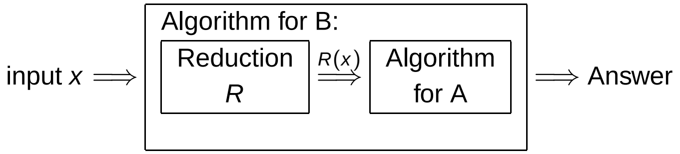
Lecture 5: Constraint satisfaction: formalisms and modelling

- 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 $Y := y_1,\dots,y_k$ and domains $D_1,\dots D_k$, 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, $cond(y_1,\dots,y_k)$ on variables $y_1,\dots,y_k$ with domains $D_1,\dots D_k$ denotes the constraint

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

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

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

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 constraint C on variables $x_{i_1}, \dots, x_{i_m}$, an n-tuple $(d_1, \dots, d_n) \in D_1 \times \cdots \times D_n$ satisfies C if $(d_{i_1}, \dots, d_{i_m}) \in C$
- **Example.** An n-tuple $(1,2,\dots,n)$ satisfies the constraint $NotEq$ on x_1, x_2 because $(1,2) \in NotEq$ but the n-tuple $(1,1,\dots,n)$ does not as $(1,1) \notin NotEq$.
- A solution to a CSP $\langle \mathbf{C}, x_1 \in D_1, \dots, x_n \in D_n \rangle$ is an n-tuple $(d_1, \dots, d_n) \in D_1 \times \cdots \times D_n$ that satisfies each 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 3-tuple $(0,1,2)$ is a solution to the CSP as it satisfies all the constraints but $(0,1,1)$ is not because 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 tuple $(t_1, \dots, t_n)$ 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 tuple $(9,5,6,7,1,0,8,2)$ 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 n-tuple $(3, 10, 7, 4, 1, 5, 2, 9, 6, 8)$ 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 $obj : Sol \mapsto \mathbb{R}$
- (P, obj) is a constrained optimization problem (COP) where the task is to find a solution d to P for which the value $obj(d)$ is optimal.
- **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 tuple $(t_1,\ldots,t_n)$ 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

$$v(1) \lor v(2) \lor v(3)$$
$$(v(1) \to \neg v(2)) \land (v(1) \to \neg v(3)) \land (v(2) \to \neg v(3))$$

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

$$(v(1) \to \neg u(1)) \land (v(2) \to \neg u(2)) \land (v(3) \to \neg u(3))$$

- 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 ϕ_1 and ϕ_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:
  $\phi_1 \to \phi_2$: $\neg\phi_1 \lor \phi_2$
  $\phi_1 \leftrightarrow \phi_2$: $(\neg\phi_1 \lor \phi_2) \land (\neg\phi_2 \lor \phi_1)$
  $\phi_1 \oplus \phi_2$: $(\neg\phi_1 \land \phi_2) \lor (\phi_1 \land \neg\phi_2)$

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 ϕ, i.e., $X(\phi) \subseteq X'$ where $X(\phi)$ be the set of Boolean variables appearing in ϕ.
- $T \models \phi$ (T satisfies ϕ) is defined inductively as follows:
  If ϕ 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 ϕ with variables $x_1,\ldots,x_n$ expresses a n-ary Boolean function f if for any n-tuple of truth values $\mathbf{t}=(t_1,\ldots,t_n)$, $f(\mathbf{t})=\textbf{true}$ if $T\models\phi$ and $f(\mathbf{t})=\textbf{false}$ 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 ϕ_f involving variables $x_1,\ldots,x_n$.

- The idea: model each case of the function 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 $f(\mathbf{t})=\textbf{true}$. For each $\mathbf{t}$, let $D_{\mathbf{t}}$ be a conjunction of literals x_i if $t_i=\textbf{true}$ and $\neg x_i$ if $t_i=\textbf{false}$.
- 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

$\phi_f=$
$(\neg x_1\land x_2)\lor$
$(x_1\land\neg x_2)$

Logical Equivalence

Definition

Formulas 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

$$(\phi_1\lor\phi_2)\equiv(\phi_2\lor\phi_1)$$
$$((\phi_1\land\phi_2)\land\phi_3)\equiv(\phi_1\land(\phi_2\land\phi_3))$$
$$\neg\neg\phi\equiv\phi$$
$$((\phi_1\land\phi_2)\lor\phi_3)\equiv((\phi_1\lor\phi_3)\land(\phi_2\lor\phi_3))$$
$$\neg(\phi_1\land\phi_2)\equiv(\neg\phi_1\lor\neg\phi_2)$$
$$(\phi_1\lor\phi_1)\equiv\phi_1$$

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

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: $(l_{11}\lor\cdots\lor l_{1n_1})\land\cdots\land(l_{m1}\lor\cdots\lor l_{mn_m})$
DNF: $(l_{11}\land\cdots\land l_{1n_1})\lor\cdots\lor(l_{m1}\land\cdots\land l_{mn_m})$
where each l_{ij} 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.

Normal Form Transformations

CNF/DNF transformation:

1. remove $\leftrightarrow$ and $\rightarrow$:
$$\begin{array}{lll}\alpha\leftrightarrow\beta & \rightsquigarrow & (\neg\alpha\lor\beta)\land(\neg\beta\lor\alpha) \quad (1)\\ \alpha\rightarrow\beta & \rightsquigarrow & \neg\alpha\lor\beta \quad (2)\end{array}$$
2. Push negations in front of Boolean variables:
$$\begin{array}{llll}\neg\neg\alpha & \rightsquigarrow & \alpha & (3)\\ \neg(\alpha\lor\beta) & \rightsquigarrow & \neg\alpha\land\neg\beta & (4)\\ \neg(\alpha\land\beta) & \rightsquigarrow & \neg\alpha\lor\neg\beta & (5)\end{array}$$
3. CNF: move $\land$ connectives outside $\lor$ connectives:
$$\begin{array}{llll}\alpha\lor(\beta\land\gamma) & \rightsquigarrow & (\alpha\lor\beta)\land(\alpha\lor\gamma) & (6)\\ (\alpha\land\beta)\lor\gamma & \rightsquigarrow & (\alpha\lor\gamma)\land(\beta\lor\gamma) & (7)\end{array}$$
DNF: move $\lor$ connectives outside $\land$ connectives:
$$\begin{array}{llll}\alpha\land(\beta\lor\gamma) & \rightsquigarrow & (\alpha\land\beta)\lor(\alpha\land\gamma) & (8)\\ (\alpha\lor\beta)\land\gamma & \rightsquigarrow & (\alpha\land\gamma)\lor(\beta\land\gamma) & (9)\end{array}$$

Example

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

$(A \lor B) \rightarrow (B \leftrightarrow C)$ (1,2)
$\neg(A \lor B) \lor ((\neg B \lor C) \land (\neg C \lor B))$ (4)
$(\neg A \land \neg B) \lor ((\neg B \lor C) \land (\neg C \lor B))$ (7)
$(\neg A \lor ((\neg B \lor C) \land (\neg C \lor B))) \land (\neg B \lor ((\neg B \lor C) \land (\neg C \lor B)))$ (6)
$((\neg A \lor (\neg B \lor C)) \land (\neg A \lor (\neg C \lor B))) \land (\neg B \lor ((\neg B \lor C) \land (\neg C \lor B)))$ (6)
$((\neg A \lor (\neg B \lor C)) \land (\neg A \lor (\neg C \lor B))) \land ((\neg B \lor (\neg B \lor C)) \land (\neg B \lor (\neg C \lor B)$
$(\neg A \lor \neg B \lor C) \land (\neg A \lor \neg C \lor B) \land (\neg B \lor \neg B \lor C) \land (\neg B \lor \neg C \lor B)$

- We can assume that normal forms do not have repeated clauses/implicants or repeated literals in clauses/implicants (for example $(\neg B \lor \neg B \lor C) \equiv (\neg B \lor C)$).
- 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.

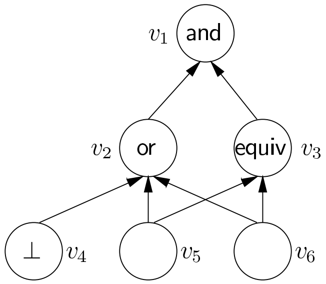
Boolean Circuits

- A Boolean circuit C is a 4-tuple (V, E, s, α) 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.
- α assigns truth values to some gates.
- 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	$equiv/2$
0	0	1
0	1	0
1	0	0
1	1	1

Example. Boolean Circuit



$s(v_1) = and/2$
$s(v_2) = or/3$
$s(v_2) = equiv/2$
$\alpha(v_4) =$ **false**

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

Boolean Circuits—Semantics

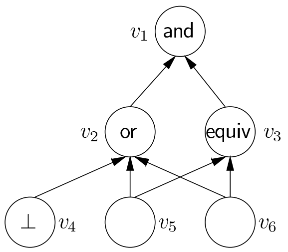
- For a circuit a truth assignment $T : X(C) \longrightarrow \{\textbf{true}, \textbf{false}\}$ gives a truth assignment to each gate in $X(C)$ where $X(C)$ is the set of input gates of C.
- This defines a truth value $T(g)$ for each gate g inductively when the gates are ordered topologically in a sequence 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.
- Given a Boolean circuit (V, E, s, α) we say a truth assignment T satisfies the circuit if it satisfies the constraints α, i.e., for each gate g for which α gives a truth value, $\alpha(g) = T(g)$ holds.
- CIRCUIT SAT problem: Given a Boolean circuit find a truth assignment T that satisfies the circuit.

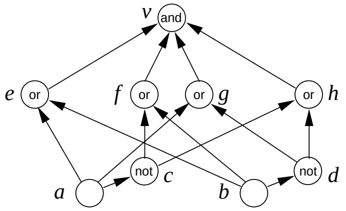
Example. Consider the circuit with constraints $\alpha(v_4) = \textbf{false}$, $\alpha(v_1) = \textbf{true}$.
This circuit has a satisfying truth assignment $T(v_4) = \textbf{false}, T(v_5) = T(v_6) = \textbf{true}$.
If the constraints are $\alpha(v_2) = \textbf{false}$, $\alpha(v_1) = \textbf{true}$,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



Boolean Circuits vs. Propositional Formulas

- For each propositional formulae ϕ, there is a corresponding Boolean circuit C_ϕ such that for any T appropriate for both, $T(g_\phi) = \textbf{true}$ iff $T \models \phi$ for an output gate g_ϕ of C_ϕ .
  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 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

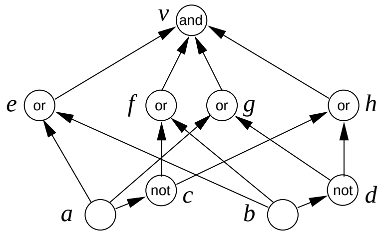
- A Boolean circuit with output gate g and variables $x_1, \ldots, x_n$ computes an n-ary Boolean function f if for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = T(g)$ where $T(x_i) = t_i$, $i = 1, \ldots, n$.
- Any n-ary Boolean function f can be computed by a Boolean 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.



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

Boolean Modelling

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

Example

Binary adder. Given input bits a, b and c
compute output bits o_2o_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:

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

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:
  $p = \mathrm{ite}(o_2,x,p_1)$
  $p_1 = \mathrm{or}(a,c)$
  `% full adder; gate` o_1 `omitted`
  $o_2 = \mathrm{or}(l,r)$
  $l = \mathrm{and}(a,b)$
  $r = \mathrm{and}(c,x)$
  $x = \mathrm{xor}(a,b)$
- Now each satisfying truth assignment for the circuit with $\alpha(p) = \mathbf{true}$ 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
  g_{ijk} with $1 \leq i,j \leq n$ and $0 \leq k \leq n$
  h_{ijk} with $1 \leq i,j,k \leq n$
- 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 $k = 1, 2, \ldots, n$:
  $h_{ijk} = \text{and}(g_{ik(k-1)}, g_{kj(k-1)})$
  $g_{ijk} = \text{or}(g_{ij(k-1)}, h_{ijk})$
- g_{1nn} is the output gate of $R(G)$.
- Constraints α:
  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 α on input gates there is at most one possible truth assignment T.
- It can be shown by induction on $k = 0, 1, \ldots, n$ 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.

Example. Reachability with choices

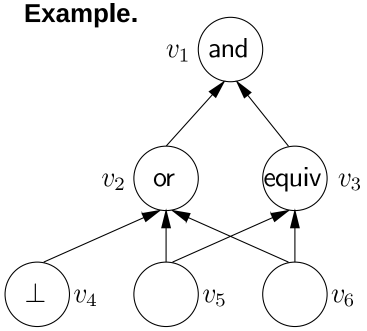
- Consider now a more challenging (search) problem.
- Given a graph $G = (\{1, \ldots, n\}, E)$ and a set of edges $E' \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$, is there a subset $S \subseteq E'$ such that there is a path from 1 to n in $G' = (\{1, \ldots, n\}, E \cup S)$ but not from 1 to $n-1$.
- To solve this problem we can use the circuit $R(G)$ and modify it as follows:
  - remove constraints $\alpha(g_{i,j,0}) = t$ for each edge $(i,j) \in E'$ and
  - add the constraint $\alpha(g_{1,n-1,n}) =$ **false**
- Now the modified $R(G)$ is satisfiable iff there is a set of edges S such that there is a path from 1 to n but not from 1 to $n-1$.
- Moreover, the set of edges S is given by the gates $g_{i,j,0}$ true in a satisfying truth assignment where $(i,j) \in E'$.

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 using 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 is written in a clausal form for each intermediate and output gate and the corresponding literal for each gate g with a constraint $\alpha(g) = t$.
- This transformation preserves satisfiability and even truth assignments in the following sense:
  if C is a Boolean circuit and Σ its Tseitin translation, then for every truth assignment T of C there is a satisfying truth assignment T' of Σ which agrees with T and vice versa.

From Circuits to CNF II

Example.



Gate equations
for non-input gates:
$v_1 \leftrightarrow (v_2 \wedge v_3)$
$v_2 \leftrightarrow (v_4 \vee v_5 \vee v_6)$
$v_3 \leftrightarrow (v_5 \leftrightarrow v_6)$

In CNF:

$$(\neg v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee \neg v_3) \wedge$$
$$(v_2 \vee \neg v_4) \wedge (v_2 \vee \neg v_5) \wedge (v_2 \vee \neg v_6) \wedge (\neg v_2 \vee v_4 \vee v_5 \vee v_6) \wedge$$
$$(v_3 \vee v_5 \vee v_6) \wedge (v_3 \vee \neg v_5 \vee \neg v_6) \wedge (\neg v_3 \vee v_5 \vee \neg v_6) \wedge (\neg v_3 \vee \neg v_5 \vee v_6) \wedge$$
$$(\neg v_4) \text{ [for the constraint } \alpha(v_4) = \textbf{false}]$$