# Lecture 5: Constraint satisfaction: formalisms and modelling

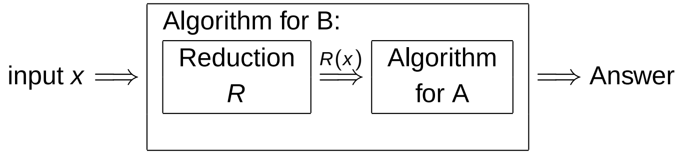
- When solving a search problem the most efficient solution methods are typically based on special purpose algorithms.
- In Lectures 3 and 4 important approaches to developing such algorithms have been discussed.
- However, developing a special purpose algorithm for a given problem requires typically a substantial amount of expertise and considerable resources.
- Another approach is to exploit an efficient algorithm already developed for some problem through reductions.





## Exploiting Reductions

- Given an efficient algorithm for a problem $A$ we can solve a problem $B$ by developing a reduction from $B$ to $A$.




- Constraint satisfaction problems (CSPs) offer attractive target problems to be used in this way:
  - CSPs provide a flexible framework to develop reductions, i.e., encodings of problems as CSPs such that a solution to the original problem can be easily extracted from a solution of the CSP encoding the problem.
  - Constraint programming offers tools to build efficient algorithms for solving CSPs for a wide range of constraints.
  - There are efficient software packages that can be directly used for solving interesting classes of constraints.





## Constraints

- Given variables $Y := y_1, \ldots, y_k$ and domains $D_1, \ldots D_k$, a constraint $C$ on $Y$ is a subset of $D_1 \times \cdots \times D_k$ .
- If $k = 1$, the constraint is called unary and if $k = 2$, binary.

**Example.** Consider variables $y_1, y_2$ both having the domain $D_i = \{0, 1, 2\}$. Then

$$NotEq = \{(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)\}$$

can be taken as a binary constraint on $y_1, y_2$ and then we denote it by $NotEq(y_1, y_2)$ and if it is on $y_2, y_1$, then by $NotEq(y_2, y_1)$.

- In what follows we use a shorthand notation for constraints by giving directly the condition on the variables when it is clear how to interpret the condition on the domain elements.
- Hence, $cond(y_1, \ldots, y_k)$ on variables $y_1, \ldots, y_k$ with domains $D_1, \ldots D_k$ denotes the constraint

$$\{(d_1, \ldots, d_k) \mid d_i \in D_i \text{ for } i = 1, \ldots, k \text{ and } cond(d_1, \ldots, d_k) \text{ holds } \}$$





## Constraints

### Example

Condition $y_1 \neq y_2$ on variables $y_1, y_2$ with domains $D_1, D_2$ denotes the constraint

$$\{(d_1, d_2) \mid d_1 \in D_1, d_2 \in D_2, d_1 \neq d_2\}.$$

So if $y_1, y_2$ both have the domain $\{0, 1, 2\}$, then $y_1 \neq y_2$ denotes the constraint $NotEq(y_1, y_2)$ above.

### Example

Condition $y_1 \leq \frac{y_2}{2} + \frac{1}{4}$ on $y_1, y_2$ both having the domain $\{0, 1, 2\}$ denotes the constraint

$$\{(d_1, d_2) \mid d_1, d_2 \in \{0, 1, 2\}, d_1 \leq \frac{d_2}{2} + \frac{1}{4}\} = \{(0,0),(0,1),(0,2),(1,2)\}.$$

# Constraint Satisfaction Problems (CSPs)

- Given variables $x_1,\dots,x_n$ and domains $D_1,\dots D_n$, a constraint satisfaction problem (CSP):

$$\langle \mathbf{C}; x_1 \in D_1,\dots,x_n \in D_n\rangle$$

where **C** is a set of constraints each on a subsequence of $x_1,\dots,x_n$.

Example

$$\langle\{NotEq(x_1,x_2), NotEq(x_1,x_3), NotEq(x_2,x_3)\},\\ x_1 \in \{0,1,2\}, x_2 \in \{0,1,2\}, x_3 \in \{0,1,2\}\rangle$$

is a CSP. We often use shorthands for the constrains and write

$$\langle\{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\}, x_1 \in \{0,1,2\}, x_2 \in \{0,1,2\}, x_3 \in \{0,1,2\}\rangle$$





# CSPs II

- For a CSP $\langle \mathbf{C}; x_1 \in D_1,\dots,x_n \in D_n\rangle$ a potential solution is given by a value assignment which a mapping $T$ from $\{x_1,\dots,x_n\}$ to $D_1 \cup \dots \cup D_n$ such that for each variable $x_i$, $T(x_i) \in D_i$.
- A value assignment $T$ satisfies a constraint $C$ on variables $x_{i_1},\dots,x_{i_m}$ if $(T(x_{i_1}),\dots,T(x_{i_m})) \in C$.
- **Example.** A value assignment $T = \{x_1 \mapsto 1, x_2 \mapsto 2, \dots, x_n \mapsto n\}$ satisfies the constraint $NotEq$ on $x_1,x_2$ because $(T(x_1),T(x_2)) = (1,2) \in NotEq$ but $T' = \{x_1 \mapsto 1, x_2 \mapsto 1, \dots, x_n \mapsto 1\}$ does not as $(T'(x_1),T'(x_2)) = (1,1) \notin NotEq$.
- A solution to a CSP $\langle \mathbf{C}, x_1 \in D_1,\dots,x_n \in D_n\rangle$ is a value assignment that satisfies every constraint $C \in \mathbf{C}$.

**Example.** Consider a CSP
$\langle\{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\}, x_1 \in \{0,1,2\}, x_2 \in \{0,1,2\}, x_3 \in \{0,1,2\}\rangle$
The assignment $\{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 2\}$ is a solution to the CSP as it satisfies all the constraints but $\{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1\}$ is not as it does not satisfy the constraint $x_2 \neq x_3$ ($NotEq(x_2,x_3)$).





# Example: Graph Coloring Problem

Given a graph $G$, the coloring problem can be encoded as a CSP as follows.

- For each node $v_i$ in the graph introduce a variable $V_i$ with the domain $\{1,\dots,n\}$ where $n$ is the number of available colors.
- For each edge $(v_i,v_j)$ in the graph introduce a constraint $V_i \neq V_j$.
- This is a reduction of the coloring problem to a CSP because the solutions to the CSP correspond exactly to the solutions of the coloring problem:
  a value assignment $\{V_1 \mapsto t_1,\dots,V_n \mapsto t_n\}$ satisfying all the constraints gives a valid coloring of the graph where node $v_i$ is colored with color $t_i$.





# Example: SEND + MORE = MONEY

- Replace each letter by a different digit so that

| | |
|---|---|
| SEND | 9567 |
| + MORE | + 1085 |
| MONEY | 10652 |
| is a correct sum. | The unique solution. |

- Variables: S, E, N, D, M, O, R, Y
- Domains: [1..9] for S, M and [0..9] for E, N, D, O, R, Y
- Constraints:

$$\begin{aligned} & 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ & + 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = {} & 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{aligned}$$

  $x \neq y$ for every pair of variables $x,y$ in {S, E, N, D, M, O, R, Y}.
- It is easy to check that the value assignment $\{S \mapsto 9, E \mapsto 5, N \mapsto 6, D \mapsto 7, M \mapsto 1, O \mapsto 0, R \mapsto 8, Y \mapsto 2\}$ satisfies the constraints, i.e., is a solution to the problem.

## N Queens

Problem: Place $n$ queens on a $n \times n$ chess board so that they do not attack each other.

- Variables: $x_1, \ldots, x_n$ ($x_i$ gives the position of the queen on ith column)
- Domains: [1..n] for each $x_i, i = 1, \ldots, n$
- Constraints: for $i \in [1..n-1]$ and $j \in [i+1..n]$:
  (i) $x_i \neq x_j$ (rows)
  (ii) $x_i - x_j \neq i - j$ (SW-NE diagonals)
  (iii) $x_i - x_j \neq j - i$ (NW-SE diagonals)
- When $n = 10$, the value assignment $\{x_1 \mapsto 3, x_2 \mapsto 10, x_3 \mapsto 7, x_4 \mapsto 4, x_5 \mapsto 1, x_6 \mapsto 5, x_7 \mapsto 2, x_8 \mapsto 9, x_9 \mapsto 6, x_{10} \mapsto 8\}$ gives a solution to the problem.





## Constrained Optimization Problems

- Given: a CSP $P := \langle \mathbf{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ and a function $obj$ which maps solutions of the CSP to real numbers.
- $(P, obj)$ is a constrained optimization problem (COP) where the task is to find a solution $T$ to $P$ for which the value $obj(T)$ is optimal (minimal/maximal).
- **Example.** KNAPSACK: a knapsack of a fixed volume and n objects, each with a volume and a value. Find a collection of these objects with maximal total value that fits in the knapsack.
- Representation as a COP:
  Given: knapsack volume $v$ and $n$ objects with volumes $a_1, \ldots, a_n$ and values $b_1, \ldots, b_n$.
  Variables: $x_1, \ldots, x_n$
  Domains: $\{0, 1\}$
  Constraint: $\sum_{i=1}^{n} a_i \cdot x_i \leq v$,
  Objective function: $\sum_{i=1}^{n} b_i \cdot x_i$.





## Solving CSPs

- Constraints have varying computational properties.
- For some classes of constraints there are efficient special purpose algorithms (domain specific methods/constraint solvers).
  **Examples**
  - Linear equations
  - Linear programming
  - Unification
- For others general methods consisting of
  - constraint propagation algorithms and
  - search methods

  must be used.
- Different encodings of a problem as a CSP utilizing different sets of constraints can have substantial different computational properties.
- However, it is not obvious which encodings lead to the best computational performance.





## Constraints

- In the course we consider more carefully two classes of constraints: linear constraints and Boolean constraints.
- Linear constraints (Lectures 7–9) are an example of a class of constraints which has efficient special purpose algorithms.
- Now we consider Boolean constraints as an example of a class for which we need to use general methods based on propagation and search.
- However, boolean constraints are interesting because
  - highly efficient general purpose methods are available for solving Boolean constraints;
  - they provide a flexible framework for encoding (modelling) where it is possible to use combinations of constraints (with efficient support by solution techniques).

## Boolean Constraints

- A Boolean constraint $C$ on variables $x_1, \ldots, x_n$ with the domain $\{\textbf{true}, \textbf{false}\}$ can be seen as a Boolean function $f_C : \{\textbf{true}, \textbf{false}\}^n \longrightarrow \{\textbf{true}, \textbf{false}\}$ such that a value assignment $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ satisfies the constraint $C$ iff $f_C(t_1, \ldots, t_n) = \textbf{true}$.
- Typically such functions are represented as *propositional formulas*.
- Solution methods for Boolean constraints exploit the structure of the representation of the constraints as formulas.





## Example: Graph coloring

- Consider the problem of finding a 3-coloring for a graph.
- This can be encoded as a set of Boolean constraints as follows:
  - For each vertex $v \in V$, introduce three Boolean variables $v_1, v_2, v_3$ (intuition: $v_i$ is true iff vertex $v$ is colored with color $i$).
  - For each vertex $v \in V$ introduce the constraints

$$v_1 \lor v_2 \lor v_3$$
$$(v_1 \rightarrow \neg v_2) \land (v_1 \rightarrow \neg v_3) \land (v_2 \rightarrow \neg v_3)$$

  - For each edge $(v, u) \in E$ introduce the constraint

$$(v_1 \rightarrow \neg u_1) \land (v_2 \rightarrow \neg u_2) \land (v_3 \rightarrow \neg u_3)$$

- Now 3-colorings of a graph $(V, E)$ and solutions to the Boolean constraints (satisfying truth assignments) correspond: vertex $v$ colored with color $i$ iff $v_i$ assigned true in the solution.





## Propositional formulas

- Syntax (what are well-formed propositional formulas):
  Boolean variables (atoms) $X = \{x_1, x_2, \ldots\}$
  Boolean connectives $\lor, \land, \neg$
- The set of (propositional) formulas is the smallest set such that all Boolean variables are formulas and if $\phi_1$ and $\phi_2$ are formulas, so are $\neg\phi_1$, $(\phi_1 \land \phi_2)$, and $(\phi_1 \lor \phi_2)$.
  For example, $((x_1 \lor x_2) \land \neg x_3)$ is a formula but $((x_1 \lor x_2)\neg x_3)$ is not.
- A formula of the form $x_i$ or $\neg x_i$ is called a literal where $x_i$ is a Boolean variable.
- We employ usual shorthands:
  $\phi_1 \rightarrow \phi_2$: $\neg\phi_1 \lor \phi_2$
  $\phi_1 \leftrightarrow \phi_2$: $(\neg\phi_1 \lor \phi_2) \land (\neg\phi_2 \lor \phi_1)$
  $\phi_1 \oplus \phi_2$: $(\neg\phi_1 \land \phi_2) \lor (\phi_1 \land \neg\phi_2)$





## Semantics

- Atomic proposition (Boolean variables) are either true or false and this induces a truth value for any formula as follows.
- A truth assignment $T$ is mapping from a finite subset $X' \subset X$ to the set of truth values $\{\textbf{true}, \textbf{false}\}$.
- Consider a truth assignment $T : X' \longrightarrow \{\textbf{true}, \textbf{false}\}$ which is appropriate to $\phi$, i.e., $X(\phi) \subseteq X'$ where $X(\phi)$ be the set of Boolean variables appearing in $\phi$.
- $T \models \phi$ ($T$ satisfies $\phi$) is defined inductively as follows:
  If $\phi$ is a variable, then $T \models \phi$ iff $T(\phi) = \textbf{true}$.
  If $\phi = \neg\phi_1$, then $T \models \phi$ iff $T \not\models \phi_1$
  If $\phi = \phi_1 \land \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ and $T \models \phi_2$
  If $\phi = \phi_1 \lor \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ or $T \models \phi_2$

### Example

Let $T(x_1) = \textbf{true}$, $T(x_2) = \textbf{false}$.
Then $T \models x_1 \lor x_2$ but $T \not\models (x_1 \lor \neg x_2) \land (\neg x_1 \land x_2)$

## Representing Boolean Functions

- A propositional formula $\phi$ with variables $x_1, \ldots, x_n$ expresses a $n$-ary Boolean 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 propositional formula $\phi_f$ involving variables $x_1, \ldots, x_n$.

- The idea: model each case of the function $f$ having value **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 $\phi_f = \bigvee_{\mathbf{t} \in F} D_{\mathbf{t}}$

**Example.**

| $x_1$ | $x_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

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





## Logical Equivalence

### Definition

Formulas $\phi_1$ and $\phi_2$ 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 $\varphi_1 \wedge \cdots \wedge \varphi_n$





## Normal Forms

- Many solvers for Boolean constraints require that the constraints are represented in a normal form (typically in conjunctive normal form).

**Proposition.** Every propositional formula is equivalent to one in conjunctive (disjunctive) normal form.

CNF: $(l_{11} \vee \cdots \vee l_{1n_1}) \wedge \cdots \wedge (l_{m1} \vee \cdots \vee l_{mn_m})$
DNF: $(l_{11} \wedge \cdots \wedge l_{1n_1}) \vee \cdots \vee (l_{m1} \wedge \cdots \wedge l_{mn_m})$
where each $l_{ij}$ is a literal (Boolean variable or its negation).

A disjunction $l_1 \vee \cdots \vee l_n$ is called a clause.

A conjunction $l_1 \wedge \cdots \wedge l_n$ is called an implicant.





## Normal Form Transformations

CNF/DNF transformation:

1. remove $\leftrightarrow$ and $\rightarrow$:
   $\alpha \leftrightarrow \beta \rightsquigarrow (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha)$ (1)
   $\alpha \rightarrow \beta \rightsquigarrow \neg\alpha \vee \beta$ (2)
2. Push negations in front of Boolean variables:
   $\neg\neg\alpha \rightsquigarrow \alpha$ (3)
   $\neg(\alpha \vee \beta) \rightsquigarrow \neg\alpha \wedge \neg\beta$ (4)
   $\neg(\alpha \wedge \beta) \rightsquigarrow \neg\alpha \vee \neg\beta$ (5)
3. CNF: move $\wedge$ connectives outside $\vee$ connectives:
   $\alpha \vee (\beta \wedge \gamma) \rightsquigarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$ (6)
   $(\alpha \wedge \beta) \vee \gamma \rightsquigarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$ (7)
   DNF: move $\vee$ connectives outside $\wedge$ connectives:
   $\alpha \wedge (\beta \vee \gamma) \rightsquigarrow (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$ (8)
   $(\alpha \vee \beta) \wedge \gamma \rightsquigarrow (\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$ (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 (for example $(\neg B \vee \neg B \vee C) \equiv (\neg B \vee C)$).
- Normal form can be exponentially bigger than the original formula in the worst case.





## Boolean Circuits

- Normal forms are often quite an unnatural way of encoding problems and it is more convenient to use full propositional logic.
- In many applications the encoding is of considerable size and different parts of the encoding have a substantial amount of common substructure.
- Boolean circuits offer an attractive formalism for representing the required Boolean functions where compactness is enhanced by sharing common substructure.





## Boolean Circuits

- A Boolean circuit $C$ is a tuple $(V, E, s)$ where
- $(V, E)$ is an acyclic graph whose nodes are called gates. The nodes are divided into three categories:
  - output gates (outdegree 0)
  - intermediate gates
  - input gates (indgree 0)
- $s$ assigns a Boolean function $s(g)$ to each intermediate and output gate $g$ of appropriate arity corresponding to the indegree of the gate.
- Typical Boolean functions used in the gates are $and/n$ (n-input AND function), $or/n, not, equiv/2, xor/2, \ldots$

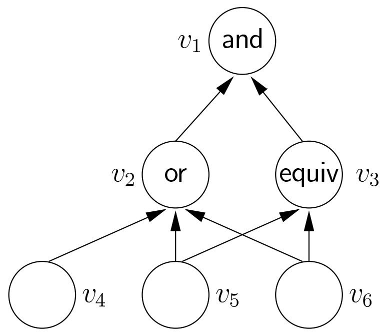
For example

| $x_1$ | $x_2$ | $equiv/2$ | $xor/2$ |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |





## Example. Boolean Circuit




$s(v_1) = and/2$
$s(v_2) = or/3$
$s(v_2) = equiv/2$

$v_1$ is the output gate of the circuit
$v_4, v_5, v_6$ are the input gates

## Boolean Circuits—Semantics

- For a circuit a truth assignment $T : X(C) \longrightarrow \{\textbf{true}, \textbf{false}\}$ gives a truth assignment to each gate in $X(C)$ where $X(C)$ is the set of input gates of $C$.
- This defines a truth value $T(g)$ for each gate $g$ inductively when the gates are ordered topologically in a sequence so that no gate appears in the sequence before its input gates (this is always possible because the circuit is acyclic):
  - If $g \in X(C)$, then the truth assignment $T(g)$ gives the truth value.
  - Otherwise $T(g) = f(T(g_1), \ldots, T(g_n))$ where $(g_1, g), \ldots$ and $(g_n, g)$ are the edges entering $g$ and $f$ is the Boolean function $s(g)$ associated to $g$.

**Example.** For the previous example circuit $C$, $X(C) = \{v_4, v_5, v_6\}$. For a truth assignment $T(v_4) = T(v_5) = T(v_6) = \textbf{false}$, $T(v_3) = \mathit{equiv}(\textbf{false}, \textbf{false}) = \textbf{true}, T(v_2) = \textbf{false}, T(v_1) = \textbf{false}$.

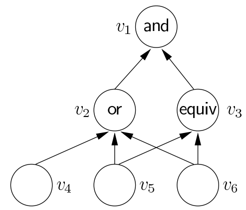




## Circuit Satisfiability Problem

- An interesting computational (search) problem related to circuits is the circuit satisfiability problem.
- A constrained Boolean circuit is a pair $(C, \alpha)$ with a circuit $C$ and constraints $\alpha$ assigning truth values for some gates.
- Given a constrained Boolean circuit $(C, \alpha)$ a truth assignment $T$ satisfies $(C, \alpha)$ if it satisfies the constraints $\alpha$, i.e., for each gate $g$ for which $\alpha$ gives a truth value, $\alpha(g) = T(g)$ holds.
- CIRCUIT SAT problem: Given a constrained Boolean circuit find a truth assignment $T$ that satisfies it.

**Example.** Consider the circuit with constraints $\alpha(v_4) = \textbf{false}$, $\alpha(v_1) = \textbf{true}$.
This circuit has a satisfying truth assignment $T(v_4) = \textbf{false}, T(v_5) = T(v_6) = \textbf{true}$.
If the constraints are $\alpha(v_2) = \textbf{false}$, $\alpha(v_1) = \textbf{true}$, the circuit is unsatisfiable.



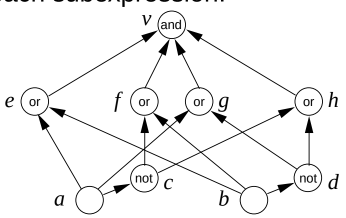





## Boolean Circuits vs. Propositional Formulas

- For each propositional formula $\phi$, there is a corresponding Boolean circuit $C_\phi$ such that for any $T$ appropriate for both, $T(g_\phi) = \textbf{true}$ iff $T \models \phi$ for an output gate $g_\phi$ of $C_\phi$ .
  Idea: just introduce a new gate for each subexpression.

$(a \lor b) \land (\neg a \lor b) \land$
$(a \lor \neg b) \land (\neg a \lor \neg b)$




- For each Boolean circuit $C$, there is a corresponding formula $\phi_C$.
- Notice that Boolean circuits allow shared subexpressions but formulas do not.
  For instance, in the circuit above gates $a, b, c, d$.





## Circuits Compute Boolean Functions

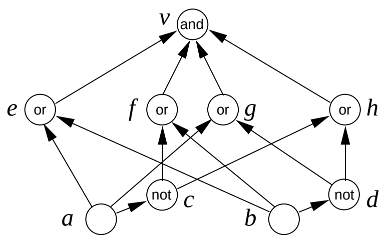
- A Boolean circuit with output gate g and variables $x_1, \ldots, x_n$ computes an $n$-ary Boolean function $f$ if for any $n$-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = T(g)$ where $T(x_i) = t_i$, $i = 1, \ldots, n$.
- Any $n$-ary Boolean function $f$ can be computed by a Boolean circuit involving variables $x_1, \ldots, x_n$.
- Not every Boolean function can be computed using a concise circuit.

### 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.*

## Boolean Circuits as Equation Systems

A Boolean circuit can be written as a system of equations.




$v = \text{and}(e, f, g, h)$
$e = \text{or}(a, b)$
$f = \text{or}(b, c)$
$g = \text{or}(a, d)$
$h = \text{or}(c, d)$
$c = \text{not}(a)$
$d = \text{not}(b)$





## Boolean Modelling

- Propositional formulas/Boolean circuits offer a natural way of modelling many interesting Boolean functions.
- Example. IF-THEN-ELSE $\text{ite}(a, b, c)$ (if $a$ then $b$ else $c$.).
  As a formula:
  $\text{ite}(a, b, c) \equiv (a \wedge b) \vee (\neg a \wedge c)$
  As a circuit:
  $ite = \text{or}(i_1, i_2)$
  $i_1 = \text{and}(a, b)$
  $i_2 = \text{and}(a_1, c)$
  $a_1 = \text{not}(a)$
- Given gates $a, b, c$, $\text{ite}(a, b, c)$ can be thought as a shorthand for a subcircuit given above.
- In the `bczchaff` tool used in the course $\text{ite}(a, b, c)$ is provided as a primitive gate functions.





### Example

Binary adder. Given input bits $a$, $b$ and $c$
compute output bits $o_2 o_1$ which give the sum of $a$, $b$, and $c$ in binary.

As a formula:

$o_1 \equiv ((a \oplus b) \oplus c)$
$o_2 \equiv (a \wedge b) \vee (c \wedge (a \oplus b))$

As a circuit:

$o_1 = \text{xor}(x, c)$
$o_2 = \text{or}(l, r)$
$l = \text{and}(a, b)$
$r = \text{and}(c, x)$
$x = \text{xor}(a, b)$





## Encoding Problems Using Circuits

- Circuits can be used to encode problems in a structured way.
- Example. Given three bits $a, b, c$ find their values such that if at least two of them are ones then either a or b is one else a or c is one.
- We use IF-THEN-ELSE and adder circuits to encode this as a CIRCUIT SAT problem as follows:
  $p = \text{ite}(o_2, x, p_1)$
  $p_1 = \text{or}(a, c)$
  `% full adder; gate` $o_1$ `omitted`
  $o_2 = \text{or}(l, r)$
  $l = \text{and}(a, b)$
  $r = \text{and}(c, x)$
  $x = \text{xor}(a, b)$
- Now each satisfying truth assignment for the circuit with constraint $\alpha(p) = \textbf{true}$ gives a solution to the problem.

## Example. Reachability

Given a graph $G = (\{1,\dots,n\},E)$, constructs a circuit $R(G)$ such that $R(G)$ is satisfiable iff there is a path from 1 to $n$ in $G$.

- The gates of $R(G)$ are of the form
  $g_{ijk}$ with $1 \le i,j \le n$ and $0 \le k \le n$
  $h_{ijk}$ with $1 \le i,j,k \le n$
- $g_{ijk}$ is **true**: there is a path in $G$ from $i$ to $j$ not using any intermediate node bigger than $k$.
- $h_{ijk}$ is **true**: there is a path in $G$ from $i$ to $j$ not using any intermediate node bigger than $k$ but using $k$.





## Example—cont'd

$R(G)$ is the following circuit:

- For $k = 0$, $g_{ijk}$ is an input gate.
- For $k = 1,2,\dots,n$:
  $h_{ijk} = \text{and}(g_{ik(k-1)}, g_{kj(k-1)})$
  $g_{ijk} = \text{or}(g_{ij(k-1)}, h_{ijk})$
- $g_{1nn}$ is the output gate of $R(G)$.
- Constraints $\alpha$:
  For the output gate: $\alpha(g_{1nn}) =$ **true**
  For the input gates: $\alpha(g_{ij0}) =$ **true** if $i = j$ or $(i,j)$ is an edge in $G$ else $\alpha(g_{ij0}) =$ **false**.





## Example—cont'd

- Because of the constraints $\alpha$ on input gates there is at most one possible truth assignment $T$.
- It can be shown by induction on $k = 0,1,\dots,n$ that in this assignment the truth values of the gates correspond to their given intuitive readings.
- From this it follows:
  $R(G)$ is satisfiable iff $T(g_{1nn}) =$ **true** in the truth assignment iff there is a path from 1 to $n$ in $G$ without any intermediate nodes bigger than $n$ iff there is a path from 1 to $n$ in $G$.



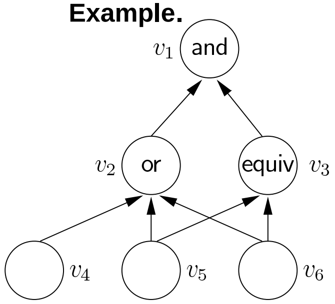


## From Circuits to CNF

- Translating Boolean Circuits to an equivalent CNF formula can lead to exponential blow-up in the size of the formula.
- Often exact equivalence is not necessary but auxiliary variables can be used as long as at least satisfiability is preserved.
- Then a linear size CNF representation can be obtained, e.g., using the co-called Tseitin's translation where given a Boolean circuit $C$ the corresponding CNF formula is obtained as follows
  - a new variable is introduced to each gate of the circuit,
  - the set of clauses in the normal form consists of the gate equation (taken as an equivalence) written in a clausal form for each intermediate and output gate with
  - for each constraint $\alpha(g) = t$, the corresponding literal for $g$ added.
- This transformation preserves satisfiability and even truth assignments in the following sense:
  if $C$ is a Boolean circuit and $\Sigma$ its Tseitin translation, then for every truth assignment $T$ of $C$ there is a satisfying truth assignment $T'$ of $\Sigma$ which agrees with $T$ and vice versa.

# From Circuits to CNF II

**Example.**




Consider the circuit with constraints $\alpha(v_1) = \textbf{true}, \alpha(v_4) = \textbf{false}$.
Gate equations (taken as equivalences) for non-input gates:

$v_1 \leftrightarrow (v_2 \wedge v_3)$
$v_2 \leftrightarrow (v_4 \vee v_5 \vee v_6)$
$v_3 \leftrightarrow (v_5 \leftrightarrow v_6)$

The resulting CNF for the translation:

$$(\neg v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee \neg v_3) \wedge$$
$$(v_2 \vee \neg v_4) \wedge (v_2 \vee \neg v_5) \wedge (v_2 \vee \neg v_6) \wedge (\neg v_2 \vee v_4 \vee v_5 \vee v_6) \wedge$$
$$(v_3 \vee v_5 \vee v_6) \wedge (v_3 \vee \neg v_5 \vee \neg v_6) \wedge (\neg v_3 \vee v_5 \vee \neg v_6) \wedge (\neg v_3 \vee \neg v_5 \vee v_6) \wedge$$
$$v_1 \wedge \neg v_4 \text{ [for constraints]}$$