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 \( A \).

\[
\begin{array}{c|c|c}
\text{input } x & \text{Reduction } R & \text{Algorithm for } A \\
\hline
\end{array}
\]

- 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 \( D_1 = \{0, 1, 2\} \). Then

\[ \text{NotEq} = \{(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)\} \]

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

- 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, \( \text{cond}(y_1, \ldots, y_k) \) on variables \( y_1, \ldots, y_k \) with domains \( D_1, \ldots, D_k \) denotes the constraint

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

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 \( \text{NotEq}(y_1, y_2) \) 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 \( x_1, \ldots, x_n \) and domains \( D_1, \ldots, D_n \), a constraint satisfaction problem (CSP):
  \[
  \langle C; x_1 \in D_1, \ldots, x_n \in D_n \rangle
  \]
  where \( C \) is a set of constraints each on a subsequence of \( x_1, \ldots, x_n \).

Example

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

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

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

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 \( \{1, \ldots, n\} \) 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 \( \{V_1 \mapsto t_1, \ldots, V_n \mapsto t_n\} \) 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
  
  \[
  \begin{align*}
  \text{SEND} & \quad 9567 \\
  + \text{MORE} & \quad + 1085 \\
  \text{MONEY} & \quad 10652
  \end{align*}
  \]
  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:
  \[
  \begin{align*}
  1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\
  + 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\
  = 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y
  \end{align*}
  \]

\( x \neq y \) 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 \( i \)th column)
- Domains: \([1..n]\) for each \( x_i, i = 1, \ldots, n \)
- Constraints: for \( i \in [1..n-1] \) and \( j \in [i+1..n] \):
  1. \( x_i \neq x_j \) (rows)
  2. \( x_i - x_j \neq i - j \) (SW-NE diagonals)
  3. \( x_i - x_j \neq j - i \) (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 \( P := \langle C; x_1 \in D_1, \ldots, x_n \in D_n \rangle \) and a function \( \text{obj} \) which maps solutions of the CSP to real numbers.
- \( (P, \text{obj}) \) is a constrained optimization problem (COP) where the task is to find a solution \( T \) to \( P \) for which the value \( \text{obj}(T) \) 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.
Boolean Constraints

- A Boolean constraint $C$ on variables $x_1, \ldots, x_n$ with the domain $\{\text{true, false}\}$ can be seen as a Boolean function $f_C : \{\text{true, false}\}^n \rightarrow \{\text{true, false}\}$ such that a value assignment $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ satisfies the constraint $C$ if $f_C(t_1, \ldots, t_n) = \text{true}$.

- Typically such functions are represented as propositional formulas.

- Solution methods for Boolean constraints exploit the structure of the representation of the constraints as formulas.

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 $((x_1 \lor x_2) \land \neg x_3)$ 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:
  - $\phi_1 \rightarrow \phi_2 : \neg \phi_1 \lor \phi_2$
  - $\phi_1 \leftrightarrow \phi_2 : (\neg \phi_1 \lor \phi_2) \land (\neg \phi_2 \lor \phi_1)$
  - $\phi_1 \oplus \phi_2 : (\neg \phi_1 \land \phi_2) \lor (\phi_1 \land \neg \phi_2)$

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 $v_1 \lor v_2 \lor v_3$.
  - 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.

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 $\{\text{true, false}\}$.

- Consider a truth assignment $T : X' \rightarrow \{\text{true, false}\}$ 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 $T(\phi) = \text{true}$.
  - 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 $T(x_1) = \text{true}$, $T(x_2) = \text{false}$.

Then $T \models x_1 \lor x_2$ but $T \not\models (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2)$.
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
  \[ t = (t_1, \ldots, t_n), \quad f(t) = \text{true} \text{ if } T \models \phi \quad \text{and} \quad f(t) = \text{false} \text{ if } T \not\models \phi \]

  where \( T(x_i) = t_i, \quad i = 1, \ldots, n. \)

**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 \( \text{true} \) as a disjunction of conjunctions.
- Let \( F \) be the set of all \( n \)-tuples
  \[ t = (t_1, \ldots, t_n) \] with \( f(t) = \text{true}. \)

  For each \( t \), let \( D_t \) be a conjunction of literals \( x_i \) if \( t_i = \text{true} \) and \( \neg x_i \) if \( t_i = \text{false}. \)

- Let \( \phi_f = \bigvee_{t \in F} D_t \)

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:** \((l_1 \lor \cdots \lor l_m) \land \cdots \land (l_{m_1} \lor \cdots \lor l_{m_{m_1}}) \)

  **DNF:** \((l_1 \land \cdots \land l_{n_1}) \lor \cdots \lor (l_{m_1} \land \cdots \land l_{m_{m_1}}) \)

  where each \( l_j \) is a literal (Boolean variable or its negation).

  A disjunction \( l_1 \lor \cdots \lor l_n \) is called a clause.

  A conjunction \( l_1 \land \cdots \land l_n \) is called an implicant.

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, \( T \models \phi_1 \) if \( T \models \phi_2 \).

**Example**

\[
\begin{align*}
(\phi_1 \lor \phi_2) & \equiv (\phi_1 \lor \phi_1) \\
((\phi_1 \land \phi_2) \land \phi_3) & \equiv ((\phi_1 \land \phi_2) \land \phi_3) \\
\neg \neg \phi & \equiv \phi \\
((\phi_1 \land \phi_2) \lor \phi_3) & \equiv ((\phi_1 \lor \phi_3) \land (\phi_2 \lor \phi_3)) \\

(\neg \phi_1 \lor \neg \phi_2) & \equiv (\neg \phi_1 \lor \neg \phi_2) \\
(\phi_1 \lor \phi_1) & \equiv \phi_1
\end{align*}
\]

**Simplified notation:**

\[
\begin{align*}
\bigl((x_1 \lor \neg x_2) \lor x_2) \lor x_3 \lor (x_2 \lor x_3)\bigr) \text{ is written as } x_1 \lor \neg x_2 \lor x_2 \lor x_2 \lor x_3 \text{ or } x_1 \lor \neg x_3 \lor x_2 \lor x_2 \lor x_3.
\end{align*}
\]

- Let \( \bigvee_{i=1}^{n} \phi_i \) stands for \( \phi_1 \lor \cdots \lor \phi_n \)

**Normal Form Transformations**

**CNF/DNF transformation:**

1. \( \alpha \leftrightarrow \beta \sim \neg \alpha \lor \beta \land \neg \beta \lor \alpha \) \quad (1)

2. \( \alpha \lor \beta \sim \neg \alpha \lor \beta \) \quad (2)

3. \( \neg \neg \alpha \sim \alpha \) \quad (3)

4. \( \neg (\alpha \lor \beta) \sim \neg \alpha \land \neg \beta \) \quad (4)

5. \( \neg (\alpha \land \beta) \sim \neg \alpha \lor \neg \beta \) \quad (5)

6. \( \alpha \lor (\beta \land \gamma) \sim (\alpha \lor \beta) \land (\alpha \lor \gamma) \) \quad (6)

7. \( (\alpha \land \beta) \lor \gamma \sim (\alpha \land \gamma) \lor (\beta \land \gamma) \) \quad (7)

8. \( (\alpha \lor \beta) \land \gamma \sim (\alpha \land \gamma) \lor (\beta \land \gamma) \) \quad (8)

9. \( (\alpha \lor \beta) \land \gamma \sim (\alpha \lor \gamma) \lor (\beta \lor \gamma) \) \quad (9)
Example

Transform \((A \lor B) \rightarrow (B \iff C)\) to CNF.

\[
\begin{align*}
(A \lor B) & \rightarrow (B \iff C) & (1,2) \\
\neg(A \lor B) & \lor ((B \lor C) \land (\neg C \lor B)) & (4) \\
(A \land \neg B) & \lor ((\neg B \lor C) \land (\neg C \lor B)) & (7) \\
(A \land (\neg B \lor C)) & \land (\neg A \lor (B \land C)) \land (\neg B \lor ((\neg B \lor C) \land (\neg C \lor B))) & (6) \\
((\neg A \land B) \lor (\neg A \land (B \land C)) \land (\neg B \lor ((\neg B \lor C) \land (\neg C \lor B))) & (6) \\
(A \land \neg B \lor C) & \land (A \land \neg C \lor B) \land (\neg B \lor B \lor C) \land (\neg B \lor \neg C \lor B) & \\
\end{align*}
\]

- We can assume that normal forms do not have repeated clauses/implicants or repeated literals in clauses/implicants (for example \((\neg B \lor B \lor C) \equiv (\neg B \lor C)\)).
- Normal form can be exponentially bigger than the original formula in the worst case.

### 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 \(\text{and}/n, \text{or}/n, \text{not}, \text{equiv}/2, \text{xor}/2, \ldots\)

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>equiv/2</th>
<th>xor/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Example. Boolean Circuit

\[
\begin{align*}
\text{s}(v_1) &= \text{and}/2 \\
\text{s}(v_2) &= \text{or}/3 \\
\text{s}(v_2) &= \text{equiv}/2 \\
\end{align*}
\]

\(v_1\) 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 \( T : X(C) → \{true, false\}\) 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 \( g ∈ X(C) \), 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 \( T(v_4) = T(v_5) = T(v_6) = false \).
\( T(v_3) = equiv(false, false) = true, T(v_2) = false, T(v_1) = false \).

Circuit Satisfiability Problem

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

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

Boolean Circuits vs. Propositional Formulas

- For each propositional formula \( φ \), there is a corresponding Boolean circuit \( C_φ \) such that for any \( T \) appropriate for both, \( T(g_0) = true \) iff \( T ⪯ φ \) for an output gate \( g_0 \) of \( C_φ \).
  Idea: just introduce a new gate for each subexpression.

\[(a ∨ b) ∧ (¬a ∨ b) ∧ (a ∨ ¬b) ∧ (¬a ∨ ¬b)\]

- For each Boolean circuit \( C \), there is a corresponding formula \( φ_C \).
- 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 \( t = (t_1, \ldots, t_n) \), \( f(t) = T(g) \) 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 \( n ≥ 2 \) there is an \( n \)-ary Boolean function \( f \) such that no Boolean circuit with \( \binom{2^n}{2} \) 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:
    \[ \text{ite}(a, b, c) \equiv (a \land b) \lor (\neg a \land c) \]
  - As a circuit:
    \[ \text{ite} = \text{or}(i_1, i_2) \]
    \[ i_1 = \text{and}(a, b) \]
    \[ i_2 = \text{and}(a_1, c) \]
    \[ a_1 = \text{not}(a) \]
  - 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.

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
    \[ o_2 = \text{or}(i, r) \]
    \[ i = \text{and}(a, b) \]
    \[ r = \text{and}(c, x) \]
    \[ x = \text{xor}(a, b) \]
  - Now each satisfying truth assignment for the circuit with constraint \( \alpha(p) = \text{true} \) gives a solution to the problem.
Example. Reachability
Given a graph \( G = (\{1, \ldots, n\}, E) \), 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 
  \( \text{g}_{ijk} \) with \( 1 \leq i, j \leq n \) and \( 0 \leq k \leq n \)
  \( \text{h}_{ijk} \) with \( 1 \leq i, j, k \leq n \)
- \( \text{g}_{ijk} \) is true: there is a path in \( G \) from \( i \) to \( j \) not using any intermediate node bigger than \( k \).
- \( \text{h}_{ijk} \) 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 \( k = 0 \), \( \text{g}_{ij0} \) is an input gate.
- For \( k = 1, 2, \ldots, n \):
  \( \text{h}_{ijk} = \text{and}(\text{g}_{ik(k-1)}, \text{g}_{kj(k-1)}) \)
  \( \text{g}_{ijk} = \text{or}(\text{g}_{ik(k-1)}, \text{h}_{ijk}) \)
- \( \text{g}_{1nn} \) is the output gate of \( R(G) \).
- Constraints \( \alpha \):
  - For the output gate: \( \alpha(\text{g}_{1nn}) = \text{true} \)
  - For the input gates: \( \alpha(\text{g}_{ij0}) = \text{true} \) if \( i = j \) or \( (i, j) \) is an edge in \( G \)
  - else \( \alpha(\text{g}_{ij0}) = \text{false} \).

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
\( \alpha(v_1) = \text{true}, \alpha(v_4) = \text{false}. \)

Gate equations (taken as equivalences) for non-input gates:

\[ v_1 \leftrightarrow (v_2 \land v_3) \]
\[ v_2 \leftrightarrow (v_4 \lor v_5 \lor v_6) \]
\[ v_3 \leftrightarrow (v_5 \leftrightarrow v_6) \]

The resulting CNF for the translation:

\[ \neg v_1 \lor v_2 \land \neg v_1 \lor v_3 \land v_1 \lor \neg v_2 \lor \neg v_3 \land (v_2 \lor \neg v_4) \land (v_2 \lor \neg v_5) \land (v_2 \lor \neg v_6) \land (\neg v_2 \lor v_4 \lor v_5 \lor v_6) \land (v_3 \lor v_5 \lor v_6) \land (v_3 \lor \neg v_5 \lor \neg v_6) \land (\neg v_3 \lor v_5 \lor \neg v_6) \land (\neg v_3 \lor \neg v_5 \lor \neg v_6) \land v_1 \lor \neg v_4 \] [for constraints]