Logic

Why Study Logic?

- Logic provides a basis for
  - computer hardware
  - computer programming
  - program optimization
  - specification
  - verification and testing

In a certain sense

Computing *is* Logic

Is all Logic Computing?

No, but

some logic can be *reduced to* computing.

Flavors of Logic

- Proposition Logic
- Predicate Logic
- Temporal Logic
- Modal Logics
- Programming Logics
- Fuzzy Logic

Studied in CS 42 (Proposition Logic, Predicate Logic)

Some exposure in CS *81* (Temporal Logic, Modal Logics, Programming Logics)

Some exposure in CS152 (Neural Networks) (Fuzzy Logic)

Proposition Logic

- Also known as Switching Logic
- Basic elements are
  - 0 (false)
  - 1 (true)
  - proposition variables (take values 0 or 1)
  - either
    - functions (functional view)
    - connectives (expression view)

Mostly we use

- the function view
- and occasionally the expression view

Expressions to Functions

Analogous to lambda notation:

- $x + yz$ is an expression, not a function
- $f(x, y, z) = x + yz$
- An expression E can be thought of as a function (lambda …vars… E)

Proposition Logic Domain

{false, true} (Java)

or

{#f, #t} (Racket)

or

{0, 1} (more readable)

or

{⊥, ⊤ } (more symmetric)

Proposition Logic Functions

- and
- or
- not
- implies
- iff (if, and only if)
- others

and

form 1 table:

x	y	and(x, y)
0	0	0
0	1	0
1	0	0
1	1	1

arguments results

and

form 2 table:

and(x, y)		y: 0	y: 1
	0	0	0
x	1	0	1

results

and

association list version

```
'(((0 0) 0)
  ((0 1) 0)
  ((1 0) 0)
  ((1 1) 1))
```

common *and* symbols

- infix $\wedge$ (mathematical logic)
- infix $\cdot$ (engineering)
- juxtaposition, as in xy
- infix && (Java, C, ...)
- infix , (Prolog)

Way to Remember *and*

- Let x represent any truth value.
- 0 and x is always 0
- 1 and x is always x
- and is symmetric (commutative):

  x and y = y and x

or

form 1 table:

x	y	or(x, y)
0	0	0
0	1	1
1	0	1
1	1	1

or

form 2 table:

or(x, y)	0	1
0	0	1
1	1	1

common *or* symbols

- infix ∨ (mathematical logic)
- infix + (engineering)
- infix || (Java, C, ...)
- infix + (a08)
- infix ; (Prolog)

Way to Remember *or*

- Let x represent any truth value.
- 0 or x is always x
- 1 or x is always 1
- or is symmetric (commutative):

  x or y = y or x

and and *or*

- 0 and x is
- 1 and x is

- 0 or x is
- 1 or x is

not

form 1 table same as form 2 table:

x	not(x)
0	1
1	0

common *not* symbols

- prefix ¬ (mathematical logic)
- postfix ' , $\overline{\text{overbar}}$ (engineering)
- prefix ! (Java, C, ...)
- prefix \+ (Prolog)

implies

form 1 table:

x	y	implies(x, y)
0	0	1
0	1	1
1	0	0
1	1	1

implies

form 2 table:

implies(x, y)		y: 0	1
	0	1	1
x	1	0	1

common *implies* symbols

- infix $\Rightarrow$ $\rightarrow$ $\supset$(mathematical logic)
- Often a $\rightarrow$ b can be treated as an abbreviation for $\neg$a $\vee$ b
- **Note**: Binding tightest to weakest: $\neg$, $\wedge$, $\vee$

Way to Remember *implies*

- Let x represent any truth value.
- 0 implies x is always 1
- 1 implies x is always x
- implies is **not symmetric**

and, or, implies

- 0 and x is
- 1 and x is

- 0 or x is
- 1 or x is

- 0 implies x is
- 1 implies x is

iff

form 1 table:

x	y	iff(x, y)
0	0	1
0	1	0
1	0	0
1	1	1

iff

form 2 table:

iff(x, y)	0	1
0	1	0
1	0	1

common *iff* symbols

- infix $\equiv$ $\leftrightarrow$ $\Leftrightarrow$ (mathematical logic)
- infix == (Java, C)

xor (exclusive or)

form 1 table:

x	y	xor(x, y)
0	0	0
0	1	1
1	0	1
1	1	0

xor

form 2 table:

iff(x, y)	0	1
0	0	1
1	1	0

Expression Forms

- Use for greater readability of certain equalities
- Similar to ordinary discourse
- Binding order, tightest to weakest: not, and, or

Logical Expression Equivalences engineering style

- $a\ b = b\ a$ — Commutative
- $a + b = b + a$

- $(a\ b)\ c = a\ (b\ c)$ — Associative
- $(a + b) + c = a + (b + c)$

- $(a + b)\ c = (a\ c) + (b\ c)$ — Distributive
- $(a\ b) + c = (a + c)\ (b + c)$

Important Logical Equivalences

- $(a\,b)' = (a' + b')$
- $(a + b)' = (a'\,b')$

DeMorgan's Laws

- $(a + a'\,b) = a + b$
- $(a\,(a' + b)) = a\,b$

Worth-remembering laws

Logical Equivalences for Implies

- $(a \rightarrow b) = (a' + b)$
- $(a \rightarrow b) = (ab')'$

- $(0 \rightarrow b) = 1$
- $(1 \rightarrow b) = b$
- $(a \rightarrow 0) = a'$
- $(a \rightarrow 1) = 1$

More Logical Equivalences for Implies

- $(a \rightarrow bc) = (a \rightarrow b)\ (a \rightarrow c)$
- $((a + b) \rightarrow c) = (a \rightarrow c)\ (b \rightarrow c)$
- $((a \rightarrow b)\ (b \rightarrow c)) \rightarrow (a \rightarrow c)$
- $(a \rightarrow b) = (b' \rightarrow a')$
- $(a \rightarrow b)' = (a\ b')'$

Checking Tautology using the **Boole-Shannon Principle**

- Relations hold **iff** they hold for **every substitution** of 0 and 1 for the variables (**uniformly** throughout the expression)
- Therefore, a relation holds if, choosing *any* variable V, it holds for V = 0 and for V = 1.
- But substituting 0 or 1 for a variable often yields **simplifications** that make the relation obvious.

Simple Ways to Check Identities

- Pick any variable in the identity.
- Replace the identity with two separate identities:
  - One where the variable has value 0
  - The other where the variable has value 1
- Check that **both** identities are true.

Example

- $((a + b) \rightarrow c) = (a \rightarrow c)\ (b \rightarrow c)$
- Pick c as the variable, giving:
  - $((a + b) \rightarrow 0) = (a \rightarrow 0)\ (b \rightarrow 0)$
  - $((a + b) \rightarrow 1) = (a \rightarrow 1)\ (b \rightarrow 1)$
- These equations are equivalent to:
  - $(a + b)' = a'\ b'$ (one of deMorgan's laws)
  - $1 = 1\ 1$ (obviously correct)
- Conclusion: The identity is correct.

Making Your Work Easier

- Analyze the simpler sub-goal first. If it is invalid, you can stop there.
- Use recursion if necessary (pick another variable).
- The process can be described as a tree.

Which of these are Correct?

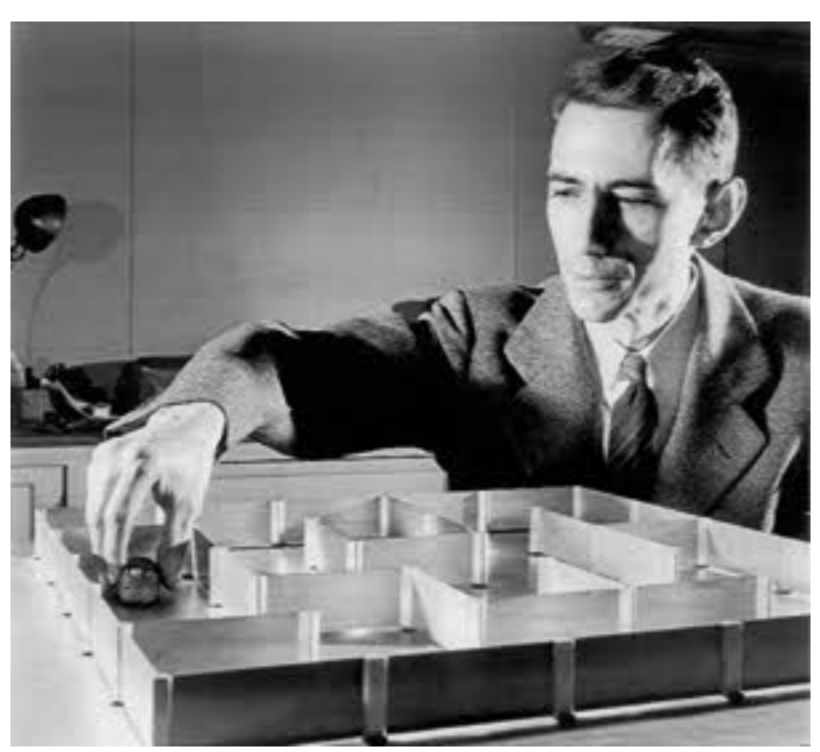
- $((a \rightarrow c) + (b \rightarrow c)) = ((a + b) \rightarrow c)$
- $(ab \rightarrow c) = (a \rightarrow c)\,(b \rightarrow c)$
- $(ab \rightarrow c) = (a \rightarrow c) + (b \rightarrow c)$

Boole and Shannon

- Boole
  - Invented "Boolean algebra" (switching theory)
  - (In modern mathematics, "Boolean algebra" is a more general, abstract, system)
- Shannon
  - Wrote thesis on switching theory
  - Invented "Information theory"
  - Maze-solving mouse
  - Wrote first chess-playing program
  - Wrote paper on the mathematics of juggling

Boole and Shannon

George Boole (1815-1864)



Claude Shannon (1916-2001)

Truth Assignment or “Combination”

- By a ***truth assignment*** or ***combination*** for a formula, we mean a function that assigns a value to every variable in the formula.
- Example formula

$$((a + b) \rightarrow c) = (a \rightarrow c) + (b \rightarrow c)$$

- **One** combination a:0, b:1, c:0
  or [a, b, c]:[0, 1, 0]
- A formula together with a combination has a **value** in the obvious way.

Counting Combinations

For a formula with n distinct variables, how many combinations are there?

Tautologies, etc.

- A formula that has value true for **every** combination is called a ***tautology***, or is said to be ***valid***.
- A formula that has value true for **some** (including possibly every) combination is said to be ***satisfiable***.
- A formula that has value true for **no** combination is called a ***contradiction***, or is said to be ***unsatisfiable***.

A Trichotomy

- Every formula is exactly one of:
  - Tautology
  - Satisfiable but not a tautology
  - Unsatisfiable

- Formula F is a tautology iff F' is unsatifiable.

Counterexamples

- To establish that a function is not a tautology, it suffices to give a single combination where it is 0.
- This combination is called a **counterexample**.

Constructing a Truth Table

- A truth table is an enumeration of a function for all possible combinations.
- Make a table with each variable as a column header and a column for the expression to be checked.
- Enumerate all combinations for the variables.
- Evaluate the expression for each combination.

Example: Checking Tautology using Full Enumeration or “Truth Table” method

a	b	c	$((a + b) \rightarrow c) = (a \rightarrow c)\ (b \rightarrow c)$
0	0	0	0 0 0 0 0 0 0
			0 1 1
			1 1
			1

Example: Checking Tautology using Full Enumeration or "Truth Table" method

a	b	c	$((a + b) \rightarrow c) = (a \rightarrow c)(b \rightarrow c)$
0	0	0	0 0 1 0 1 0 1 0 1 0 1 0
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Example: Checking Tautology using Full Enumeration or "Truth Table" method

a	b	c	$((a + b) \rightarrow c) = (a \rightarrow c)\ (b \rightarrow c)$
0	0	0	1
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Example: Checking Tautology using Full Enumeration or "Truth Table" method

a	b	c	$((a + b) \rightarrow c) = (a \rightarrow c)(b \rightarrow c)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

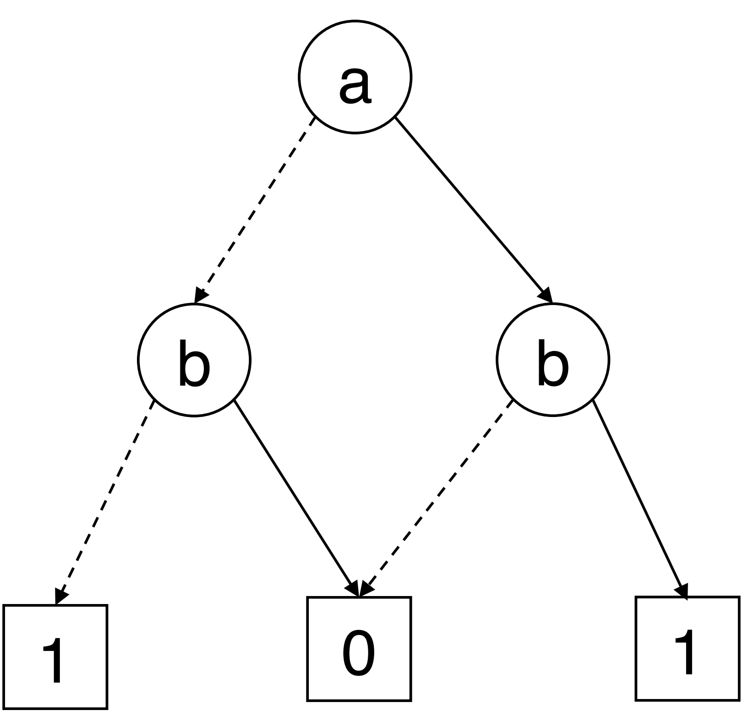
Binary Decision Diagrams (BDD's)

- A way to evaluate an expression
- Used extensively in computer engineering

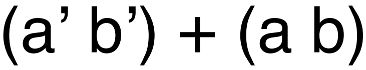
Binary Decision Diagrams (BDD's)

- A BDD is a directed acyclic graph.
- The leaves are labeled either 0 or 1.
- Each non-leaf node is
  - labeled with a variable
  - has two arcs leaving it:
    - true branch ———
    - false branch - - - - -

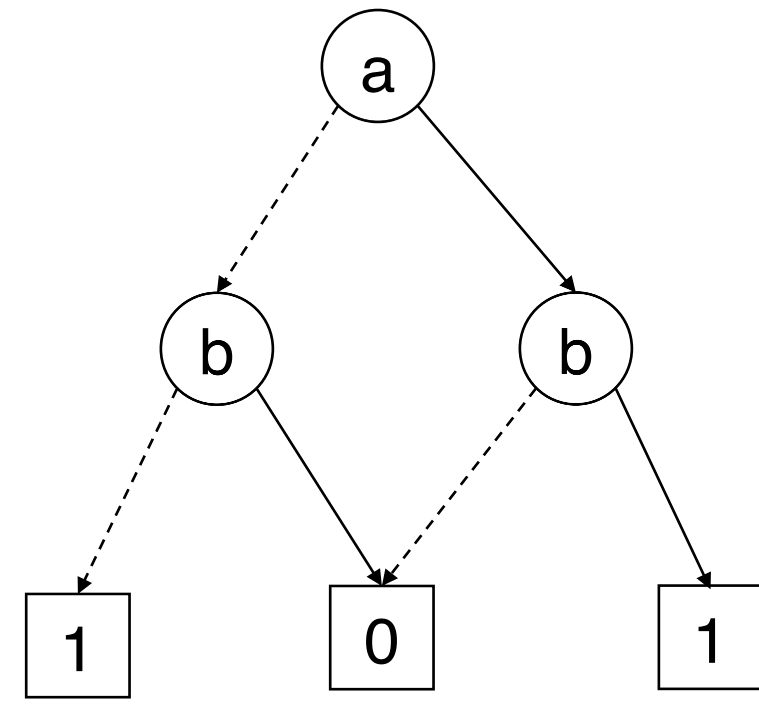
Example BDD



Each BDD represents some formula



a = b



Constructing a BDD from a Formula using the Boole-Shannon Expansion

- ConstructBDD(F):
  - If F contains no variable, return a node with its value (1 or 0).
  - Choose a variable v in F. Let $F_{v=0}$ mean F with all instances of v replaced with 0 (false), and similarly for $F_{v=1}$.
  - Return a tree with root labeled v, with the false branch connecting to ConstructBDD($F_{v=0}$) and the true branch connecting to ConstructBDD ($F_{v=1}$).

Constructing a BDD from a Formula using the Boole-Shannon Expansion

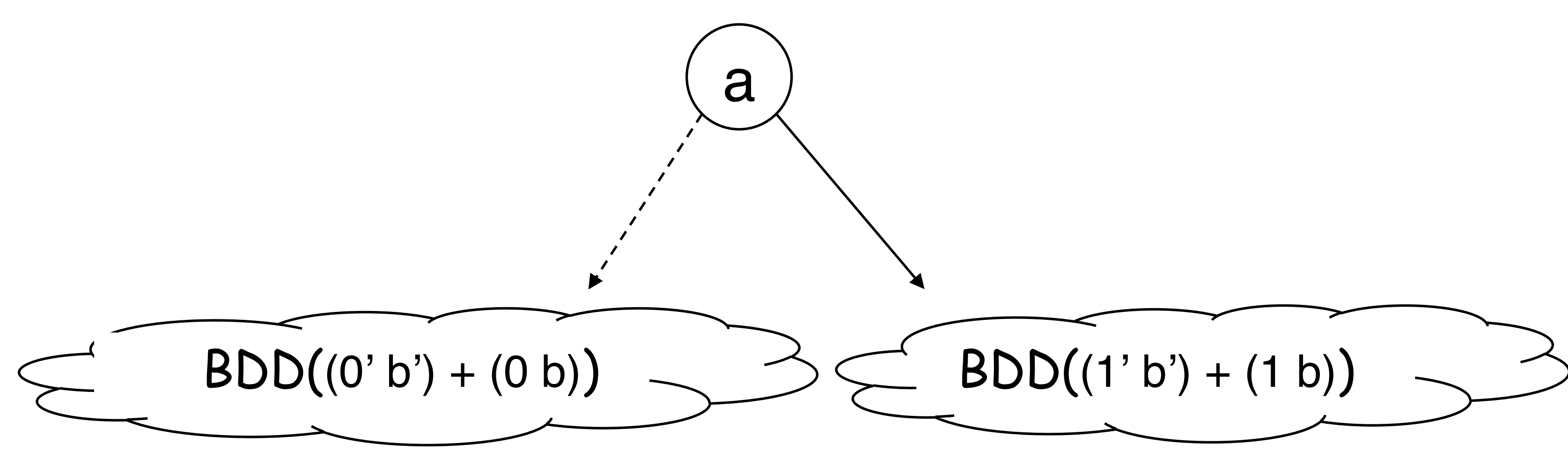
- ConstructBDD((0′ 1′) + (0 1)):
- The basis rule applies:
  - Evaluate: 0
  - Return

0

Constructing a BDD from a Formula using the Boole-Shannon Expansion

ConstructBDD((a′ b′) + (a b)):

- The basis rule does not apply.
- Choose a variable, say a:
  - Construct BDD((0′ b′) + (0 b))
  - Construct BDD((1′ b′) + (1 b))
  - Connect to node labeled a

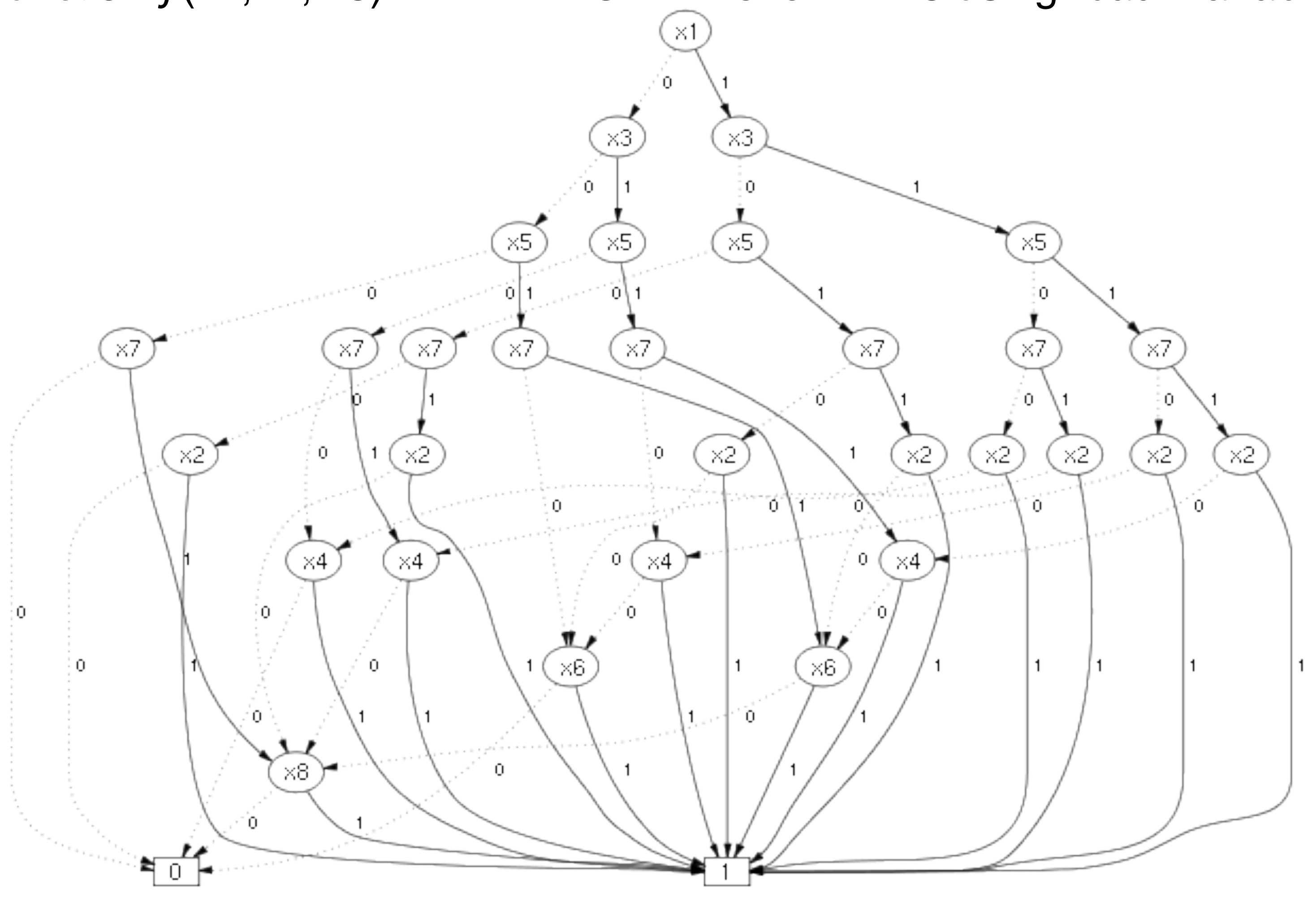


Some Simplifying Formulas using BDD's

- Two nodes that are the roots of **identical sub-graphs** can be merged together (two references sharing a sub-graph).
- A node having **both true and false branches going to the sub-graph** can be replaced with the sub-graph itself.
- Re-ordering nodes sometimes enables the above simplifications.

Example where Ordering Matters

BDD for function $f(x1, ..., x8) = x1x2 + x3x4 + x5x6 + x7x8$ using “bad” variable ordering



Example where Ordering Matters