Stanford Little Engines Course (Fall 2003; Lecture 4)

Little Engines of Proof

N. Shankar, L. de Moura, H. Ruess, A. Tiwari

shankar@csl.sri.com

URL: http://www.csl.sri.com/~shankar/LEP.html

Computer Science Laboratory

SRI International

Menlo Park, CA

Stålmarck Method

Stålmarck Method = *Lookahead* + *equivalence classes*.

Input: *triplets* ($p = l_i \wedge l_j$, and $p = l_i \Leftrightarrow l_j$).

Ex: Write a program to convert a formula into a set of triplets.

Configuration: *triplets*, *equalities between literals*.

Equivalence classes are usually used to represent the set of equalities between literals.

$\dfrac{l_1 = l_2, l_2 = l_3}{l_1 = l_2, l_2 = l_3, l_1 = l_3}\,trans$	$\dfrac{l_1 = l_2}{l_1 = l_2, l_2 = l_1}\,symm$
$\dfrac{l_1 = l_2}{\bar{l}_1 = \bar{l}_2}$	$\dfrac{l_1 = l_2, l_1 = \bar{l}_2}{\bot}$

Stålmarck Method: $\wedge$-triplets rules

$\dfrac{p = l_1 \wedge l_2, p = \top}{p = \top, l_1 = \top, l_2 = \top}$	
$\dfrac{p = l_1 \wedge l_2, p = \bar{l}_1}{l_1 = \top, l_2 = \bot, p = \bot}$	$\dfrac{p = l_1 \wedge l_2, p = \bar{l}_2}{l_1 = \bot, l_2 = \top, p = \bot}$
$\dfrac{p = l_1 \wedge l_2, l_1 = \top}{p = l_2, l_1 = \top}$	$\dfrac{p = l_1 \wedge l_2, l_2 = \top}{p = l_1, l_2 = \top}$
$\dfrac{p = l_1 \wedge l_2, l_1 = \bot}{p = \bot, l_1 = \bot}$	$\dfrac{p = l_1 \wedge l_2, l_2 = \bot}{p = \bot, l_2 = \bot}$
$\dfrac{p = l_1 \wedge l_2, l_1 = l_2}{p = l_1, p = l_2, l_1 = l_2}$	$\dfrac{p = l_1 \wedge l_2, l_1 = \bar{l}_2}{p = \bot, l_1 = \bar{l}_2}$

Stålmarck Method: $\Leftrightarrow$-triplets rules

$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = \top}{p = l_2, l_1 = \top}$	$\dfrac{p = l_1 \Leftrightarrow l_2, l_2 = \top}{p = l_1, l_2 = \top}$
$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = \bot}{p = \bar{l}_2, l_1 = \bot}$	$\dfrac{p = l_1 \Leftrightarrow l_2, l_2 = \bot}{p = \bar{l}_1, l_2 = \bot}$
$\dfrac{p = l_1 \Leftrightarrow l_2, p = \top}{p = \top, l_1 = l_2}$	$\dfrac{p = l_1 \Leftrightarrow l_2, p = \bot}{p = \bot, l_1 = \bar{l}_2}$
$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = l_2}{p = \top, l_1 = l_2}$	$\dfrac{p = l_1 \Leftrightarrow l_2, l_1 = \bar{l}_2}{p = \bot, l_1 = \bar{l}_2}$
$\dfrac{p = l_1 \Leftrightarrow l_2, p = l_1}{p = l_1, l_2 = \top}$	$\dfrac{p = l_1 \Leftrightarrow l_2, p = l_2}{p = l_2, l_1 = \top}$
$\dfrac{p = l_1 \Leftrightarrow l_2, p = \bar{l}_1}{p = \bar{l}_1, l_2 = \bot}$	$\dfrac{p = l_1 \Leftrightarrow l_2, p = \bar{l}_2}{p = \bar{l}_2, l_1 = \bot}$

Ex: Show the *triplet* rules are sound.

Stålmarck Method (cont.)

The *triplet* rules are *constraint propagation rules*.

A formula is n-easy if it can be refuted using *LA*(n) and the *triplet* rules.

Strategy: $t\_rules^*; la(1)^*; la(2)^*; la(3)^*; \ldots$.

Stålmarck Method is usually used as an optimization, since it is infeasible to perform $la(n)^*$ ($n \leq 2$) for big formulas.

The Stålmarck Method is a *breadth-first search* procedure.

Remark: the *triplet* rules can be used in a *depth-first search* procedure based on *case-splits*.

Stålmarck Method (example)

Proof of: $F \equiv (p_1 \Leftrightarrow p_2) \wedge (p_2 \Leftrightarrow p_3) \Rightarrow (p_1 \Leftrightarrow p_3)$

Refute: $\Gamma = \{a_1 = (p_1 \Leftrightarrow p_2), a_2 = (p_2 \Leftrightarrow p_3), a_3 = a_1 \wedge a_2, a_4 = (p_1 \Leftrightarrow p_3), a_5 = a_3 \wedge \neg a_4, \neg a_5 = \bot\}$

Γ	$\neg a_5 = \bot$
$\Gamma, a_5 = \top$	$a5 = a_3 \wedge \neg a_4$
$\Gamma, a_5 = a_3 = \neg a_4 = \top$	$a_3 = a_1 \wedge a_2$
$\Gamma, a_5 = a_3 = \neg a_4 = a_1 = a_2 = \top$	$a_1 = (p_1 \Leftrightarrow p_2)$
$\Gamma, a_5 = a_3 = \neg a_4 = a_1 = a_2 = \top, p_1 = p_2$	$a_2 = (p_2 \Leftrightarrow p_3)$
$\Gamma, a_5 = a_3 = \neg a_4 = a_1 = a_2 = \top, p_1 = p_2 = p_3$	$a_4 = (p_1 \Leftrightarrow p_3)$
$\Gamma, \ldots, \neg a_4 = \top, a_4 = \top$	
$\bot$	

F is a 0-easy formula.

Ex: Use *DP* to prove F.

Binary Decision Diagrams

Boolean Functions

$f : \{0,1\}^n \rightarrow \{0,1\}$

A propositional formula F (over $\{p_1, \ldots, p_n\}$) induces a boolean function f:

$f(x_1, \ldots, x_n) = 1$ iff $\{p_1 \leftarrow x_1, \ldots, p_n \leftarrow x_n\}$ satisfies F.

We assume all functions have the same arguments: $\{x_1, \ldots x_n\}$.

Notation:

constant:	$\mathbf{0}$, $\mathbf{1}$
identity:	x_i
negation:	$\bar{f}$
conjunction:	$f \cdot g$
disjunction:	$f + g$

Boolean Functions (definitions)

A restriction or cofactor of f is obtained by replacing one of its arguments by a constant $(0, 1)$:

$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$

The *Shannon expansion* over x_i is:

$f = x_i \cdot f|_{x_i=1} + \bar{x}_i \cdot f|_{x_i=0}$

Ex: Show that *Shannon expansion* is correct.

The *composition* of f and g is:

$f|_{x_i=g}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$

The *dependency set* of a function f contains the arguments on which f depends: $I_f = \{i \mid f|_{x_i=0} \neq f|_{x_i=1}\}$

Examples: $I_{\mathbf{0}} = \emptyset$, $I_{x_1+x_2} = \{1, 2\}$, $I_{x_1 \cdot (x_1+x_2)} = \{1\}$.

Representation of Boolean Functions

There are 2^n elements in the set $0, 1^n$, so there are 2^{2^n} boolean functions.

Every representation consumes *exponential size* in the *worst-case*.

Forms to represent boolean functions:

- Truth table
- List of cubes: Sum of Products, DNF
- List of conjuncts: Product of sums, CNF
- Boolean formula
- Binary Decision Tree, Binary Decision Diagram

Binary Decision Trees

A *binary decision tree* contains two kinds of vertices: *terminal vertices* and *nonterminal vertices*.

A *terminal vertex* v has no children and its labeled by $value(v) \in \{0, 1\}$.

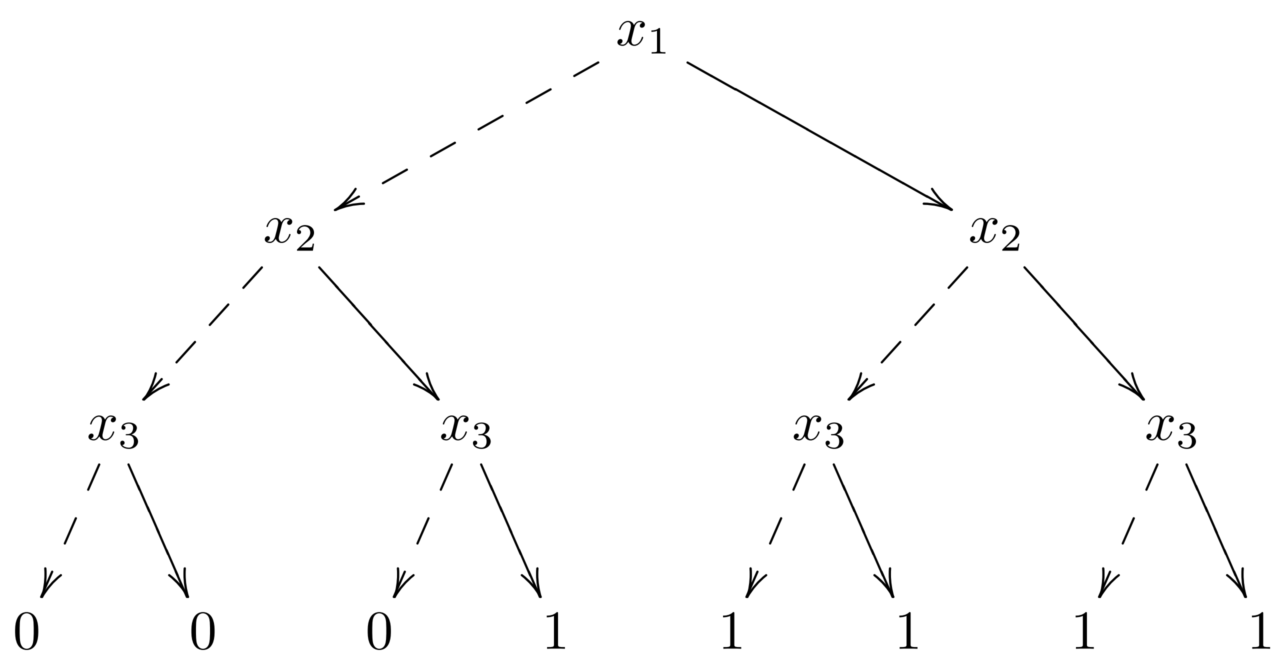
A *nonterminal vertex* v is labeled by a variable $var(v)$ and has two children:

- $low(v)$ corresponding to the case where $var(v)$ is assignment to 0.
- $high(v)$ corresponding to the case where $var(v)$ is assignment to 1.

Ex: Show that every vertex of a binary decision tree corresponds to a boolean function.

Binary Decision Trees (Example)

Binary decision tree for the function: $f \equiv x_1 + (x_2 \cdot x_3)$ (dashed is *low*, solid is *high*).



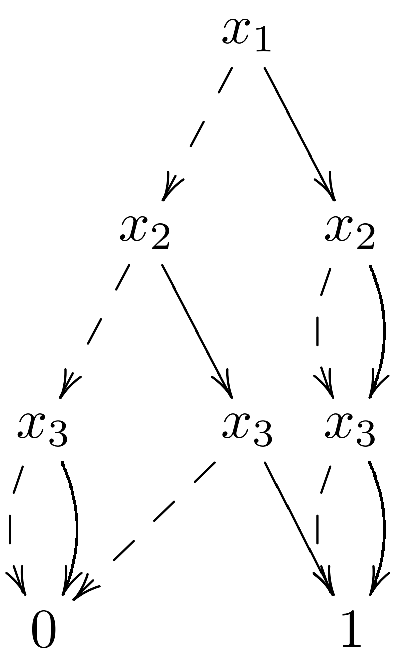
To find the value of the function for a given truth assignment, simply traverse the tree from the root.

Binary Decision Diagrams (BDD's)

Binary decision trees usually contain a lot of redundancy.

Optimization: *merge isomorphic subtrees*.

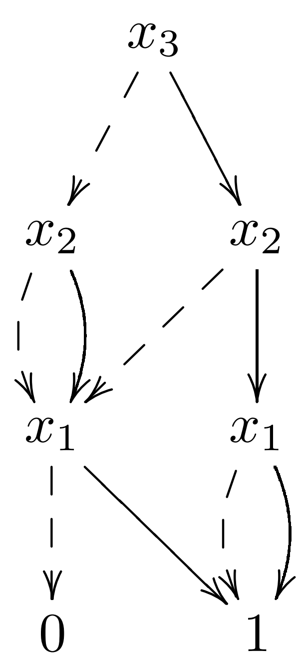
The result is a directed acyclic graph (DAG), called a binary decision diagram (BDD).



Ordered Binary Decision Diagrams (OBDD's)

Given an ordering $\prec$ of the variables $\{x_1, \ldots, x_n\}$. An *ordered binary decision diagram* (OBDD) is a BDD which satisfies the properties $var(v) \prec var(low(v))$, and $var(v) \prec var(high(v))$ for each vertex v.

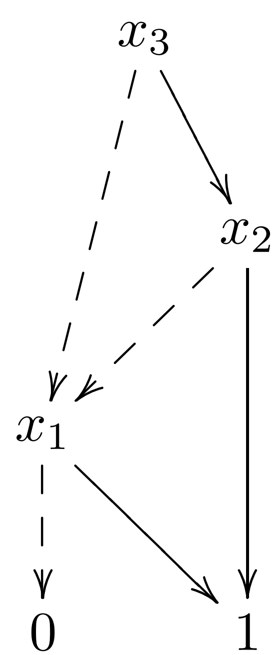
Example: $x_3 \prec x_2 \prec x_1$



Reduced Binary Decision Diagrams

Optimization: Eliminate vertices v such that $low(v) = high(v)$.

Reduced Ordered Binary Decision Diagrams (ROBDD) are commonly used to represent *boolean functions*.



Exercises

1. Construct the OBDD for the boolean function $(a_1 \cdot b_1) + (a_2 \cdot b_2) + (a_3 \cdot b_3)$ with ordering $a_1 \prec a_2 \prec a_3 \prec b_1 \prec b_2 \prec b_3$.

2. Construct the OBDD for the boolean function $(a_1 \cdot b_1) + (a_2 \cdot b_2) + (a_3 \cdot b_3)$ with ordering $a_1 \prec b_1 \prec a_2 \prec b_2 \prec a_3 \prec b_3$.

3. Show that for each boolean function f there is an unique ROBDD representing f (hint: induction on the size of I_f).

4. Show that the ROBDD for f is the OBDD with fewer vertices.

Canonicity

ROBDD are *canonical*.

Two boolean functions are *equivalent* iff they are represented by the same ROBDD.

A *tautology* is represented by the ROBDD with a *single vertex labeled 1*.

A formula is *unsatisfiable* iff it is represented by the ROBDD with a *single vertex labeled 0*.

Ex: Implement a linear time algorithm (called Reduce) to convert a OBDD in a ROBDD.

From now on, when we refer to BDD's, we mean ROBDD's.

Computing Function Restrictions(Cofactors)

$f|_{x_i=b}$ can be computed in linear time using a *depth-first traversal*.

Main idea: for any vertex v_1 which has a reference to a vertex v_2 such that $var(v_2) = x_i$, we replace the reference with $low(v_2)$ if $b = 0$ and $high(v_2)$ if $b = 1$.

We must apply *Reduce* to ensure the result is canonical.

Ex: Implement the restriction(cofactor) algorithm.

Ex: Show the algorithm is correct.

Apply Operation

All binary boolean operators $\odot$ on ROBDD's are implemented using the $apply(\odot, v_1, v_2)$ operation.

For all *boolean operators* $\odot$ the following holds:
$f \odot g = \bar{x} \cdot (f|_{x=0} \odot g|_{x=0}) + x \cdot (f|_{x=1} \odot g|_{x=1})$

The result of $apply(\odot, v_1, v_2)$ is constructed by recursively constructing the *low* and *high*-branches.

We ensure the result is reduced.

We avoid an exponential blow-up of *recursive calls* by using *dynamic programming*.

Optimization: When operating on a terminal node with a *dominant value*(e.g., 1 is the dominant value for $+$) then return the terminal value.

Apply Operation (pseudo-code)

$apply(\odot, v_1, v_2) =$

$mk\_leaf(value(v_1) \odot value(v_2))$ if v_1 and v_2 are terminal

$mk\_node(var(v_1), l', h')$ if $var(v_1) = var(v_2)$

where:

$l' = apply(\odot, low(v_1), low(v_2))$

$h' = apply(\odot, high(v_1), high(v_2)))$

$mk\_node(var(v_1), l', h')$ if $v_1 \prec v_2$

where:

$l' = apply(\odot, low(v_1), v_2)$

$h' = apply(\odot, high(v_1), v_2)$

$mk\_node(var(v_1), l', h')$ if $v_2 \prec v_1$

where:

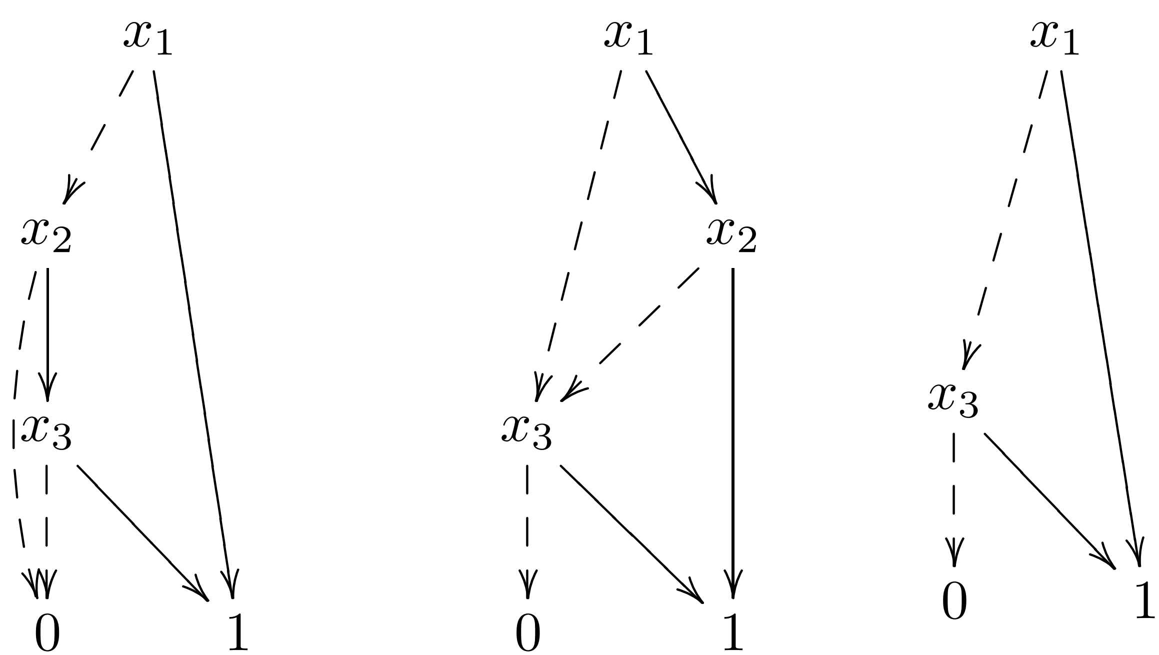
$l' = apply(\odot, v_1, low(v_2))$

$h' = apply(\odot, v_1, high(v_2)))$

Remark: $mk\_node(mk\_leaf)$ creates a reduced non-terminal(terminal).

Apply Operation (example)

$f = x_1 + (x_2 \cdot x_3)$ $g = (x_1 \cdot x_2) + x_3$ $f + g$



Ex: Implement the *apply* operation.

Ex: Show that using *dynamic programming* the complexity of *apply* is $O(|v_1| \times |v_2|)$.

Quantified Boolean Formulas

Quantified Boolean Formulas (QBF): propositional logic with quantifiers $\exists$, $\forall$.

Deciding satisfiability of *QBF* is *PSPACE-complete*.

$\exists x.f = f|_{x=0} + f|_{x=1}$

$\forall x.f = f|_{x=0} \cdot f|_{x=1}$

Variable Ordering

ROBDD's are unique for given variable order.

Ordering can have large effect on size.

Finding good ordering is essential (*NP-complete*).

Simple heuristic: related variables should be close to each other.

Dynamic Variable Reordering: variable order changes as computation progress (*invisible* to the user).

Example: simple *greedy* algorithm. Choose a variable; Try all positions in the variable order (swap); Move to best position found.

Ex: Implement the simple dynamic reordering heuristic.

Applications

Equivalence of Combinatorial Circuits.

Logic Synthesis.

Symbolic Model Checking.

Data-structure for representing (huge) finite sets.

Data Compression.

Constraint Satisfaction Problems.

Ex: Solve the N-Queen Problem using ROBDD's.

Available BDD packages

CUDD (http://vlsi.colorado.edu/ fabio/CUDD)

BuDDy (http://www.itu.dk/research/buddy)

MuDDy - ML interface for BuDDy (http://www.itu.dk/research/muddy)

CAL (Berkeley) (http://www-cad.eecs.berkeley.edu/Respep/Research/bdd/cal_bdd)