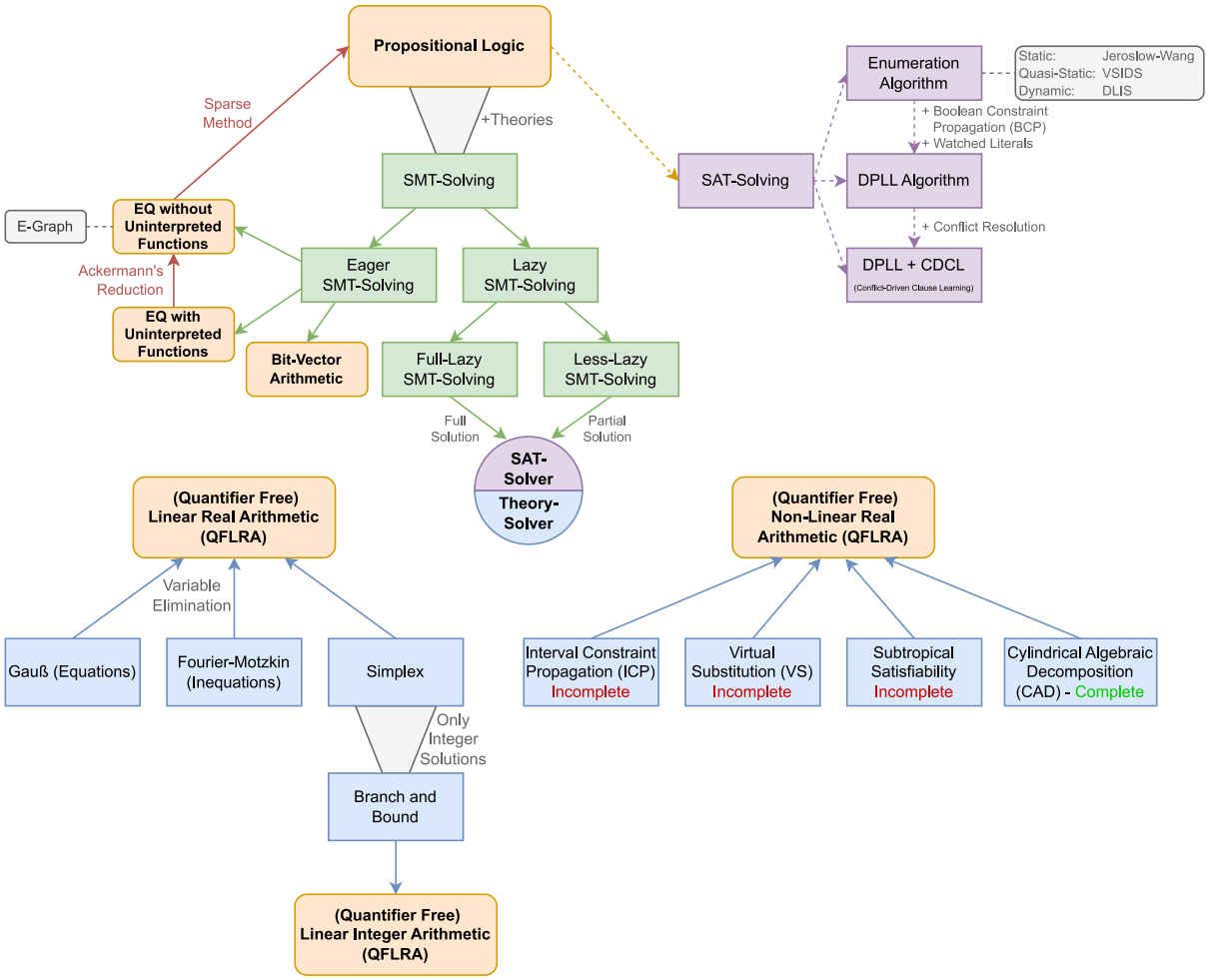
Satisfiability Checking – Summary

Content

Logic and Solver Overview

Logic	Example	Decidable	Solvers
Propositional Logic	$(x \vee y) \wedge (\neg x \vee y)$	Yes	DPLL + CDCL Algorithm
Equality	$(x = y \wedge y \neq z)$	Yes	Sparse Method
Uninterpreted Functions	$F(x) = F(y) \wedge y = z$	Yes	Ackermann's Reduction
Linear Real Arithmetic	$2x + y > 0 \wedge x + y \leq 0$ $2x = 1$	Yes	Gauß, Fourier-Motzkin, Simplex
Linear Integer Arithmetic		Yes	Branch-and-Bound
Nonlinear Real Arithmetic	$x^2 + 2xy + y^2 < 0$ $x^3y^2 = 0$	No	ICP, Subtropical SAT, Virtual Substitution, CAD
Nonlinear Integer Arithmetic		No	-

Topic Overview and Relations



(Thanks @Piccola Radge for the idea :D)

1 Propositional Logic, Theories, Normal Forms

1.1 Propositional Logic and Theories

Syntax

Abstract Grammar: $\varphi := a \mid (\neg\varphi) \mid (\varphi \wedge \varphi)$ with $a \in AP$

Syntactic Sugar: $\bot := (a \wedge \neg a)$, $\top := (a \vee \neg a)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$, …

Order of Binding: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ (left-most binds strongest)

Structures for Predicate Logic

1. Domain: $\mathbb{B} = \{0,1\}$
2. Interpretation: Assignment $\alpha: AP \rightarrow \{0,1\}$
3. Assign: Set of all Assignments, equivalently. $\alpha \in 2^{AP}$ or $\alpha \in \{0,1\}^{AP}$
   (i.e., $AP = \{a, b\}, \alpha(a) = 0, \alpha(b) = 1$)

Satisfaction Relation $\models$

$\alpha \models \varphi$ means "α satisfies φ" or "α is a model of φ"

$\models \subseteq (Assign \times Formula)$ is defined recursively:

$\alpha \models p$ iff $\alpha(p) = true$ $\quad\quad$ $\alpha \models \neg\varphi$ iff $\alpha \not\models \varphi$

$\alpha \models \varphi_1 \wedge \varphi_2$ iff $\alpha \models \varphi_1$ and $\alpha \models \varphi_2$ $\quad\quad$ …

Algorithm: Eval(α, φ)

```
Eval(α, φ) {
   if φ ≡ a return α(a) // Just return the eval.
   if φ ≡ (¬φ1) return not Eval(α, φ1) // Negate the result of eval.
   if φ ≡ (φ1 op φ2) return Eval(α, φ1) [op] Eval(α, φ2) // Recursively eval.
}
```

Complexity: Polynomial in time and space

Satisfiability and Validity

Semantic Classification	Condition	Notation
Valid / Tautology (永真式)	$sat(\varphi) = Assign$	$\models \varphi$
Not Valid	$sat(\varphi) \neq Assign$	$\not\models \varphi$
Satisfiable	$sat(\varphi) \neq \emptyset$	$\not\models \neg\varphi$
Unsatisfiable / Contradiction (矛盾式)	$sat(\varphi) = \emptyset$	$\models \neg\varphi$

$sat(\ldots)$ the set of all satisfying assignments

Satisfiability problem for Propositional Logic

"Given an input propositional formula φ, decide whether φ is satisfiable". ⇒ *NP-complete*

1.2 Normal Forms

Recall:

A *literal* is either a variable or its negation. A *term* is a conjunction of literals, a *clause* a disjunction.

Form	Definition	Negative Ex.	Positive Ex.
Negation Normal Form (*NNF*)	Contains only ¬,∧,∨ Only variables are negated	$a \rightarrow b$	$\neg a \wedge b$
Conjunctive Normal Form (*CNF*)	$\bigwedge_i (\bigvee_j l_{i,j})$	$\neg(a \wedge b)$	$a \wedge (b \vee c)$
Disjunctive Normal Form (*DNF*)	$\bigvee_i (\bigwedge_j l_{i,j})$	$(a \vee b) \wedge c$	$(a \wedge b) \vee (b \wedge c)$

Converting Normal Forms

Converting to NNF

4. Eliminate all connectives other than ∧,∨ ¬
5. Use De-Morgan and Double-Negation Rules to push negations to operands.

⇒ *Linear in Time and Space*

Converting to DNF

1. Convert to NNF
2. Distribute Disjunctions using: $\vDash \varphi_1 \wedge (\varphi_2 \vee \varphi_3) \leftrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

⇒ *Exponential in Time and Space* (otherwise, SAT wouldn't be NP-Complete, because SAT-checking a DNF is linear in time and space)

Converting to CNF: Option 1

1. Convert to NNF
2. Distribute Conjunctions using: $\vDash \varphi_1 \vee (\varphi_2 \wedge \varphi_3) \leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$

⇒ *Exponential in Time and Space*

Converting to CNF: Option 2 (Tseitin's Encoding)

⇒ *Linear in Time and Space* (if new variables are added)

⇒ $3n + 1$ clauses instead of 2^n, but $3n$ variables rather than $2n$

⇒ Converted formulae are *not equivalent, but equi-satisfiable*.

1. Construct the Parse Tree (the order in which we have to evaluate the formula)
2. Associate new auxiliary variable with each inner (non-leaf) node.
3. Add constraints that define these new variables.
4. Enforce the truth of the root node. (Important – don't forget!)

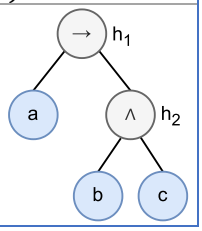
Example: Tseitin's Encoding

Convert the following formula into a CNF using Tseitin's Encoding: $\varphi = (a \rightarrow (b \wedge c))$

1. & 2. Construct the Parse Tree & Associate new variables (see right)

Construct Formula:

$$\varphi_{CNF} = \underbrace{h_1}_{4.} \wedge \underbrace{(h_1 \leftrightarrow (a \rightarrow h_2)) \wedge (h_2 \leftrightarrow (b \wedge c))}_{3.}$$



Note that each node's encoding has a CNF-representation with 3 or 4 clauses.

1.3 Resolution

Soundness and Completeness

Soundness: We can only make "correct" conclusions (and no incorrect ones)

Completeness: We can conclude (at least) all "correct" conclusions

(Note that "all 'correct' conclusions" can also imply that we additionally conclude incorrect ones)

Resolution

The resolution inference rule for CNF:

$$\frac{(l \vee l_1 \vee \ldots \vee l_n) \quad (\neg l \vee l'_1 \vee \ldots \vee l'_m)}{(l_1 \vee \ldots \vee l_n \vee l'_1 \vee \ldots \vee l'_m)} \text{ Resolution}$$

Sound and complete proof system for CNF

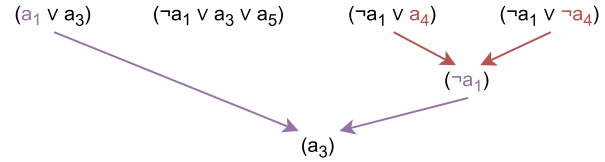
Inference Rules

$$\frac{Antecedents}{Consequents} \text{ (rule name)}$$

Example: Resolution to prove a formula

Use the following formula to prove $\varphi \rightarrow a_3$*:*

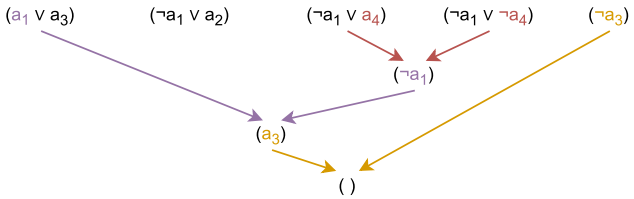
$\varphi = (a_1 \vee a_3) \wedge (\neg a_1 \vee a_3 \vee a_5) \wedge (\neg a_1 \vee a_4) \wedge (\neg a_1 \vee \neg a_4)$



Example: Resolution to prove UNSAT

Prove that the following formula is UNSAT:

$\varphi = (a_1 \vee a_3) \wedge (\neg a_1 \vee a_2) \wedge (\neg a_1 \vee a_4) \wedge (\neg a_1 \vee \neg a_4) \wedge (\neg a_3)$



(If we can derive the empty clause, the formula must be UNSAT)

Example: Apply Resolution to eliminate propositions

Apply resolution to eliminate the propositions in the order A, B, C, D *from the following formula:*

$\varphi = (A \vee \neg B \vee C) \wedge (\neg B) \wedge (B \vee \neg C \vee D) \wedge (B \vee D)$

1. Eliminate A: Only occurs positive, thus we can trivially set it to true and remove the clause
   $(\neg B) \wedge (B \vee \neg C \vee D) \wedge (B \vee D)$
2. Eliminate B: For this, combine positive and negative literals and apply resolution
   $(\neg C \vee D) \wedge (D)$
3. Eliminate C: Only occurs negated, so we can trivially set it to false and remove the clause
   (D)
4. Eliminate D: Only occurs positive, we can trivially set it to true
   $true$

1.4 Modeling with Propositional Logic

Why does it help us to solve the SAT-Problem?

Because numerous problems in the industry can be converted into SAT-formulas and subsequently solved via the satisfiability problem (i.e., logistics, planning, cryptography, …)

The task of modeling a problem using propositional logic is not really about understanding but more about just practicing in my opinion. So, in the following I'll give one example with explanations, and this can be applied to other examples pretty straightforward.

Normally, it is easiest to think about all necessary requirements step-by-step and construct the formula accordingly. For example, "exactly 1", could be converted using one formula for "at most one" and one for "at least one".

Example: Modeling with Propositional Logic: Seminar Topic Assignment

Given n *participants,* n *topics and a set of preferences* $E \subseteq \{1, \dots, n\} \times \{1, \dots, n\}$ *with* $(p, t) \in E$ *indicating that participant* p *would like to take topic* t*.*
Construct a formula to solve the problem using the constraints.

$x_{p,t}$ means "participant p gets assigned topic t"

Each participant is assigned at least one topic: $\bigwedge_{p=1}^{n}\left(\bigvee_{t=1}^{n}(x_{p,t})\right)$

Each participant must get assigned at least one topic.

Each participant is assigned at most one topic: $\bigwedge_{p=1}^{n}\left(\bigwedge_{t_1=1}^{n-1}\left(\bigwedge_{t_2=t_1+1}^{n}\neg\left(x_{p,t_1}\wedge x_{p,t_2}\right)\right)\right)$

For each participant and all pairs of distinct topics t_1, t_2 the participant cannot get assigned to both. To remove symmetry and identical topics, we pick the bounds as they are)

Each participant gets a preferred topic: $\bigwedge_{p=1}^{n}\left(\bigwedge_{t=1,(p,t)\notin E}^{n}\left(\neg x_{p,t}\right)\right)$

For each participant and each topic where (p,t) is not in our relation E (that is, p hasn't chosen t as a preference), the participant doesn't get assigned topic t.

Each topic gets assigned to at most one participant: $\bigwedge_{t=1}^{n}\left(\bigwedge_{p_1=1}^{n-1}\left(\bigwedge_{p_2=p_1+1}^{n}\neg\left(x_{p_1,t}\wedge x_{p_2,t}\right)\right)\right)$

For each topic and all pairs of distinct participants p_1, p_2 they cannot get assigned to the same topic.

Subsequently, the final answer would be the conjunction of the 4 formulas mentioned previously.

2 Propositional SAT Solving

SAT-Solving includes a Combination of three techniques:

1. *Enumeration / Exploration*
   List one assignment after another incremental – take a variable and just try a value (given by the selected heuristic).
2. *Propagation*
   If you have a clause in which the current assignment assigns all but one literal to false, the last (unassigned) one must be true – otherwise the clause is evaluates to false.
   I.e., consider $(a \vee b \vee \neg c)$ and $\alpha(a) = \alpha(b) = False$. Then we can conclude that we must assign $False$ to c to fulfil the clause)
3. *Resolution*
   Instead of "pure backtracking", try to *generate a reason* which part of the assignment causes the unsatisfiability. Afterwards, we can exclude the source of conflict instead of just flipping variable guesses randomly.

2.1 Decision Heuristics – Overview

Note: This is only a brief overview. For a detailed explanation with examples of the heuristics, please refer to "Decision Heuristics.pdf"

Name	Heuristics
Naïve Decision Static	1. Stick to the given variable and sign ordering.
Dynamic Largest Individual Sum (DLIS) Dynamic	1. For each literal l, let C_l be the number of unresolved clauses (= not already SAT) in which l appears and decide for the literal with highest C_l. In case of a tie, we use the fall-back variable ordering i.e., $(\neg x \vee y)$ would give us $C_x = 0, C_{\neg x} = 1, C_y = 1, C_{\neg y} = 0$ 2. Choose an assignment that increases the # of satisfied clauses the most.
Jeroslow-Wang Static	1. For each literal l compute $J(l) = \sum_{\text{clause } c \text{ in the CNF containing } l} 2^{-\lvert c\rvert}$ This gives *exponentially higher weights to literals in shorter clauses* i.e., $(\neg x \vee y) \wedge (\neg x)$ would give us: $J(x) = 0, J(\neg x) = \frac{1}{2^2} + \frac{1}{2^1} = \frac{3}{4}, J(y) = \frac{1}{2^2}, J(\neg y) = 0$ 2. Choose a literal l that maximizes $J(l)$
Variable State Independent Decaying Sum (VSIDS) Dynamic	1. Each literal has a counter initialized to 0. 2. When resolution gets applied to a clause, the counters of its literals are increased. This gives *priority to variables involved in recent conflicts*. 3. Decision: The unassigned variable with the highest counter is chosen. 4. Periodically, all the counters are divided by a constant.

Static = Compute once at the beginning is sufficient; Dynamic = Re-Compute in each iteration.

2.2 Boolean Constraint Propagation (BCP)

BCP can be used to speed up computation by deriving consequences of decisions. Given a partial assignment, a clause can be:

1. SAT — at least one literal is SAT.
2. UNSAT — all literals are assigned but none are SAT.
3. Unit — all but one literal is assigned but none are satisfied.
4. Unresolved — else

Because we're interested in satisfiability, *we can make use of unit clauses*. If all but one literal is assigned, but none is satisfied, we can *derive a satisfying assignment* of the unassigned literal.

Boolean Constraint Propagation (BCP)

After an assignment has been made, propagate the assignment, and try to derive other assignments. I.e., consider $(a \vee \neg b) \wedge (a \vee b \vee c)$. If we decide for $\alpha(a) = false$ and we propagate this, we can derive $\alpha(b) = false$. Afterwards, we can also propagate the assignment of b and additionally derive $\alpha(c) = true$. By this, we've derived a satisfying assignment in a single step (sort of).

BCP using Watched Literals

A weakness of pure BCP is the large effort of checking all literals in all clauses for each propagation. To bypass this limitation, we can use watched literals.

Idea:

In each clause, *watch two different literals such that either one is true, or both are unassigned*.
If a literal l gets false, we propagate it by checking each clause c in which it is watched. Let l' be the other watched literal:

1. If $\alpha(l') = true$, the clause is SAT.
2. Else, if we find a non-false literal different from l, l' to be watched instead of l, we are done.
3. Else, if l' is unassigned, the clause is unit; $\alpha(l') = true$
4. Else, if $\alpha(l') = false$, the clause is conflicting.

Example: Watched Literals
Given $(a \vee \neg b \vee \neg c \vee d)$ *with* $\alpha(a) = \alpha(c) = 0, \alpha(b) = 1$. *List all pairs of literals suitable to be watched.*
$(a, \neg c), (\neg b, \neg c), (\neg c, d)$

2.3 Conflict Resolution and Backtracking

We represent (partial) variable assignments in the form of an implication graph (labelled directed acyclic graph). I leave out the definition here, as I don't think that it is that important – focus more on the concept and idea behind it. Intuitively, the implication graph stores the assignments and shows us the implications of all assignments. This will become handy when backtracking to find the root cause of UNSAT assignments later.

Notation: $x_1 = 0@1$ means "Proposition x_1 is assigned 0 at decision level 1"

Create the Implication Graph

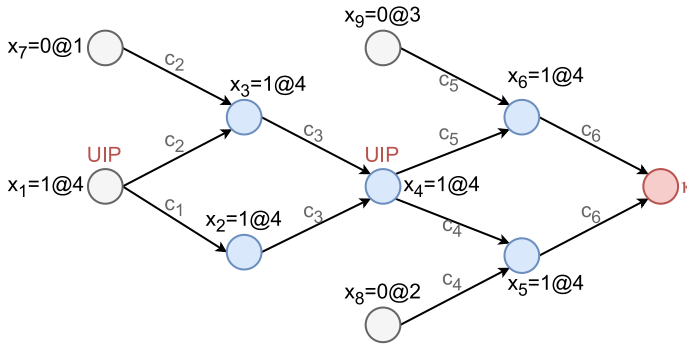
1. Draw all Decisions (gray circles)
2. For each unit clause, add the implication by drawing the node, decision, and clause name. Do step 2 recursively until no more unit clauses exist. For this, I would recommend keeping track of true/false assignments in the original formula using two different colors.

Example: Implication Graph

Construct the implication graph for the following setting:

$\underbrace{(\neg x_1 \vee x_2)}_{c_1} \wedge \underbrace{(\neg x_1 \vee x_3 \vee x_7)}_{c_2} \wedge \underbrace{(\neg x_2 \vee \neg x_3 \vee x_4)}_{c_3} \wedge \underbrace{(\neg x_4 \vee x_5 \vee x_8)}_{c_4} \wedge \underbrace{(\neg x_4 \vee x_6 \vee x_9)}_{c_5} \wedge \underbrace{(\neg x_5 \vee \neg x_6)}_{c_6}$

The decisions are: $\{x_7 = 0@1,\ x_8 = 0@2,\ x_9 = 0@3,\ x_1 = 1@4\}$



(κ indicates that we have a conflict!)

Terminology

Asserting Clause: Conflict Clause with a *single literal from the current Decision Level*

UIP: A unique implication point (UIP) for κ is a node $n \neq \kappa$ such that *all paths from the last decision to κ go through n*.
The decision itself is trivially a UIP.

Unsatisfiable Core: All nodes that are "involved" in the conflict. In the above example, the whole formula is the minimal unsatisfiable core.

2.4 DPLL + CDCL Algorithm

Basic SAT Algorithm

Algorithm: DPLL + CDCL

```
DPLL_CDCL(φ) {
   if (!BCP()) return UNSAT; // BCP returns false ⇒ Conflict
   while (true) {
      if (!Decide()) return SAT; // If all variables have been assigned.
      while (!BCP()) // Recursively propagate & try to resolve conflicts
         if (!Resolve_Conflict()) return UNSAT; // Conflict Res. Failed?
   }
}
```

DPLL = Davis-Putnam-Logemann-Loveland, CDCL = Conflict-Driven Clause Learning
The main functions decide(), BCP(), and resolve_conflict() are described in the following.

Decide()

Chooses the next variable and value ("guess") and returns false, if all variables are assigned without a conflict (SAT)

BCP()

(Recursively) propagate the decisions to unit clauses. I.e., consider $(a \vee b)$ with $\alpha(a) = 0$ then propagating this decision would give us $\alpha(b) = 1$.

Resolve_Conflict()

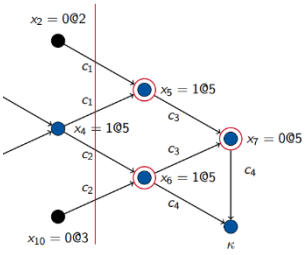
Start with the conflicting clause. Apply resolution with the antecedent of the last assigned literal, until we get an asserting clause.

When is a decision unsatisfiable? We enumerate our decision levels, and sometimes we can derive things that are *contradictory at Decision Level 0 = Must-Assignments = We didn't do any decisions yet!* ⇒ then the *formula must be unsatisfiable*.

Example: Conflict Resolution

Given the assignment order x_4, x_5, x_6, x_7 and the following formula, apply conflict resolution to c_4.

$\underbrace{(\neg x_4 \vee x_2 \vee x_5)}_{c_1} \wedge \underbrace{(\neg x_4 \vee x_{10} \vee x_6)}_{c_2} \wedge \underbrace{(\neg x_5 \vee \neg x_6 \vee \neg x_7)}_{c_3} \wedge \underbrace{(\neg x_6 \vee x_7)}_{c_4} \wedge \dots$



Start by resolving c_4 (conflicting clause) with c_3 using x_7 (x_7 is the last assigned variable):

$$\frac{c_4: (\neg x_6 \vee x_7) \quad c_3: (\neg x_5 \vee \neg x_6 \vee \neg x_7)}{c_6: (\neg x_5 \vee \neg x_6)}$$

Because both, x_5 and x_6 are from our current decision level (5), we must resolve again. Resolve c_6 with c_2 using x_6:

$$\frac{c_6: (\neg x_5 \vee \neg x_6) \quad c_2: (\neg x_4 \vee x_{10} \vee x_6)}{c_7: (\neg x_4 \vee \neg x_5 \vee x_{10})}$$

Because both, x_4 and x_5 are from our current decision level (5), we must resolve again. Resolve c_7 with c_1 using x_5:

$$\frac{c_7: (\neg x_4 \vee \neg x_5 \vee x_{10}) \quad c_1: (\neg x_4 \vee x_2 \vee x_5)}{c_8: (x_2 \vee \neg x_4 \vee x_{10})}$$

Only x_4 is from decision level 5. Thus, the clause is asserting, and the resolution is finished.

3 First-Order Logic

Sometimes, propositional logic is not expressive enough for modelling certain problems. First-Order (FO) logic is a framework that uses predicate symbols to lift from "theory" to the "logical" level.

"Theory Parts": Constants (0, 1), Variables (x, y, z), Function symbols (binary $+$)

"Logical Parts": Predicates: $\geq, >, =, <, \leq, \ldots$

Logical Connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \ldots$

Universal Quantifier: $\forall, \exists$

Example: FO-Logic Formulas

Formalize the following argumentation using FO-Logic:
(1) All men are mortal. (2) Socrates is a man (3) Subsequently, Socrates is mortal.

Formal definitions:

Constants: Socrates

Variables: x

Predicates: unary $isMen$, $isMortal$

Formalization:

1. $\forall x. isMen(x) \rightarrow isMortal(x)$
2. $isMen(\text{Socrates})$
3. $isMortal(\text{Socrates})$

Free and Bounded Variables

Each variable occurrence in a formula is either bound or free. Intuitively, a variable is bound if it "defined" by $\forall$ or $\exists$.

For example, in $P(z) \vee \forall x. \big(P(x) \rightarrow Q(z)\big)$, z occurs free in P, Q and x occurs bound.

Signature Σ, Σ-formula, Σ-sentence

- A *signature* Σ fixes the set of non-logical symbols (up to variables).
- A *Σ-formula* is a formula with non-logical symbols from Σ.
- A *Σ-sentence* is a Σ-formula without free variable occurrences.

Intuitively: a Σ-formula must contain at least one constant/variable (so almost any FO-Formula :D)

Σ-Structures

A Σ-Structure is given by:

- A *domain* D (i.e., $\mathbb{N}_0$)
- An *interpretation* I of the non-logical symbols in Σ that maps
  - each *constant symbol* to a domain element,
  - each *function symbol* of arity n to a function of type $D^n \rightarrow D$, and
  - each *predicate symbol* of arity n to a predicate of type $D^n \rightarrow \{0,1\}$

Example: Σ-Structures

Given $\Sigma = \{0,1,+,=\}$ and $\varphi = \exists x. x + 0 = 1$ a Σ-formula. Is φ satisfiable?

Yes. Consider the following structure S:

- Domain $\mathbb{N}_0$
- Interpretation:
  - $0, 1$ are mapped to $0,1$ in $\mathbb{N}_0$,
  - $+$ means addition,
  - $=$ means equality.

S satisfies φ / S is a model of φ / $S \vDash \varphi$

Theories

- A *Σ-Theory* T is defined by a set of Σ-sentences.
- A Σ-Formula φ is *T-satisfiable* if there exists a structure that satisfies both, the sentences of T and φ
- A Σ-formula φ is *T-valid* if all structures that satisfy the sentences defining T also satisfy φ.

Intuitively: A theory is just a set of FO-formulas, and a formula must not only be satisfiable/valid, but must also satisfy the sentences of the theory ("satisfy the formula and the constraints")

Tradeoff: Expressivity vs. computational hardness

- The more expressive the logic the harder it might be to decide the SAT/validity of formulas.

Further Remarks

- 2-CNF is a logic fragment of Propositional Logic
- 2-CNF is less expressive than Propositional Logic
- There exists undecidable FO-Theories

4 Eager SMT-Solving

Recall: SAT-Modulo-Theories (SMT) solving extends propositional logic with theories.

We have introduced two different SMT-Solving approaches:

- **Eager SMT-Solving:** "Theory First"
  - *Transform* logical formulas over some theory into *satisfiability-equivalent* propositional logic formulas. 先转换为命题逻辑公式, 再使用sat求解
  - Apply SAT-Solving.
- **Lazy SMT-Solving:** "Theory Later"
  - Use *SAT-Solver* to *find solution for the Boolean Skeleton* of the formula.
  - Utilize a *Theory Solver* to check whether the solution satisfies the underlying theory.
    先提取布尔骨架使用sat进行判断, 再对sat的结果使用theory solver进行检查, 检查的结果将反馈给sat solver

4.1 Equality Logic with Uninterpreted Functions

Equality Logic with uninterpreted functions

We extend propositional logic with equalities and uninterpreted functions (UFs)

Terms: $t ::= c \mid x \mid F(t, \dots, t)$ 常量, 变量或函数

Formulas: $\varphi ::= t = t \mid (\varphi \wedge \varphi) \mid (\neg\varphi)$

(UFs are functions whose semantics aren't defined. We only know that they're functions.)

Purpose of UFs

- Replacing interpreted functions by UFs can make reasoning easier.
- UFs ignore the semantics, but the functional congruence must still hold:
  $x = y \rightarrow f(x) = f(y)$
- φ^{EQ} denotes formulas with equalities, φ^{UF} with equalities and UFs.

Conversion: $UF \xrightarrow{\text{Ackermann}} EQ \xrightarrow{\text{Bryan/Sparse}} Prop.Logic$

4.1.1 Ackermann's Reduction (UF → EQ)

Idea:

- Transform formula φ^{UF} with UF F into a satisfiability-equivalent formula φ^{E} without the UF.
- $\varphi^{EQ} := \varphi_{flat} \wedge \varphi_{cong}$
  φ_{flat}: Flattening of φ^{UF}, where each function application is replaced by a fresh variable.
- φ_{cong}: Encode functional congruence.

To *check the validify* of φ^{UF}, check the satisfiability of $\varphi^{EQ} := \varphi_{cong} \wedge \neg\varphi_{flat}$ and *invert the answer*.

Algorithm: Ackermann's Reduction

1. $\varphi_{flat} := \mathcal{T}(\varphi^{UF})$ where $\mathcal{T}$ replaces each occurrence F_i of F by a fresh theory variable f_i.
   (By this, we've only encoded function applications by variables representing the value. What is missing is the congruence of functions)
2. $\varphi_{cong} := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} (\mathcal{T}(\arg(F_i)) = \mathcal{T}(\arg(F_j)) \rightarrow f_i = f_j$
   (If the arguments are the same, the function values must also be the same)
3. Return $\varphi^{EQ} = \varphi_{flat} \wedge \varphi_{cong}$

Example: Ackermann's Reduction

Convert $\varphi^{UF} := (x_1 \neq x_2) \vee (F(x_1) = F(x_2)) \vee (F(x_1) \neq F(x_3))$ *into a satisfiability-equivalent formula* φ^{EQ}.

$\varphi_{flat} = (x_1 \neq x_2) \vee (f_1 = f_2) \vee (f_2 = f_3)$

$\varphi_{cong} = \big((x_1 = x_2) \rightarrow (f_1 = f_2)\big) \wedge \big((x_1 = x_3) \rightarrow (f_1 = f_3)\big) \wedge \big((x_2 = x_3) \rightarrow (f_2 = f_3)\big)$
$\varphi^{EQ} = \varphi_{flat} \wedge \varphi_{cong}$

4.1.2 Bryant's Reduction / Sparse Method (EQ → SAT)

Idea of E-Graph

- A E-Graph is just a graphical representation of (dis)equalities.
- For each variable we add a node and connect them according to the (dis)equalities
  Equalities: solid line, disequalities: dashed line.
- An *Equality Path* is a path that only uses edges from $E_=$ (*Disequality Path* respectively)
- A *Contradictory Cycle* is a cycle that uses exactly one edge from $E_{\neq}$ 只有一个边是不等的cycle
- A *Simple (Contradictory) Cycle* is a Contradictory Cycle with no repeating nodes.

Simplify formulas using E-Graphs 所有不在simple Contradictory cycle的进行替换

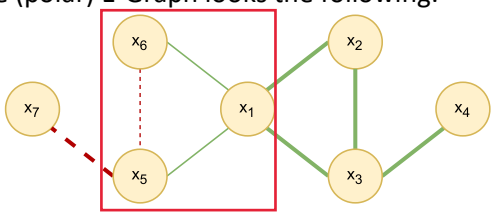
We can replace all edges that are not part of any simple contradictory cycle by replacing disequality edges with false, and equality edges with true. *This preserves satisfiability (Not equivalence!)*.
Note that this could simplify the formula, which then might simplify the E-Graph which again could simplify the formula (i.e., $(x_1 = x_2 \vee x_1 = x_4) \wedge (x_1 \neq x_4 \vee x_2 = x_3)$).

Graphically

Consider the following formula:

$\varphi \coloneqq x_1 = x_2 \wedge x_1 = x_3 \wedge x_1 = x_5 \wedge x_1 = x_6 \wedge x_2 = x_3 \wedge x_3 = x_4 \wedge x_5 \neq x_6 \wedge x_5 \neq x_7$

The (polar) E-Graph looks the following:



One example for an equality path would be x_3x_4, whereas $x_7x_5x_6$ is a disequality path. The only Simple Contradictory Cycle is $x_1x_5x_6$. Subsequently, all other nodes can be simplified.

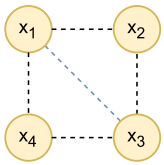
After simplifying, our formula would look like:

$\rightarrow \underbrace{x_1 = x_2}_{true} \wedge \underbrace{x_1 = x_3}_{true} \wedge x_1 = x_5 \wedge x_1 = x_6 \wedge \underbrace{x_2 = x_3}_{true} \wedge \underbrace{x_3 = x_4}_{true} \wedge x_5 \neq x_6 \wedge \underbrace{x_5 \neq x_7}_{\neg false}$

$\rightarrow x_1 = x_5 \wedge x_1 = x_6 \wedge x_5 \neq x_6$

Some more terminology

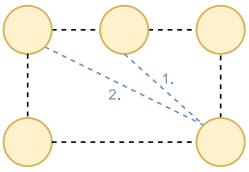
In the following, we neglect polarity (dashed edge ≡ equality or disequality)



- A cycle is *chord-free* iff there is no edge between any two nodes of the cycle, that are not part of the cycle itself. 除了cycle本身之外, 任意两点间没有边相连
- A graph is *chordal* iff every simple cycle of at least 4 nodes has a chord.

In the example on the right, $x_1x_2x_3x_4$ wouldn't be chord-free because of x_1x_3. The graph is chordal.
每个含有4个以上点的simple cycle有chord(弦)

Making the E-Graph Chordal



- Iteratively connect the neighbors of the vertices
  In the example on the right, we've started on the lower right edge, and iteratively connected the other edges so that we only have triangles left. 连接点, 最后只剩下三角形

How can E-Graphs help us to transform EQ to SAT?

- Simply converting the EQ into SAT would be possible but is far to inefficient.
- E-Graphs can help us to easily detect, which parts of the EQ-formula are important. However, we can have *exponentially* many simple contradictory cycles that need to be transformed.
- Thus, we've looked at how to simplify the E-Graph and construct a Chordal Graph. Note, that the *simplification is only satisfiability equivalent*.
- By making the graph chordal, we only have triangles left. Those can easily be transformed into SAT-equivalent Propositional Logic Formula by encoding the transitivity of each triangle.
- This approach is *polynomial in time and space*.

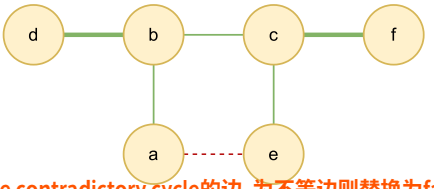
Algorithm: Bryant's Reduction / Sparse Method

1. Construct φ_{sk} by replacing each equality $t_i = t_j$ in φ^{EQ} by a fresh Boolean variable e_{ij}.
2. Construct non-polar E-Graph $G^E(\varphi^{EQ})$ and make it chordal.
3. For each triangle $(e_{ij}, e_{jk}, e_{kj}) \in G^E(\varphi^{EQ})$: 任意两边为真->第三边为真
   $\varphi_{trans} := \varphi_{trans} \wedge (e_{ij} \wedge e_{jk}) \rightarrow e_{ki} \wedge (e_{ij} \wedge e_{ik}) \rightarrow e_{jk} \wedge (e_{ik} \wedge e_{jk}) \rightarrow e_{ij}$
   (Encode the transitivity of equalities – If we have an edge between "ij" and "jk", then we must also have an edge between "ki" – same for the other 2 cases)
4. Return $\varphi_{sk} \wedge \varphi_{trans}$

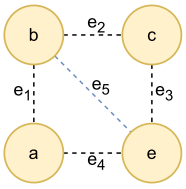
Example: Sparse Method

Convert $\varphi^{EQ} := a = b \wedge (b = c \vee c = e) \wedge (b = d \vee c = f) \wedge a \neq e$ into a satisfiability-equivalent formula φ^{SAT} in propositional logic.

1. $\varphi_{sk} := e_1 \wedge (e_2 \vee e_3) \wedge (e_4 \vee e_5) \wedge \neg e_6$ (Boolean skeleton)
   (Note, that we must negate disequalities!)
2. E-Graph



Complete (Polar) E-Graph



Non-Polar Chordal E-Graph

对于不属于simple contradictory cycle的边, 为不等边则替换为false, 为等边则替换为true

Optional: We can simplify the E-Graph by replacing $(b = d \vee c = f)$ with $true$.
Thus, we only need to consider $\varphi^{EQ}_{simpl} := a = b \wedge (b = c \vee c = e) \wedge a \neq e$ with Boolean Skeleton $\varphi_{sk} = e_1 \wedge (e_2 \vee e_3) \wedge \neg e_4$

没有顶点重复的cycle为simple cycle

3. $\varphi_{trans} = true$
4. Encode transitivity by encoding each triangle:

$$\varphi_{trans} = \underbrace{((e_1 \wedge e_4) \rightarrow e_5 \wedge (e_1 \wedge e_5) \rightarrow e_4 \wedge (e_4 \wedge e_5) \rightarrow e_1)}_{\text{Triangle } abe} \wedge \underbrace{((e_2 \wedge e_3) \rightarrow e_5 \wedge (e_3 \wedge e_5) \rightarrow e_2 \wedge (e_2 \wedge e_5) \rightarrow e_3)}_{\text{Triangle } bce}$$

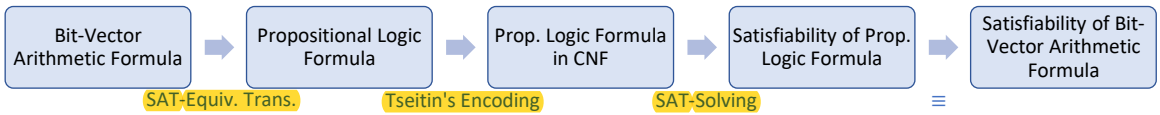
5. Return $\varphi^{SAT} = \varphi_{sk} \wedge \varphi_{trans}$

Note, that we need φ_{sk} the Boolean Skeleton, because the E-Graph cannot decide between $(a = b \wedge a \neq b)$ and $(a = b \vee a \neq b)$ – However, this might make a huge difference.

4.2 Finite-Precision Bit-Vector Arithmetic

4.2.1 Bit-Blasting

"Pipeline" of Bit-Blasting



Finite-Precision Bit-Vector Arithmetic

The arithmetic contains mostly the "normal" bit operations such as Bit-Shifts, Bitwise XOR, …
The detailed definition is despaired here.

Bit-Vector

A bit-vector x of length l is a function $x: \{0, \dots, l-1\} \to \{0,1\}$. We also write $x_{l-1} \dots x_1 x_0$.

Different Encodings of Bit-Vectors

Encoding	Definition	Example
Binary Encoding	$[\![x_{[l]U}]\!] := \sum_{i=1}^{l-1} x_i 2^i$	$[\![1100\ 1000_{[8]U}]\!] = 128 + 64 + 8 = 200$
Two's Complement	$[\![x_{[l]S}]\!] = -2^{l-1} x_{l-1} + \sum_{i=1}^{l-2} x_i 2^i$	$[\![1100\ 1000_{[8]S}]\!] = -128 + 64 + 8 = -56$

How to transform a Bit-Vector Arithmetic into SAT-Equivalent Propositional Logic Formula?

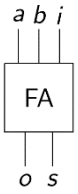
1. Build the propositional flattening (Boolean Skeleton).
2. Add a Boolean variable for each bit of each sub-expression (term)
3. Add constraints to define the meaning of each sub-expression. The constraints will vary depending on the Arithmetic.

Example Encoding

$a|b$ (Bitwise OR) can be encoded as bitwise operation by: $\bigwedge_{i=1}^{l}(c_i \Leftrightarrow (a_i|b_i))$

Bit-Wise Full Adder

We want to construct a formula for $a + b$ where $a + b$ are bits.
Furthermore, we want to have an additional input bit i (i.e., for the carry-over when chaining multiple adders)



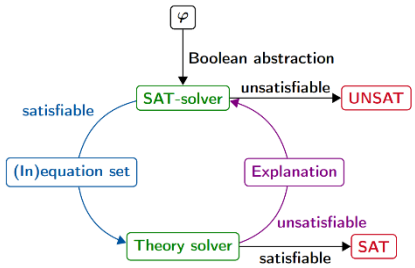
a	b	i	o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

First, we can construct a table with all cases (given on the right). i is an additional input Note, s the sum, and o the overflow. We observe:
$o \equiv (a + b + i)\ div\ 2$ (We have an overflow iff we have more than one "1")
$s \equiv (a + b + i)\ mod\ 2$ (The sum is "0" if we have an even number of "1")

This can be converted into the following Propositional Formulas:
$o \equiv (a \wedge b) \vee (a \wedge i) \vee (b \wedge i)$ $\quad$ $s \equiv a \oplus b \oplus i$
Afterwards, we could convert them into CNF (which gives us $6 + 8$ clauses) and apply SAT-Solving.
The Full-Adder can simply be chained together by connecting i_n with o_{n-1}.

5 Lazy SMT Solving

Idea of Lazy SMT-Solving



- In contrast to Eager SMT-Solving, we *solve Boolean first* instead of Theory first.
- For this, we hide the theory by replacing it with Boolean auxiliary constraints and try to satisfy the Boolean structure of the formula using a SAT-Solver.
- If the SAT-Solver doesn't find a solution ⇒ UNSAT
- Otherwise, the *SAT-Solver consults the theory solver* which then checks whether the assignment is consistent (that is, "works" in the given theory with its constraints)
- The theory solver then gives either SAT, or UNSAT *with a reason/explanation*, why the theory failed. This then helps the SAT-Solver in the next iteration, to exclude the given solution for the following iterations.
- We continue this cycle until we've either found a solution that is also consistent with the theory, or we have no more possible assignments (then the problem is UNSAT)

Full-Lazy vs Less-Lazy SMT-Solving

In *Full-Lazy SMT-Solving*, the Theory-Solver is only "asked" for full solutions (= full variable assignments), whereas in *Less-Lazy SMT-Solving* also partial solutions can be tested against the theory.

5.1AlgorithFullm: Full-Lazy S-Lazy SMT-Solving

5.1 Full-Lazy SMT-Solving

Algorithm: Full-Lazy SMT-Solving

1. Construct the Boolean Abstraction/Skeleton φ_{abs} by replacing each theory constraint with a new variable.
2. Search for a solution for φ_{abs} using *SAT-Solving*
   If there is no solution ⇒ return UNSAT
3. Given a solution for φ_{abs}, check the set of all ***true*** *theory constraints for consistency*
   If they are consistent ⇒ return SAT
4. Compute an *explanation* for the inconsistency in form of a CNF formula.
5. Learn the new Boolean Abstraction φ_{abs}
6. Apply *conflict resolution* if the learned clause is not asserting.
7. Goto 2.

Example: Full-Lazy SMT-Solving

Apply Full-Lazy SMT-Solving for the following formula:

$$\varphi := (\underbrace{p_1 = 0}_{a_1} \vee \underbrace{p_2 = 0}_{a_2} \vee \underbrace{p_3 = 0}_{a_3}) \wedge \underbrace{p_1 + p_2 + p_3 \geq 100}_{a_4} \wedge (\underbrace{p_1 \geq 5}_{a_5} \vee \underbrace{p_2 \geq 5}_{a_6}) \wedge \underbrace{p_3 \geq 10}_{a_7}$$
$$\wedge \underbrace{p_1 + 2p_2 + 5p_3 \leq 180}_{a_8} \wedge \underbrace{3p_1 + 2p_2 + p_3 \leq 300}_{a_9}$$

For the SAT-Solving, assume a fixed variable order $a_1 > \cdots > a_9$ and assign false

Boolean Abstraction:

$\varphi_{abs} = (a_1 \vee a_2 \vee a_3) \wedge a_4 \wedge (a_5 \vee a_6) \wedge a_7 \wedge a_8 \wedge a_9$

SAT-Solving gives us (left out here, should be clear by now :D):

$a_1 = a_2 = a_5 = 0$, $a_3 = a_4 = a_6 = a_7 = a_8 = a_9 = 1$

Check whether the **TRUE** theory constraints are satisfied:

a_3: $p_3 = 0$ a_4: $p_1 + p_2 + p_3 \geq 100$ a_6: $p_2 \geq 5$

a_7: $p_3 \geq 10$ a_8: $p_1 + 2p_2 + 5p_3 \leq 180$ a_9: $3p_1 + 2p_2 + p_3 \leq 300$

观察到a3和a7冲突

If we look at the original constraints (corresponding to the abstractions), we observe that a_3 and a_7 are conflicting ⇒ One of them must be false ⇒ Add new clause $(\neg a_3 \vee \neg a_7)$ ¬(a3∧a7)=(¬a3∨¬a7)
⇒ $\varphi := \varphi \wedge (\neg a_3 \vee \neg a_7)$

Learn the Boolean Abstraction:
$\varphi_{abs} = (a_1 \vee a_2 \vee a_3) \wedge a_4 \wedge (a_5 \vee a_6) \wedge a_7 \wedge a_8 \wedge a_9 \wedge (\neg a_3 \vee \neg a_7)$
Conflict Resolution: Clause is already asserting, since only a_3 is from the current DL.
Goto Step 2.

SAT-Solving gives us (again, left out):
$a_1 = a_3 = a_5 = 0$ $\quad$ $a_4 = a_7 = a_8 = a_9 = a_2 = a_6 = 1$

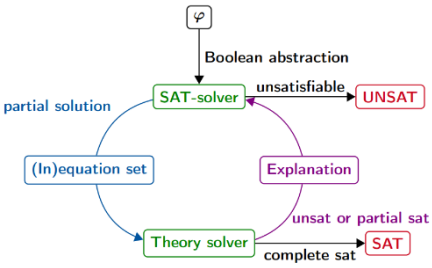
Check whether all **TRUE** theory constraints are satisfied:
No, $\underbrace{p_2 = 0}_{a_2} \wedge \underbrace{p_2 \geq 5}_{a_6}$ are conflicting ⇒ One of them must be false ⇒ Add new clause $(\neg a_2 \vee \neg a_6)$
⇒ $\varphi := \varphi \wedge (\neg a_2 \vee \neg a_6)$

Learn the Boolean Abstraction:
$\varphi_{abs} = (a_1 \vee a_2 \vee a_3) \wedge a_4 \wedge (a_5 \vee a_6) \wedge a_7 \wedge a_8 \wedge a_9 \wedge (\neg a_3 \vee \neg a_7) \wedge (\neg a_2 \vee \neg a_6)$
Conflict Resolution: Clause is already asserting, since only a_6 is from the current DL.
Goto Step 2.

SAT-Solving gives us:
$a_1 = a_3 = a_6 = 0$ $\quad$ $a_4 = a_7 = a_8 = a_9 = a_2 = a_5 = 1$

Check whether all **TRUE** theory constraints are satisfied:
Yes, i.e., $p_1 = 90, p_2 = 0, p_3 = 10$

5.2 Less-Lazy SMT-Solving



In Less-Lazy SMT-Solving, also *partial solutions* can be tested against the theory. This usually happens after each DL. For this to work, the theory solver must fulfil three requirements:

1. *Incrementality*
   In Less-Lazy SMT-Solving we incrementally extend the set of constraints. For a good performance, the Theory Solver should re-use of previous (theory) satisfiability checks for the check of extended sets.
   ⇒ Check shouldn't be re-done from ground up each time.
2. (Preferably minimal) *infeasible subset*
   Compute a (preferably minimal) reason for unsatisfaction
   (Because finding the minimum is expensive, we try to find a minimal subset instead)
3. *Backtracking*
   The theory solver should be able to remove constraints in inverse chronological order.

5.3 Full- and Less-Lazy SMT-Solving for Equality Logic

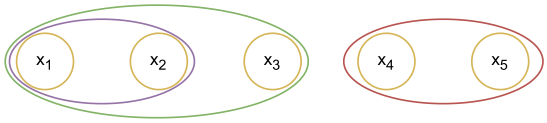
Basic Idea

1. In Equality Logic, we have no function symbols, but only equalities and disequalities
2. For SMT-Solving, in the first step the SAT-solver decides, which equalities and disequalities should hold, and which shouldn't. Thus, we only have sets of (dis-)equalities that should hold and sets of those who shouldn't.
3. Thus, we can directly construct the *equivalence classes* (defined by the equivalences' transitivity, reflexivity. and transitivity)
4. If variables are in the *same equivalence class*, they *MUST be equal*. If they are in different classes, they *MUST NOT necessarily be equal* (*but might still be equal*)

Constructing Equivalence Classes

We start with the finest equivalence partition (= each variable has its own class), and then merge them together iteratively. For example, consider φ^E: $x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge x_5 \neq x_1$

Start with finest partition. Then merge x_1, x_2; Merge x_2, x_3; Merge x_4, x_5.



5.3.1 Full-Lazy SMT-Solving for EQ + UF

Algorithm: Full-Lazy SAT-Check for EQ + UF (Theory Solver)

1. *Initial partition*: Each variable in V has its own equivalence class.
2. *Assure transitivity for equality*: For each input equation $t = t'$, if the equivalence classes $[t]$ and $[t']$ of the two sides differ, then merge them. 等式两边的equivalence class 合并
   (i.e., given $\{x_1, x_2\}, \{x_2, x_3\}$ and $x_2 = x_3$ this will be merged to $\{x_1, x_2, x_3\}$)
3. *Assure functional congruence* for uninterpreted functions: Recursively merge the equivalence classes $[F(t)]$ and $[F(t')]$ for all $[t] = [t']$.
   Intuitively: If t, t' are in the same equivalence class, and we have $F(t)$ and $F(t')$, they must also be in the same equivalence class. The "recursive" part is necessary for nested things like $G(F(x)) \neq G(F(y)) \wedge x = y$. 对于uninterpreted function, 同一函数如果参数在equivalence class, 则对应的函数值也是同一个equivalence class
4. For each disequation $(t \neq t') \in E$, if the equivalence classes of the two sides coincide then Return UNSAT. 对于不等式, 如果不等式两边的equivalence class相等, 则为unsat
   (i.e., $\{x_1, x_2\}, \{x_2, x_3\}$ and $x_1 \neq x_2$ is UNSAT since x_1, x_2 both are in equivalence class 1)
5. Return SAT

Algorithm: Full-Lazy Get Explanation for EQ + UF (Theory Solver)

1. Construct the $E_=$-Graph (considering only equalities)
2. In the graph we search for the *shortest path* between the nodes of the variables of the conflicting disequality. The equations corresponding to this path together with the conflicting disequality form the minimal infeasible subset.

Example: Full-Lazy SMT-Solving for EQ + UF

Apply Full-Lazy SMT-Solving for the following formula:

$$\varphi^{EQ} := \underbrace{x_1 = x_2}_{a_1} \wedge (\underbrace{\neg x_1 = x_4}_{\neg a_2} \vee \underbrace{\neg x_2 = x_3}_{\neg a_3}) \wedge \underbrace{x_1 = x_3}_{a_4}$$

For the SAT-Solver, use the ordering $a_1 < \cdots < a_n$ and assign false first.

Boolean Abstraction: $a_1 \wedge (\neg a_2 \vee \neg a_3) \wedge a_4$

#1 SAT-Solver

$DL0: a_1: 1, a_4: 1$
$DL1: a_2: 0$
$DL2: a_3: 0$

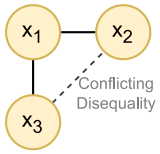
对于设为a1的等式, 如果sat结果为a1, 则需要输送给theory solver; 如果sat结果为¬a1, 则不需要输送给theory solver.
同样的, 对于设为¬a1的等式, 如果sat结果为¬a1, 则需要输送给theory solver; 如果sat结果为a1, 则不需要输送给theory solver.
即: 前后不同的变量不需要考虑

#1 Theory Solver

Creating & Merging equivalence classes for the TRUE constraints (a_1, a_4):
$\langle x_1 \rangle = \{x_1, x_2, x_3\}, \langle x_4 \rangle = \{x_4\}$
We have two inequalities - $x_1 \neq x_4$ (valid) and $x_2 \neq x_3$ (invalid!). Thus, we must find the minimal conflicting set:

Get an Explanation for the conflict

Construct the E-Graph with all equations containing the variables in $\langle x_1 \rangle$. The minimal infeasible subset is defined by the shortest path between the nodes of the variables in the conflicting disequality (x_2, x_3) and the conflicting disequality itself. Thus, our minimal infeasible subset is: $\{\underbrace{x_2 \neq x_3}_{\neg a_3}, \underbrace{x_1 = x_2}_{a_1}, \underbrace{x_1 = x_3}_{a_4}\}$



#2 SAT-Solver

Exclude the assignment corresponding to the infeasible subset by considering the conflicting clause $(\neg a_1 \vee a_3 \vee \neg a_4)$. The conflict is already asserting, so we can skip the resolution step.
We backtrack to the second-highest decision level ($DL0$) and apply BCP:
$\varphi = a_1 \wedge (\neg a_2 \vee \neg a_3) \wedge a_4 \wedge (\neg a_1 \vee a_3 \vee \neg a_4)$
$DL0: a_1: 1, a_4: 1, a_3: 1, a_2: 0$ (Using propagation)

#2 Theory Solver

The equivalence classes are the same. However, we excluded a_3: $(x_2 \neq x_3)$. Thus, our only disequality is $x_1 \neq x_4$ which is still valid.

We've found a complete assignment that is consistent with our theory ⇒ return SAT.

5.3.2 Less-Lazy SMT-Solving

Remember the requirements:

1. Incrementality
   Adding new equation ⇒ Update the partition to check the previously added disequations.
   Add new disequation ⇒ Check the satisfiability of the new disequation.
2. (Preferably minimal) infeasible subset
   A conflict appears when a disequation $t \neq t'$ cannot be true together with current equations;
   Build the set of this disequations $t \neq t'$ and (a minimal number of) equations that imply $t = t'$ by transitivity and congruence
   (By keeping track of the equations that imply $t = t'$, we remember all equations that would cause a conflict with $t \neq t'$, and can mutually exclude them by adding new constraints)
3. Backtracking
   Remember computational history.

Algorithm: Less-Lazy SMT-Solving for EQ + UF (addEquation – Theory Solver)

1. Remember the new equation $E := E \cup \{t_1 = t_2\}$
2. If problem was already unsatisifiable, then it is still unsat. ⇒ return UNSAT
3. Otherwise, merge the equivalence classes and re-check the satisfiability of the disequations.

Algorithm: Less-Lazy SMT-Solving for EQ + UF (addDisequation – Theory Solver)

1. Remember the new equation $E := E \cup \{t_1 \neq t_2\}$
2. If problem was already unsatisfiable, then it is still unsatisfiable ⇒ return UNSAT
3. Otherwise, check the satisfiability of the new disequation.

Algorithm: Less -Lazy SMT-Solving for EQ + UF (getExplanation – Theory Solver)

Same as for Full-Lazy, see Algorithm: Full-Lazy Get Explanation for EQ + UF (Theory Solver)

Algorithm: Less -Lazy SMT-Solving for EQ + UF (backtrack – Theory Solver)

1. Remove the minimal infeasible subset S from E
2. Re-Check for SAT using Algorithm: Full-Lazy SAT-Check for EQ + UF (Theory Solver)

(There are much more elaborate ways to backtrack, but those weren't part of our lecture)

Example: Less-Lazy SMT-Solving for EQ + UF

Apply Full-Lazy SMT-Solving for the following formula:

$\varphi^{EQ} := \underbrace{x_1 = x_2}_{a_1} \wedge (\underbrace{\neg x_1 = x_4}_{\neg a_2} \vee \underbrace{\neg x_2 = x_3}_{\neg a_3}) \wedge \underbrace{x_1 = x_3}_{a_4}$

For the SAT-Solver, use the ordering $a_1 < \cdots < a_n$ and assign false first.

Boolean abstraction: $\varphi_{abs} = a_1 \wedge (\neg a_2 \vee \neg a_3) \wedge a_4$

#1 SAT-Solver

$DL0$: $a_1 : 1, a_4 : 1$

每个DL, SAT-solver的结果都传给theory solver

#1 Theory Solver

Creating & Merging equivalence classes for the TRUE constraints (a_1, a_4):

$\langle x_1 \rangle = \{x_1, x_2, x_3\}$

As there is no disequality, no conflict can occur ⇒ continue.

#2 SAT-Solver

$DL0$: $a_1 : 1, a_4 : 1$

$DL1$: $a_2 : 0$

#2 Theory Solver

The equivalence classes stay the same because no new equality has been added.

However, we have one disequality $a_2 : (x_1 \neq x_4)$, which is satisfied ⇒ continue.

#3 SAT-Solver

$DL0$: $a_1 : 1, a_4 : 1$

$DL1$: $a_2 : 0$

$DL2$: $a_3 : 0$

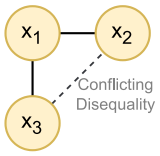
#3 Theory Solver

The equivalence classes stay the same because no new equality has been added.

But the disequality $a_3 : (x_2 \neq x_3)$ is violated because x_2, x_3 are in the same equivalence class $\langle x_1 \rangle$.

#3 Find Minimal Infeasible Subset

Construct the E-Graph with all equations containing the variables in $\langle x_1 \rangle$. The minimal infeasible subset is defined by the shortest path between the nodes of the variables in the conflicting disequality (x_2, x_3) and the conflicting disequality itself. Thus, our minimal infeasible subset is: $\{\underbrace{x_2 \neq x_3}_{\neg a_3}, \underbrace{x_1 = x_2}_{a_1}, \underbrace{x_1 = x_3}_{a_4}\}$



#4 SAT-Solver

Exclude the assignment corresponding to the infeasible subset by considering the conflicting clause $(\neg a_1 \vee a_3 \vee \neg a_4)$. The conflict is already asserting, so we can skip the resolution step. We backtrack to the second-highest decision level ($DL0$) and apply BCP:

$\varphi_{abs} = a_1 \wedge (\neg a_2 \vee \neg a_3) \wedge a_4 \wedge (\neg a_1 \vee a_3 \vee \neg a_4)$

$DL0$: $a_1 : 1, a_4 : 1$, by propagation we get $a_3 : 1$; Propagating again gives us $a_2 : 0$

a3此时和赋值不一样(不是¬a3, 可以不用考虑)

#4 Theory Solver

The equivalence classes are: $\langle x_1 \rangle = \{x_1, x_2, x_3\}$, $\langle x_4 \rangle = \{x_4\}$

Because we have assigned $a_3 : 1, \neg a_3 : 0$ and thus we don't have to consider this disequality.

However, we've decided for $a_2 : 0, \neg a_2 : 1$. Subsequently, $x_1 \neq x_4$ must hold.

x_1 and x_4 are in different equivalence classes ⇒ ok.

Since we have a full assignment that is consistent with the theory, we can return SAT.

6 Linear Real Arithmetic

此处signature为: const, real-value variable, addition +
使用signature 来创建 terms

Quantifier-Free Linear Real Arithmetic (QFLRA)

- Terms (const, x, $t+t$), Constraints ($t < t$) and Formulas ($c, \neg\varphi, \varphi \wedge \varphi, \exists x.\ \varphi$)
  cosntraints: predicate(terms)
- Syntactic sugar: $t_1 = t_2, t_1 \leq t_2, 5x + \frac{2}{3}y < 5$
- We transform $\neg(t < t')$ into $t > t'$, so that we have no negated constraints.

Note: If the input CNF doesn't contain negated constraints, it is sufficient to check the theory consistency for the true constraints for each Boolean solution.

Problem Statement

Decide *satisfiability of conjunctions of linear constraints over the reals* (n variables, k inequations, $\sim_i \in \{=, \leq, <\}$).

i不等式数量 j:变量数量

$$\bigwedge_{1 \leq i \leq k} \sum_{1 \leq j \leq n} a_{ij} x_j \sim_i b_i$$

aij, bi: 常量constant
xj: 变量variable
~i: {=, <, ≤}比较符号 comparison operators

Idea: Eliminate one variable and thus construct a "new, simpler" problem with each iteration

6.1 Gauß Variable Elimination for Linear Real Arithmetic

Intuitive Idea: Isolate x_n to one side and the replace all occurrences of x_n by the found equation.

Variable Elimination

Eliminate variable x_n:

If there exists an equation $\sum_{1 \leq j \leq n} c_{ij} x_j = d_i$ with $c_{in} \neq 0$, then:

- Remove this equation,
- Replace x_n by $\frac{d_i}{c_{in}} - \sum_{j=1}^{n-1} \frac{c_{ij}}{c_{in}} x_{ij}$ in all remaining constraints.

What remains?

Assume that after applying variable elimination as long as possible, we have m weak inequalities in n variables are left. The elimination leads to an equisatisfiable problem $A\vec{x} \leq \vec{b}$.

通过高斯消元后获得只关于一个变量的不等式

The constraint system can be solved using Fourier-Motzkin.

6.2 Fourier-Motzkin Variable Elimination

Algorithm: Fourier-Motzkin Variable Elimination

- For each inequality, we can isolate x_i to one side.
- Afterwards, we can collect all lower and upper bounds:
  - Lower Bounds $\beta_{l1}, \ldots, \beta_{lL}$: inequalities of the form $\beta_j \leq x_i$
  - Upper Bounds $\beta_{u1}, \ldots, \beta_{uU}$: inequalities of the form $x_i \leq \beta_k$

  此时需要让下界的最大值≤ 上界的最小值
  =>因为下上下界不是有序的, 难以区分最值
  =>每一个下界≤ 每一个上界
- For each pair of lower bound β_l and upper bound β_u we have $\beta_l \leq x_n \leq \beta_u$ and can add the new constraint $\beta_l \leq \beta_u$ (which subsequently eliminates x_n)

Note that the *constraint is solvable iff the interval between highest and lowest bound is not empty*, because $\mathbb{R}$ is dense.

weak inequality: 可以包含等号, 即 ≤, ≥
strong inequality: 不包含等号, 即 <, >

Extension to Strict Inequalities

For each pair of lower and upper bounds, if any of them is strict, then we add the constraint $\beta_l < \beta_u$ instead of $\beta_l \leq \beta_u$.

Further Remarks

- This approach does NOT work for linear integer arithmetic, as the $\mathbb{N}$ is not dense.
- This method also does NOT work for non-linear real arithmetic, because in general it might be impossible to isolate x_i to one side.

Example: Fourier-Motzkin Variable Elimination

Use the Fourier-Motzking Elimination to determine whether the following set of constraints is satisfiable: $\{2x - y \leq 8, 2x + y \leq -8, -2x - y \leq 8, -2x + y \leq 8\}$

Eliminate y

$2x - y \leq 8 \quad \Rightarrow 2x - 8 \leq y$
$-2x - y \leq 8 \quad \Rightarrow -2x - 8 \leq y$
$2x + y \leq -8 \quad \Rightarrow y \leq -2x - 8$
$-2x + y \leq 8 \quad \Rightarrow y \leq 2x + 8$

Combine Upper and Lower Bounds:

$2x - 8 \leq -2x - 8 \quad \Rightarrow x \leq 0$
$2x - 8 \leq 2x + 8 \quad \Rightarrow 0 \leq 0$
$-2x - 8 \leq -2x - 8 \quad \Rightarrow 0 \leq 0$
$-2x - 8 \leq 2x + 8 \quad \Rightarrow -4 \leq x$

Combine Upper and Lower Bounds

$-4 \leq 0$

$\Rightarrow$ SAT

6.3 Simplex Algorithm

The *simplex method is exponential*. There are polynomial techniques, however in practice the polynomial methods aren't really used but rather *Simplex Method is used for linear problems*.

Remark: The simplex method was originally designed for solving linear programming problems.

We're only interested in the feasibility problem (= satisfiability problem), which is solved in the first phase of the simplex method.

Assumptions

- No equalities $t_1 = t_2$ (transform into $t_1 \leq t_2 \wedge t_2 \geq t_1$)
- No disequalities $t_1 \neq t_2$ (transform into $t_1 < t_2 \vee t_1 > t_2$)
- No strict inequalities (Simplex can be extended to strict inequalities, but it is a bit involved)

Structure of Explanation

I'll divide the following into two sub-chapters. The first one gives an intuition into the Simplex Algorithm and explains the basic concepts such as Tableau, Pivoting, … The second chapter describes the actual Simplex Algorithm and how you apply it.

6.3.1 Introduction into Simplex and Intuition

General Form required for Simplex

Recall our problem definition $\bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_j \sim_i b_i$

By introducing slack variables $s_1, \ldots, s_m$ with $s_i \sim_i b_i$ we get: $\bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_j - s_i = 0$

This can again be transformed into the *General Form*:

$$A \cdot \begin{pmatrix} \vec{x} \\ \vec{s} \end{pmatrix} = 0 \text{ and } \bigwedge_{i=1}^{m} l_i \leq s_i \leq u_i$$

By introducing the slack variables, we end up with a set of equations (that is relatively easy to solve) and bounds on the slack variables.

松弛变量 slack variable: 可以将不等式转化为等式.
e.g. x1+x2≤0引入松弛变量后转化为 x1+x2+s=0 且s≥0
转化后与原问题等价, 最优解不变

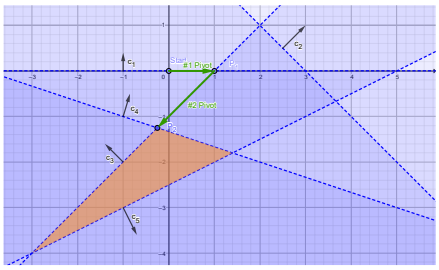
此处将松弛变量si设为等于aij xj, si需要满足原来aij xj的约束条件

Geometric View

The solution set of a conjunction of non-strict linear real-arithmetic constraints is a (possibly empty) convex polyhedron. For a graphical intuition, let's only focus on the 2D-Case (xy). Each constraint can be thought of as a line separating the space into a "valid" and an "invalid" region it. If we have n linear independent constraints, they all separate the space, and we try to find a region where all constraints have a "valid" region. To do so, we can use Simplex Algorithm, where we always start at $(0,0)$ and iteratively try to fulfill more and more bounds. In the example below, initially we satisfy all constraints except c_3 and c_4. After the first pivoting step, we satisfy all constraints but c_3. After the second pivoting step, we've found a spot where we satisfy all the constraints. By pivoting, we try to "move" the point along the boundaries towards a valid region (solution).

Example

Constraint	General Form with Slack Variables	Bound on Slack Variables
$c_1: y < 0$	$s_1 = 0x + 1y$	$s_1 < 0$
$c_2: y < -x + 3$	$s_2 = 1x + 1y$	$s_2 < 3$
$c_3: y < x - 1$	$s_3 = -1x + 1y$	$s_3 < -1$
$c_4: y < -\frac{1}{3}x - \frac{4}{3}$	$s_4 = \frac{1}{3}x + y$	$s_4 < -\frac{4}{3}$
$c_5: y > 0.5x - 2.5$	$s_5 = -0.5x + y$	$s_5 > -2.5$



Note: We're interested in an area where all arrows point AWAY from – like the orange area (i.e., all constraints are satisfied). You can also see the pivot steps.

Data Structures

Throughout the algorithm, Simplex maintains:

- a *tableau* (see below), and
- an *assignment* α to all variables.

Tableau

The tableau is "Simplex' data structure", whose values change throughout the algorithm.

Non-Basic Variables $\mathcal{N}$

Basic Variables $\mathcal{B}$ →

$$\begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array} \begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{pmatrix} 1 & 1 \\ 2 & -1 \\ 1 & 2 \end{pmatrix} \end{array}$$

- For the non-basic variables $\mathcal{N}$, we always ensure that they are on their bounds.
- The basic variables $\mathcal{B}$ don't need to satisfy their bounds.
- The cell values (numbers) indicate which slack variable encodes which sum.
- The idea is, to *maintain some invariants*, but modify the tableau such that we get close and closer to the solution (if any exists)
  - $A\vec{x} = 0$
  - All non-basic variables ($\mathcal{N}$) *satisfy their bounds*

  If the invariants don't hold ⇒ Pivot!
- Originally, these equations hold if we assign all variables $= 0$

Pivoting

Pivoting essentially just swaps a Basic with a Non-Basic Variable. Afterwards, we must adjust all relations accordingly.

Complex Way

1. Isolate the Non-Basic variable
   I.e., pivoting y with s_1 in $s_1 = 2x + y$ would give us $y = s_1 - 2x$
2. Update all other rows accordingly, by plugging in the isolated equation.

Slightly more Convenient Way

Pivoting B_i with NB_j

	…	NB_j	…	NB_l	…
…					
B_i		a_{ij}		a_{il}	
…					
B_k		a_{kj}		a_{kj}	
…					

→

	…	NB_j	…	NB_l	…
…					
B_i		$\frac{1}{a_{ij}}$		$-\frac{a_{il}}{a_{ij}}$	
…					
B_k		$\frac{a_{kj}}{a_{ij}}$		$a_{kj} - \frac{a_{kj}a_{il}}{a_{ij}}$	
…					

6.3.2 The actual Simplex Algorithm

Algorithm: Simplex Algorithm

1. Transform the system into the general form $A \cdot \begin{pmatrix} \vec{x} \\ \vec{s} \end{pmatrix} = 0$ and $\bigwedge_{i=1}^{m} l_i \leq s_i \leq u_i$.
2. Construct the initial tableau and determine a fixed order on the variables (Blant's Rule).
3. If the bounds of all basic variables are satisfied by α, return SAT.
4. Otherwise, assume the basic variable x_i that violates its bounds with $\alpha(x_i) < l_i$.
5. Find the first suitable non-basic variable x_j. Suitable means, that we have "some space left, to adjust it". For example, suppose $\alpha(x_i) < l_i$. To fulfil the constraint, we must increase $\alpha(x_i)$ by either:
   - *Increasing* $\alpha(x_j)$ if the coefficient a_{ij} of x_j is *positive*. To be able to increase it, additionally $\alpha(x_j) < u_j$ must hold – so x_j isn't already at the upper bound.
   - *Decrease* $\alpha(x_j)$ if the coefficient a_{ij} of x_j *negative*. To be able to decrease it, additionally $\alpha(x_j) > l_j$ must hold – so x_j isn't already at the lower bound.
6. If there is no suitable variable, return UNSAT.
7. Otherwise, *pivot* x_i and x_j.
8. Goto step 3.

For a complete example, refer to "SAT-Checking Sample Task.pdf"

Some Remarks

- Only additional variables have bounds, and those are static.
- Additional variables enter the base only on extreme points.
- When entering the base, they shift towards the other bound and possibly cross it.
- It cannot happen that we pivot x_i, x_j followed by x_j, x_i (since x_i wouldn't be suitable)

Termination

- Termination is NOT guaranteed – there might be bigger circles than 1.
- To achieve completeness, we use *Bland's rule:*
  1. Determine a *total order on the variables.*
  2. Choose the first basic variable that violates its bounds, and the *first non-basic suitable variable for pivoting.*

6.3.3 Simplex as a Theory Solver in SMT

Full-Lazy SMT

Everything except the Theory Solver is as before (Boolean Abstraction, SAT-Solving, Explanations)

Theory Solver: 只考虑经过sat后和赋值一样的变量(e.g.如果赋值为¬a=>x1<x2, 则经过sat后¬a为赋值一样要考虑,a为赋值不一样不需考虑)

Use Simplex Algorithm which only considers the *true theory constraints* (=SAT-Solver assigned "1").

- Convert all true theory constraints into the "General Form" utilizing slack variables, create a Tableau, and apply the Simplex Algorithm.
- After the Simplex Algorithm has finished, we can either have:
  - No conflicts ⇒ SAT
  - Have Conflicting clauses a_i, a_j – then add the new clause $(\neg a_i \vee \neg a_j)$ and repeat.

Less-Lazy SMT

Everything except the Theory Solver is as before (Boolean Abstraction, SAT-Solving, Explanations)

Theory Solver:

Use Simplex Algorithm which considers *ALL theory constraints*, but *only "activates" the bounds of true theory constraints*. The rest is pretty much identical to Full-Lazy SMT using Simplex.

This gives us:

- Minimal Infeasible Subsets: Same as Full-Lazy-SMT (constraints corresponding to the basic variable of the contradictory row and all non-basic variables with non-zero coefficients in this row are together unsatisfiable)
- Incrementality (since we always use the complete tableau, but just activate/deactivate bounds)
- Backtracking (by just deactivating bounds)

7 Linear Integer Arithmetic

An *integer linear system* has basically the same definition as real linear systems but of type integer…

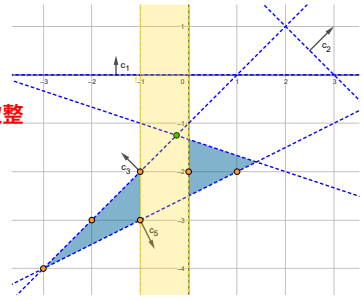
What is the Problem of Integer Solutions?

Reals are dense – Integers aren't. Thus, it can happen, that "there is no number in between" and we cannot find any solution.

7.1 Branch and Bound

Idea

- We search for a real solution (i.e., using simplex)
- If the solution is not integer, we
  - split the search space (*branch*), branch 划分:≤向下取整 或 ≥向上取整
  - exclude the points that are non-integer (*bound*), and
  - search the two halves again. bound : branch 排除的区域不取值 即: 向上取整≤ ≤向上取整 不取值
    For this, start at *the largest integer below the found solution* and the *smallest integer above the found solution*)



The green point illustrates the found solution, the yellow part shows the "excluded" part, and orange dots indicate valid solutions (although unknown to the algorithm).

Algorithm: Branch and Bound

1. Find a solution using the *relaxed problem* (with possibly real solution), i.e., using simplex.
2. if solution doesn't exist: return UNSAT // no real solution ⇒ no integer solution exists
3. else if solution exists and is integral: return SAT
4. else: // Branch and Bound
   - Select a variable v that is assigned a non-integral value r
     Now we make a recursive call and add a new constraint to limit the search space to start at the next-closest integer above/below the non-integer value r 即加上条件: ≤向下取整 或 ≥向上取整
   - if `Branch-and-Bound`($S \cup \{v \leq \lfloor r \rfloor\}$) == SAT: return SAT // Lower Branch (向下取整)
   - else if `Branch-and-Bound`($S \cup \{v \geq \lceil r \rceil\}$) == SAT: return SAT // Upper Branch (向上取整)
   - else: return UNSAT

As the algorithm could loop/recurse forever, *Branch and Bound is incomplete*.

poly: many
nomials: terms polynomials: many terms e.g. (2xy^2 +5-3x)

8 Non-Linear Real Arithmetic

Non-Linear Real Arithmetic

Basically, just Polynomials with constraints $(<, =, >)$ and formulas $(\varphi \wedge \varphi, \neg\varphi)$.
In general, SAT-Checking algorithms for Non-Linear Real Arithmetic have exponential complexity.

Approaches we'll look at include:

- Interval Constraint Propagation (ICP) — Incomplete (but cheap)
- Subtropical Satisfiability — Incomplete
- Virtual Substitution (VS) — Incomplete
- Cylindrical Algebraic Decomposition (CAD) — Complete

Although the first three methods are incomplete and not necessarily find a solution. However, they can still help us to narrow down the search space and thus reduce the time CAD needs.

8.1 Interval Constraint Propagation (ICP)

8.1.1 Interval Arithmetic

Definition of an Interval

An *interval* $A = [\underline{A}; \bar{A}]$ with *lower bound* $\underline{A} \in \mathbb{R} \cup \{-\infty\}$ and *upper bound* $\bar{A} \in \mathbb{R} \cup \{+\infty\}$ denotes the set closed connected set $[\![A]\!] = \{v \in \mathbb{R} | \underline{A} \leq v \leq \bar{A}\}$.

Terminology

- If $\underline{A} \neq -\infty$ and $\bar{A} \neq \infty$ we call A *bounded* (otherwise *unbounded*).
- The *width/diameter* of an interval A is $\bar{A} - \underline{A}$ (or $+\infty$ if A is unbounded)

Computing with Infinity

Note: $-\infty + \infty$ is NOT defined, the rest should be pretty easy

Addition and Subtraction is obvious (i.e., $a + \infty = +\infty$, $+\infty - b = \infty$, or $-\infty - (+\infty) = -\infty$)
Multiplication: $0 \cdot \pm\infty = 0$, $a \cdot \infty$ is always $\pm\infty$ (depending on a's sign), otherwise $a \cdot b$
Division: 0 if the divisor (b) is $\pm\infty$ and otherwise a/b.

Interval Arithmetic

Honestly, I wouldn't recommend learning these rules by heart. The key to understand the idea behind them: The lower bound should be the smallest possible value, and the upper bound the largest respectively. By this, you can easily derive the rules on the fly (i.e., for subtracting, we take "smallest –largest" as the lower interval and "largest – smallest" for the upper interval)

If one interval is empty / division by zero or something, we just return $[1; 0] = \emptyset$.

Addition: $A + B = [\underline{A} + \underline{B}; \bar{A} + \bar{B}]$

Subtraction: $A - B = [\underline{A} - \bar{B}; \bar{A} - \underline{B}]$

Multiplication: $A \cdot B = [\min(\underline{A} \cdot \underline{B},\ \underline{A} \cdot \bar{B},\ \bar{A} \cdot \underline{B},\ \bar{A} \cdot \bar{B})\ ;\ \max(\underline{A} \cdot \underline{B},\ \underline{A} \cdot \bar{B},\ \bar{A} \cdot \underline{B},\ \bar{A} \cdot \bar{B})]$

Square: $A^2 = (A \cdot A) \cap [0;\ +\infty)$ (to rule out negative values)

Square Root: $\pm\sqrt{A} = \left[-\sqrt{\bar{A}}; \sqrt{\bar{A}}\right]$ if $\underline{A} \leq 0 \leq \bar{A}$; $= \left[-\sqrt{\bar{A}}, -\sqrt{\underline{A}}\right] \cup \left[\sqrt{\underline{A}}, \sqrt{\bar{A}}\right]$ if $0 < \underline{A} \leq \bar{A}$

Division: $\mathbf{0 \notin B}$: $A \div B = A \cdot \frac{1}{B} = A \cdot \left[\frac{1}{\bar{B}}; \frac{1}{\underline{B}}\right]$

If $\mathbf{0 \in B}$, division is a bit trickier since the values can go to $\pm\infty$ ($\lim_{x \to \pm\infty} \frac{1}{x} = \pm\infty$).

Here you could learn the "division table" by heart, or also use common sense and try to understand what happens, i.e., if you have $[1;\ 2] \div [-1; 1] = [-\infty;\ -1] \cup [1;\ +\infty]$. Again: Try to observe what values might appear when dividing different values in the interval.

8.1.2 Interval Contraction Methods

Given an interval domain for each variable, polynomials can be evaluated to an interval value. However, the *interval evaluation of polynomials will be in general over-approximative* (due to different occurrences of the same variable). Subsequently, we need to contract the interval.

Contraction 1: Using Interval Arithmetic

Algorithm: Contraction 1 – Preprocessing

For all constraints $e_1 \sim e_2$ with $\sim \in \{<, \leq, =, \geq, >\}$:

1. Bring $e_1 \sim e_2$ to the normal form $r_1 m_1 + \cdots + r_k m_k \sim 0$
2. Replace each non-linear monomial m_i (i.e., xy^2) with a fresh variable h_i,
   a) add an equation $h_i - m_i = 0$, and
   b) initialize the bounds of h_i to the interval we get when we substitution the variable bounds in m_i and evaluate the result using interval arithmetic

Algorithm: Contraction 1 – Method

Choose a constraint $c \in C$

1. Bring c to a form $x \sim e$ where e does not contain x (possible due to preprocessing).
2. Replace all variables in e by their current bounds.
3. Apply interval arithmetic to evaluate the right-hand-side to a union of intervals.
4. For each interval B in that union, derive from the current bound A for x and the computed bound B for e a new bound on x.

Example: Contracting using Interval Arithmetic

Use Interval Arithmetic to contract the constraints $c_1: y = x$, $c_2: y = x^2$ *with the initial intervals* $x \in [1;3], y \in [1;2]$

Note that we don't need to add h_i (preprocessing), since we only have linear constraints. I.e., if we would have $2x^2y - 5y + xy^2z < 0$, we would transform it and add $h_1 = x^2y, h_2 = xy^2z$).

Contract x using c_2:

(c_2, x): $x = \pm\sqrt{[1;2]} = [-\sqrt{2}; -1] \cup [1; \sqrt{2}]$ $\rightarrow x \in [1;3] \cap ([-\sqrt{2}; -1] \cup [1; \sqrt{2}]) = [1; \sqrt{2}]$

Contract y using c_1

(c_1, y): $y = [1; \sqrt{2}]$ $\rightarrow y \in [1;2] \cap [1; \sqrt{2}] = [1; \sqrt{2}]$

Contraction 2: Newton Contraction

Algorithm: Newton Contraction – Preprocessing

For all constraints $e_1 \sim e_2$ with $\sim \in \{<, \leq, =, \geq, >\}$:

1. Bring $e_1 \sim e_2$ to the form $e_1 - e_2 \sim 0$ p在这里等于e_1-e_2
2. For each inequation $p \sim 0$ with $\sim \in \{<, \leq, \geq, >\}$ replace p by a fresh variable h,
   c) add an equation $h - p = 0$, and
   d) initialize the bounds of h_i to the interval we get when we substitution the variable bounds in p and evaluate the result using interval arithmetic

Algorithm: Newton Contraction – Method

If we have $x \sim 0$ with $\sim \in \{\leq, <, \geq, >\}$, we use same proceeding as in the first method. Otherwise, assume Function $f(x)$, starting interval A, sample point s, and the constraint c: $f(x) = 0$

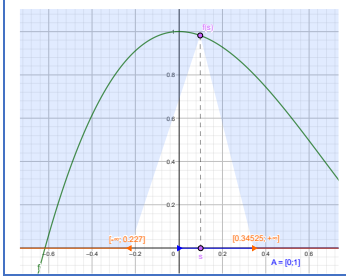
1. Compute $f(s)$, the derivative $f'(x)$, and $f'(A)$
2. Possible roots in A: $R := s - \frac{f(s)}{f'(A)}$
3. Contract the interval by: $A_{new} = A \cap R$

Example: Contracting using Interval Arithmetic

Use Newton's method to contract $f(x) = x^3 - 2x^2 + 1$ *with the sample point* $s = 0.1$ *and starting interval* $A = [0; 1]$.

1. $f(s) = f(0.1) = 0.981$
   $f'(x) = 3x^2 - 4x$
   $f'(A) = 3[0;1]^2 - 4[0;1] = 3\left([0^2; 1^2] \cap [0, \infty)\right) - [0;4] = [0;3] - [0;4] = [-4;3]$
2. $R := s - \frac{f(s)}{f'(A)} = 0.1 - \frac{0.981}{[-4;3]} = 0.1 - 0.981 \cdot \left[-\frac{1}{4}; \frac{1}{3}\right]$
   $= [0.1; 0.1] - ([-\infty;\ -0.24525] \cup [0.327;\ +\infty])$
   $= ([-\infty;\ -0.277] \cup [0.34525;\ +\infty])$
3. $A_{new} = A \cap R = [0.34525; 1]$

Visualization



We start by computing $f'(A)$ – an interval that contains all possible derivatives from our starting interval (and possibly more, as interval arithmetic is an over-approximation!). Afterwards, we use the Newton Method $\left(s - \frac{f(s)}{f'(A)}\right)$ to get the orange interval.
The over-approximation can be contracted by intersecting the new interval with the starting interval. We end up with a smaller (contracted) interval that contains the root.

8.1.3 Global ICP Algorithm

Idea

The original interval can be thought of as a box. By contraction, we can reduce the boxes' size. The union of all these boxes contains all solutions of the original box. If the diameter of a box is less than the fixed *box diameter threshold* D, we apply SAT-Solving (in this case we've narrowed down our solution space "enough" to use more elaborate SAT-Solvers). To ensure termination, we can also split boxes if we make no further progress (= reducing a boxes diameter).

Algorithm: Global ICP

1. Initialize $\mathcal{B} := \{B_0\}$ // $\mathcal{B}$ = set of Boxes
2. If $\mathcal{B}$ is empty, return UNSAT.
3. Choose box $B_i \in \mathcal{B}$ and remove it from $\mathcal{B}$
   a) if diameter of B_i is at most D, then pass on B_i to a complete procedure for satisfiability check (Subtropical, VS, or CAD); if B_i contains a solution then return SAT; else select new box and go to 2.
   b) else if the contraction condition for B_i holds, then try to reduce this box, add the resulting box(es) to $\mathcal{B}$, and go to 2.
   c) else split the box into two halves, add them to $\mathcal{B}$, and go to 2.

Algorithmic Aspects

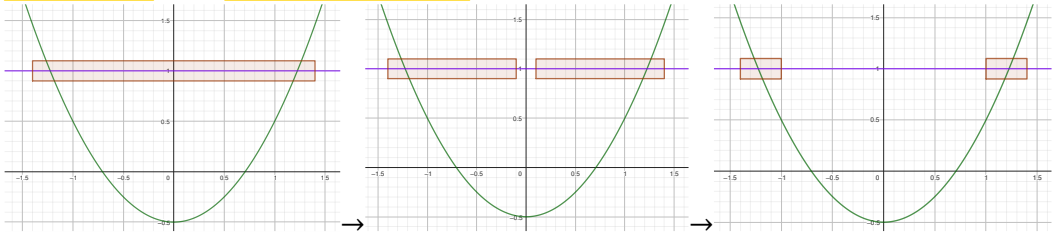
- We need a *heuristic to choose Contraction Candidates*:

  Relative Contraction: $gain_{rel} = 1 - \frac{D(new)}{D(old)}$

  Use Weights: $W_{k+1}^{(ij)} = W_k^{(ij)} + \alpha\left(gain_{rel,k+1}^{(ij)} - W_k^{(ij)}\right)$

  (Note that finding the optimal CC is as hard as solving the SAT-Problem itself)

- We must *assure termination*:
  Split the box if we don't make progress (weight of all CCs is below the threshold)



- ICP doesn't behave well on linear constraints:
  Make use of linear solvers (i.e., simplex) and e.g. separate linear and non-linear constraints.
- ICP needs to work incrementally & return an explanation for unsatisfiable problem:
  Store the search history in a tree-structure.

8.2 Subtropical Satisfiability

Method is incomplete; We either quickly find a positive solution or return unknown.

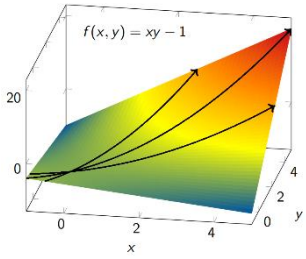
Notation

Monomial	Product of variables (i.e., xy^2)
Term	Product of a coefficient and a monomial (i.e., $2xy^2$)
Polynomial	Sum of terms with pairwise different monomials (i.e., $2xy^2+3u^3$)
Polynomial Constraint	$p \sim 0, 0 \in \{<, \leq, =, \geq, >\}$ (i.e., $2xy^2 + 3u^3vz^2 - 5 < 0$)
Univariate	Polynomial with one variable (otherwise: *Multivariate*)

Idea

即,使用newton polytope找到能使 正数项/负数项 占主导的方向, 然后在此方向上增大变量的取值, 进而使得整个多项式>0/<0

- If I move into a certain direction, one monomial will become larger than all others ("dominating"). So, we must "only" find out this direction $\vec{n}$, which can be done using a "*Newton Polytope*".
- If we have found such a direction, the idea is to map the function to the *exponential space*. So instead of $p(x,y)$ we use $p(a^{n_1}, a^{n_2})$. Increasing a for a fixed direction $\vec{n}$ gives exponentially high weight to this direction. Eventually (for large enough a), the "direction" becomes dominating.



The Frame of a Multivariate Polynomial

The frame basically just stores the exponents of each term. For example:

$p(x,y) = \underbrace{y}_{+} + \underbrace{2xy^3}_{+} - \underbrace{3x^2y^2}_{-} - \underbrace{x^3}_{-} - \underbrace{4x^4y^4}_{-}$

$frame^{+}(p) = \{(0,1),(1,3)\}, frame^{-}(p) = \{(2,2),(3,0),(4,4)\}, frame = frame^{+} \cup frame^{-}$

frame+为正数项的次幂数, frame-为负数项的次幂数

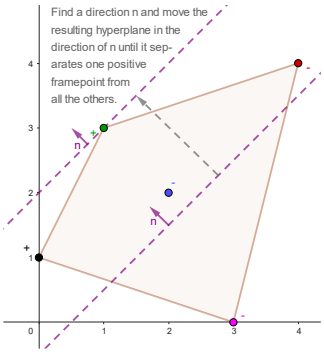
Newton Polytope

Convex Hull of a polynomial is the convex hull of its frame (= smallest convex set containing all frame points).

A *face* is basically just a "border" of the polytope, a *vertex* V is just an edge.

face为凸包在某一方向上的边界

Our goal is, finding a hyperplane ($nx^T = b$) that *separates a positive frame point from all others*.



Convex Hull凸包: 给定二维平面上的点集，凸包就是将最外层的点连接起来构成的凸多边形，它能包含点集中所有的点.
点集Q的凸包（convex hull）是指一个最小凸多边形，满足Q中的点或者在多边形边上或者在其内。
用一个扎头发的橡皮筋套住Q，橡皮筋自然收缩后圈住的部分就是Q的凸包

Algorithm: Subtropical Satisfiability

Handle Strict Inequalities $p > 0$ 求使得p>0的变量取值

1. Construct $frame^+(p), frame^-(p)$
2. Determine halfplane encoding that separates the positive frame point from all others.
3. Find a solution $\vec{s}$ to the encoding (i.e., using Simplex)
4. $a := 2$, exponentially increase a until $p(a^{s_1}, a^{s_2}) > 0$ (which must happen at some point).
5. a^{s_1}, a^{s_2} is our solution.

Handle Equalities $p = 0$

1. If $p(1, \dots, 1) = 0$: we already have a solution
   If $p(1, \dots, 1) > 0$, consider $-p = 0$ instead of $p = 0$.
   Assume $p(1, \dots, 1) < 0$ 已知p(1,...1)<0, 欲求p=0的变量取值 ==>求出使得p>0的变量取值, p=0的解即在两者之间
2. Find a (positive) solution v for $p(v) > 0$
3. By the Intermediate Value Theorem (a continuous function with positive and negative values has a root) we're able to construct a root of p
   a) Construct a line equation $l(t) = (Start + t(End - Start))$ from $(1, \dots, 1)$ to v
   b) Plug in the x- and y-coordinates of $l(t)$ into p to obtain p^* 求出x, y对应值带入p得到p*
   c) Find the roots of p^* (i.e., by using bisection)
   d) Construct the root of p by plugging in the root t_0 with $0 \le t_0 \le 1$ into $p(\quad)$

Note: $p < 0$ can be transformed to $-p > 0$ and all inequalities will be handled as strict inequalities. $(1, \dots, 1)$ is fixed arbitrary by convention.

Example: Subtropical SAT with Inequalities

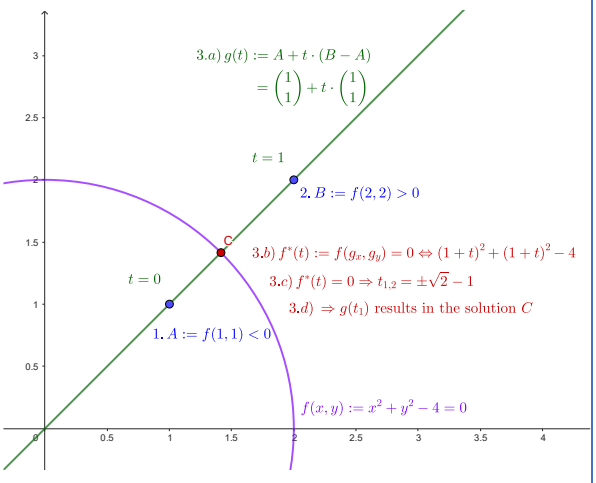
Determine whether $p(x, y) = 2x^2y - 3x - 4y > 0$ is SAT using Subtropical Satisfiability.

1. $frame^+(p) = \{(2,1)\}, frame^-(p) = \{(1,0), (0,1)\}$
2. $2n_x + 1n_y > b \wedge 1n_x + 0n_y < b \wedge 0n_x + 1n_y < b$
3. SAT-Solving (i.e., using simplex) could give us: $n_x = 1, n_y = 0$
4. $p(a^{n_x}, a^{n_y})$
   $p(2^1, 2^0) = -2 < 0$ $\qquad$ $p(4^1, 4^0) = 16 > 0$
5. Thus, one solution is $x = 4^1 = 4, y = 4^0 = 1$

Example: Subtropical SAT with Equalities

Determine whether $p(x, y) = x^2 + y^2 - 4 = 0$ is SAT using Subtropical Satisfiability.

1. $p(1,1) = -4 < 0$
2. $p(2,2) = 4 > 0$
3. Constructing the root:
   a) $g(t) = \begin{pmatrix}1\\1\end{pmatrix} + t\left(\begin{pmatrix}2\\2\end{pmatrix} - \begin{pmatrix}1\\1\end{pmatrix}\right)$
   Thus, $x = 1 + t, y = 1 + t$
   b) $p^*(t) := p(1 + t, 1 + t)$
   $= (1 + t)^2 + (1 + t)^2 - 4$
   $= 2t^2 + 4t - 2$
   c) $p^*(t) = 0 \Rightarrow t_{1,2} = \pm\sqrt{2} - 1$
   d) Plug the root back into g (Note: we must pick the root with $0 \le t_i \le 1$)
   $C := g(\sqrt{2} - 1) = \begin{pmatrix}\sqrt{2}\\\sqrt{2}\end{pmatrix}$



8.3 Virtual Substitution

Idea:

- Use the *solution equations* (well-known for univariate polynomials) also for multivariate ones.
  The roots divide the function into sign-invariant regions (that is, the sign doesn't change)
- Generate *test candidates* based on the solution.
  By applying the solution equation, we computed the roots, that define our sign-invariant regions. Based on this, we can generate test candidates to determine whether our constraint is satisfied. For example, if we have $x^2 - x \geq 0$, we can compute the roots, and test whether the constraint is satisfied either for one of its roots or for $-\infty$.
- Apply *virtual substitution*.

Solution Equation(s) for Univariate Polynomials

Case	Real Root	Side Condition
Constant in x	All real numbers	If $a = 0 \wedge b = 0 \wedge c = 0$
Linear in x	$\xi_0 = -\frac{c}{b}$	If $a = 0 \wedge b \neq 0$
Quadratic in x, 1st solution	$\xi_1 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	If $a \neq 0 \wedge b^2 - 4ac \geq 0$
Quadratic in x, 2nd solution	$\xi_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	If $a \neq 0 \wedge b^2 - 4ac > 0$

What about Multivariate Polynomials?

- We can just treat them as univariate polynomials with polynomial coefficients
  i.e., $5yx^2 + 3zx \in \mathbb{Z}[y,z,x] \rightsquigarrow 5yx^2 + 3zx \in \mathbb{Z}[y,z][x]$

Algorithm: Test Candidates for a Set of Constraints		
For each constraint $p \sim 0$ we add the following test candidates:		
$p = 0, p \leq 0, p \geq 0$:	1. Each real root of p	2. $-\infty$
$p < 0, p > 0, p \neq 0$:	1. Each real root of p plus an infinitesimal ϵ	2. $-\infty$

Note, that we need the infinitesimal, because we want to test "slightly to the left/right" of the root, to account for the strict inequalities.

Graphical Explanation

Given the constraint $x^3 - 5x^2 + 3x > 0$, the roots are given by ξ_1, ξ_2, ξ_3. Between those roots, the sign is invariant. Thus, it is sufficient to pick one example from each sign-invariant region. However, because we've a strict inequality, we need to "go a tiny bit to the right". Thus, we test $\xi_1 + \epsilon$, $\xi_2 + \epsilon$, $\xi_3 + \epsilon$. Because we always looked "a bit to the right", we haven't tested the left-most sign-invariant region yet. Thus, we additionally test $-\infty$. By doing so, we observe, that the constraint is satisfiable i.e., using $\xi_1 + \epsilon$.

每次都右移，则最左边的区域没有被test

Example: Test Candidate Generation

Generate all test candidates for $\varphi\big((xy - 1 = 0 \vee y - x \geq 0) \wedge y^2 - 1 < 0\big)$ *to eliminate* y.

As we want to eliminate y, we must only consider terms containing y (all in this case).

- Compute the roots:

  $xy - 1 = 0 \Rightarrow y = \frac{1}{x}$ $\quad$ $y - x \geq 0 \Rightarrow y \geq x$ $\quad$ $y^2 - 1 < 0 \Rightarrow y < \pm 1$

- Derive our Test Candidates:
  a) $-\infty$ (From all constraints)
  b) $\frac{1}{x}$ (From $xy - 1 = 0$, if $x \neq 0$)
  c) x (From $y - x \geq 0$)
  d) $1 + \epsilon$ (From $y^2 - 1 < 0$)
  e) $-1 + \epsilon$ (From $y^2 - 1 < 0$)

Substitute y by the test candidates:
For φ to be satisfiable, it must be satisfiable for at least one of our test candidates. Thus, we can replace y by each of the test candidates, disjunct every equation, and have eliminated y.

$\exists x.\, \exists y.\, \varphi \leftrightarrow \exists x.\, (\varphi[-\infty \,/\!/\, y]) \vee \left(\varphi\left[-\frac{1}{x} \,/\!/\, y\right]\right) \vee (\varphi[x \,/\!/\, y]) \vee (\varphi[1 + \epsilon \,/\!/\, y]) \vee (\varphi[-1 + \epsilon \,/\!/\, y])$

Virtual Substitution of a Variable by a Test Candidate

The only thing left to discuss is, how we substitute a variable. *Standard substitution* could lead to formulas containing $\epsilon, \infty, \sqrt{\ }, \div$ which we can't really handle. *Virtual substitution* generates real algebraic formulas that are semantically equivalent to the application of standard substitution, but these formulas do not contain $\epsilon, \infty, \sqrt{\ }, \div$.

The rules heavily differ between constraint's relation symbol, and the test candidate's type. In the exam, a table would most likely be given and we just need to know how to use it.

Example

Example: Test Candidate Generation

Eliminate y *using test candidate* $-\infty$ *in* $\exists x.\, \exists y.\, \big((xy - 1 = 0 \vee y - x \geq 0) \wedge y^2 - 1 < 0\big)$

$\exists x.\, \Big(\big((xy - 1 = 0)[y \,/\!/\, -\infty] \vee (y - x \geq 0)[y \,/\!/\, -\infty]\big) \wedge (y^2 - 1 < 0)[y \,/\!/\, -\infty]\Big)$

$\Leftrightarrow \exists x.\, \Big(\big((x = 0 \wedge -1 = 0) \vee \big(1 < 0 \vee (1 = 0 \wedge -x \geq 0)\big)\big)$

$\wedge \big(1 < 0 \vee (1 = 0 \wedge 0 > 0) \vee (1 = 0 \wedge 0 = 0 \wedge -1 < 0)\big)\Big)$

$\Leftrightarrow \exists x.\, \big((false \vee false) \wedge false\big)$

$\Leftrightarrow \exists x.\, false$

$\Rightarrow -\infty$ as a test-candidate doesn't satisfy the constraints.

8.4 Cylindrical Algebraic Decomposition

What is the Intuition behind Cylindrical Algebraic Decomposition?

Cylindrical Algebraic Decomposition basically just divides our solution space into several "cylinders"/cells (think of it just as an area for now). Each cell has the property that it covers a region, where all constraints "behalf" the same (i.e., all are valid, one is valid and one is invalid, …) – more formally, each cylinder is a P-sign-invariant region. 即在某一cylinder上,所有多项式P的正负保持不变, 为P-sign-invariant region
Subsequently, it is sufficient to select one sample point out of each cylinder and check it for satisfiability with respect to the constraint and we're done (sounds easy, but constructing the cells is double-exponentially in general).

Cells are…

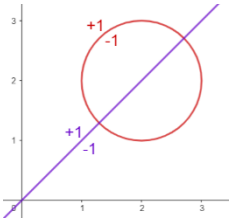
Cylindrical	…ordered into stack-like cylinders that are sign-invariant!
Algebraic	…defined by real-algebraic formulas (i.e., $x + y < 0$)
Decomposition	…disjoint and cover $\mathbb{R}^n$ (the union over all cells is $\mathbb{R}^n$)

Sign-Invariant Regions

Sign of a Polynomial: $sgn(a) := \begin{cases} -1, & a < 0 \\ 0, & a = 0 \\ 1 & a > 0 \end{cases}$

$P = \{p_1, \dots, p_m\}$ is *P-sign-invariant* if $sgn(p_i(a)) = sgn(p_i(b))$ $\forall i \in \{1, \dots, m\}$

(So basically, all $p \in P$ must have the same sign)



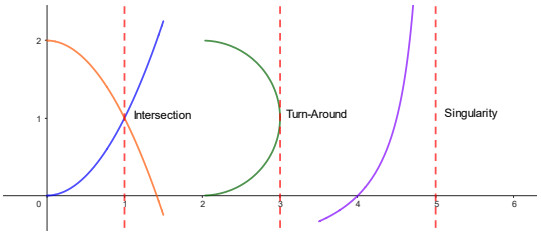
Terminology

- A *decomposition* of $\mathbb{R}^n$ is a finite set C of pairwise disjoint regions in $\mathbb{R}^n$ with $\bigcup_{c \in C} c = \mathbb{R}^n$.
- A decomposition is *semi-algebraic* if each $c \in C$ can be constructed by finite union, intersection, and complementation of solutions sets of polynomial constraints $p \sim 0$ ($p \in \mathbb{Q}$)
- A decomposition is *cylindrical* if either $n = 1$ or the set of the projections of the regions in C to the first $n - 1$ dimensions is a cylindrical decomposition of $\mathbb{R}^{n-1}$
- A *cylindrical algebraic decomposition (CAD)* of $\mathbb{R}^n$ is a cylindrical and semi-algebraic decomposition of $\mathbb{R}^n$, *we call* $c \in C$ *a cell*.
- A CAD for $P \subset \mathbb{Q}[x_1, \dots, x_n]$ is a CND of $\mathbb{R}^n$ whose cells are all P-sign-invariant.

Where do we need to add Cylinders?

Every time the number or order of roots change. This is the case for:

1. The *intersection points* of at least two polynomials ⇒ at least one common root
2. Discriminant of a polynomial: Change in the number and order of roots (*"turn arounds"*)
3. Coefficients: Roots cover divergence points (*singularities*)



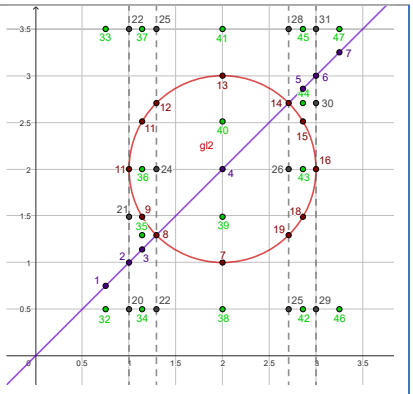
We also call such a cylinder *delineable region* (that is, the number and order of roots is constant)

Example: CAD with 47 Cells – Visualization
Construct the CADs for $P = \begin{pmatrix} (x-2)^2 + (y-2)^2 - 1 \\ x - y \end{pmatrix}$
The figure on the right visualizes all cells of the CAD. - The grey lines must be drawn for: • *each intersection* of at least two different polynomials (ξ_2, ξ_3), and • for *each "turning point"* (root of the discriminant of p) (ξ_1, ξ_4)

We end up with 5 cylinders.

- Each of those cylinders is then separated by the two constraints, such that we get smaller cylinders. Basically, we get one cylinder for each area separated from all others by lines (green cells).
- Additionally, *the lines itself and the intersection of lines form cells by itself*.

⇒ We end up with 47 cells in total and pick one sample out of each cylinder/cell. The cells completely cover the respective intervals, and thus, one sample each is sufficient.



P-delineable Regions (= sign-invariant regions):

$$(-\infty;\underbrace{1}_{\xi_1}),\{\underbrace{1}_{\xi_1}\},(\underbrace{1}_{\xi_1};\underbrace{2-\sqrt{2}/2}_{\xi_2}),\{\underbrace{2-\sqrt{2}/2}_{\xi_2}\},(\underbrace{2-\sqrt{2}/2}_{\xi_2};\underbrace{2+\sqrt{2}/2}_{\xi_3}),\{\underbrace{2+\sqrt{2}/2}_{\xi_3}\},(\underbrace{2+\sqrt{2}/2}_{\xi_3};\underbrace{3}_{\xi_4}),\{\underbrace{3}_{\xi_4}\},(\underbrace{3}_{\xi_4},\infty)$$

Additional explanation for cylinder boundaries:
A cylinder bound is necessary each time the order/number of zeros change. For example, for $x < \xi_1$ we only have one zero. For $x = \xi_1$, we have two zeros, for $\xi_1 < x < \xi_2$ we have three zeros, $x = \xi_2$ collapses down to two zeros, …
However, you can also just remember to draw a boundary for each intersection and each "turning point" (like the left- and right-most points of the red circle)

Representing Real Roots (zeros)

An interval representation (of a real root) is a pair (p, I) of a univariate polynomial p and a non-empty *open interval* $I = (l, r)$, $l, r \in \mathbb{Q} \cup \{-\infty, \infty\}$ such that I contains *exactly one real root of* p.
Representing roots like $\sqrt{2}$ is problematic. This representation helps us to circumvent this. For example, $p := x^2 - 2$ as roots $\pm\sqrt{2}$ and a valid interval representation would be $\big(p, (-\infty, 0)\big)$
Note, that i.e., $\big(p, (-\infty, 0)\big)$ is still a NUMBER – meaning "the root of p in the interval $(-\infty, 0)$.

Cauchy Bound

Assume a univariate polynomial $p = a_k x^k + a_{k-1}x^{k-1} + \cdots + a_0 x^0 \in \mathbb{Q}[x]$. If $\xi \in \mathbb{R}$ is a root of p (i.e., $p(\xi) = 0$), then it must be in the interval $[-C, C]$, or in other terms:

$$|\xi| \leq 1 + \max_{i=0,\dots,k-1} \frac{|a_i|}{|a_k|} := C$$

Small reminder: A polynomial p has between 0 and $\deg(p)$ real roots.

Sturm Sequence

A Sturm sequence for p allows us to count the real roots of p in an interval. For a square-free univariate polynomial p, the *Sturm Sequence* $p_0, p_1, \dots, p_l$ is defined as:

$$p_0 = p, \qquad p_1 = p' \text{ (derivative)}, \qquad p_i = -rem(p_{i-2}, p_{i-1}), \qquad rem(p_{l-1}, p_l) = 0$$

(Where rem is the remainder of the polynomial division of p_{i-2} by p_{i-1}).

Furthermore, $\sigma(\xi)$ denotes the *number of sign changes* (**ignoring zeroes**) in the sequence.
Then for each $a, b \in \mathbb{R}$ with $a < b$, the *number of real roots of* p *in* $(a, b]$ *is* $\sigma(a) - \sigma(b)$.

Example: Cauchy Bound and Sturm Sequence
Compute the number of roots of $p(x) = x^2 + x + 1$
Cauchy Bound: $C = 1 + \max\{\frac{1}{1}; \frac{1}{1}\} = 2$ Because $p(-2) \neq 0$, all real roots are in $I := (-2, 2]$

Sturm Sequence	Values at	
	-2	2
$p_0 = x^2 + x + 1$	+3	+7
$p_1 = 2x + 1$	-3	+5
$p_2 = 0 - \frac{3}{4}$	$-\frac{3}{4}$	$-\frac{3}{4}$
$\sigma(\cdot)$	1	1

Thus, we have $\sigma(a) - \sigma(b) = 0$ real roots in I. Because I was given by the Cauchy Bound, we can further conclude, that p has no real root at all (otherwise, it must have been inside I).

8.4.1 Compute CAD for $\mathbb{R}$ (Univariate Polynomials)

Algorithm: Cylindrical Algebraic Decomposition for Univariate Polynomials (CAD)

1. Compute the *Cauchy Bound* C, $I = [-C, C]$ contains all real roots.
   Split I into $[-C, -C]$, $(-C, C)$, $[C, C]$ (required to compute Sturm-Sequence)
2. Use the *Sturm-Sequence* to count the real roots for each interval.
3. *Split each sub-interval* that contains either *more than one real root of the same polynomial* or *two different roots of two different polynomials*. For (a, b) choose $a < c < b$ with the sub-intervals $(a, c), [c, c], (c, b)$ (in our lecture: always split in the middle)
4. Go back to 1 until no more split in step 3 is possible.

The result is kind of an interval representation of all roots.

Example: CAD for Univariate Polynomials

Apply CAD for the univariate polynomial $\underbrace{x^2 - 2}_{:=p} > 0$

Cauchy Bound:	$C = 1 + \max\{\frac{2}{1}\} = 3$		
Split $[-3;\ 3]$:	$[-3; -3]$,	$(-3; 3)$,	$[3; 3]$
Sturm-Sequence (computation left out):	0,	2,	0
Split $(-3;\ 3)$:	$I_1 := (-3; 0)$,	$I_2 := [0; 0]$,	$I_3 := (0; 3)$
Sturm-Sequence (computation left out):	1,	0,	1

⇒ Finished, as each interval only has one Root. Thus, our CAD is:

$$\{\underbrace{(-\infty; (p_1, I_1))}_{\text{left from } \xi_1}, \underbrace{((p_1, I_1); (p_1, I_1))}_{\text{root } \xi_1}, \underbrace{((p_1, I_1); (p_1, I_3))}_{\text{between } \xi_1 \text{ and } \xi_2}, \underbrace{((p_1, I_3); (p_1, I_3))}_{\text{root } \xi_2}, \underbrace{((p_1, I_3); +\infty)}_{\text{right from } \xi_2}\}$$

Recall, that (p_1, I_1) is a NUMBER – it just means "the root of p_1 in I_1".

Recall: The (Less-Lazy) Theory Solver must satisfy some requirements…

- Incrementality:
  - Compute new Cauchy-bound: $C = \max\{C_{p_1}, C_{p_2}\}$
  - Continue with already existing intervals (found by earlier iterations), and check whether the new polynomial p_2 satisfies the requirements (that is, at most one distinct root in each interval – same roots must be in the same interval)
  - Split as before until all requirements are satisfied.
- Infeasible Subsets:
  - From each interval we take one sample. If a sample doesn't satisfy the condition from the corresponding polynomial, we collect them. In the end, this gives us a "list" of all samples that violate polynomials (= infeasible subset)

8.4.2 Compute CAD for $\mathbb{R}^n$ (Multivariate Polynomials)

CAD Phases in a Nutshell

We haven't really discussed details here, however in a nutshell we can divide it into two phases:

1. Projection Phase
   Just construct the cylinders and cells.
   Afterwards: Compute the roots.
2. Construction / Lifting Phase
   Generate 1D-Samples by plugging in the roots and one sample for each other cell.
   Plugging those samples into 2D-Polynomial yields 2D-Samples.
   Continue until we have samples from $\mathbb{R}^n$.

8.4.3 Further CAD-Construction Examples