

- historically, the first implementation of an algorithm for deciding whether a propositional formula is unsatisfiable was by Paul Gilmore in 1960
- he used the so-called **quantification method** which turned out to be very inefficient
- Martin Davis and Hillary Putnam introduced a more efficient method also in 1960
- it was significantly improved by G. Logeman and Donald Loveland, so it is called **DPLL procedure**
- it is a **negative calculus**, so it shows unsatisfiability of a formula
- it is a **analysing calculus**, so it starts working with the formula wanted to be proved unsatisfiable
- its only axiom is  $\perp$
- it accepts as input formulas in CNF

## DPLL Procedure

Problem: say if a propositional formula  $F$  is unsatisfiable

Input a propositional formula  $F$

Output yes or no

- 1: If  $F$  contains the empty clause, then terminate with yes
- 2: If none of the following rules is applicable, then terminate with no
- 3: *Tautology Rule* a clause can be removed if it contains a pair of complementary atoms  $A$  and  $\neg A$
- 4: *Pure Literal Rule* a clause can be removed if it contains a literal  $L$ , and  $F$  does not contain an occurrence of the complementary literal  $\bar{L}$
- 5: *Unit Clause Rule* a literal  $L$  can be removed in a clause if  $F$  contains a unit clause of the form  $\bar{L}$
- 6: *Subsumption Rule* a clause can be removed if it is a superset of another clause in  $F$

7: *Splitting Rule* if  $F$  can be put of the form

$$(G_1 \vee L) \wedge \dots \wedge (G_m \vee L) \wedge (H_1 \vee \neg L) \wedge \dots \wedge (H_n \vee \neg L) \wedge G$$

where neither  $L$  nor  $\bar{L}$  occur in  $G_i, H_j, G$ , then  $F$  can be replaced by the CNF of

$$(G_1 \wedge \dots \wedge G_m \wedge G) \vee (H_1 \wedge \dots \wedge H_n \wedge G)$$

Example of DPLL application

$$F = \{\{A, B, \neg C\}, \{A, \neg B\}, \{\neg A\}, \{C\}, \{D\}\}$$

$\{\{\underline{A}, B, \neg C\}, \{A, \neg B\}, \{\neg A\}, \{C\}, \{D\}\}$	unit clause
$\{\{B, \neg C\}, \{\underline{A}, \neg B\}, \{\neg A\}, \{C\}, \{D\}\}$	unit clause
$\{\{B, \neg C\}, \{\neg B\}, \{\underline{\neg A}\}, \{C\}, \{D\}\}$	pure literal
$\{\{B, \underline{\neg C}\}, \{\neg B\}, \{C\}, \{D\}\}$	unit clause
$\{\{\underline{B}\}, \{\neg B\}, \{C\}, \{D\}\}$	unit clause
$\{\{\}, \{C\}, \{D\}\}$	termination rule, formula unsatisfiable

Example of DPLL application

$$F = \{\{A, \neg B\}, \{\neg C, \neg B\}, \{C, B\}, \{A, \neg C\}\}$$

$\{\{A, \neg B\}, \{\neg C, \neg B\}, \{C, B\}, \{A, \neg C\}\}$	splitting rule
$(A \wedge \neg C \wedge (A \vee \neg C)) \vee (C \wedge (A \vee \neg C))$	
$\{\{A, C\}, \{\underline{\neg C}, C\}, \{A, \underline{\neg C}, C\}, \{A, \neg C\}, \{\neg C, A\}\}$	tautology rule
$\{\{A, C\}, \{\underline{A, \neg C}\}, \{\neg C, A\}\}$	subsumption rule
$\{\{A, \underline{C}\}, \{\underline{\neg C}, A\}\}$	splitting rule
$\{\{\underline{A}, \underline{A}\}\}$	subsumption rule
$\{\{A\}\}$	pure literal
$\{\}$	termination rule, formula satisfiable

- **Exercise:** Prove using DPLL the same formulas as done for sequent calculus. Compare the proofs.

- **Theorem 9.** *The DPLL procedure is sound and complete*
- **Exercise:** prove the soundness part of this theorem

- the DPLL procedure was originally designed for proving the unsatisfiability of FOL formulas
- it is quite inefficient in doing so, so in 1965 J. Alan Robinson developed the **resolution principle** to overcome this inefficiency
- now we present the propositional version of resolution
- if  $\{L, L_1, \dots, L_n\}$  and  $\{\bar{L}, K_1, \dots, K_m\}$  are two clauses, then  $\{L_1, \dots, L_n, K_1, \dots, K_m\}$  is the **resolvent**
- if  $C_1$  and  $C_2$  are two clause with resolvent  $C$ , then Robinson showed  $C$  is logical consequence of  $\{C_1, C_2\}$
- the resolution principle can be repeatedly used to simplified clauses in showing unsatisfiability, until the empty clause is reached

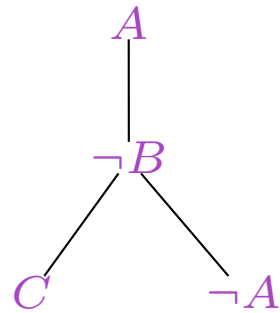
- a **deduction** of a clause  $C$  from  $F$  in CNF is a finite sequence of clauses  $C_1, \dots, C_n = C$  such that each  $C_i$  is either a clause in  $F$ , or a resolvent from previous  $C_k, C_j, j < i, k < i$
- a deduction of the empty clause from  $F$  is called a **refutation**
- in this case, since  $F \models \perp$  the  $F$  should be unsatisfiable
- Example of resolution application:  $\{\{A, B\}, \{\neg A, C\}, \{\neg A, \neg C\}, \{A, \neg B\}\}$
- **Theorem 10.** *The resolution calculus for propositional logic is sound and complete*
- **Exercise:** prove the completeness part of this theorem

- **semantic tableaux** are another means to show logical entailment
- by showing unsatisfiability of the negated formula, with the help of theorems 3 and 4
- so it is **a negative, analysing calculus**
- in order to simplify the presentation, we assume the connectives  $\Leftrightarrow$  and  $\Rightarrow$  have been already removed from the input formula

- a **tableau** is a tree whose nodes are labeled with formulas
- the root of the tree is labeled with the formula that we want to show unsatisfiable
- each branch represent the **conjunction of all formulas** appearing in the branch
- the tableau represent the **disjunction of all its branches**
- then a formula in a tableau is unsatisfiable iff all its branches are unsatisfiable
- this means each branch contains a formula and its negation (ie it is a **closed branch**)

Example of a tableau

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



### Tableaux construction rules

1. initially, start with a single node labeled with the formula
  2. if a branch contains  $\neg\neg G$ , then expand the branch with a new node  $G$
  3. if a branch contains  $G_1 \wedge G_2$ , then expand the branch with two new nodes  $G_1$  and  $G_2$
  4. if a branch contains  $\neg(G_1 \wedge G_2)$ , then add a left and right child  $\neg G_1$  and  $\neg G_2$  to the last node of the branch
  5. if a branch contains  $G_1 \vee G_2$ , then add a left and right child  $G_1$  and  $G_2$  to the last node of the branch
  6. if a branch contains  $\neg(G_1 \vee G_2)$ , then expand the branch with two new nodes  $\neg G_1$  and  $\neg G_2$
- a branch is closed if it contains a pair of complementary formulas; a tableau is unsatisfiable if all its branches are closed.

Example of tableau construction

$$\neg(\neg(C \wedge ((D \vee E) \wedge (\neg E \vee P))) \vee (C \wedge (D \vee P)))$$

- **Exercise:** make tableau proofs for the same set of formulas as in sequent calculus and resolution. Compare the proofs.

- tableaux proofs can be significantly **shorter** than proofs by truth tabling
- if we observe the example, we can say that
  - each node was expanded only one. Such tableaux are called **strict**
  - each branch was closed because of complementary literals. Such tableaux are called **atomically closed**
- **Theorem 11.** *The propositional (strict and atomically closed) tableau method is sound and complete*
- the restriction to strict and atomically closed tableaux considerably **restricts the search space**

- complete calculi for deciding logical entailment run into **time and space problem** as soon as formulas are larger
- unfortunately, many real world application (planning, scheduling) require too many propositional variables
- recently, a new kind of non-complete procedures were developed, which can take care of large formulas
- we will introduce **a general framework** for this kind of algorithms
- and then two instantiations: **GSAT** and **GWSAT**

GenSAT framework Problem: tell if a propositional formula in CNF is satisfiable

Input a propositional formula  $F$

Output a model for  $F$ , or no

```
1: for  $j := 1$  to  $maxtries$  do
2:    $I := initial(F)$ 
3:   for  $k := 1$  to  $maxloops$  do
4:     if  $I \models F$  then
5:       return  $I$ 
6:     else
7:        $I := flip(I, pick(hillclimb(F, I)))$ 
8:     end if
9:   end for
10: end for
11: return "no model found"
```

- where *maxtries* and *maxloops* are some natural numbers
- *initial(F)* selects randomly an interpretation for *F*
- *pick()* randomly selects an element from a given set of proposition
- *flip()* changes the truth value of a selected proposition in an interpretation
- and *hillclimb()* computes a set of variables which can be potentially flipped.

Hillclimb procedure for GenSAT

Input a propositional formula  $F$  and an interpretation  $I$  such that  $I \not\models F$

Output a set of propositions in  $F$

- 1:  $r := \text{rand}(0, 1)$
- 2: return  $\{q \mid \text{select}(F, I, q, r) = \text{true}\}$

- this procedure call a function  $\text{select}()$  which yields a set of variables in function to a random number  $r$

- now we introduce **GSAT** (greedy satisfiability testing) algorithm
- recall that we assume  $F$  in CNF, and  $I$  an interpretation. Then  $F_u(I)$  is the set of unsatisfied clauses of  $F$  in  $I$
- let  $A$  be a proposition in  $F$ , then the **rank** of  $A$  in  $F$  and  $I$  is

$$\text{rank}(F, I, A) := |F_u(I)| - |F_u(\text{flip}(I, A))|$$

- if  $\text{rank}(F, I, A) > 0$  then the number of unsatisfied clauses in  $F$  decreases when  $A$  is flipped

- in GSAT the  $select()$  function is defined as

$$select(F, I, A, r) = \begin{cases} \text{true} & \text{if } rank(F, I, P) \text{ is maximal between all propositions} \\ \text{false} & \text{otherwise} \end{cases}$$

- ie  $select()$  returns all proposition which make a maximal change in the number of satisfied clauses
- this is the reason why it is called “greedy” algorithm
- this procedure will inevitable end in a local minimum of  $|F_u(I)|$

- in experiments, it was observed that the performance of algorithm that avoid local minum is better
- **GWSAT** (greedy satisfiability testing with **random walks**) is an extension of GSAT which randomly selects arbitrary propositions to flip
- in this way proposition with not maximal rank, or even negative, can be selected

$$\text{select}(F, I, A, r) = \begin{cases} \text{true} & \text{if } r > p \text{ and } \text{rank}(F, I, A) \text{ is maximal between all propositions} \\ \text{true} & \text{if } r \leq p \text{ and } A \in C \in F_u(I) \\ \text{false} & \text{otherwise} \end{cases}$$

Comparison of GSAT and GWSAT

$N$	algorithm	forced	unforced
100	GSAT	76.1%	23.5%
	GWSAT	100%	82.2%
200	GSAT	75.3%	8.4%
	GWSAT	99%	48.7%
300	GSAT	77.0%	2.7 %
	GWSAT	100%	26.3%
500	GSAT	70.2%	–
	GWSAT	100 %	

with 3-CNF satisfiable formulas randomly generated (forced), and 3-CNF formulas randomly generated and later tested to be satisfiable (unforced).  $N$  is the number of propositions,  $maxtries = 1$  and  $maxloops = 100N$

- with an increase in *maxtries*, the percentages become even better

$$P(S_n) = 1 - P(\neg S_1)^n$$

where  $S_n$  is the probability of success with *maxtries* =  $n$

- for example, with  $N = 300$  GWSAT will find a model of a satisfiable formula in 15 attempts in 98.8% of cases

- **Exercise:** Install one of the SAT solvers listed in SATLIB

`http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`

Test it with the benchmark problems available also on this site. Write a report describing the characteristics of the selected solver, and the performance on the different benchmarks.

- a certain class of clauses has become important because of its nice **theoretical properties**, as well as it allows to view deduction as procedure execution
- clauses that contain only one literal are called **unit clauses**
- clauses that contain at most one positive literal are called **Horn clauses**
- instead of  $\{A, \neg B_1, \dots, \neg B_n\}$  it is often written as

$$A \leftarrow B_1, \dots, B_n$$

- Horn clauses with only negative literals are called **goal clauses**

$$\leftarrow B_1, \dots, B_n$$

- a Horn clause with a positive literal is called a **definite clause**
- a **definite logic program** is a set of definite clauses
- a **fact** is a clause with only a positive literal

$$A \leftarrow$$

- these clauses have a **declarative meaning** and a **procedural meaning** as well

- we will now see the **theoretical properties** of Horn clauses
- **Proposition 1.** *Let  $\Pi$  be a definite logic program. If  $I_1$  and  $I_2$  are models of  $\Pi$ , then  $I_1 \cap I_2$  is also a model of  $\Pi$ .*
- so let  $M_\Pi$  be the interpretation  $\{A : A \text{ is a proposition and } \Pi \models A\}$
- **Proposition 2.** *Let  $\Pi$  be a definite logic program. Then  $M_\Pi$  is a model of  $\Pi$ , and it is contained in every other model of  $\Pi$ .*
- **Theorem 12.**  *$M_\Pi$  is **the least model** model of program  $\Pi$ .*

- the least model of a program  $\Pi$  can also be computed by means of a **consequence function**  $T_\Pi$  which maps interpretations onto interpretations:

$$T_\Pi(I) = \{A : A \leftarrow B_1, \dots, B_n \in \Pi \text{ and } \{B_i\}_{i=1}^n \subseteq I\}$$

- $T_\Pi$  is **monotone** on the lattice of interpretations under the subset relation
- then it is possible to define the sequence:

$$T_\Pi \uparrow 0 = \emptyset$$

$$T_\Pi \uparrow 1 = T_\Pi(\emptyset)$$

...

$$T_\Pi \uparrow (n + 1) = T_\Pi(T_\Pi \uparrow n)$$

- it can be proved that this sequence of interpretations is **monotonic increasing**, and then (Knaster-Tarski theorem) has a **least fixpoint**  $\text{lfp}(T_{\Pi})$
- **Theorem 13.** *Let  $\Pi$  be a definite logic program. Then  $M_{\Pi} = \text{lfp}(T_{\Pi})$*
- **Exercise:** prove this theorem
- in fact, for each finite propositional logic program the least fixpoint can be reached in a **finite** number of steps
- so in order to check if a formula  $F$  is a logical consequence of a propositional logic program  $\Pi$ , then it is sufficient to check the least fixpoint of  $T_{\Pi}$

- this procedure can be carried out in **polynomial time** on the length of the formulas
- let HORNSAT be the problem of deciding if a set of Horn clauses is satisfiable

**Theorem 14.** *HORNSAT is P*

- then it is much easier to compute consequences of Horn clause, than to compute consequences of arbitrary set of formulas
- calculi for logic program are generally based on **SLD resolution**

- most logic programming systems are based on **SLD resolution**, where SLD stands for selection, linear, definite
- **Definite** is because resolution is applied only to definite clauses (positive logic programs)
- **Linear** is a restriction to resolution in which after the initial clause is chosen, every resolvent is obtained from resolving the previous clause in the deduction, with other arbitrary clause
- **Selection** stands because there is a function that selects which is the next literal to resolve from the current clause
- in Prolog systems this functions stands for selecting the first literal in the clause as a list
- **Theorem 15.** *SLD resolution is complete for definite logic programs.*
- SLD resolution is a complete and efficient procedure for computing satisfiability of logic programs