The Shannon expansion and BDDs (Boolean Decision Diagrams)

Representing Boolean functions in a computer

David Hathaway
david.hathaway.78@gmail.com

(questions welcome during or after the talk)

Topics

- Motivation
- Boolean algebra
- Other Boolean function representations
  - Truth table
  - Sum of products, product of sums
- Shannon expansion
- BDDs

Motivation

1) Testing satisfiability
   - Consider a complex set of interacting conditions
     - Logical constraints on some problem or system
   - Need to know whether we can satisfy all of them
     - Represent each constraint as a separate function
     - Overall constraint is AND of all these
     - Can the overall constraint *ever* be true?
     - Is it possible to meet all the constraints?
   - Called "satisfiability", or SAT
   - Many optimization problems can be cast as SAT

- Dual problem of SAT is tautology -
  Is function *always* true?

Motivation

2) Equivalence checking

- All digital systems are based on logic
  - Signals are 1 (high voltage) or 0 (low voltage)
  - "Gate" circuits perform AND, OR, and NOT functions on input signals
  - Systems are too complex to verify by hand
- Equivalence checking
  - Compares some specification with an implementation
  - Ensures we build what we meant to

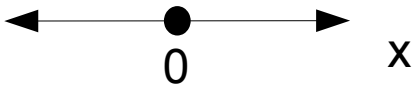
Boolean algebra

- Named for George Boole (1815-1864)
- Applies algebraic approaches to logic
- Only two values: T/F, or 1/0
- AND, OR replace multiplication, addition as basic operations
- Operators we'll use:
  - ~a (NOT) (also !a, $\overline{a}$, ¬a)
    True (1) if a is 0
  - a+b (OR) (also a∨b, a|b)
    True (1) if a is 1, b is 1, or both are 1
  - a*b (AND) (also ab, a∧b, a&b)
    True (1) both a and b are 1
  - a ⊕ b (exclusive OR, or XOR)
    True (1) if exactly one of a and b is 1
    For > 2 input XOR, true (1) if odd number of are 1

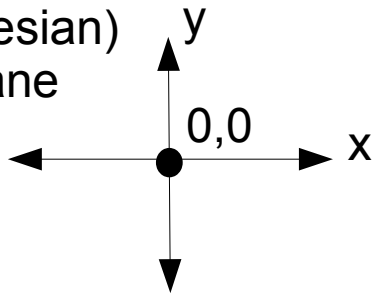
Real vs Boolean space

- Reals - infinite number of values
- Booleans - two values

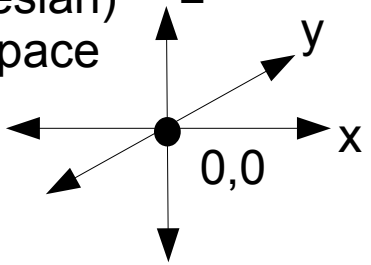
Real number line



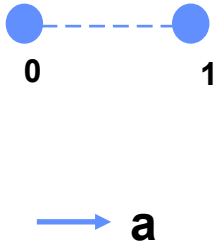
Real (Cartesian) plane



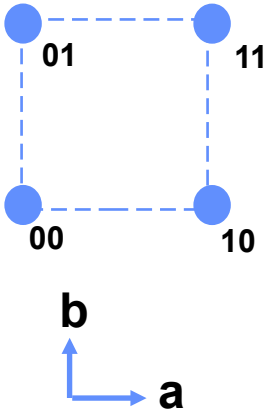
Real (Cartesian) 3D space



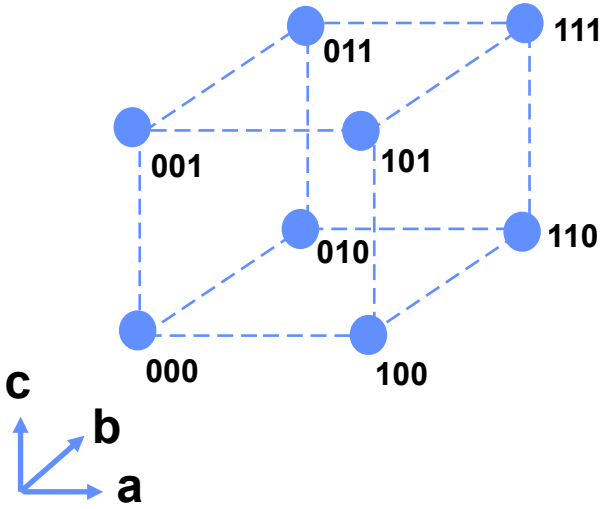
Boolean "number line"



Boolean "plane"



Boolean 3D "space"



Boolean algebra

- Many properties are same as for algebra with reals
  - +, *, ⊕ are commutative, associative
  - 0 + a = a, 1*a = a, 0*a = 0
  - * distributes over +, and + also distributes over *

    a * (b + c) = (a * b) + (a * c)

    a + (b * c) = (a + b) * (a + c)
- Things we'll use today:

  a + ~a = 1 (a must be 1 or 0)

  a * ~a = 0 (a can never be both 1 and 0)
  - Demorgen's law

    a + b = ~(~a * ~b)

    a * b = ~(~a + ~b)

Boolean vs Real functions

- Real functions
  - Infinite domain and range (all real numbers)
  - Infinite number of possible functions
  - Can say interesting things about single variable functions
    sin, tan, $x^3 + 3x^2 + 5x + 2$, ...
- Boolean functions
  - Domain and range are both finite (0, 1)
  - Finite input space: 2^N for N inputs
    - Function completely defined by specifying the 0 or 1 output value for each input combination
    - $2^{(2^N)}$ different possible functions for N inputs

Inputs	# output values	# of possible functions
1	2	4
2	4	16
3	8	256
4	16	64K
5	32	4G
6	64	1.8×10^{19}

→ 0, 1, a, ~a

Boolean function representation

- List of values for each input combination is a ***truth table***
  - Each entry is a ***minterm***, each variable or complement is a ***literal***
  - Truth tables for basic Boolean operators:

NOT

Input	Out
a	~a
0	1
1	0

OR

Inputs		Out
a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

AND

Inputs		Out
a	b	a*b
0	0	0
0	1	0
1	0	0
1	1	1

XOR

Inputs		Out
a	b	a⊕b
0	0	0
0	1	1
1	0	1
1	1	0

Boolean function representation

- Use truth tables to show some Boolean algebra properties:

OR

Inputs		Out
a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

a+~a = 1

a	~a	a+~a
0	1	1
1	0	1

a + b = ~(~a * ~b)

a	b	a+b	~a	~b	~a*~b	~(~a*~b)
0	0	0	1	1	1	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	1	0	0	0	1

same

DeMorgen‘s Law

AND

Inputs		Out
a	b	a*b
0	0	0
0	1	0
1	0	0
1	1	1

a*~a = 0

a	~a	a*~a
0	1	0
1	0	0

a * b = ~(~a +~b)

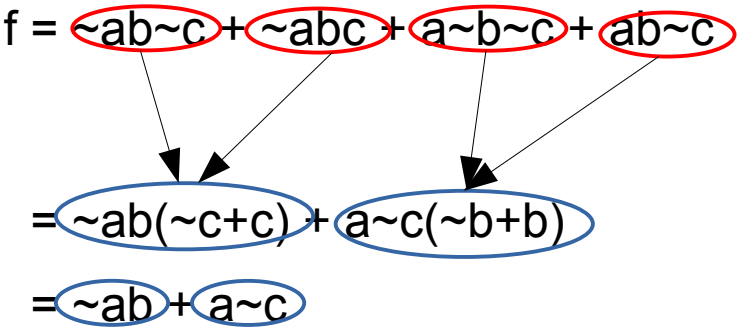
a	b	a*b	~a	~b	~a+~b	~(~a+~b)
0	0	0	1	1	1	0
0	1	0	1	0	1	0
1	0	0	0	1	1	0
1	1	1	0	0	0	1

same

Sum of products

- For large numbers of inputs, truth tables are big
  - 64 bit adder, 129 inputs, $2^{129} \cong 6.8 \times 10^{38}$ (680 exa-exa-bits)
- Truth table can be expressed as sum (OR) of true minterms
- Find sets of TT 1 entries whose inputs cover all possible values
  example: a + b = ab + a~b + ~ab
- Can combine terms to make products with fewer literals, that cover more minterms of function. Example:

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



- Called Sum of Products (SOP) form
- Need at least one product to cover every 1 in TT

Product of sums

- Instead of covering 1s in TT, cover all 0s
- Get SOP for all 0s of function
- Use DeMorgen's Law to convert to Product of Sums (POS)
- Example:

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

~f = ~a~b~c + a~b~c + a~bc

= ~b~c + a~b

f = ~(~b~c + a~b)

apply DeMorgen's Law:

= ~(~b~c) *~(a~b)

...again:

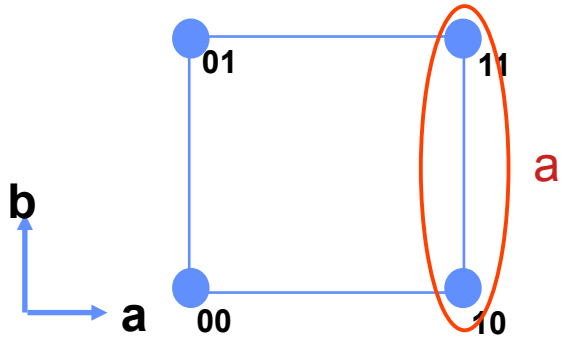
= (~~b + ~~c) *(~a + ~~b)

remove double NOTs:

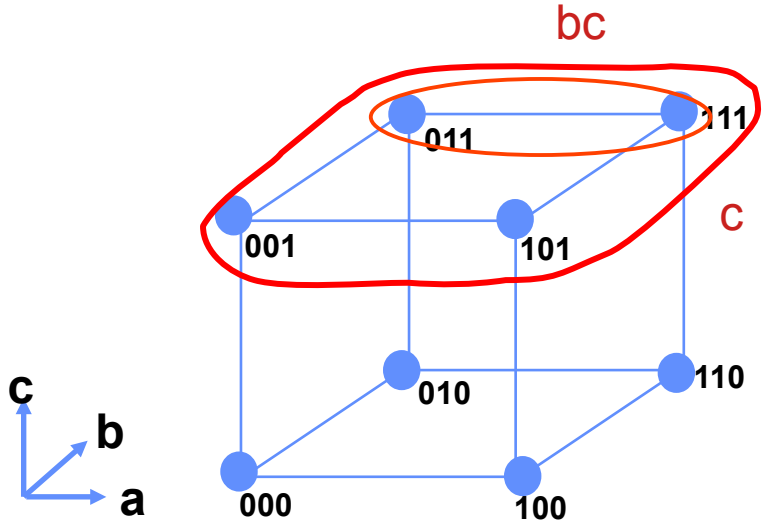
= (b + c) * (~a + b) POS form

Plotting functions in Boolean space

- Another way of looking at a truth table
  - Each vertex is a minterm
  - Lines are N-1 literal products
  - Planes are N-2 literal products



2 variables



3 variables

Issues with SOP, POS

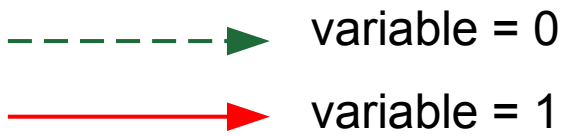
- Still large for common functions
  - Parity (multi-way XOR) truth table can't be simplified
- Many different SOP expressions possible for same function
  - Not canonical
    - Canonical representation of a function means a representation in which the function can be written on only one way
  - Makes it hard to compare functions
- Must redo simplification after Boolean operations

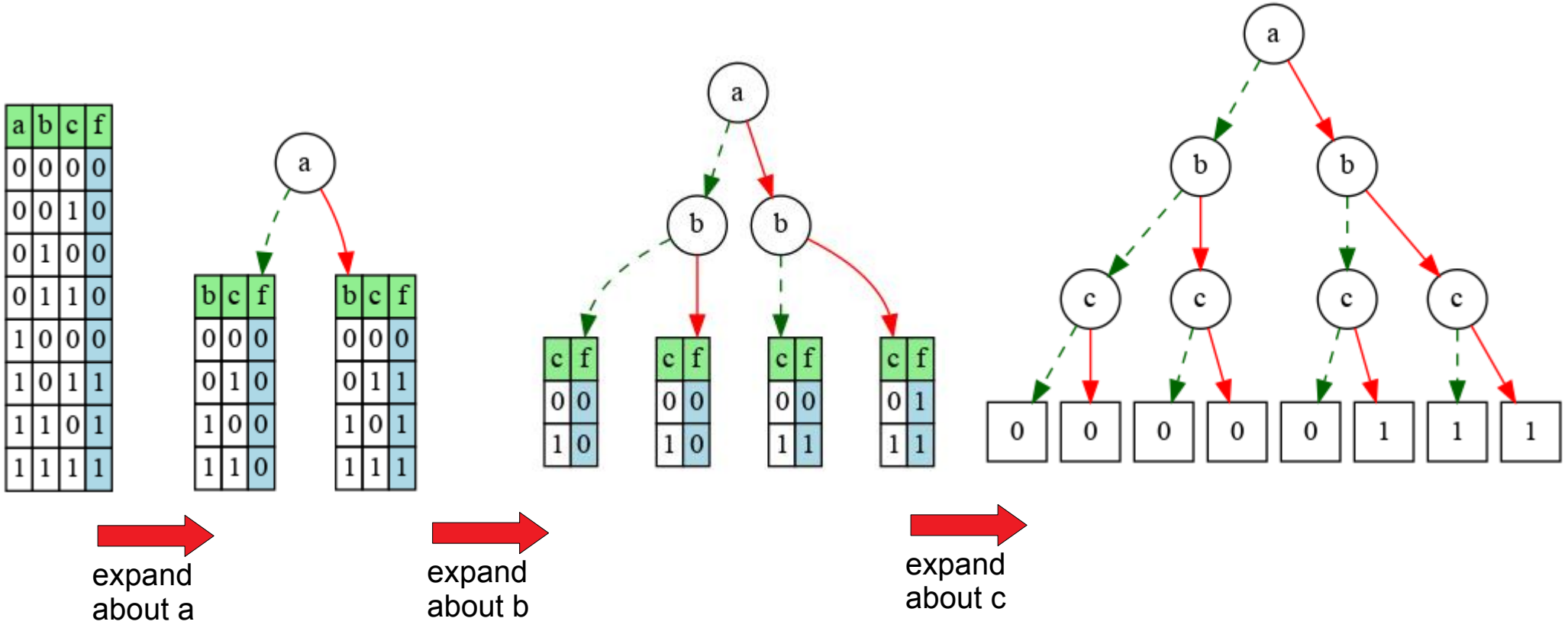
Shannon expansion

- From 1938 paper by Claude Shannon
- Start with function f(a,b,c, ...)
  - Can show f(a,b,c...) = a*f(1,b,c,...) + ~a*f(0,b,c,...)
  - Called the expansion of f about a
- What does this mean?
  - If a is 1, we need to return the value of f when a is 1
  - If a is 0, we need to return the value of f when a is 0
  - a must be 0 or 1, so these are the only two possibilities
- If original function took N inputs, when we fix the value of a, we have new functions of N-1 inputs
- Repeat this, expanding both new functions about b
- After N levels, we have removed all variables, left with 1 or 0

Shannon expansion

- Example: f(a,b,c) = a*(b+c)



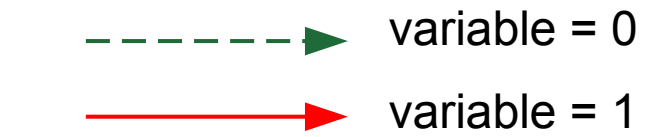


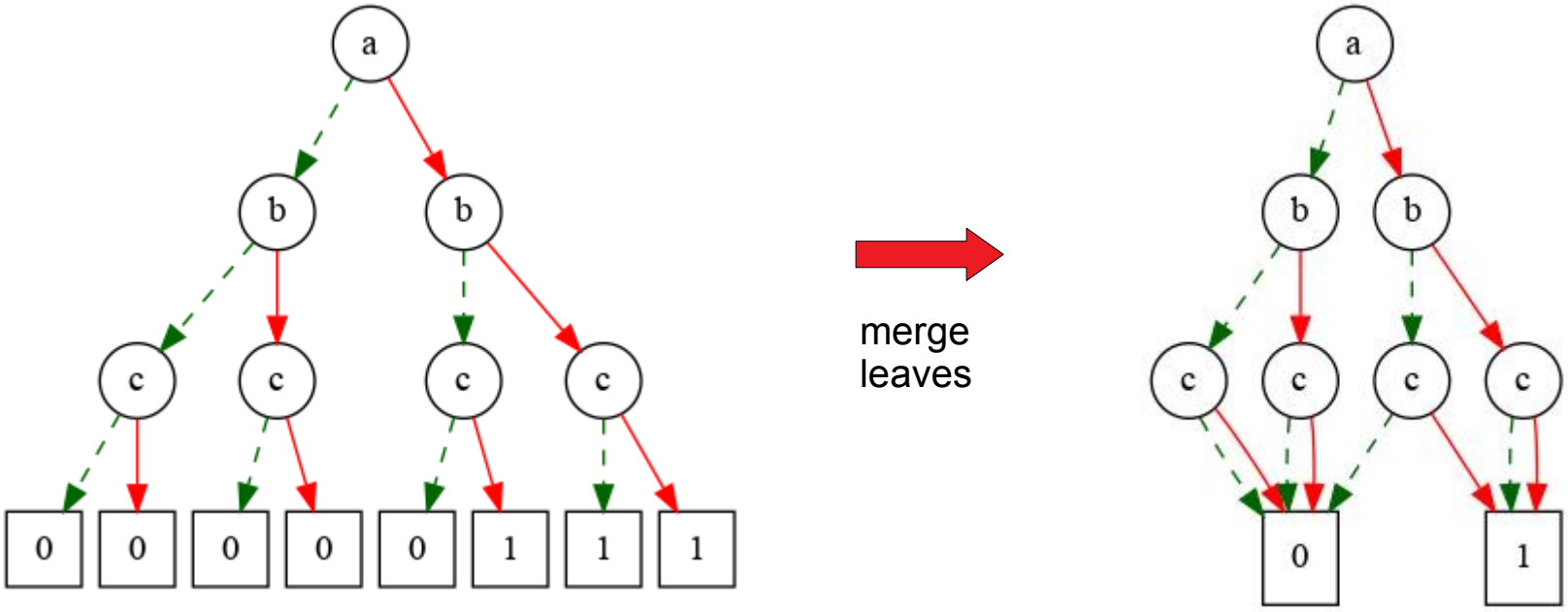
BDDs (Binary Decision Diagrams)

- Shannon expansion creates a tree, but is still big
  - 2^N leaves, same size as truth table
- But notice many leaves are the same (all are 0 or 1)
- Change structure to have only one 1 and one 0 node
  - Many nodes can point to same leaf
  - Changes from tree to DAG
    - DAG - directed acyclic graph
      - directed: lines have arrows
      - acyclic: you can't follow arrows around any loop
      - graph: structure made of nodes and lines

Tree to DAG

- Example: f(a,b,c) = a*(b+c)





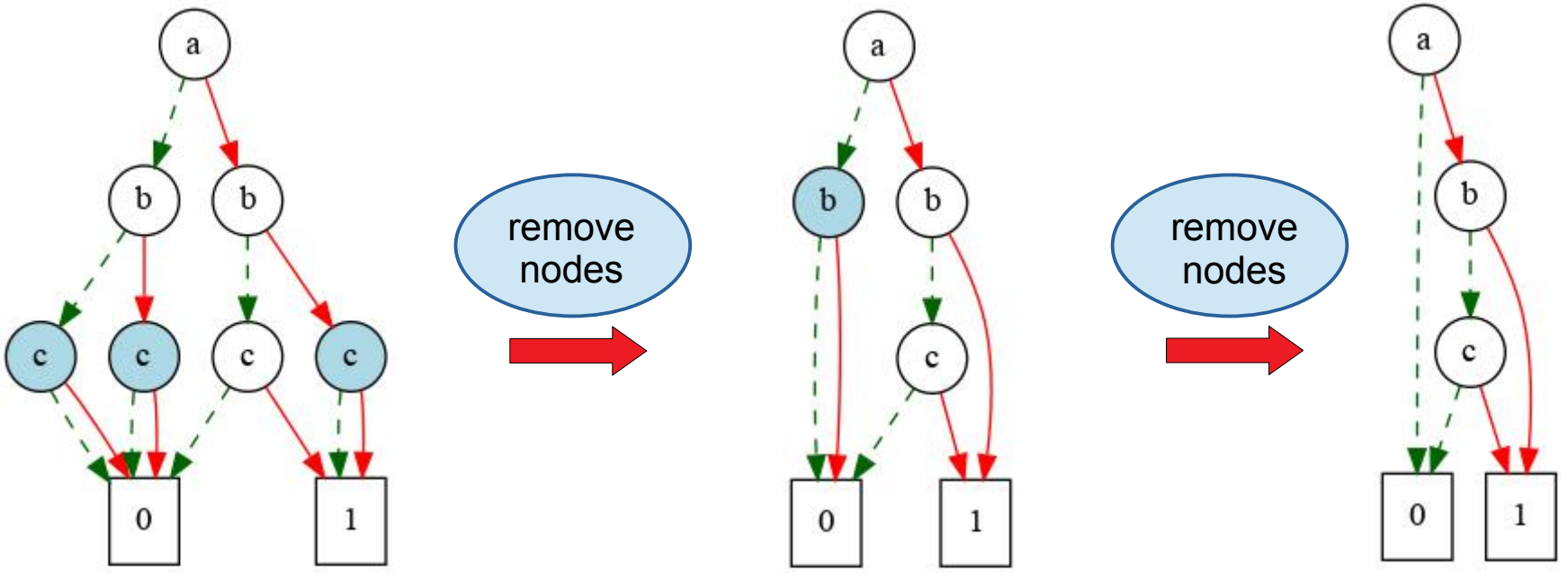
Reducing BDDs

- For some nodes, 0 and 1 arrows point to same node
  - Node function is independent of the node variable
  - Can remove those nodes (skip a level)

f(a,b,c) = a*(b+c)

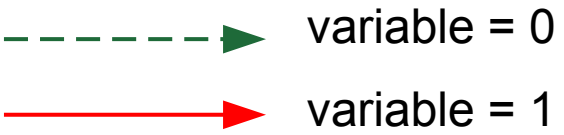
variable = 0

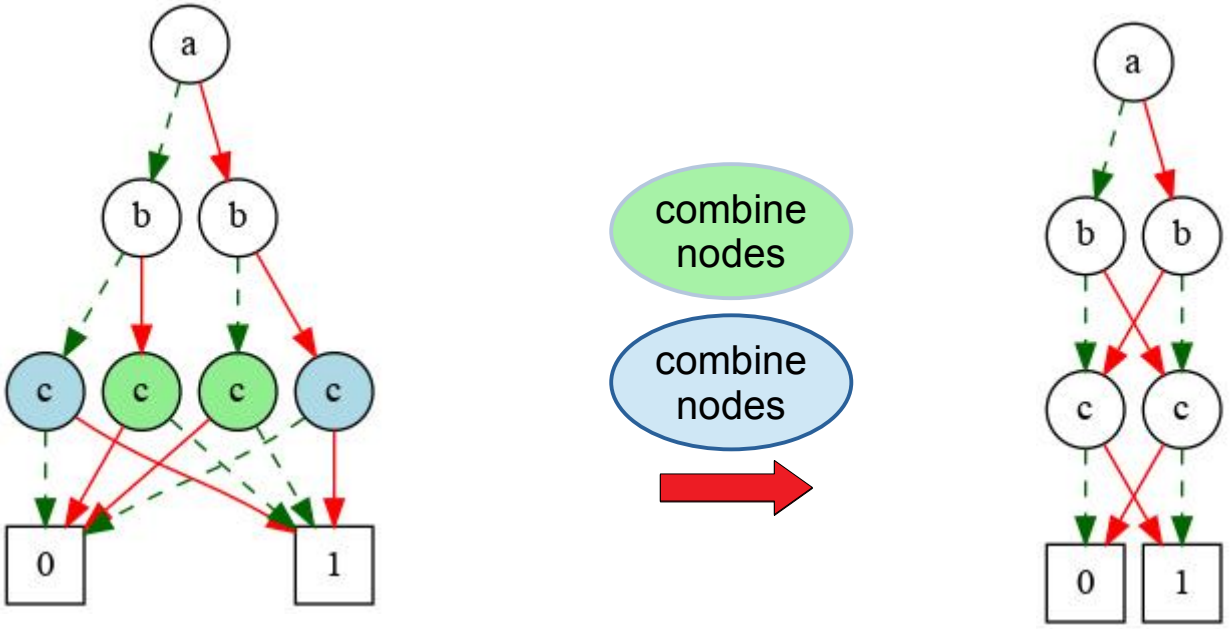
variable = 1



Reducing BDDs

- Some nodes are not redundant, but are identical to others
  - Make one copy of such nodes
  - Example: f = a ⊕ b ⊕ c





Variable ordering

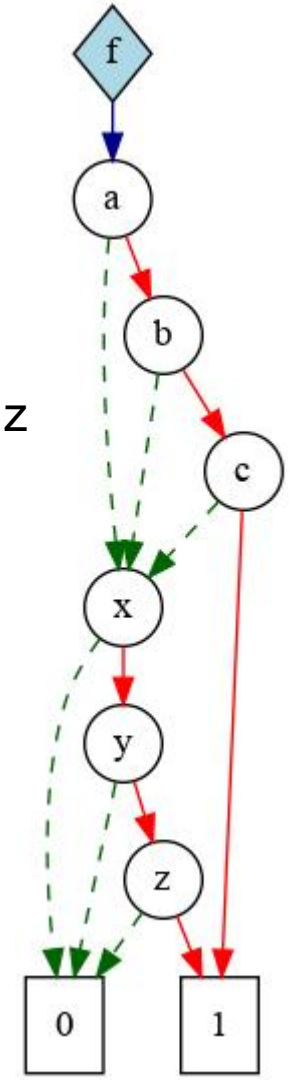
- We could have expanded variables in a different order
  - Could even use different orders on different branches
    - Ordered BDDs - same order on all branches
- Variable ordering affects size of BDD
  - BDD software often includes code to reorder variables
    - Tries to find "good" order to reduce sizes
- Some functions are exponential in size for all orderings
  - Example: binary multiplier
  - BDDs not a good tool for these

Variable ordering

- Example: f = (abc) + (xyz)

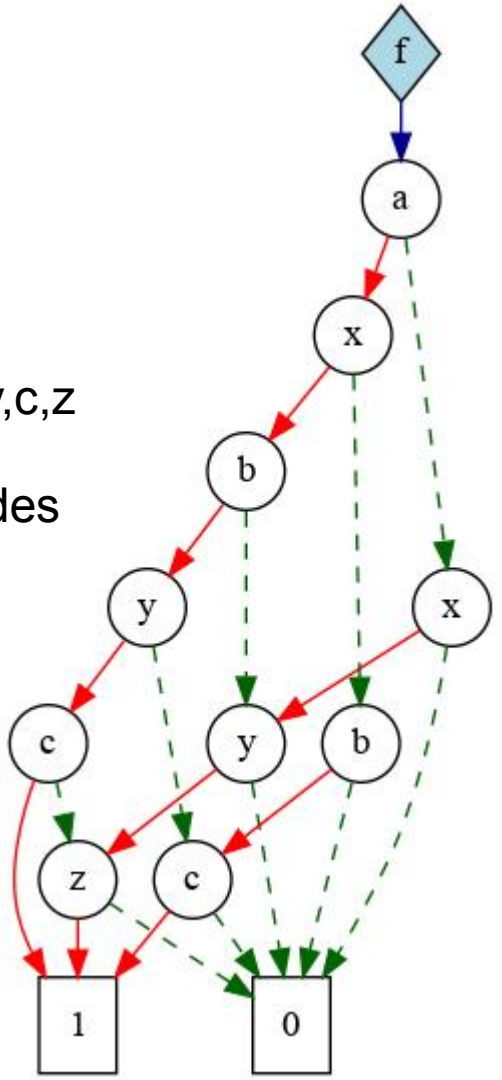
order:
a,b,c,x,y,z

8 nodes



order:
a,x,b,y,c,z

12 nodes

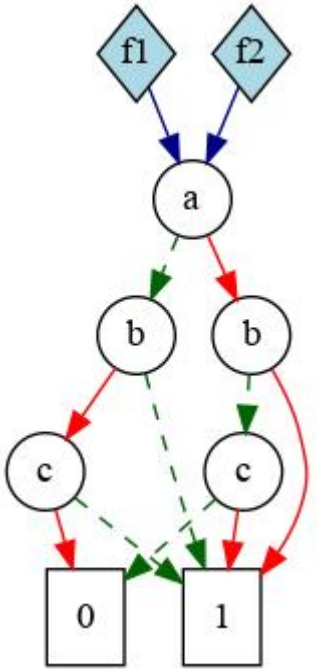


Reduced Ordered BDDs

- Reduced Ordered BDDs (ROBDDs) are canonical
  - Given:
    - A specified variable order
    - All redundant nodes removed
    - No duplicate nodes
  - There is only one BDD for any Boolean function
    - Makes satisfiability & tautology trivial
      - Does function point to 0?
        - If not, it's satisfiable
          (we can pick input values that make it true)
      - Does function point to 1?
        - If so it's a tautology
          (it's always true)

Shared BDDs

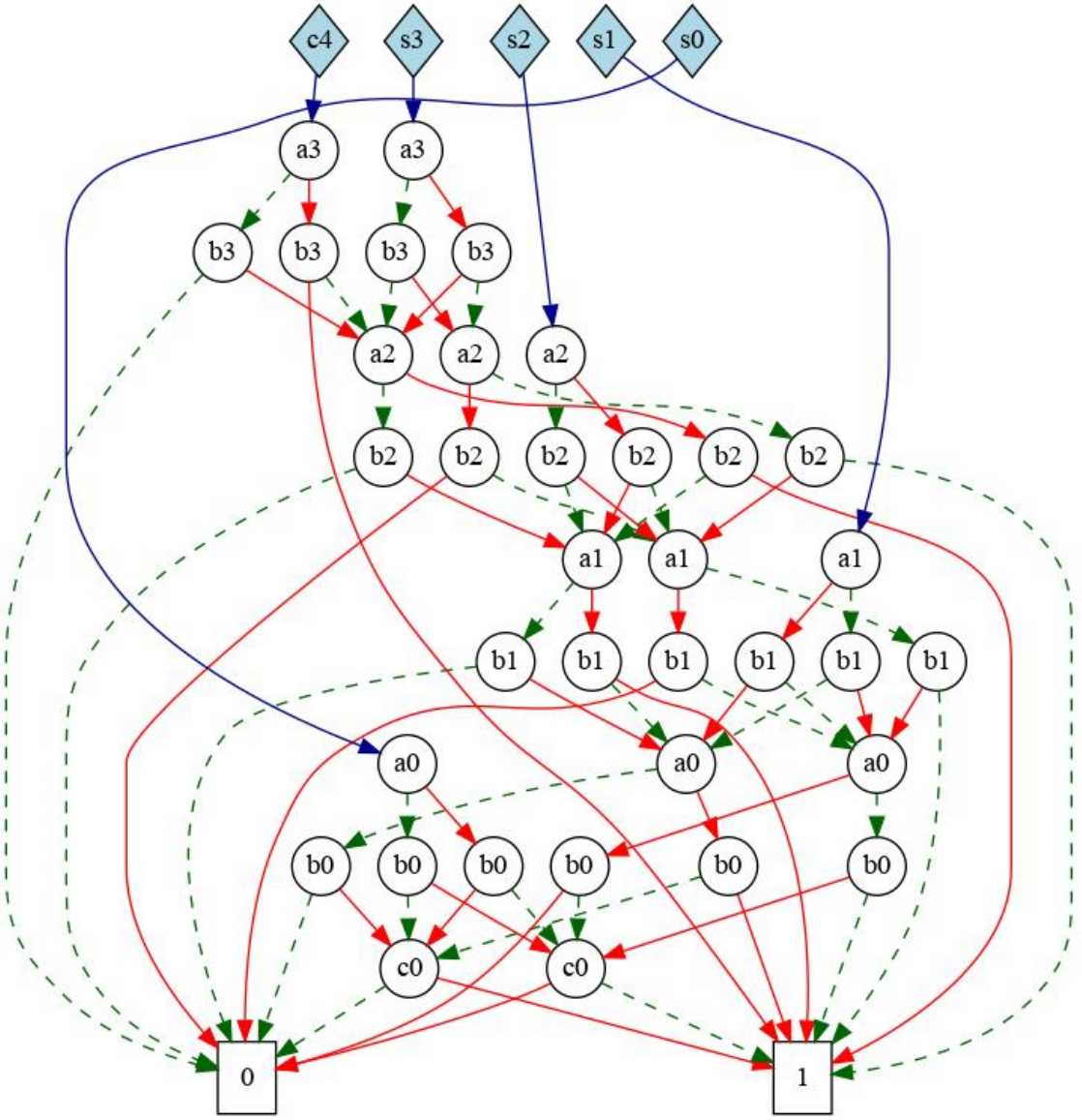
- Build BDDs for many functions at once
  - No duplicate nodes across all BDDs
- Makes equivalence checking trivial:
  - Two functions are the same if and only if they are represented by the same node
- Example: f1 = ~a~b + b~c + ac
  f2 = ~a~c + ~bc + ab



Shared BDDs

- Example: 4 bit adder (arithmetic plus here):

```
   a3a2a1a0
+  b3b2b1b0
+        c0
 c4s3s2s1s0
```

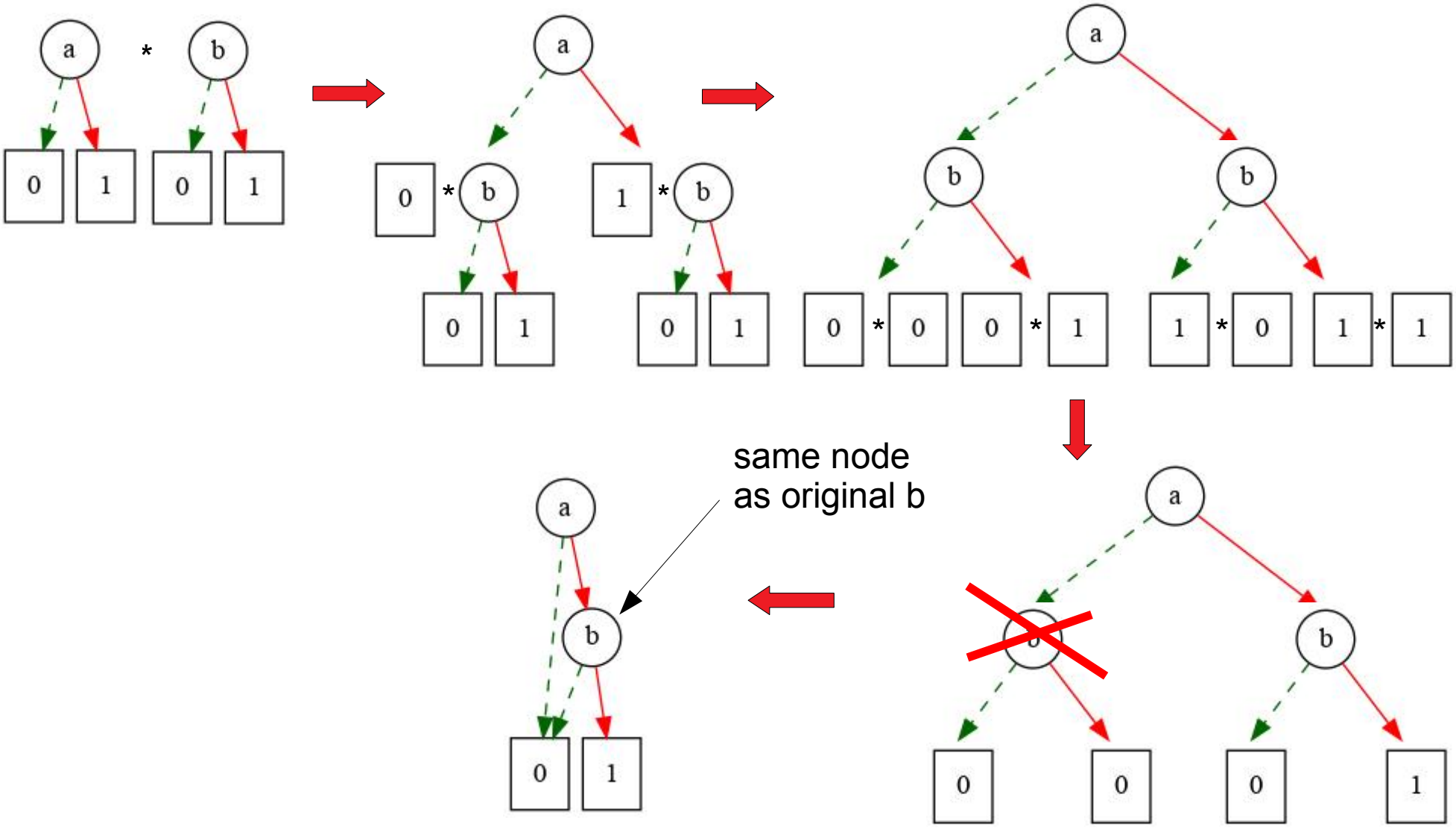


How to build BDDs

- We started with truth table
  - Defeats the purpose of using smaller representation
- In practice
  - Build BDDs for variables
  - Perform Boolean operations on BDDs
    - Use recursive parallel top down search through all paths in both BDDs
  - Check whether a copy of the node already exists before creating, using lookup on variable, 0 child, and 1 child
  - Don't create any nodes with same 0 and 1 children

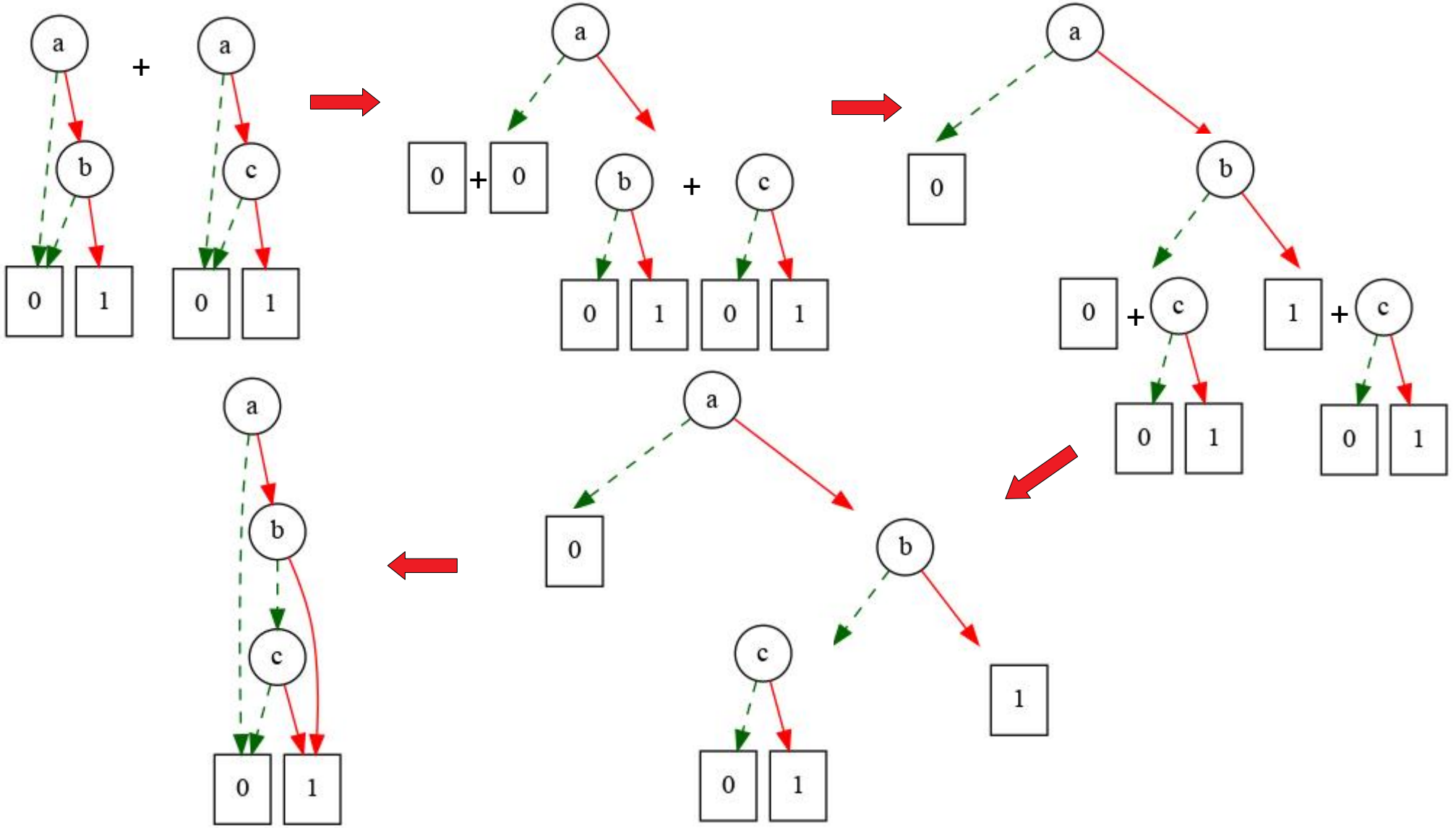
Building BDDs

- Example: ab, variable order a, b



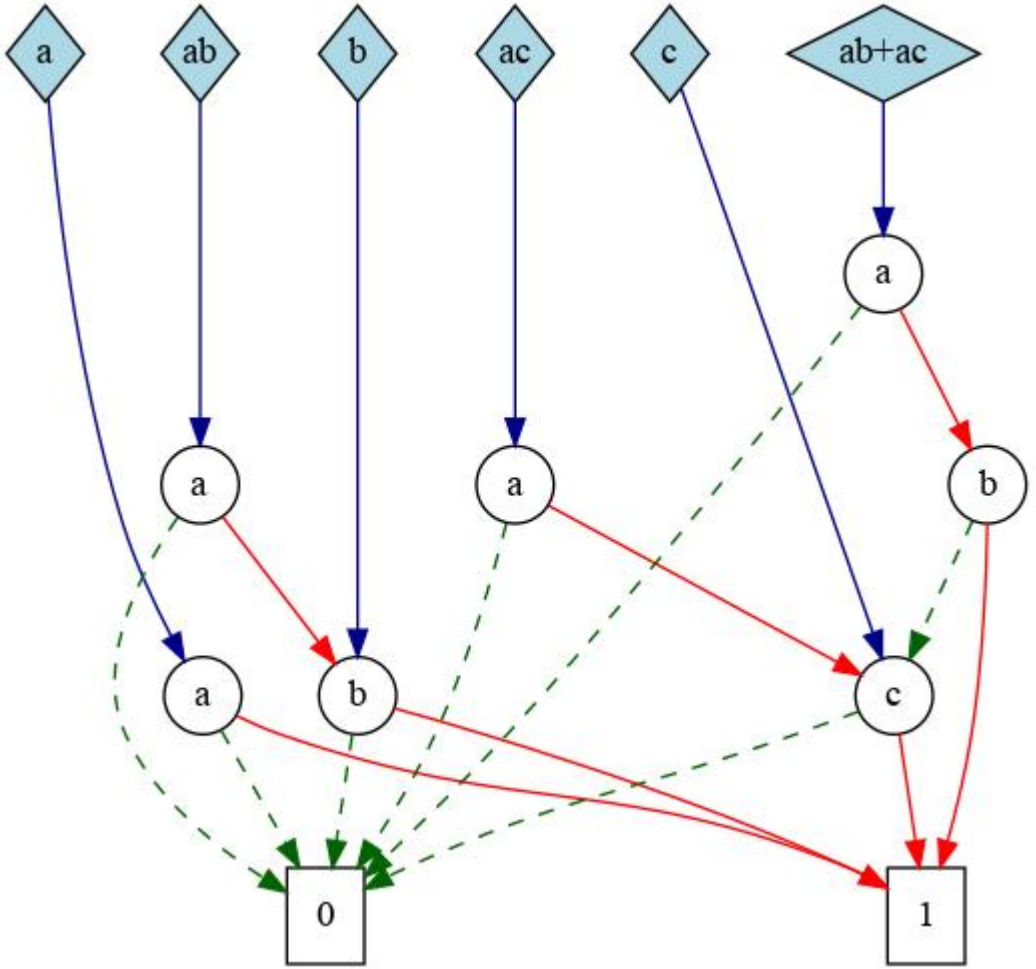
Building BDDs

- Example: ab + ac, variable order a, b, c



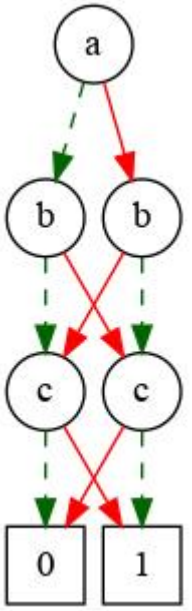
Building BDDs

- All shared BDDs after building ab + ac



How to build BDDs

- Steps above are enough to build small ROBDDs
- But could still take exponential *time*
- Problem: There can be an exponential number of *paths* even for small BDDs
- Example: XOR



2^N paths for N input XOR

- Solution:
  - Keep track of recent operations on pairs of nodes (including internal nodes)
  - Reuse stored result

Some other applications

- Probability calculation
  - Compute probability function is true, given independent inputs with known probabilities
- Sensitivity to an input
  - What values of *other* inputs make the function sensitive to some input?
    - When does switching that input change the output?
  - Used in testing digital chips
- Concensus / Universal quantifier (∀x)
  - What values of *other* inputs make the function always true, regardless of the value of some input x?
- Smoothing / Existential quantifier (∃x)
  - For what values of *other* inputs can the function be true for at least one value of some input x?