SAT: Propositional Satisfiability

22c:145 Artificial Intelligence
Russell & Norvig, Ch. 7.6

Validity vs. Satisfiability

- Validity:
  - A sentence is valid if it is true in every interpretation (every interpretation is a model).
  - A sentence s is a valid consequence of a set S of sentences if (S => s) is valid.
  - Proof methods: Truth -Tables and Inference Rules
- Satisfiability:
  - A set of sentences is satisfiable if there exists an interpretation in which every sentence is true (it has at least one model).
  - Proof Methods: Truth-Tables and
    The Davis-Putnam-Logeman-Loveland procedure (DPLL).

SAT: Propositional Satisfiability

- An instance of SAT is defined as (X, S)
  - X: A set of 0-1 (propositional) variables
  - S: A set of sentences (formulas) on X
- Goal: Find an assignment f: X -> {0, 1} so that every sentence becomes true.
- SAT is the first NP-complete problem.
  - Good News: Thousands of problems can be transformed into SAT
  - Bad News: There are no efficient algorithms for SAT

Truth Table for Satisfiability

- A propositional formula φ is satisifiable iff one of the values of φ is *True*.
- Example: $\varphi = (a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$

a	*b*	*c*	$a \vee c$	$b \vee c$	$\neg a \vee \neg b \vee \neg c$	φ
0	0	0	0	0	1	0
0	0	1	1	1	1	1
0	1	0	0	1	1	0
0	1	1	1	1	1	1
1	0	0	1	0	1	0
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	0	0

{ a = 0, b = 0, c = 1 } is a model of φ

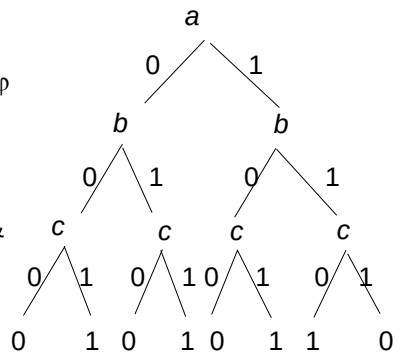
Simplification of Truth Table

- As long as φ has a value *True*, we may stop working.
- Several rows may be merged into one with don't-care values (x)
- Example: $\varphi = (a \vee c) \,\&\, (b \vee c) \,\&\, (\neg a \vee \neg b \vee \neg c)$

a	*b*	*c*	$a \vee c$	$b \vee c$	$\neg a \vee \neg b \vee \neg c$	φ
0	x	0	0	~~0~~	~~1~~	0
0	0	1	1	1	1	1
~~0~~	~~1~~	~~0~~	~~0~~	~~1~~	~~1~~	~~0~~
0	1	1	1	1	1	1
1	0	0	1	0	1	0
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	0	0

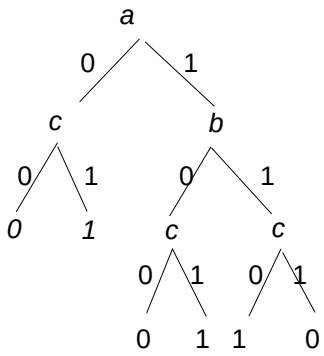
Truth Table as a Binary Tree

- Each internal node has a variable
- Two children represent *True* and *False*.
- The leaf nodes have the value of φ
- Each row of Truth Table is a path from the root to a leaf.
- Example: $\varphi = (a \vee c) \,\&\, (b \vee c) \,\&\, (\neg a \vee \neg b \vee \neg c)$



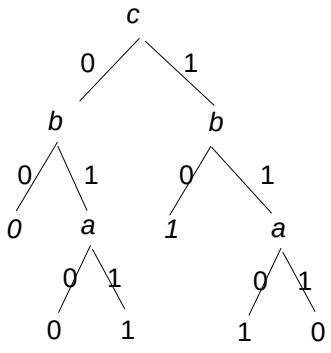
Simplification of Binary Tree

- Don't-care variables can be removed from the tree
- Example: $\varphi = (a \vee c) \,\&\, (b \vee c) \,\&\, (\neg a \vee \neg b \vee \neg c)$

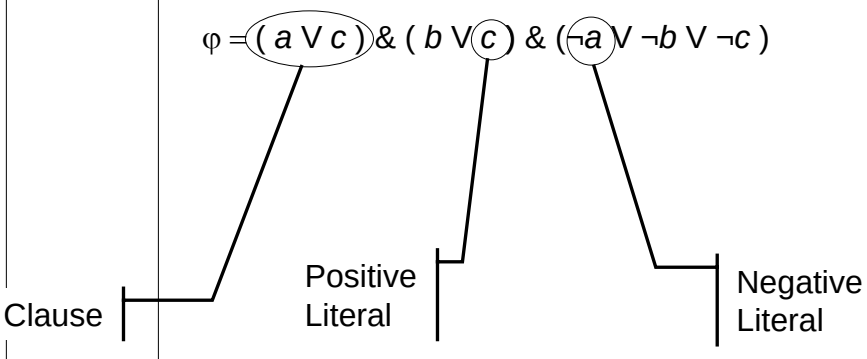


Simplification of Binary Tree

- Order of variables may affect the size of the tree
- Example: $\varphi = (a \vee c) \,\&\, (b \vee c) \,\&\, (\neg a \vee \neg b \vee \neg c)$
- The Davis-Putnam-Logmann-Loveland method exploit these ideas.



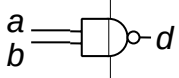
Conjunctive Normal Form (CNF)



Propositional Clauses

- Every propositional constraint can be converted into a set of equivalent clauses.
  - **$S = \{ C_1, C_2, \ldots, C_m \} = C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_m$**
- A clause is a disjunction of literals.
  - **$C = (L_1 \vee L_2 \vee \ldots \vee L_k)$**
- A literal is either a variable or the negation of a variable.
  - **$L = x$ or $L = \neg x$**
- A set of clauses is also said to be in Conjunctive Normal Form (CNF).

Gate CNF



$\varphi_d = [d \equiv \neg(a \,\&\, b)]$
$= [d \rightarrow \neg(a \,\&\, b)] \,\&\, [\neg(a \,\&\, b) \rightarrow d]$
$= (\neg d \vee \neg a \vee \neg b)[\neg d \rightarrow (a \,\&\, b)]$
$= (a \vee d)(b \vee d)(\neg a \vee \neg b \vee \neg d)$

$\varphi_d = [d \equiv \neg(a\ b)]$
$= \neg[d \oplus \neg(a\ b)]$
$= \neg[\neg(a\ b)\neg d \vee a\ b\ d]$
$= \neg[\neg a\ \neg d \vee \neg b\ \neg d \vee a\ b\ d]$
$= (a \vee d)(b \vee d)(\neg a \vee \neg b \vee \neg d)$

DIMACS Format

```
c This is an example of
c an SAT instance in DIMACS format

p cnf 3 5
1 2 0
1 3 0
-1 -2 0
-1 -3 0
-2 -3 0
```

X1 V X2
X1 V X3
-X1 V -X2
-X1 V -X3
-X2 V -X3

DIMACS: Discrete Mathematics and Computer Science

More on Assignments

- Assignments: {$a = 0, b = 1$} = $\neg a \,\&\, b$
  - Partial (some variables still unassigned)
  - Complete (all variables assigned)
  - Conflicting (imply $\neg\varphi$)

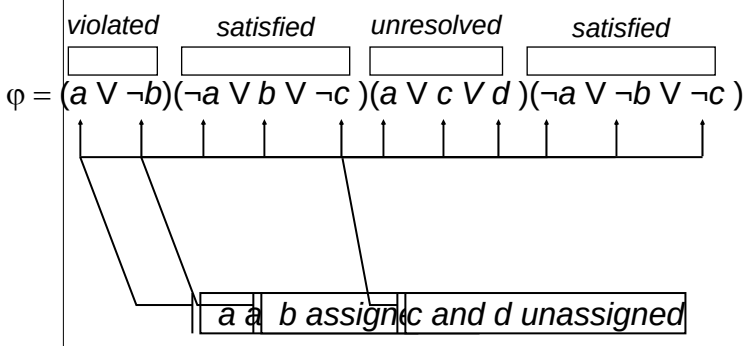
$\varphi = (a \vee c) \,\&\, (b \vee c) \,\&\, (\neg a \vee \neg b \vee \neg c)$

$\varphi \rightarrow (a \vee c)$

$\neg(a \vee c) \rightarrow \neg\varphi$

$\neg a \,\&\, \neg c \rightarrow \neg\varphi$

Literal & Clause Classification



Unit Clause Rule - Implications

- An unresolved clause is *unit* if it has exactly one unassigned literal

  $\varphi = (a \vee c)(b \vee c)(\neg a \vee \neg b \vee \neg c)$

- A unit clause has exactly one option for being satisfied

  $$a\ b \rightarrow \neg c$$

  i.e. *c* must be set to 0.

Pure Literal Rule

- A variable is *pure* if its literals are either all positive or all negative
- Satisfiability of a formula is unaffected by assigning pure variables the values that satisfy all the clauses containing them

  $\varphi = (a \vee \bigcirc{c})(b \vee \bigcirc{c})(b \vee \neg d)(\neg a \vee \neg b \vee d)$

- Set *c* to 1; if φ becomes unsatisfiable, then it is also unsatisfiable when *c* is set to 0.

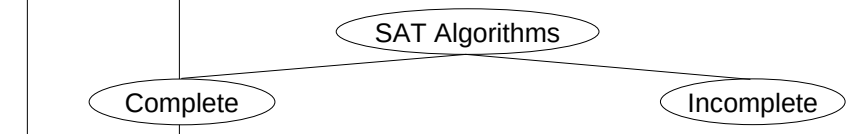
Resolution/Consensus

- General technique for deriving new clauses
  Example: $\omega_1 = (\neg a \vee b \vee c)$, $\omega_2 = (a \vee b \vee d)$
  Resolution:

$$res(\omega_1, \omega_2, a) = (b \vee c \vee d)$$

- Complete procedure for satisfiability [Davis, JACM'60]
- Impractical for real-world problem instances
- Application of restricted forms has been successful**!**
  - E.g., always apply restricted resolution
    - $res((\neg a \vee \alpha), (a \vee \alpha), a) = (\alpha)$
      α is a disjunction of literals

A Taxonomy of SAT Algorithms



Can prove unsatisfiability

Backtrack search (DP)

Resolution (original DP)

Recursive learning (RL)

BDDs (Binary Decision Diagram)

...

Cannot prove unsatisfiability

Local search (hill climbing)

Continuous formulations

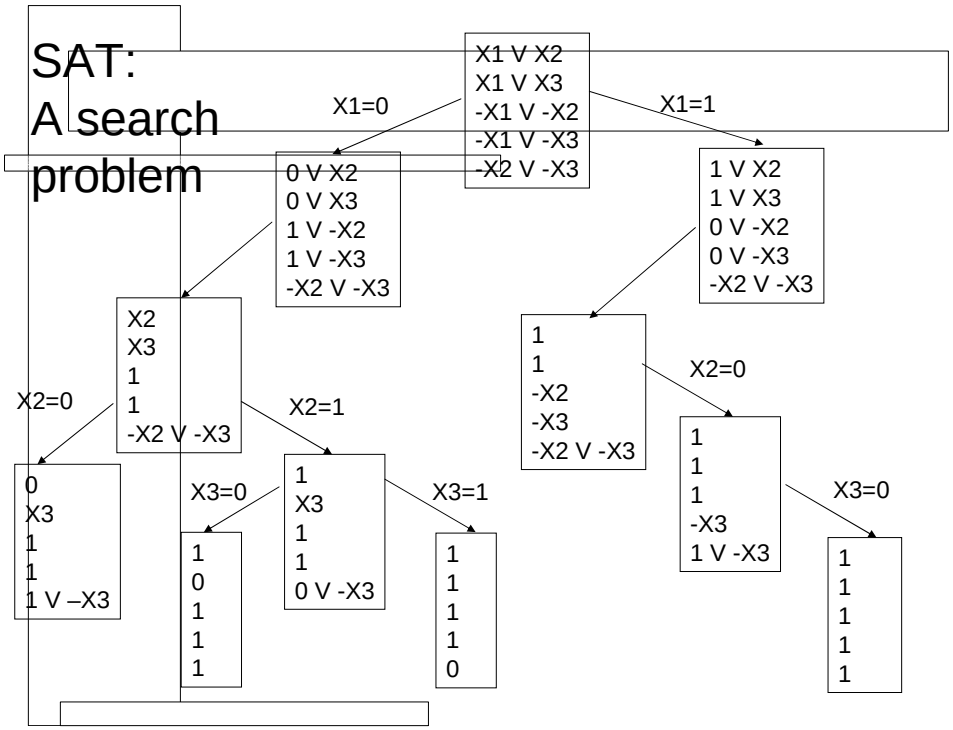
Genetic algorithms

Simulated annealing

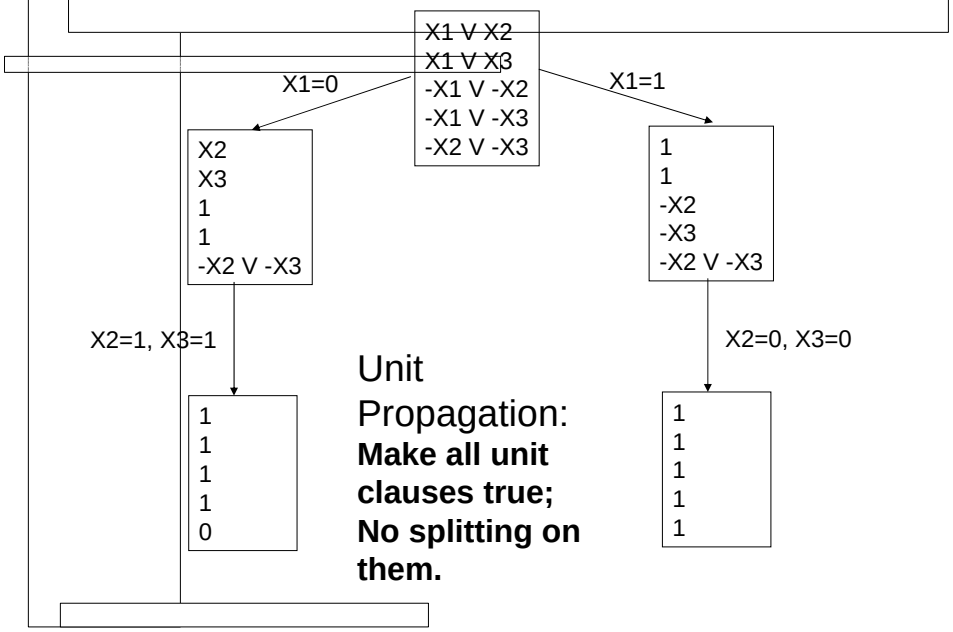
Tabu search

...

SAT: A search problem



Simplification Rules: 1 V C = 1, 0 V C = C



Unit Propagation:
Make all unit clauses true;
No splitting on them.

The Davis-Putnam-Logemann-Loveland Algorithm (1960)

```
function Satisfiable ( clause set S )  return { 0, 1 }
  repeat              /* unit propagation */
      for each unit clause L in S do
          delete from S every clause containing L
          delete -L from each C in S in which -L occurs
      if  S is empty then return 1
      else if  a clause in S is empty then return 0
  until no more new unit clauses changes

  choose a literal L occurring in S          /* splitting  */
  if  Satisfiable( S U { L } ) then return 1
  else if  Satisfiable( S U { -L } )  then return 1
  else return 0
```

DPLL uses Depth-First-Search

- DFS with Backtrack: Instead of maintaining a path of nodes, only one node is maintained. The node is modified when going down and everything is undone when going up.
- The branching factor is dictated by the splitting rule.

DPLL uses Backtrack Search

- Implicit enumeration
- Iterated unit-clause rule
  - Boolean constraint propagation
- Pure-literal rule
- Chronological backtracking in presence of conflicts
- The worst-time complexity is exponential in terms of the number of variables.

Implementing The DPLL Algorithm

- A destructive data structure is needed for clauses: Instead of copying clauses, modify them and then undo modification when backtracking.
- Efficient algorithms for unit-propagation.
- There are many choices for selecting a literal to split (heuristics are needed).

The n-queen problem

- Place n queens on an n x n chessboard so that no two queens attack each other.
- Conditions:
  - Each row has a unique queen
  - No two queens on the same column
  - No two queens on the same diagonal
- Use n^2 boolean variables: q_{ij} is true iff the queen on row i is in column j.

Example: 4-queen problem

- 16 variables: q11, q12, q13, q14, q21, …, q44
- Each row has a unique queen
  1. q11 | q12 | q13 | q14
  2. q21 | q22 | q23 | q24
  3. q31 | q32 | q33 | q34
  4. q41 | q42 | q43 | q44
  5. -q11 | -q12

     ...

     -q43 | -q44

Example: 4-queen problem

- 16 variables: q11, q12, q13, q14, q21, …, q44
- No two queens on the same column
  1. -q11 | -q21
  2. -q11 | -q31
  3. -q11 | -q41

  …

  -q34 | -q44

- No two queens on the same diagonal
  1. -q11 | -q22, -q11 | -q33, -q11 | -q44
  2. -q12 | -q21, -q12 | -q23, -q12 | -q34

  …

Satbox's Results on n-queen prob.

n	solutions	1 soln.	all soln.
8	92	0.00	0.01
10	724	0.00	0.04
20	>100,000	0.00	240
40	-	0.16	-
60	-	1.30	-
80	-	4.30	-
100	-	11.20	-

- Times are in seconds.
- There are 1646800 clauses for n=100

The Einstein Puzzle

Supposedly, Albert Einstein wrote this riddle, and said 98% of the world could not solve it.

- There are 5 houses in five different colors.
- In each house lives a person with a different nationality.
- These 5 owners drink a certain drink, smoke a certain brand of tobacco and keep a certain pet.
- No owners have the same pet, smoke the same tobacco, or drink the same drink.
- The question is: Who owns the fish?

Hints to Einstein Puzzle

- The Brit lives in the red house
- The Swede keeps dogs as pets
- The Dane drinks tea
- The green house is adjacent on the left of the white house
- The green house owner drinks coffee
- The person who smokes Pall Mall raises birds
- The owner of the yellow house smokes Dunhill
- The man living in the house right in the center drinks milk

Hints to Einstein Puzzle (cont)

- The Norwegian lives in the first house
- The man who smokes Blends lives next to the one who keeps cats
- The man who keeps horses lives next to the one who smokes Dunhill
- The owner who smokes Bluemaster drinks juice
- The German smokes Prince
- The Norwegian lives next to the blue house
- The man who smokes Blend has a neighbor who drinks water.

Specify Einstein Puzzle in SAT

- The houses are presented by 1, 2, 3, 4, 5.
- Definition of colors */
  - #define red 0
  - #define green 1
  - #define white 2
  - #define blue 3
  - #define yellow 4
- #define color(x,y) ((x)+5*(y))
- Answer: 3 9 15 17 21

Specify Einstein Puzzle in SAT

- The houses are presented by 1, 2, 3, 4, 5.
- Definition of nationality
  - #define brit 5
  - #define swede 6
  - #define dane 7
  - #define norwegian 8
  - #define german 9
- #define lives(x,y) ((x)+5*(y))
- Answer: 28 35 37 41 49

Specify Einstein Puzzle in SAT

- The houses are presented by 1, 2, 3, 4, 5.
- Definition of drinks
  - #define tea 10
  - #define coffee 11
  - #define water 12
  - #define juice 13
  - #define milk 14
- #define drinks(x,y) ((x)+5*(y))
- Answer: 52 59 61 70 73

Clauses in DIMACS Format

```
printf(“p cnf 125 1000\n”);  // actual clauses: 885

for (k = 0; k < 5; k++) {
  // every house has a color
  for (a = 1; a <= 5; a++) printf("%d ", color(a, k));
  printf("0\n");
  for (a = 1; a <= 5; a++) {
     for (b = 1; b < a; b++)     // a color can be used once
        printf("-%d -%d 0\n", color(a, k), color(b, k));
     for (b = 0; b < 5 ; b++) if (b != k)
        // a house can have only one color.
        printf("-%d -%d 0\n", color(a, k), color(a, b));
  }
```

Clauses in DIMACS Format

```
 // The Brit lives in the red house
  for (a = 1; a <= 5; a++) {
    printf("-%d %d 0\n", lives(a, brit), color(a, red));
    printf("%d -%d 0\n", lives(a, brit), color(a, red));
  }

  // The Swede keeps dogs as pets
  for (a = 1; a <= 5; a++) {
    printf("-%d %d 0\n", lives(a, swede), pets(a, dog));
    printf("%d -%d 0\n", lives(a, swede), pets(a, dog));
  }
```

Clauses in DIMACS Format

```
// The man living in the house right in the center drinks milk
    printf("%d 0\n", drinks(3, milk));

// The Norwegian lives in the first house
    printf("%d 0\n", lives(1, norwegian));
```

Clauses in DIMACS Format

```
// The man who smokes Blends lives next to the one
// who keeps cats
  printf("-%d %d 0\n", smokes(1, Blends), pets(2, cat));
  printf("-%d %d 0\n", smokes(5, Blends), pets(4, cat));
  for (a = 2; a <= 4; a++) {
    printf("-%d %d %d 0\n",
      smokes(a, Blends),
      pets(a-1, cat), pets(a+1, cat));
  }
```

Sato's Result

- ------- SATO 3.2.1, 04/2000 on serv16.divms.uiowa.edu ------
- c Input file "einstein.cnf" is open.
- c Reading clauses in DIMACS's format.
- c Max_atom = 125, Max_clause = 1000
- There are 885 input clauses (3 unit, 251 subsumed, 637 retained).

- Model #1: (indices of true atoms)
- 3 9 15 17 21 28 35 37 41 49 52 59 61 70 73 77 81 90 93 99
- 103 106 112 120 124
- The number of found models is 1.
- There are 13 branches (1 succeeded, 9 failed, 0 jumped).

- ---------- Stats ----------
- run time (seconds) 0.00
- build time 0.00
- search time 0.00
- mallocated (K bytes 96.49
- ----------

SUDOKU Puzzle

- Fill numbers between 1 and 9 on a 9x9 square such that each row, each column and each small 3x3 square is a permutation of 1 to 9.

5						4		9
4	9							
			5		9			
9	2	4		3			6	
		8	4		6	9		
	5			2		3	8	4
			3		4			
							3	7
3		9						6

SUDOKU Puzzle

- Fill numbers between 1 and 9 on a 9x9 square such that each row, each column and each small 3x3 square is a permutation of 1 to 9.

5	6	3	2	8	1	4	7	9
4	9	2	7	6	3	8	1	5
1	8	7	5	4	9	6	2	3
9	2	4	8	3	5	7	6	1
7	3	8	4	1	6	9	5	2
6	5	1	9	2	7	3	8	4
2	1	6	3	7	4	5	9	8
8	4	5	6	9	2	1	3	7
3	7	9	1	5	8	2	4	6

How to Code the Puzzle

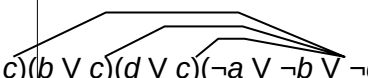
- Let p(x,y,z) be the boolean variable such that the number at (x, y) is z, where x, y, z are in { 1..9 }
- In DIMACS Fomat, p(x,y,z) = 81(x-1)+9(y-1)+z
- The total number of boolean variables is $9^3 = 729$.

How to Code the Puzzle

- At cell (x, y), the number is between 1 and 9:
  p(x,y,1) | p(x,y,2) | p(x,y,3) | … | p(x,y,9)
  There are 81 such clauses.
- Row x is a permutation of 1..9: For any z,
  -p(x,1,z) | -p(x,2,z), -p(x,1,z) | -p(x,3,z), …
  There are 81*36 such clauses.
- Column y is a permutation of 1..9: For any z,
  -p(1,y,z) | -p(2,y,z), -p(1,y,z) | -p(3,y,z), …
- Each small square is a permutation of 1..9: For any z,
  -p(1,1,z) | -p(2,2,z), -p(1,1,z) | -p(2,3,z), …

Resolution (original DP)

- Iteratively apply resolution (consensus) to eliminate one variable each time
  - i.e., resolution between all pairs of clauses containing x and $\neg x$
  - formula satisfiability is preserved
- Stop applying resolution when,
  - Either empty clause is derived $\Rightarrow$ instance is unsatisfiable
  - Or only clauses satisfied or with pure literals are obtained $\Rightarrow$ instance is satisfiable



$\varphi = (a \vee c)(b \vee c)(d \vee c)(\neg a \vee \neg b \vee \neg c)$ Eliminate variable c

$\varphi_1 = (a \vee \neg a \vee \neg b)(b \vee \neg a \vee \neg b)(d \vee \neg a \vee \neg b)$
$= (d \vee \neg a \vee \neg b)$

Instance is SAT **!**

Elimination of variables

- Let Sn be the set of input clauses.
- Let variables in S be ordered as x1, x2, …, xn and we eliminate them from xn to x1 by the following algorithm:

```
For k := n to 1 step -1 do {
    Let Sk be divided into Sk = Pk U Nk U S', where
    Pk contains all clauses having positive xk
    Nk contains all clauses having negative –xk
    Let S(k-1) := S' U Resolution(Pk, Nk, xk)
    If the empty clause is in S(k-1), return UNSAT
}
return SAT.
```

- Assume Resolution(Pk, Nk, xk) apply the resolution to any pair of clauses from Pk and Nk on xk. Duplicate literals in a clause are removed. Tautology clauses are removed from Sk.

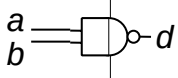
Completeness of Resolution

- If the algorithm returns UNSAT, then S is unsatisfiable.
  - Because resolution is sound, if the empty clause is generated, then S must be unsatisfiable.
- If the algorithm returns SAT, then S is satisfiable, i.e., S has a model.
  - We construct an assigment by assigning truth values from x1 to xn such that when xk is assigned, then all clauses in Sk are true under the current assignement.
  - This assignment is a model when xn is assigned a truth value.

Completeness of Resolution

- The procedure for assigning truth values from x1 to xn:
  - At first, either P1 or N1 is empty, otherwise, the empty clause will be generated from Resolution(P1, N1, x1). We assign P1 to true if N1 is empty; otherwise P1 to false.
  - Suppose we have assigned a truth value to x1, x2, …, x(k-1). If a clause in Nk after removing –xk becomes false, we assign false to xk; otherwise assign true to xk.
- Claim: All clauses in Sk are true in the current assignment after xk is assigned a truth value.
- Proof:
  - Basic case: k = 1.
  - Inductive hypothesis: After x(k-1) is assigned, clauses in S(k-1) are true.
  - Inductive case: The assignment of xk will make every clause in Sk true.

Gate CNF



$\varphi_d = [d \equiv \neg(a \,\&\, b)]$
$= [d \rightarrow \neg(a \,\&\, b)] \,\&\, [\neg(a \,\&\, b) \rightarrow d]$
$= (\neg d \vee \neg a \vee \neg b)[\neg d \rightarrow (a \,\&\, b)]$
$= (a \vee d)(b \vee d)(\neg a \vee \neg b \vee \neg d)$

$\varphi_d = [d \equiv \neg(a\ b)]$
$= \neg[d \oplus \neg(a\ b)]$
$= \neg[\neg(a\ b)\neg d \vee a\ b\ d]$
$= \neg[\neg a\ \neg d \vee \neg b\ \neg d \vee a\ b\ d]$
$= (a \vee d)(b \vee d)(\neg a \vee \neg b \vee \neg d)$

Converting Formula into CNF

- Replace *equivalence, xor*, etc. by *and* (&), *or* (|), *negation* (¬)
  - x ≡ y by (¬x | y) & (x | ¬y)
  - *x* ⊕ *y* by (¬x | ¬y) & (x | y)
  - x → y by (¬x | y)
- Push *negation* as low as possible:
  - ¬(x | y) by (¬x & ¬y)
  - ¬(x & y) by (¬x | ¬y)
  - ¬¬x by x
- Distribute *or* over *and*
  - x | (y & z) by (x | y) & (x | z)
  - (y & z) | x by (y | x) & (z | x)

Techniques for Backtrack Search

- Conflict analysis
  - Clause/implicate recording
  - Non-chronological backtracking
- Incorporate and extend ideas from:
  - Resolution
- Formula simplification & Clause inference
- Randomization & Restarts

Clause Recording

- During backtrack search, for each conflict create clause that explains and prevents recurrence of same conflict

$$\varphi = (a \vee b)(\neg b \vee c \vee d)\ (\neg b \vee e)(\neg d \vee \neg e \vee f)\ldots$$

Assume (decisions) $c = 0$ and $f = 0$

Assign $a = 0$ and imply assignments

A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfiable

$(a = 0) \wedge (c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$

$(\varphi = 1) \Rightarrow (a = 1) \vee (c = 1) \vee (f = 1)$

$\therefore$ create new clause: $(a \vee c \vee f)$

Clause Recording

- Clauses derived from conflicts can also be viewed as the result of applying selective resolution

$$\varphi = (a \vee b)(\neg b \vee c \vee d)\ (\neg b \vee e)(\neg d \vee \neg e \vee f)\ldots$$

resolution

$(a \vee c \vee d)$ $\qquad$ $(a \vee e)$

$(a \vee c \vee \neg e \vee f)$

$(a \vee c \vee f)$

Unit clause $(a \vee c \vee f)$ conflict and implies assignment $a = 1$!

Clause $(a \vee c \vee f)$ would have prevented the conflict

Non-Chronological Backtracking

- During backtrack search, in the presence of conflicts, backtrack to one of the causes of the conflict

$$\varphi = (a \vee b)(\neg b \vee c \vee d)\,(\neg b \vee e)(\neg d \vee \neg e \vee f)$$
$$(a \vee c \vee f)(\neg a \vee g)(\neg g \vee b)(\neg h \vee j)(\neg i \vee k)\ldots$$

Assume (decisions) $c = 0$, $f = 0$, $h = 0$ and $i = 0$

Assignment $a = 0$ caused conflict $\Rightarrow$ clause $(a \vee c \vee f)$ created
$(a \vee c \vee f)$ implies $a = 1$

A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsat

$(a = 1) \wedge (c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$

$(\varphi = 1) \Rightarrow (a = 0) \vee (c = 1) \vee (f = 1)$

$\therefore$ create new clause: $(\neg a \vee c \vee f)$

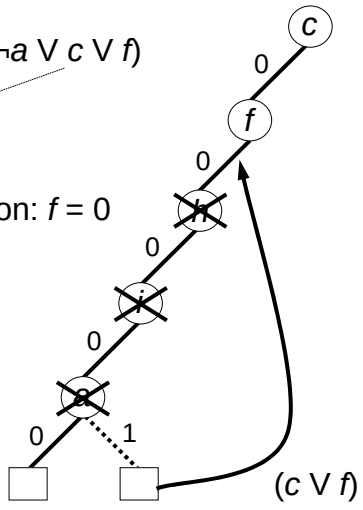
Non-Chronological Backtracking

Created clauses: $(a \vee c \vee f)$ and $(\neg a \vee c \vee f)$

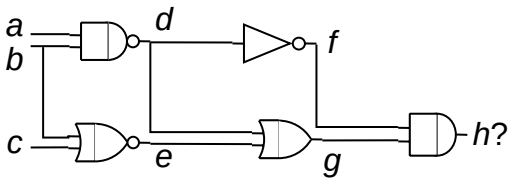
Apply resolution:
new unsat clause $(c \vee f)$

$\therefore$ backtrack to most recent decision: $f = 0$

$\therefore$ created clauses/implicates:
$(a \vee c \vee f)$,
$(\neg a \vee c \vee f)$, and
$(c \vee f)$

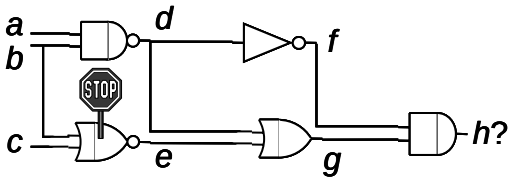


Circuit Satisfiability



$\varphi = [d \equiv \neg(ab)]\ [e \equiv \neg(b \vee c)]\ [f \equiv \neg d]\ [g \equiv d \vee e]\ [h \equiv fg]\ h$

Circuit Satisfiability



$\varphi = h\ [d \equiv \neg(ab)]\ [e \equiv \neg(b \vee c)]\ [f \equiv \neg d]\ [g \equiv d \vee e]\ [h \equiv fg]$

$= h$

$(a \vee d)(b \vee d)(\neg a \vee \neg b \vee \neg d)$

$(\neg b \vee \neg e)(\neg c \vee \neg e)(b \vee c \vee e)$

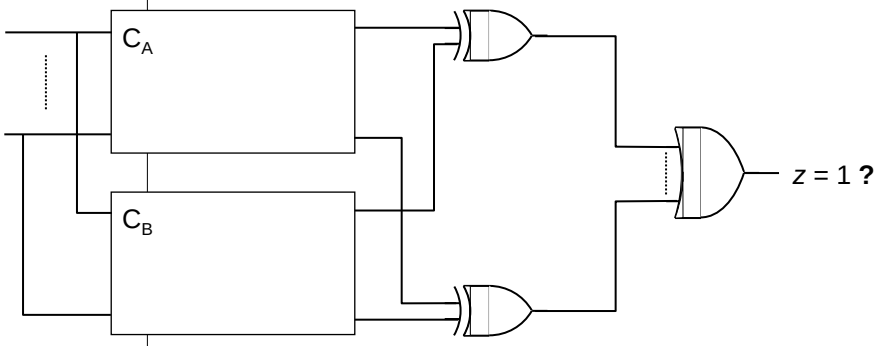
$(\neg d \vee \neg f)(d \vee f)$

$(\neg d \vee g)(\neg e \vee g)(d \vee e \vee \neg g)$

$(f \vee \neg h)(g \vee \neg h)(\neg f \vee \neg g \vee h)$

Equivalence Checking

- Combinational Circuits:



If $z = 1$ is unsatisfiable, the two circuits are equivalent **!**

SAT Problem Hardness in EDA

- Bounded Model Checking (BMC)
- Superscalar processor verification
- FPGA routing
- Equivalence Checking (CEC)
- Circuit Delay Computation
- Test Pattern Generation (ATPG):
  - Stuck-at, Delay faults, etc.
  - Redundancy Removal
- Noise analysis
- **...**

Hardest

Easiest

Unknown

Conclusions

- Many recent SAT algorithms and (EDA) applications
- Hard Applications
  - Bounded Model Checking
  - Combinational Equivalence Checking
  - Superscalar processor verification
  - FPGA routing
- “Easy” Applications
  - Test Pattern Generation: Stuck-at, Delay faults, etc.
  - Redundancy Removal
  - Circuit Delay Computation
- Other Applications
  - Noise analysis, etc.

Conclusions

- Complete vs. Incomplete algorithms
  - Backtrack search (DP)
  - Resolution (original DP)
  - Recursive learning
  - Local search
- Techniques for backtrack search (infer implicates)
  - conflict-induced clause recording
  - non-chronological backtracking
  - resolution, SM and RL within backtrack search
  - formula simplification & clause inference conditions
  - randomization & restarts

Research Directions

- Algorithms:
  - Explore relation between different techniques
    - backtrack search; conflict analysis; recursive learning; branch-merge rule; randomization & restarts; clause inference; local search (?); BDDs (?)
  - Address specific solvers (circuits, incremental, etc.)
  - Develop visualization aids for helping to better understand problem hardness
- Applications:
  - Industry has applied SAT solvers to different applications
  - SAT research requires challenging and representative publicly available benchmark instances **!**