Automated Theorem Proving
– *Foundations of SAT-Solving* –

September 2019

Wolfgang Küchlin

Symbolic Computation Group
Wilhelm-Schickard-Institute of Informatics
Faculty of Mathematics and Sciences

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

SR

Contents

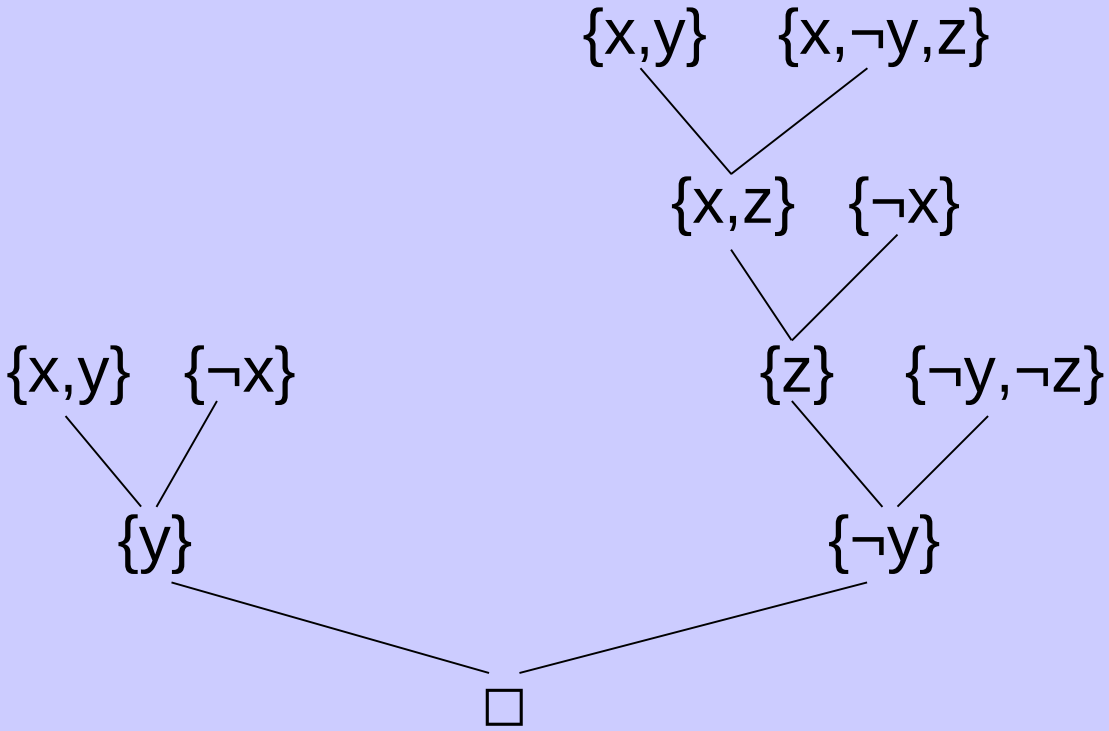
- Propositional Resolution (cut elimination)
  - $(x \vee \neg y), (y \vee z) \vdash (x \vee z)$
- Short History of SAT Solving
  - Davis-Putnam (1960): Variable Elimination
    - Eliminate complementary literals by computing resolvents
  - Davis-Logemann-Loveland (1962): Search for model
    - Buy space, pay with time
  - J. P. Marques-Silva, K. A. Sakallah (1996): CDCL
    - Combine DPLL search with controlled resolution when search fails
- Important add-ons
  - Explain UNSAT by resolution, SAT by prime implicant cover
  - Optimization: Maximum satisfiability MaxSAT

Contents

- Propositional Resolution
  - Theorem Proving by Deduction, based on *clauses*
  - Clause: disjunction of literals, e.g. $(x \vee \neg y \vee z) \equiv \{x, \neg y, z\}$
  - From 2 clauses $C = (A \vee x)$ and $D = (B \vee \neg x)$, deduce new resolvent clause $R = (A \vee B)$ where
  - $C \wedge D \models R$, hence $\{C, D\} \equiv \{C, D, R\}$
  - Prove UNSAT($\mathcal{F}$) by deducing *empty clause* $\{\ \} =: \square$
- Clause C *subsumes* clause D iff $C \subseteq D$.
  - Lemma: Subsumed clauses are redundant and can be cancelled
    - If $C \subseteq D$ then $C \models D$ and $\mathcal{F} \cup C \equiv \mathcal{F} \cup C \cup D$
- In *Unit Resolution* one of the parent clauses is a *Unit* (singleton)
  - For parent clauses $(A \vee x)$ and $(\neg x)$, the resolvent is (A), and $A \subseteq (A \vee x)$
  - Since $A \subseteq (A \vee x)$ we have $A \models (A \vee x)$. Therefore replace $(A \vee x)$ by A.

Example: Resolution proof-tree

F3={{x,y}, {x,¬y,z}, {¬x}, {¬y,¬z}}



Resolution proof procedure

- Let C be a set of axioms (constraints). In order to prove a theorem D, i.e. C ⊨ D, proceed as follows:

1. Negate D. Convert C ∪ ¬D into CNF. Call the result F.
2. Repeat
   i. Compute all non-tautological resolvents R from F
   ii. If □∈ R, return „proof“.
   iii. Delete from R all subsumed clauses. // forward subsumption
   iv. If R = { }, return „disproof“. // □ not found, no more deductions possible
   v. Delete from F all clauses subsumed by R. // backward subsumption
   vi. F := F ∪ R

- Theorem [refutation completeness]: If C ⊨ D, the proc. stops in (ii)
- Otherwise the procedure stops in (iv), because there are at most 3^n subsumption-free clauses in n variables (x, ¬x, don´t care x).
- Bad news: far too many useless deductions, memory explodes.

SAT as Existential Quantifier Elimination (EQE)

- SAT as an EQE problem:
  - Definition: $\exists p.F \equiv F_{[p/\top]} \vee F_{[p/\bot]}$
  - SAT(F) *iff* $\exists x_1,\ldots,x_n:F \equiv \top$ and UNSAT(F) *iff* $\exists x_1,\ldots,x_n:F \equiv \bot$
- Development of DP and DPLL
  - W.l.o.g rearrange F into $F = (A' \vee p) \wedge (B' \vee \neg p) \wedge R$
  - Then $\exists p.F \equiv (A' \wedge R) \vee (B' \wedge R) \equiv (A' \vee B') \wedge R$
- Elimination rules
  1. If there is a **unit** clause {p}: Then $\{\ \} \in A'$, hence $\exists p.F \equiv B' \wedge R$
  2. If p is **pure** in F: Then B´ is empty and $\exists p.F \equiv A' \wedge R$
  3. If **complementary** literals p and ¬p occur: $\exists p.F \equiv (A' \vee B') \wedge R$
- DP (1960): solve $CNF(A' \vee B') \wedge R$
- DLL (1962): solve $(A' \wedge R)$ or solve $(B' \wedge R)$

History: the Davis-Putnam Algorithm (DP-1960)

- Martin Davis, Hilary Putnam: A Computing Procedure for Quantification Theory. *Journal of the ACM 7:201--215* (1960)
  - Context: proving theorems of predicate logic
    - Formula P is a contradiction *iff* there is a finite contradiction
    - Iteratively generate the Herbrand universe H_i and substitute into P
    - Each instance $P(H_i)$ is a propositional formula, then solve $SAT(P(H_i))$
- The DP-1960 algorithm consists of 3 rules
  1. One-Literal Rule (*Unit Propagation – UP*)
  2. Affirmative-Negative Rule (*Pure Literal – PL*)
  3. Elimination of conflicts (*Resolution*)
- Later, rule 3 was replaced by 3* [DPLL, 1962]

  3*. Splitting Rule (*Case distinction → SAT-Solving*)

Example: EQE with DP-1960

- First approach: Use rule 3 until all literals are pure
  - Rule 1 is special case of rule 3
- Example
  - $S_0 = \{\{x, y, z\}, \{\neg x, y, z\}, \{\neg x\}, \{z, \neg y\}\}$
  - Rule 3 (resolution on y): $S_1 = \{\{x, z\}, \{\neg x, z\}, \{\neg x\}\}$
  - Rule 3 (resolution on x): $S_2 = \{\{z\}\}$
  - Rule 2 (PL of z): $S_3 = \{\ \}$, hence consistent.
- Second approach: Prefer rule 1 over rule 2 over rule 3
  - Rule 1 (UP of ¬x): $S_1 = \{\{y, z\}, \{z, \neg y\}\}$
  - Rule 2 (PL of z): $S_2 = \{\ \}$, hence consistent
- But EQE is *not really* SAT-Solving
  - We may not get a satisfying assignment on impure literals

History: the Davis-Putnam Algorithm (DP-1960)

- Rule 1 (One-Literal Clauses)
  a) F is inconsistent, if it contains two unit clauses {p} and {¬p}
  b) Else, if F contains a unit clause {p}, then delete all clauses containing p, and delete ¬p from all clauses.

     The result F´ is inconsistent iff F is inconsistent.
  a) The case of a unit clause {¬p} is analogous to (b).

  If F´ is empty, then F is consistent.
  - All clauses were deleted, hence all are satisfied.
- Rule 2 (Affirmative-Negative Rule)
  - If an atom p appears only positively (affirmative) or only negatively, then delete all clauses containing p.
  - The result F´ is inconsistent iff F is inconsistent.
  - If F´ is empty, then F is consistent.

History: the Davis-Putnam Algorithm (DP-1960)

- Rule 3 (Elimination of Atomic Formulas)
  - If an atom p appears ***both*** positively (in clause subset A) ***and*** negatively (in clause subset B), then

    form clause-sets A´ and B´ with A = A´ ∨ p and B = B´ ∨ ¬p

    and rearrange F into F = (A´ ∨ p) ∧ (B´ ∨ ¬p) ∧ R,

    where p does not occur in A´, B´ and R.

    Now F is inconsistent *iff* F´ = (A´ ∨ B´) ∧ R is inconsistent
  - Proof:

    F is inconsistent *iff* it is inconsistent for *both* p=0 *and* p=1.

    F= A´ ∧ R for p=0, and F= B´ ∧ R for p=1,

    hence F is inconsistent *iff*

    F´ = (A´ ∧ R) ∨ (B´ ∧ R) = (A´ ∨ B´) ∧ R is inconsistent.

History: the Davis-Putnam Algorithm (DP-1960)

- Implementation of Rule 3 (Eliminating Atomic Formulas)
  - Rearrange F into F = (A´ ∨ p) ∧ (B´ ∨ ¬p) ∧ R
  - F is inconsistent iff F´ = (A´ ∨ B´) ∧ R is inconsistent
  - In short: factor out p and resolve on p. (A´ ∨ B´) consists of all resolvents between clauses in A and clauses in B.
    - Ex.: (a ∨ p) ∧ (b ∨ p) ∧ (c ∨ ¬p) ∧ (d ∨ ¬p) = [(a ∧ b) ∨ p] ∧ [(c ∧ d) ∨ ¬p] Form (a ∧ b) ∨ (c ∧ d) = (a ∨ c) ∧ (a ∨ d) ∧ (b ∨ c) ∧ (b ∨ d). These are exactly the resolvents. The parent clauses can be deleted: if (a ∧ b) is satisfied in F´, then F is satisfied by additionally setting p=0, and analogously, if (c ∧ d) is satisfied in F´, then we may set p=1 to satisfy F.
- Bad news: clauses get longer, and clause set explodes: F´ = (A´ ∨ B´) ∧ R is no longer in CNF
  - There may be n^2 resolvents of size 2k-2 if F has n clauses of size k each

Example: the Davis-Putnam Algorithm (DP-1960)

- The DP-Algorithm consists of 3 rules
  1. One-Literal (Unit Propagation – UP)
  2. Affirmative-Negative (Pure Literal – PL)
  3. Elimination of conflicts (Resolution)
- Example
  - $S_0 = \{\{x, y, z\}, \{\neg x, y, z\}, \{\neg x\}, \{z, \neg y\}\}$
  - Rule 1c (UP of ¬x): $S_1 = \{\{y, z\}, \{z, \neg y\}\}$
  - Rule 3 (resolution on y): $S_2 = \{\{z\}, \{z\}\} = \{\{z\}\}$
  - Rule 2 (PL of z}: $S_3 = \{\ \}$, hence consistent.
- Rule 3 renders DP-1960 inefficient
  - In order to eliminate p from F, *ALL* resolvents over p have to be computed. This leads to an explosion of new clauses.

The Davis-Logemann-Loveland Algorithm (DPLL 1962)

- Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem Proving. *Communications of the ACM 5:394—397* (1962).
- Rule 3* (*Splitting Rule*, replaces Rule 3)
  - If p occurs both positively and negatively in F, rearrange F into $F = (A \vee p) \wedge (B \vee \neg p) \wedge R$, where p does not occur in R.
  - F is inconsistent iff *both* $(A \wedge R)$ *and* $(B \wedge R)$ are inconsistent
  - *Proof*: F is inconsistent iff it is inconsistent for *both* p=0 *and* for p=1. Now $F= A \wedge R$ for p=0, and $F= B \wedge R$ for p=1.
- Implementation of Rule 3*
  - Set p=1 and p=0 ***one after another*** in F (e.g. as unit clauses)
  - Do not form new clauses, instead perform unit propagation.
  - Clauses are shortened, sometimes eliminated, the problem is simplified, especially through unit propagation.

History: The Davis-Logemann-Loveland Algorithm

The forms of Rule III are interchangeable; although theoretically they are equivalent, in actual applications each has certain desirable features. We used Rule III* because of the fact that Rule III can easily increase the number and the lengths of the clauses in the expression rather quickly after several applications. This is prohibitive in a computer if ones fast access storage is limited. Also, it was observed that after performing Rule III, many duplicated and thus redundant clauses were present. Some success was obtained by causing the machine to systematically eliminate the redundancy; but the problem of total length increasing rapidly still remained when more complicated problems were attempted. Also use of Rule III can seldom yield new one-literal clauses, whereas use of Rule III* often will.

In programming Rule III*, we used auxiliary tape storage. The rest of the testing for consistency is carried out using only fast access storage. When the "Splitting Rule" is used one of the two formulas resulting is placed on tape. Tape memory records are organized in the cafeterial stack-of-plates scheme: the last record written is the first to be read.

History: From Gilmore to DP to D(P)LL

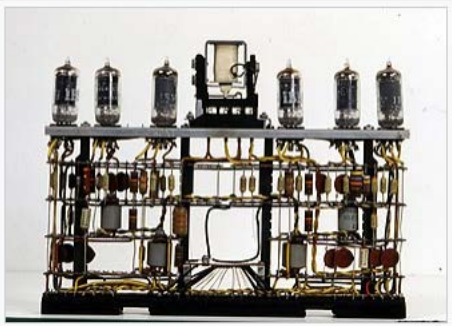
- Common motivation: Proofs in Predicate Calculus
  - (1) Enumerate the Herbrand Universe (HU) of the formula F
  - (2) Substitute each HU level of F into F and solve the resulting propositional problem P
- Gilmore (1960): Solve P by conversion to DNF
  - implemented on an IBM 704
- Davis and Putnam (1960): Solve P by variable elimination
  - No implementation: Hand computation of Gilmore´s example
  - Solved by hand in less than 30 minutes
    - Trick: Checked only levels 10, 20, 30. Inconsisteny first at level 25!
  - Gilmore´s program failed (ran out of memory?) after 21 minutes
    - Only 7 levels were tested. DNF conversion is obviously inefficient!

History: From Gilmore to DP to D(P)LL

- Davis, Logemann, Loveland (1962): Solve P by recursively solving P[x=1] and P[x=0]
  - P in conjunctive normal form
  - Implementation in „SAP“ Assembler „with many time-saving devices employed“ on IBM 704 (32K words memory = 144KB)
  - Gilmore´s example was proved in under 2 minutes!
  - $\exists x,y \forall z[F(x,y) \rightarrow (F(y,z)\&F(z,z))$
    $\& ((F(x,y)\&G(x,y)) \rightarrow (G(x,z)\&G(z,z)))]$
  - „we hoped that some mathematically meaningful and, perhaps nontrivial, theorems could be solved. The actual achevements in this direction were somewhat disappointing“.

IBM 704 (1954 – 1960) (source: wikipedia)

- The **IBM 704**, introduced by IBM in 1954, is the first mass-produced computer with floating-point arithmetic hardware. The 704 can execute up to 12,000 floating-point additions per second. Like the 701, the 704 uses vacuum tube logic circuitry and 36-bit binary words. Changes from the 701 include the use of core memory instead of Williams tubes … IBM sold 123 type 704 systems between 1955 and 1960.
- Controls are included in the 704 for: one 711 Punched Card Reader, one 716 Alphabetic Printer, one 721 Punched Card Recorder, five 727 Magnetic Tape Units and one 753 Tape Control Unit, one 733 Magnetic Drum Reader and Recorder, and one 737 Magnetic Core Storage Unit. Weight: about 19,466 pounds (8.8 t).
- The 737 Magnetic Core Storage Unit serves as RAM and provides 4,096 36-bit words, the equivalent of 18,432 bytes. The 727 Magnetic Tape Units store over five million six-bit characters per reel.



IBM 704 vacuum tube circuit module

Modern Form of the D(P)LL Algorithm of 1962

```
boolean DPLL(ClauseSet S){

    //1. Simplify S (unit constraint propagation)
    while (S contains a unit clause {ℓ}) {
        delete from S clauses containing ℓ;      // unit-subsumption
        delete ¬ℓ from all clauses in S          // unit-resolution mit subsumption
    }

    //2. Trivial case?
    if (□∈S)  return false;                      // constraint unsatisfiable
    if (S=={}) return true;                      // nothing left to satisfy

    //3. Case split and recursion
    choose a literal ℓ occurring in S;           // Heuristic (intelligence) needed!
    if( DPLL(S ∪ {ℓ}) ) return true;             // first recursive branch: try ℓ := true
    else if ( DPLL(S ∪ {¬ℓ}) ) return true;      // backtracking: try ℓ := false
    else return false;
}
```

Observations for DPLL in practice

- No deduction of new clauses
  - No dynamic storage allocation
- Algorithm lives (and dies) with unit propagation
  - UP dominates run-time in practice (> 90% UP).
  - This is necessarily so:
    - With 100 variables there are 2^{100} cases without UP
    - With complete UP there are only 99 propagations
    - Typical value in practice: 90 propagations, 2^{10} remaining cases
- Lesson from practice (and secret behind DPLL)
  - Only few decisions are essential, the remaining cases follow as immediate consequences, ruling out many theoretical alternatives.

Example: SAT-Solving with DPLL

- $S_0 = \{\{x, y, z\}, \{\neg x, y, z\}, \{\neg x\}, \{z, \neg y\}\}$
  - unit propagation of $\neg x$
    - $\{\neg x\}$ subsumes $\{\neg x, y, z\}$, hence $\neg x \wedge (\neg x \vee y \vee z) \equiv \neg x$
    - $\{\neg x\}$ unit-resolves with $\{x, y, z\}$ to $\{y, z\}$, and $\{y, z\}$ subsumes $\{x, y, z\}$
- $S_1 = \{\{y, z\}, \{z, \neg y\}\}$
  - Heuristically choose y as decision variable:
  - Case 1: let y=1
  - $S_2 = \{\{y\}, \{y, z\}, \{z, \neg y\}\}$
  - unit propagation of y yields $S_3 = \{\{z\}\}$,
  - unit propagation of z yields $S_4 = \{\ \}$, return true.

Variable Selection Heuristics for DPLL

- Some Heuristics for variable selection:
  - Coose the literal which occurs most often.
    - Then the formula is simplified in the most places
  - Choose a literal L from 2-clause (binary clause) {K, ¬L}.
    - Then K=1 if L=1, resp. L=0 if K=0, because clause encodes (L → K)
    - Clever choice of K resp. L immediately leads to UP, eliminating a decision
  - Choose a literal from a short(est) clause.
    - Will soon produce a binary clause, then a UP
  - For each literal L, compute how F would be shortened by UP after assigning L. Choose that L which has the greatest effect.
    - This simplifies F most before the next decision.

Principle of Conflict Driven Clause Learning (CDCL)

- Learning to avoid a bad sequence of decisions
  - A sequence of decisions and propagations may hit a root F=0.
  - But not all of these decisions may be relevant for the root.
- Key insight: start learning process with *conflict clause* K
  - Conflict clause (failure clause) K is the clause which becomes empty in Step 2 of DPLL, i.e. $\beta(K)=0$.
  - The failure is caused by all literals in K becoming 0. This set is relatively small.
  - Now we need to find the subset of decisions, whose conjunction D caused all these literals to become 0.
  - Negating this conjunction gives us a clause L = ¬D which is implied by F, hence can be added to F.
    - ¬L (=D) implies ¬F, so $\models(L \vee \neg F)$, i.e. F implies L

Example: Principle of Learning in CDCL

- $S_0 = \{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$. We made the assignments:
  - x=0 (Decision), y=1 (Unit Propagation), z=1 (Unit Propagation)
  - Conflict clause is K={¬z, x}, Reason for conflict is R = {¬y, z}
  - Resolvent on conflict literal z (*first learnt clause*) is $L_1 = \{x, \neg y\}$
  - Notice that L_1 is false under current assignment. It contains both a decision variable x and a unit propagation variable y. After backtracking, L_1={x, ¬y} is not unit and not immediately useful.
  - We can get rid of ¬y by resolving with *its* reason {x, y}. So L_2={x}
  - Now if we backtrack to before the assignment on x, there is no decision left: x=1 now becomes a unit propagation of {x}.
  - In general we continue learning clauses until we hit the first „UIP clause“ (*unique implication point*): It contains a single variable on the highest level of assignment. After backtracking, it is unit.

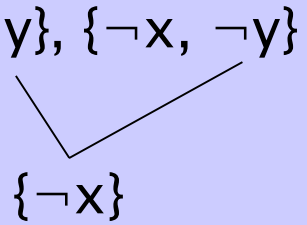
CDCL based proof learning and UNSAT explanation

- Clause set S_0 = { {x, y}, {x, ¬y}, {¬x, y}, {¬x, ¬y} }
  - S_0 [x=1] = { {1, y}, {1, ¬y}, {0, y}, {0, ¬y} } : choose UP y=1
  - S_0 [x=1, y=1] = { {1, 1}, {1, 0}, {0, 1}, {0, 0} } = 0 = conflict!
  - A decision (x=1) forced conflicting propagations y=1 and y=0, obviously by 2 clauses containing {.., y, ..} and {.., ¬y, ..}
  - Hence there is a resolvent on y, in this case {¬x, y}, {¬x, ¬y} ⊢ {¬x}.
  - Add {¬x} to C, because it is a logical consequence of C.
  - Backtrack to just before the decision on x (no matter how far!). Now x=0 is a forced unit propagation by {¬x} (no more decision)
  - S_1 = { {¬x},{x, y}, {x, ¬y}, {¬x, y}, {¬x, ¬y} } : propagate x=0
  - S_1[x=0] = { {1}, {0, y}, {0, ¬y}, {1, y}, {1, ¬y} } : choose UP y=0
  - S_1[x=0, y=1] = { {1}, {0, 0}, {0, 1}, {1, 1}, {1, 0} } = conflict !
  - Hence there is a resolvent on y, in this case {x, y}, {x, ¬y} ⊢ {x}, add {x} to S_1
  - Without any decision on x, we have a final conflict in S_2 = {{¬x}, …, {x}} ⊢ □

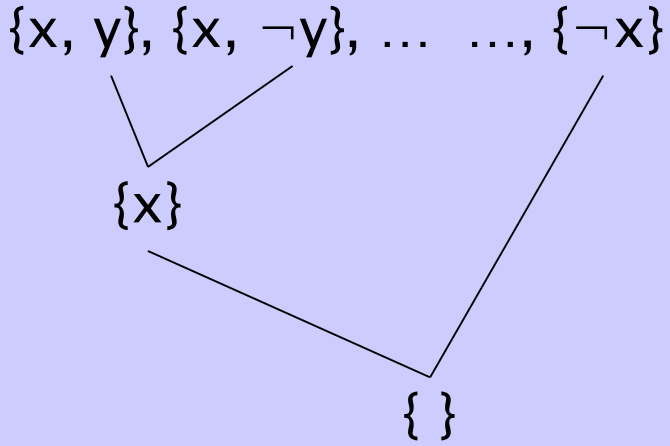
Resolution proof explaining UNSAT(C)

$S_0 = \{ \{x, y\}, \{x, \neg y\}, \{\neg x, y\}, \{\neg x, \neg y\} \}$

Resolution Proof of UNSAT($S_0[x=1]$), respectively of $S_0 \models \{\neg x\}$:



Final proof of UNSAT(S_0):



The answer is easy if you take it logically (Paul Simon)

Note on UnSAT Cores

- Usually, only a (small) subset of S is unsatisfiable
- This is called an UnSAT Core
  - S may contain very many different UnSAT cores
  - The clauses used in the final resolution proof of UnSAT produced by the solver contains an UnSAT Core
- An UnSAT Core is minimal if it becomes SAT after any clause is removed (all clauses are necessary)
  - In practice, cores produced by solvers are close to minimal
  - can be easily reduced to minimal by trial and error
- UnSAT Cores are essential to explain UnSAT
  - If UnSAT(S) points to a defect in S, explanation is essential
  - Result is useless without explanation

Note on Implicants and Prime Implicants

- $S_0 = \{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$
- $SAT(S_0) = \{x=1, z=1, y=1\}$ yields an *implicant*, $x \wedge y \wedge z \models S_0$
  - but y is unnecessary: $x \wedge z$ is also an implicant; y is „don´t care“
- A Prime Implicant is an implicant which cannot be shortened (minimal implicant)
  - there may be many prime implicants contained in a model
  - different models may/will contain different prime implicants
- Extracting Prime Implicants from the final clause state
  - SAT solver always computes complete models
  - remove any unnecessary literal (all clauses must remain true)
    - Now some literals may become necessary which were not before
  - Now iterate the process as long as possible.

Note on Maximum Satisfiability

- MaxSAT(F): the maximum number of satisfiable clauses
- Key ingredients
  - add a **blocking variable** to each clause C by $\neg b_C \rightarrow C$, i.e. $b_C \vee C$
    - solver can „block“ (switch off) constraint C by setting b_C = true
  - representing an **arithmetic constraint** in CNF: CNF($\Sigma(x_i)<k$)
    - direct combinatorial encoding or boolean representation of arithmetic circuits
- From NP complete decision problem to optimization
  - If F has n clauses, let k = n
    - while SAT($F \wedge$ CNF($\Sigma(b_i)<k$) let k = k-1
    - return n-k // k clauses must be blocked, n-k can be satisfied
    - Note: learned clauses kann be retained between iterations
  - Further optimization e.g. binary search

Summary: Decision Procedures for Propositional Logic

- Davis-Putnam 1960 (*variable elimination rules 2 and 3*)
- DP 1960 (Rule 2): Let F contain a pure literal p.
  - Then F ≡ (A ∨ p) ∧ R (where p not in A, R)
  - Hence F ≗ R (p is eliminated from F)
- DP 1960 (Rule 3) : Let F contain a complementary literal p
  - Then F ≡ (A ∨ p) ∧ (B ∨ ¬p) ∧ R (where p not in A, B, R)
  - Hence F ≗ (A ∨ B) ∧ R (p is eliminated from (A ∨ B) ∧ R)
  - F increases in size, but is strictly „simpler“ because p is eliminated.
- Davis-Logemann-Loveland 1962 *(splitting rule)*:
  - F is inconsistent *iff* both (A ∧ R) and (B ∧ R) are inconsistent
    - No fresh CNF conversion necessary, split in 2 smaller formulas, divide&conquer
- SAT-Solving (optimization of *splitting rule*):
  - F inconsistent iff both $F|_{p=0}$ and $F|_{p=1}$ inconsistent
- Resolution: (A ∨ p)∧(B ∨ ¬p)∧R ≡ (A ∨ p)∧(B ∨ ¬p)∧(A ∨ B)∧R

Literature

1. Paul C. Gilmore. A proof method for quantification theory. *IBM J. Research and Development* **4** (1960), 28—35.
2. M. Davis and H. Putnam. A computing procedure for quantification theory. *J.ACM* 7(3), 1960
3. M. Davis, G. Logemann and D. Loveland. A machine program for theorem proving. *C.ACM* 5, 1962
4. A. Biere, M. Heule, H. van Maaren, T. Walsh (eds.). *Handbok of Satisfiability*. IOS Press 2009. (Comprehensive current account of SAT based methods)
5. J. P. Marques-Silva. *Search Algorithms for Satisfiability Problems in Combinatorial Switching Circuits*. PhD Thesis, U. Michigan, 1995
6. J. P. Marques-Silva, K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In: *Intl. Conf. Computer Aided Design*., Nov 1996.
7. J. P. Marques-Silva, K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. In: *IEEE Transactions on Computers*., May 1999.
8. D. E. Knuth. Satisfiability. *The Art of Computer Programming* Vol 4 Fasc. 6. Addison Wesley, 2016