LOGIC

- Logic involves interesting computation problems.
- Logic is "calculus of computer science":
  digital circuit design, programming language semantics, specification and verification, constraint programming, logic programming, databases, artificial intelligence, knowledge representation, machine learning
- In computational complexity theory:
  computational problems from logic are of central importance; they can be used to express computation at various levels.
  This leads to important connections between complexity concepts and actual computational problems.

Boolean logic (propositional logic)

- Syntax (what are well-formed Boolean expressions):
  Boolean variables (atoms) $X = \{x_1, x_2, \ldots\}$
  Boolean connectives $\vee, \wedge, \neg$
- The set of Boolean expressions (formulae) is the smallest set such that all Boolean variables are Boolean expressions and if ϕ_1 and ϕ_2 are Boolean expressions, so are $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$, and $(\phi_1 \vee \phi_2)$
  For example, $(x_1 \vee x_2) \wedge \neg x_3$ is Boolean expression but $(x_1 \vee x_2)\neg x_3$ is not.
- An expression of the form x_i or $\neg x_i$ is called a *literal* where x_i is a Boolean variable.

Semantics (how to interpret Boolean expressions)

- Boolean expressions are propositions that are either true or false.
  They speak about a world where certain atomic proposition (Boolean variables) are either true or false.
  This induces are truth value for any Boolean expression as follows.
- A truth assignment T is mapping from a finite subset $X' \subset X$ to the set of truth values $\{\mathbf{true}, \mathbf{false}\}$.
- Let $X(\phi)$ be the set of Boolean variables appearing in ϕ.
  A truth assignment $T : X' \mapsto \{\mathbf{true}, \mathbf{false}\}$ is *appropriate* to ϕ if $X(\phi) \subseteq X'$.

- Let a truth assignment $T : X' \mapsto \{\mathbf{true}, \mathbf{false}\}$ be appropriate to ϕ, i.e., $X(\phi) \subseteq X'$.
- $T \models \phi$ (T *satisfies* ϕ) is defined inductively as follows:
  If ϕ is a variable, then $T \models \phi$ iff $T(\phi) = \mathbf{true}$.
  If $\phi = \neg\phi_1$, then $T \models \phi$ iff $T \not\models \phi_1$
  If $\phi = \phi_1 \wedge \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ and $T \models \phi_2$
  If $\phi = \phi_1 \vee \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ or $T \models \phi_2$

Example. Let $T(x_1) = \mathbf{true}$, $T(x_2) = \mathbf{false}$.
Then $T \models x_1 \vee x_2$ but $T \not\models (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_2)$

Usual shorthands:

$\phi_1 \Rightarrow \phi_2$: $\neg\phi_1 \vee \phi_2$

$\phi_1 \Leftrightarrow \phi_2$: $(\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$

Definition. Expressions ϕ_1 and ϕ_2 are *equivalent* ($\phi_1 \equiv \phi_2$) iff for all truth assignments T appropriate to both of them, $T \models \phi_1$ iff $T \models \phi_2$.

Example.

$(\phi_1 \vee \phi_2) \equiv (\phi_2 \vee \phi_1)$
$((\phi_1 \wedge \phi_2) \wedge \phi_3) \equiv (\phi_1 \wedge (\phi_2 \wedge \phi_3))$
$\neg\neg\phi \equiv \phi$
$((\phi_1 \wedge \phi_2) \vee \phi_3) \equiv ((\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3))$
$\neg(\phi_1 \wedge \phi_2) \equiv (\neg\phi_1 \vee \neg\phi_2)$
$(\phi_1 \vee \phi_1) \equiv \phi_1$

- Simplified notation:
  $(((x_1 \vee \neg x_3) \vee x_2) \vee x_4 \vee (x_2 \vee x_5))$ is written as
  $x_1 \vee \neg x_3 \vee x_2 \vee x_4 \vee x_2 \vee x_5$
  or
  $x_1 \vee \neg x_3 \vee x_2 \vee x_4 \vee x_5$
- $\bigvee_{i=1}^{n} \varphi_i$ stands for $\varphi_1 \vee \cdots \vee \varphi_n$
  $\bigwedge_{i=1}^{n} \varphi_i$ stands for $\varphi_1 \wedge \cdots \wedge \varphi_n$

Theorem. Every Boolean expression is equivalent to one in conjunctive (disjunctive) normal form.

CNF: $(l_{11} \vee \cdots \vee l_{1n}) \wedge \cdots \wedge (l_{m1} \vee \cdots \vee l_{mn})$
DNF: $(l_{11} \wedge \cdots \wedge l_{1n}) \vee \cdots \vee (l_{m1} \wedge \cdots \wedge l_{mn})$
where each l_{ij} is a literal (Boolean variable or its negation).

CNF/DNF transformation:

1. remove $\Leftrightarrow$ and $\Rightarrow$:
   $\alpha \Leftrightarrow \beta \quad \rightsquigarrow \quad (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha) \qquad (1)$
   $\alpha \Rightarrow \beta \quad \rightsquigarrow \quad \neg\alpha \vee \beta \qquad (2)$
2. Push negations in front of Boolean variables:
   $\neg\neg\alpha \quad \rightsquigarrow \quad \alpha \qquad (3)$
   $\neg(\alpha \vee \beta) \quad \rightsquigarrow \quad \neg\alpha \wedge \neg\beta \qquad (4)$
   $\neg(\alpha \wedge \beta) \quad \rightsquigarrow \quad \neg\alpha \vee \neg\beta \qquad (5)$

3. CNF: move $\wedge$ connectives outside $\vee$ connectives:

$$\alpha \vee (\beta \wedge \gamma) \;\rightsquigarrow\; (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \quad (6)$$

$$(\alpha \wedge \beta) \vee \gamma \;\rightsquigarrow\; (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \quad (7)$$

DNF: move $\vee$ connectives outside $\wedge$ connectives:

$$\alpha \wedge (\beta \vee \gamma) \;\rightsquigarrow\; (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \quad (8)$$

$$(\alpha \vee \beta) \wedge \gamma \;\rightsquigarrow\; (\alpha \wedge \gamma) \vee (\beta \wedge \gamma) \quad (9)$$

Example. Transform $(A \vee B) \Rightarrow (B \Leftrightarrow C)$ to CNF.

$(A \vee B) \Rightarrow (B \Leftrightarrow C)$ (1,2)

$\neg(A \vee B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$ (4)

$(\neg A \wedge \neg B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$ (7)

$(\neg A \vee ((\neg B \vee C) \wedge (\neg C \vee B))) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$ (6)

$((\neg A \vee (\neg B \vee C)) \wedge (\neg A \vee (\neg C \vee B))) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$ (6)

$((\neg A \vee (\neg B \vee C)) \wedge (\neg A \vee (\neg C \vee B))) \wedge ((\neg B \vee (\neg B \vee C)) \wedge (\neg B \vee (\neg C \vee B)))$

$(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee \neg B \vee C) \wedge (\neg B \vee \neg C \vee B)$

- We can assume that normal forms do not have repeated clauses/implicants or repeated literals in clauses/implicants.
- Normal form can be exponentially bigger than the original expression.

Satisfiability and validity

- A Boolean expression ϕ is *satisfiable* iff there is a truth assignment T appropriate to it such that $T \models \phi$.
- ϕ is *valid* (or *tautology*) iff for every truth assignment T appropriate to it, $T \models \phi$. If ϕ is valid, we write $\models \phi$.
- ϕ is unsatisfiable iff $\neg\phi$ is valid.

An important computational problem:

- SAT problem: Given φ in CNF, is φ satisfiable?

  Example: $(x_1 \vee \neg x_2) \wedge \neg x_1$ is satisfiable

  $(x_1 \vee \neg x_2) \wedge \neg x_1 \wedge x_2$ is not.
- SAT can be solved in $n^2 2^n$ time (e.g., truth table method).
- SAT $\in \mathbf{NP}$.

  Nondeterministic Turing machine for $\varphi \in$ SAT:
  for all variables x in φ **do**
  choose nondeterminically: $T(x) := \mathbf{true}$ or $T(x) := \mathbf{false}$;
  if $T \models \varphi$ **then** return "yes" **else** return "no"
- It is open whether SAT $\in \mathbf{P}$.

Horn clauses

- An interesting special case of SAT concerns *Horn clauses*, i.e., clauses (disjunction of literals) with *at most one positive literal*.

  Example:

  $\neg x_1 \vee x_2 \vee \neg x_3$, $\neg x_1 \vee \neg x_3$, x_2 are Horn clauses but

  $\neg x_1 \vee x_2 \vee x_3$ is not.

  A Horn clause with a positive literal is called an *implication* and can be written as: $(x_1 \wedge x_3) \Rightarrow x_2$

- HORNSAT problem:

  Given a conjunction of Horn clauses, is it satisfiable?

HORNSAT $\in \mathbf{P}$:

Algorithm $hornsat(S)$ /* solves $S \in$ HORNSAT in polynomial time */

$T := \emptyset$ /* T is the set of true atoms */

repeat

  if there is an implication $(x_1 \wedge x_2 \wedge \cdots \wedge x_n) \Rightarrow y$ in S

    such that $\{x_1, \ldots, x_n\} \subseteq T$ but $y \notin T$ **then**

    $T := T \cup \{y\}$

until T does not change

if for all purely negative clauses $\neg x_1 \vee \cdots \vee \neg x_n$ in S,

  there is some literal $\neg x_i$ such that $x_i \notin T$ **then**

  return S is satisfiable

else return S is not satisfiable

Boolean functions and expressions

- An n-ary Boolean function is a mapping $\{\mathbf{true}, \mathbf{false}\}^n \mapsto \{\mathbf{true}, \mathbf{false}\}$.

  For example, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ are possible binary Boolean functions and $\neg$ is a unary function.

- A Boolean expression ϕ can be seen as an n-ary Boolean function f_ϕ where $n = |X(\phi)|$.

- A Boolean expression ϕ with variables $x_1, \ldots, x_n$ *expresses* the n-ary function f if for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = \mathbf{true}$ if $T \models \phi$ and $f(\mathbf{t}) = \mathbf{false}$ if $T \not\models \phi$ where $T(x_i) = t_i$, $i = 1, \ldots, n$.

Proposition. Any n-ary Boolean function f can be expressed as a Boolean expression ϕ_f involving variables $x_1, \ldots, x_n$.

- The idea: model the rows of the truth table giving $\mathbf{true}$ as a disjunction of conjunctions.

- Let F be the set of all n-tuples $\mathbf{t} = (t_1, \ldots, t_n)$ with $f(\mathbf{t}) = \mathbf{true}$.

  For each $\mathbf{t}$, let $D_\mathbf{t}$ be a conjunction of literals x_i if $t_i = \mathbf{true}$ and $\neg x_i$ if $t_i = \mathbf{false}$.

- Let $\phi_f = \bigvee_{\mathbf{t} \in F} D_\mathbf{t}$

Note ϕ_f is big: $\mathcal{O}(n2^n)$.

Not all Boolean function can be expressed concisely.

Example.

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

$\phi_f =$

$(\neg x_1 \wedge x_2) \vee$

$(x_1 \wedge \neg x_2)$

Boolean circuits

A more economical way to represent Boolean functions?

Syntax

Graph $C = (V, E)$, $V = \{1, 2, \ldots, n\}$ (gates)
C acyclic (for all $(i, j) \in E, i < j$).

All gates i have a sort $s(i) \in \{\mathbf{true}, \mathbf{false}, \wedge, \vee, \neg\} \cup \{x_1, x_2, \ldots\}$.

if $s(i) \in \{\mathbf{true}, \mathbf{false}\} \cup \{x_1, x_2, \ldots\}$, indegree 0 (inputs)
if $s(i) = \neg$, indegree 1
if $s(i) \in \{\vee, \wedge\}$, indegree 2
Node n output

Semantics: $X(C)$: variables in C
Truth assignment $T : X(C) \mapsto \{\mathbf{true}, \mathbf{false}\}$

Truth value $T(i)$ for each gate i inductively:

- If $s(i) = \mathbf{true}$, $T(i) = \mathbf{true}$ and if $s(i) = \mathbf{false}$, $T(i) = \mathbf{false}$.
- If $s(i) \in X(C)$, then $T(i) = T(s(i))$.
- If $s(i) = \neg$, then $T(i) = \mathbf{true}$ if $T(j) = \mathbf{false}$, otherwise $T(i) = \mathbf{false}$ where (j, i) is the unique edge entering i.
- If $s(i) = \wedge$, then $T(i) = \mathbf{true}$ if $T(j) = T(j') = \mathbf{true}$ else $T(i) = \mathbf{false}$ where (j, i) and (j', i) are the two edges entering i.
- If $s(i) = \vee$, then $T(i) = \mathbf{true}$ if $T(j) = \mathbf{true}$ or $T(j') = \mathbf{true}$ else $T(i) = \mathbf{false}$ where (j, i) and (j', i) are the two edges to i.
- $T(C) = T(n)$ (*truth value of C*).

- For each Boolean circuit C, there is a corresponding Boolean expression ϕ_C.
- For each Boolean expression ϕ, there is a corresponding Boolean circuit C_ϕ such that for any T appropriate for both, $T(C_\phi) = \mathbf{true}$ iff $T \models \phi$.
  Idea: just introduce a new gate for each subexpression.
- Notice that Boolean circuits allow shared subexpressions but Boolean expressions do not.

Important computational problems:

- CIRCUIT SAT:
  Given a circuit C, is there a truth assignment T such that $T(C) = \mathbf{true}$?
- CIRCUIT SAT $\in \mathbf{NP}$
- CIRCUIT VALUE:
  Given a circuit C with no variables, is there a truth assignment T such that $T(C) = \mathbf{true}$?
- CIRCUIT VALUE $\in \mathbf{P}$
  (No truth assignment is needed).

- A Boolean circuit with variables $x_1, \ldots, x_n$ *computes* an n-ary Boolean function f if for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = T(C)$ where $T(x_i) = t_i$, $i = 1, \ldots, n$.
- Any n-ary Boolean function f can be computed by a Boolean circuit involving variables $x_1, \ldots, x_n$.
- Not all Boolean function can be expressed concisely as circuits.

Theorem. For any $n \geq 2$ there is an n-ary Boolean function f such that no Boolean circuit with $\frac{2^n}{2n}$ or fewer gates can compute it.

However, all natural families of Boolean functions seem to need only a linear number of gates to compute!