1.1 Declarative Sentences

Propositions (Declarative Sentences): a statement that can be declared true or false.

Use Symbols for **atomic** propositions:

- Ex.
  - p = "the train arrived late"
  - q = "there were no taxis at the station"
  - r = "Jane was late for her meeting"

Building Propositions

More complex sentences can be created using connectives:

- Negation ($\neg$p): "not p"
- Disjunction (p $\vee$ q): "p or q"
- Conjunction (p $\wedge$ q): "p and q"
- Implication (p $\rightarrow$ q): "if p then q"

Conventions:

- Operator precedence, $\neg$ (highest), $\wedge$, $\vee$, $\rightarrow$
- Associativity $\rightarrow$ (right), $\wedge$/$\vee$ (doesn't matter)
  - This means everything right of $\rightarrow$ is grouped together

1.2 Natural Deduction

Deducing Propositions

Sequent: $\phi_1, \phi_2, \ldots, \phi,_n \vdash \psi$

- Premises: $\phi_1, \phi_2, \ldots, \phi,_n$
- Conclusion: ψ

A sequent is valid if a proof can be found to make the premises into the conclusion.

*Sequent != Implication

Rules for Conjunction (AND)

AND-INTRO: $\dfrac{\phi \quad \psi}{\phi \wedge \psi}$

AND-ELIM1: $\dfrac{\phi \wedge \psi}{\phi}$

AND-ELIM2: $\dfrac{\phi \wedge \psi}{\psi}$

Rules for Double Negation (NOT NOT)

NOT-NOT-INTRO: $\dfrac{\phi}{\neg\neg\phi}$

NOT-NOT-ELIM: $\dfrac{\neg\neg\phi}{\phi}$

Rules for Implication ($\to$)

IMP-INTRO: $\frac{\phi ... | \psi}{\phi \to \psi}$

IMP-ELIM: $\frac{\phi \quad \phi \to \psi}{\psi}$

- This is also known is Modus Ponens

Rules for Disjunction (OR)

OR-INTRO1: $\frac{\phi}{\phi \lor \psi}$

OR-INTRO2: $\frac{\psi}{\phi \lor \psi}$

OR-ELIM: $\frac{\phi \lor \psi \quad \phi ... | \chi \quad \psi ... | \chi}{\chi}$

Rules for Negation (NOT)

NOT-INTRO: $\frac{\phi ... | \bot}{\neg \phi}$

NOT-ELIM: $\frac{\phi \quad \neg \phi}{\bot}$

Rules for False

FALSE-ELIM: $\frac{\bot}{\phi}$

Other Helpful Rules

MODUS-TOLLENS: $\frac{\phi \to \psi \quad \neg \psi}{\neg \phi}$

PROOF-BY-CONTRADICTION: $\frac{\neg \phi ... | \bot}{\phi}$

LAW-EXCL-MID: $\frac{}{\phi \lor \neg \phi}$

Provable Equivalence

If $\phi \vdash \psi$ and $\psi \vdash \phi$, we say ϕ and ψ are **provably equivalent**

Notation: $\phi \dashv\vdash \psi$

Theorem

If $\vdash \phi$, we say that ϕ is a *thoerem*

- Theorems are conclusions that do not require any premises
- Ex. $\vdash \phi \to \phi$

- To prove this sequent valid, simply assume ϕ and then use ImpIntro.

1.3 Propositional Logic as a Formal Language

Want to use propositional logic as a **specification language**

We need to formalize:

- What this language looks like (syntax)
- What sentences in the language mean (semantics)

Well-formed Formulas

Proof rules apply for "any" formulas ϕ, ψ

- Ex. Show: p $\lor$ $\neg$r, (p $\lor$ $\neg$r) $\rightarrow$ (r $\rightarrow$ p) $\vdash$ r $\rightarrow$ p

  ϕ = p $\lor$ $\neg$r

  ψ = r $\rightarrow$ p

  Proof rules apply to any formula, ϕ and ψ can be as complicated as they want, as long as it still fit the pattern.

Caveat: not just any formula - only well-formed

- Ex. a, a $\rightarrow \land \vdash \land$
  - This formula makes no sense

Alphabet of propositional logic ($\sum$): {$\neg$, $\lor$, $\land$, $\rightarrow$, (,)} $\cup$ {a,b,c,...}

- Infinitely-many atomic propositions a,b,c, ...

Well-Formed Formulas - Inductive Definition

Inductively defining the set W of well-formed formulas for propositional logic.

Well-formed formulas of propositional logic are obtained form using **only** these construction rules:

- Every propositional atom a,b,c is a well-formed formula.
- $\neg$: If ϕ is a well-formed formula, then so is $(\neg\phi)$
- $\land$: If ϕ and ψ are well-formed formulas, then so is $(\phi \land \psi)$
- $\lor$: If ϕ and ψ are well-formed formulas, then so is $(\phi \lor \psi)$
- $\rightarrow$: If ϕ and ψ are well-formed formulas, then so is $(\phi \rightarrow \psi)$

Syntax of Propositional Logic

Alternative: describe well-formed formulas using grammar.

Grammar for propositions (Backus Naur Form):

ϕ ::= p | (ϕ) | $\phi \land \phi$ | $\phi \lor \phi$ | $\phi \rightarrow \phi$

- This is the well-formed construction rules condensed.

Example: Is $(((\neg p) \land q) \rightarrow (p \land (q \lor (\neg r))))$ well-formed?

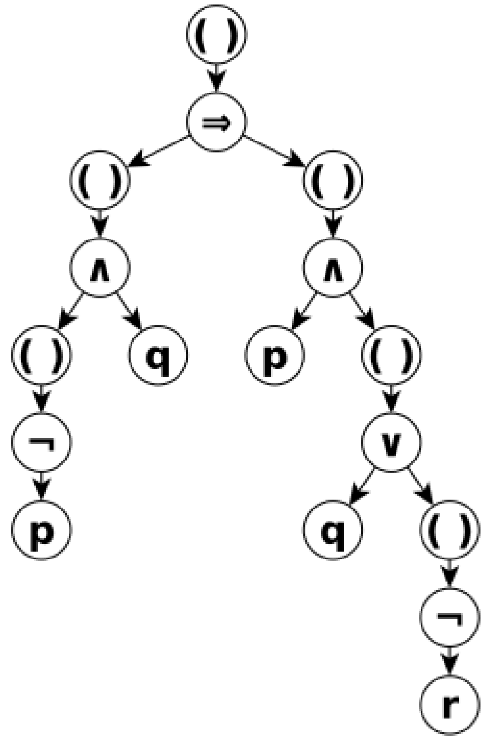


Figure 1: Example problem above converted into a parse tree

- Parse tree also encodes **subformulas** (Any sub-tree corresponds to a subformula)
  - Ex. Right-most subtree with () as root is a subformula
- For a parser, the starting node is ϕ from the Back Naur form equation.

Non-well-formed Formulas

How can we show that $a \lor b\neg$ is not well-formed?

- Based on grammar, $\neg$ must be followed by a proposition
- In general, need to search through potential parse trees
  - Parse does this for us automatically

1.4 Semantics of Propositional Logic

Truth Tables

Truth tables: define meaning of formulas based on whether their subformulas are true/false.

p	q	$\neg p$	$\neg q$	$p \implies \neg q$	$q \lor \neg p$	$(p \implies \neg q) \implies (q \lor \neg p)$
T	T	F	F	F	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	T
F	F	T	T	T	T	T

Figure 2: Example of truth table for $(p \rightarrow \neg q) \rightarrow (q \lor \neg p)$

Strong Induction

Assume: P(0), P(1), . . . , P(K) are true

Use assumption to prove P(K + 1)

Strong Induction Example

Problem: prove that for any propositional logic well-formed formula, the number of left parentheses equals the number of right.

Basic Idea: Use induction over the height of formula's parse tree T

1. Base Case (height = 1) holds: the parse tree T can only be a single node (atomic proposition), so there are no parentheses.
2. Inductive Step (height > 1): assume that the property holds for all parse trees of height n-1. Consider a parse tree T of height n. There are 3 cases:

- If the root node is one of $\rightarrow$, $\wedge$, $\vee$ then the left/right subtrees both have height n-1. By the induction hypothesis, both subtrees have matching numbers of left/right parentheses. Therefore, so does T.
- If the root node is $\neg$, then the subtree has height n-1. By the induction hypothesis, this subtree has matching numbers of left/right parentheses. Therefore, so does T.
- If the root node is (), then the subtree has height n-1. By the induction hypothesis, this subtree has matching numbers of left/right parentheses, so adding 1 left and 1 right maintains this property for T.

Entailment

Entailment: semantic concept.

$\phi_1, \phi_1, \ldots, \phi_n \models \psi$

This is different from the syntactic concept of sequent

$\phi_1, \phi_1, \ldots, \phi_n \vdash \psi$

Entailment says: for all valuations which makes $\phi_1, \phi_1, \ldots, \phi_n$ true, ψ is also true.

Soundness of Propositional Logic

Natural deduction doesn't let us reach incorrect conclusions

- If $\phi_1, \phi_1, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_1, \ldots, \phi_n \models \psi$ holds
- Soundness in formal methods typically has this form: if we can derive ψ using syntactic rules, then ψ is semantically correct.

Completeness of Propositional Logic

Correct facts can always be deduced via natural deduction

If $\phi_1, \phi_1, \ldots, \phi_n \models \psi$ holds, then $\phi_1, \phi_1, \ldots, \phi_n \vdash \psi$ is valid

Completeness in formal methods typically has this form: if ψ is semantically correct, then we can prove that ψ's correctness using syntactic rules.

Syntactic VS Semantic Reasoning

$\phi_1, \phi_1, \ldots, \phi_n \vdash \psi$ is valid **if and only if** $\phi_1, \phi_1, \ldots, \phi_n \models \psi$ holds

For propositional logic, we can do either syntactic or semantic reasoning!

$\vdash$ requires proof search, and $\models$ requires examining a truth table of exponential size.

Interesting consequences:

$\vdash \psi$ **if and only if** $\models \psi$

ψ is a theorem if and only if ψ is a tautology.

A proposition is a theorem if and only if a proposition is a tautology.

1.5 Normal Forms

Semantic Equivalence and Validity

Syntactic equivalence: $\phi \dashv\vdash \psi$

Semantic equivalence:

- Say ϕ, ψ are semantically equivalent if $\phi \models \psi$ and $\psi \models \phi$
- Denoted: $\phi \equiv \psi$
- Means equal truth tables

Entailment VS Implication

$\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds

if and only if

$\models \phi_1 \implies (\phi_2 \implies (\ldots \implies (\phi_n \implies \psi)))$ holds

above equation is equivalent to

$(\phi_1 \land \phi_2 \land \ldots \land \phi_n) \implies \psi$

Satisfiability

A formula ϕ is **satisfiabile** if there is a valuation that makes ϕ evaluate to T (If there is a T row in ϕ's truth table).

- Ex. $a \land \neg a$ is unsatisfiable

Useful theorem: ϕ is satisfiable if and only if $\neg\phi$is not a tautology.

Conjunctive Normal Form (CNF)

Propositions is in CNF if it's a conjunction of disjunctions.

Syntax of CNF formulas:

C::= D | D $\land$ C (conjunction)

D::= L | L $\lor$ D (disjunction)

L::= p | $\neg$p (literal)

Important theorem: Disjunction $L_1 \lor L_2 \lor \ldots \lor L_n$ is a tautology if and only if there are i, j such that L_i is $\neg L_j$

Building CNF from Truth Table:

Steps:

- Build disjunction ψ_{k} for each FALSE (F) row k
- Negate truth atoms, leave false atoms

Example:

p	q	r	ϕ
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	T

- ψ_2 = ¬p ∨ ¬q ∨ r
- ψ_5 = p ∨ ¬q ∨ ¬r
- ψ_6 = p ∨ ¬q ∨ r
- ψ_7 = p ∨ q ∨ ¬r
- CNF result: $\phi_2 \wedge \phi_5 \wedge \phi_6 \wedge \phi_7$

CNF Algorithm

CNF(NNF(IMPL-FREE(ϕ)))

1. IMPL-FREE: Remove implications
   - Ex. Convert $\phi \implies \psi$ to $

2. NNF: Convert to negation-normal form
   - Ex. a ∧ ¬(a ∧ b) (NOT NEGATION NORMAL)
   - a ∧ (¬a ∧ ¬b) (YES NEGATION NORMAL)
3. CNF: Convert to CNF

Algorithm: IMPL-FREE

Overall task: get rid of implications

Convert $\phi \implies \psi$ to $\neg\phi \vee \psi$

Algorithm: NNF

Overall task: move negation "inward" as much as possible

def NNF(ϕ) = {

ϕ match {

case (p | ¬p) → ϕ

case ¬¬ → NNF(ψ)

case $\psi_1 \wedge \psi_2 \rightarrow$ NNF(ψ_1) $\wedge$ NNF(ψ_2)

case $\psi_1 \vee \psi_2 \rightarrow$ NNF(ψ_1) $\vee$ NNF(ψ_2)

case $\neg(\psi_1 \vee \psi_2) \rightarrow$ NNF($\neg\psi_1$) $\vee$ NNF($\neg\psi_2$)

case $\neg(\psi_1 \wedge \psi_2) \rightarrow$ NNF($\neg\psi_1$) $\wedge$ NNF($\neg\psi_2$)

}

}

Algorithm: CNF

Overall task: handle all but negation

def CNF(ϕ) = {

ϕ match {

case (p | ¬p) → ϕ

case $\psi \wedge \psi \rightarrow$ CNF(ψ_1) $\wedge$ CNF(ψ_2)

case $\psi \vee \psi \rightarrow$ DISTR(CNF(ψ_1), CNF(ψ_2))

}

}

Algorithm: DISTR

Overall task: use distributive property (a $\wedge$ b) $\vee$ c $\equiv$ (a $\vee$ c) $\wedge$ (b $\vee$ c)

def DISTR(ϕ_1, ϕ_2) = {

(ϕ_1, ϕ_2) match {

case ($\psi_1 \wedge \psi_2$, _) →

DISTR(ψ_1, ϕ_2) $\wedge$ DISTR(ψ_2, ϕ_1)

case (_, $\psi_1 \wedge \psi_2$) →

DISTR(ϕ_1, ψ_1) $\wedge$ DISTR(ϕ_1, ψ_2)

case _ → $\phi_1 \vee \phi_2$

}

}

Horn Clauses

F = Horn formula

C = Single horn clause

B = Collection of conjunctions

A = Single atom

F ::= C | C $\wedge$ F (Horny Formula)

C ::= B $\implies$ A (Clause)

B ::= A | A $\wedge$ A (Body)

A ::= p | $\bot$ | $\top$

* Atomic propositions cannot be negated

Horn Satisfiability

Overall goal: check for chain of implications $\top \implies \ldots \implies \bot$

```
def HORN(φ) = {
  val φ₂ = mark all ⊤ in φ
  def helper(ψ) = {
    ψ match {
      case (Ȧ₁ ∧ Ȧ₂ ∧ ··· ∧ Ȧₙ ⟹ A) ∧ ψ′ →
          helper((Ȧ₁ ∧ Ȧ₂ ∧ ··· ∧ Ȧₙ ⟹ Ȧ) ∧ ψ′)
      case _ → ψ
    }
  }
  if(⊥ is marked in helper(φ₂)) UNSAT else SAT
}
```

1.6 SAT Solvers

3-SAT

Format: CNF formula = (a ∨ b ∨ c) ∧ (b ∨ d ∨ f) ∧ (b ∨ d ∨ f)

2 Possible Answers: SAT or UNSAT

Rewriting Propositions

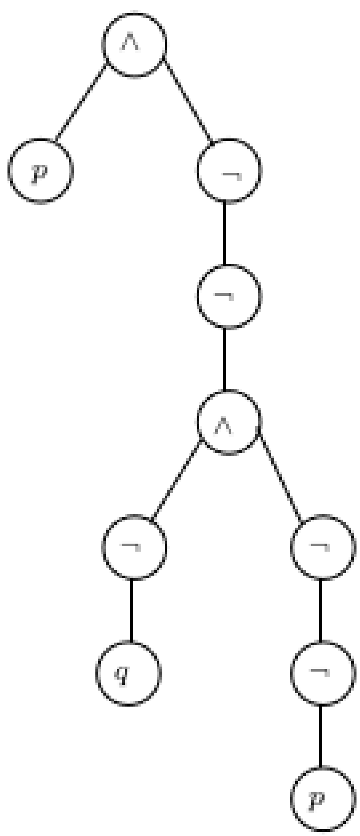
Rewrite everything with just ∧ and ¬

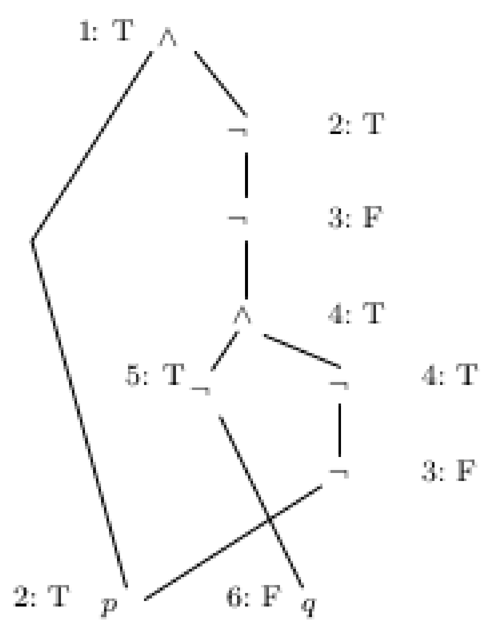
- T(p) = p
- $T(\neg\phi) = \neg T(\phi)$
- $T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2)$
- $T(\phi_1 \implies \phi_2) = \neg(T(\phi_1) \wedge \neg T(\phi_2))$
- $T(\phi_1 \vee \phi_2) = \neg(\neg T(\phi_1) \wedge \neg T(\phi_2))$

Example

Consider $\phi = p \wedge \neg(q \vee \neg p)$. What is $T(\phi)$?

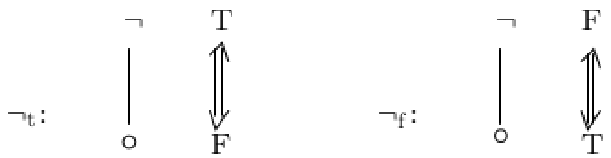
$T(\phi) = p \wedge \neg\neg(\neg q \wedge \neg\neg p)$



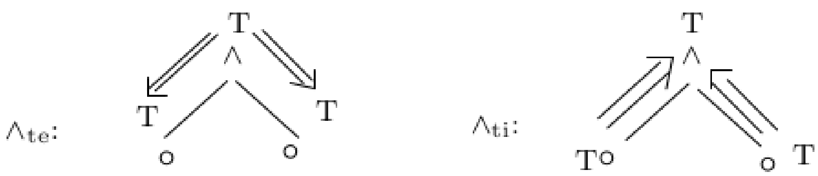


- Combine proposition in tree (leaves)
- Start with True at top and go down

Sat Forcing Algorithms

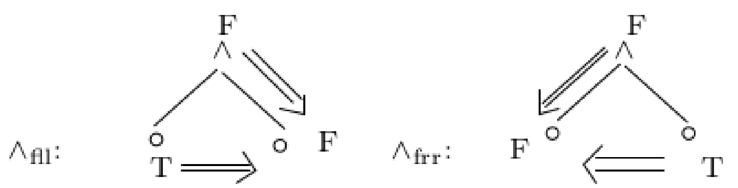


forcing laws for negation



true conjunction forces true conjuncts   true conjunctions force true conjunction

false conjuncts
force false conjunction



false conjunction and true conjunct
force false conjunction

Incompleteness

$\neg(\phi_1 \wedge \phi_2)$

No rule for labeling both children of an F node

This algorithm will terminate with "unknown" on some inputs (incomplete)

Need a way to handle incompleteness

DPLL

Algorithm to determine satisfiability of a set of clauses

General concept:

- Start from a ground CNF formula (a ∨ b ∨ c) ∧ (b ∨ d ∨ f) ∧ (b ∨ d ∨ f)
- Try to build an assignment that verifies the formula
- The assignment is built using a backtracking mechanism
- Worst Case: iterate through entire truth table

Simple Sketch: A tree of possible assignments is used to guide the procedure

- Each node is a set of clauses S_i
- At each node one of the Literal is assigned a truth value
- Truth values are propagated to reduce the number of future assignments

DPLL Algorithm

Clause = $(x \vee x \vee x)$

Algorithm:

- Input = S = C_0 = Set of clauses $\{C_1, \dots, C_k\}$
- Set C_0 as the root of the tree
- Apply (inference) rules to leaves, expanding the tree
- A branch of the tree is no longer expanded if $S_i = \{\}$ or $\{\{\}\} \in S_i$ where $\{\{\}\}$ is the empty clause.
- If $S_i = \{\}$ then S is satisfiable and we can stop the procedure
- If $\{\{\}\} \in S_i$ for all branches then the set is unsatisfiable

Applying Rules

We apply a given set of rules that preserve satisfiability. When we apply a rule, we build at the same time a partial interpretation for S

Rules:

1. Tautology Elimination
2. One-Literal
3. Pure-Literal
4. Splitting

Tautology Elimination

Delete all the ground clauses from S that are tautologies. The remaining set S' is unsatisfiable iff S is unsatisfiable

Example:

> $S = \{(\neg P \vee Q \vee P \vee \neg R) \wedge Q \wedge R\}$
>
> $S' = \{Q \wedge R\}$

One-Literal

If there is a unit ground clause $L \in S$, obtain S' from S by deleting those ground clauses in S containing L. If $S' = \{\}$ then S is satisfiable, otherwise obtain a set S''from S' by deleting $\neg L$ from all clauses. S is unsatisfiable iff S" is unsatisfiable. When we apply this rule we fix $L = \top$ in the partial assignment.

Example:

> $S = \{P \vee Q \vee \neg R, P \vee \neg Q, \neg P, R, U\}$
>
> $S' = \{P \vee Q \vee \neg R, P \vee \neg Q, R, U\}$ $\mathbf{L = \neg P}$
>
> $S'' = \{Q \vee \neg R, \neg Q, R, U\}$

If there's nothing combined with a clause and you're deleting something, it becomes empty set.

> Example: $\mathbf{L = \neg R}$ $S' = \{R, Q \vee P\}$, $S'' = \{\{\}, Q \vee P\}$

Pure-Literal

L $\in$ S is a pure literal iff $\neg$L $\notin$ S

If there is a pure literal L $\in$ S, obtain S from S by deleting clauses where L appears. S' is unsatisfiable iff S' is unsatisfiable. When we apply this rule we fix L = $\top$ in the partial assignment.

Example:

S = {P $\vee$ Q, P $\vee$ $\neg$Q, R $\vee$ Q, R $\vee$ $\neg$Q}

S' = {R $\vee$ Q, R $\vee$ $\neg$Q} **L = P**

Splitting

If you have three clauses in the format of:

- C = set of clauses with L
- D = set of clauses with $\neg$L
- R = set of clauses with neither L or $\neg$L

Then you can split on L to make:

S' = C', D', R where L is TRUE

S''= C', D', R where L is FALSE

Conflict Driven Clause Learning (CDCL)

In DPLL, a previous assignment may not have cause the conflict

Need non-chronological back tracking

Basic idea on conflict:

- Generate conflict clause explaining decisions that caused the conflict
- Jump back to a point before one of the decisions that causes the conflict