Binary Decision Diagrams

15-414 Bug Catching: Automated Program Verification and Testing

based on slides by Sagar Chaki

BDDs in a nutshell

Typically mean Reduced Ordered Binary Decision Diagrams (ROBDDs)

Canonical representation of Boolean formulas

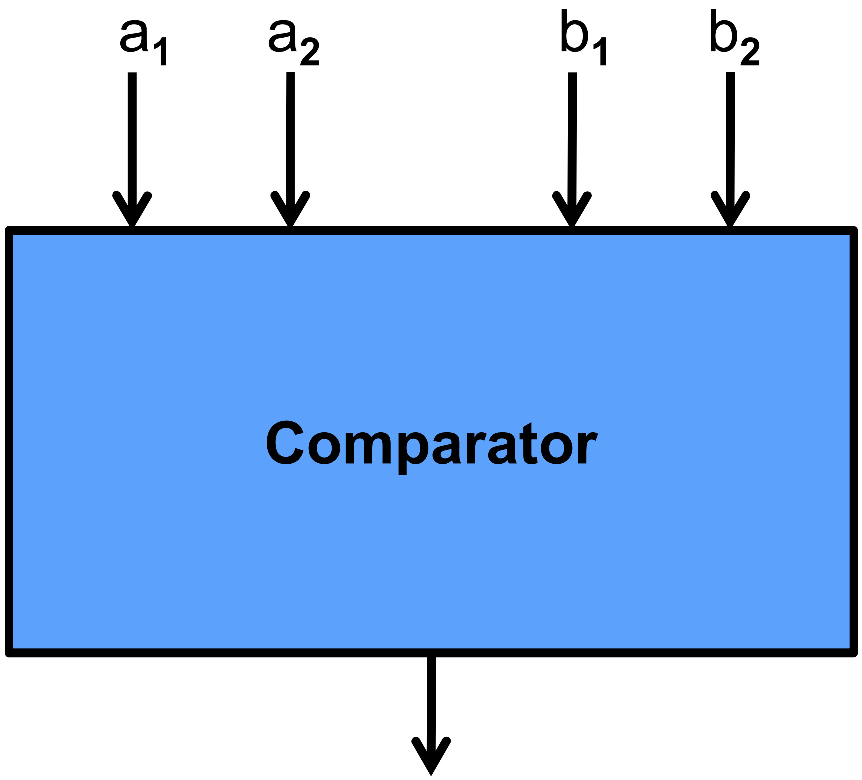
Often substantially more compact than a traditional normal form

Can be manipulated very efficiently

- Conjunction, Disjunction, Negation, Existential Quantification

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers, C-35(8), 1986.*

Running Example: Comparator



f = 1 , $a_1 = b_1$ **Æ** $a_2 = b_2$

Conjunctive Normal Form

f

a_1 = b_1 Æ a_2 = b_2

a_1) b_1 Æ b_1) a_1 Æ a_2) b_2 Æ b_2) a_2

(: a_1 Ç b_1) Æ (: b_1 Ç a_1) Æ (: a_2 Ç b_2) Æ (: b_2 Ç a_2)

(: b_1 Ç a_1) Æ (: a_1 Ç b_1) Æ (: a_2 Ç b_2) Æ (: b_2 Ç a_2)

Not Canonical

Truth Table (1)

a_1	b_1	a_2	b_2	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Still Not Canonical

Truth Table (2)

a_1	a_2	b_1	b_2	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Canonical if you fix variable order.

But always exponential in # of variables. Let's try to fix this.

Shannon's / Boole's Expansion

Every Boolean formula $f(a_0, a_1, \ldots, a_n)$ can be written as

$(a_0$ Æ $f(true, a_1, \ldots, a_n))$ Ç $(:a_0$ Æ $f(false, a_1, \ldots, a_n))$

or, simply,

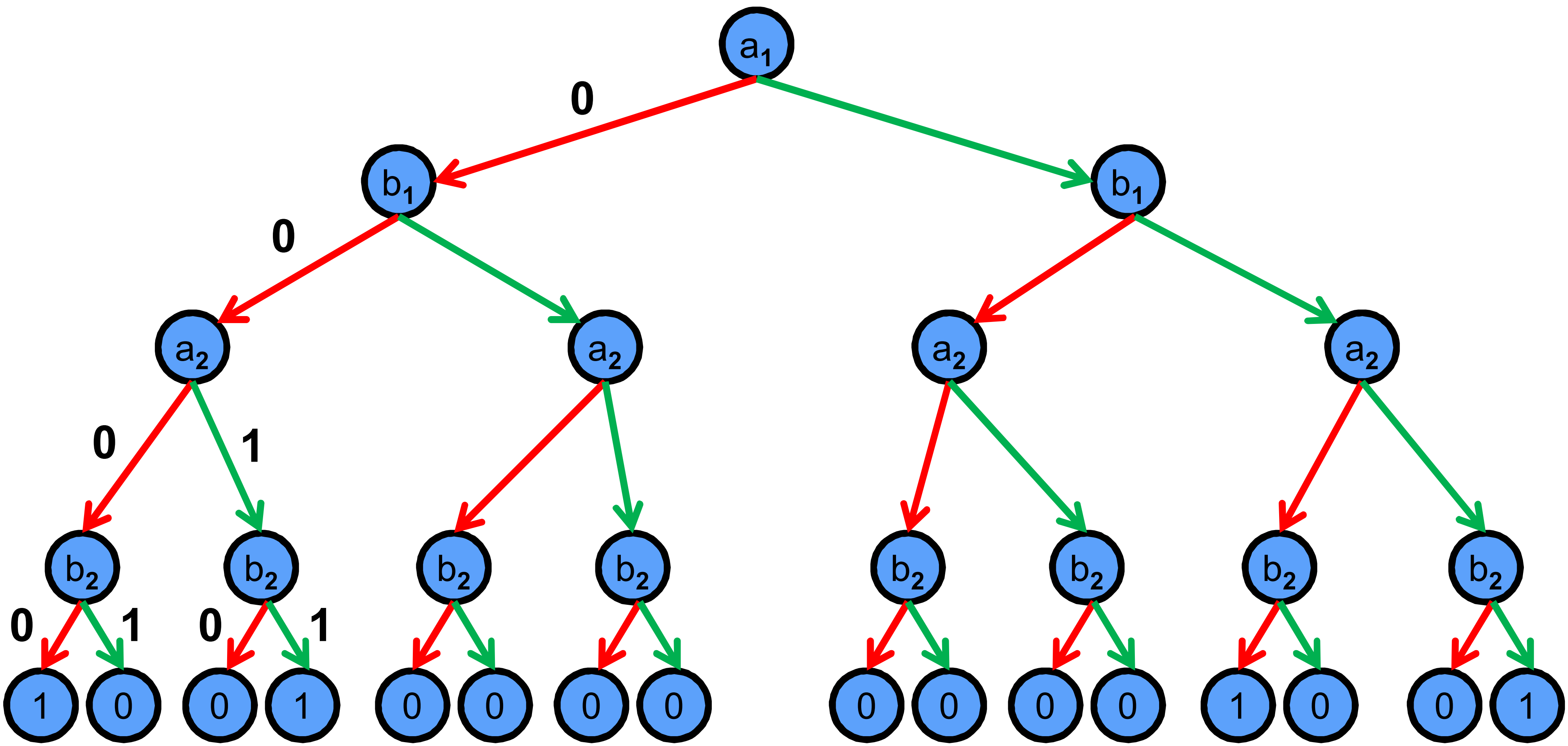
$ITE\ (a_0, f(true, a_1, \ldots, a_n), f(false, a_1, \ldots, a_n))$

where ITE stands for If-Then-Else

The formula $f(true, a_1, \ldots, a_n)$ is called the *cofactor* of f w.r.t. a_0

The formula $f(false, a_1, \ldots, a_n)$ is called the *cofactor* of f w.r.t. $:a_0$

Representing a Truth Table using a Graph



Binary Decision Tree (in this case ordered)

Binary Decision Tree: Formal Definition

Balanced binary tree. Length of each path = # of variables

Leaf nodes labeled with either 0 or 1

Internal node v labeled with a Boolean variable var(v)

- Every node on a path labeled with a different variable

Internal node v has two children: low(v) and high(v)

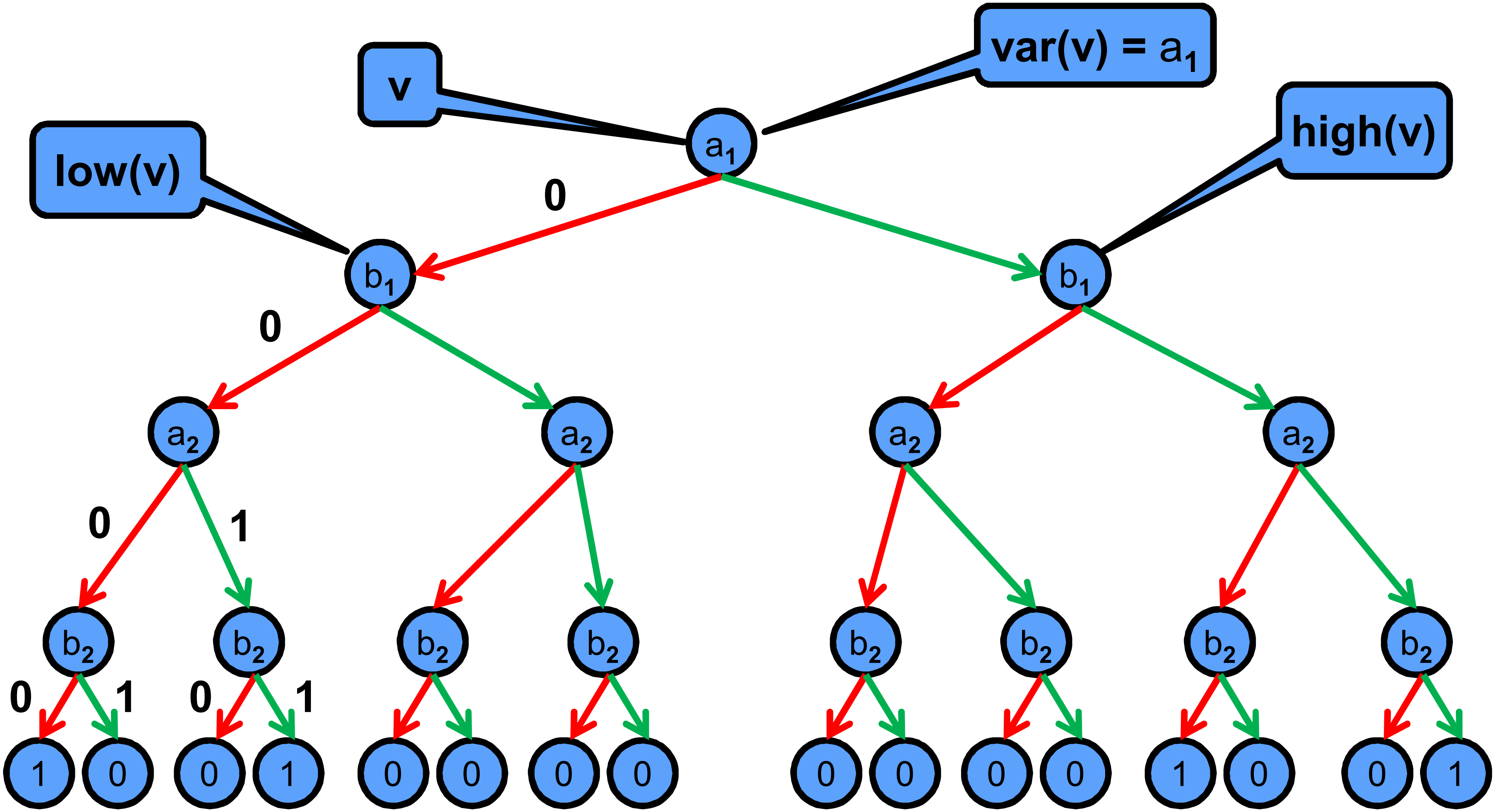
Each path corresponds to a (partial) truth assignment to variables

- Assign 0 to var(v) if low(v) is in the path, and 1 if high(v) is in the path

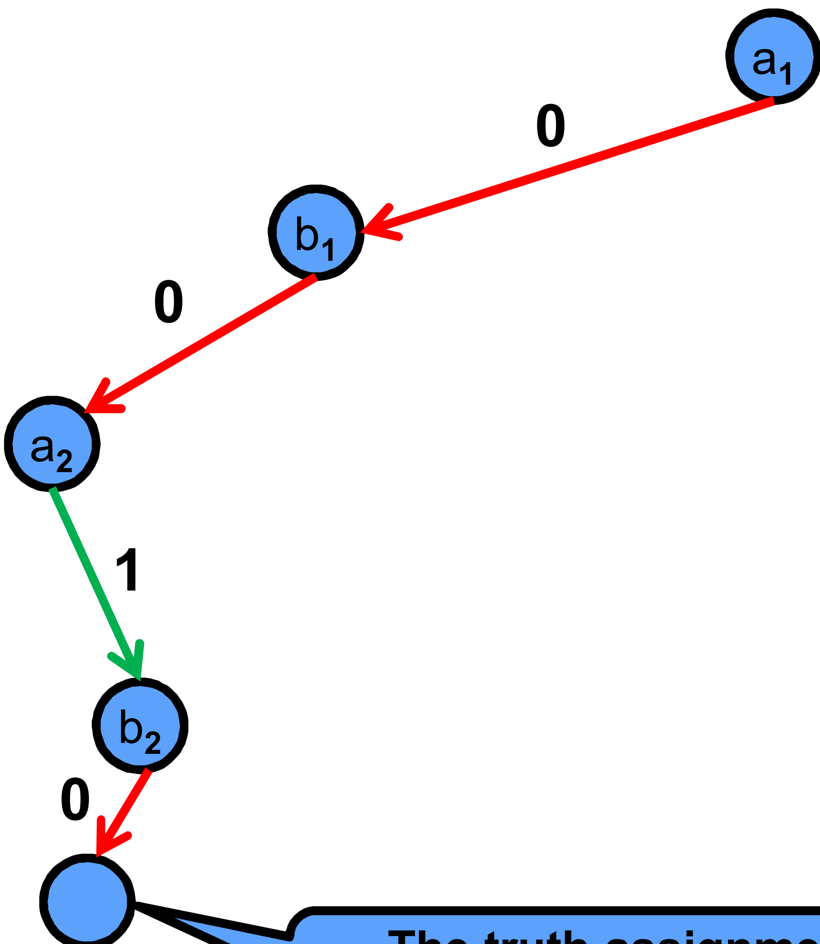
Value of a leaf is determined by:

- Constructing the truth assignment for the path leading to it from the root
- Looking up the truth table with this truth assignment

Binary Decision Tree



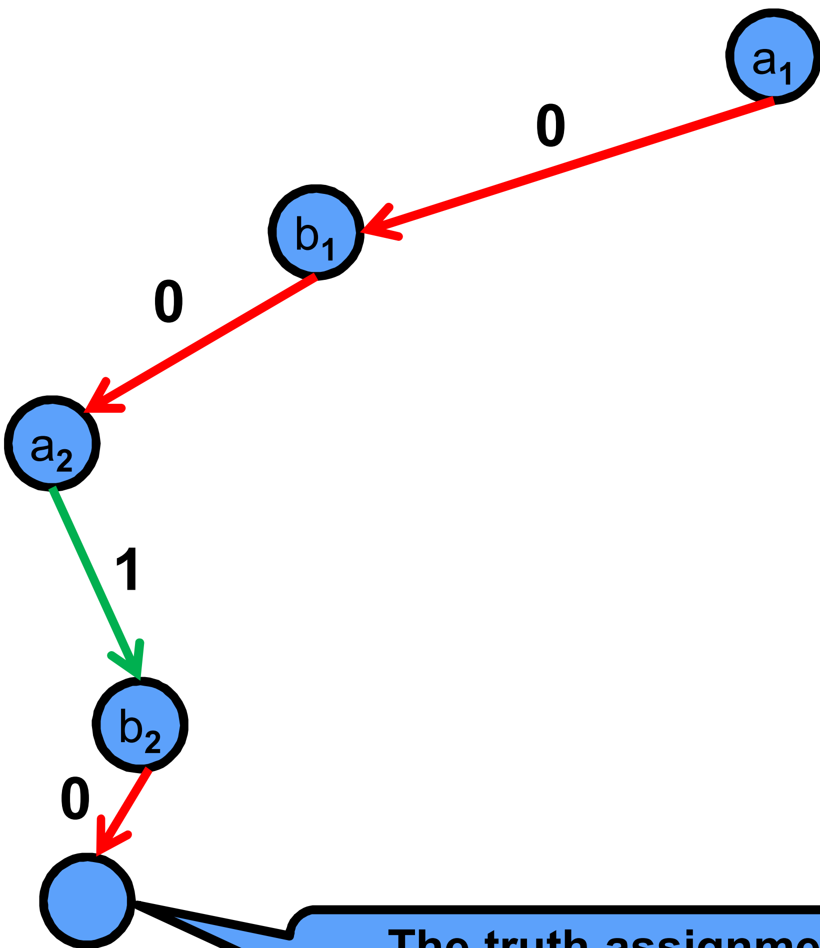
Binary Decision Tree



The truth assignment corresponding to the path to this leaf is:

a_1 = ? b_1 = ? a_2 = ? b_2 = ?

Binary Decision Tree

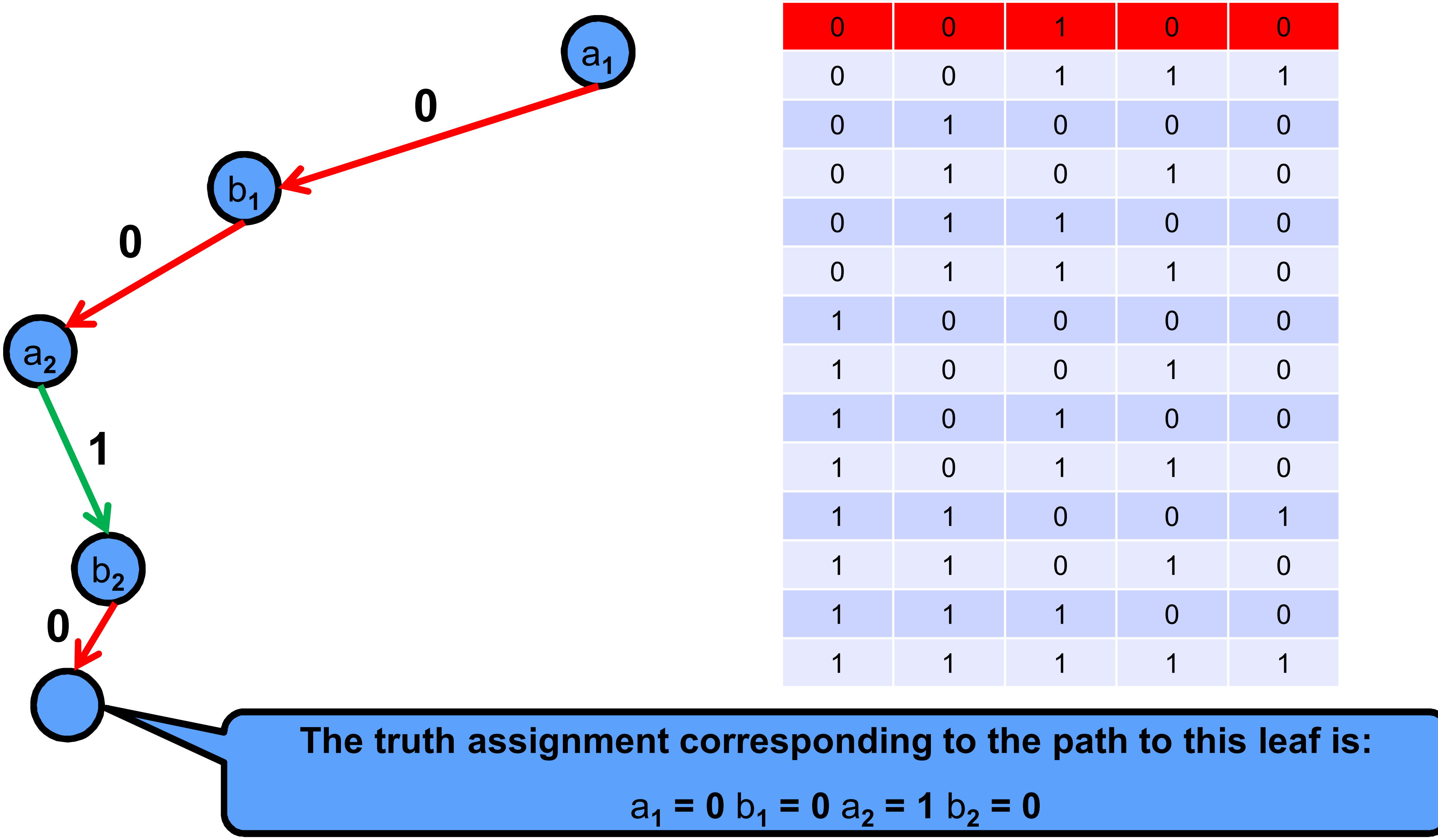


a_1	b_1	a_2	b_2	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

The truth assignment corresponding to the path to this leaf is:

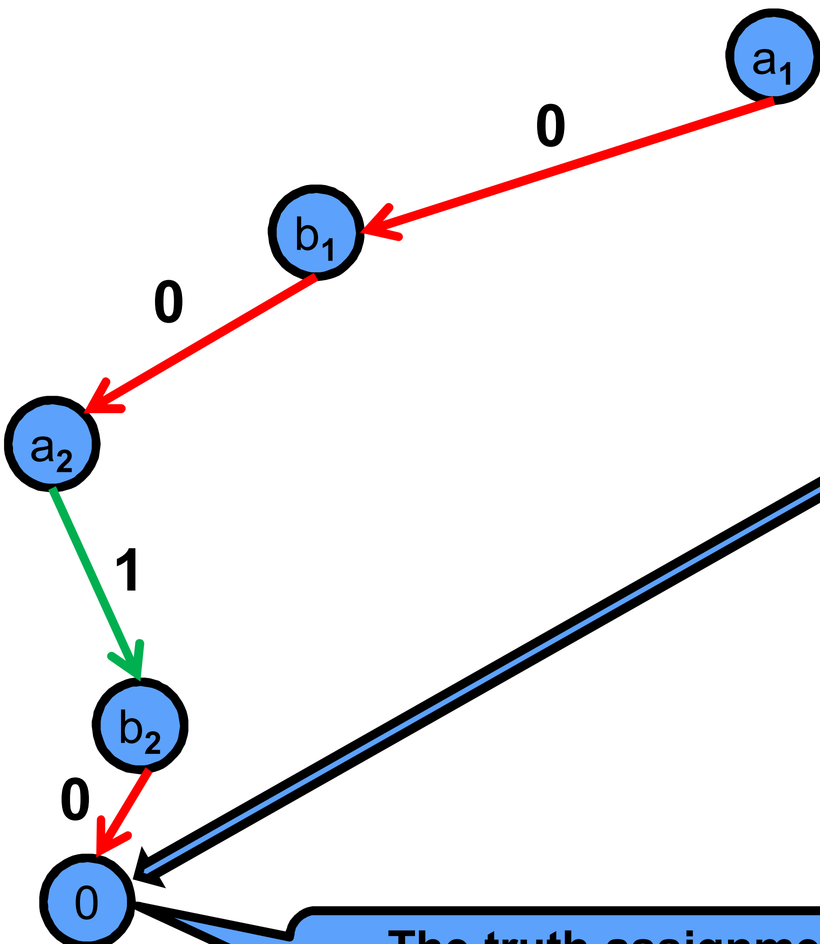
$a_1 = 0$ $b_1 = 0$ $a_2 = 1$ $b_2 = 0$

Binary Decision Tree



a_1	b_1	a_2	b_2	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Binary Decision Tree

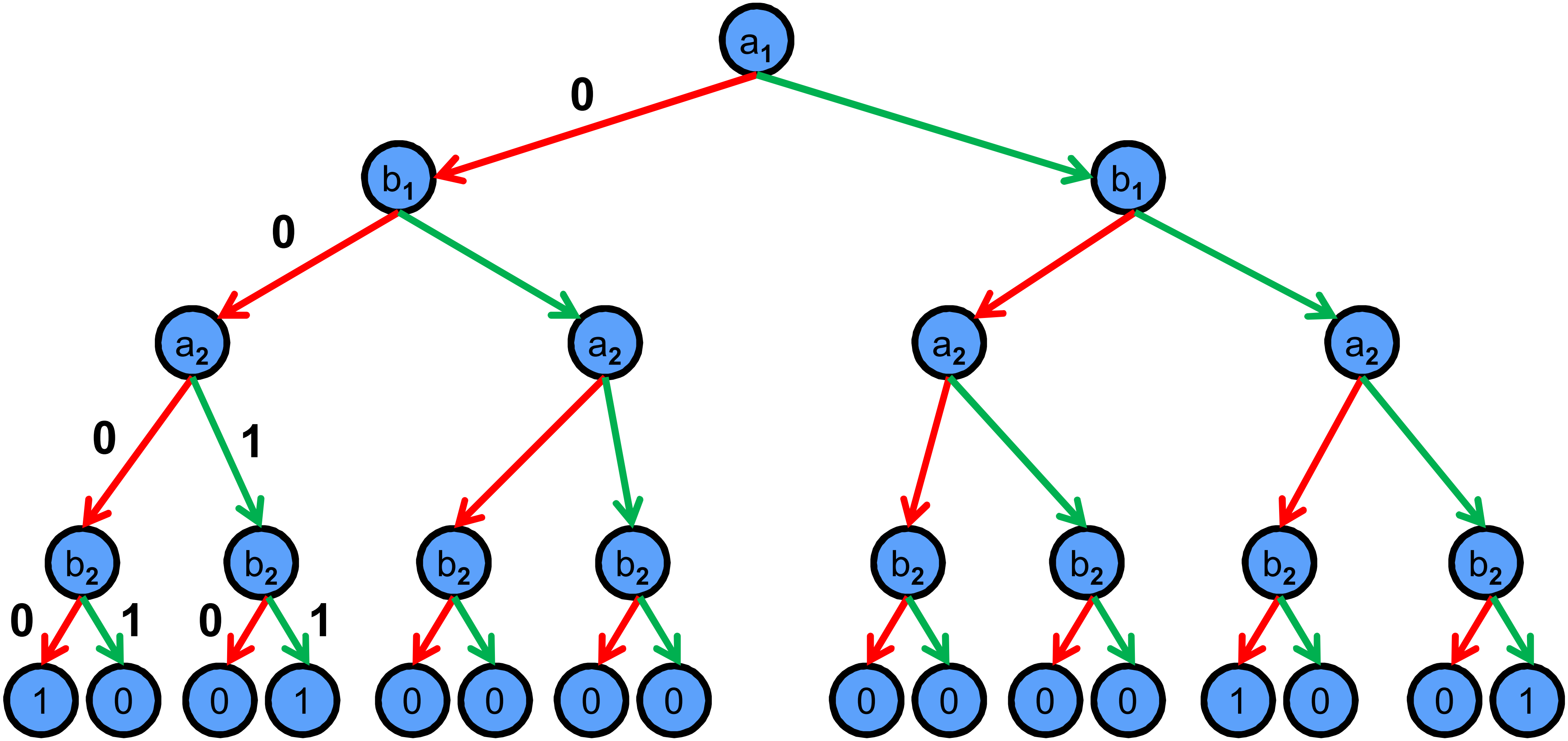


a_1	b_1	a_2	b_2	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

The truth assignment corresponding to the path to this leaf is:

$a_1 = 0$ $b_1 = 0$ $a_2 = 1$ $b_2 = 0$

Binary Decision Tree (BDT)



Canonical if you fix variable order (i.e., use ordered BDT)

But still exponential in # of variables. Let's try to fix this.

Reduced Ordered BDD

Conceptually, a ROBDD is obtained from an ordered BDT (OBDT) by eliminating redundant sub-diagrams and nodes

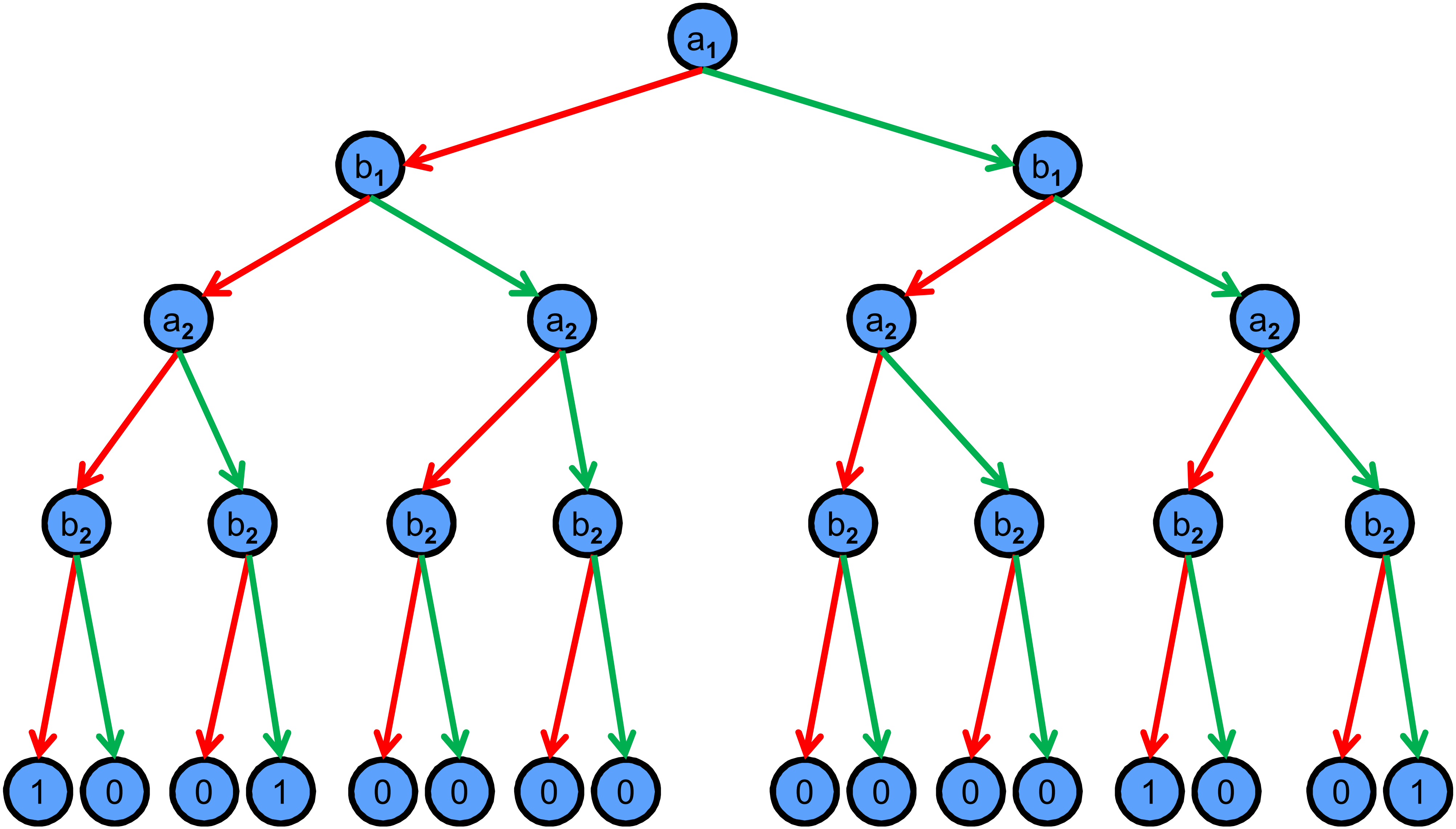
Start with OBDT and repeatedly apply the following two operations as long as possible:

1. Eliminate duplicate sub-diagrams. Keep a single copy. Redirect edges into the eliminated duplicates into this single copy.
2. Eliminate redundant nodes. Whenever low(v) = high(v), remove v and redirect edges into v to low(v).

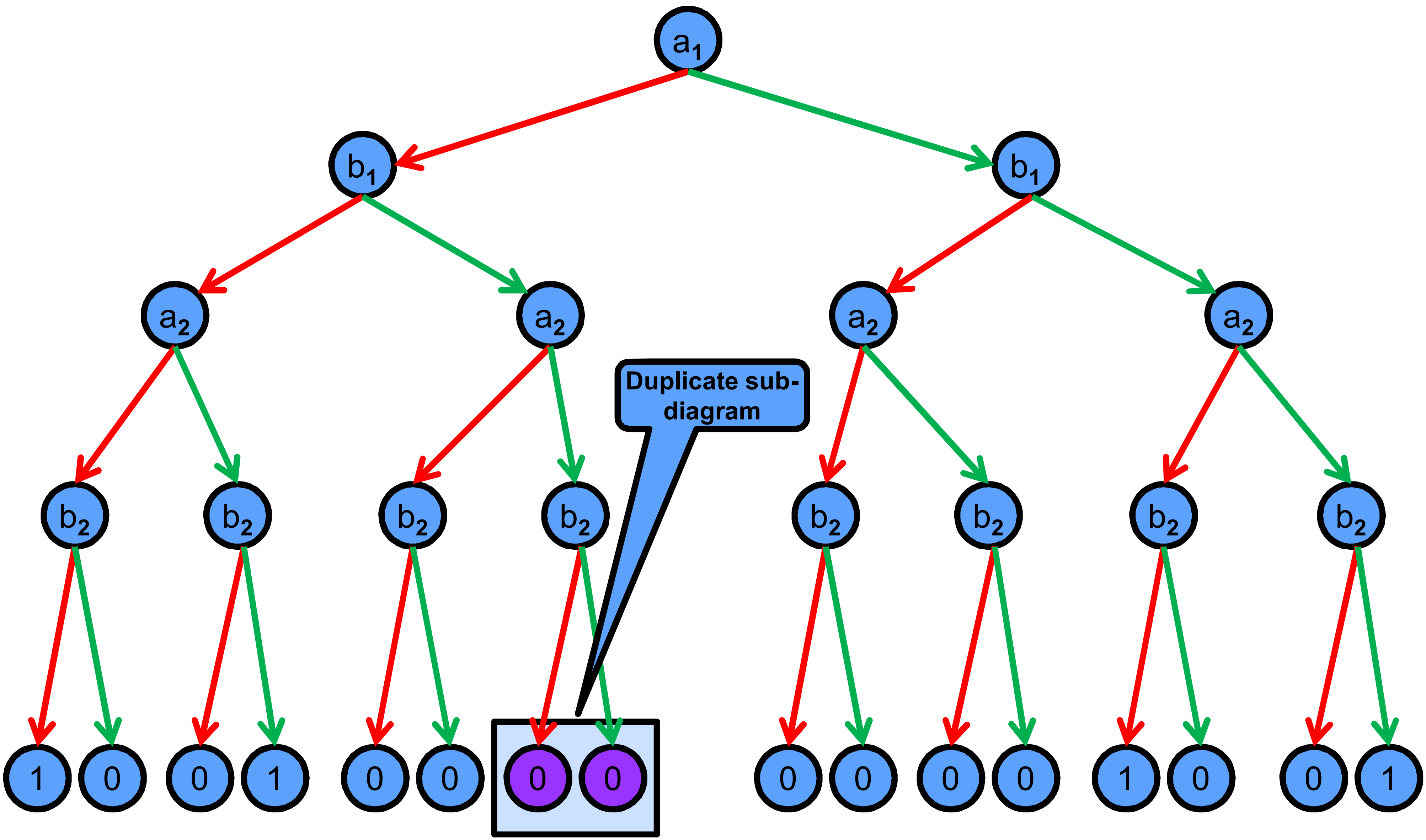
- Why does this terminate?

ROBDD is often exponentially smaller than the corresponding OBDT

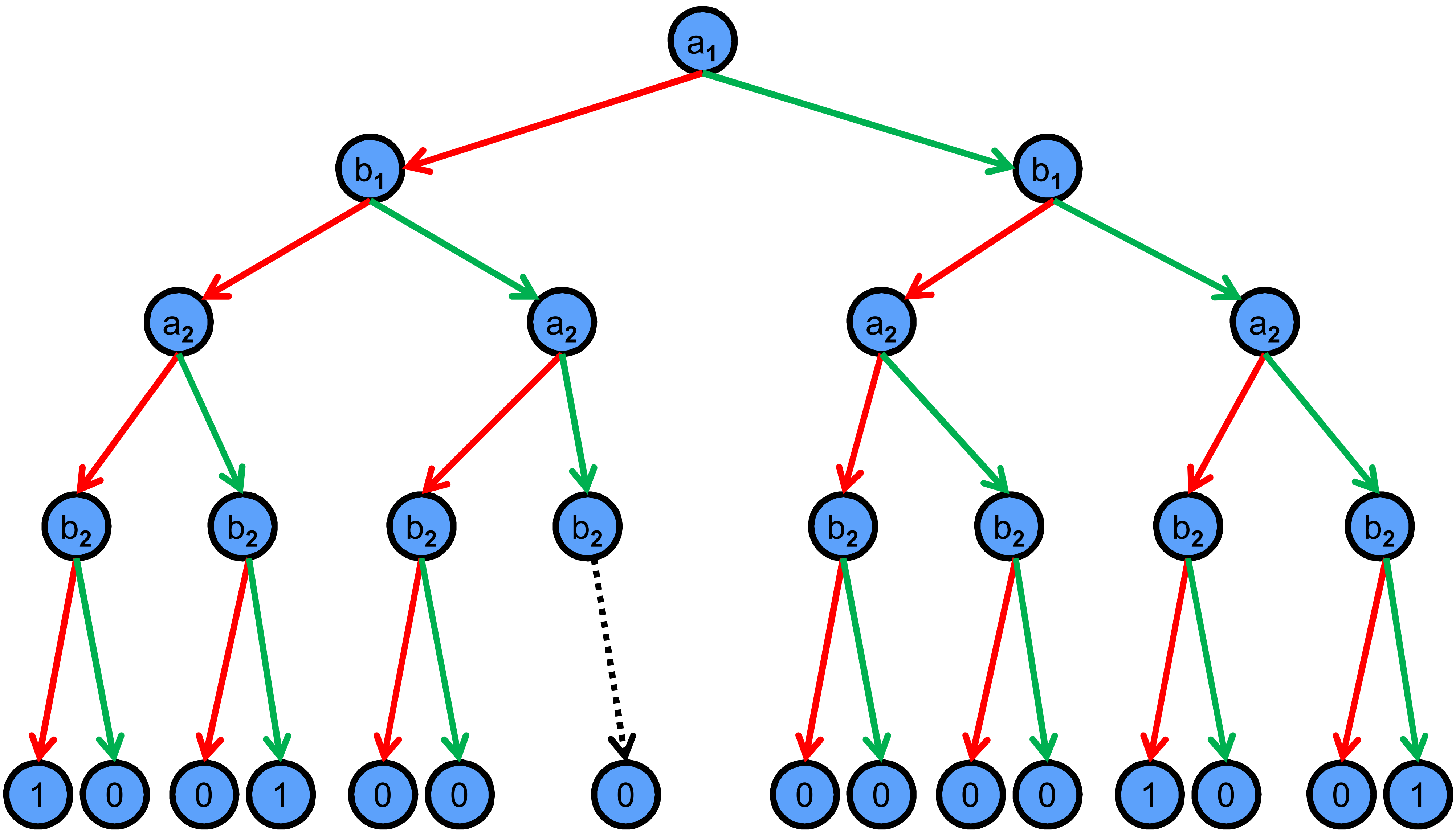
OBDT to ROBDD



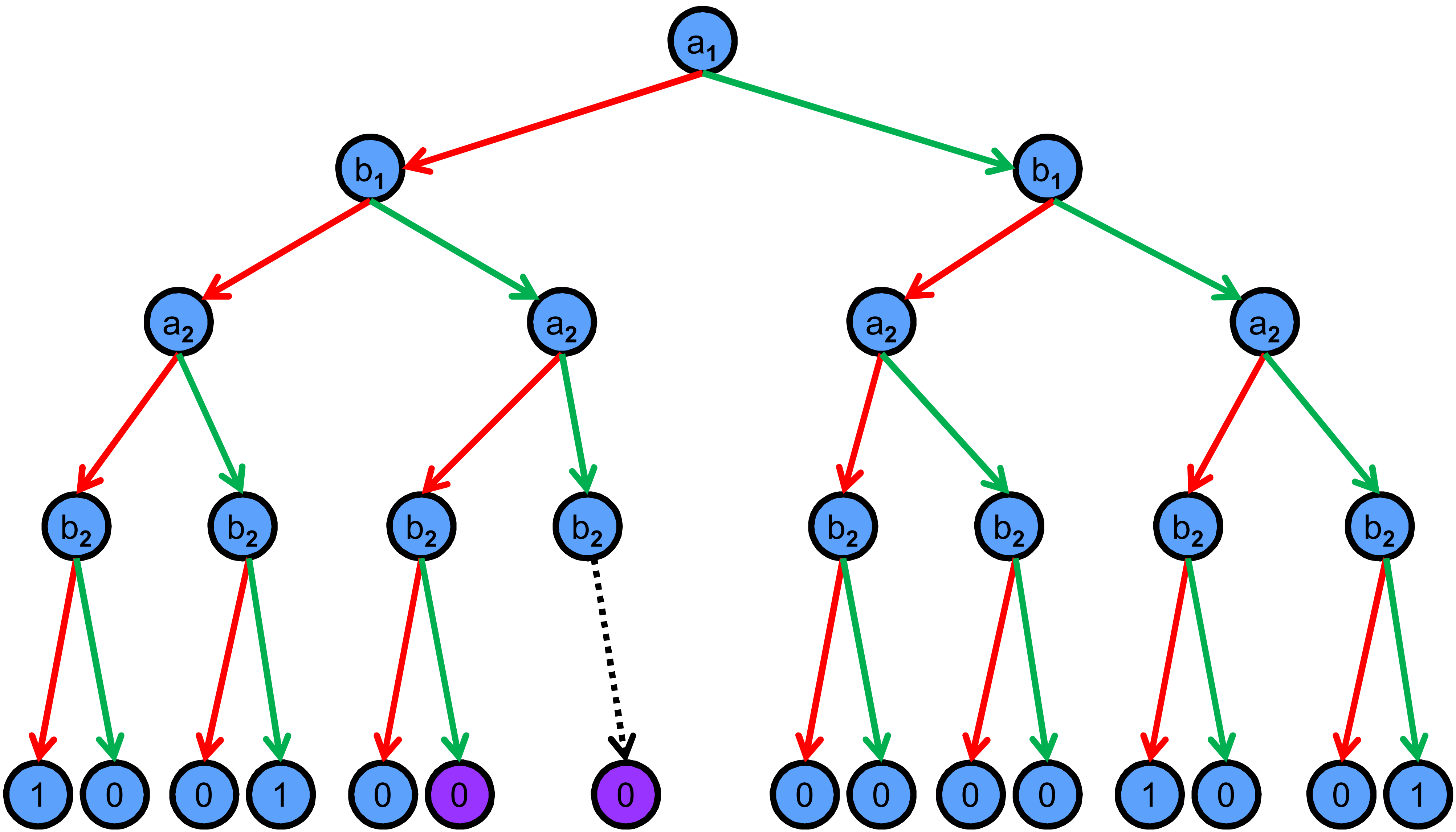
OBDT to ROBDD



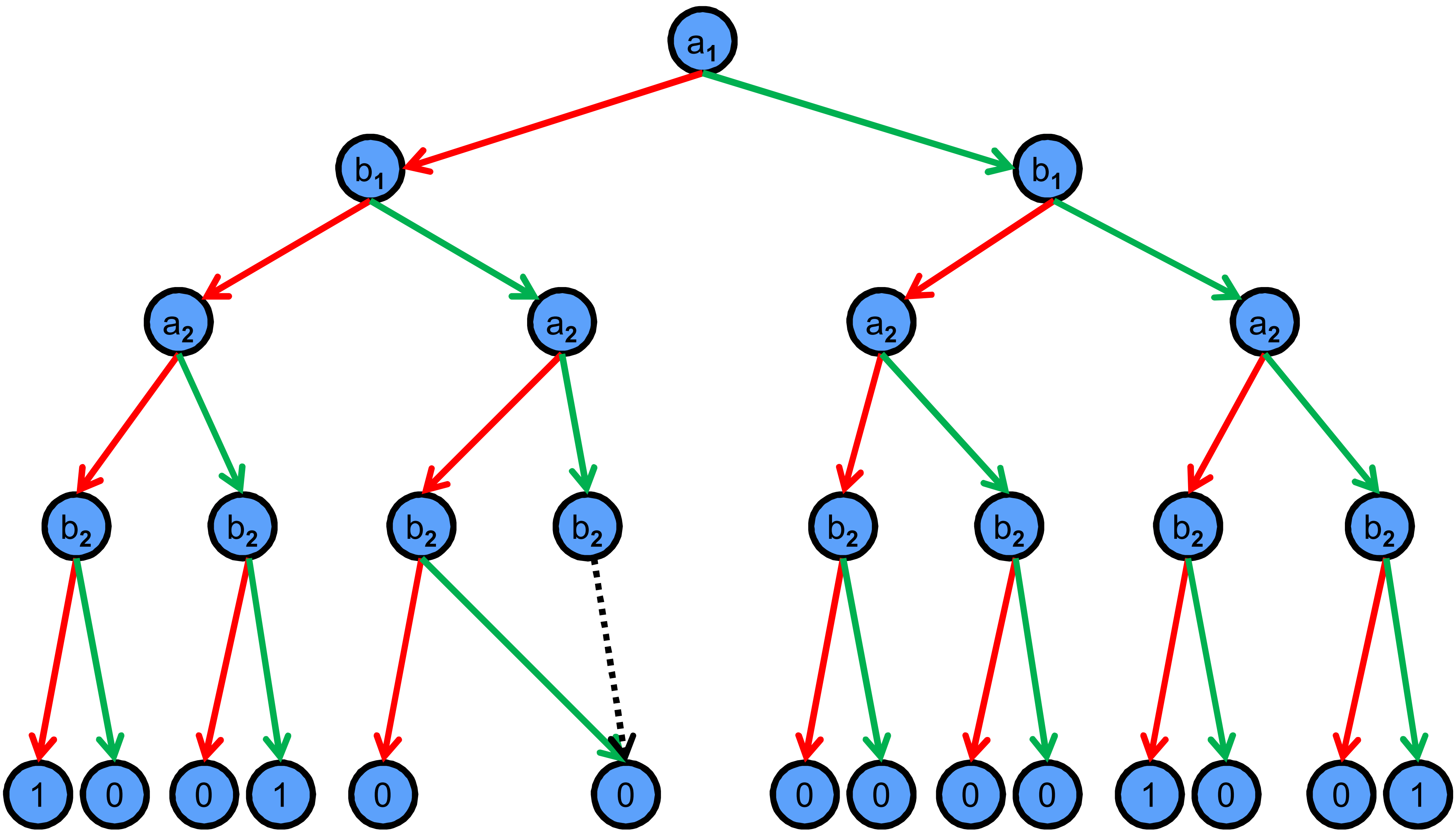
OBDT to ROBDD



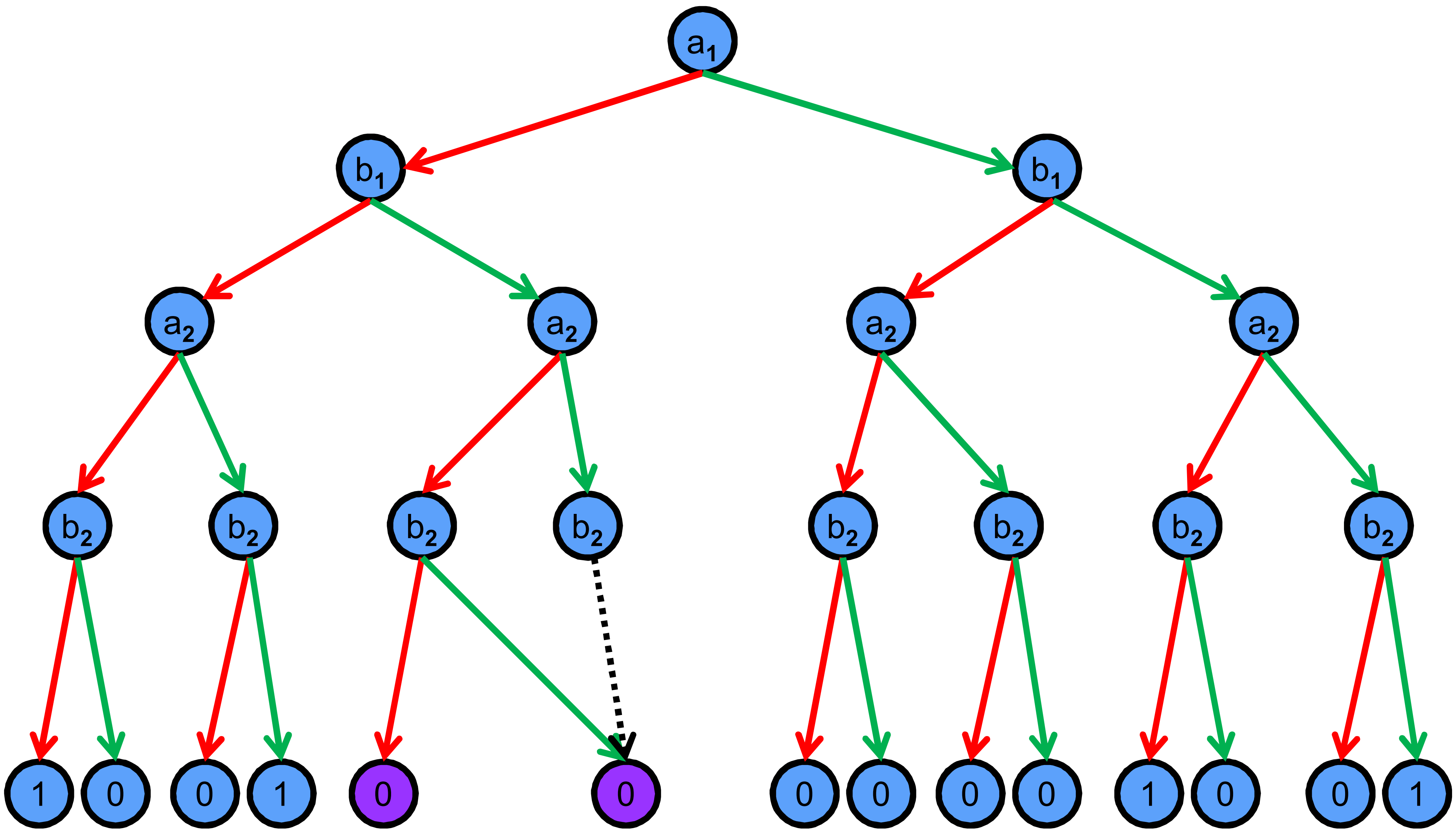
OBDT to ROBDD



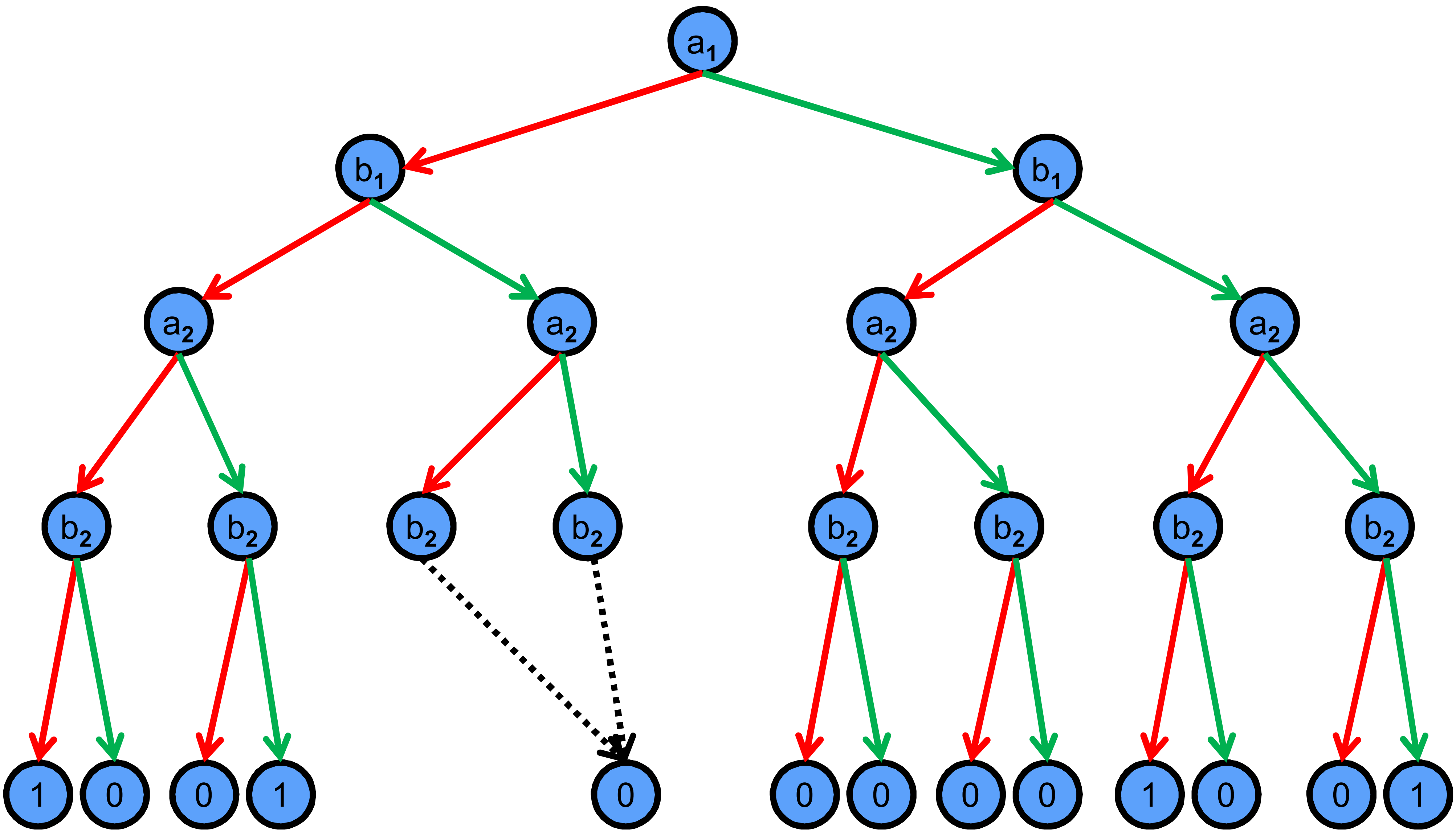
OBDT to ROBDD



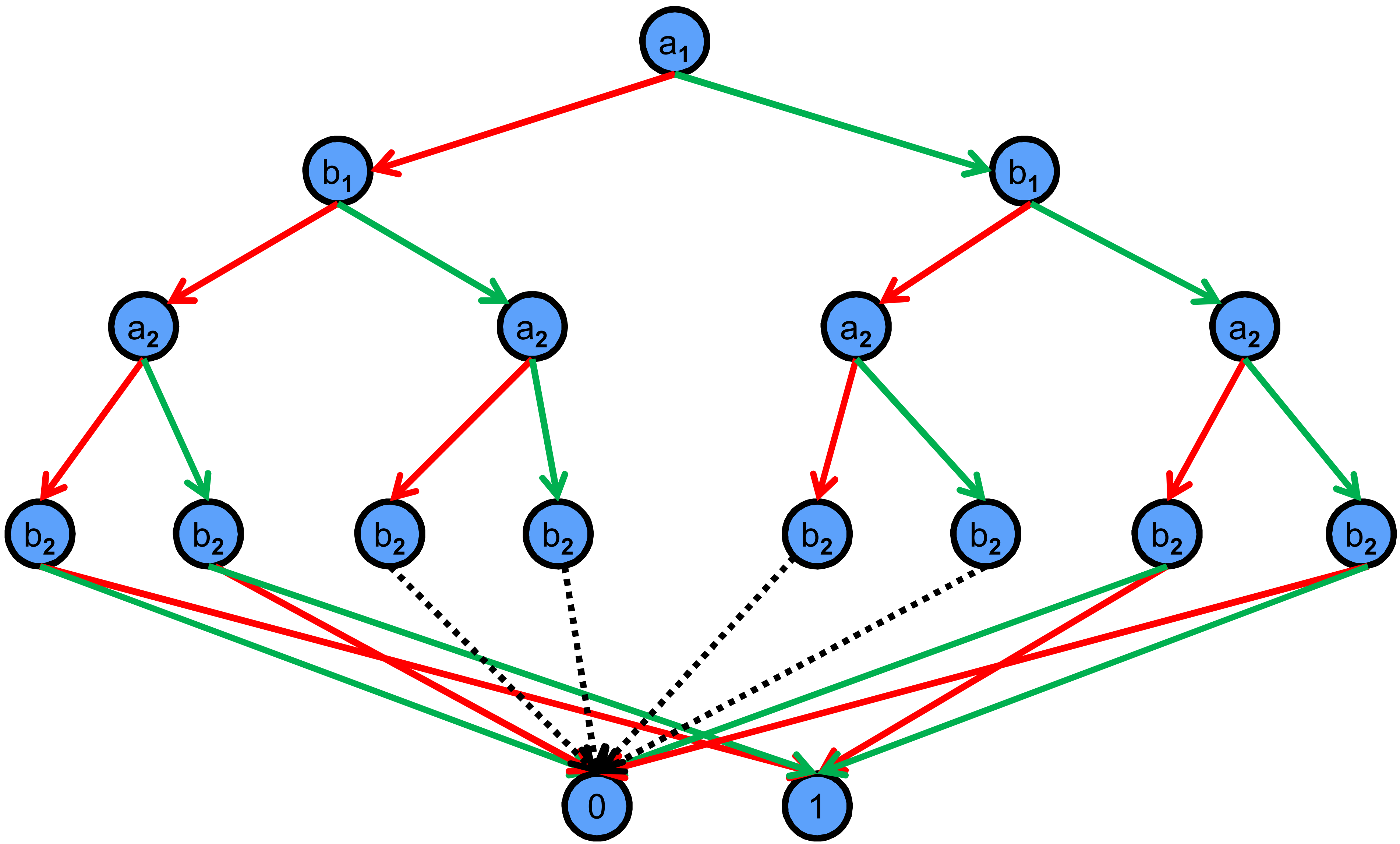
OBDT to ROBDD



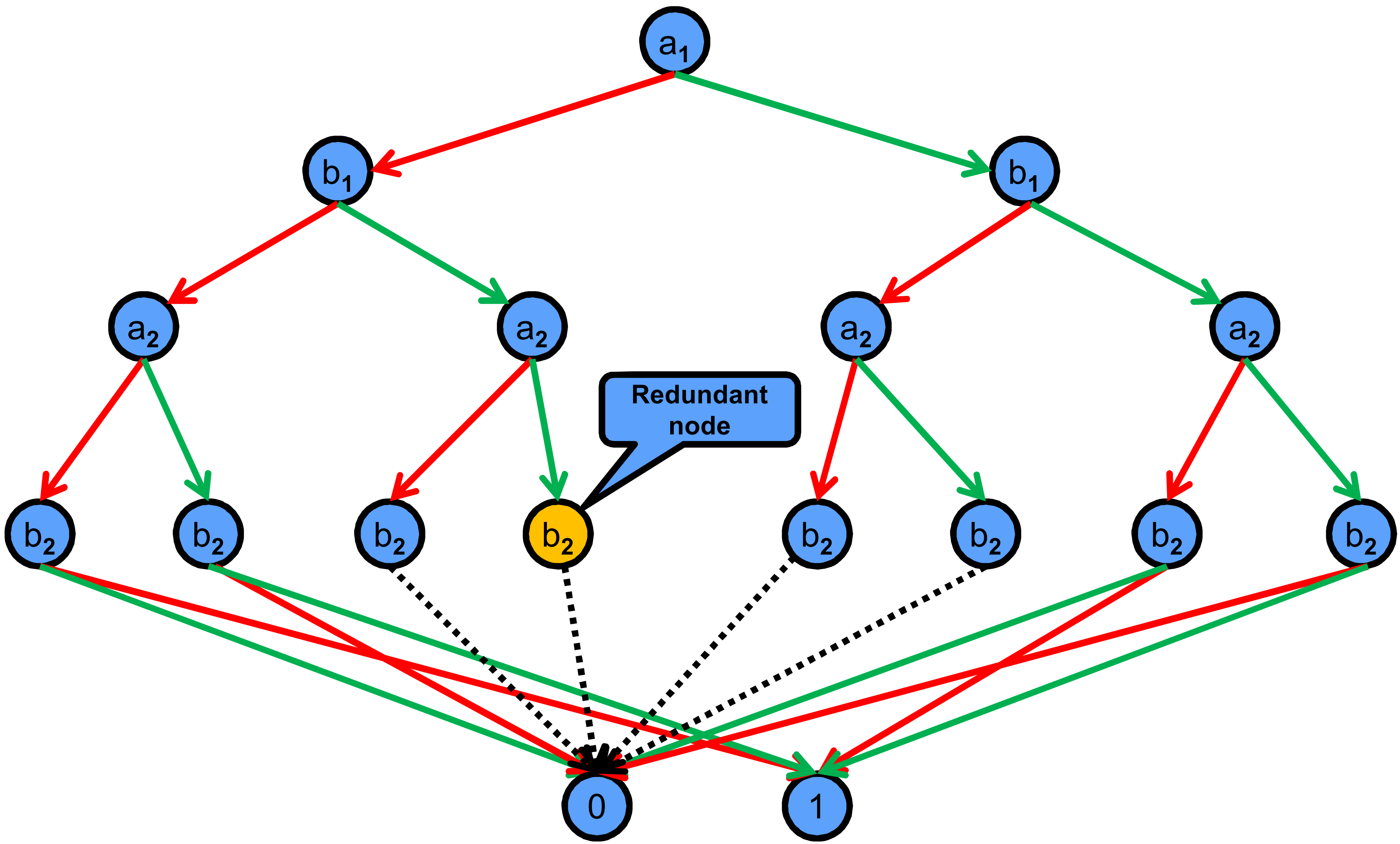
OBDT to ROBDD



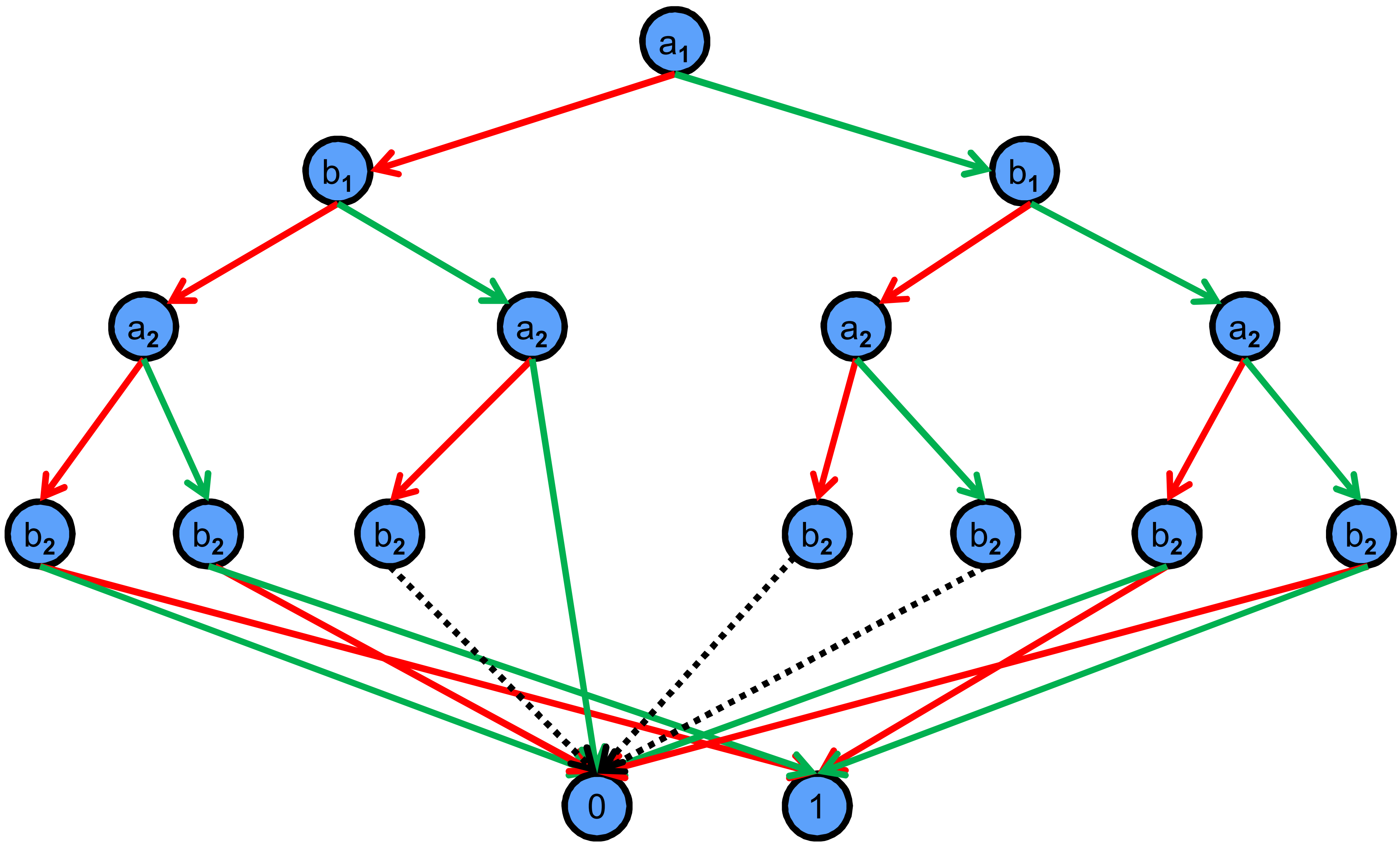
OBDT to ROBDD



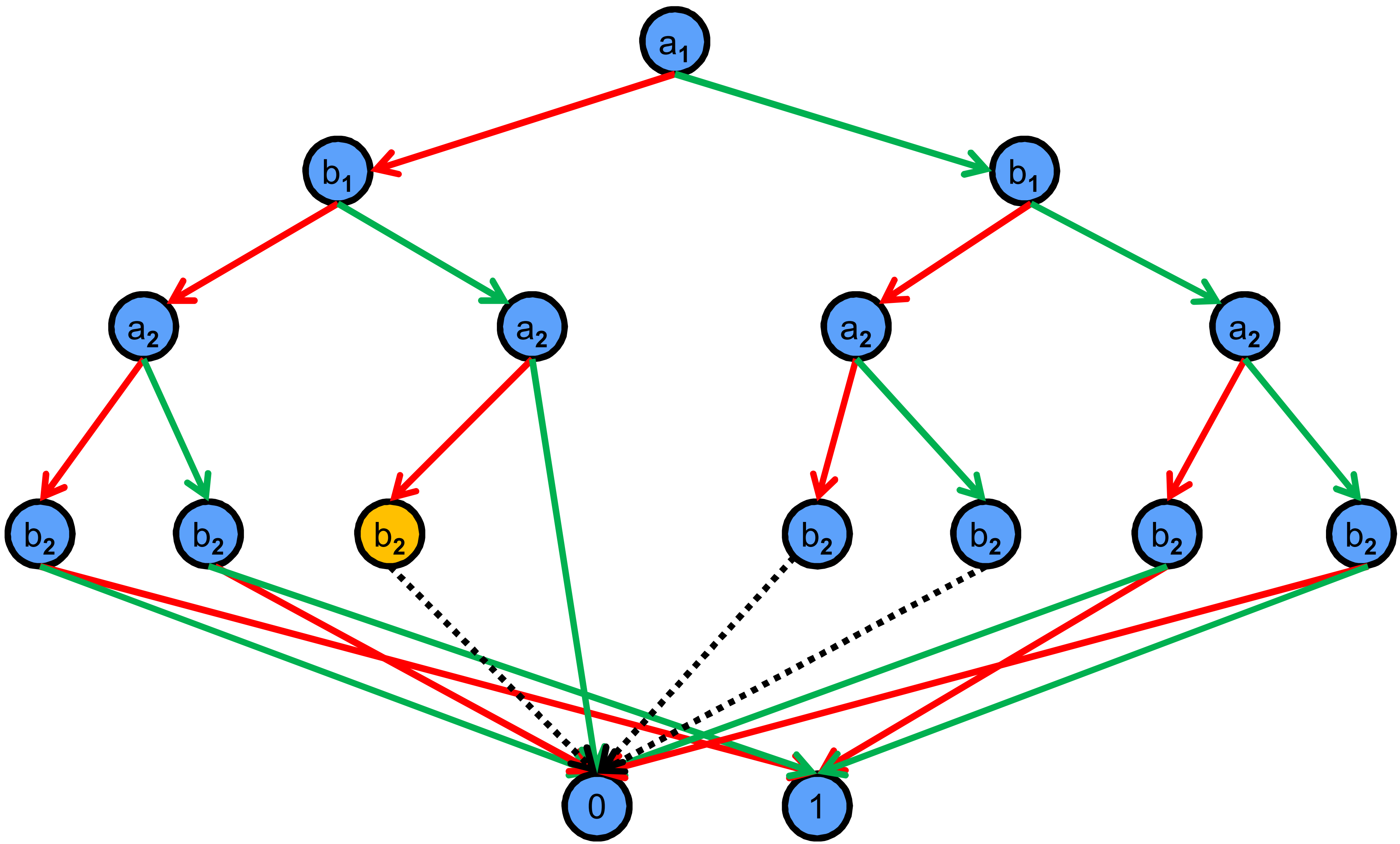
OBDT to ROBDD



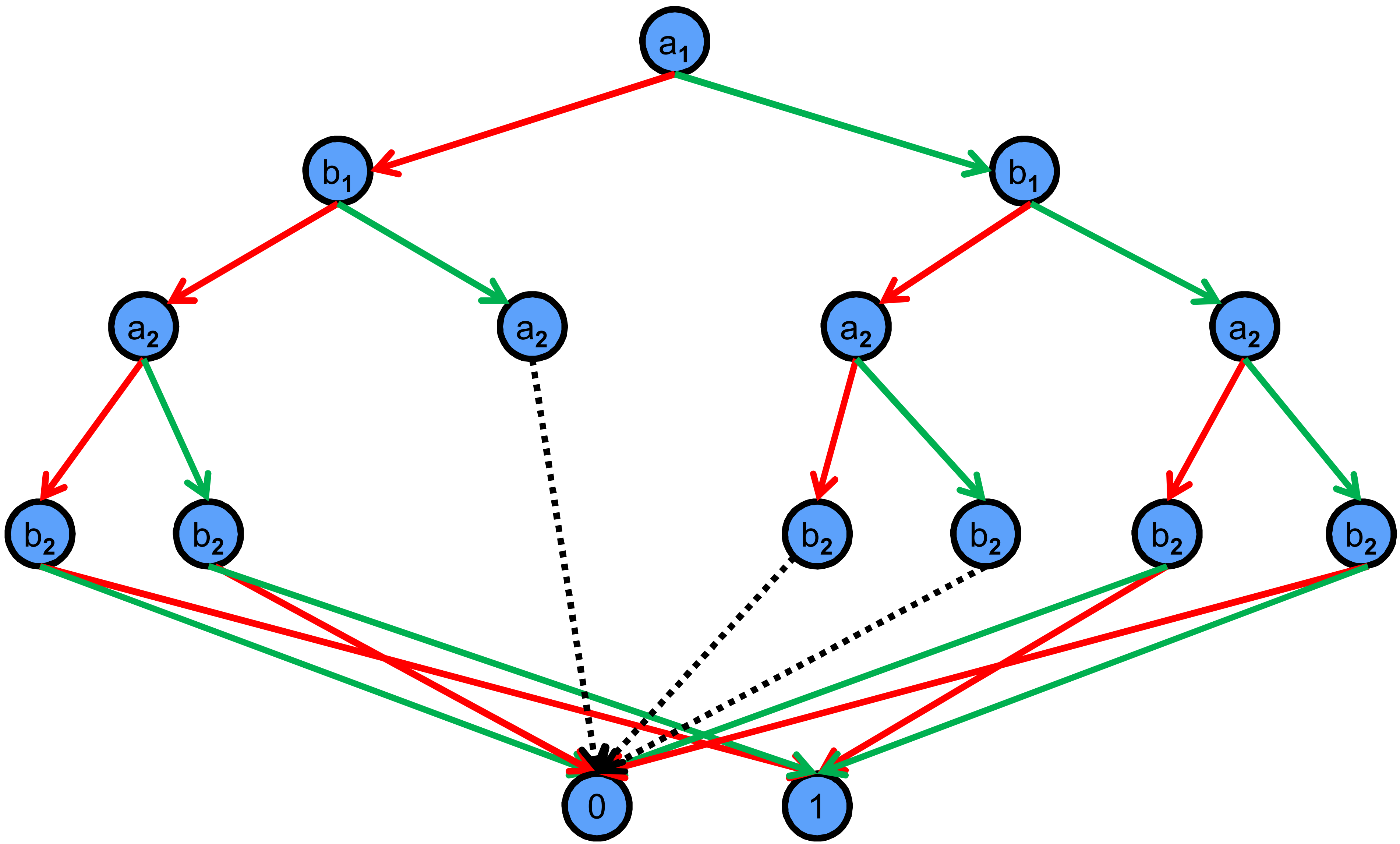
OBDT to ROBDD



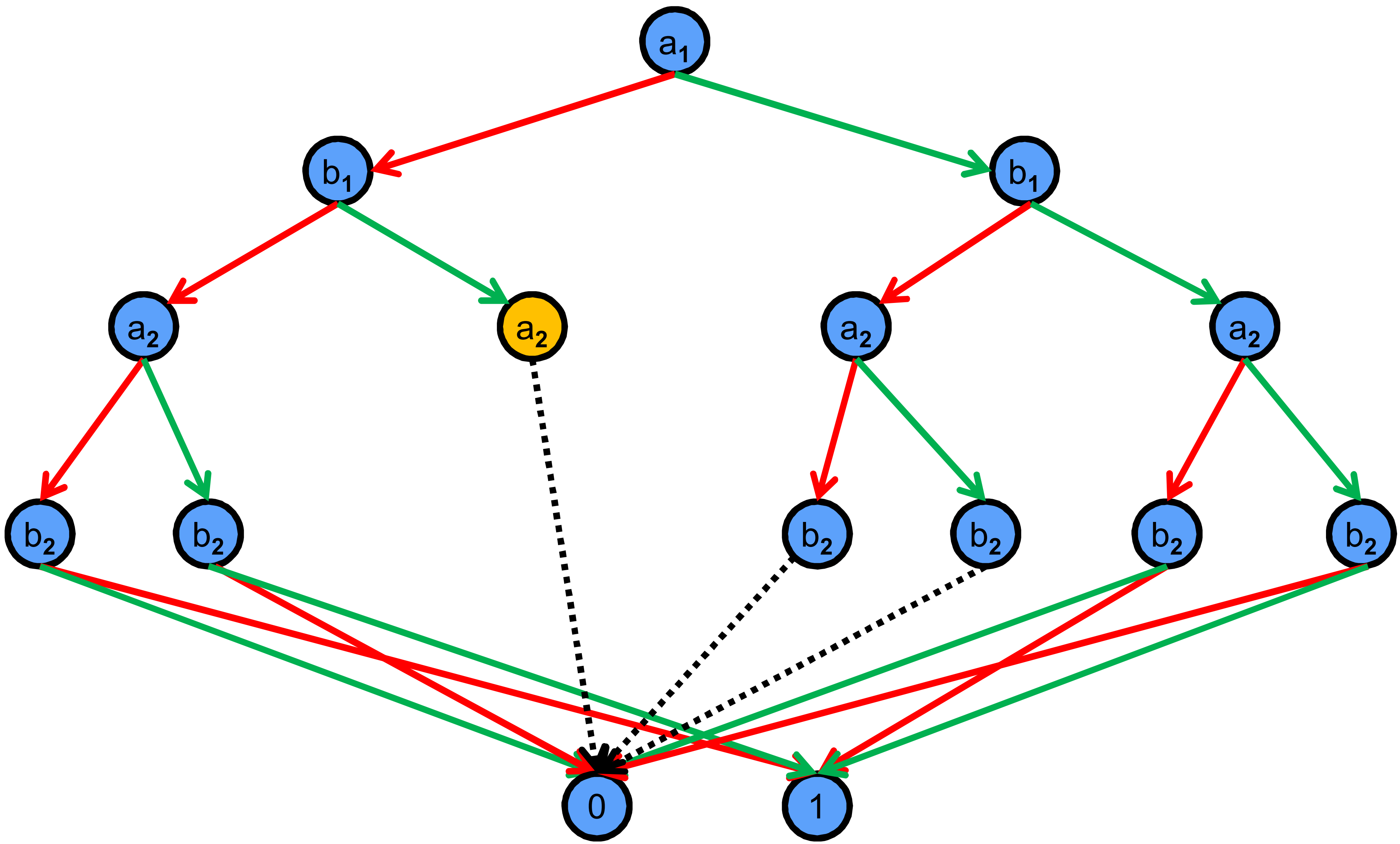
OBDT to ROBDD



OBDT to ROBDD

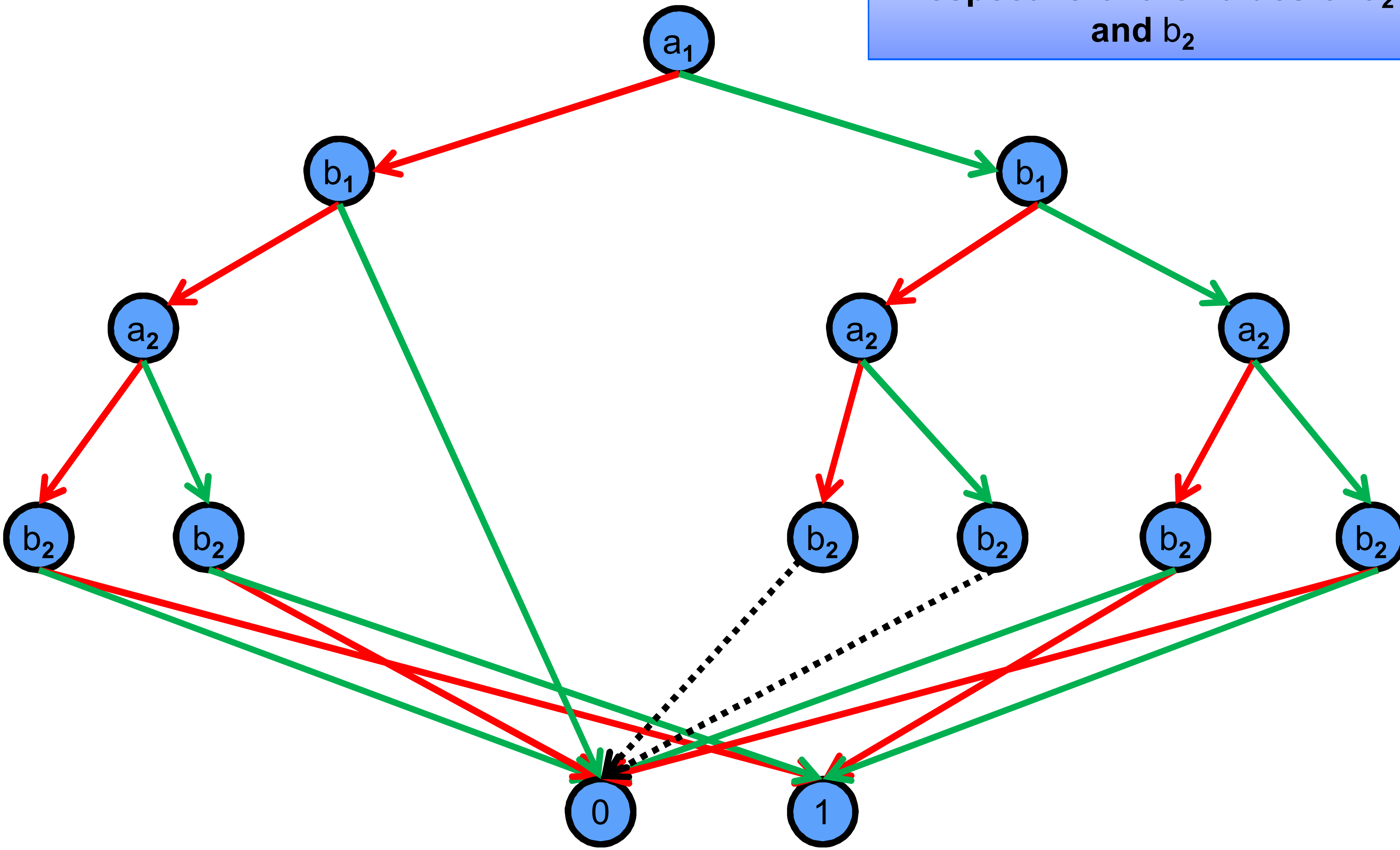


OBDT to ROBDD

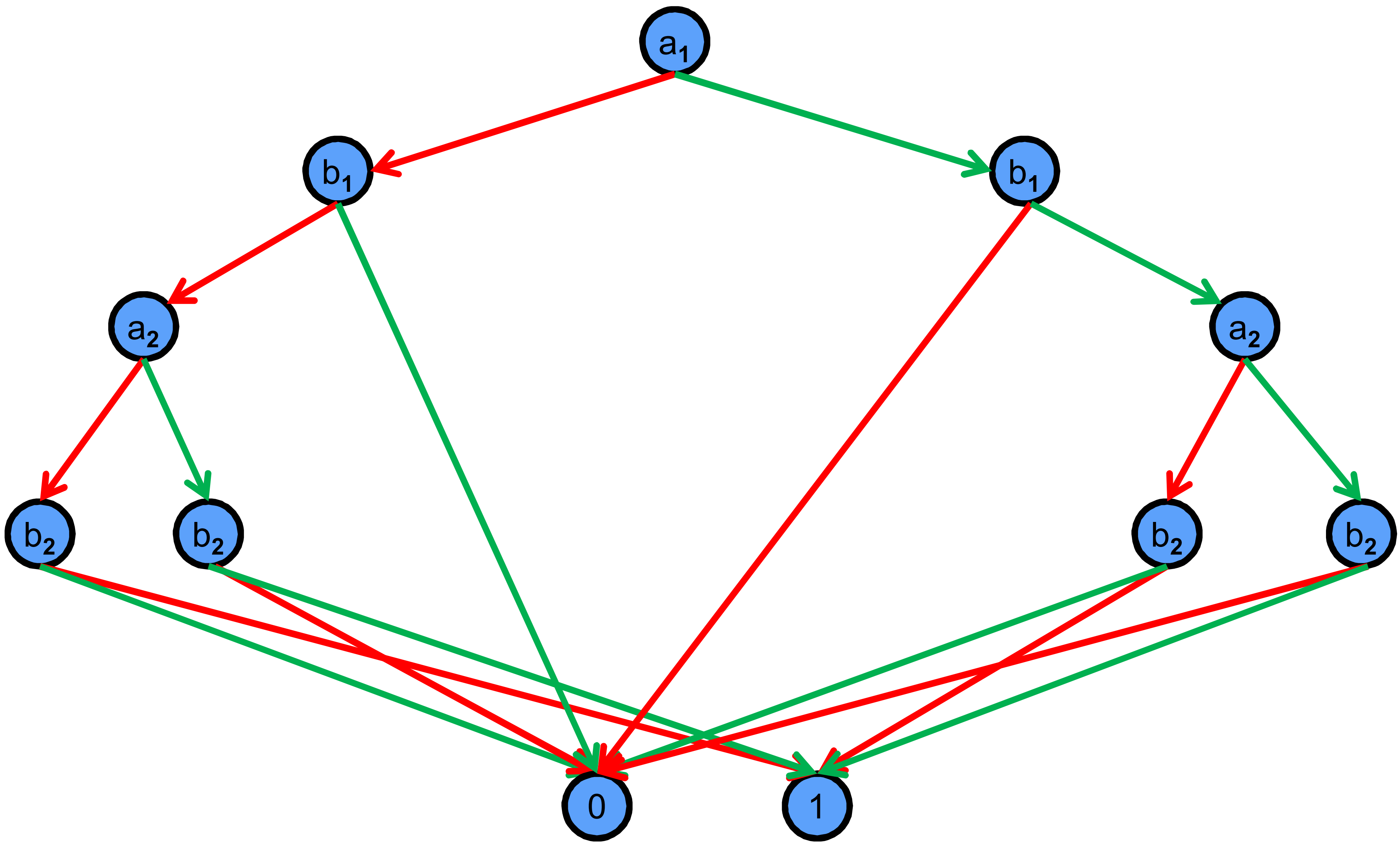


OBDT to ROBDD

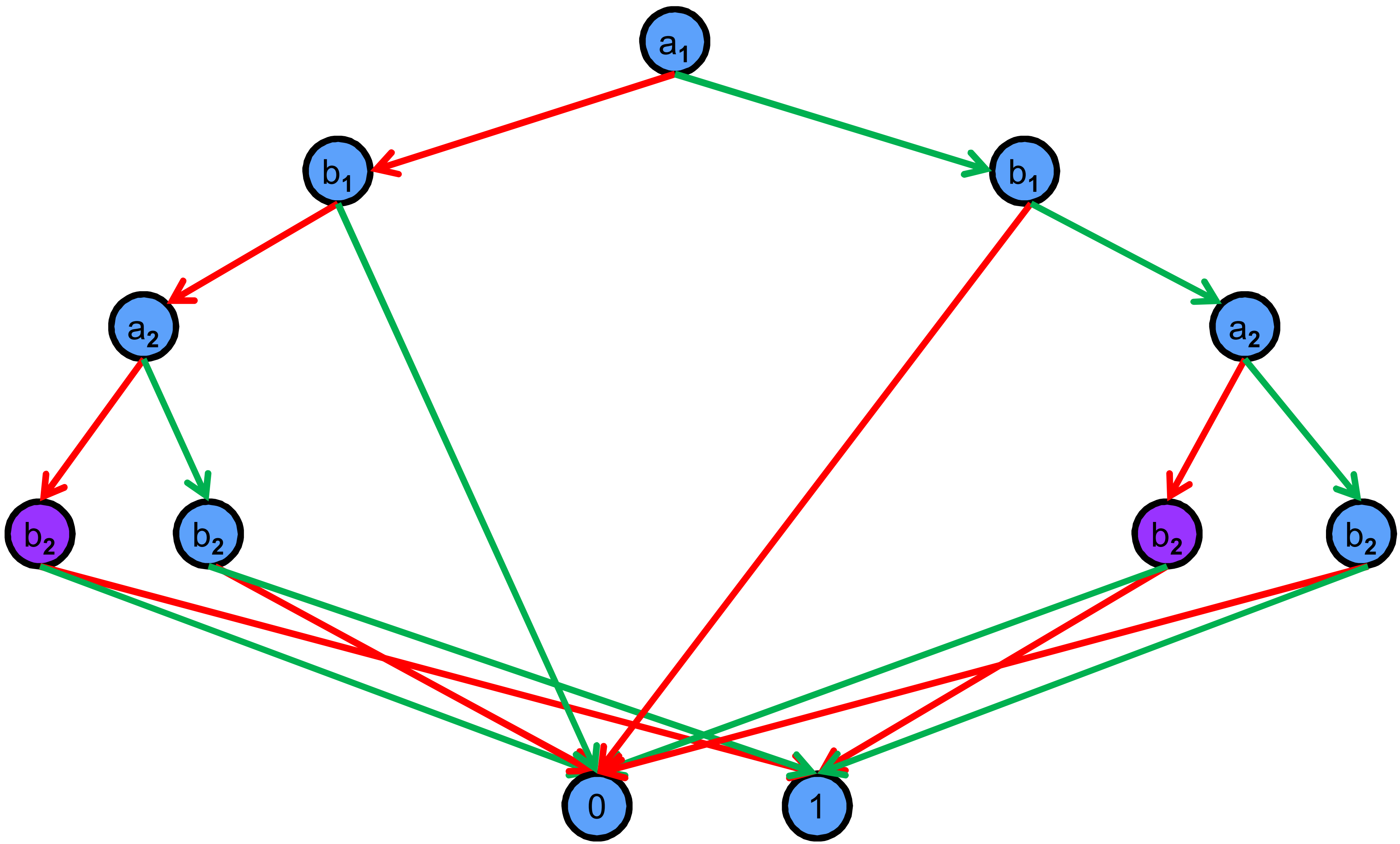
If a_1 = 0 and b_1 = 1 then f = 0 irrespective of the values of a_2 and b_2



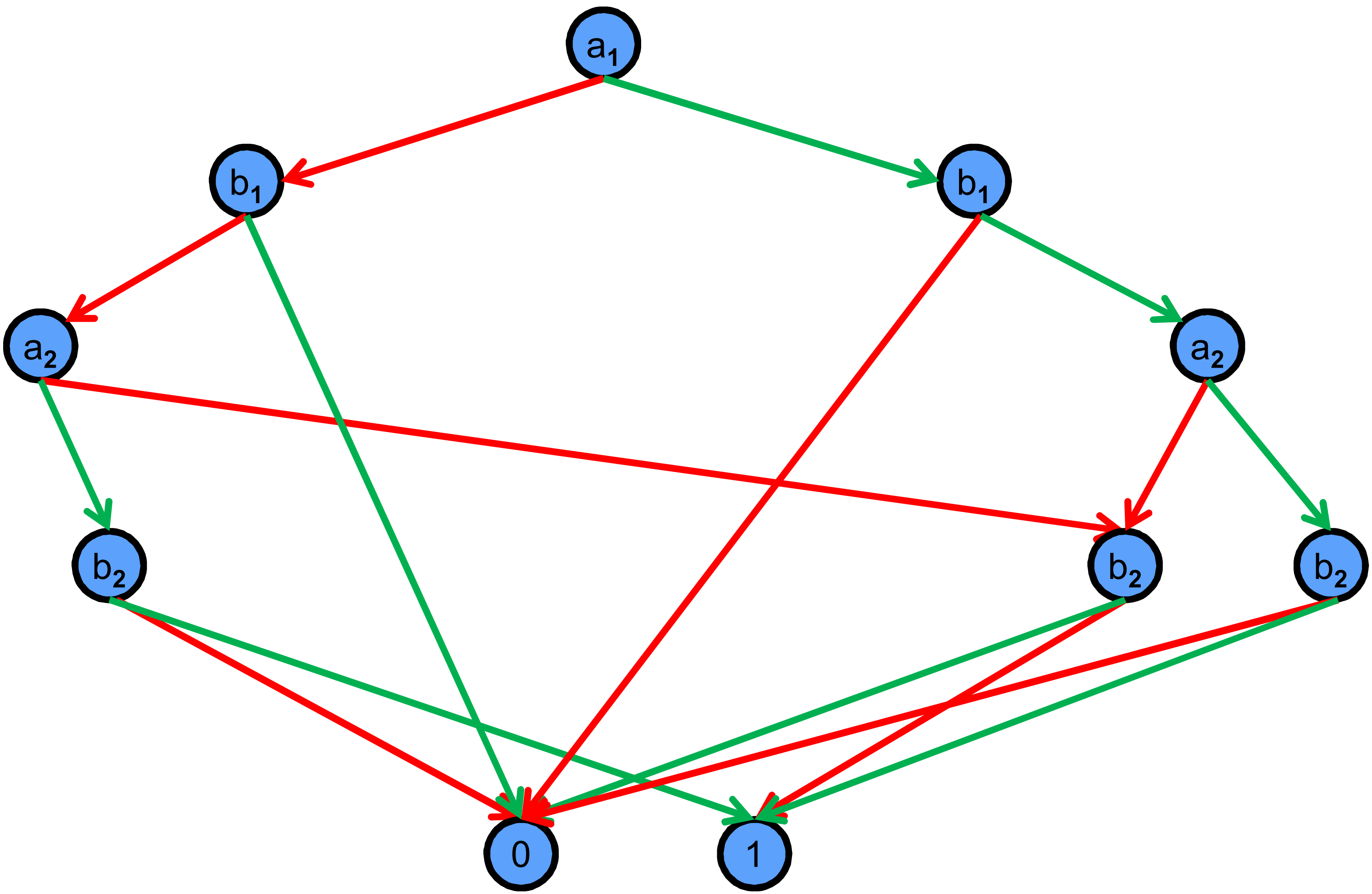
OBDT to ROBDD



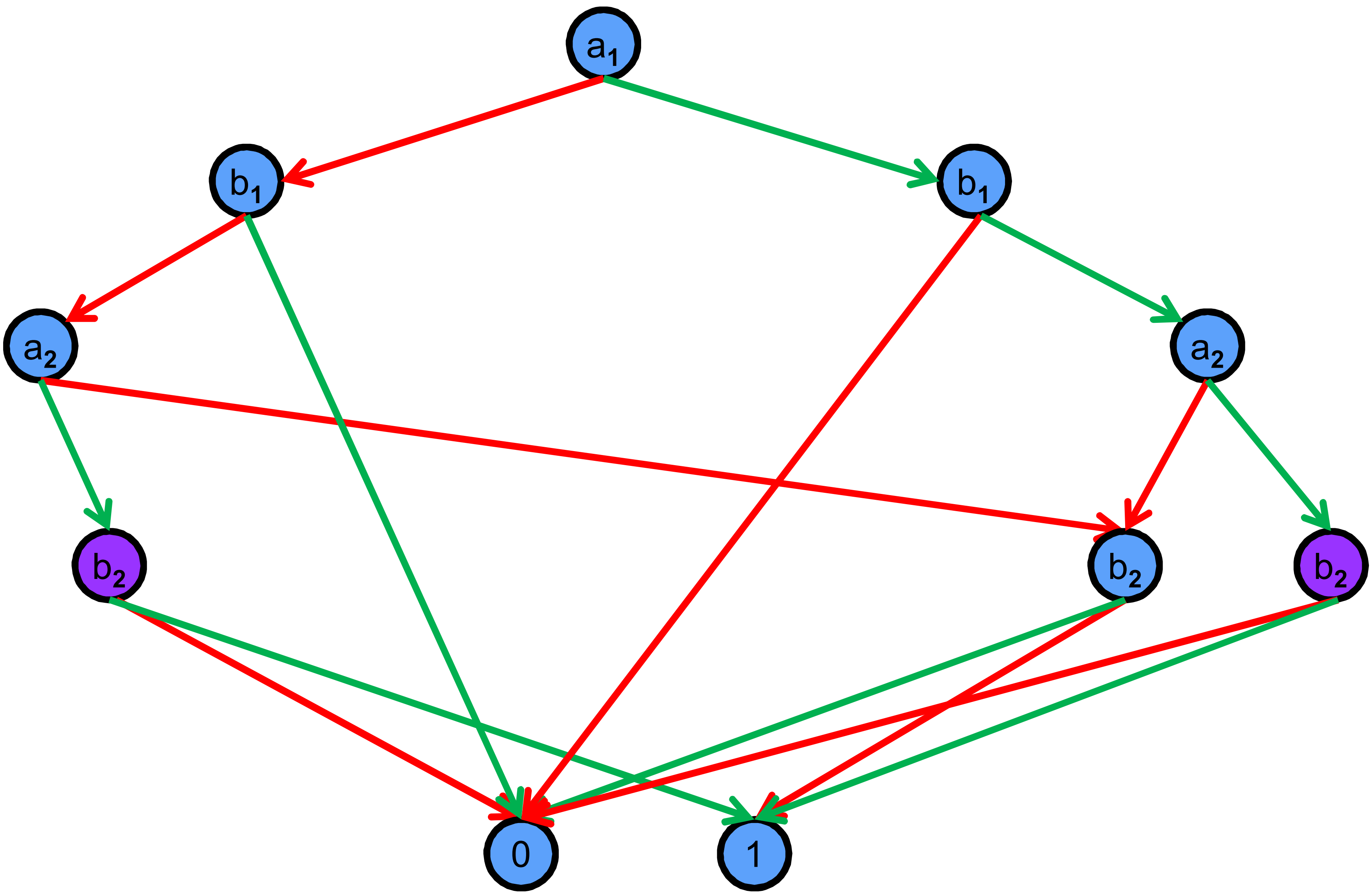
OBDT to ROBDD



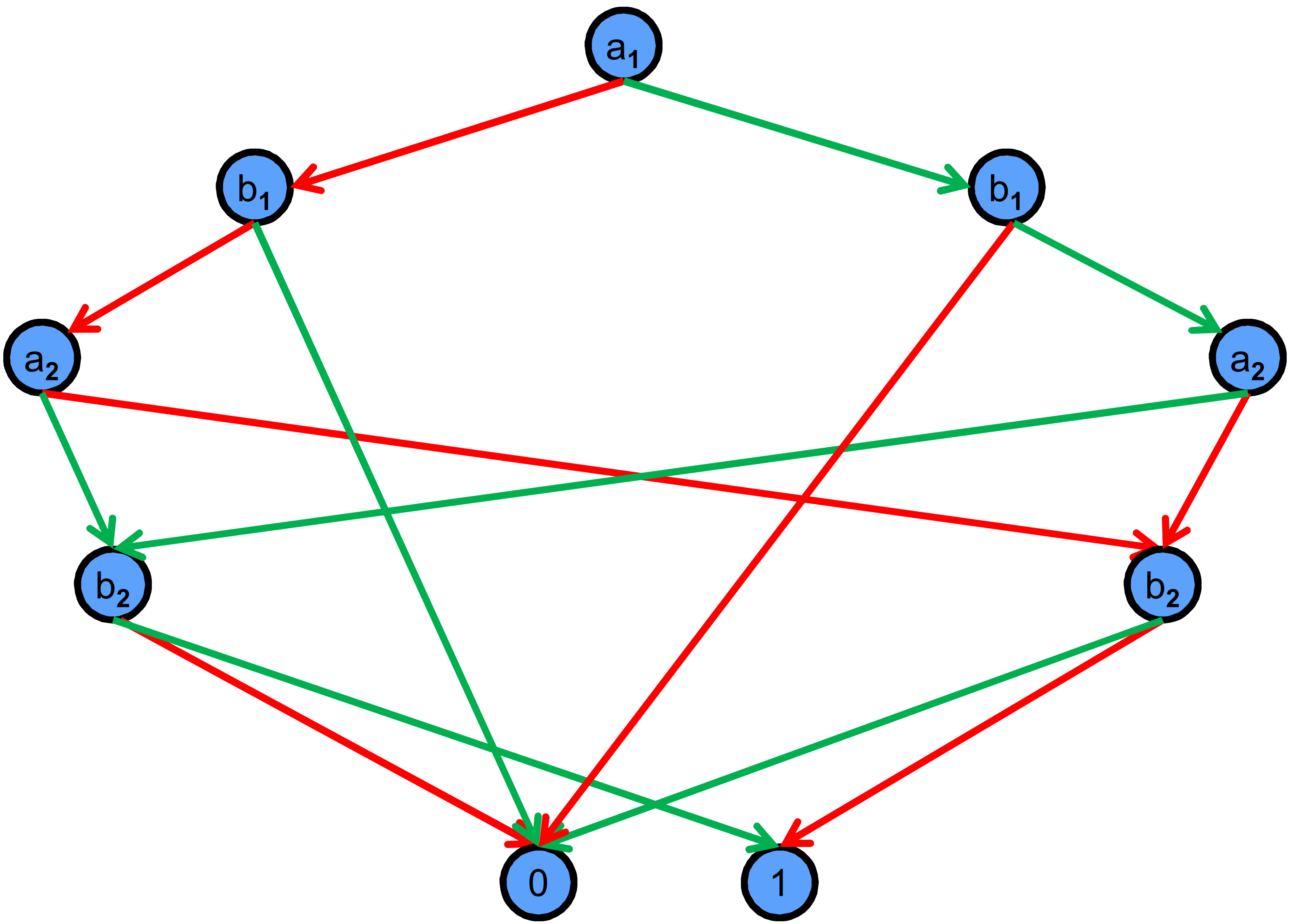
OBDT to ROBDD



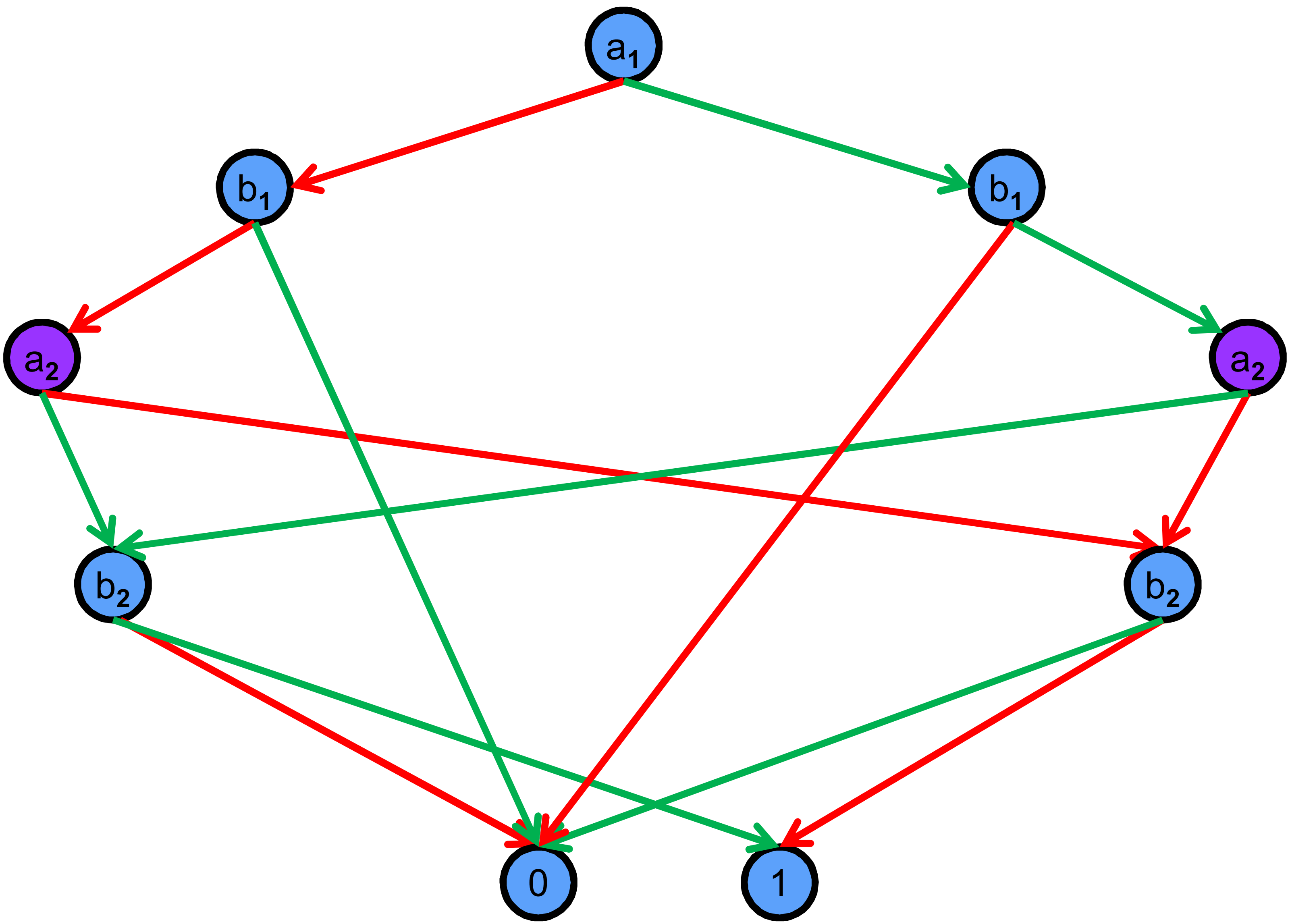
OBDT to ROBDD



OBDT to ROBDD

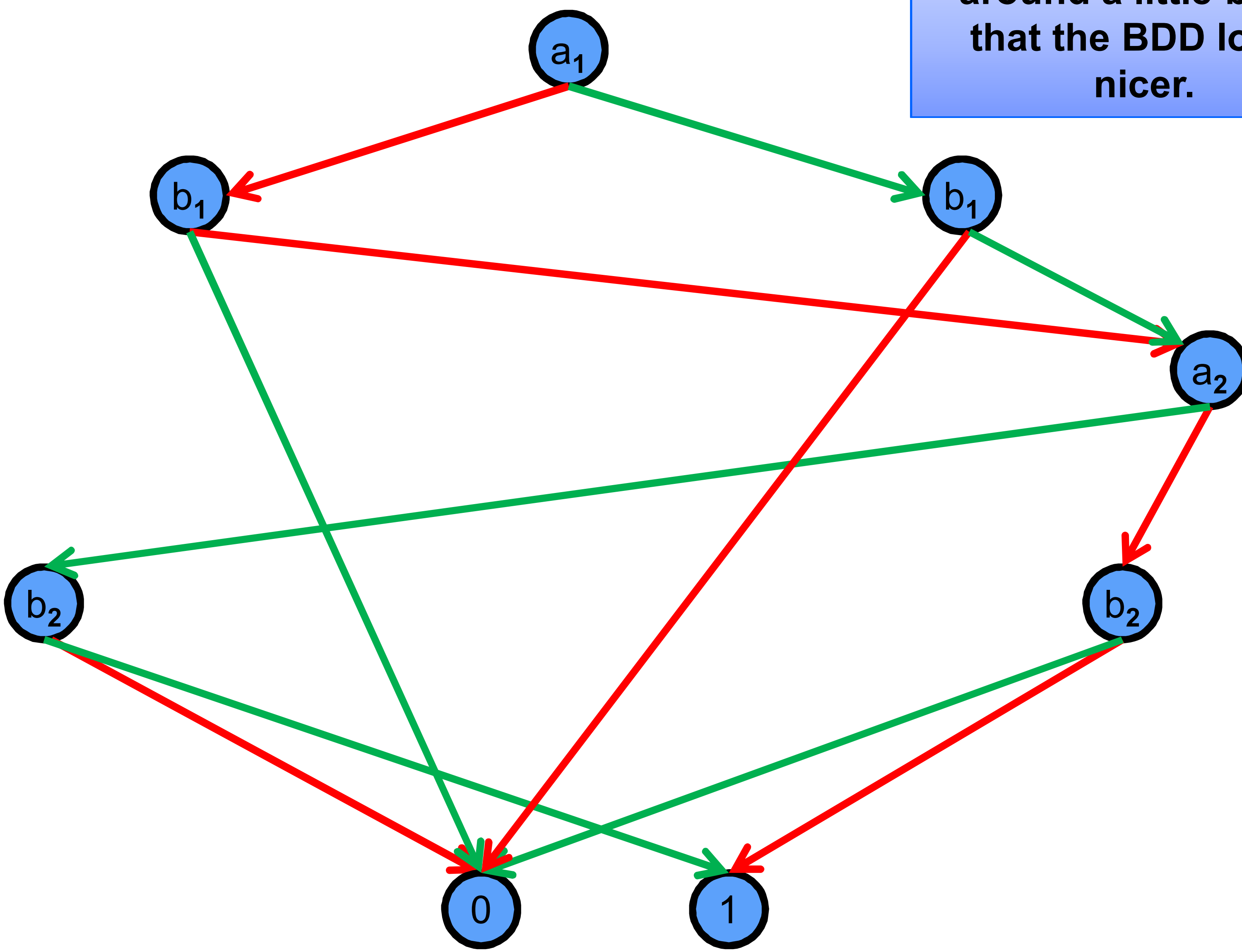


OBDT to ROBDD

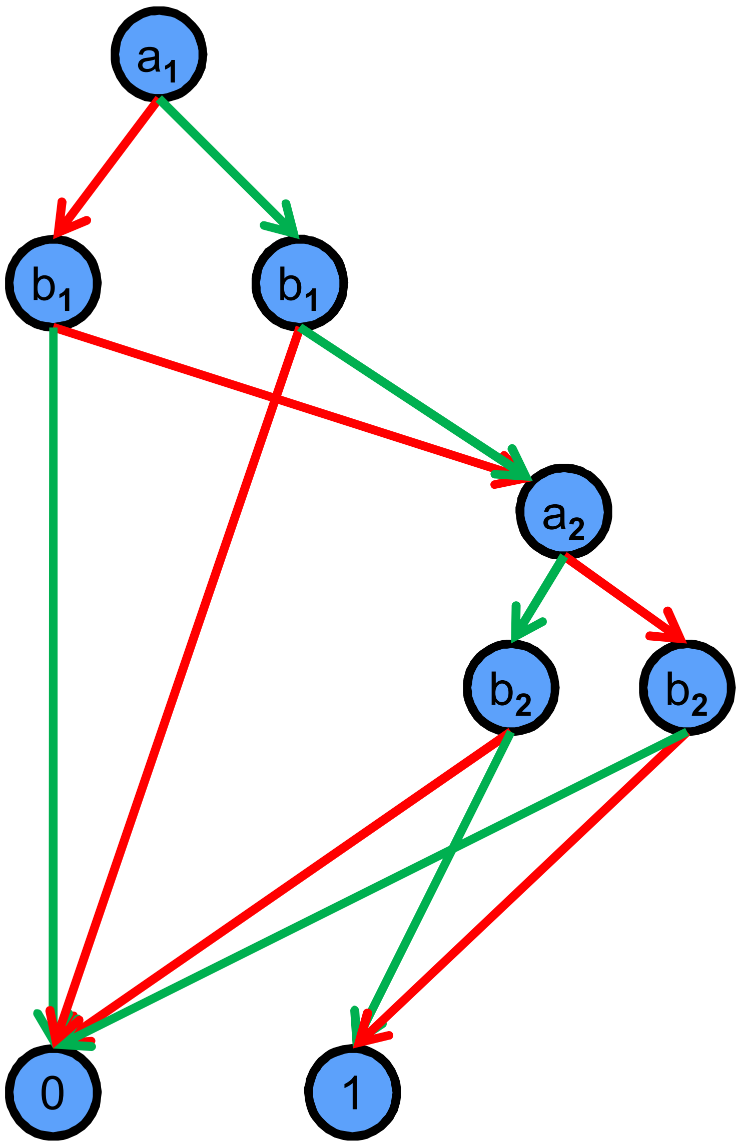


OBDT to ROBDD

Let's move things around a little bit so that the BDD looks nicer.



OBDT to ROBDD



Bryant gave a linear-time algorithm (called Reduce) to convert OBDT to ROBDD.

In practice, BDD packages don't use Reduce directly. They apply the two reductions on-the-fly as new BDDs are constructed from existing ones. Why?

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

- $f_1 = f_2$, ?

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

- $f_1 = f_2$, BDD(f_1) and BDD(f_2) are isomorphic
- f is unsatisfiable , ?

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

- $f_1 = f_2$, BDD(f_1) and BDD(f_2) are isomorphic
- f is unsatisfiable , BDD(f) is the leaf node “0”
- f is valid , ?

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

- $f_1 = f_2$, BDD(f_1) and BDD(f_2) are isomorphic
- f is unsatisfiable , BDD(f) is the leaf node “0”
- f is valid , BDD(f) is the leaf node “1”
- BDD packages do these operations in constant time

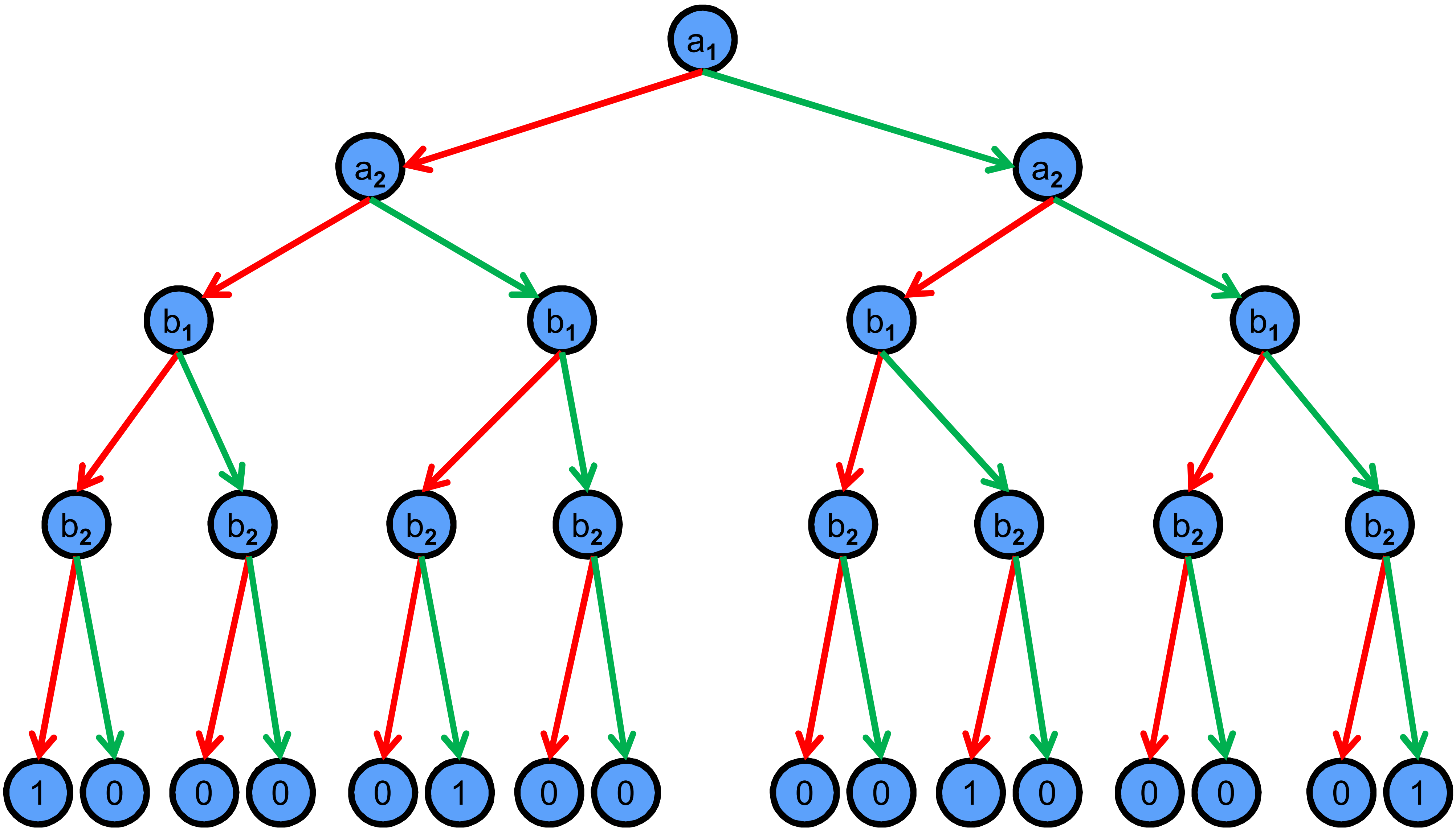
Logical operations can be performed efficiently on BDDs

- Polynomial in argument size

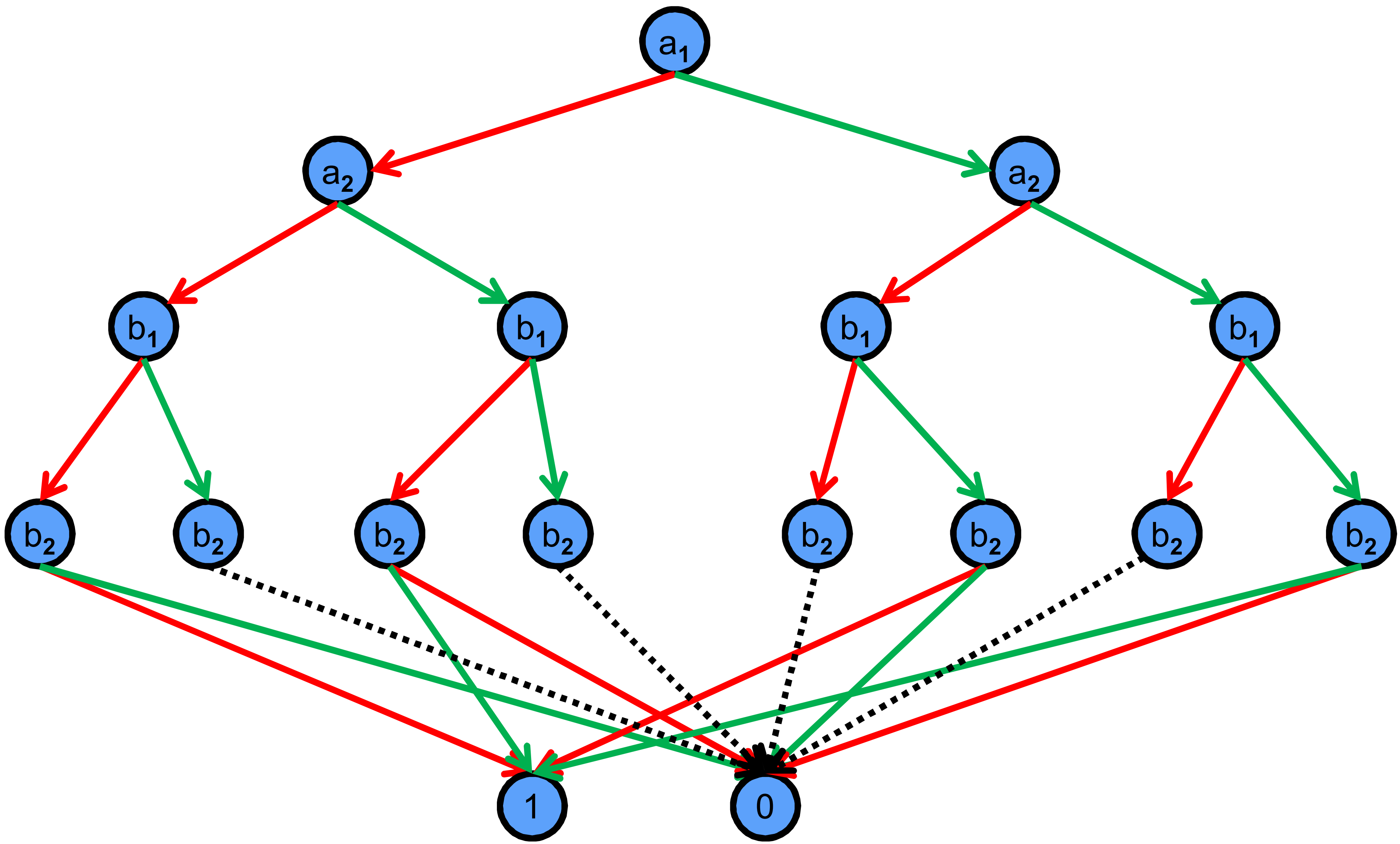
BDD size depends critically on the variable ordering

- Some formulas have exponentially large sizes for all ordering
- Others are polynomial for some ordering and exponential for others

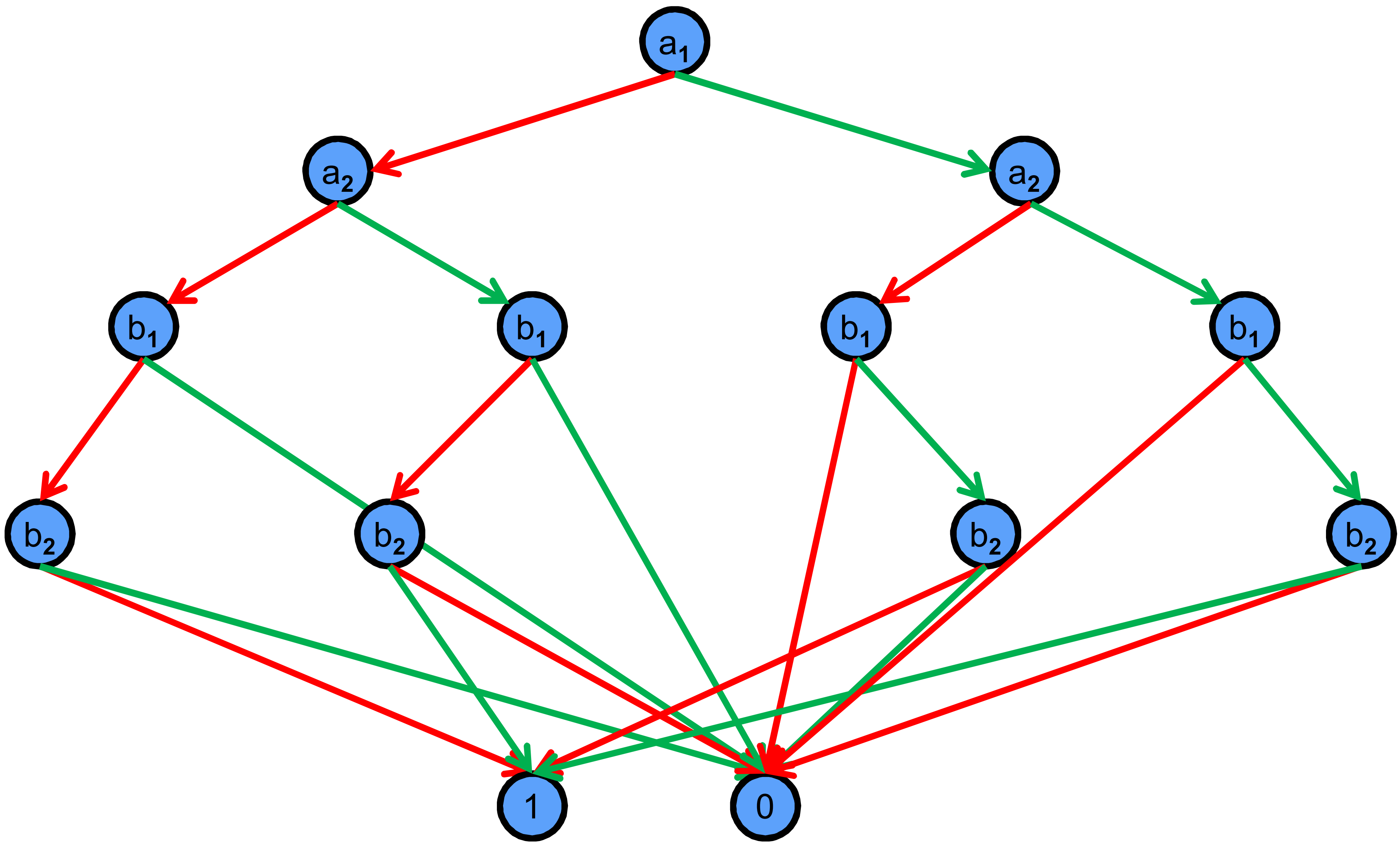
ROBDD and variable ordering



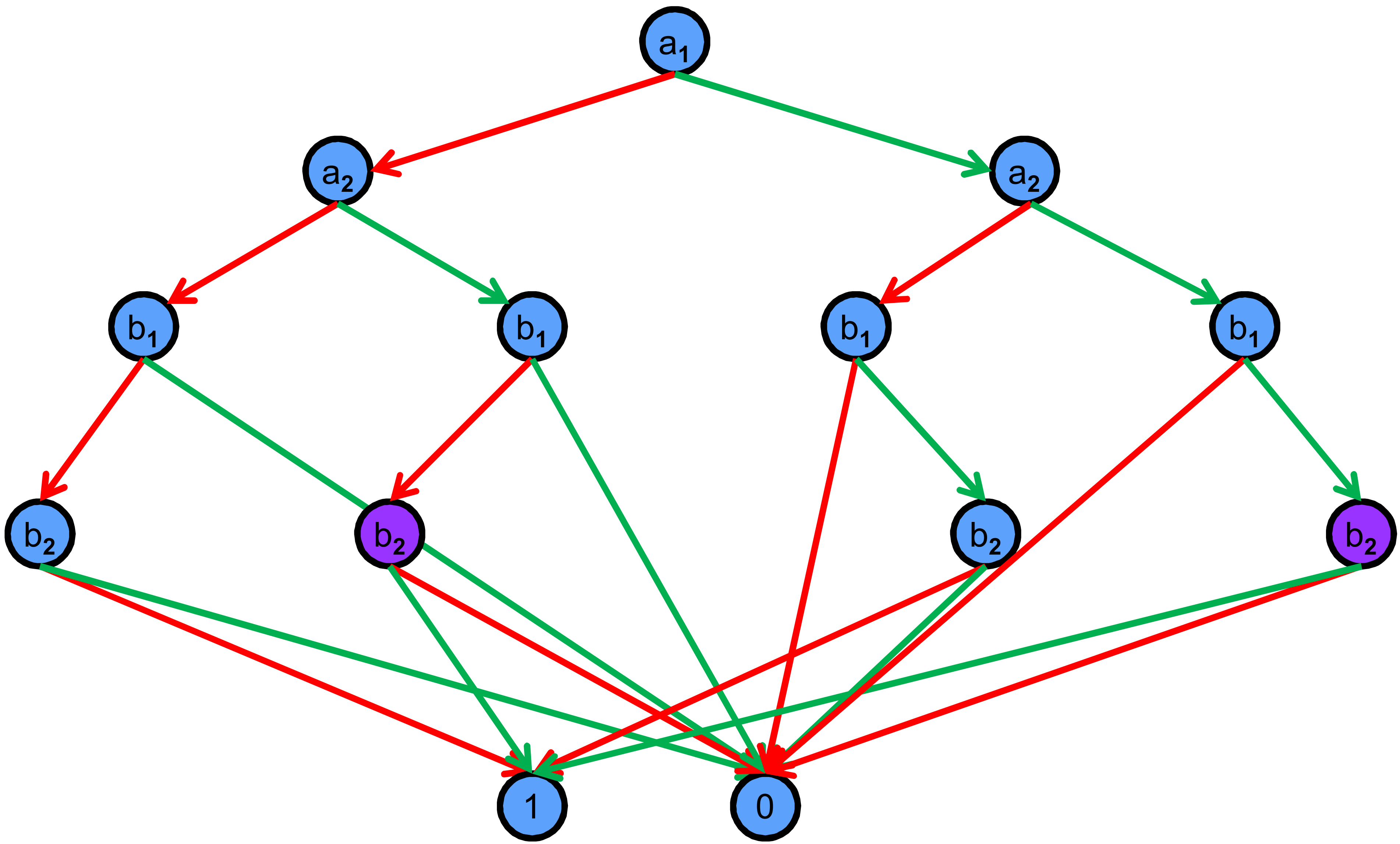
ROBDD and variable ordering



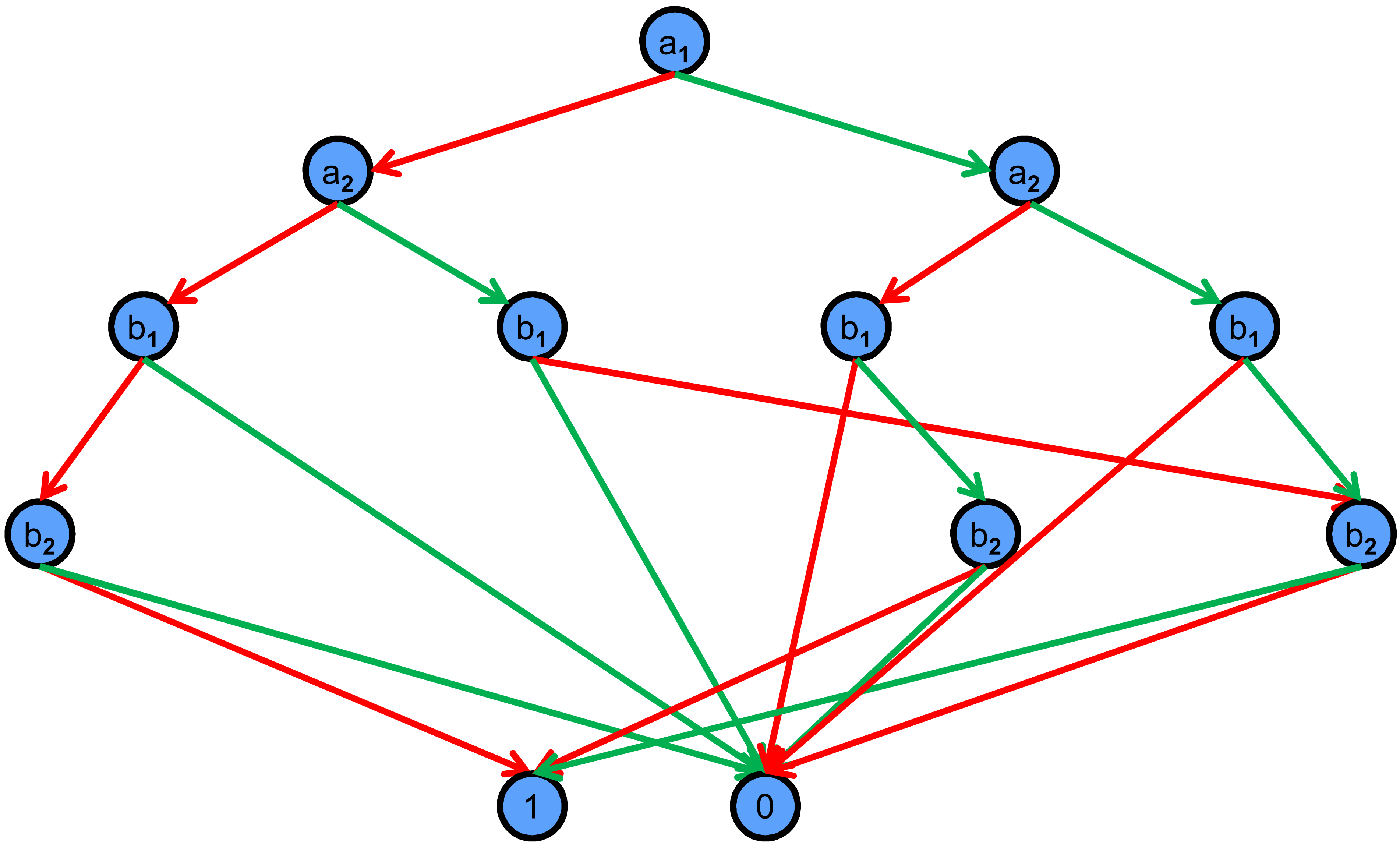
ROBDD and variable ordering



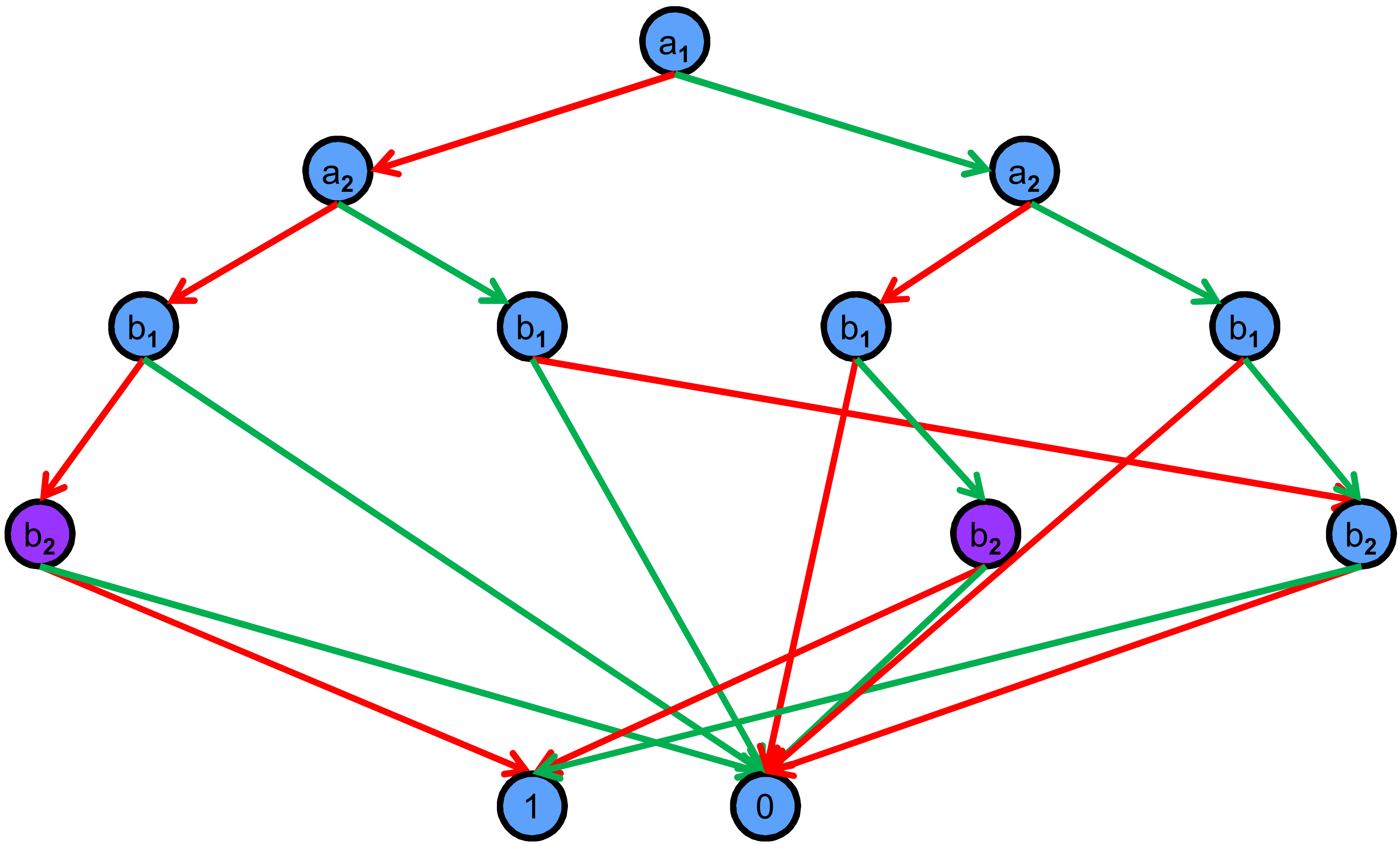
ROBDD and variable ordering



ROBDD and variable ordering

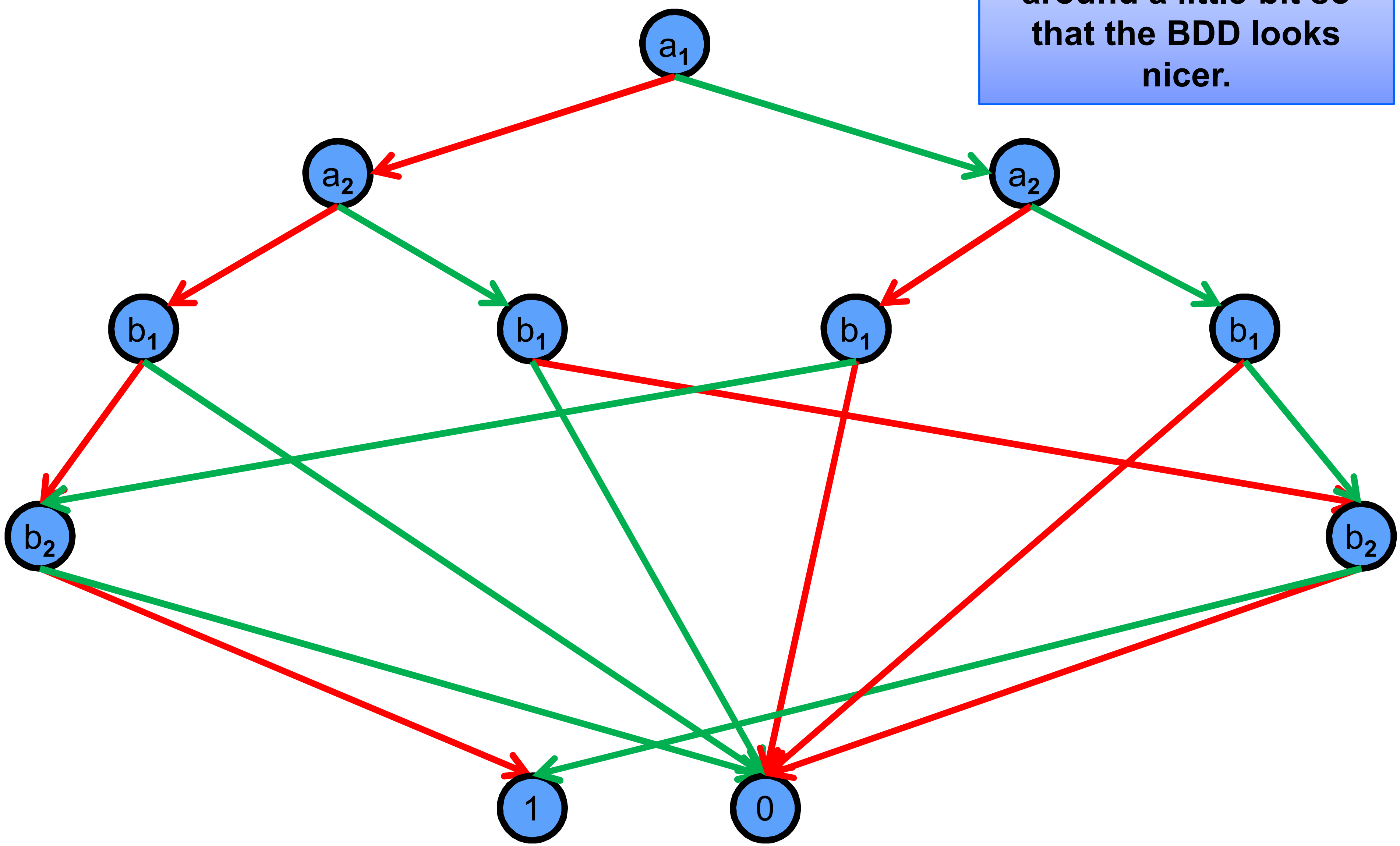


ROBDD and variable ordering

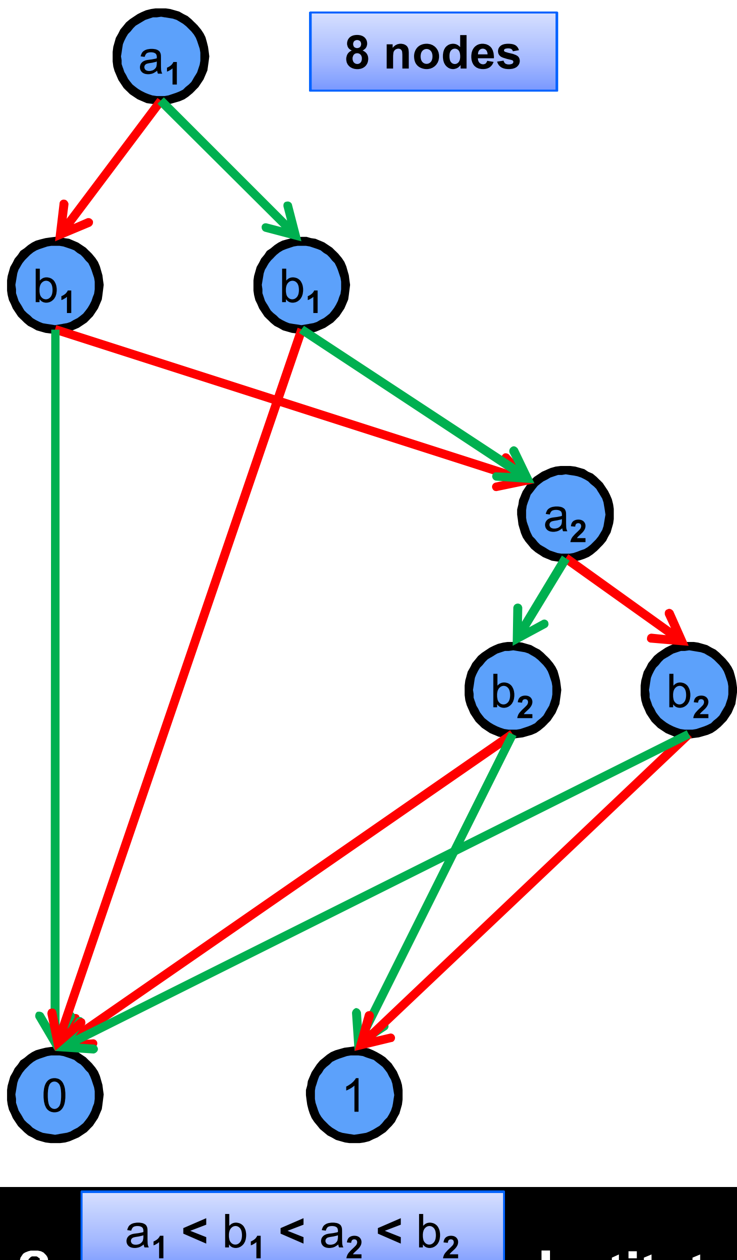


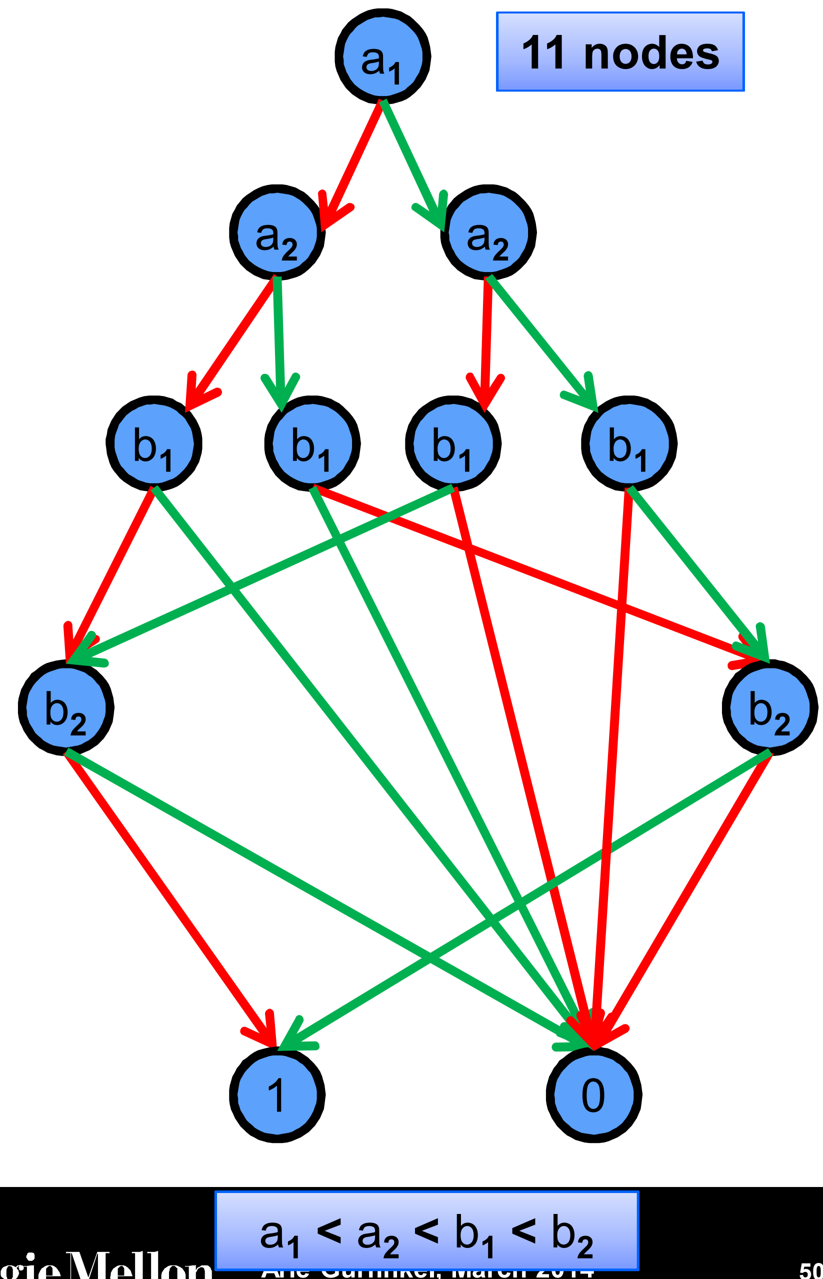
ROBDD and variable ordering

Let's move things around a little bit so that the BDD looks nicer.

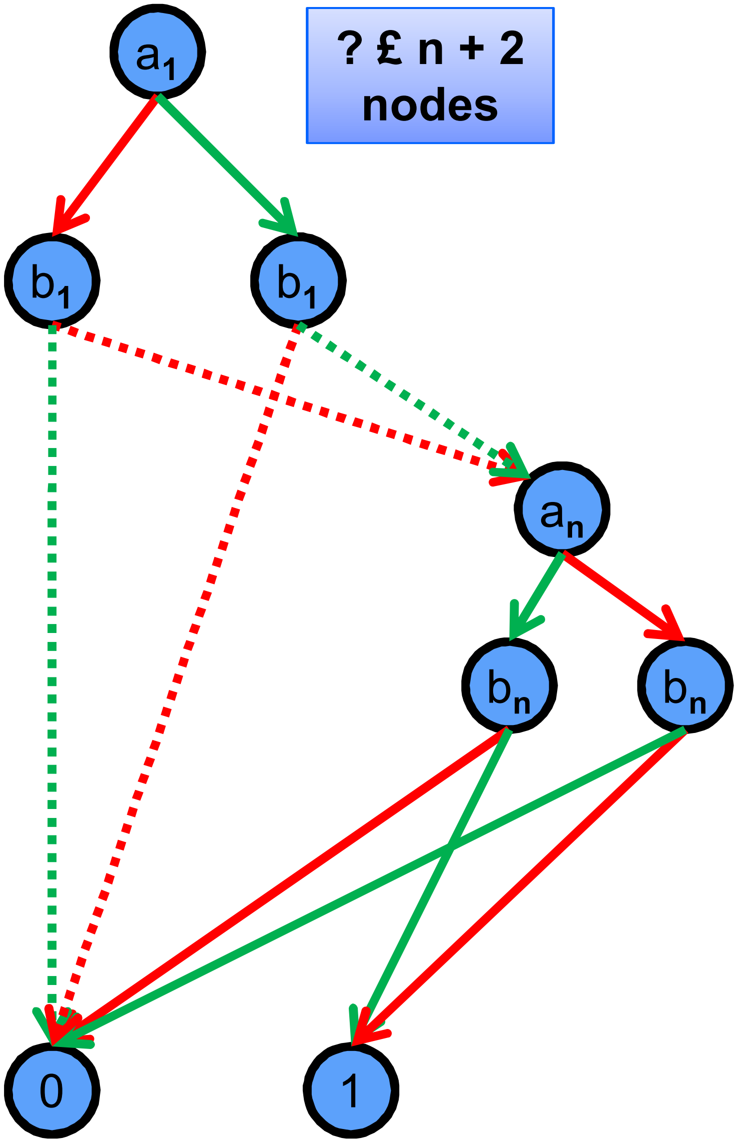


ROBDD and variable ordering



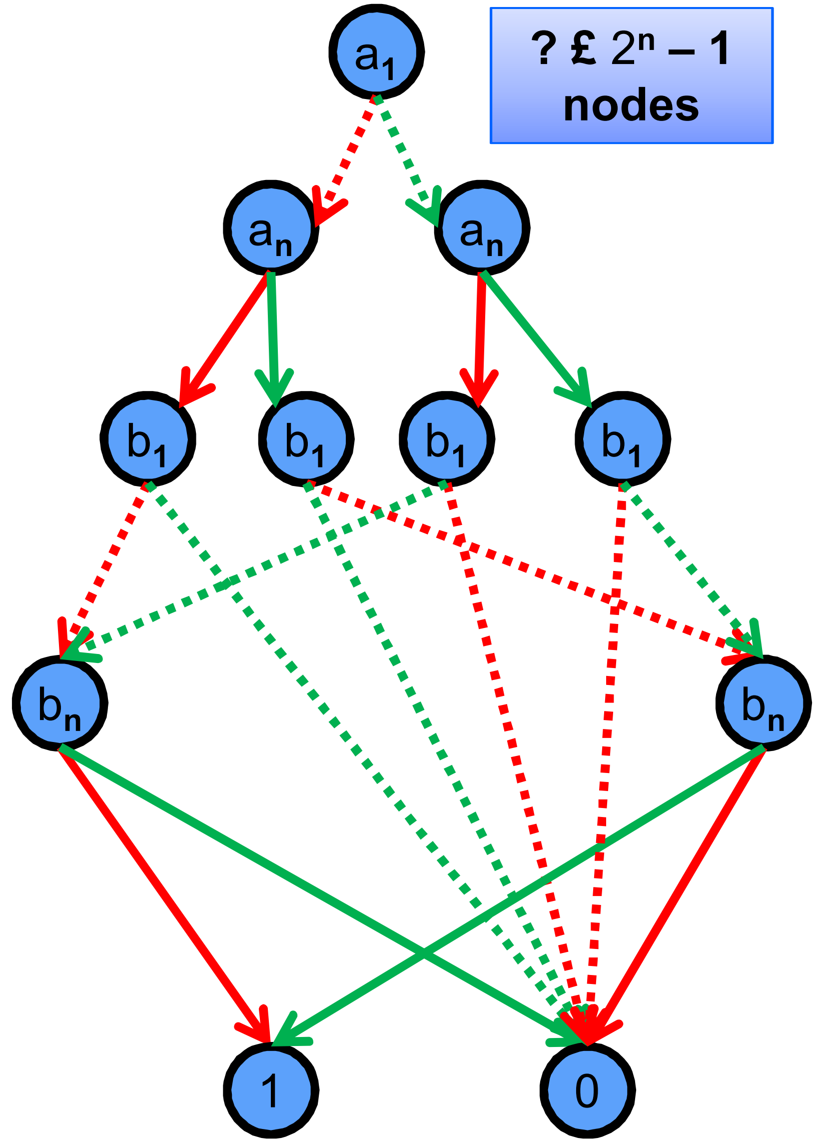


ROBDD and variable ordering



? £ n + 2
nodes

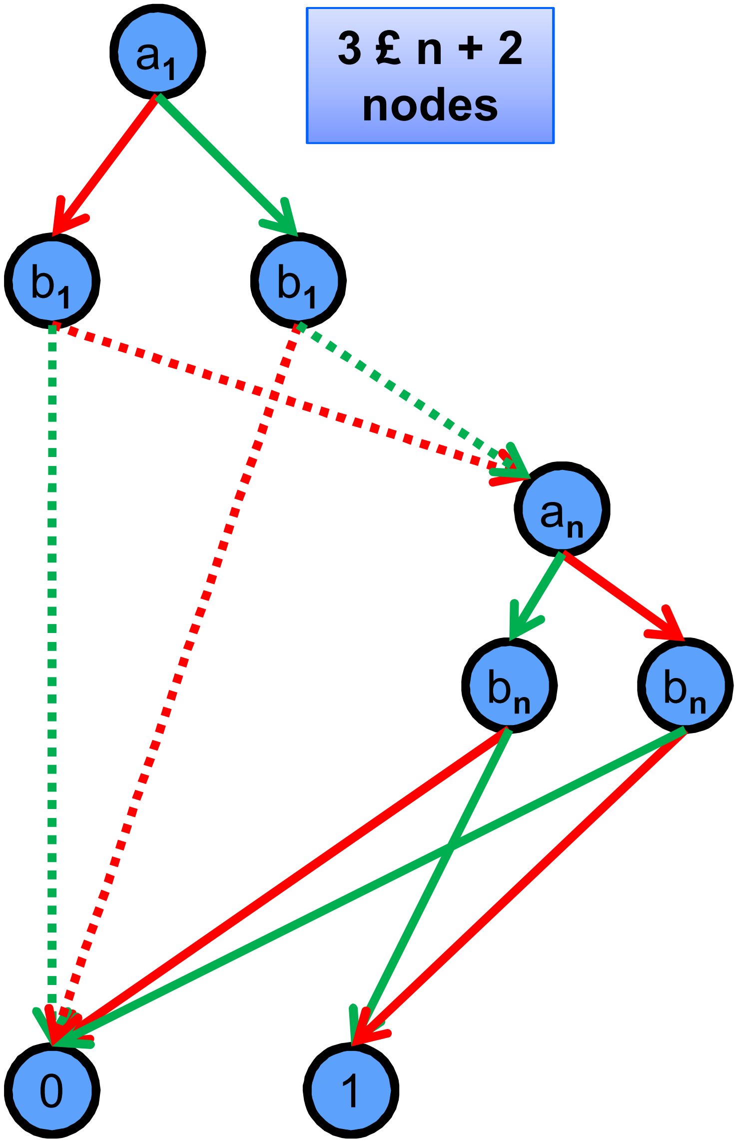
$a_1 < b_1 < \ldots < a_n < b_n$



? £ 2^n – 1
nodes

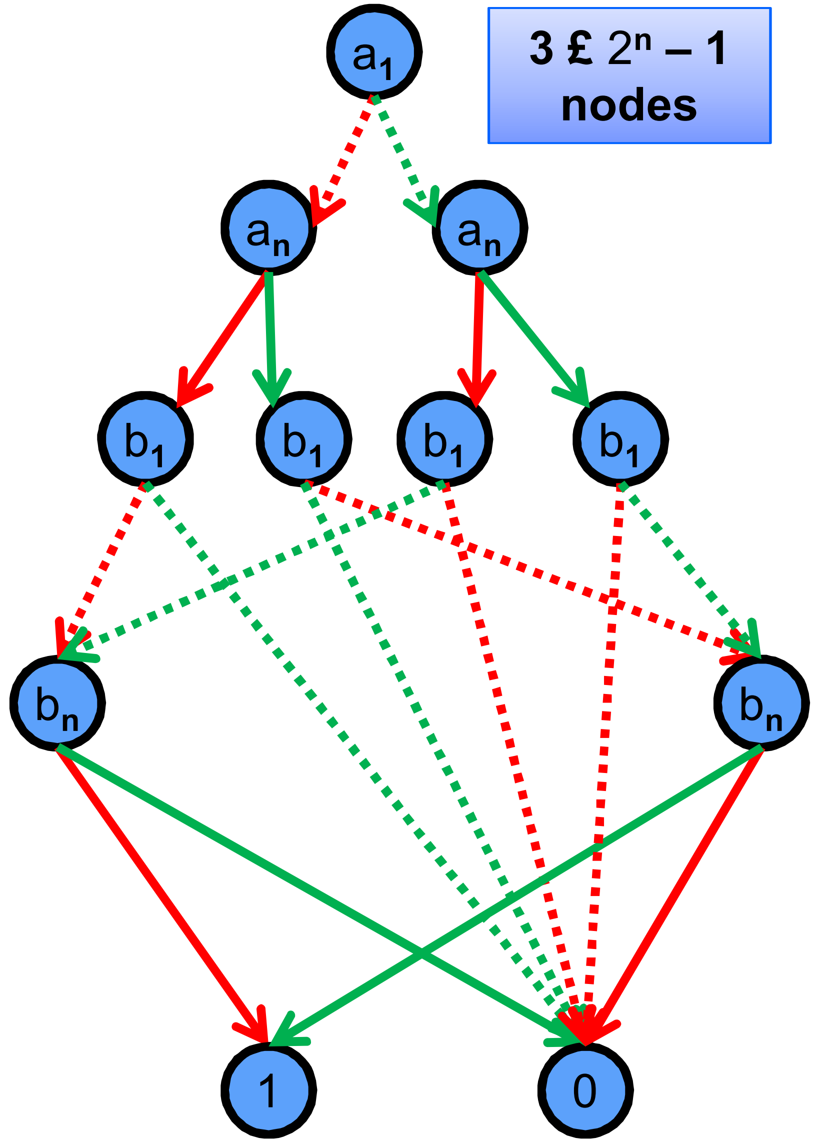
$a_1 < \ldots < a_n < b_1 < \ldots < b_n$

ROBDD and variable ordering



3 £ n + 2 nodes

$a_1 < b_1 < \ldots < a_n < b_n$



3 £ 2^n – 1 nodes

$a_1 < \ldots < a_n < b_1 < \ldots < b_n$

BDD Operations

True : BDD(TRUE)

False: BDD(FALSE)

Var : v BDD(v)

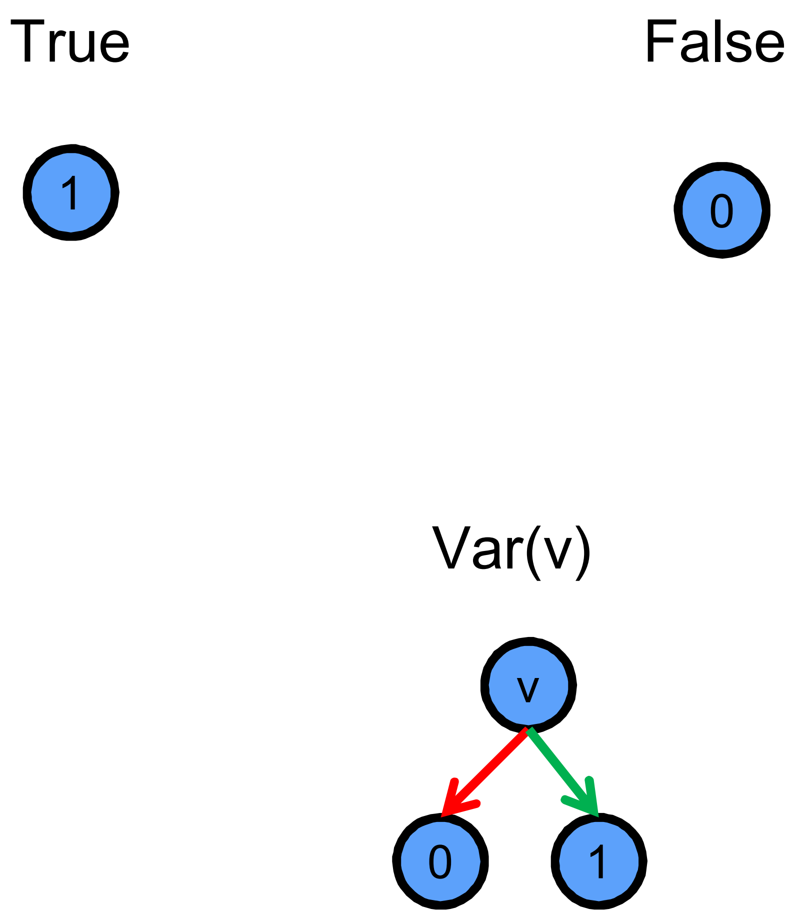
Not : BDD(f) BDD(:f)

And : BDD(f_1) £ BDD(f_2) BDD(f_1 Æ f_2)

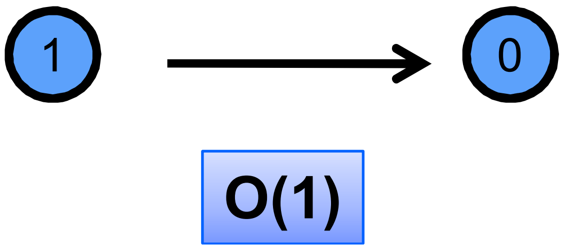
Or : BDD(f_1) £ BDD(f_2) BDD(f_1 Ç f_2)

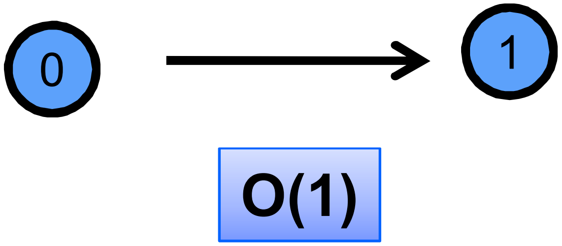
Exists : BDD(f) £ v BDD(9 v. f)

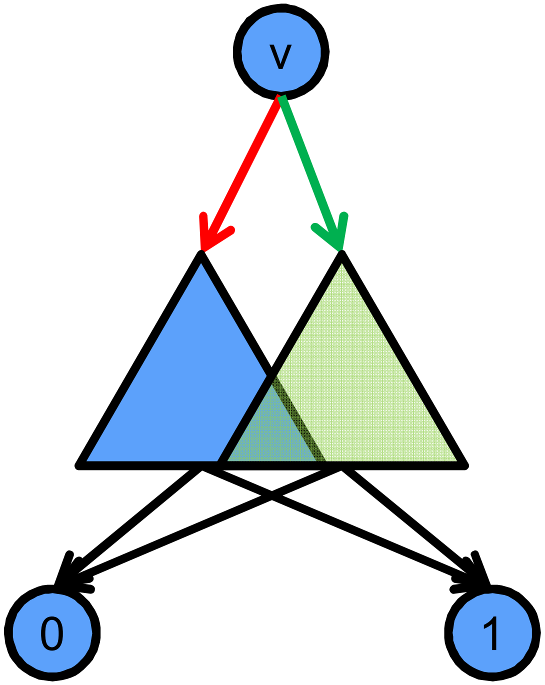
Basic BDD Operations



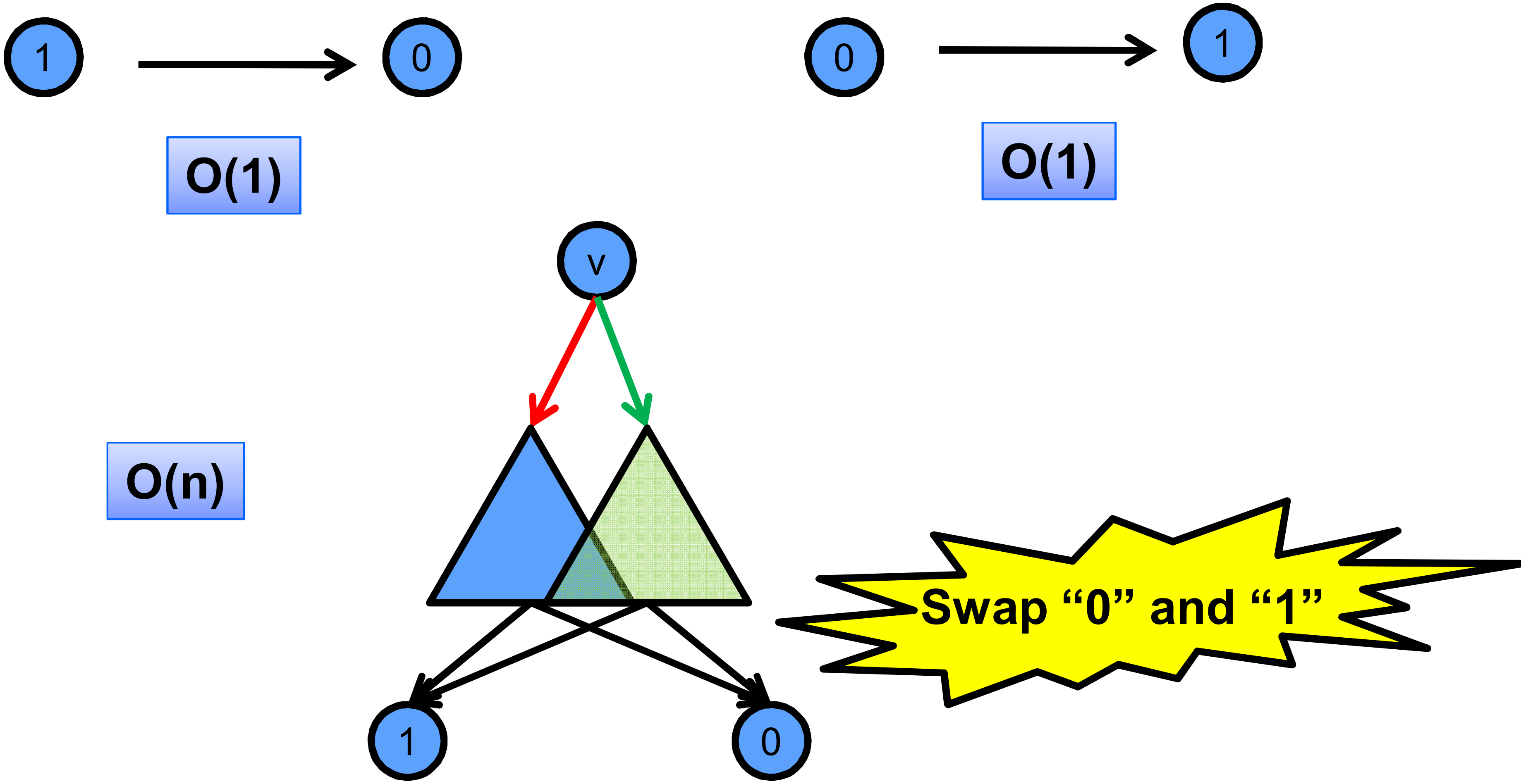
BDD Operations: Not



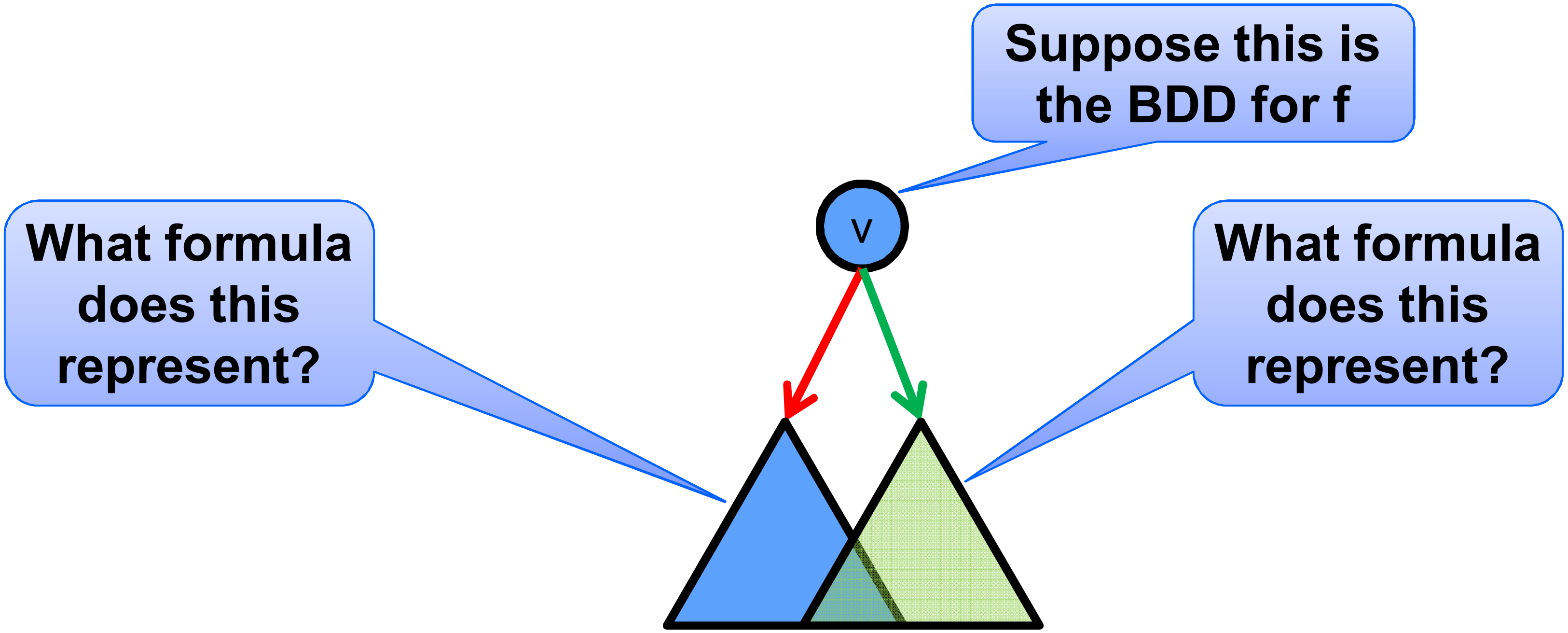




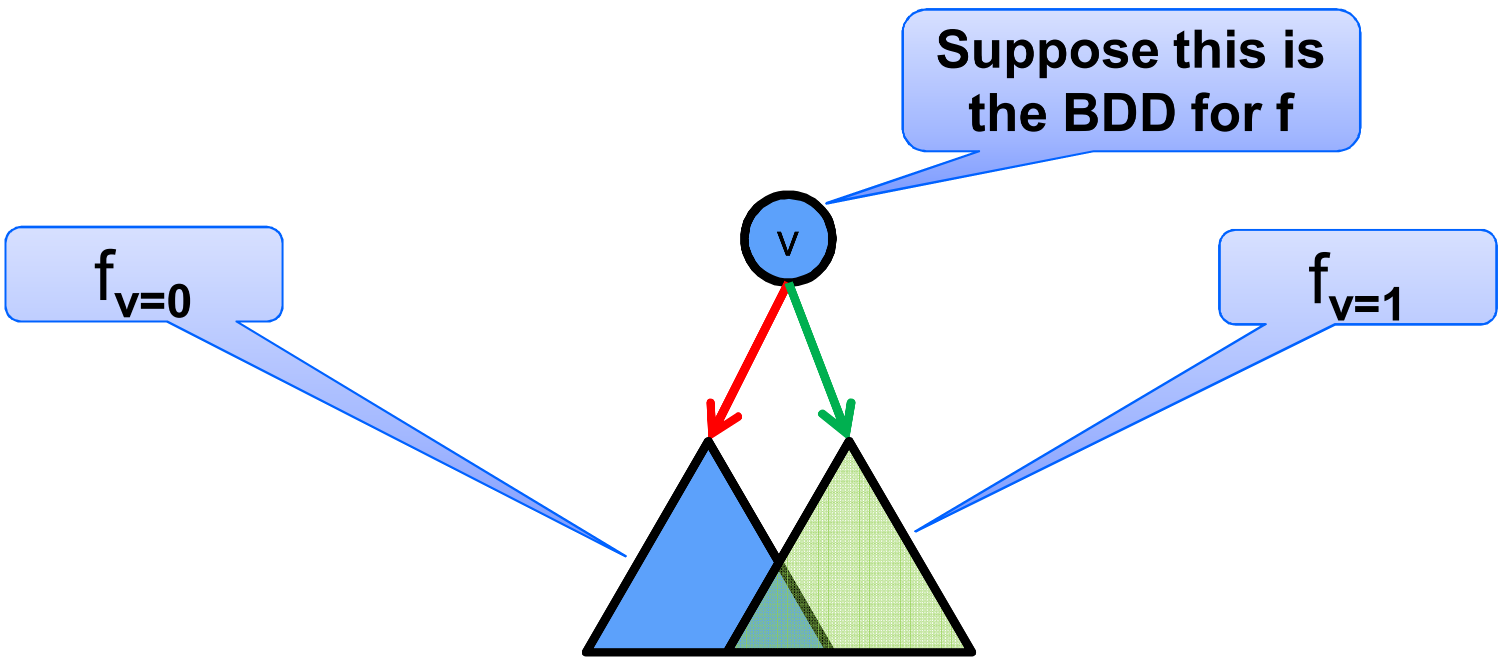
BDD Operations: Not



BDD Operations: And



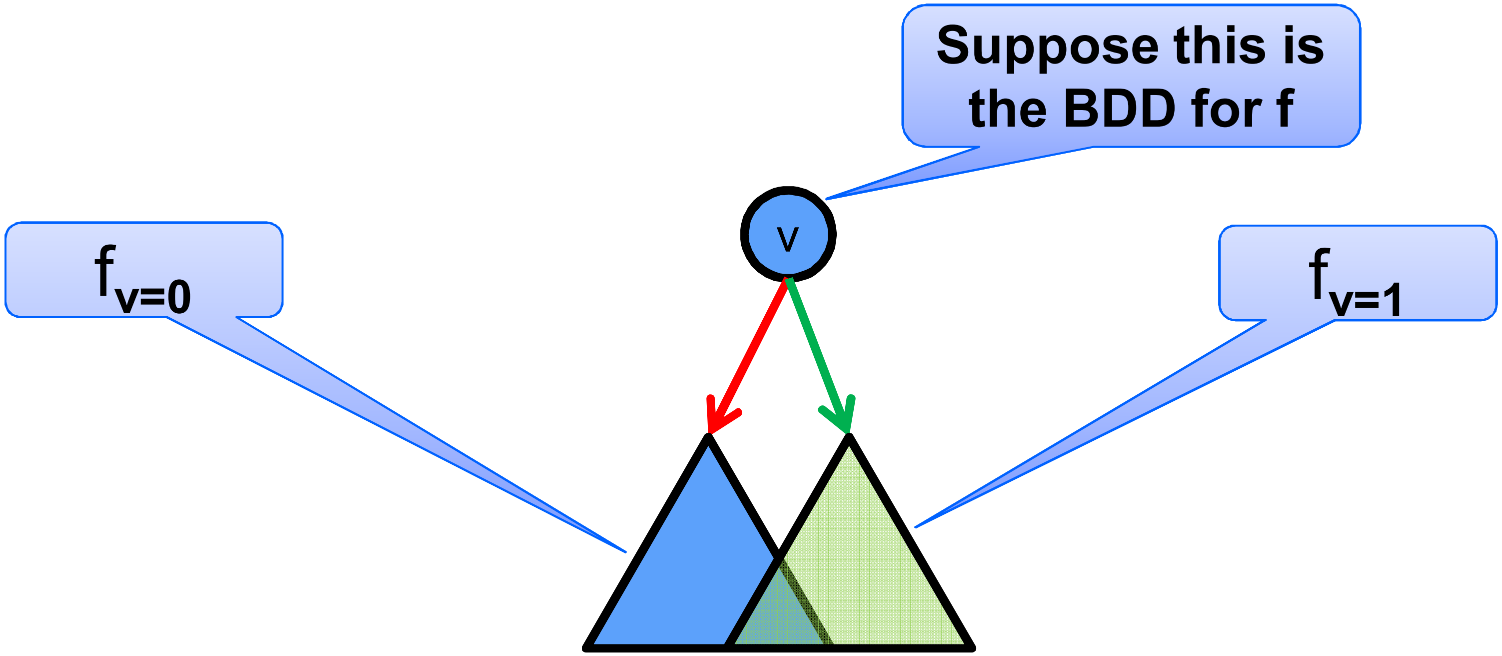
BDD Operations: And



$f_{v=0}$ and $f_{v=1}$ are known as the co-factors of f w.r.t. v

f = (X Æ $f_{v=0}$) Ç (Y Æ $f_{v=1}$)

BDD Operations: And



$f_{v=0}$ and $f_{v=1}$ are known as the co-factors of f w.r.t. v

f = (: v Æ $f_{v=0}$) Ç (v Æ $f_{v=1}$)

BDD Operations: And (Simple Cases)

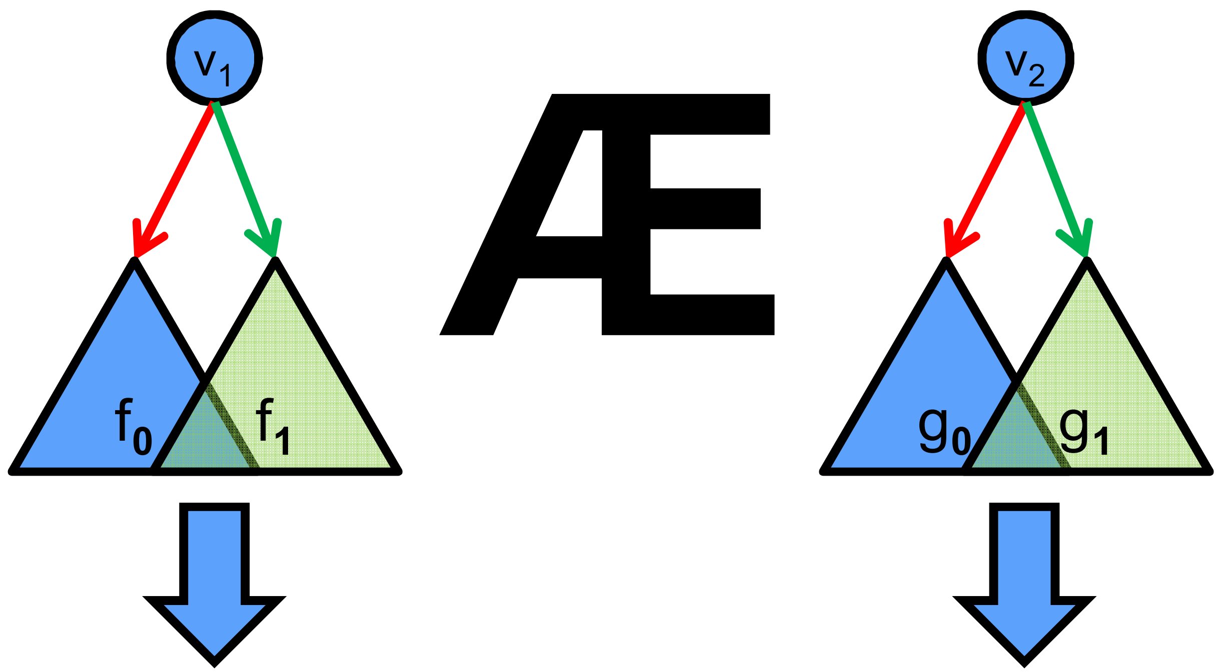
And (f, 0) = 0

And (f, 1) = f

And (1 ,f) = f

And (0 ,f) = 0

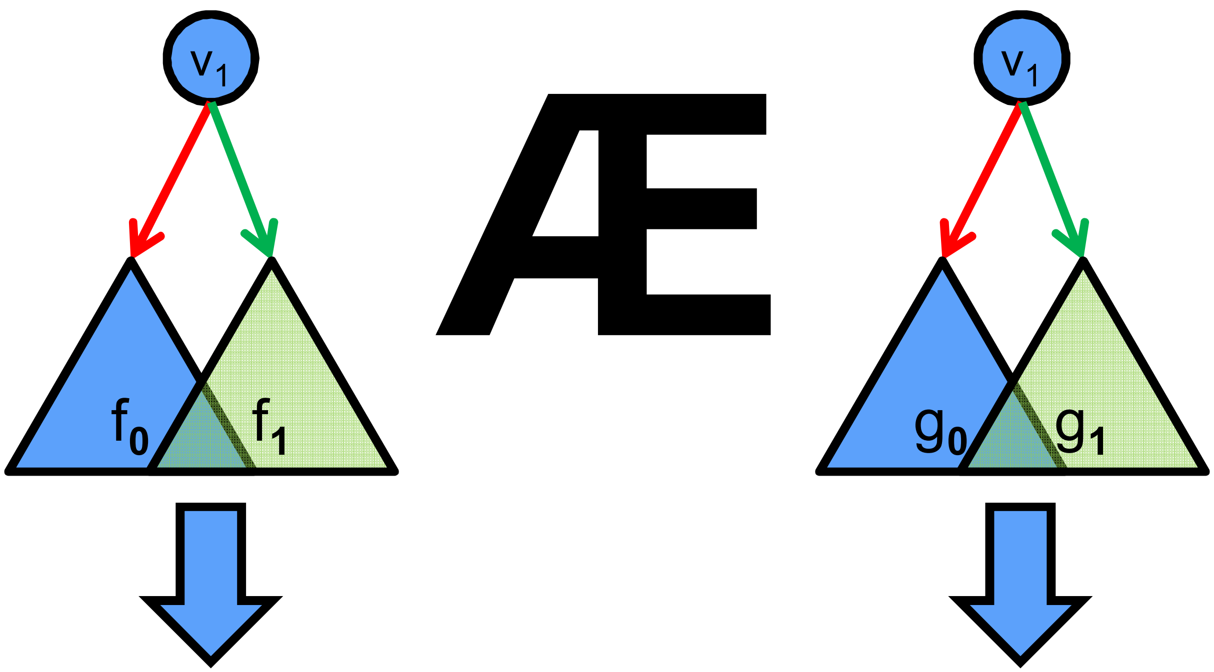
BDD Operations: And (Complex Case)



(: v_1 Æ f_0) Ç (v_1 Æ f_1) Æ (: v_2 Æ g_0) Ç (v_2 Æ g_1)

BDD Operations: And (Complex Case 1)

$v_1 = v_2$



(: v_1 Æ f_0) Ç (v_1 Æ f_1) Æ (: v_1 Æ g_0) Ç (v_1 Æ g_1)

BDD Operations: And (Complex Case 1)

$v_1 = v_2$

(: v_1 Æ X) Ç (v_1 Æ Y)

(: v_1 Æ f_0) Ç (v_1 Æ f_1) Æ (: v_1 Æ g_0) Ç (v_1 Æ g_1)

BDD Operations: And (Complex Case 1)

$v_1 = v_2$

Compute recursively

(: v_1 Æ (f_0 Æ g_0)) Ç (v_1 Æ (f_1 Æ g_1))

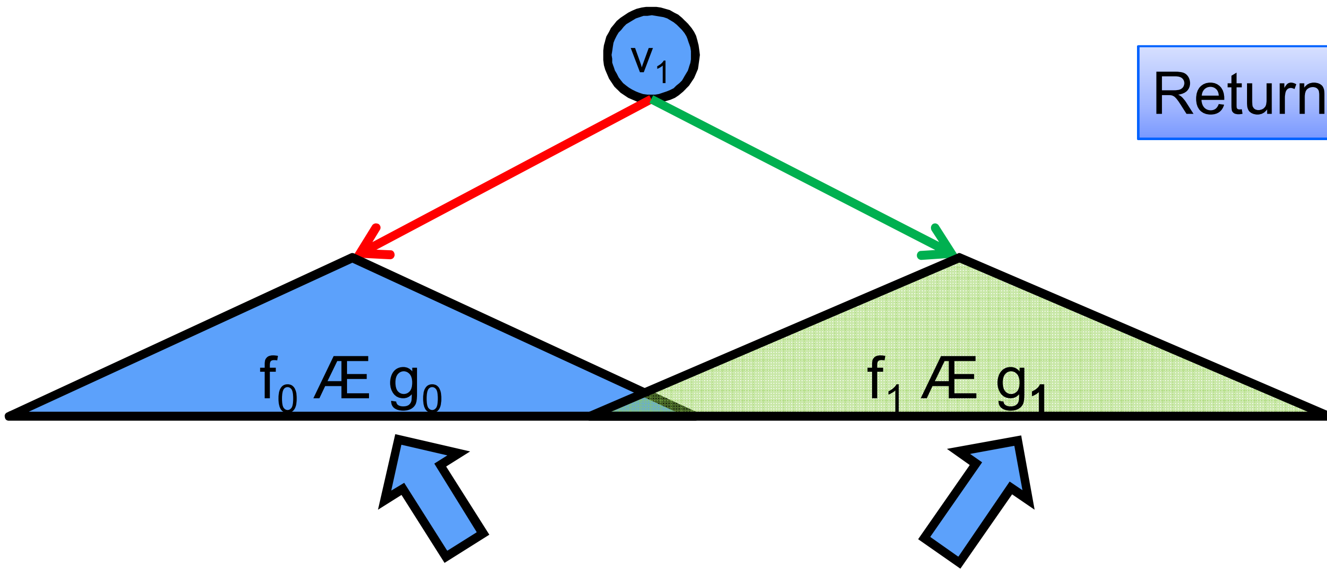
(: v_1 Æ f_0) Ç (v_1 Æ f_1) Æ (: v_1 Æ g_0) Ç (v_1 Æ g_1)

BDD Operations: And (Complex Case 1)

$v_1 = v_2$

What if f_0 Æ g_0 = f_1 Æ g_1 ?

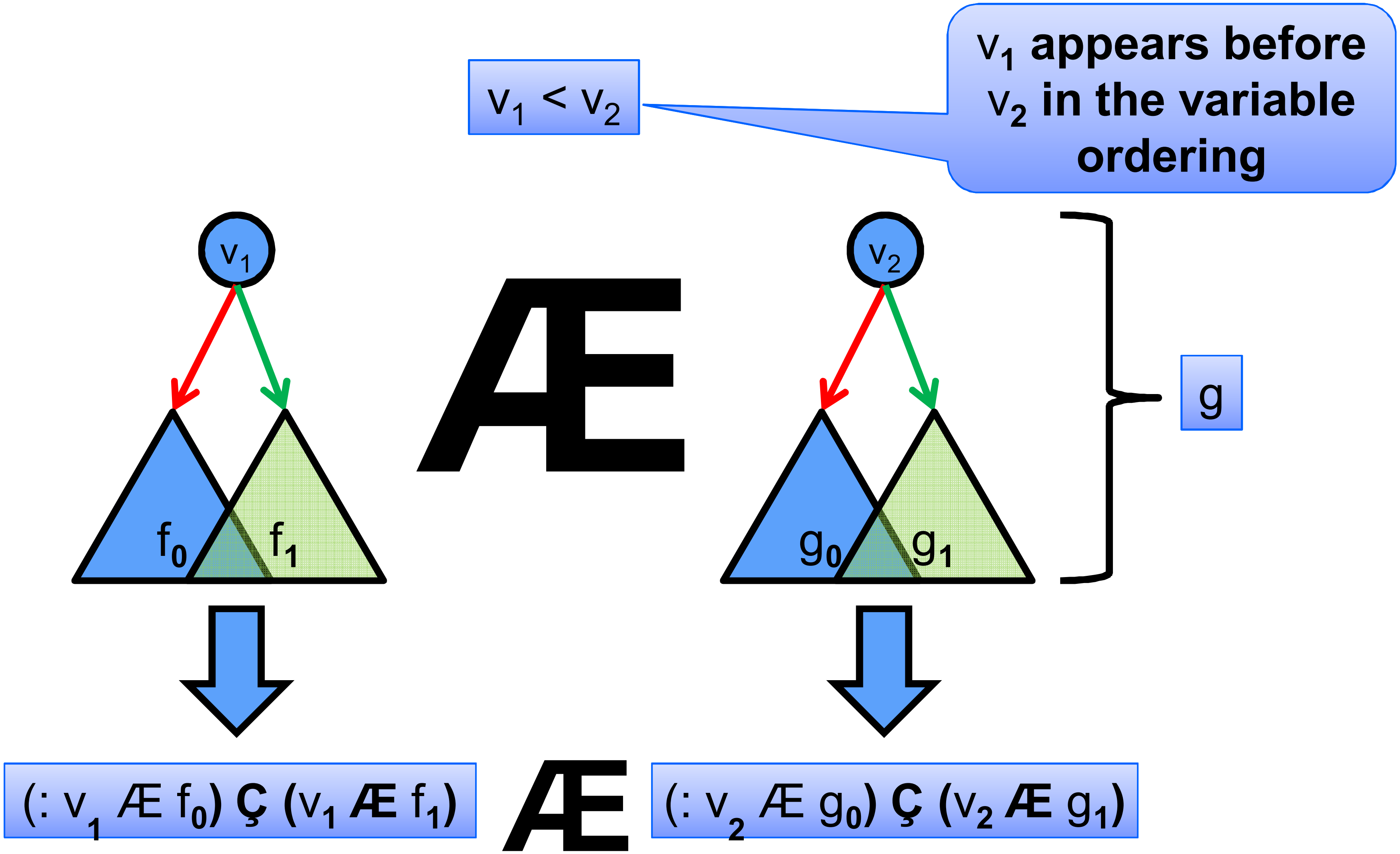
Return f_0 Æ g_0



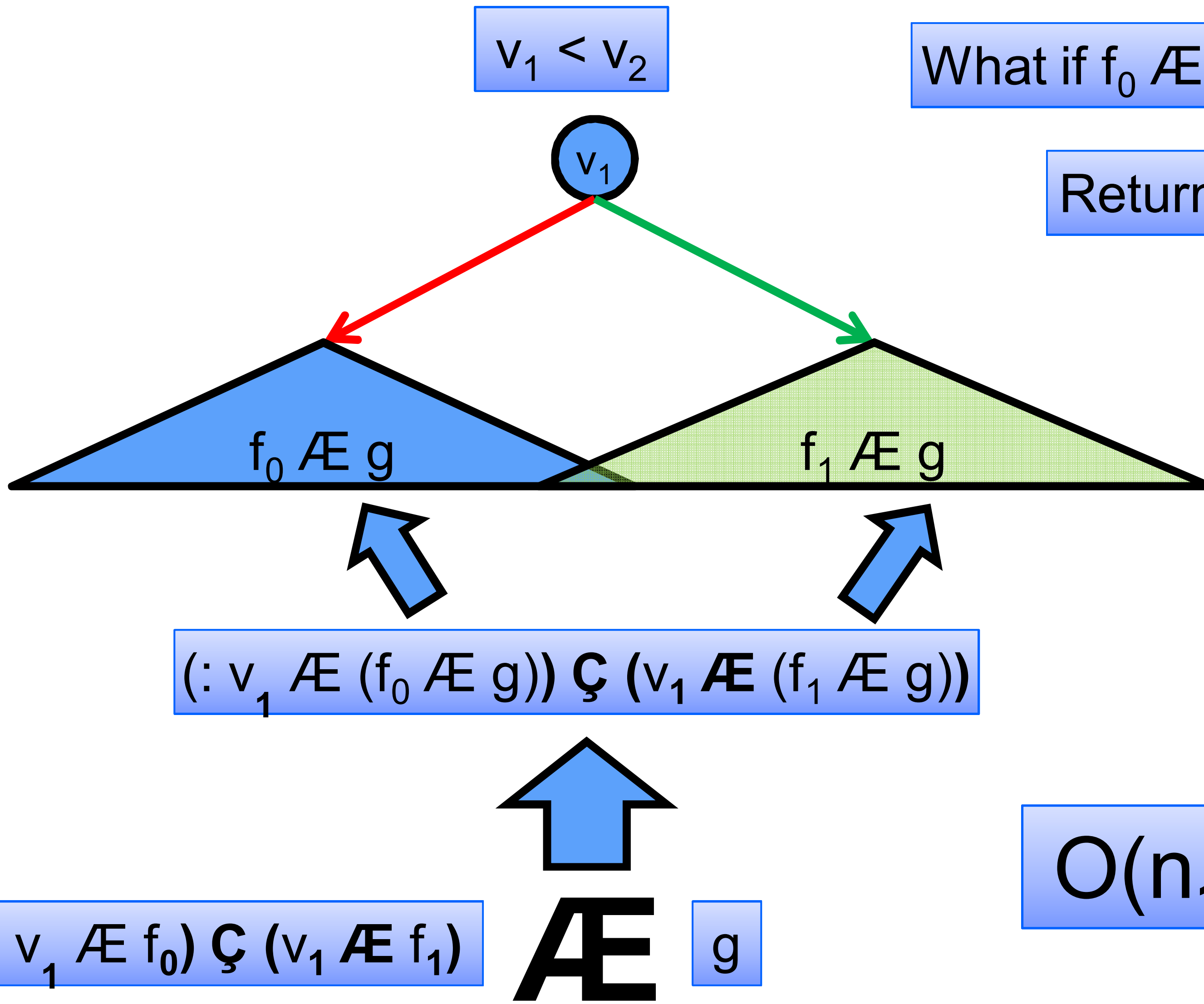
(: v_1 Æ (f_0 Æ g_0)) **Ç** (v_1 **Æ** (f_1 Æ g_1))

(: v_1 Æ f_0) **Ç** (v_1 **Æ** f_1) **Æ** (: v_1 Æ g_0) **Ç** (v_1 **Æ** g_1)

BDD Operations: And (Complex Case 2)



BDD Operations: And (Complex Case 2)



What if f_0 Æ g = f_1 Æ g ?

Return f_0 Æ g

O(n_1 £ n_2)

BDD Operations: And

```
BDD bddAnd (BDD f, BDD g)
  if (f == g || f == True) return g
  if (g == True) return f
  if (f == False || g == False) return False

  v = (var(f) < var(g)) ? var(f) : var(g)
  f0 = (v == var(f)) ? low(f) : f
  f1 = (v == var(f)) ? high(f) : f

  g0 = (v == var(g)) ? low (g) : g
  g1 = (v == var(g)) ? high (g) : g

  T = bddAnd (f1, g1); E = bddAnd (f0, g0)
  if (T == E) return T

  return mkUnique (v, T, E)
```

returns unique BDD for `ite(v,T,E)`

BDD Operations: Or

Or(f,g)

=

Not (And (Not(f**)**, Not(g**)))**

O(n_1 £ n_2)

BDD Operations: Exists

Exists("0",v) = ?

BDD Operations: Exists

Exists(“0”,v) = “0”

Exists(“1”,v) = ?

BDD Operations: Exists

Exists("0",v) = "0"

Exists("1",v) = "1"

Exists((: v Æ f) Ç (v Æ g) , v) = ?

BDD Operations: Exists

Exists("0",v) = "0"

Exists("1",v) = "1"

Exists((: v Æ f) Ç (v Æ g) , v) = Or(f,g)

Exists((: v' Æ f) Ç (v' Æ g) , v) = ?

BDD Operations: Exists

$O(n^2)$

Exists(“0”,v) = “0”

Exists(“1”,v) = “1”

Exists((: v Æ f) Ç (v Æ g) , v) = Or(f,g)

Exists((: v’ Æ f) Ç (v’ Æ g) , v) =

(: v’ Æ Exists(f,v)) Ç (v’ Æ Exists(g,v))

But f is SAT iff 9 V. f is not “0”. So why doesn’t this imply P = NP?

Because the BDD size changes!

BDD Applications

SAT is great if you are interested to know if a solution exists

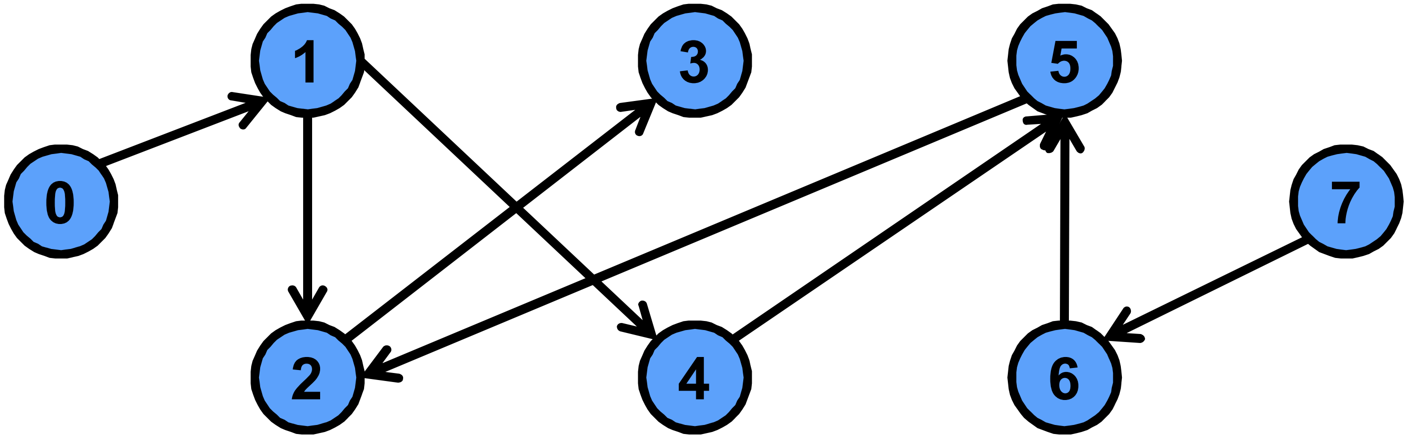
BDDs are great if you are interested in the set of all solutions

- How many solutions are there?
- How do you do this on a BDD?

BDDs are great for computing a fixed points

- Set of nodes reachable from a given node in a graph

Graph Reachability

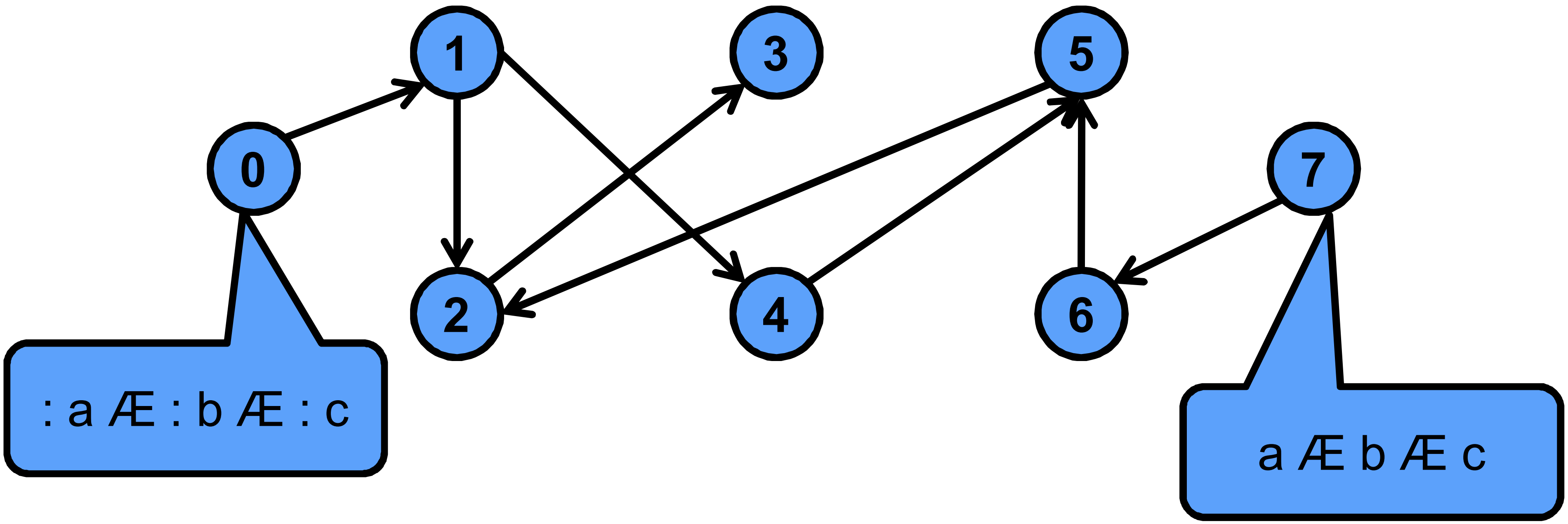


Which nodes are reachable from "7"?

{2,3,5,6,7}

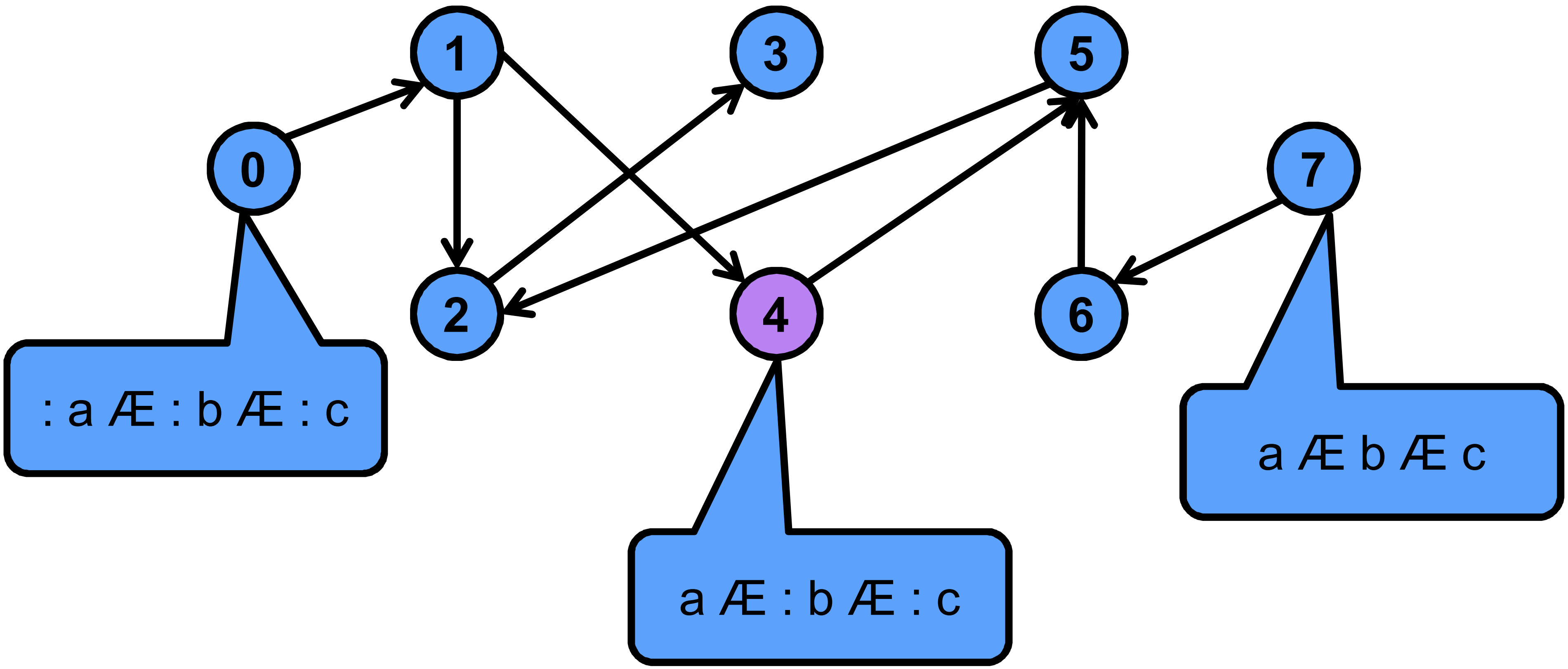
But what if the graph has trillions of nodes?

Graph Reachability



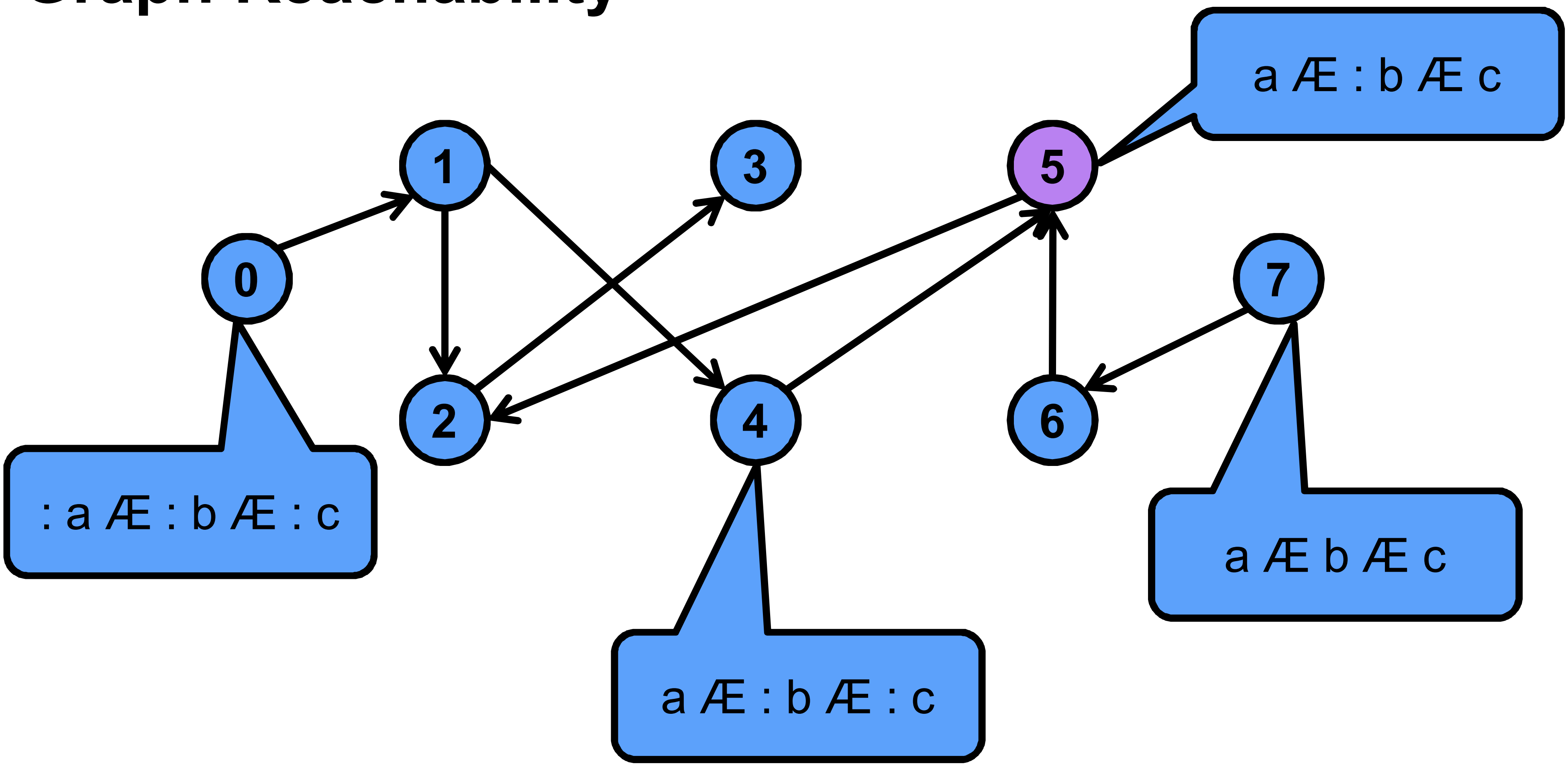
Use three Boolean variables (a,b,c) to encode each node?

Graph Reachability



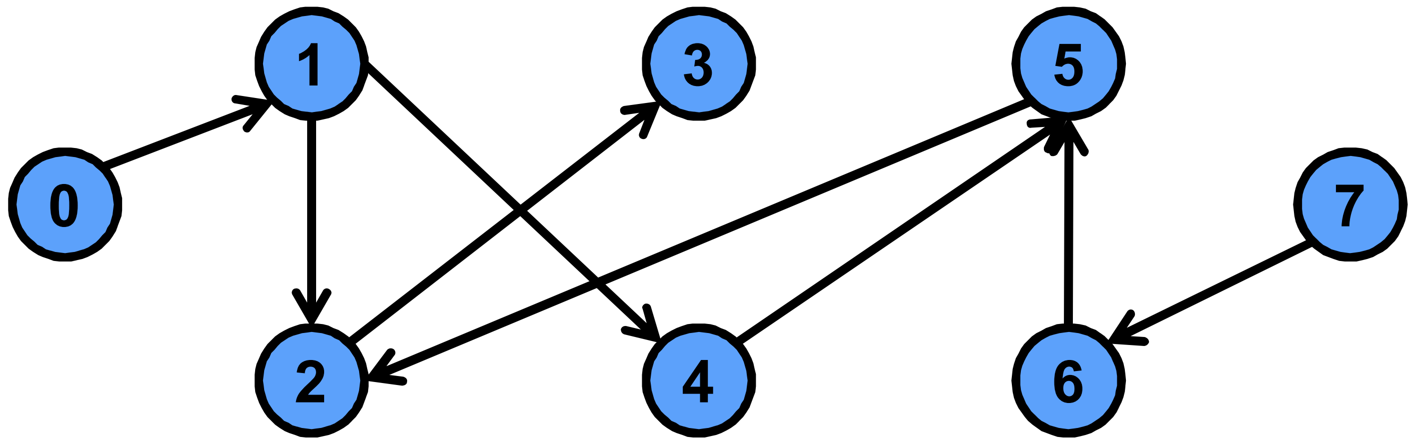
Use three Boolean variables (a,b,c) to encode each node?

Graph Reachability



Use three Boolean variables (a,b,c) to encode each node?

Graph Reachability

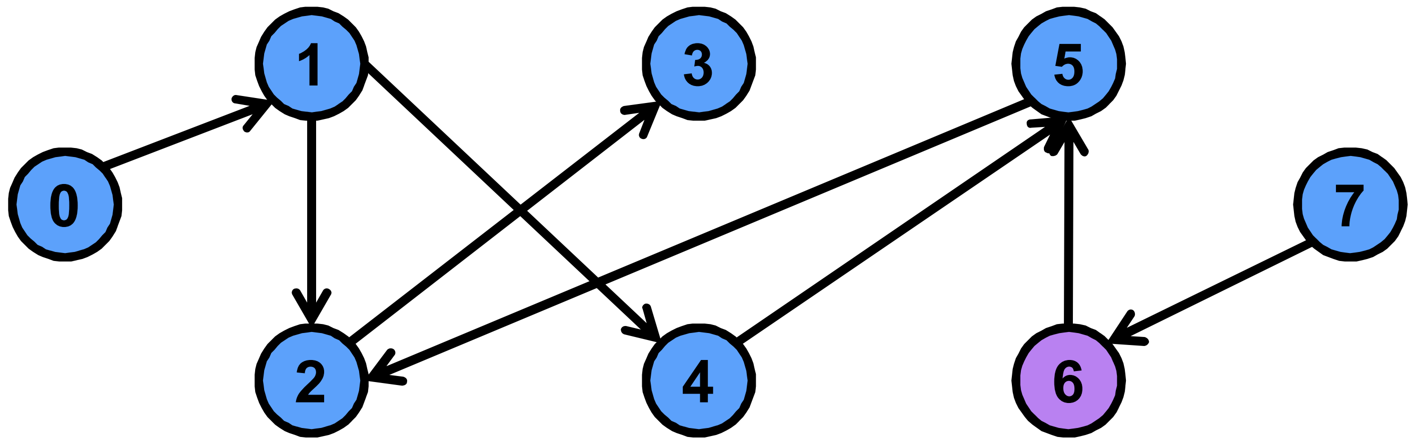


a Æ b Æ : c = ?

Key Idea 1: Every Boolean formula represents a set of nodes!

The nodes whose encodings satisfy the formula.

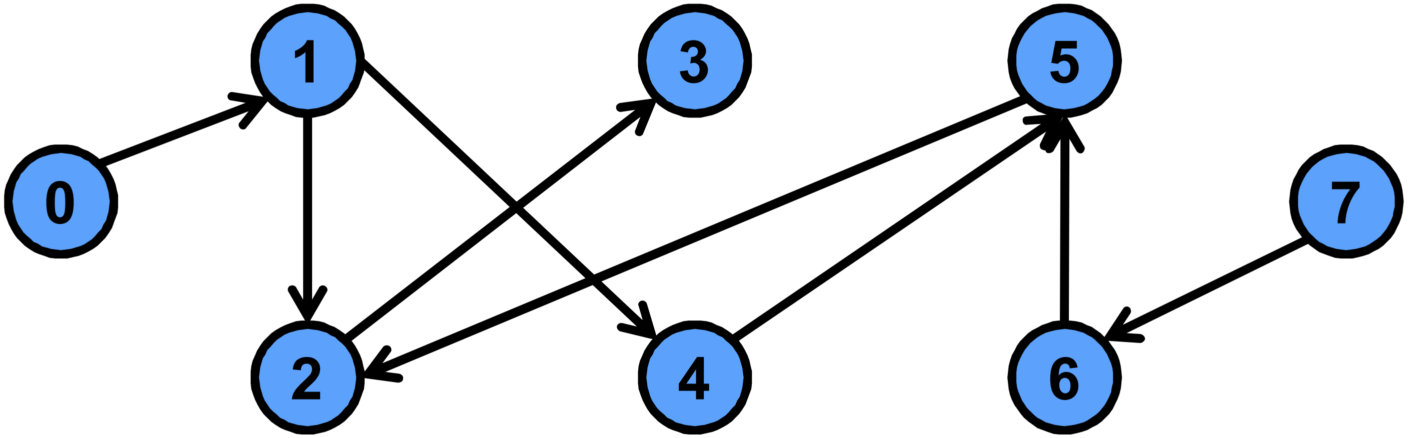
Graph Reachability



a Æ b Æ : c = {6}

Key Idea 1: Every Boolean formula represents a set of nodes!

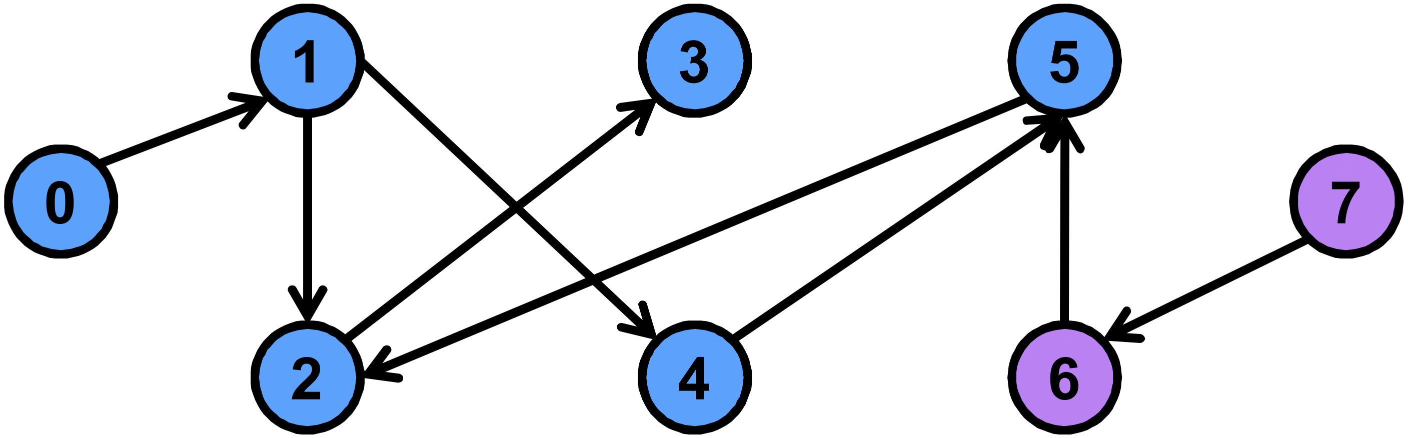
Graph Reachability



a Æ b = ?

Key Idea 1: Every Boolean formula represents a set of nodes!

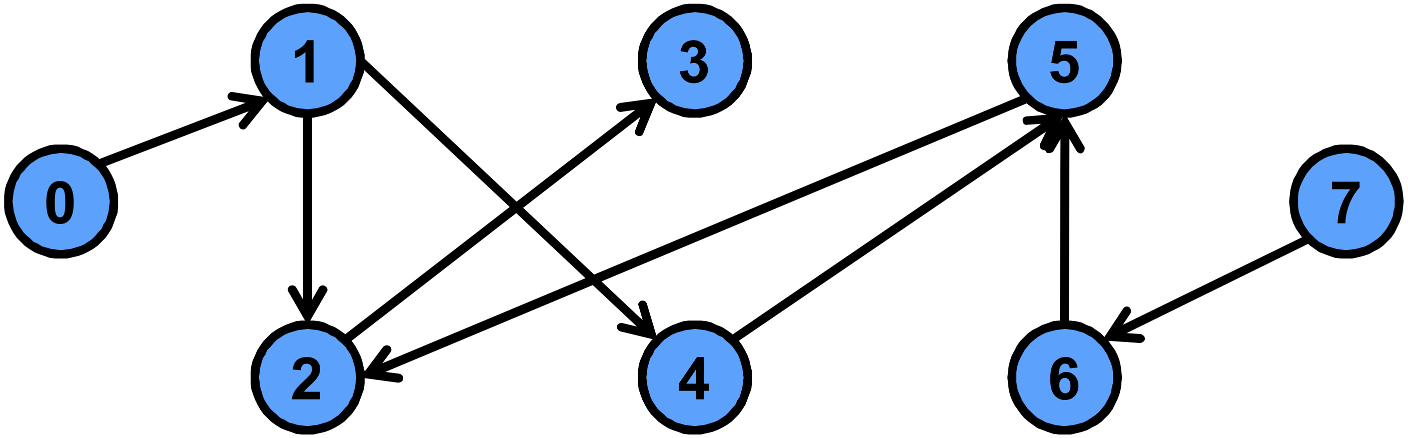
Graph Reachability



a Æ b = {6,7}

Key Idea 1: Every Boolean formula represents a set of nodes!

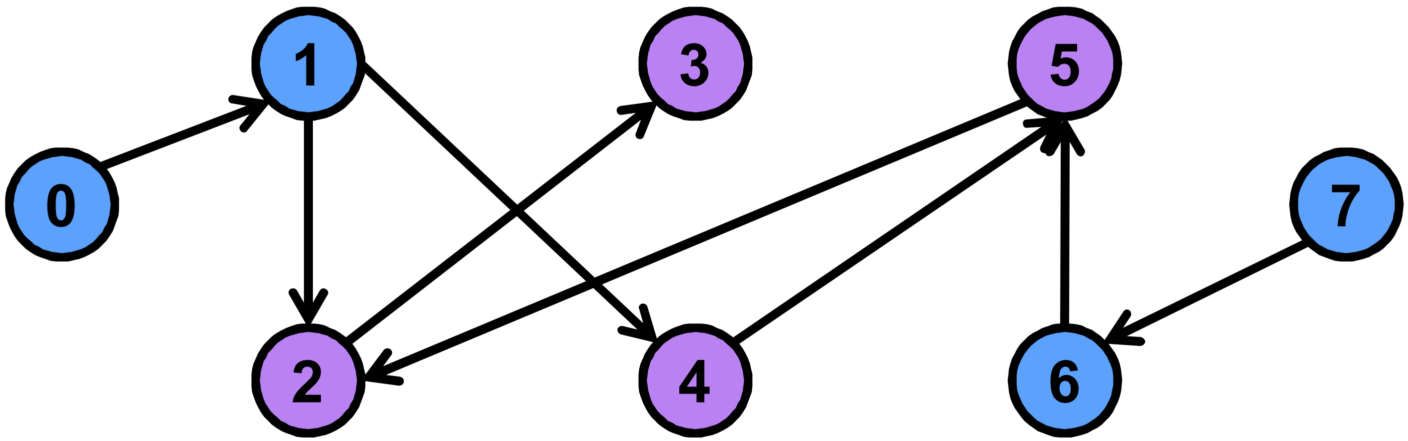
Graph Reachability



a xor b = ?

Key Idea 1: Every Boolean formula represents a set of nodes!

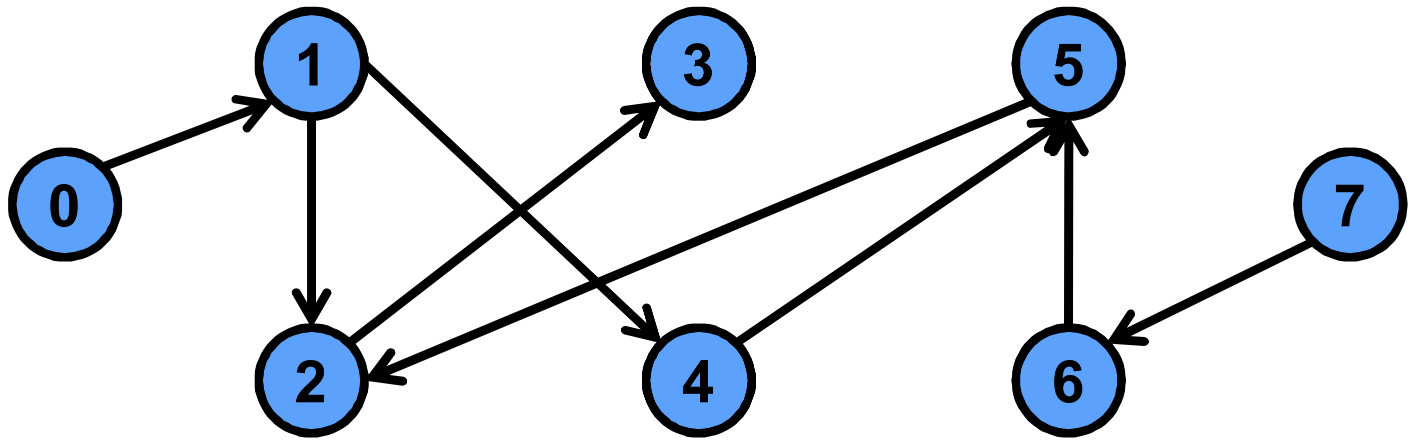
Graph Reachability



a xor b = {2,3,4,5}

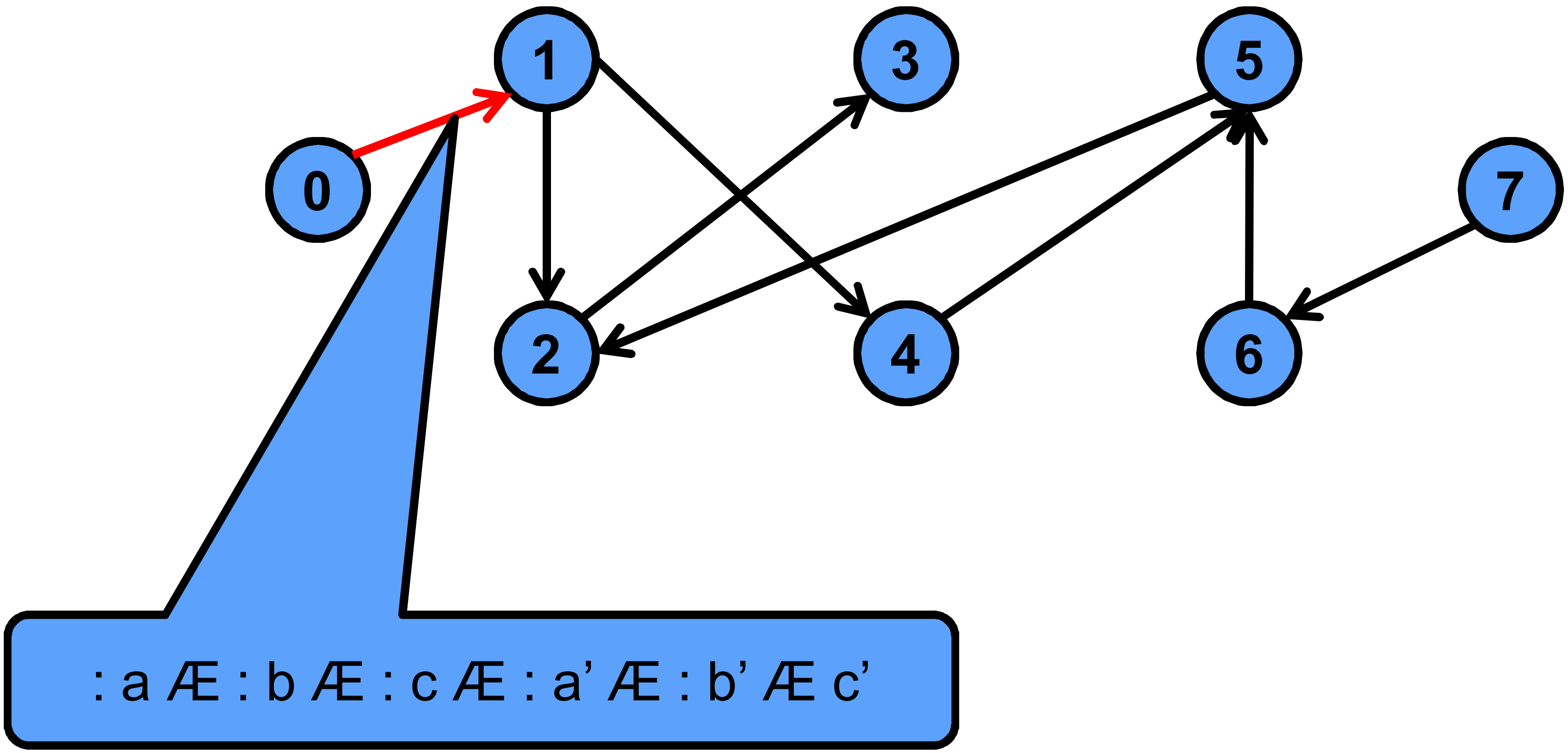
Key Idea 1: Every Boolean formula represents a set of nodes!

Graph Reachability



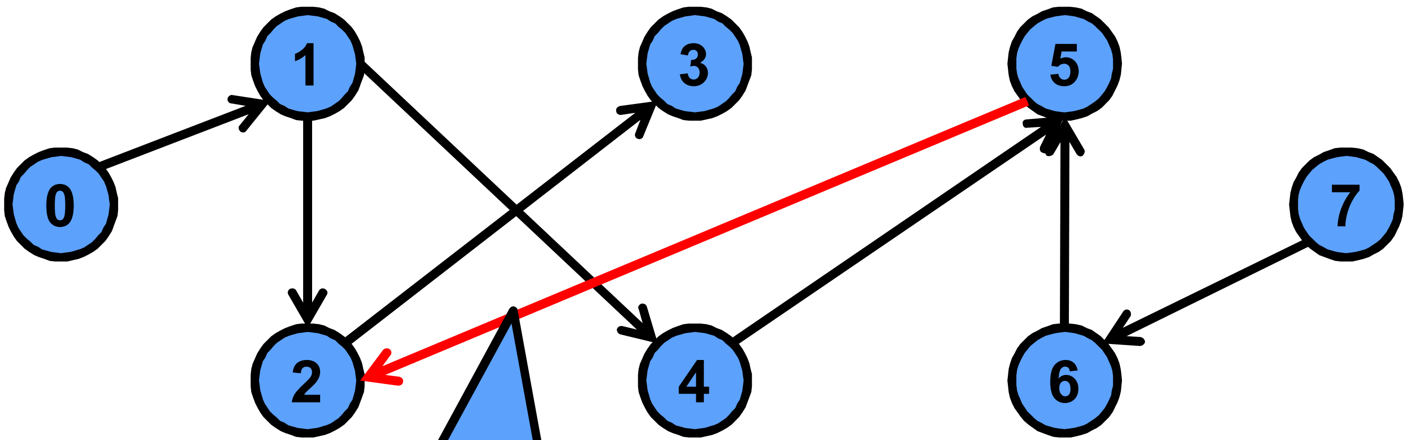
- Key Idea 2: Edges can also be represented by Boolean formulas
- An edge is just a pair of nodes
- Introduce three new variables: a’, b’, c’
- Formula © represents all pairs of nodes (n,n’) that satisfy © when n is encoded using (a,b,c) and n’ is encoded using (a’,b’,c’)

Graph Reachability



Key Idea 2: Edges can also be represented by Boolean formulas

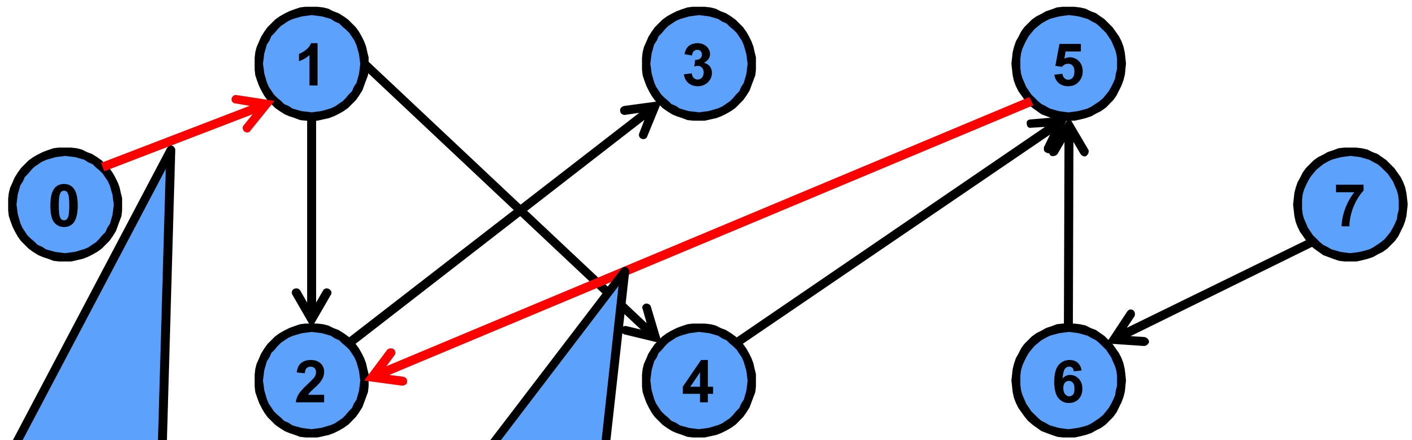
Graph Reachability



a Æ : b Æ c Æ : a’ Æ b’ Æ : c’

Key Idea 2: Edges can also be represented by Boolean formulas

Graph Reachability



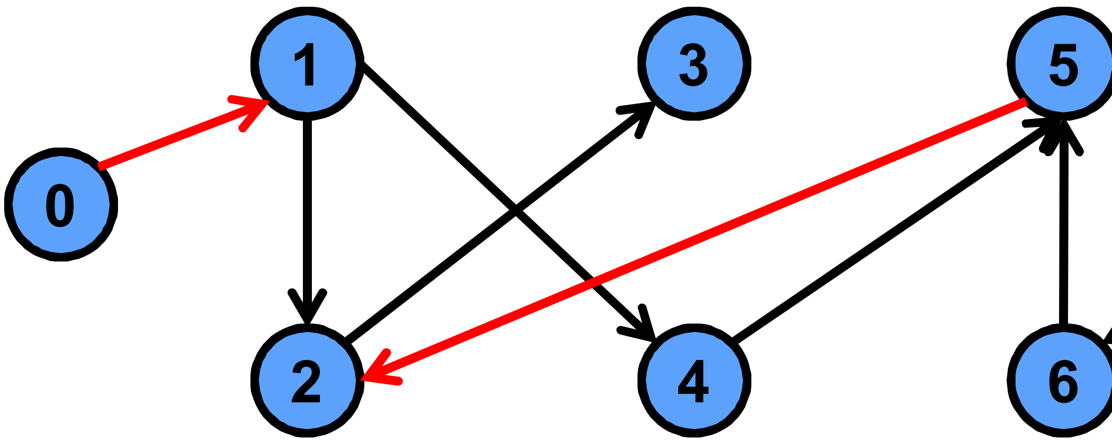
a Æ : b Æ c Æ : a’ Æ b’ Æ : c’

ç

: a Æ : b Æ : c Æ : a’ Æ : b’ Æ c’

Key Idea 2: Edges can also be represented by Boolean formulas

Graph Reachability



Variable renaming : replace a’ with a

Image(S,R) =

(9 a,b,c . (S Æ R)) [a \ a’, b \ b’, c \ c’]

Key Idea 3: Given the BDD for a set of nodes S, and the BDD for the set of all edges R, the BDD for all the nodes that are adjacent to S can be computed using the BDD operations

Graph Reachability Algorithm

```
S = BDD for initial set of nodes;
R = BDD for all the edges of the graph;

while (true) {
   I = Image(S,R); // compute adjacent nodes to S
   if (And(Not(S),I) == False) // no new nodes found
      break;
   S = Or(S,I); // add newly discovered nodes to result
}

return S;
```

Symbolic Model Checking. Has been done for graphs with 10^{20} nodes.