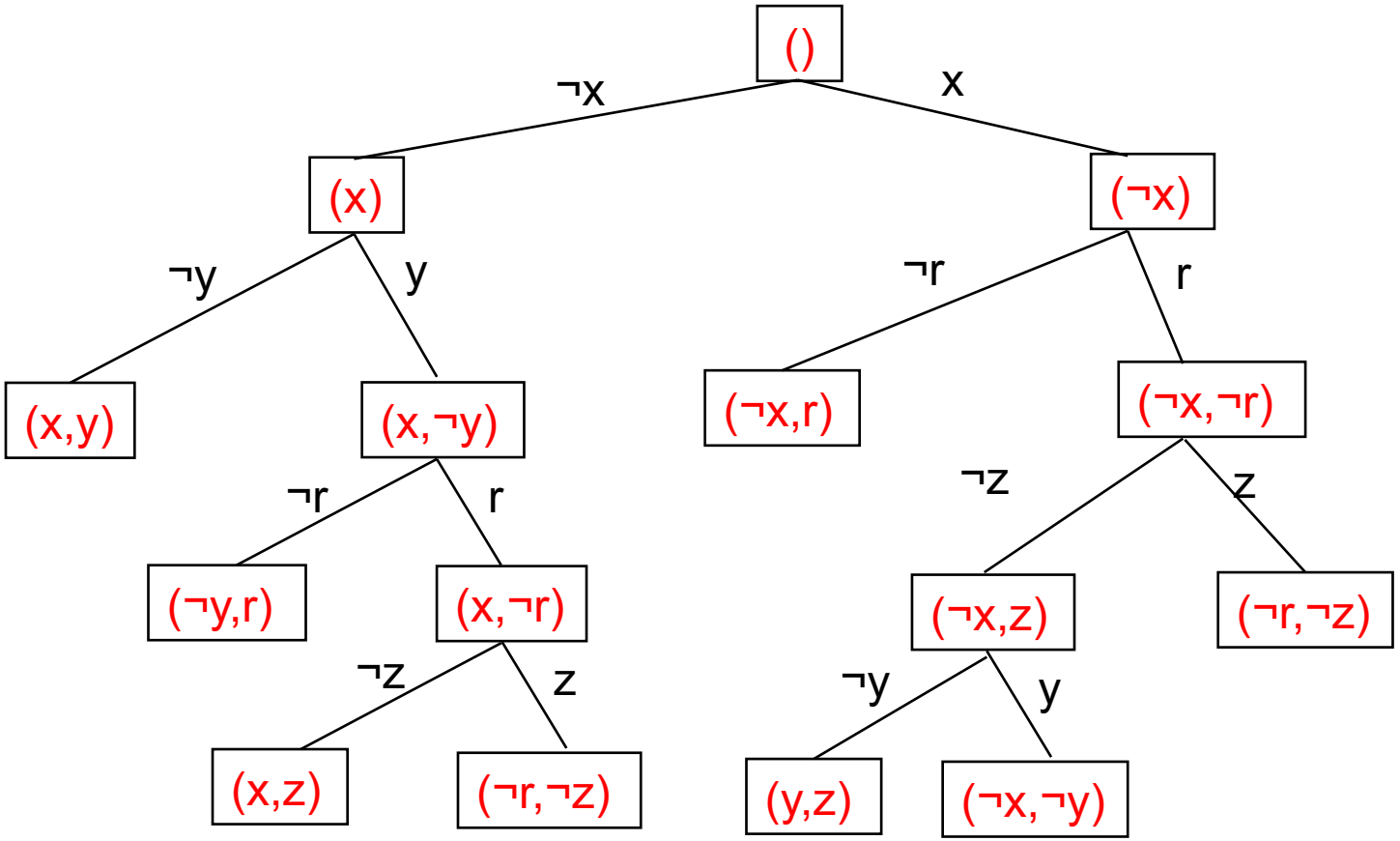
Example:

F = (¬x, r), (¬y, r), (x, z), (y, z), (x, y), (¬x, ¬y), (¬z, ¬r)



DPLL

From every execution of DPLL yielding UNSAT we can extract a resolution refutation. Label each node with resolvent of its two children



DPLL

The resultant resolution DAG is a tree.