

Logic background

Kyle Richardson

Allen Institute for Artificial Intelligence (AI2)

July 2025

<Propositional Logic>

The Basics of Propositional Logic: Syntax

- **Basic syntax** defined as follows:

$$F := P_1 \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \rightarrow F_2) \mid (F_1 \leftrightarrow F_2) \mid \neg F \mid \top \mid \perp.$$

From atomic propositions (*atoms*) P , formulas F are created by repeated application of logical connectives: $\wedge, \vee, \rightarrow, \neg, \leftrightarrow$

The Basics of Propositional Logic: Syntax

- **Basic syntax** defined as follows:

$$F := P_1 \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \rightarrow F_2) \mid (F_1 \leftrightarrow F_2) \mid \neg F \mid \top \mid \perp.$$

From atomic propositions (*atoms*) P , formulas F are created by repeated application of logical connectives: $\wedge, \vee, \rightarrow, \neg, \leftrightarrow$

Example well-formed formulas:

The Basics of Propositional Logic: Syntax

- **Basic syntax** defined as follows:

$$F := P_1 \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \rightarrow F_2) \mid (F_1 \leftrightarrow F_2) \mid \neg F \mid \top \mid \perp.$$

From atomic propositions (*atoms*) P , formulas F are created by repeated application of logical connectives: $\wedge, \vee, \rightarrow, \neg, \leftrightarrow$

Example well-formed formulas:

$$\begin{aligned} & (P_1 \wedge \neg P_2) \rightarrow (P_1 \vee \neg P_3) \\ & ((P_1 \vee \neg P_1) \wedge P_2) \\ & ((P_1 \rightarrow \neg P_1) \wedge (P_2 \leftrightarrow \neg P_2)) \\ & (((P_1 \rightarrow P_2) \rightarrow P_2) \rightarrow \neg P_1) \rightarrow \neg P_3 \end{aligned}$$

The Basics of Propositional Logic: Syntax

- **Basic syntax** defined as follows:

$$F := P_1 \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \rightarrow F_2) \mid (F_1 \leftrightarrow F_2) \mid \neg F \mid \top \mid \perp.$$

From atomic propositions (*atoms*) P , formulas F are created by repeated application of logical connectives: $\wedge, \vee, \rightarrow, \neg, \leftrightarrow$

Not well-formed formulas:

$$\begin{aligned} & (P_1 \wedge \neg P_2) \rightarrow \rightarrow (P_1 \vee \neg P_3) \\ & \wedge \wedge \neg P_1 \end{aligned}$$

The Basics of Propositional Logic: Semantics

Semantics: propositions are *bivalent* (take values **true**, **false**), **truth assignment** $v : \{\text{atoms}\} \rightarrow \{\text{true}, \text{false}\}$.

The Basics of Propositional Logic: Semantics

Semantics: propositions are *bivalent* (take values **true**, **false**), **truth assignment** $v : \{\text{atoms}\} \rightarrow \{\text{true}, \text{false}\}$.

1. $\bar{v}(F) = v(F)$ if F is an atom

The Basics of Propositional Logic: Semantics

Semantics: propositions are *bivalent* (take values **true**, **false**), **truth assignment** $v : \{\text{atoms}\} \rightarrow \{\text{true}, \text{false}\}$.

1. $\bar{v}(F) = v(F)$ if F is an atom
2. $\bar{v}(\neg F) = \begin{cases} \text{true} & \text{if } \bar{v}(F) = \text{false} \\ \text{false} & \text{otherwise} \end{cases}$
3. $\bar{v}(F_1 \wedge F_2) = \begin{cases} \text{true} & \text{if } \bar{v}(F_1) = \text{true} \text{ and } \bar{v}(F_2) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$
4. $\bar{v}(F_1 \vee F_2) = \begin{cases} \text{true} & \text{if } \bar{v}(F_1) = \text{true} \text{ or } \bar{v}(F_2) = \text{true} \text{ (or both)} \\ \text{false} & \text{otherwise} \end{cases}$
5. $\bar{v}(F_1 \rightarrow F_2) = \begin{cases} \text{false} & \text{if } \bar{v}(F_1) = \text{true} \text{ and } \bar{v}(F_2) = \text{false} \\ \text{true} & \text{otherwise} \end{cases}$
6. $\bar{v}(F_1 \leftrightarrow F_2) = \begin{cases} \text{true} & \text{if } \bar{v}(F_1) = \bar{v}(F_2) \\ \text{false} & \text{otherwise} \end{cases}$

The Basics of Propositional Logic: Semantics

Example: Given the truth assignment $v(P_1) = \mathbf{false}$, $v(P_2) = \mathbf{true}$

The Basics of Propositional Logic: Semantics

Example: Given the truth assignment $v(P_1) = \text{false}$, $v(P_2) = \text{true}$

$$\bar{v}(\neg P_1) = \text{true}$$

$$\bar{v}(\neg P_1 \vee P_2) = \text{true}$$

$$\bar{v}(P_2 \wedge P_1) = \text{false}$$

$$\bar{v}(P_1 \rightarrow P_2) = \text{true}.$$

The Basics of Propositional Logic: Semantics

Example: Given the truth assignment $v'(P_1) = \text{false}$, $v'(P_2) = \text{true}$

$$\bar{v}'(\neg P_1) = \text{true}$$

$$\bar{v}'(\neg P_1 \vee P_2) = \text{true}$$

$$\bar{v}'(P_2 \wedge P_1) = \text{false}$$

$$\bar{v}'(P_1 \rightarrow P_2) = \text{true}.$$

Visualizing semantics using **truth tables**

	<i>atoms</i>		<i>complex formulas</i>				
	P_1	P_2	$\neg P_1$	$(P_1 \wedge P_2)$	$(P_1 \vee P_2)$	$(P_1 \rightarrow P_2)$	$(P_1 \leftrightarrow P_2)$
w_1	true	true	false	true	true	true	true
w_2	true	false	false	false	true	false	false
w_3	false	true	false	false	true	true	false
w_4	false	false	true	false	false	true	true

The Basics of Propositional Logic: Logical Equivalences

- **Truth tables** useful for convincing ourselves of certain *logic equivalences* (\equiv), for example::

$$P_1 \rightarrow P_2 \equiv \neg P_1 \vee P_2$$

$$P_1 \leftrightarrow P_2 \equiv (P_1 \rightarrow P_2) \wedge (P_2 \rightarrow P_1)$$

	P_1	P_2	$(\neg P_1 \vee P_2)$	$P_1 \rightarrow P_2$
w_1	true	true	true	true
w_2	true	false	false	false
w_3	false	true	true	true
w_4	false	false	true	true

The Basics of Propositional Logic: Logical Equivalences

- **Truth tables** useful for convincing ourselves of certain *logic equivalences* (\equiv), for example::

$$P_1 \rightarrow P_2 \equiv \neg P_1 \vee P_2$$

$$P_1 \leftrightarrow P_2 \equiv (P_1 \rightarrow P_2) \wedge (P_2 \rightarrow P_1)$$

	P_1	P_2	$(\neg P_1 \vee P_2)$	$P_1 \rightarrow P_2$
w_1	true	true	true	true
w_2	true	false	false	false
w_3	false	true	true	true
w_4	false	false	true	true

The Basics of Propositional Logic: Logical Equivalences

► Standard **algebraic** rules

$$\neg(F_1 \wedge F_2) \equiv (\neg F_1 \vee \neg F_2)$$

$$(F \wedge F) \equiv F$$

$$(F \wedge \top) \equiv \top$$

$$(F_1 \wedge F_2) \equiv (F_2 \wedge F_1)$$

$$(F_1 \wedge (F_1 \wedge F_3)) \equiv ((F_1 \wedge F_1) \wedge F_3)$$

$$(F_1 \wedge (F_1 \vee F_3)) \equiv ((F_1 \wedge F_2) \vee (F_1 \wedge F_3))$$

$$\neg\neg F \equiv F$$

$$\neg(F_1 \vee F_2) \equiv (\neg F_1 \wedge \neg F_2)$$

$$(F \vee F) \equiv F$$

$$(F \vee \perp) \equiv F$$

$$(F_1 \vee F_2) \equiv (F_2 \vee F_1)$$

$$(F_1 \vee (F_1 \vee F_3)) \equiv ((F_1 \vee F_1) \vee F_3)$$

$$(F_1 \vee (F_1 \wedge F_3)) \equiv ((F_1 \vee F_2) \wedge (F_1 \vee F_3))$$

Negation

De Morgan

Idempotency

Absorption

Commutativity

Associativity

Distributivity

Basic Concepts in Propositional Inference

- ▶ **Satisfiability:** The set $\Gamma = \{F_1, \dots, F_n\}$ is said to be **satisfiable** iff there is an assignment to all variables in Γ such that $\bar{v}(F_j) = \mathbf{true}$ for all $F_j \in \Gamma$

Basic Concepts in Propositional Inference

- **Satisfiability:** The set $\Gamma = \{F_1, \dots, F_n\}$ is said to be **satisfiable** iff there is an assignment to all variables in Γ such that $\bar{v}(F_j) = \mathbf{true}$ for all $F_j \in \Gamma$

$$\Gamma = \{P_1 \rightarrow P_2, P_3, P_2 \wedge \neg P_4\}$$

Basic Concepts in Propositional Inference

- **Satisfiability:** The set $\Gamma = \{F_1, \dots, F_n\}$ is said to be **satisfiable** iff there is an assignment to all variables in Γ such that $\bar{v}(F_j) = \mathbf{true}$ for all $F_j \in \Gamma$

$$\Gamma = \{P_1 \rightarrow P_2, P_3, P_2 \wedge \neg P_4\}$$

satisfying assignment:

$$v(P_1) = \mathbf{true}, v(P_2) = \mathbf{true}, v(P_3) = \mathbf{true}, v(P_4) = \mathbf{false}$$

Basic Concepts in Propositional Inference

- **Satisfiability:** The set $\Gamma = \{F_1, \dots, F_n\}$ is said to be **satisfiable** iff there is an assignment to all variables in Γ such that $\bar{v}(F_j) = \mathbf{true}$ for all $F_j \in \Gamma$

$$\Gamma = \{P_1 \rightarrow P_2, P_3, P_2 \wedge \neg P_4 \wedge \neg P_1\}$$

Basic Concepts in Propositional Inference

- **Satisfiability:** The set $\Gamma = \{F_1, \dots, F_n\}$ is said to be **satisfiable** iff there is an assignment to all variables in Γ such that $\bar{v}(F_j) = \mathbf{true}$ for all $F_j \in \Gamma$

$$\Gamma = \{P_1 \rightarrow P_2, P_3, P_2 \wedge \neg P_4 \wedge \underline{\neg P_1}\}$$

Validity: F is **valid** (or a *tautology*) if it is true in all assignments

Basic Concepts in Propositional Inference

- ▶ **Entailment:** Any formula α is a **logical consequence** of (or *logically entailed by*) a set of formulas $\Gamma = \{F_1, F_2, \dots\}$

$$\Gamma \models \alpha$$

iff there is no interpretation in which Γ is **true** and α is **false**.

Basic Concepts in Propositional Inference

- **Entailment:** Any formula α is a **logical consequence** of (or *logically entailed by*) a set of formulas $\Gamma = \{F_1, F_2, \dots\}$

$$\Gamma \models \alpha$$

iff there is no interpretation in which Γ is **true** and α is **false**.

Example entailments:

$$(P_1 \wedge P_2) \models P_1$$

$$(P_1 \rightarrow P_2) \models ((P_1 \wedge P_3) \rightarrow P_2)$$

$$((P_1 \wedge P_2) \rightarrow P_3) \rightarrow P_4 \models ((P_1 \rightarrow P_3) \rightarrow P_4)$$

Relating these different semantic concepts

Theorem : Relating Propositional Entailment, Validity and Satisfiability

For any $\Gamma = F_1, \dots, F_m$ and formula α , the entailment relation $\Gamma \models \alpha$ is equivalent to the following statements:

1. The formula $(F_1 \wedge \dots \wedge F_m) \rightarrow \alpha$ is **valid**, or a **tautology**;
2. The formula $(F_1 \wedge \dots \wedge F_m \wedge \neg \alpha)$ is **unsatisfiable**.

Relating these different semantic concepts

Theorem : Relating Propositional Entailment, Validity and Satisfiability

For any $\Gamma = F_1, \dots, F_m$ and formula α , the entailment relation $\Gamma \models \alpha$ is equivalent to the following statements:

1. The formula $(F_1 \wedge \dots \wedge F_m) \rightarrow \alpha$ is **valid**, or a **tautology**;
2. The formula $(F_1 \wedge \dots \wedge F_m \wedge \neg \alpha)$ is **unsatisfiable**.

Important take-away: All problems are reducible to satisfiability testing.

Using propositional solvers

```
1  from z3 import * ## to install: pip install z3-solver
2
3  solver = Solver()
4  P,Q,R,S = Bools("P Q R S")
5
6  ###  $\Gamma = [((P \wedge Q) \rightarrow R) \rightarrow S]$ , add as assertion to solver
7  Gamma = [ Implies(Implies(And(P,Q),R),S) ]
8  solver.add(Gamma)
9
10 ## check if satisfiable
11 solver.check() ## => "sat",
12 solver.model() ## => [S = True, R = False, ...]
13
14 ## query,  $\alpha = ((P \wedge Q) \rightarrow S)$ 
15 alpha = Implies(Implies(P,Q),S)
16
17 ## add negated query, creating assertion  $\Gamma \wedge \neg \alpha$ 
18 solver.add(Not(alpha))
19 solver.check() # => unsat, i.e.,  $\Gamma \models \alpha$ 
```

Boolean Satisfiability (SAT)

Boolean Satisfiability (SAT)

- ▶ **SAT**: The classic NP-complete problem ([Cook, 1971](#)), widely studied throughout computer science. *Variations*:

Boolean Satisfiability (SAT)

- ▶ **SAT**: The classic NP-complete problem ([Cook, 1971](#)), widely studied throughout computer science. *Variations*:
 1. **Model counting** ($\#SAT$): Counting the number of satisfying assignments for a given formula, $\#P$ -complete ([Valiant, 1979](#)).

Boolean Satisfiability (SAT)

- ▶ **SAT**: The classic NP-complete problem ([Cook, 1971](#)), widely studied throughout computer science. *Variations*:
 1. **Model counting** ($\#SAT$): Counting the number of satisfying assignments for a given formula, $\#P$ -complete ([Valiant, 1979](#)).
 2. **Majority SAT** (MajSAT): Determine whether the *majority* of assignments make a formula true, PP-complete ([Gill III, 1974](#)).
 3. **Max-SAT**: Determine the maximum number of *clauses* that can be satisfied by any assignment.

Boolean Satisfiability (SAT)

- ▶ **SAT**: The classic NP-complete problem ([Cook, 1971](#)), widely studied throughout computer science. *Variations*:
 1. **Model counting** ($\#SAT$): Counting the number of satisfying assignments for a given formula, $\#P$ -complete ([Valiant, 1979](#)).
 2. **Majority SAT** (MajSAT): Determine whether the *majority* of assignments make a formula true, PP-complete ([Gill III, 1974](#)).
 3. **Max-SAT**: Determine the maximum number of *clauses* that can be satisfied by any assignment.

Challenge: Efficient inference, (*for us*) making our logic amenable to gradient-based learning, differentiable.

Towards Algorithms for SAT: Normal Forms

- ▶ A formula is in **Negation Normal Form (NNF)** if negation is allowed over atoms, and \wedge, \vee, \neg are the only Boolean connectives used.

Towards Algorithms for SAT: Normal Forms

- ▶ A formula is in **Negation Normal Form (NNF)** if negation is allowed over atoms, and \wedge, \vee, \neg are the only Boolean connectives used.

Example: $\neg(P_1 \rightarrow P_2)$

Towards Algorithms for SAT: Normal Forms

- ▶ A formula is in **Negation Normal Form (NNF)** if negation is allowed over atoms, and \wedge, \vee, \neg are the only Boolean connectives used.

Example: $\neg(P_1 \rightarrow P_2)$

$$\neg(P_1 \rightarrow P_2) \equiv \neg(\neg P_1 \vee P_2)$$

$$\equiv (\neg\neg P_1 \wedge \neg P_2)$$

$$\equiv (P_1 \wedge \neg P_2)$$

Conversion of \rightarrow to \vee

De-morgan's rule

negation rule

Towards Algorithms for SAT: Normal Forms

Conjunctive Normal Form (CNF) has the following form:

$$\bigwedge_{j=1} \left(\bigvee_{i=1}^{k_j} \textcolor{blue}{p}_i^j \right),$$

or consists of a conjunction of **clauses**, or a *disjunction of **literals*** (an atom or its negation, e.g., $\textcolor{blue}{P}_1$ or $\neg \textcolor{blue}{P}_1$.)

Towards Algorithms for SAT: Normal Forms

Conjunctive Normal Form (CNF) has the following form:

$$\bigwedge_{j=1}^{k_j} \left(\bigvee_{i=1}^{l_j} p_i^j \right),$$

or consists of a conjunction of **clauses**, or a *disjunction of **literals*** (an atom or its negation, e.g., p_1 or $\neg p_1$.)

A formula in **Disjunctive Normal Form (DNF)** has the following form:

$$\bigvee_{j=1}^{k_j} \left(\bigwedge_{i=1}^{l_j} p_i^j \right)$$

or consists of a disjunction of **terms**, or a *conjunction of literals*

Why normal forms matter

Example: $P_1 \rightarrow (P_2 \wedge P_3)$

Why normal forms matter

Example: $P_1 \rightarrow (P_2 \wedge P_3)$

$$\text{CNF: } \underbrace{(\neg P_1 \vee P_2)}_{\text{clause } C_1} \wedge \underbrace{(\neg P_1 \vee P_3)}_{\text{clause } C_2}$$

$$\text{DNF: } \underbrace{\neg P_1}_{\text{term } T_1} \vee \underbrace{(P_1 \wedge P_2)}_{\text{term } T_2}$$

Why normal forms matter

Example: $P_1 \rightarrow (P_2 \wedge P_3)$

$$\text{CNF: } \underbrace{(\neg P_1 \vee P_2)}_{\text{clause } C_1} \wedge \underbrace{(\neg P_1 \vee P_3)}_{\text{clause } C_2}$$

$$\text{DNF: } \underbrace{\neg P_1}_{\text{term } T_1} \vee \underbrace{(P_1 \wedge P_2)}_{\text{term } T_2}$$

Any formula has an equivalent CNF and DNF formula. **Satisfiability** is easy for DNF, but tautology is hard (*vice versa for CNF*).

Why normal forms matter

Example: $P_1 \rightarrow (P_2 \wedge P_3)$

$$\text{CNF: } \underbrace{(\neg P_1 \vee P_2)}_{\text{clause } C_1} \wedge \underbrace{(\neg P_1 \vee P_3)}_{\text{clause } C_2}$$

$$\text{DNF: } \underbrace{\neg P_1}_{\text{term } T_1} \vee \underbrace{(P_1 \wedge P_2)}_{\text{term } T_2}$$

Why normal forms matter

Example: $P_1 \rightarrow (P_2 \wedge P_3)$

$$\text{CNF: } \underbrace{(\neg P_1 \vee P_2)}_{\text{clause } C_1} \wedge \underbrace{(\neg P_1 \vee P_3)}_{\text{clause } C_2}$$

$$\text{DNF: } \underbrace{\neg P_1}_{\text{term } T_1} \vee \underbrace{(P_1 \wedge P_2)}_{\text{term } T_2}$$

A more complicated case for variables P_0^1, \dots, P_0^n and P_1^1, \dots, P_1^n :

$$(P_0^1 \vee P_1^1) \wedge (P_0^2 \vee P_1^2) \wedge \dots \wedge (P_0^n \vee P_1^n)$$

the corresponding DNF:

$$(P_0^1 \wedge P_0^1 \wedge \dots \wedge P_0^{n-1} \wedge P_0^n) \vee$$

$$(P_0^1 \wedge P_0^1 \wedge \dots \wedge P_0^{n-1} \wedge P_1^n) \vee$$

...

$$(P_1^1 \wedge P_1^2 \wedge \dots \wedge P_1^{n-1} \wedge P_0^n) \vee$$

$$(P_1^1 \wedge P_1^2 \wedge \dots \wedge P_1^{n-1} \wedge P_1^n) \vee$$

Converting representations in SymPy

```
1  from sympy import *
2  from sympy.logic.boolalg import to_dnf
3
4  bool_vars = "P01,P02,P03,P04,P11,P12,P13,P14"
5  ### defining variables
6  P01,P02,P03,P04,P11,P12,P13,P14 = symbols(bool_vars)
7
8  ### cnf formula
9  cnf_formula = And(
10      Or(P01,P11),Or(P02,P12),
11      Or(P03,P13),Or(P04,P14)
12  )
13
14  ### conversion to DNF
15  to_dnf(cnf_formula)
```

Towards Algorithms for SAT: Conditioning

- **Set based representations** of CNF formulas, Δ , e.g., will represent the formulas

$$(\neg P_1 \vee P_2) \wedge (\neg P_1 \vee P_3)$$

as $\Delta = \{\{\neg P_1, P_2\}, \{\neg P_1, P_3\}\}$.

Towards Algorithms for SAT: Conditioning

- **Set based representations** of CNF formulas, Δ , e.g., will represent the formulas

$$(\neg P_1 \vee P_2) \wedge (\neg P_1 \vee P_3)$$

as $\Delta = \{\{\neg P_1, P_2\}, \{\neg P_1, P_3\}\}$.

Conditioning to a literal l :

$$\Delta|l = \{C - \{\neg l\} \mid C \in \Delta, l \notin C\}.$$

e.g., $\Delta|P_1 = \{\{P_2\}, \{P_3\}\}$

Towards Algorithms for SAT: Splitting and DPLL

Conditioning to a literal l :

$$\Delta|l = \{C - \{\neg l\} \mid C \in \Delta, l \notin C\}.$$

Theorem : Splitting Rule

Given a CNF formula Δ and a variable l , Δ is satisfiable if and only if $\Delta|l$ is satisfiable or $\Delta|\neg l$ is satisfiable.

Towards Algorithms for SAT: Splitting and DPLL

Theorem : Splitting Rule

Given a CNF formula Δ and a variable l , Δ is satisfiable if and only if $\Delta|l$ is satisfiable or $\Delta|\neg l$ is satisfiable.

input : A CNF formula Δ

output: Satisfiability of Δ

```
1 Function DPLL( $\Delta$ ):  
2   if  $\{\}$   $\in \Delta$  then  
3     return unsat                                // empty clause  
4   else if  $\Delta = \{\}$  then  
5     return sat                                    // No more clauses  
6   select literal  $l$  from  $\Delta$                         // Branching rule ;  
7   return DPLL( $\Delta|l$ )  $\vee$  DPLL( $\Delta|\neg l$ )           // Splitting Rule ;
```

Towards Algorithms for SAT: Splitting and DPLL

input : A CNF formula Δ
output: Satisfiability of Δ

```
1 Function DPLL( $\Delta$ ):  
2   if  $\{\}$   $\in \Delta$  then  
3     return unsat                                // empty clause  
4   else if  $\Delta = \{\}$  then  
5     return sat                                    // No more clauses  
6   select literal  $/$  from  $\Delta$                         // Branching rule ;  
7   return DPLL( $\Delta|/$ )  $\vee$  DPLL( $\Delta|\neg /$ )          // Splitting Rule ;
```

Towards Algorithms for SAT: Splitting and DPLL

input : A CNF formula Δ

output: Satisfiability of Δ

```
1 Function DPLL( $\Delta$ ):  
2   if  $\{\}$   $\in \Delta$  then  
3     return unsat                                // empty clause  
4   else if  $\Delta = \{\}$  then  
5     return sat                                    // No more clauses  
6   select literal  $l$  from  $\Delta$                         // Branching rule ;  
7   return DPLL( $\Delta|l$ )  $\vee$  DPLL( $\Delta|\neg l$ )           // Splitting Rule ;
```

Example: $(P_1 \vee P_2) \wedge (P_1 \vee \neg P_2 \vee \neg P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$

Towards Algorithms for SAT: Splitting and DPLL

```
input  : A CNF formula  $\Delta$ 
output: Satisfiability of  $\Delta$ 

1 Function DPLL( $\Delta$ ):
2   if  $\{\}$   $\in \Delta$  then
3     return unsat                                // empty clause
4   else if  $\Delta = \{\}$  then
5     return sat                                    // No more clauses
6   select literal  $l$  from  $\Delta$                     // Branching rule ;
7   return DPLL( $\Delta|l$ )  $\vee$  DPLL( $\Delta|\neg l$ )      // Splitting Rule ;
```

Example: $(P_1 \vee P_2) \wedge (P_1 \vee \neg P_2 \vee \neg P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$

► $\Delta = \{\{P_1, P_2\}, \{P_1, \neg P_2, \neg P_3\}, \{\neg P_1, P_2, \neg P_3\}\}, \text{DPLL}(\Delta)$

Towards Algorithms for SAT: Splitting and DPLL

```
input  : A CNF formula  $\Delta$ 
output: Satisfiability of  $\Delta$ 

1 Function DPLL( $\Delta$ ):
2   if  $\{\}$   $\in \Delta$  then
3     return unsat                                // empty clause
4   else if  $\Delta = \{\}$  then
5     return sat                                    // No more clauses
6   select literal  $l$  from  $\Delta$                     // Branching rule ;
7   return DPLL( $\Delta|l$ )  $\vee$  DPLL( $\Delta|\neg l$ )        // Splitting Rule ;
```

Example: $(P_1 \vee P_2) \wedge (P_1 \vee \neg P_2 \vee \neg P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$

► $\Delta = \{\{P_1, P_2\}, \{P_1, \neg P_2, \neg P_3\}, \{\neg P_1, P_2, \neg P_3\}\}, \text{DPLL}(\Delta)$
 select P_1 : $\Delta|P_1 = \{\{\neg P_1, P_2, \neg P_3\}\}$ **call** $\text{DPLL}(\Delta|P_1)$

Towards Algorithms for SAT: Splitting and DPLL

```
input  : A CNF formula  $\Delta$ 
output: Satisfiability of  $\Delta$ 

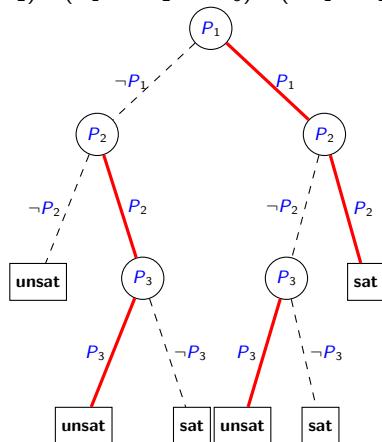
1 Function DPLL( $\Delta$ ):
2   if  $\{\}$   $\in \Delta$  then
3     return unsat                                // empty clause
4   else if  $\Delta = \{\}$  then
5     return sat                                    // No more clauses
6   select literal  $l$  from  $\Delta$                     // Branching rule ;
7   return DPLL( $\Delta|l$ )  $\vee$  DPLL( $\Delta|\neg l$ )        // Splitting Rule ;
```

Example: $(P_1 \vee P_2) \wedge (P_1 \vee \neg P_2 \vee \neg P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$

► $\Delta = \{\{P_1, P_2\}, \{P_1, \neg P_2, \neg P_3\}, \{\neg P_1, P_2, \neg P_3\}\}$, DPLL(Δ)
 select P_1 : $\Delta|P_1 = \{\{\neg P_1, P_2, \neg P_3\}\}$ **call** DPLL($\Delta|P_1$)
 select P_2 : $\Delta|P_2 = \{\}$ **call** DPLL($\Delta|P_2$) (returns **sat**)

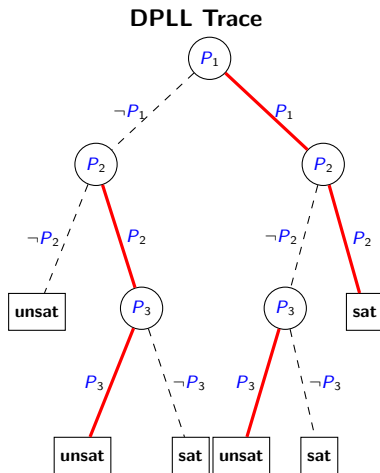
Visualizing DPLL

$$(P_1 \vee P_2) \wedge (P_1 \vee \neg P_2 \vee \neg P_3) \wedge (\neg P_1 \vee P_2 \vee \neg P_3)$$

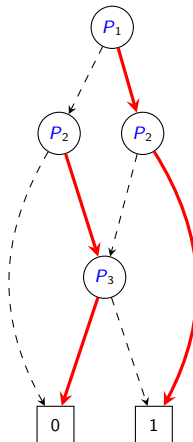


- With some extra work we can store a trace of the search, *exhaustive DPLL* (Huang and Darwiche, 2007; Oztok and Darwiche, 2018).

Visualizing DPLL



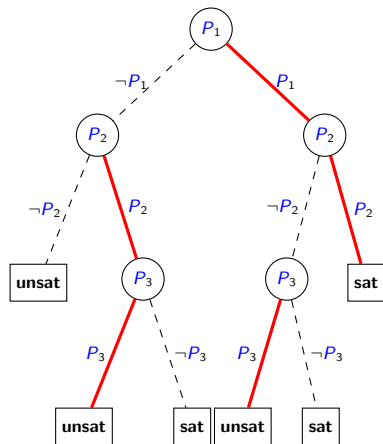
Binary Decision Diagram



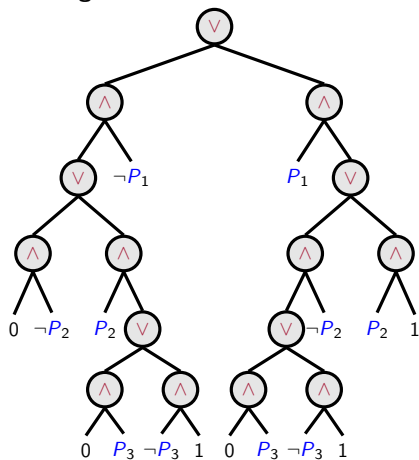
- A little more work, can get (ordered) BDDs (Bryant, 1986).

Visualizing DPLL

DPLL Trace



Negation Normal Form Circuit



► Or equivalent NNF circuit representation ([Darwiche and Marquis, 2002](#)).

Interim Summary

- ▶ SAT is at the foundations of much of automated reasoning, classical algorithm is **DPLL** ([Davis and Putnam, 1960](#); [Davis et al., 1962](#)).

Interim Summary

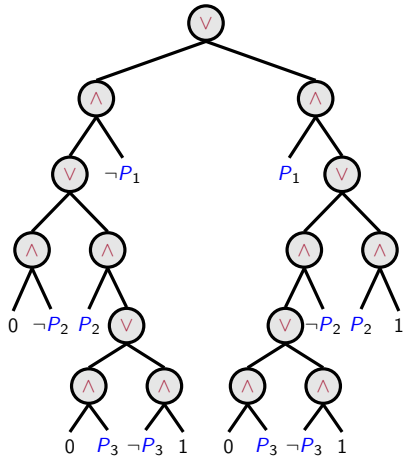
- ▶ SAT is at the foundations of much of automated reasoning, classical algorithm is **DPLL** ([Davis and Putnam, 1960](#); [Davis et al., 1962](#)).
- ▶ SAT algorithms have played another role ([Huang and Darwiche, 2005](#)), *top-down compilers* that transform logic into structures that:

Interim Summary

- ▶ SAT is at the foundations of much of automated reasoning, classical algorithm is **DPLL** ([Davis and Putnam, 1960](#); [Davis et al., 1962](#)).
- ▶ SAT algorithms have played another role ([Huang and Darwiche, 2005](#)), *top-down compilers* that transform logic into structures that:
 - Facilitate tractable inference, i.e., polytime queries or transformations;
 - Are amendable to gradient-based learning, differentiable.

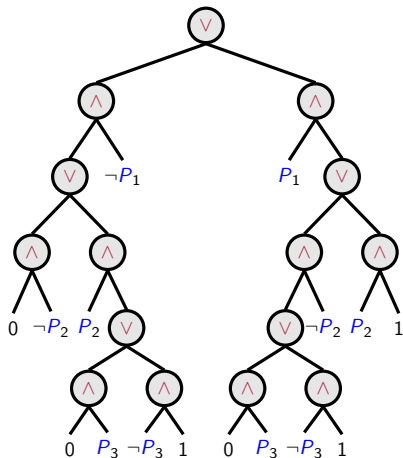
Knowledge Compilation

Negation Normal Form (NNF) Circuits



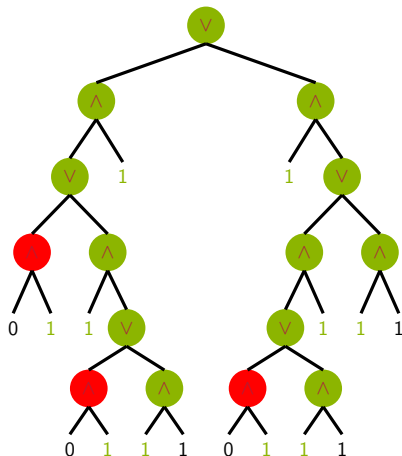
- **NNF**: A rooted DAG where, *leaf nodes* are labels with **true**, **false** (0,1), P or $\neg P$ (from set of *atoms*), internal nodes labeled with \wedge, \vee .

Why?



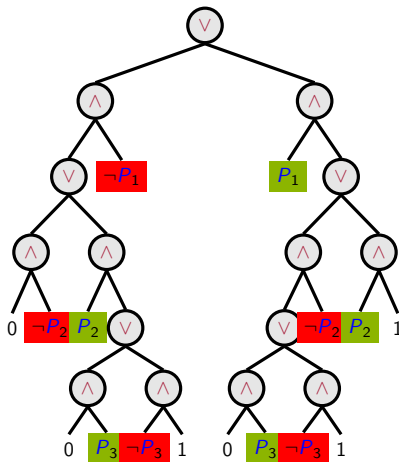
- **Decomposable NNF** (Darwiche, 2001a): A type of NNF that has some interesting properties.

Why?



- ▶ Certain *queries* can be answered in linear time, e.g., **SAT** (*make literal nodes 1 and evaluate*).

Why?



- ▶ Entailment, e.g., $\Delta \models (\neg P_1 \vee \neg P_2 \vee \neg P_3)$ (Negate query, assign literal values accordingly in DAG, evaluate)

Tractable Classes of Boolean Circuits and Their Properties

Odd-parity function, assigns **true** when odd number of variables are true, **false** otherwise. E.g., for variables P_1, P_2, P_3, P_4 in DNF form

Tractable Classes of Boolean Circuits and Their Properties

Odd-parity function, assigns **true** when odd number of variables are true, **false** otherwise. E.g., for variables P_1, P_2, P_3, P_4 in DNF form

$$(P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4) \vee$$

$$(\neg P_1 \wedge P_2 \wedge \neg P_3 \wedge \neg P_4) \vee$$

$$(\neg P_1 \wedge \neg P_2 \wedge P_3 \wedge \neg P_4) \vee$$

$$(\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge P_4) \vee$$

$$(P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4) \vee$$

$$(P_1 \wedge P_2 \wedge \neg P_3 \wedge P_4) \vee$$

$$(P_1 \wedge \neg P_2 \wedge P_3 \wedge P_4) \vee$$

$$(\neg P_1 \wedge P_2 \wedge P_3 \wedge P_4) \vee$$

Tractable Classes of Boolean Circuits and Their Properties

Odd-parity function, assigns **true** when odd number of variables are true, **false** otherwise. E.g., for variables P_1, P_2, P_3, P_4 in DNF form

$$(P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4) \vee$$

$$(\neg P_1 \wedge P_2 \wedge \neg P_3 \wedge \neg P_4) \vee$$

$$(\neg P_1 \wedge \neg P_2 \wedge P_3 \wedge \neg P_4) \vee$$

$$(\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge P_4) \vee$$

$$(P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4) \vee$$

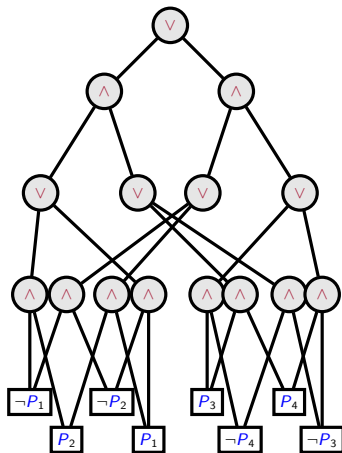
$$(P_1 \wedge P_2 \wedge \neg P_3 \wedge P_4) \vee$$

$$(P_1 \wedge \neg P_2 \wedge P_3 \wedge P_4) \vee$$

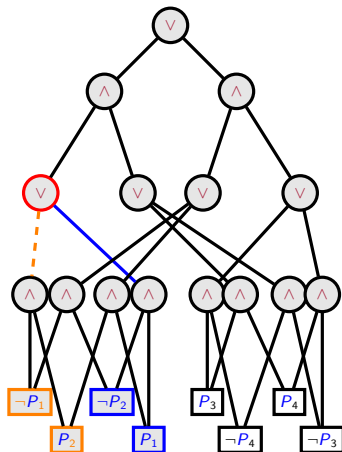
$$(\neg P_1 \wedge P_2 \wedge P_3 \wedge P_4) \vee$$

How many satisfying assignments (or models) does this have?

Tractable Classes of Boolean Circuits and Their Properties

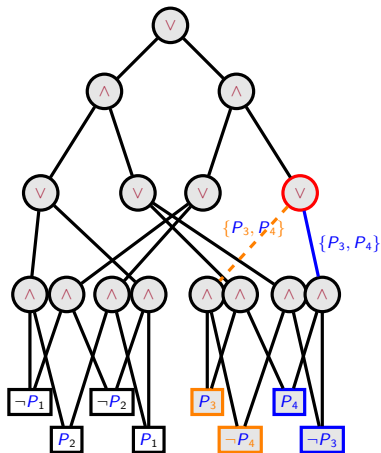


Tractable Classes of Boolean Circuits and Their Properties



Determinism: Disjunctions (Or-gates) have at most one true input, d-DNNF circuits ([Darwiche, 2001b](#)).

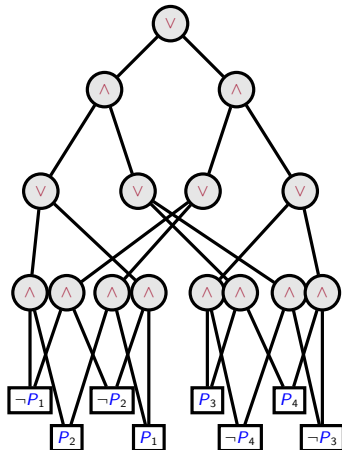
Tractable Classes of Boolean Circuits and Their Properties



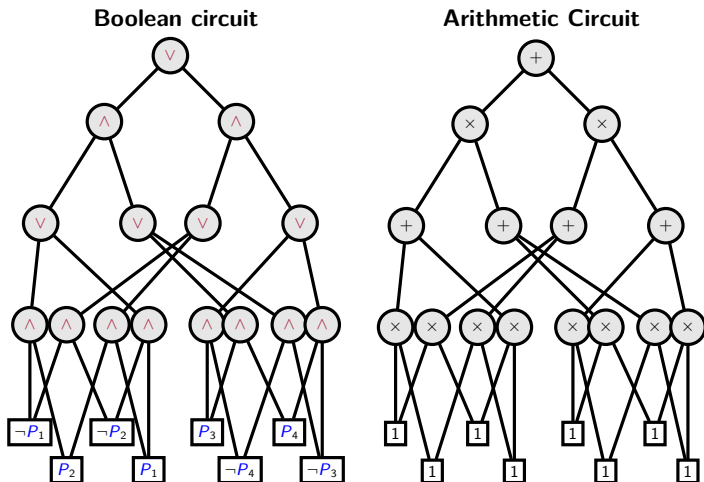
Smoothness: Disjunctions (Or-gates) mention the same variables.

Example: Model Counting

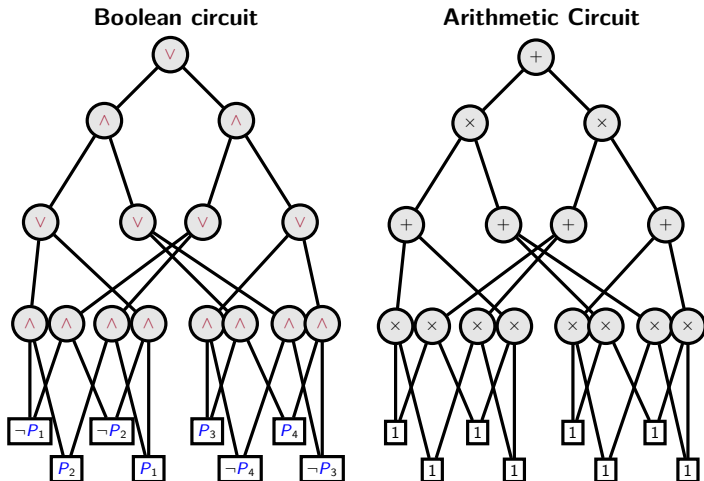
Boolean circuit



Example: Model Counting

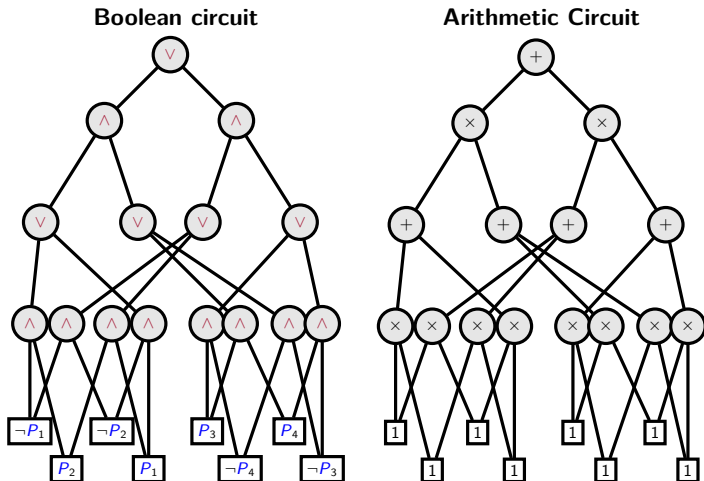


Example: Model Counting



Weighted Model Counting: is useful for tractable probabilistic reasoning (Chavira and Darwiche, 2008), supports differentiable computation.

Example: Model Counting



Weighted Model Counting: is useful for tractable probabilistic reasoning (Chavira and Darwiche, 2008), supports differentiable computation.

Compilation Software

```
1  from pysdd.sdd import SddManager
2
3  sdd = SddManager(var_count=3)
4  p1, p2, p3 = sdd.vars
5  parity_1 = (p1 & -p2 & -p3) | (-p1 & p2 & -p3) \
6             | (-p1 & -p2 & p3)
7  parity_3 = (p1 & p2 & p3)
8  parity = sdd.disjoin(parity_1, parity_3)
9
10 count = parity.wmc(log_mode=False)
11 print(f"model count: {count.propagate()}")
12 ## 4.0
```

- More conditions can be imposed, we will use **Sentential Decision Diagrams (SDD)** later ([Darwiche, 2011](#)), *bottom-up compilation*.

Credits and more reading

Many examples taken/adapted from the work cited throughout, see also [Darwiche \(2022\)](#), many relevant lectures from his group at UCLA:

<https://www.youtube.com/@UCLA.Reasoning>.

- ▶ **Logic background** follows [Davis et al. \(1994\)](#). Other books consulted: [Kroening and Strichman \(2016\)](#); [Enderton \(2001\)](#); [Raedt et al. \(2016\)](#)
- ▶ More on **Knowledge Compilation**: ([Darwiche and Marquis, 2002](#); [Marquis, 2008](#))

Thank you.

References I

- Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691.
- Chavira, M. and Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799.
- Cook, S. (1971). The complexity of theorem proving procedure. In *Proc. 3rd Symp. on Theory of Computing*, pages 151–158.
- Darwiche, A. (2001a). Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4):608–647.
- Darwiche, A. (2001b). On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34.
- Darwiche, A. (2011). Sdd: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*.
- Darwiche, A. (2022). Tractable boolean and arithmetic circuits. *arXiv preprint arXiv:2202.02942*.
- Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264.
- Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.
- Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215.

References II

- Davis, M., Sigal, R., and Weyuker, E. J. (1994). *Computability, complexity, and languages: fundamentals of theoretical computer science*. Elsevier.
- Enderton, H. B. (2001). *A mathematical introduction to logic*. Elsevier.
- Gill III, J. T. (1974). Computational complexity of probabilistic turing machines. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 91–95.
- Huang, J. and Darwiche, A. (2005). Dpll with a trace: From sat to knowledge compilation. In *IJCAI*, volume 5, pages 156–162.
- Huang, J. and Darwiche, A. (2007). The language of search. *Journal of Artificial Intelligence Research*, 29:191–219.
- Kroening, D. and Strichman, O. (2016). *Decision procedures*. Springer.
- Marquis, P. (2008). Knowledge compilation: a sightseeing tour. *Tutorial notes, ECAI*, 8.
- Oztok, U. and Darwiche, A. (2018). An exhaustive dpll algorithm for model counting. *Journal of Artificial Intelligence Research*, 62:1–32.
- Raedt, L. D., Kersting, K., Natarajan, S., and Poole, D. (2016). Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis lectures on artificial intelligence and machine learning*, 10(2):1–189.
- Valiant, L. G. (1979). The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201.