Algorithms for Computational Logic

Summary

Contents

Chapter 1

SAT and Modeling with SAT

1.1 Cardinality Constraints

In order to handle cardinality constraints we have two options: encode the cardinality constraints to CNF and use a SAT solver, or use a pseudo boolean (PB) solver.

1.1.1 AtMost1

- $\sum_{j=1}^{n} x_j = 1$ can be encoded with $\left(\sum_{j=1}^{n} x_j \leq 1\right) \wedge \left(\sum_{j=1}^{n} x_j \geq 1\right)$
- $\sum_{j=1}^{n} x_j \geq 1$ can be encoded with $(x_1 \vee x_2 \vee ... \vee x_n)$
- $\sum_{j=1}^{n} x_j \leq 1$ can be encoded with:
  - Pairwise encoding
  - Sequential counter encoding
  - Bitwise encoding

Sequential Counter

In order to realize this encoding, we need to add new variables s_i for the fact "there is a 1 on some position $1..i$":

$$s_i \text{ is true if } \sum_{j=1}^{i} x_j \geq 1$$

Encoding $\sum_{j=1}^{n} x_j \leq 1$ with sequential counter:

$$\begin{aligned}
&(\neg x_1 \vee s_1) \wedge \\
&(\neg x_i \vee s_i), i \in 2..n-1 \wedge \\
&(\neg s_{i-1} \vee s_i), i \in 2..n-1 \wedge \\
&(\neg x_i \vee \neg s_{i-1}), i \in 2..n
\end{aligned}$$

If $x_j = 1$, then all s_i variables are assigned and all other x variables must take value 0. There are $\mathcal{O}(n)$ clauses and $\mathcal{O}(n)$ auxiliary variables.

Bitwise Encoding

In bitwise encoding, we represent the constraint $\sum_{j=1}^{n} x_j \leq 1$ by encoding the index of the potential true variable in binary. For this, we add new auxiliary variables:

$$v_0, ...v_r - 1;\ r = \lceil \log n \rceil (\text{with } n > 1)$$

Each variable x_j is assigned a unique binary number that represents its index. Then, for each variable x_j with binary index representation i, we create clauses that enforce the condition:

- If $x_j = 1$, assignment to v_i variables must encode $j - 1$, and all other x variables must take value 0
- If all $x_j = 0$, any assignment to v_i variables is consistent

For example, $x_1 + x_2 + x_3 \leq 1$:

	$j-1$	$v_1 v_0$
x_1	0	00
x_2	1	01
x_3	2	10

$$(\neg x_1 \vee \neg v_1) \wedge (\neg x_1 \vee \neg v_0)$$
$$(\neg x_2 \vee \neg v_1) \wedge (\neg x_2 \vee v_0)$$
$$(\neg x_3 \vee v_1) \wedge (\neg x_3 \vee \neg v_0)$$

There are $\mathcal{O}(n \log n)$ clauses and $\mathcal{O}(\log n)$ auxiliary variables

1.1.2 General Cardinality Constraints

Constraints of the form $\sum_{j=1}^{n} x_j \leq k$ or $\sum_{j=1}^{n} x_j \geq k$ can be added with:

- Sequential Counters
- BDDs
- Sorting Networks
- Cardinality Networks
- Totalizer

Sequential Counter Encoding

For each variable x_i, create k additional variables $s_{i,j}$ that are used as counters:

- $s_{i,j} = 1$ if at least j variables $\{x_1 ... x_i\}$ are assigned value 1
- $s_{i,j} = 0$ if at most $j - 1$ variables $\{x_1 ... x_i\}$ are assigned value 1

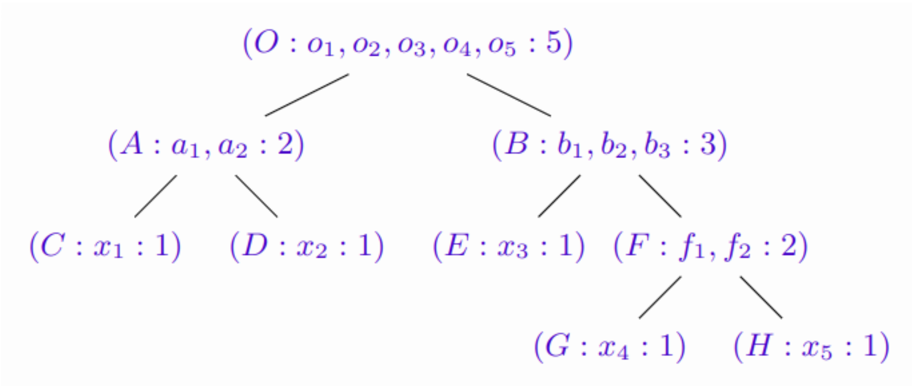
Encoding:

$$
\begin{array}{ll}
(\neg x_1 \vee s_{1,1}) & \\
(\neg s_{1,j}), & \forall j : 1 < j \leq k \\
 & \\
(\neg x_i \vee s_{i,1}), & \forall i : 1 < i < n \\
(\neg s_{i-1,1} \vee s_{i,1}), & \forall i : 1 < i < n \\
 & \\
(\neg s_{i-1,j} \vee s_{i,j}) & \forall i,j : 1 < i < n, 1 < j \leq k \\
(\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j}) & \forall i,j : 1 < i < n, 1 < j \leq k \\
 & \\
(\neg x_i \vee \neg s_{i-1,k}) & \forall i : 1 < i \leq n
\end{array}
$$

Totalizer Encoding

In this encoding we count in unary how many of the n variables $(x_1 ... x_n)$ are assigned to 1. It can be visualized as a tree:

- Each node is $(name : variable : sum)$
- Root node has the output variables $(o_1 ... o_n)$ that count how many variables are assigned to 1
- Literals are at the leaves
- Each node counts in unary how many leaves are assigned to 1 in its subtree
- Example: if $b_2 = 1$, then at least 2 of the leaves (x_3, x_4, x_5) are assigned to 1



To encode $x_1 + x_2 + x_3 + x_4 + x_5 \leq 3$ just set $o_4 = 0$ and $o_5 = 0$.
Encoding:

$$
\bigwedge_{\substack{0 \leq \alpha \leq n_2 \\ 0 \leq \beta \leq n_3 \\ 0 \leq \sigma \leq n_1 \\ \alpha + \beta = \sigma}} \neg q_\alpha \vee \neg r_\beta \vee p_\sigma \quad \text{where, } p_0 = q_0 = r_0 = 1
$$

There are $\mathcal{O}(n \log n)$ new variables and $\mathcal{O}(n^2)$ new clauses

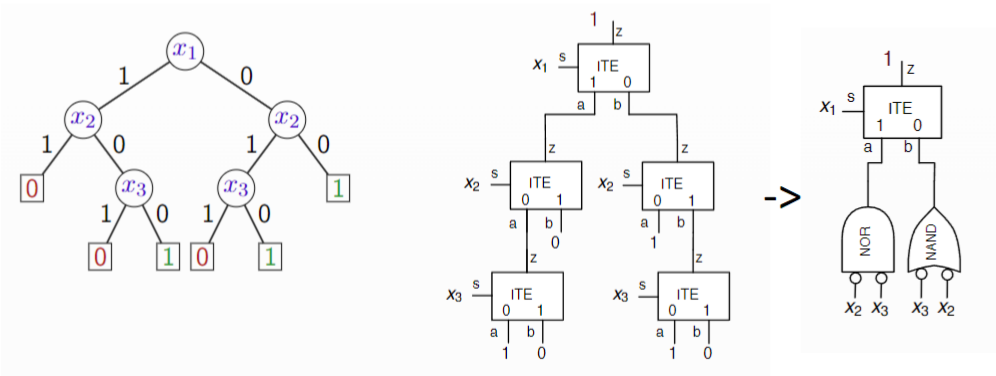
1.2 Pseudo-Boolean Constraints

The general form of these constraints is $\sum_{j=1}^{n} a_j x_j \leq b$

1.2.1 Encodings

BDD Encoding

BDDs can be used to encode pseudo-boolean constraints. For example, to encode $3x_1 + 3x_2 + x_3 \leq 3$, we can construct the following BDD and extract its ITE-based circuit:



Sequential Weighted Counter Encoding

Assuming the general form $\sum_{i=1}^{n} w_i x_i \leq k$, where the weights are all non-negative:

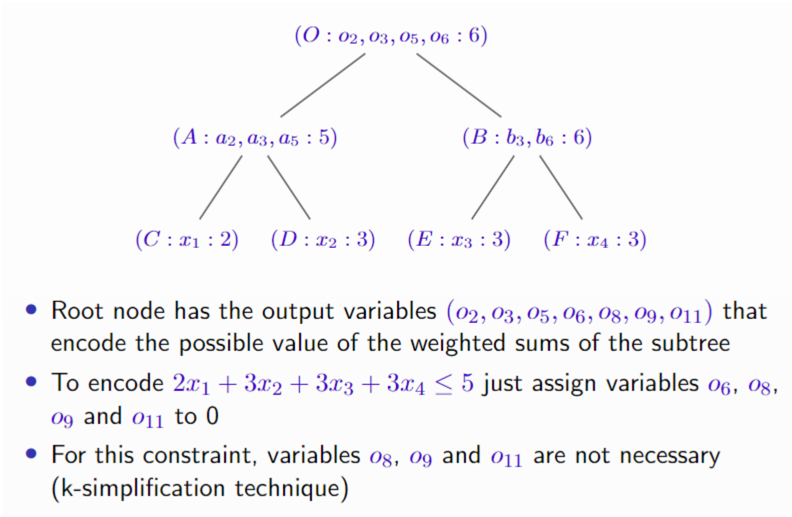
- For each variable x_i, create k additional variables $s_{i,j}$ that are used as counters
  - $s_{i,j} = 1$ if the weighted sum of the first i variables $\{x_1 ... x_i\}$ is at least j
  - $s_{i,j} = 0$ if the weighted sum of the first i variables $\{x_1 ... x_i\}$ is at most $j - 1$

Encoding:

$$
\begin{array}{lr}
(\neg x_1 \vee s_{1,j}) & \forall j : 1 \leq j \leq w_1 \\
(\neg s_{1,j}), & \forall j : w_1 < j \leq k \\
\\
(\neg x_i \vee s_{i,j}), & \forall i, j : 1 < i < n, 1 \leq j \leq w_i \\
\\
(\neg s_{i-1,j} \vee s_{i,j}) & \forall i, j : 1 < i < n, 1 \leq j \leq k \\
(\neg x_i \vee \neg s_{i-1,j} \vee s_{i,j+w_i}) & \forall i, j : 1 < i < n, 1 \leq j \leq k - w_i \\
\\
(\neg x_i \vee \neg s_{i-1,k+1-w_i}) & \forall i : 1 < i \leq n
\end{array}
$$

Generalized Totalizer Encoding

The goal of GTE is to account for the possible values of the left-hand side. It only considers the possible sums generated from the weights in the constraint. For example, in $2x_1 + 3x_2 + 3x_3 + 3x_4 \leq 5$ it is not possible for the weighted sum to have value 1, 4 or 7.



1.2.2 Pseudo-Boolean Optimization

Suppose we must minimize $\sum_{j=1}^{n} c_j x_j$ subject to $\sum_{j=1}^{n} a_{ij} x_j \leq b_i \;\; \forall i \in \{1, ..., m\}, \forall j \in \{1, ..., n\}, x_j \in \{0, 1\}$. To translate this to MaxSAT we should:

- Encode each pseudo-Boolean constraint into CNF. All clauses used in the encoding are hard
- For each term $c_j x_j$ in the objective function, add a soft clause $(\neg x_j)$ with weight c_j

There are several ways of solving the optimization problem:

- Translate into MaxSAT and use a Weighted MaxSAT algorithm
- Iterative Pseudo-Boolean solving
- Core-guided Pseudo-Boolean solving
- Branch-and-Bound Search

Given a PBO problem instance, where variables have an integer domain, the Linear Programming Relaxation is the corresponding linear program where the variable's integer constraints are relaxed

PBO:

Minimize $\sum_{j=1}^{n} c_j x_j$

Subject to $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$

$x_j \in \{0, 1\}$

LPR:

Minimize $\sum_{j=1}^{n} c_j x_j$

$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$

$x_j \in [0, 1]$

Linear Programming Relaxation

LPR is relevant since if can be solved quickly and if he relaxed linear program returns an optimal solution where all variables have integer value, then the solution of the relaxed linear program is also the optimal solution of the PBO problem. If the solution of the relaxed linear program is not integer for some variable, it still provides a lower bound on the optimal value of the PBO optimal solution

Branch-and-Bound Search

In the branch and bound algorithm we search by recursively dividing into smaller subproblems. It continuously uses LPR to get lower bounds (in case of minimization) and get candidade solutions.

Description of the Algorithm

1. Init: Initialize $UB = +\infty$
2. Init: Start with the original problem as the only node and mark it as active
3. LPR Solve: Select an active node k. Let z denote the value of the objective function for the optimal solution of the Linear Programming Relaxation (LPR) at node k
4. Improve Upper Bound: If the solution of the LPR is integer and $z < UB$, then let $UB = z$ and save solution
5. Split: If the LPR is feasible but optimal solution is not integer and $z < UB$, then use branching procedure to generate two new nodes and mark them as active
6. Deque: Mark node k as inactive
7. Repeat: If there are active nodes, go back to **3**. Otherwise, the algorithm ends and the optimal solution is the last one saved

1.2.3 Cutting Planes

Cutting planes are used to further prune the space. Can be used to combine two constraints:

$$\frac{\begin{array}{c}\delta(\sum_{j=1}^{n} a_j x_j \leq b) \\ \delta'(\sum_{j=1}^{n} a'_j x_j \leq b')\end{array}}{\delta \sum_{j=1}^{n} a_j x_j + \delta' \sum_{j=1}^{n} a'_j x_j \leq \delta b + \delta' b'}$$

For example:

$$\frac{\begin{array}{rl}1(x_4 + 3x_5 + 2x_3 & \leq 3) \\ 2(x_1 + x_2 + \neg x_3 & \leq 1)\end{array}}{2x_1 + 2x_2 + x_4 + 3x_5 \leq 3}$$

- $\neg x_3$ is replaced with $1 - x_3$
- Notice that x_3 does not occur in the new constraint
- The cutting plane operation in Pseudo-Boolean solving corresponds to the CNF clause resolution

Rounding can also be applied:

$$\frac{\sum_{j=1}^{n} a_j x_j \leq b}{\sum_{j=1}^{n} \lfloor a_j \rfloor x_j \leq \lfloor b \rfloor}$$

- The correctness of the rounding operation follows from $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$
- Hence, δ coefficients in cutting plane operations do not need to be integer. Rounding can be safely applied afterwards

For example:

$$\frac{0.5(3x_1 + 2x_2 + x_3 + 2x_4 + x_5 \quad \leq 5)}{1.5x_1 + x_2 + 0.5x_3 + x_4 + 0.5x_5 \leq 2.5}$$

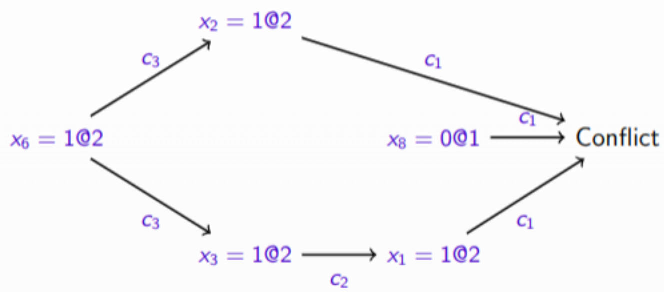
After rounding: $x_1 + x_2 + x_4 \leq 2$

And backtracking can also be applied:

$$\begin{aligned} 3x_1 + x_2 + x_7 + 2\neg x_8 &\leq 5 \\ 3\neg x_1 + x_3 + 2x_7 + x_9 &\leq 3 \\ \neg x_2 + \neg x_3 + x_6 &\leq 1 \end{aligned}$$

- Suppose you start with assignment $x_8 = 0$ at first decision level
- Next, you decide to assign $x_6 = 1$. What happens?

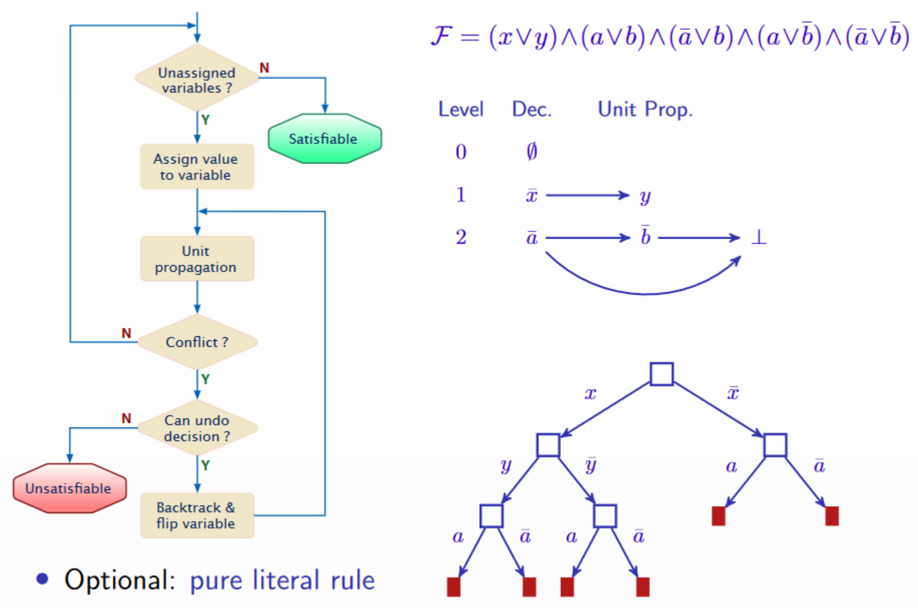
$$\begin{aligned} c_1: \quad & 3x_1 + x_2 + x_7 + 2\neg x_8 && \leq 5 \\ c_2: \quad & 3\neg x_1 + x_3 + 2x_7 + x_9 && \leq 3 \\ c_3: \quad & \neg x_2 + \neg x_3 + x_6 && \leq 1 \end{aligned}$$



Backward traversal to the decision variable x_6
Learned constraint: $x_6 + 3x_7 + 2\neg x_8 + x_9 \leq 4$
Backtrack to level 1 and imply $x_7 = 0$

1.3 SAT Algorithms

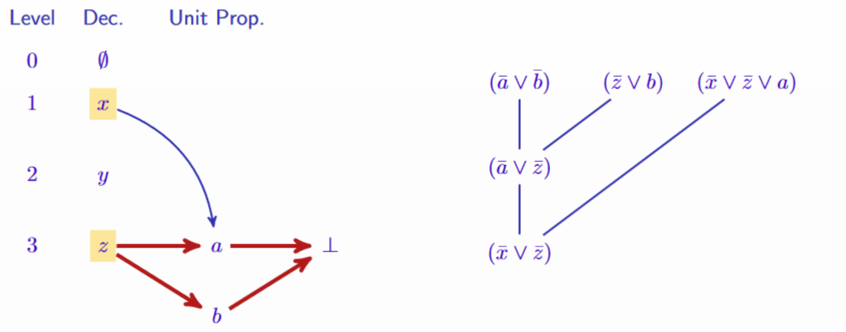
1.3.1 DPLL Solvers



1.3.2 CDCL Solvers

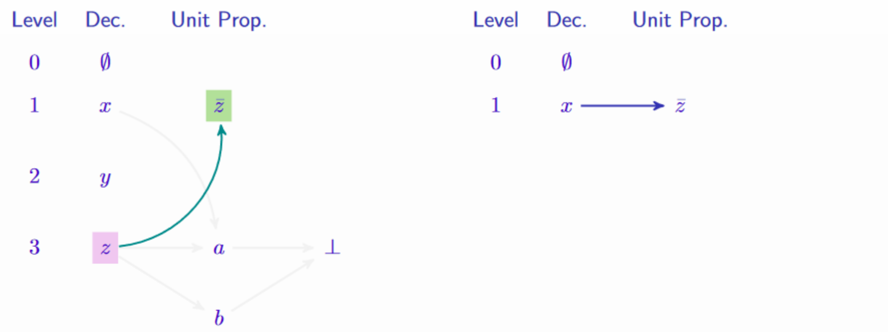
CDCL solvers extend DPLL solvers with clause learning and non-chronological backtracking, search restarts, lazy data structures, conflict-guided branching, etc.

Clause Learning



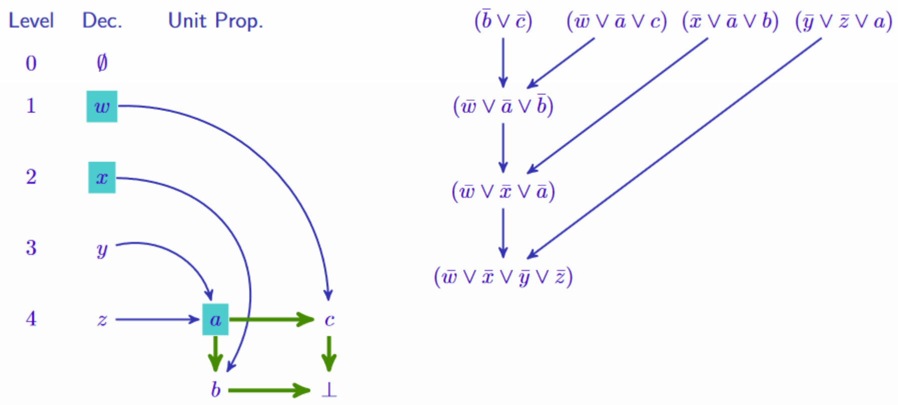
- Analyze conflict
  - Reasons: x and z
    - Decision variable & literals assigned at lower decision levels
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution
  - Learned clauses result from (**selected**) resolution operations

And after backtracking:



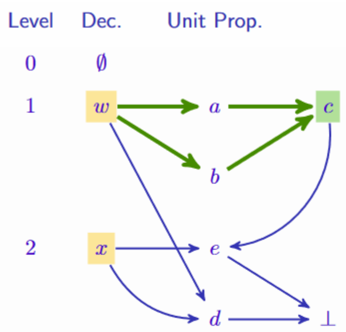
- Clause $(\bar{x} \vee \bar{z})$ is asserting at dec. lev. 1 — it forces “flipping” z
- Learned clauses are always asserting [MSS96,MSS99]

Unique Implication Points



- ~~Learn clause $(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$~~
- But a is an UIP — it implies conflict at dec. lev. 4
- Learn clause $(\bar{w} \vee \bar{x} \vee \bar{a})$

Clause Minimization



- ~~Learn clause $(\bar{w} \vee \bar{x} \vee \bar{c})$~~
- Cannot apply self-subsuming resolution
  - Resolving with reason of c yields $(\bar{w} \vee \bar{x} \vee \bar{a} \vee \bar{b})$
- Can apply recursive minimization
- Learn clause $(\bar{w} \vee \bar{x})$

- Marked nodes: literals in learned clause [SB09]
- Trace back from c until marked nodes or new decision nodes
  - Learn clause only if search ends in marked nodes

Chapter 2

Optimization problems and SAT-Based Problem Solving

A set of constraints is overconstrained if it is inconsistent. In a given an unsatisfiable formula, there may be several explanations for its unsatisfiability. The goal of MaxSAT is to find largest subset of clauses that is satisfiable.

		Hard Clauses?	
		No	Yes
Weights?	No	Plain	Partial
	Yes	Weighted	Weighted Partial

- Must satisfy hard clauses, if any
- Compute set of satisfied soft clauses with maximum cost
  - Without weights, cost of each falsified soft clause is 1
- **Or**, compute set of falsified soft clauses with minimum cost (s.t. hard & remaining soft clauses are satisfied)
- **Note**: goal is to compute **set** of satisfied (or falsified) clauses; **not** just the cost **!**

2.1 MaxSAT Algorithms

2.1.1 Fu and Malik

$\varphi_W \leftarrow \varphi$ — Join Soft and Hard Constraints

while *true* **do**

  $(\mathsf{st}, \varphi_C, \mathcal{A}) \leftarrow \mathsf{SAT}\ (\varphi_W)$

  if $\mathsf{st} = UNSAT$ **then**

    $V_R \leftarrow \emptyset$

    foreach $\omega \in \varphi_C \wedge \mathsf{Soft}\ (\omega)$ **do** — Relax all soft clauses from unsatisfiable core

      $\omega_R \leftarrow \omega \cup \{r_i\}$ // r_i is a new relaxation variable

      $\varphi_W \leftarrow \varphi_W \backslash \{\omega\} \cup \{\omega_R\}$ — Relaxed soft clause remains soft

      $V_R \leftarrow V_R \cup \{r_i\}$

    $\varphi_W \leftarrow \varphi_W \cup \mathsf{CNF}\ (\sum_{r_i \in V_R} r_i \leq 1)$ — Cardinality constraint is hard

  else

    return $\mathcal{A}$ — Return when formula becomes SAT

2.1.2 MSU3

$V_R \leftarrow \emptyset$

$\varphi_W \leftarrow \varphi$

$LB \leftarrow 0$

while *true* **do**

  $(\mathsf{st}, \varphi_C, \mathcal{A}) \leftarrow \mathsf{SAT}\ (\varphi_W \cup \mathsf{CNF}\ (\sum_{r_i \in V_R} r_i \leq LB))$

  if $\mathsf{st} = UNSAT$ **then**

    foreach $\omega_i \in \varphi_C \wedge \mathsf{Soft}\ (\omega_i)$ **do**

      $\omega_R \leftarrow \omega_i \cup \{r_i\}$ // r_i is a new relaxation variable

      $\varphi_W \leftarrow \varphi_W \backslash \{\omega_i\} \cup \{\omega_R\}$ // ω_R is a **hard** clause

      $V_R \leftarrow V_R \cup \{r_i\}$

    $LB \leftarrow LB + 1$

  else

    return $\mathcal{A}$

2.2 Minimal Unsatisfiable Subsets

Given $\mathcal{F}$ unsatisfiable, $\mathcal{M} \subseteq \mathcal{F}$ is a MUS iff $\mathcal{M}$ is unsatisfiable and $\forall_{c \in \mathcal{M}}, \mathcal{M} \backslash \{c\}$ is satisfiable.

2.2.1 Algorithms

The following algorithms may be used to identify minimal unsatisfiable subsets.

Deletion-Based

Input : Set $\mathcal{R}$
Output: Minimal subset $\mathcal{M}$
begin
  $\mathcal{M} \leftarrow \mathcal{R}$
  foreach $c \in \mathcal{M}$ **do**
    if $\neg$SAT$(\mathcal{M} \setminus \{c\})$ **then**
      $\mathcal{M} \leftarrow \mathcal{M} \setminus \{c\}$ // Remove c from $\mathcal{M}$
  return $\mathcal{M}$ // Final $\mathcal{M}$ is minimal set
end

Insertion-Based

Input : Set $\mathcal{R}$
Output: Minimal subset $\mathcal{M}$
begin
  $\mathcal{M} \leftarrow \emptyset$
  while $\mathcal{R} \neq \emptyset$ **do**
    $\mathcal{S} \leftarrow \emptyset$ // Subset of $\mathcal{R}$
    $c_r \leftarrow \emptyset$
    while SAT$(\mathcal{M} \cup \mathcal{S})$ **do**
      $c_i \leftarrow$ SelectRemoveElement$(\mathcal{R})$
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{c_i\}$
      $c_r \leftarrow c_i$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{c_r\}$ // c_r is transition element
    $\mathcal{R} \leftarrow \mathcal{S} \setminus \{c_r\}$
  return $\mathcal{M}$ // Final $\mathcal{M}$ is minimal subset
end

Dichotomic

Input : Set $\mathcal{R} = \{c_1, \ldots, c_m\}$
Output: Minimal subset $\mathcal{M}$
begin
  $\mathcal{M} \leftarrow \emptyset$
  while $\mathsf{SAT}(\mathcal{M})$ **do**
    $\mathsf{min} \leftarrow 1$
    $\mathsf{max} \leftarrow |\mathcal{R}|$
    while $\mathsf{min} \neq \mathsf{max}$ **do**
      $\mathsf{mid} = \lfloor(\mathsf{min} + \mathsf{max})/2\rfloor$ // Execute binary search
      $\mathcal{S} \leftarrow \{c_1, \ldots, c_{\mathsf{mid}}\}$ // Extract sub-sequence of $\mathcal{R}$
      if $\mathsf{SAT}(\mathcal{M} \cup \mathcal{S})$ **then**
        $\mathsf{min} \leftarrow \mathsf{mid} + 1$
      else
        $\mathsf{max} \leftarrow \mathsf{mid}$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{c_{\mathsf{min}}\}$ // c_{min} is transition element
    $\mathcal{R} \leftarrow \{c_1, \ldots, c_{\mathsf{min}-1}\}$
  return $\mathcal{M}$ // Final $\mathcal{M}$ is minimal subset
end

2.3 Minimal Correction Subsets

$\mathcal{C} \subseteq \mathcal{F}$ is an MCS iff $\mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall_{c \in \mathcal{C}}, \mathcal{F} \setminus (\mathcal{C} \setminus \{c\})$ is unsatisfiable.

2.3.1 Algorithms

The following algorithms may be used to identify minimal correction subsets.

Basic Linear Search

- Let $\mathcal{S} \subseteq \mathcal{F}$, such that $\mathcal{S} \nvDash \bot$, initially $\mathcal{S} = \emptyset$
- Let $\mathcal{C} \subseteq \mathcal{F}$, such that $\forall_{c \in \mathcal{C}}, \mathcal{S} \cup \{c\} \vDash \bot$, initially $\mathcal{C} = \emptyset$
- At each iteration, analyze one clause of $c \in \mathcal{F} \setminus (\mathcal{S} \cup \mathcal{C})$:
  - If $\mathcal{S} \cup \{c\} \vDash \bot$, then add c to $\mathcal{C}$, i.e. c is part of MCS
  - If $\mathcal{S} \cup \{c\} \nvDash \bot$, then add c to $\mathcal{C}$, i.e. c is part of MCS

There are $\mathcal{O}(m)$ calls to the oracle. An example:

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
$(x_1 \vee x_2)$	$(x_3 \vee x_4)$	$(\neg x_3 \vee \neg x_4)$	$(\neg x_1 \vee \neg x_2)$	(x_1)	(x_5)	$(\neg x_5 \vee x_6)$	(x_2)

$\mathcal{C}$	$\mathcal{S}$	c	$\mathcal{S} \cup \{c\}$	SAT($\mathcal{S} \cup \{c\}$)	Outcome
$\emptyset$	$\emptyset$	c_1	c_1	1	Update $\mathcal{S}$
$\emptyset$	c_1	c_2	c_1c_2	1	Update $\mathcal{S}$
$\emptyset$	c_1c_2	c_3	$c_1..c_3$	1	Update $\mathcal{S}$
$\emptyset$	$c_1..c_3$	c_4	$c_1..c_4$	1	Update $\mathcal{S}$
$\emptyset$	$c_1..c_4$	c_5	$c_1..c_5$	1	Update $\mathcal{S}$
$\emptyset$	$c_1..c_5$	c_6	$c_1..c_6$	1	Update $\mathcal{S}$
$\emptyset$	$c_1..c_6$	c_7	$c_1..c_7$	1	Update $\mathcal{S}$
$\emptyset$	$c_1..c_7$	c_8	$c_1..c_8$	0	Update $\mathcal{C}$

- MCS: $\{c_8\}$

Clause D

- Pick an assignment and let $\mathcal{S} \subseteq \mathcal{F}$ be the satisfied clauses and $\mathcal{U} \subseteq \mathcal{F}$ be the falsified clauses, with $\mathcal{F} = \mathcal{S} \cup \mathcal{U}$
- Repeat:
  - Create clause $D = \cup_{l \in c, c \in \mathcal{U}} l$
  - If $\mathcal{S} \cup \{D\} \vDash \bot$, then $\mathcal{U}$ is MCS: Report MCS and terminate
  - If $\mathcal{S} \cup \{D\} \nvDash \bot$, then add to $\mathcal{S}$ the satisfied clauses in $\mathcal{U}$, remove from $\mathcal{U}$ the satisfied clauses and loop

There are $\mathcal{O}(m-r)$ calls to the oracle, where r is the size of the smallest MCS. An example:

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
$(x_1 \vee x_2)$	$(x_3 \vee x_4)$	$(\neg x_3 \vee \neg x_4)$	$(\neg x_1 \vee \neg x_2)$	(x_1)	(x_5)	$(\neg x_5 \vee x_6)$	(x_2)

$\mathcal{S}$	$\mathcal{U}$	D	SAT($\mathcal{S} \cup \{D\}$)	Variables $= 1$
$\emptyset$	$\emptyset$	—	1	$\emptyset$
$c_3c_4c_7$	$c_1c_2c_5c_6c_8$	$\{x_1, \ldots, x_5\}$	1	$\{x_1, x_3\}$
$c_1..c_5c_7$	c_6c_8	$\{x_2, x_5\}$	1	$\{x_1, x_3, x_5, x_6\}$
$c_1..c_7$	c_8	$\{x_2\}$	0	—

- MCS: $\{c_8\}$

2.4 Duality Between MUSes and MCSes

- Let $\mathcal{S}$ be a finite set

- Let $\mathcal{F}$ be a set of subsets of $\mathcal{S}, \mathcal{F} \subseteq 2^{\mathcal{S}}$
- A hitting set $\mathcal{H} \subseteq \mathcal{S}$ is such that $\forall_{\mathcal{G} \in \mathcal{F}} \mathcal{H} \cap \mathcal{G} \neq \emptyset$
- $\mathcal{H}$ is (subset) minimal if none of its subsets is a hitting set of $\mathcal{F}$
- $\mathcal{H}$ is cardinality minimal (or of minimum size) if there are no hitting sets of $\mathcal{F}$ with fewer elements

For example:

$$
\begin{aligned}
\mathcal{S} &= \{1, 2, 3, 4, 5, 6, 7\} \\
\mathcal{F} &= \{\{1, 2, 3\}, \{3, 4, 5\}, \{5, 6, 7\}\} \\
\mathcal{H}_1 &= \{1, 2, 4, 6, 7\} \\
\mathcal{H}_1 &= \{2, 4, 6\} \\
\mathcal{H}_1 &= \{3, 7\}
\end{aligned}
$$

MUSes are minimal hitting sets of MCSes, and MCSes are minimal hitting sets of MUSes. En example:

c_1	c_2	c_3	c_4	c_5	c_6	c_7
(x_1)	$(\neg x_1)$	$(\neg x_2)$	$(x_2 \vee x_3)$	$(x_2 \vee \neg x_3)$	$(x_2 \vee x_4)$	$(x_2 \vee \neg x_4)$

MUS	$\{\{c_1, c_2\}, \{c_3, c_4, c_5\}, \{c_3, c_6, c_7\}\}$
MCS	$\{\{c_1, c_3\}, \{c_2, c_3\}, \{c_1, c_4, c_6\}, \{c_1, c_4, c_7\}, \{c_1, c_5, c_6\},$ $\{c_1, c_5, c_7\}, \{c_2, c_4, c_6\}, \{c_2, c_4, c_7\}, \{c_2, c_5, c_6\}, \{c_2, c_5, c_7\}\}$

2.4.1 MHS Approach for Solving MaxSAT

The MaxSAT solution is a smallest MCS, and any MCS is a hitting set of all MUSes. This duality can be used to solve MaxSAT:

1. Let $\mathcal{K}$ be a set of unsatisfiable cores (or MUSes)
2. Find a minimum hitting set $\mathcal{H}$ of the set $\mathcal{K}$ of already computed cores (or MUSes)
3. Check satisfiability of $\mathcal{F} \setminus \mathcal{H}$
   - If satisfiable, then $\mathcal{H}$ is a smallest MCS; terminate and return $\mathcal{H}$
   - Otherwise, compute core (or MUS) and add it to $\mathcal{K}$
4. Loop from 2

2.4.2 Enumeration

MCSes

Generate and block:

1. Extract MCS $\mathcal{C}$
2. Block $\mathcal{C}$, i.e. at least one clause in $\mathcal{C}$ must be satisfied
3. Loop from 1

MUSes

The process for enumerating MUSes is different since we cannot block them: preventing a clause from being added to the MUS is infeasible. The only solution is explicit set enumeration. Compute all MCSes and then all MUSes:

- Compute all MCSes using MCS enumerator
- Compute all minimal hitting sets of the MCSes

Chapter 3

Satisfiability Modulo Theories

SMT refer to the determination of whether a logical formula can be satisfied given certain constraints or theories. There are eager and lazy approaches for SMT solving.

3.1 Eager Approaches

Eager approaches encode the problem into a CNF and solve it with a SAT solver (single SAT call).

3.1.1 Finite Models with Booleans

Finding a model of finite size n can be encoded as SAT.

Log-encoding

In log-encoding, elements of the universe are represented by $k = \lceil \log_2 n \rceil$ boolean variables:

- Equality: $(a_k, ..., a_0) = (b_k, ..., b_0) \rightsquigarrow \bigwedge_{i \in 0..k}(a_i \Leftrightarrow b_i)$
- Inequality: $(a_k, ..., a_0) < (b_k, ..., b_0) \rightsquigarrow (\neg a_k \wedge b_k) \vee ((a_k \Leftrightarrow b_k) \wedge (a_{k-1}, ..., a_0) < (b_{k-1}, ..., b_0))$
- Other operations can be encoded, e.g. summation and multiplication by school method

Unary-encoding

In unary-encoding, elements are represented by n boolean variables, with $a_i \Rightarrow a_{i-1}$ for $i \in 1..n$:

- Equality: $(a_n, ..., a_0) = (b_n, ...mb_0) \rightsquigarrow \bigwedge_{i \in 0..n}(a_i \Leftrightarrow b_i)$
- Inequality: $(a_n, ..., a_0) < (b_n, ..., b_0) \rightsquigarrow \bigvee_{i \in 0..n}(\neg a_i \wedge b_i)$

Log-encoding smaller but unary-encoding tends to give better behavior in SAT solvers.

3.1.2 Small Model Properties

In some theories, satisfiability can be encoded into finding a finite model. A formula with only equality and n variables is satisfiable iff it has a model of at most size n.

This is true seince for any larger model A' we can construct A' by considering only elements used to interpret the variables in the formula. For example, $(x_1 = x_2) \vee (x_1 = x_3) \wedge (x_3 \neq x_2)$ is decided by looking for a model up to size 3.

Integer Difference Logic

Let s be the sum of absolute values of weights and n number of variables. It is sufficient to look for a model with integers in $0..(s+n)$, since any solution can be shifted to it starts at 0. We can "compress" any solution while preserving the satisfiability of the same literals.

3.1.3 Ackerman's Reduction

When working with uninterpreted functions, for each application $f(\vec{a})$ introduce a fresh variable f_a and for each pair of applications $f(\vec{a}), f(\vec{c})$ add the implication $\vec{a} = \vec{c} \Rightarrow f_{\vec{a}} = f_{\vec{c}}$. For example:

$$
\begin{aligned}
&f(a) \neq f(c) \wedge (a = c \vee f(a) = c) \\
&f_a \neq f_c \wedge (a = c \vee f_a = c) \wedge (a = c \Rightarrow f_a = f_c)
\end{aligned}
$$

We must also consider the sub-terms:

- $f(f(a)) = f(a) \wedge f(f(f(a))) \neq f(a)$
- Applications: $\{f(a), f(f(a)), f(f(f(a)))\}$
- Reduction:

$$
\begin{aligned}
& f_{f(a)} = f_a \wedge f_{f(f(a))} \neq f_a \\
\wedge \quad & a = f_a \Rightarrow f_a = f_{f(a)} \\
\wedge \quad & a = f_{f(a)} \Rightarrow f_a = f_{f(f(a))} \\
\wedge \quad & f_a = f_{f(a)} \Rightarrow f_{f(a)} = f_{f(f(a))}
\end{aligned}
$$

- Propagate $f_{f(a)} = f_a$ in last implication $\rightsquigarrow f_{f(a)} = f_{f(f(a))}$
- Transitivity of $f_{f(a)} = f_a$ and $f_{f(a)} = f_{f(f(a))} \rightsquigarrow f_a = f_{f(f(a))}$
- Contradiction $f_a = f_{f(f(a))}$ and $f_a \neq f_{f(f(a))} \rightsquigarrow$ unsatisfiable

3.2 Lazy Approaches

Lazy approaches use SAT for the boolean structure and a theory solver for conjunctions of literals (multiple SAT calls). The following is needed:

- $\mathcal{T}2\mathcal{B}$: abstracts theory formula to Boolean formula, e.g. $\mathcal{T}2\mathcal{B}((x<y) \vee \neg(z>y)) \rightsquigarrow e_{x<y} \vee \neg e_{z>y}$
- $\mathcal{B}2\mathcal{T}$: converts Boolean literal to theory literal, e.g. $\mathcal{B}2\mathcal{T}(\neg e_{z>y}) \rightsquigarrow \neg(z>y)$
- $\mathcal{T}$-SAT($\mathcal{L}$): theory solver for set of literals $\mathcal{L}$ determines whether $\mathcal{T}$-satisfiable or $\mathcal{T}$-unsatisfiable.
- If unsatisfiable, provides explanation $\mathcal{L}' \subseteq \mathcal{L}$ so that $\mathcal{L}'$ is also $\mathcal{T}$-unsatisfiable.
- SAT(α): SAT solver for formula α determines if SAT or UNSAT. If SAT, provides model as a set of true literals.

And the algorithm is as follows:

input : formula ϕ in theory $\mathcal{T}$
output: truth value

```
α ← T2B(φ)                                  // abstract input formula
while true do
    (res, τ) ← SAT(α)                       // Boolean model
    if res = false then return false
    L ← ⋃_{l∈τ} B2T(l)                      // convert to theory model
    (res, L′) ← T-SAT(L)                    // theory check
    if res = true then return true
    α ← α ∧ ⋁_{l∈L′} ¬T2B(l)                // block explanation
end
```

3.2.1 Theory Solver

The theory solver checks models from the SAT solver within the theory.

Congruence Closure

Congruence closure is used for equality:

- Divide the set of literals $\mathcal{L}$ into positive E and negative D.
- Build a set of all sub-terms S in $\mathcal{L}$.
- Build congruence closure as a partitioning of S.
  - Put each term $t \in S$ in its own partition.
  - For each $(s=t) \in E$, merge partitions of s and t.
  - For $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$ s.t. s_i is in the same partition as t_i, merge partitions of $f(s_1, \ldots, s_k)$ and $f(t_1, \ldots, t_k)$.
  - Repeat until no congruence applies.
- If there is a $s \neq t \in D$, s.t. s and t are in the same partition, return unsatisfiable otherwise satisfiable.

For example:

$$f(a, b) = a \wedge f(f(a, b), b) \neq a$$

- Congruence closure algorithm – iteratively merge equivalence classes:

$$\{\{a\}, \{b\}, \{f(a, b)\}, \{f(f(a, b), b)\}\}$$
$$\{\{a, f(a, b)\}, \{b\}, \{f(f(a, b), b)\}\}$$
$$\{\{a, f(a, b), f(f(a, b), b)\}, \{b\}\}$$

  - But $f(f(a, b), b) \neq a$.
  - Formula is unsatisfiable.

Integer Difference Logic

IDL is used on integer variables to make a conjunction of linear inequalities of the form $x_i - x_j \leq k$. The algorithm is as follows:

- Add edge between x_j and x_i with weight k, for inequality $x_i - x_j \leq k$
- Add additional source vertex x_0
- Add edge from x_0 to x_i, for each other vertex x_i
- Use Bellman-Ford algorithm to check for negative cycles
  - ▶ Negative cycle: Elimination of variables in (some) inequalities yields $0 \leq -k$, $k > 0$

- Convert all literals to positive (i.e. remove negations):
  - $\neg(x - y \leq c)$
  - $\rightsquigarrow x - y > c$
  - $\rightsquigarrow y - x < -c$
  - $\rightsquigarrow y - x \leq -c - 1$
- Note: Also possible on reals/rationals but care is needed because we cannot directly convert strict inequalities to non-strict ones.

For example:

- Example SMT formula:
$((x_4 - x_2 \leq 3) \vee (x_4 - x_3 \geq 5)) \wedge (x_4 - x_3 \leq 6) \wedge$
$(x_1 - x_2 \leq -1) \wedge (x_1 - x_3 \leq -2) \wedge (x_1 - x_4 \leq -1) \wedge (x_2 - x_1 \leq 2) \wedge$
$(x_3 - x_2 \leq -1) \wedge ((x_3 - x_4 \leq -2) \vee (x_4 - x_3 \geq 2))$
- Represent Boolean structure as CNF formula:

$$(a \vee b) \wedge (c) \wedge (d) \wedge (e) \wedge (f) \wedge (g) \wedge (h) \wedge (i \vee j)$$

- Interaction between SAT solver & theory solver (IDL):

SAT Outcome	Boolean model	IDL Outcome	Explanation clause (sent to SAT solver)
SAT	$\{a, c, \ldots, h, i\}$	UNSAT	$(\neg e \vee \neg g \vee \neg h)$
UNSAT	$(e) \wedge (g) \wedge (h) \wedge (\neg e \vee \neg g \vee \neg h)$		

3.3 Theories

3.3.1 Real Linear Arithmetic

RLA is a theory in SMT that deals with formulas involving linear inequalities over real numbers. An RLA formula may include constraints like $a_x + b_y \leq c$, where a, b, and c are constants, and x and y are real-valued variables.

The Simplex Method

The Simplex Method is a well-known algorithm from linear programming used for solving systems of linear inequalities, especially for real-valued variables. It works in three phases:

- Formulation: The system of inequalities is converted into an objective function subject to constraints, where the goal is to either maximize or minimize a certain variable.
- Pivoting: The algorithm iteratively improves feasible solutions by "pivoting" on certain variables, adjusting values while keeping all constraints satisfied.
- Termination: It terminates when either a feasible solution is found (satisfying the constraints) or it determines that the system is infeasible.

Fourier-Motzkin

FM is an anternative to simplex. The idea is if $A \leq x$ and $x \leq B$, then $A \leq B$. Elimination by forcing all bounds to be nonempty:

$$\left(\bigwedge_i A_i \leq x \wedge \bigwedge_j x \leq B_j \right) \Leftrightarrow \bigwedge_{i,j} A_i \leq B_j$$

It works in two phases:

- Phase 1: eliminate equalities

$$\sum_{j=1}^{n} a_{ij} \cdot x_j = b_i$$

$$x_n = \frac{b_i}{a_{in}} - \sum_{j=1}^{n-1} \frac{a_{ij}}{a_{in}} \cdot x_j$$

$$\bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_i$$

- Pick equality i and remove x_n

- Phase 2: Variable Elimination

$$\sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_i$$

$$a_{in} \cdot x_n \leq b_i - \sum_{j=1}^{n-1} a_{ij} \cdot x_j$$

$$x_n \leq \frac{b_i}{a_{in}} - \sum_{j=1}^{n-1} \frac{a_{ij}}{a_{in}} \cdot x_j$$

$$\beta_l \leq x_n \leq \beta_u$$

$$\beta_l \leq \beta_u$$

- Pick inequality i and remove x_n
- Combine **all** pairs l and u
- If variable unbounded, then **remove** variable

For example:

- Initial formula:

$$\begin{array}{rcrcrcr} x_1 & - & x_2 & & & \leq & 0 \\ x_1 & & & - & x_3 & \leq & 0 \\ -x_1 & + & x_2 & + & 2x_3 & \leq & 0 \\ & & & & -x_3 & \leq & -1 \end{array}$$

- Eliminate x_1:

Bounds: $(x_2 + 2x_3) \leq x_1 \leq \min(x_2, x_3)$

$$\begin{array}{rcrl} & & 2x_3 & \leq 0 \\ x_2 & + & x_3 & \leq 0 \\ & & -x_3 & \leq -1 \end{array}$$

- Eliminate x_2 (unbounded removed):

$$\begin{array}{rcr} 2x_3 & \leq & 0 \\ -x_3 & \leq & -1 \end{array}$$

- Eliminate x_3:

Bounds: $1 \leq x_3 \leq 0$

$$1 \leq 0$$

- Formula is unsatisfiable
- Unsatisfiable core can be determined by a trace back on the dependencies of the final constraints

3.3.2 Combining Theories

Until now we have assumed there is a single theory, but in practice it is important to combine theories. To do so, we can use the Nelson-Oppen method (under certain conditions), use lazy solving and let the theory solvers communicate through equality. It works in two steps:

- First step: purification

- Iteratively replace each sub-term t by a fresh constant $c \in C$ and add the literal $c = t$.
- Separate into two sets of literals φ_1, φ_2, so that φ_i is on the signature Σ^i.

Example

- $\mathcal{T}_{\mathrm{E}}$...theory of the equality (EUF), $\mathcal{T}_{\mathbb{Z}}$...theory of integers.
- $1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $w_1 \leq x \wedge x \leq w_2 \wedge f(x) \neq f(w_1) \wedge f(x) \neq f(w_2) \wedge w_1 = 1 \wedge w_2 = 2$
- $w_1 \leq x \wedge x \leq w_2 \wedge w_1 = 1 \wedge w_2 = 2$ and $f(x) \neq f(w_1) \wedge f(x) \neq f(w_2)$
- Simplify $1 \leq x \wedge x \leq 2 \wedge w_1 = 1 \wedge w_2 = 2$ and $f(x) \neq f(w_1) \wedge f(x) \neq f(w_2)$

- Second step: guess and check

- For the separation φ_1, φ_2 consider all the shared constants X.
- Guess some equivalence relation R on X.
- Build the arrangement formula:

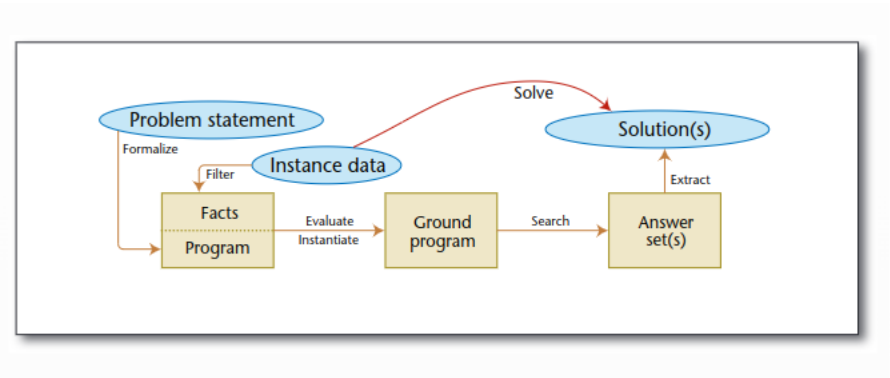
$$\alpha(X, R) = \bigwedge_{u,v \in X, uRv} u = v \ \wedge \bigwedge_{u,v \in X, \neg uRv} u \neq v$$

- Check satisfiability of $\varphi_1 \wedge \alpha(X, R)$ and $\varphi_2 \wedge \alpha(X, R)$.
- $\varphi_1 \wedge \varphi_2$ is satisfiable iff the we get satisfiability for some R.

Chapter 4

Answer Set Programming

ASP is a declarative problem solving paradigm. The problems are specified using rules similar to logic programming, with convenient extensions.



4.1 Answer Sets

The process to compute an answer set is as follows:

- First, ground the program, i.e., substitute specific values for variables in the rules
- In positive programs - simpler case: no negation

```
#show invited/1.
relative(X, Y) :- relative(Y, X).
relative(X, Y) :- relative(X, Z), relative(Z, Y).

relative(X, Y) :- child(X, Y).
relative(X, Y) :- sibling(X, Y).

child(ana, bruno).
sibling(bruno, carlos).
child(ricardo, pedro).

invited(X) :- relative(X, ana).
```

```
invited(bruno).
invited(carlos).
invited(ana).
```

- In programs with negation - consider a guess S - compute the reduct which has no negations
  - If the answer set of the reduct (positive program) is exactly S then S is an answer set

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(X) :- p(X), not q(X).
```

Grounding:

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1), not q(1).
r(2) :- p(2), not q(2).
r(3) :- p(3), not q(3).
r(4) :- p(4), not q(4).
```

- Answer set? {p(1), p(2), p(3), q(2), q(3), q(4), r(1)}
- Reduct:

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1).
```

- Conclusion: {p(1), p(2), p(3), q(2), q(3), q(4), r(1)} is an answer set

An example:

- Program: {p :- p. q :- not p.}
- Check all possible answer sets...

AnswerSet?	Reduct(R)	AnswerSet(R)	Outcome
{ }	p :- p. q.	{q}	NO
{p}	p :- p.	{ }	NO
{q}	p :- p. q.	{q}	YES
{p, q}	p :- p.	{ }	NO

4.2 Extensions

Many extensions can be implemented into ASP (definition of answer set has to be extended), for example:

- Arithmetic: rules may contain symbols for arithmetic operations and comparisons

Program

```
p(1). p(2).
q(1). q(2).
r(X+Y) :- p(X), q(Y), X<Y.
```

Answer set

```
p(1) p(2) q(1) q(2) r(3)
```

- Disjunctive rules: the head of a rule may be a disjunction of several atoms (often separated by bars or semicolons), rather than a single atom

```
p(1) | p(2).
```

```
Answer: 1
p(1)
Answer: 2
p(2)
```

- Choice rules: enclosing the list of atoms in the head in curly braces represents the "choice" construct; choose in all possible ways which atoms from the list will be included in the answer set

```
{ p(1) ; p(2) }.
```

```
Answer: 1

Answer: 2
p(1)
Answer: 3
p(2)
Answer: 4
p(1)  p(2)
```

One may specify bounds on the number of atoms that are included: the lower bound is shown to the left of the expression in braces, and the upper bound to the right

```
1 { p(1) ; p(2) }.
```

Answer sets: 2-4

```
{ p(1) ; p(2) } 1.
```

Answer sets: 1-3

- Constraints: disjunctive rule that has 0 disjuncts in the head, so that it starts with the symbol :-

```
{ p(1) ; p(2) }.
:- p(1), not p(2).
```

Answer sets: 1, 3 and 4

Eliminates the answer sets that satisfy the body of the constraint

- Classical negation: the "classical negation" sign (–) should be distinguished from the negation as failure symbol (not)

answer set `p(a) p(b) -p(c) q(a) -q(c)`
(whether b has property q we do not know)

- Closed world assumption: rule `-A :- not A`
- Frame default: rule `p(T+1) :- p(T), not -p(T+1)`