Induction, k-Induction, and Symbolic Model Checking

Automated Program Verification (APV)
Fall 2019

Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

Symbolic model checking

Model is represented symbolically using Boolean formulas

Model checking is performed on the symbolic representation **directly**

BDD-based

- Use specialized data structure, Binary Decision Diagrams, to represent and manipulate sets of states

SAT-based (most of this class)

- Represent sets of executions using Boolean formulas in Conjunctive Normal Form (CNF)
- Use efficient SAT(isfiability)-solvers for reasoning

SAT-based Model Checking

Bounded Model Checking

- Is there a counterexample of k-steps

Unbounded Model Checking

- Induction and *k*-Induction (k-IND)
- Interpolation Based Model Checking (IMC)
- Property Directed Reachability (IC3/PDR)

Mathematical Induction

To proof that a property P(n) holds for all natural numbers n

1. Show that P(0) is true

2. Show that P(k+1) is true for some natural number k, using an Inductive Hypothesis that P(k) is true

Example: Mathematical Induction

Show by induction that P(n) is true

$$0 + \cdots + n = \frac{n(n+1)}{2}$$

Base Case: P(0) is $0 = \frac{0(0+1)}{2}$

IH: Assume P(k), show P(k+1)

$$\begin{aligned} & 0 + \cdots + k + (k+1) \\ = \quad & \frac{k(k+1)}{2} + (k+1) \\ = \quad & \frac{k(k+1)+2(k+1)}{2} \\ = \quad & \frac{(k+1)((k+1)+1)}{2} \end{aligned}$$

Symbolic Safety and Reachability

A transition system P = (V, Init, Tr, Bad)

P is UNSAFE if and only if there exists a number N s.t.

$$Init(X_0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(X_i, X_{i+1}) \right) \wedge Bad(X_N) \not\Rightarrow \bot$$

P is SAFE if and only if there exists a safe inductive invariant Inv s.t.

$$Init \Rightarrow Inv$$

$$Inv(X) \wedge Tr(X, X') \Rightarrow Inv(X')$$

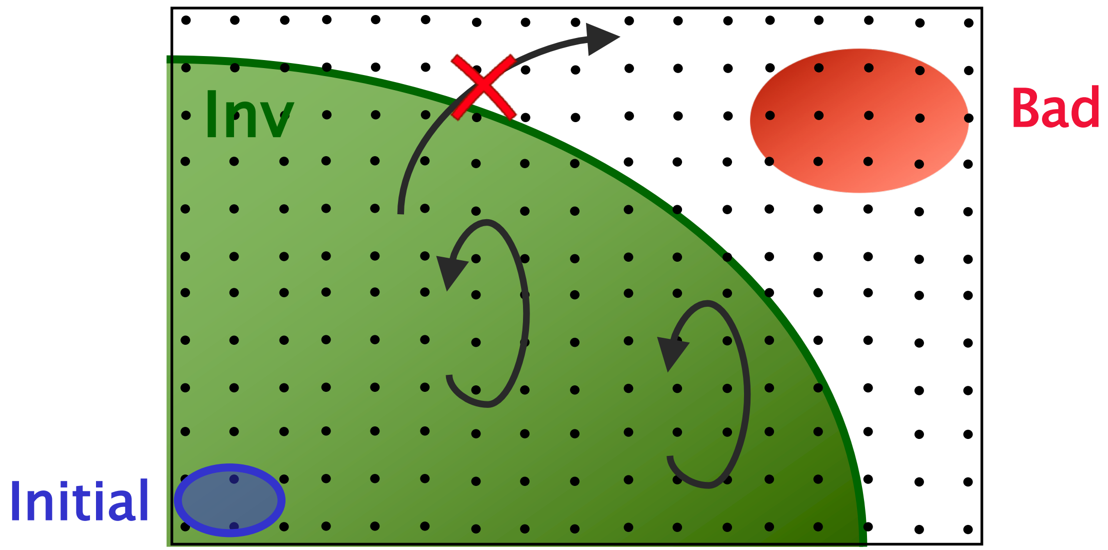
Inductive

$$Inv \Rightarrow \neg Bad$$

Safe

Inductive Invariants

System State Space

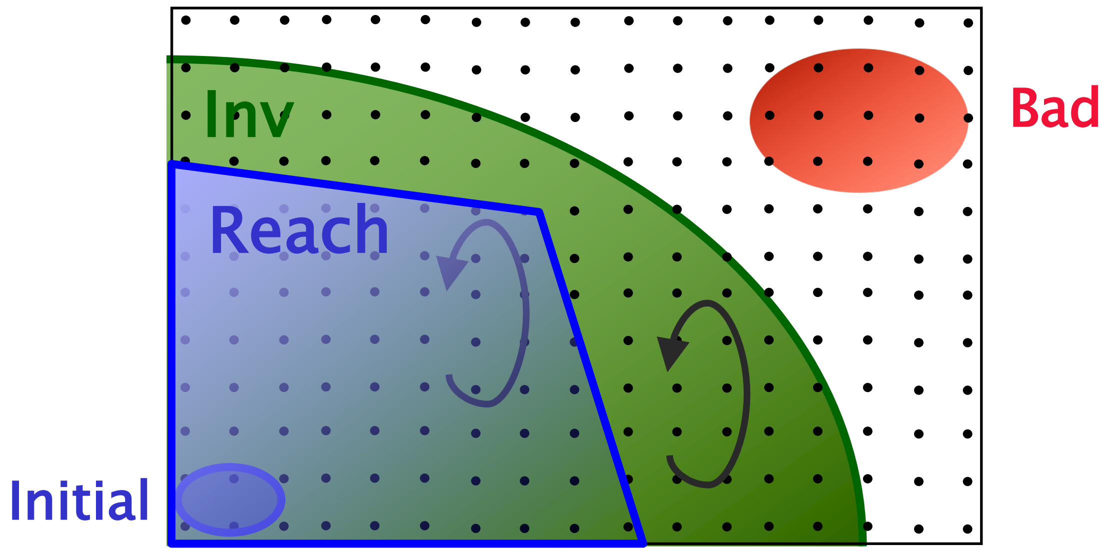


System S is safe iff there exists an inductive invariant Inv

- **Initiation** Initial ⊆ Inv
- **Safety** Inv ∩ Bad = ∅
- **Consecution** TR(Inv) ⊆ Inv

i.e., if s ∈ Inv and s⤳t then t ∈ Inv

Inductive Invariants



System S is safe iff there exists an inductive invariant Inv

- **Initiation** Initial ⊆ Inv
- **Safety** Inv ∩ Bad = ∅
- **Consecution** TR(Inv) ⊆ Inv

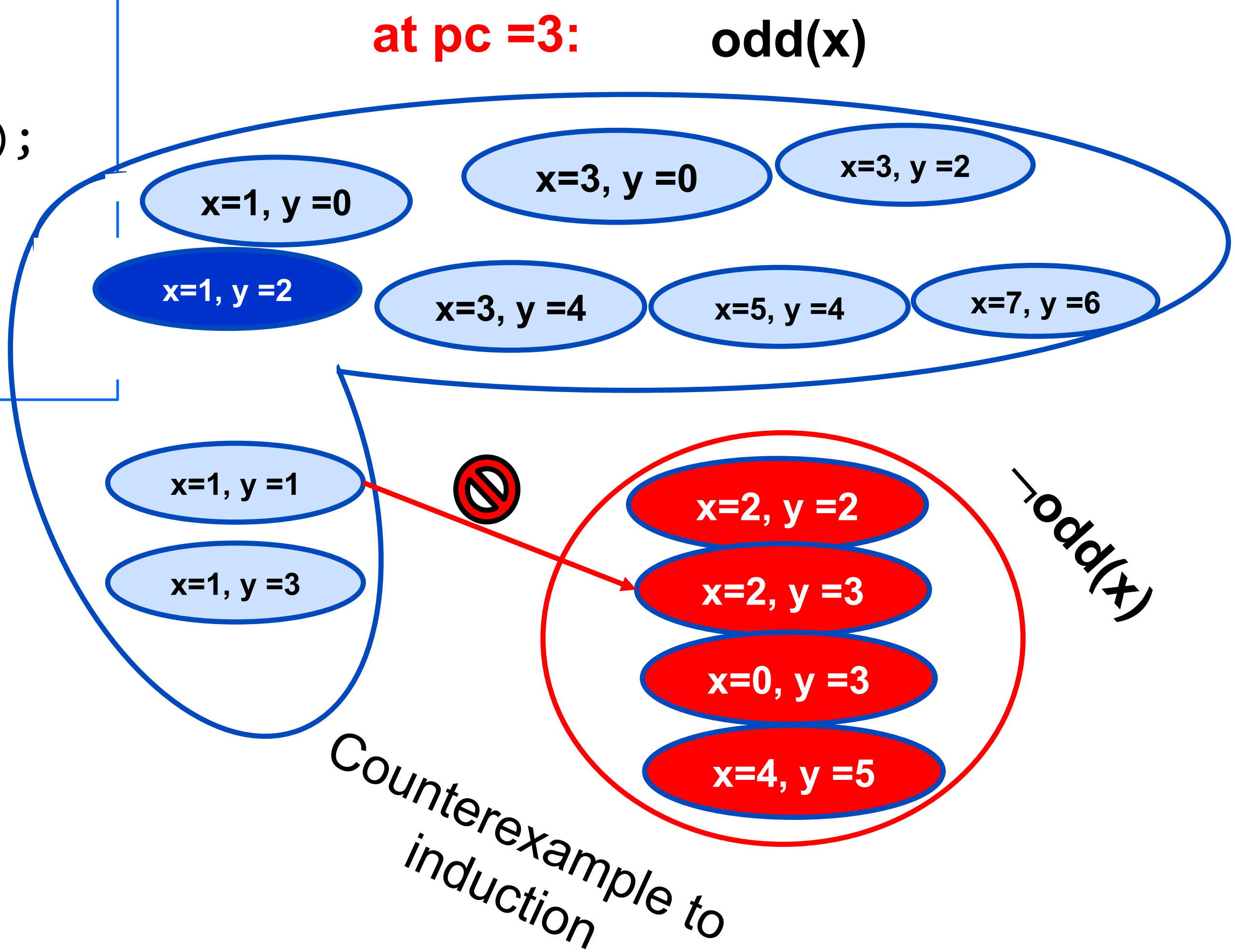
i.e., if s ∈ Inv and s⤳t then t ∈ Inv

System S is safe if Reach ∩ Bad = ∅

Induction: Simple Example

Is **pc=3 ⇒ odd(x)** an inductive invariant?

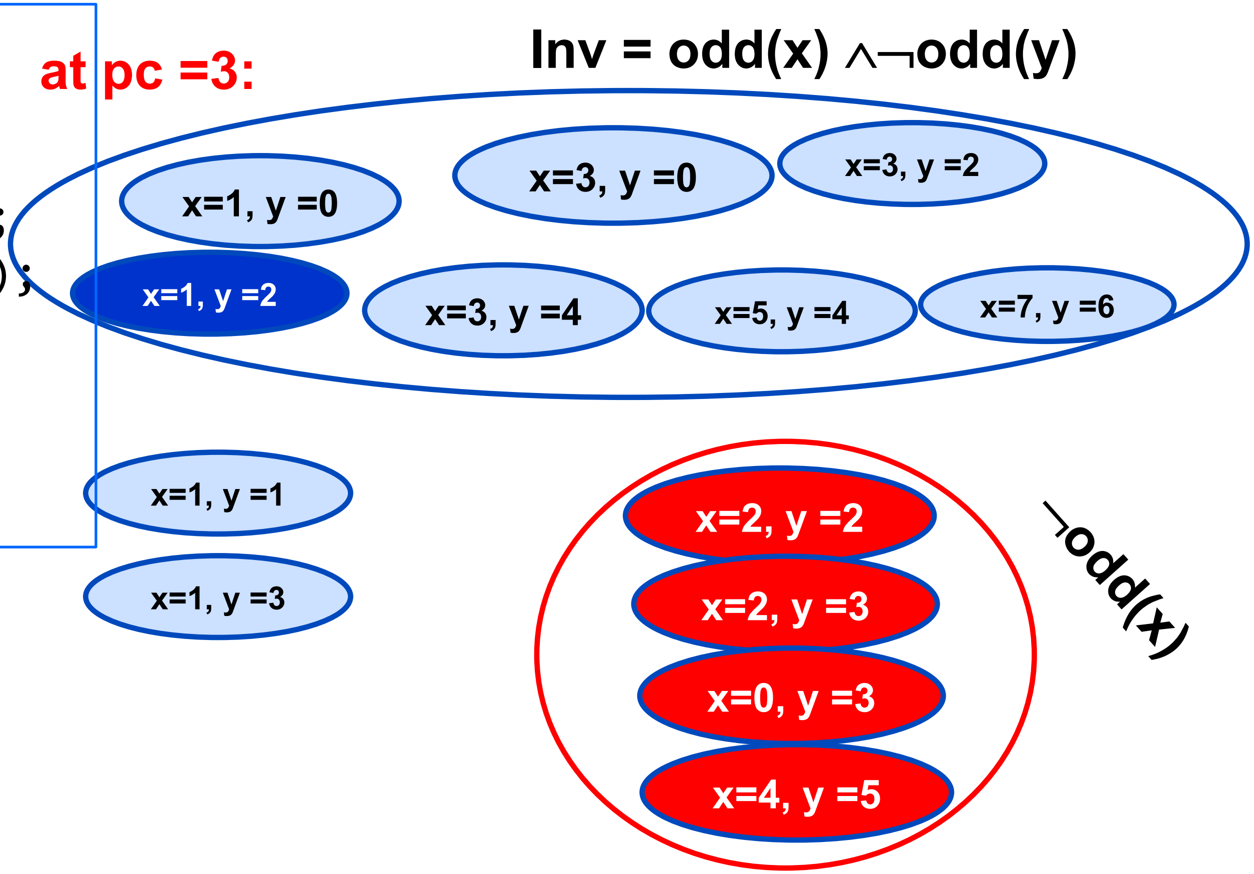
```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd(x);
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```



Inductive Invariants: Simple Example

Is **pc=3 ⇒ (odd(x) ∧ ¬odd(y))** an inductive invariant?

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd(x);
      assert !odd(y);
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```



Checking Invariance is reducible to SAT!

Inputs

- A transition system P = (V, Init, Tr, Bad)
- A formula I(V) over variables V

Decide whether I is a safe inductive invariant

- Use SAT to check that $Init \wedge \neg I$ is UNSAT
- Use SAT to check that $I(V) \wedge Tr(V, V') \wedge \neg I(V')$ is UNSAT
- Use SAT to check that $I \wedge Bad$ is UNSAT

If all checks are UNSAT, I(V) is a safe inductive invariant

If a check fails, interpretation depends on the failing check:

- Check 1: missing initial states
- Check 2: not closed under a step of transition relation
- Check 3: not safe (true invariant, but not good enough for property)

Complete SAT-based Model Checker

(Don't try this at home)

Inputs

- A transition system P = (V, Init, Tr, Bad)

For every propositional formula Cand(V) over variables V

- If Cand(V) is a safe inductive invariant, return True

If got here, return False

Is this algorithm sound?

Is this algorithm complete?

Is this algorithm efficient?

Maximal Inductive Subset

Let *L* be a set of formulas, *P=(V, Init, Tr, Bad)* a program

A subset *X* of *L* is a *maximal inductive subset* iff it is the largest subset of *X* such that

$$Init(u) \Rightarrow \bigwedge_{\ell \in X} \ell(u)$$

$$\bigwedge_{\ell \in X} \ell(u) \wedge Tr(u, v) \Rightarrow \bigwedge_{\ell \in X} \ell(v)$$

A Maximal Inductive Subset is unique

- inductive invariants are closed under conjunction

Cormac Flanagan, K. Rustan M. Leino¬ Houdini, an Annotation Assistant for ESC/Java. FME 2001¬ 500-517

Minimal Unsatisfiable Subset

Let φ be a formula and $A = \{a_1, \ldots, a_n\}$ be atomic propositions occurring negatively in φ

Assume $\varphi \wedge a_1 \wedge \cdots \wedge a_n$ is UNSAT

A *minimal unsatisfiable subset (MUS)* of φ is the smallest subset $X \subseteq A$ such that $\varphi \wedge X$ is UNSAT

There are efficient algorithms for computing MUS (a.k.a. UNSAT core) for propositional formulas

Solving MIS via MUS

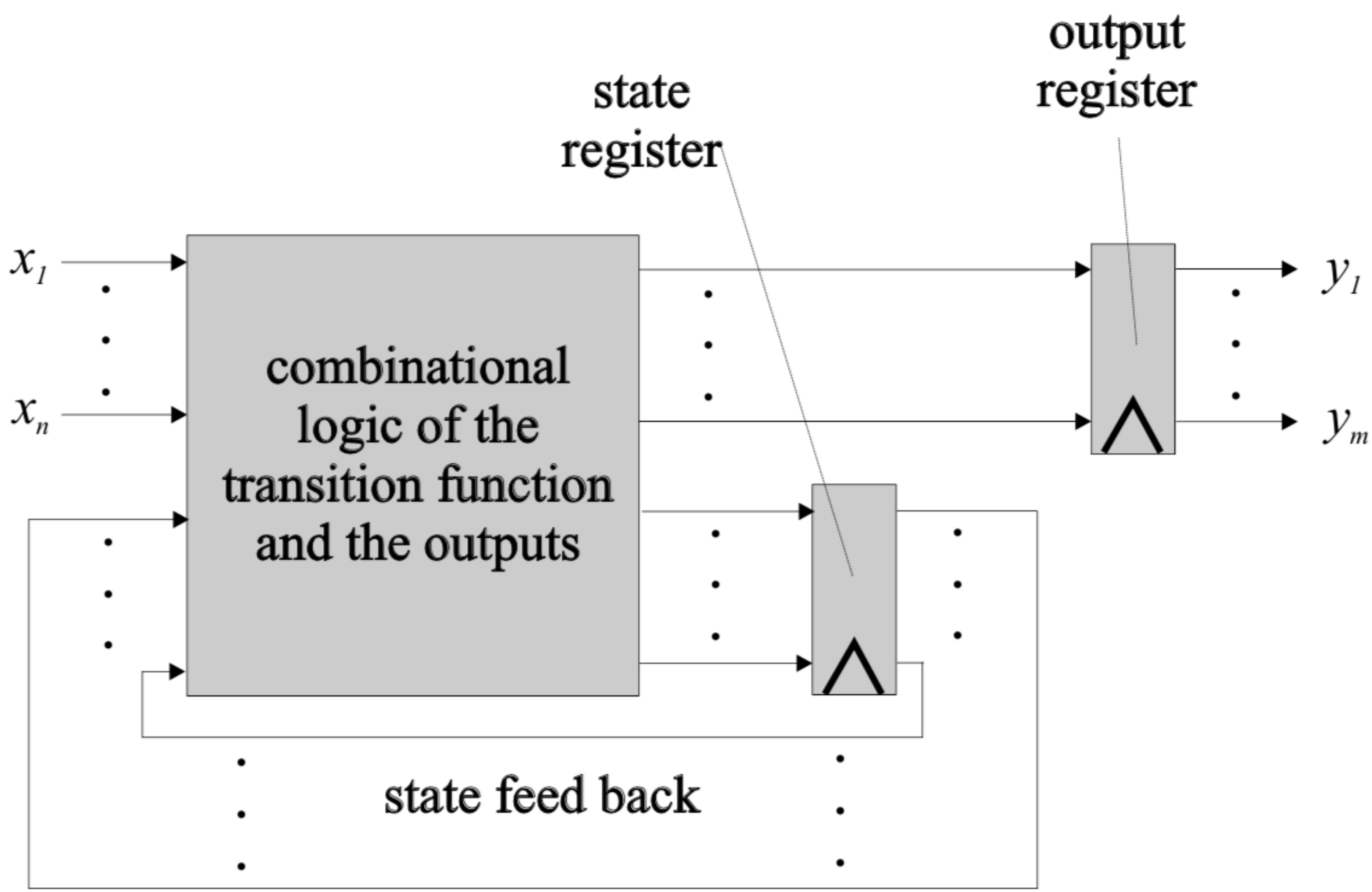
Reduce MIS to multiple calls to MUS

fresh propositional variables

Input : $\mathcal{L}, Tr$ — a set of lemmas and the transition relation (in BV)
Output: $\mathcal{L}' \subseteq \mathcal{L}$ the MIS of $\mathcal{L}$ relative to Tr

$\varphi \leftarrow (\bigwedge_{L_i \in \mathcal{L}} (pre_i \Rightarrow L_i(u))) \land Tr(u,v) \land (\bigvee_{L_i \in \mathcal{L}} (post_i \land \neg L_i(v)))$
`Sat_Add(B2P(`φ`))` — called once
$\mathcal{L}' \leftarrow \mathcal{L}$
forever do
    `Sat_Checkpoint()` — incremental SAT
    `Sat_Add(`pre_i`)` for all $L_i \in \mathcal{L}'$
    $C =$ `MUS`$(\{\neg post_i \mid L_i \in \mathcal{L}'\})$ — SAT MUS
    if $|C| = |\mathcal{L}'|$ **then return** $\mathcal{L}'$
    $\mathcal{L}' \leftarrow \{L_i \mid (\neg post_i) \in C\}$
    `Sat_Rollback()` — incremental SAT
end

A Synchronous Mealy Machine



Molitor and Mohnke. Equivalence Checking of Digital Circuits. 2004

UNIVERSITY OF WATERLOO

Terminology for Sequential Synthesis

The ***set of reachable states*** is the set of all possible valuations of the registers after arbitrary long execution from the initial state

Combinational synthesis – changing the combinational logic of the circuit without knowledge of reachable states

Sequential synthesis – modifies the circuit so that its behavior is preserved in the reachable states, but arbitrary changes are allowed on the unreachable states

Sequentially equivalent nodes – nodes having the same or opposite polarity in all reachable states

UNIVERSITY OF WATERLOO

https://people.eecs.berkeley.edu/~alanmi/publications/2007/tech07_pss.pdf

AIG: And-Inverter-Graph

A data structure for representing and manipulating arbitrary propositional formulas

A graph with 3 kinds of nodes

- input: one output, correspond to variables
- output: one input, correspond to functions, outputs
- AND: two (or more) inputs, one outputs, correspond to AND

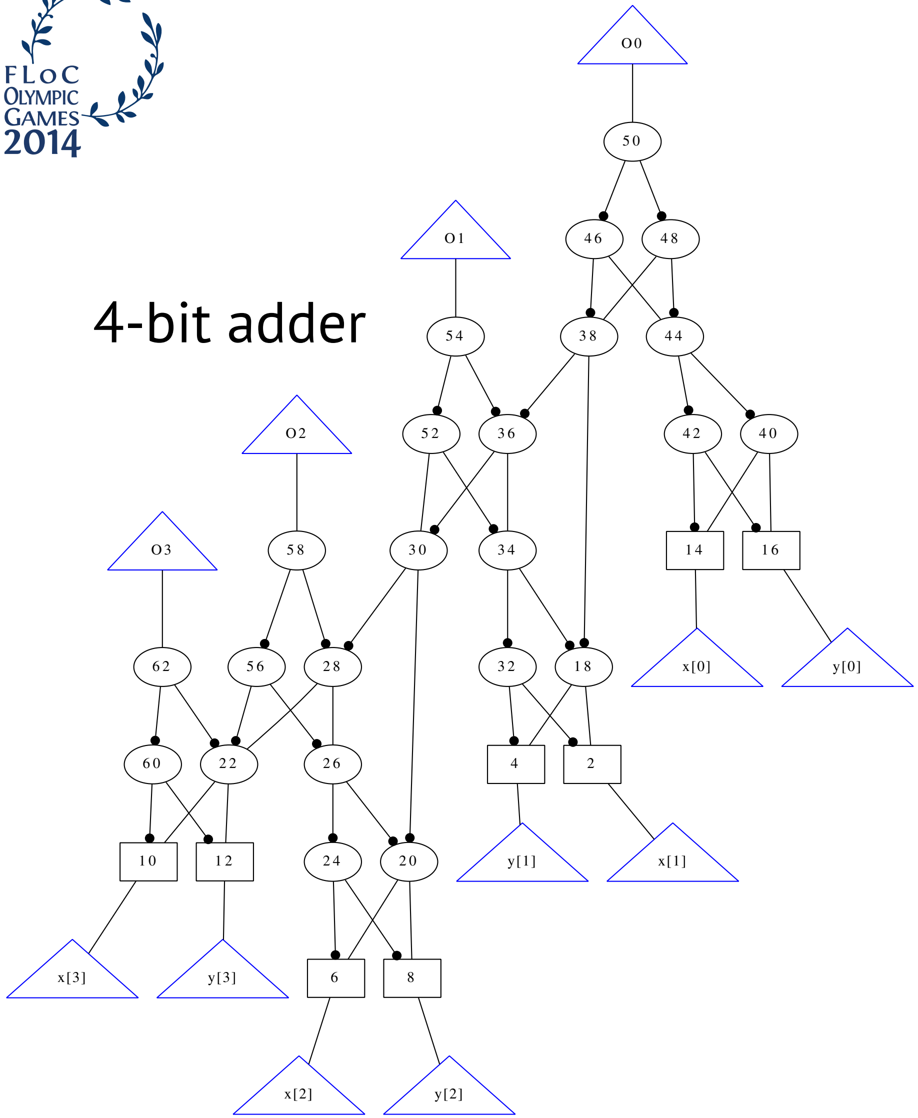
An input/output of any node can be negated

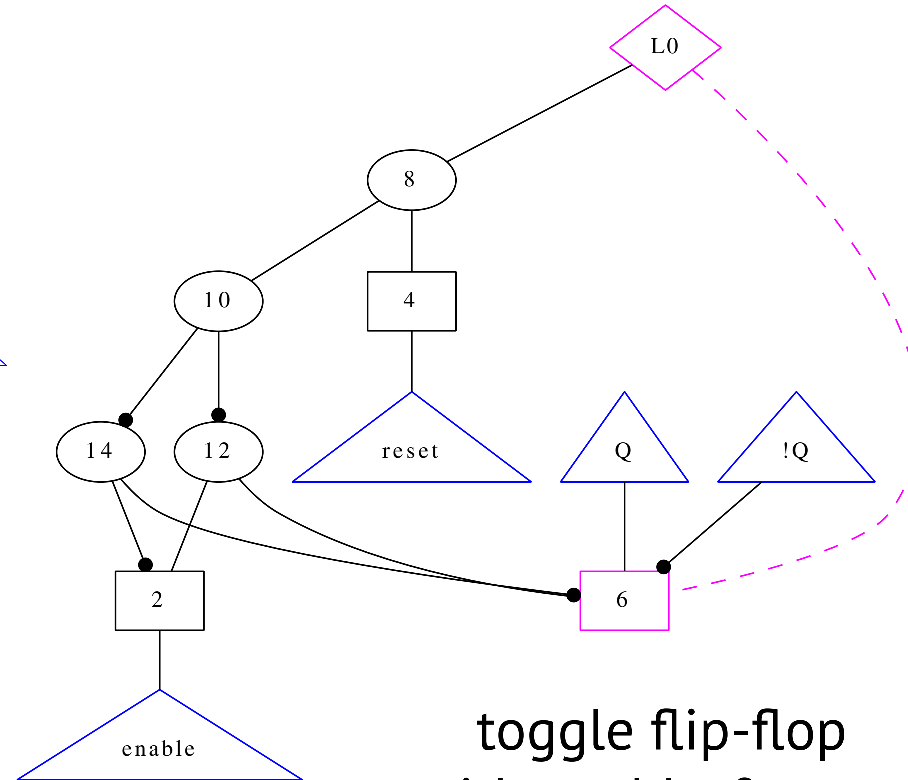
Hash-Cons

- AND nodes are kept in a hash table keyed on their children
- only one node is created for any syntactic function

AIGER

4-bit adder





toggle flip-flop
with enable & reset

http://fmv.jku.at/aiger/

Latch Correspondence Problem

Definition 10.1 (latch permutation problem) *Given two sequential circuits $F^{(1)}, F^{(2)} \in \mathcal{F}_{n,m,k}$, the latch permutation equivalence problem which is also referred as latch correspondence problem is the decision problem as to whether a correspondence π between the latches of $F^{(1)}$ and $F^{(2)}$ exists, such that the two synchronous sequential circuits $F^{(1)}$ and $F^{(2)}$ have their combinational parts functionally equivalent using this correspondence. More formally, the problem is to find a permutation $\pi \in \mathcal{P}er(\mathbb{N}_k)$ such that for all $j \in \mathbb{N}_m$*

$$\lambda_j^{(1)}(x_1, \ldots, x_n, u_{\pi(1)}^{(1)}, \ldots, u_{\pi(k)}^{(1)}) = \lambda_j^{(2)}(x_1, \ldots, x_n, u_1^{(2)}, \ldots, u_k^{(2)})$$

and for all $j \in \mathbb{N}_k$

$$\delta_{\pi(j)}^{(1)}(x_1, \ldots, x_n, u_{\pi(1)}^{(1)}, \ldots, u_{\pi(k)}^{(1)}) = \delta_j^{(2)}(x_1, \ldots, x_n, u_1^{(2)}, \ldots, u_k^{(2)})$$

hold. (For the notations, we refer to Chapter 8 Section 1.)

UNIVERSITY OF WATERLOO

Molitor and Mohnke. Equivalence Checking of Digital Circuits. 2004

Solving Latch Correspondence by MIS

Simulate the circuit with random inputs

Identify candidate equivalence classes

- latches H and K are candidates if in every simulation either
  - $H = K$ or $H = \neg K$

Refine candidate equivalences using BMC

- for every candidate H=K, use BMC to find a (short) counterexample

For all remaining candidates, compute Maximal Inductive Subset

- each call to SAT removes at least one candidate
- converges in linear time in the number of candidates

K-induction

Sheeran, Singh, Stålmarck Checking Safety Properties Using Induction and a SAT-Solver. FMCAD 2000

Induction

$$\frac{P(s_0) \qquad \forall i \,.\, P(s_i) \Rightarrow P(s_{i+1})}{\forall i \,.\, P(s_i)}$$

k-step Induction

$$\frac{P(s_{0..k-1}) \qquad \forall i \,.\, P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i \,.\, P(s_i)}$$

UNIVERSITY OF WATERLOO

http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf

2-Induction: Simple Example

Is pc=3 -> odd(x) 2-inductive invariant?

✓

Program

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd(x);
   4:  x := x + y;
   5:  y := y + 2
}
6:
```

2-Base

```
x := 1;
y := 2;
assert odd(x)
x := x + y;
y := y + 2;
assert odd(x)
```

2-IND

```
assume odd(x)
x := x + y;
y := y + 2;
assume odd(x)
x := x + y;
y := y + 2;
assert odd(x)
```

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

Induction and Strong Induction

Induction Principle

$Init(v_0) \rightarrow Inv(v_0)$

$Inv(v_0) \wedge Tr \rightarrow Inv(v_1)$

$\neg Bad(v)$

$Tr[Inv]^1$
$\rightarrow Inv(v_1)$

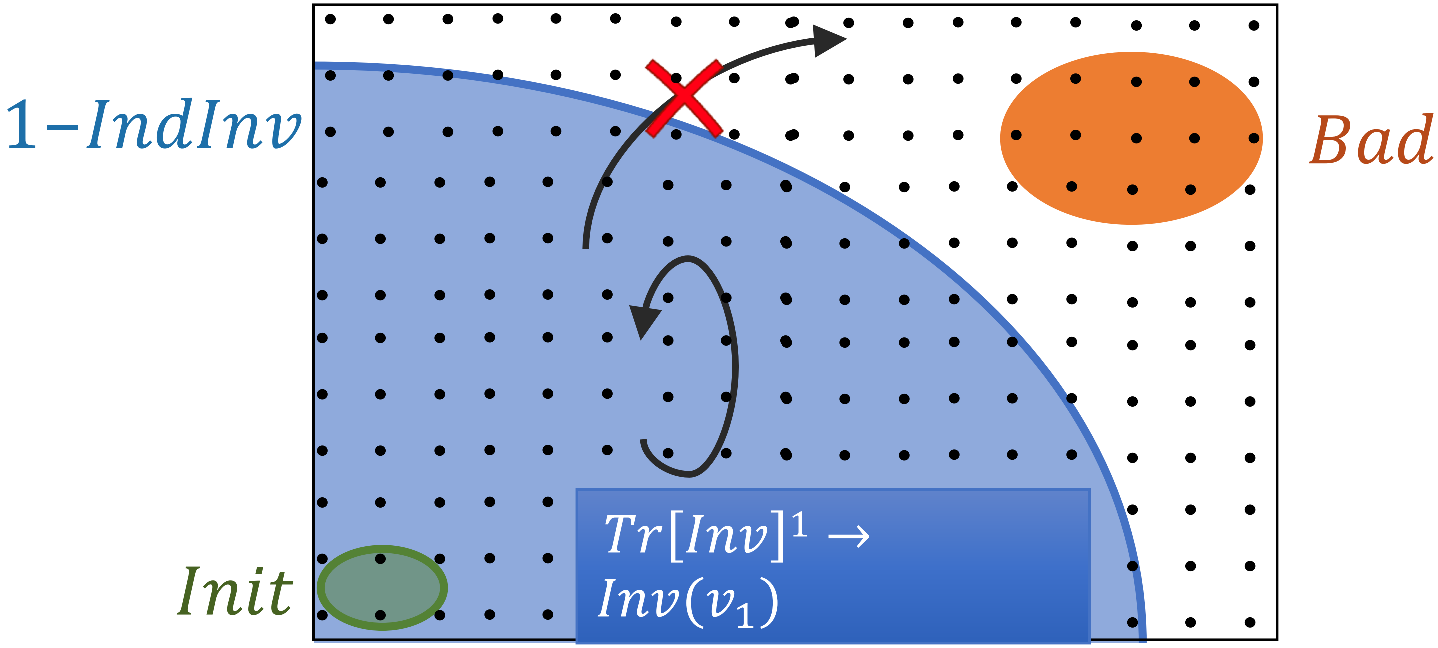
Strong Induction Principle

$Inv(v_0) \wedge Tr \wedge Inv(v_1) \wedge Tr \wedge \cdots \wedge Inv(v_{k-1}) \wedge Tr \rightarrow Inv(v_k)$

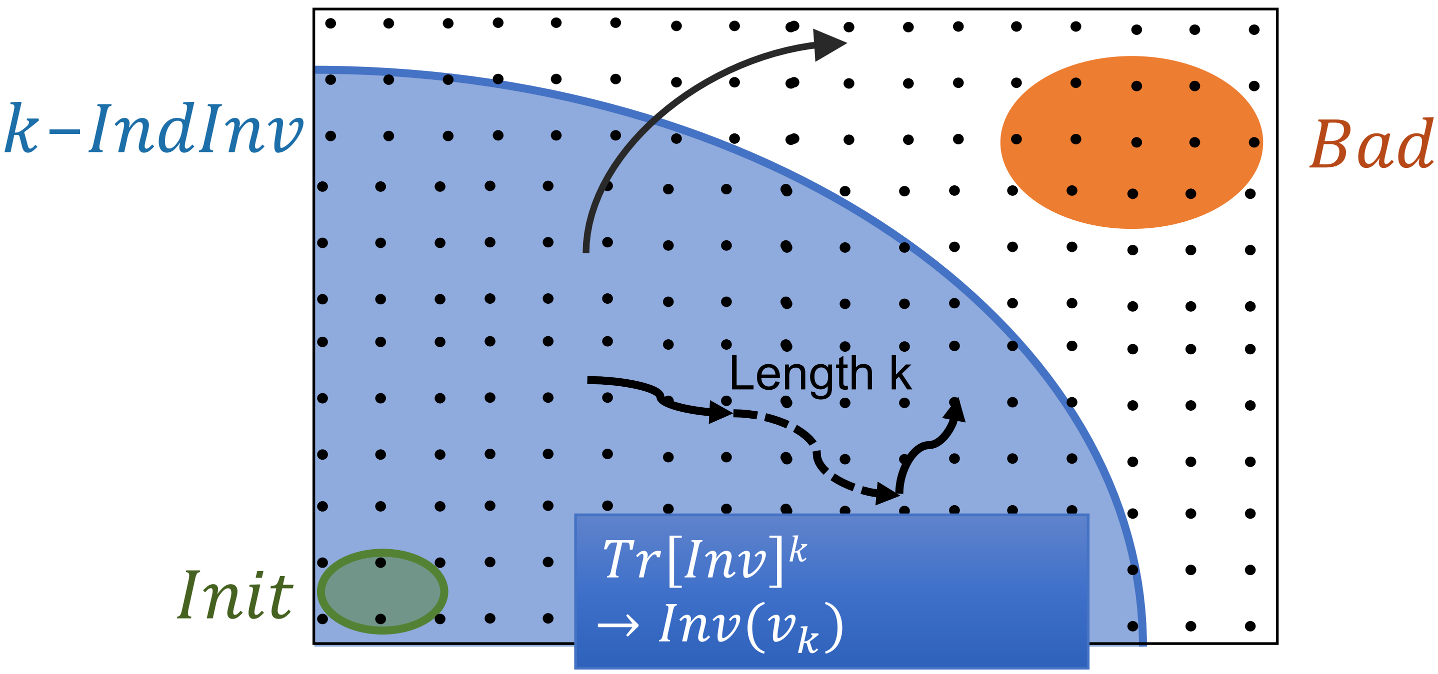
$Init($

$Tr[Inv]^k \rightarrow Inv(v_k)$

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering



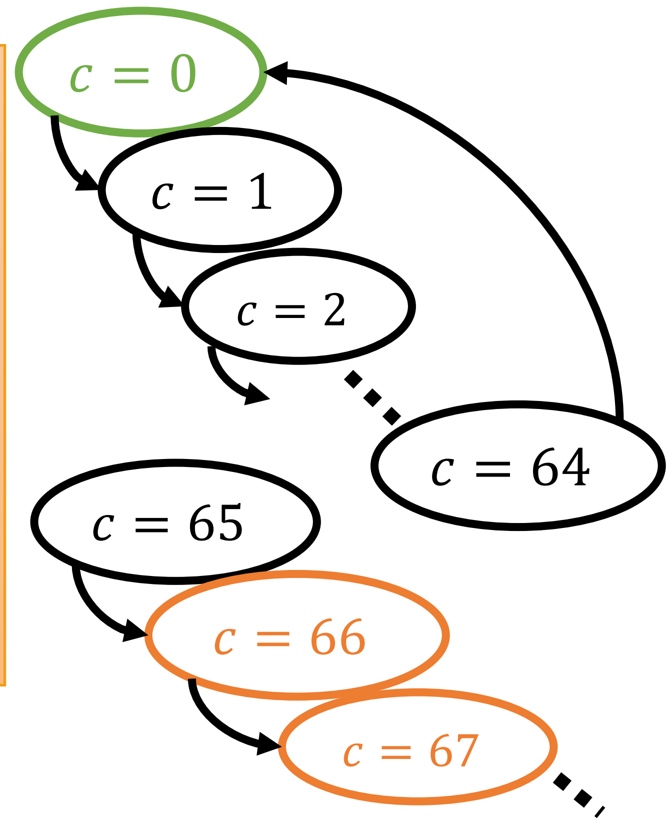
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering

Example circuit in Verilog

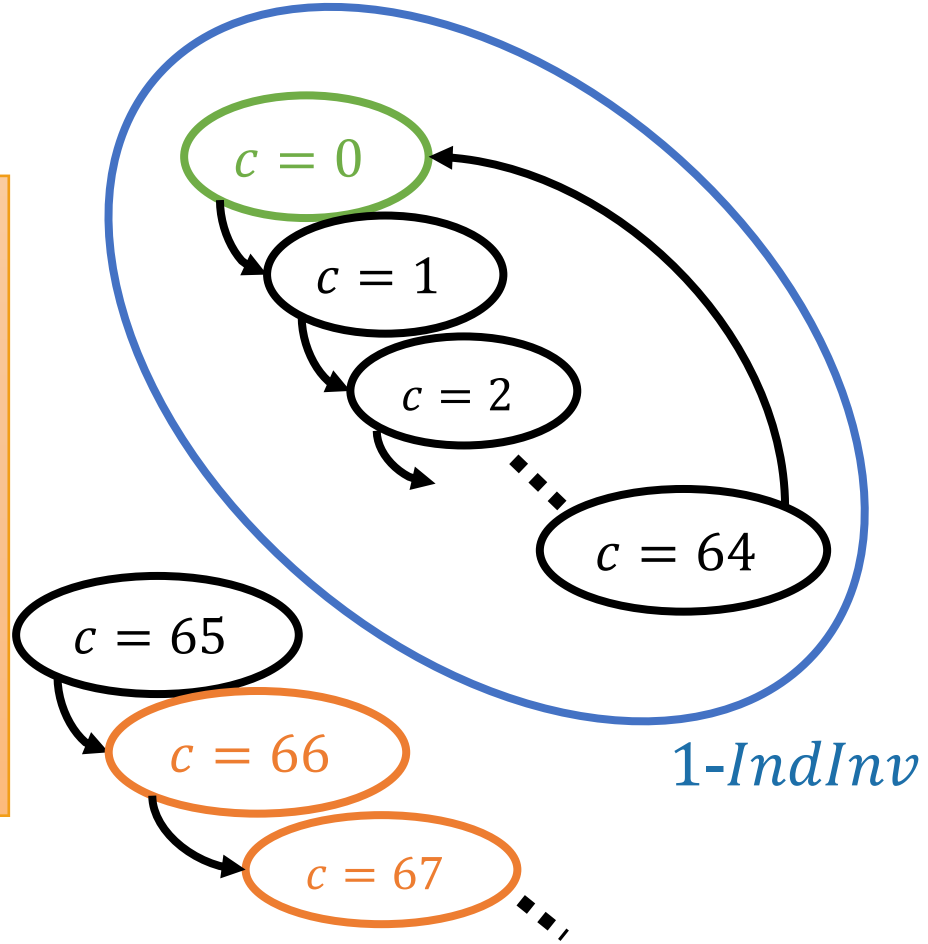
```
reg [7:0] c = 0;
always
if(c == 64)
     c = 0;
else
     c = c + 1;
end
assert property(c < 66);
```



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering

1-Inductive Invariant

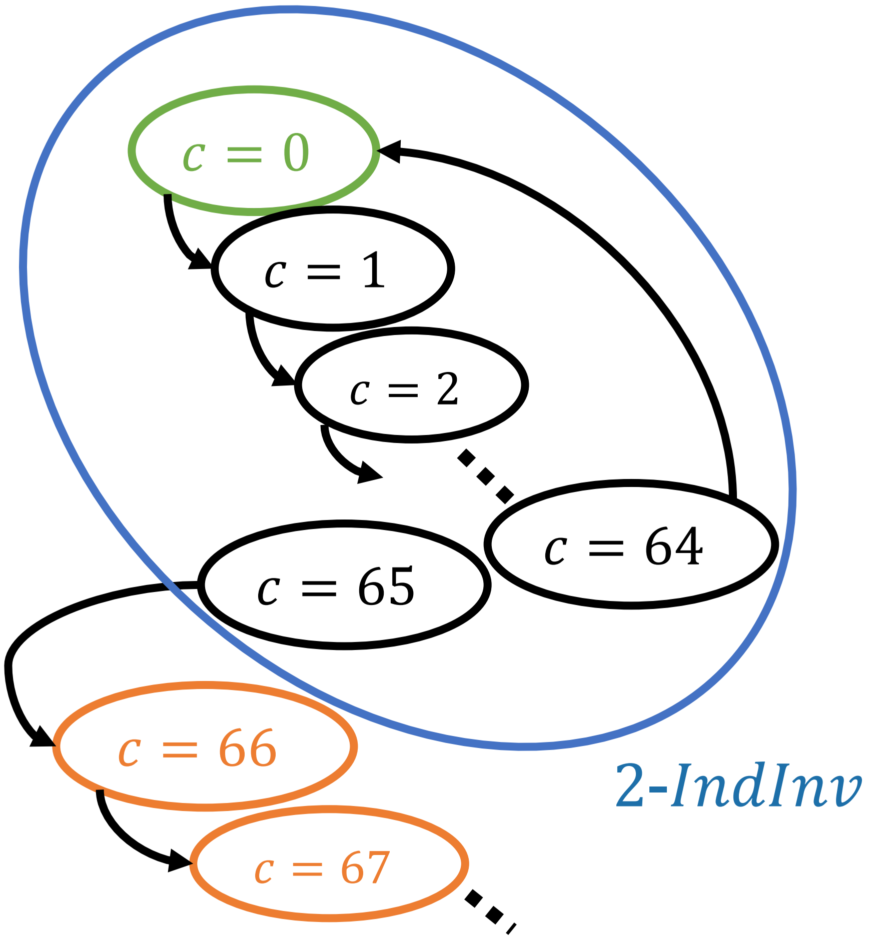
```
reg [7:0] c = 0;
always
if(c == 64)
      c = 0;
else
      c = c + 1;
end
assert property(c < 66);
```



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering

2-Inductive Invariant

```
reg [7:0] c = 0;
always
if(c == 64)
      c = 0;
else
      c = c + 1;
end
assert property(c < 66);
```



K-induction with a SAT solver (IND)

Recall:

$U_k = T^{<0>} \wedge T^{<1>} \wedge \ldots \wedge T^{<k-1>}$

Two formulas to check:

- Base case:

$$I^{<0>} \wedge U_{k-1} \Rightarrow P^{<0>} \ldots P^{<k-1>}$$

- Induction step:

$$U_k \wedge P^{<0>} \ldots P^{<k-1>} \Rightarrow P^{<k>}$$

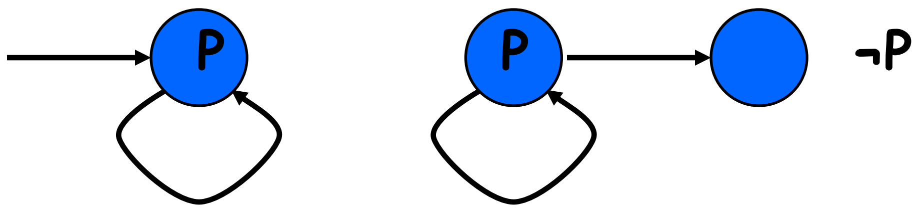
If both are valid, then *P* always holds.

If not, increase *k* and try again.

Simple path assumption

Unfortunately, k-induction is not complete.

- Some properties are not k-inductive for any k.



Simple path restriction:

- There is a path to ¬P iff there is a *simple* path to ¬P (path with no repeated states).

Induction over simple paths

Let simple($s_{0..k}$) be defined as:

- $\forall i,j$ in 0..k ¬ $(i \neq j) \Rightarrow s_i \neq s_j$

k-induction over simple paths:

$$\frac{\begin{array}{c} P(s_{0..k-1}) \\ \forall i \neg\ \text{simple}(s_{0..k}) \wedge P(s_{i..i+k-1}) \Rightarrow P(s_{i+k}) \end{array}}{\forall i \neg\ P(s_i)}$$

Must hold for k large enough, since a simple path cannot be unboundedly long. Length of longest simple path is called *recurrence diameter*.

UNIVERSITY OF WATERLOO

...with a SAT solver

For simple path restriction, let

$S_k = \forall t=0..k,\ u=t+1..k\text{:}\ \neg\forall v \text{ in } V\ \neg\ v_t = v_u$

(where V is the set of state variables).

Two formulas to check

- Base case

$$I^{<0>} \wedge U_{k-1} \Rightarrow P^{<0>}...P^{<k-1>}$$

- Induction step

$$S_k \wedge U_k \wedge P^{<0>}...P^{<k-1>} \Rightarrow P^{<k>}$$

If both are valid, then P always holds.

If not, increase k and try again.

UNIVERSITY OF WATERLOO

Termination

Termination condition

k is the length of the longest simple path of the form

$$P^*\neg P$$

This can be exponentially longer than the diameter.

- example
  - loadable mod 2^N counter where P is (count $\neq 2^N-1$)
  - diameter = 1
  - longest simple path = 2^N

Useful special cases

- P is a tautology (k=0)
- P is inductive invariant (k=1)

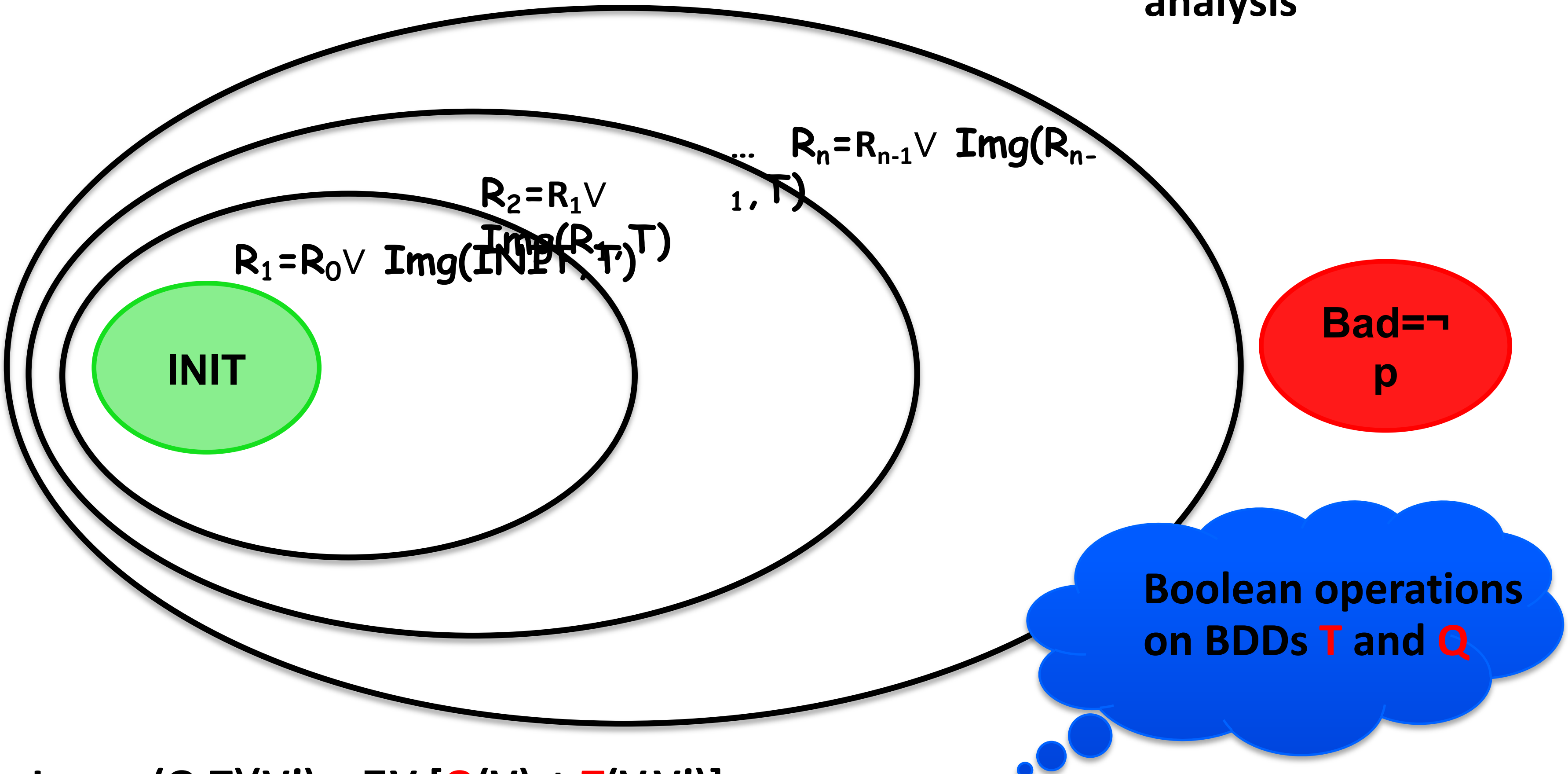
UNIVERSITY OF WATERLOO

BDD-BASED SYMBOLIC REACHABILITY

Forward Reachability Analysis with BDDs

Does AG p hold?

All safety properties reduce to reachability analysis



Image(Q,T)(V') = ∃V [Q(V) ∧ T(V,V')]

Representing Sets as Prop. Formulas

$[F]$ states satisfying F, i.e. $\{\sigma \mid \sigma \vDash F\}$	F propositional formula over V
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	$F_1 \Rightarrow F_2$ i.e. $F_1 \wedge \neg F_2$ unsatisfiable

BDDs in a nutshell

Typically mean Reduced Ordered Binary Decision Diagrams (ROBDDs)

Canonical representation of Boolean formulas

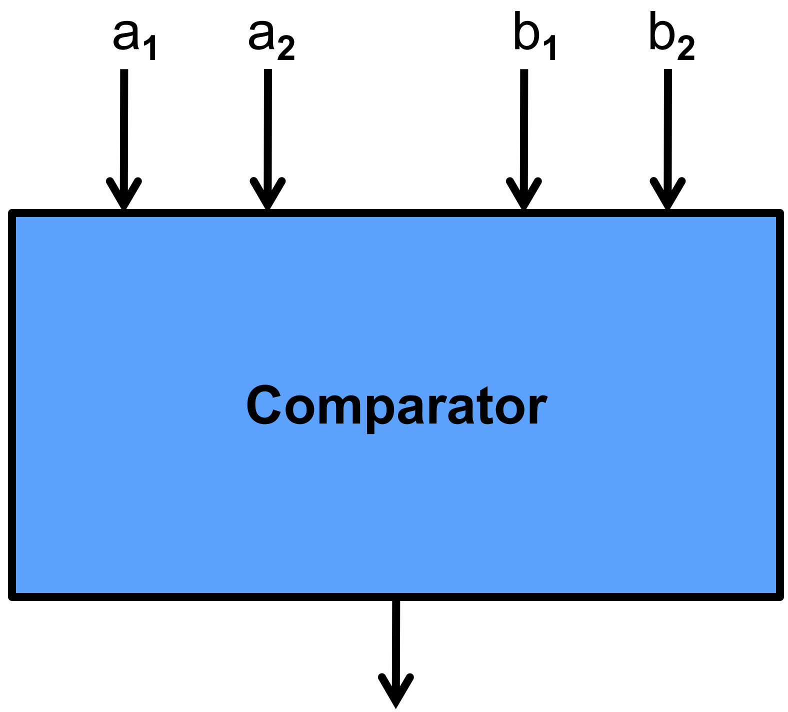
Often substantially more compact than a traditional normal form

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



$f = 1 \Leftrightarrow a_1 = b_1 \wedge a_2 = b_2$

Conjunctive Normal Form

$$f$$

$$a_1 = b_1 \quad \wedge \quad a_2 = b_2$$

$$a_1 \Rightarrow b_1 \wedge b_1 \Rightarrow a_1 \quad \wedge \quad a_2 \Rightarrow b_2 \wedge b_2 \Rightarrow a_2$$

$$(\neg a_1 \vee b_1) \wedge (\neg b_1 \vee a_1) \wedge (\neg a_2 \vee b_2) \wedge (\neg b_2 \vee a_2)$$

$$(\neg b_1 \vee a_1) \wedge (\neg a_1 \vee b_1) \wedge (\neg a_2 \vee b_2) \wedge (\neg b_2 \vee a_2)$$

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

$$(a_0 \land f(true, a_1, \ldots, a_n)) \lor (\neg a_0 \land f(false, a_1, \ldots, a_n))$$

or, simply,

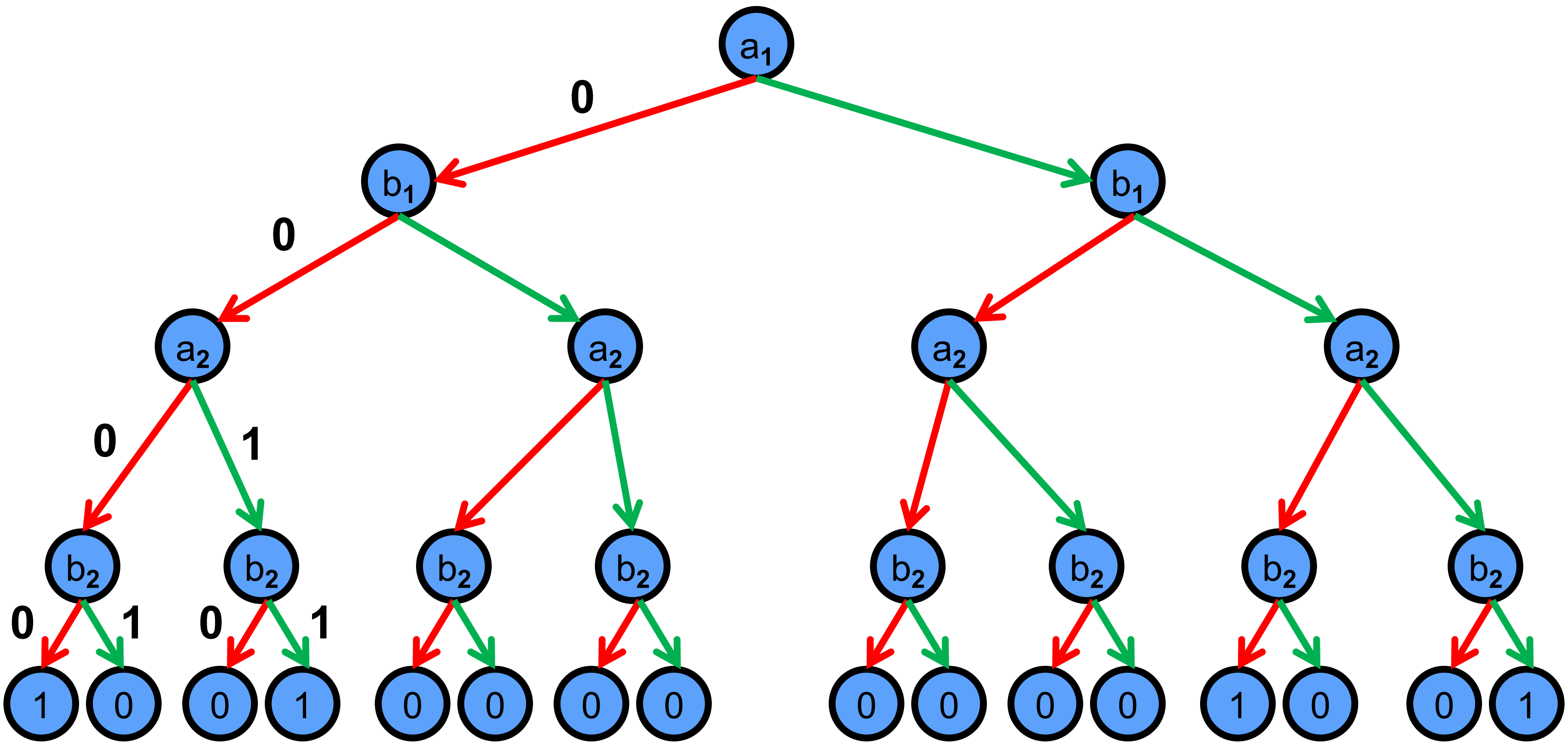
$$ITE\ (a_0, f(true, a_1, \ldots, a_n), f(false, a_1, \ldots, a_n))$$

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. $\neg a_0$

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)

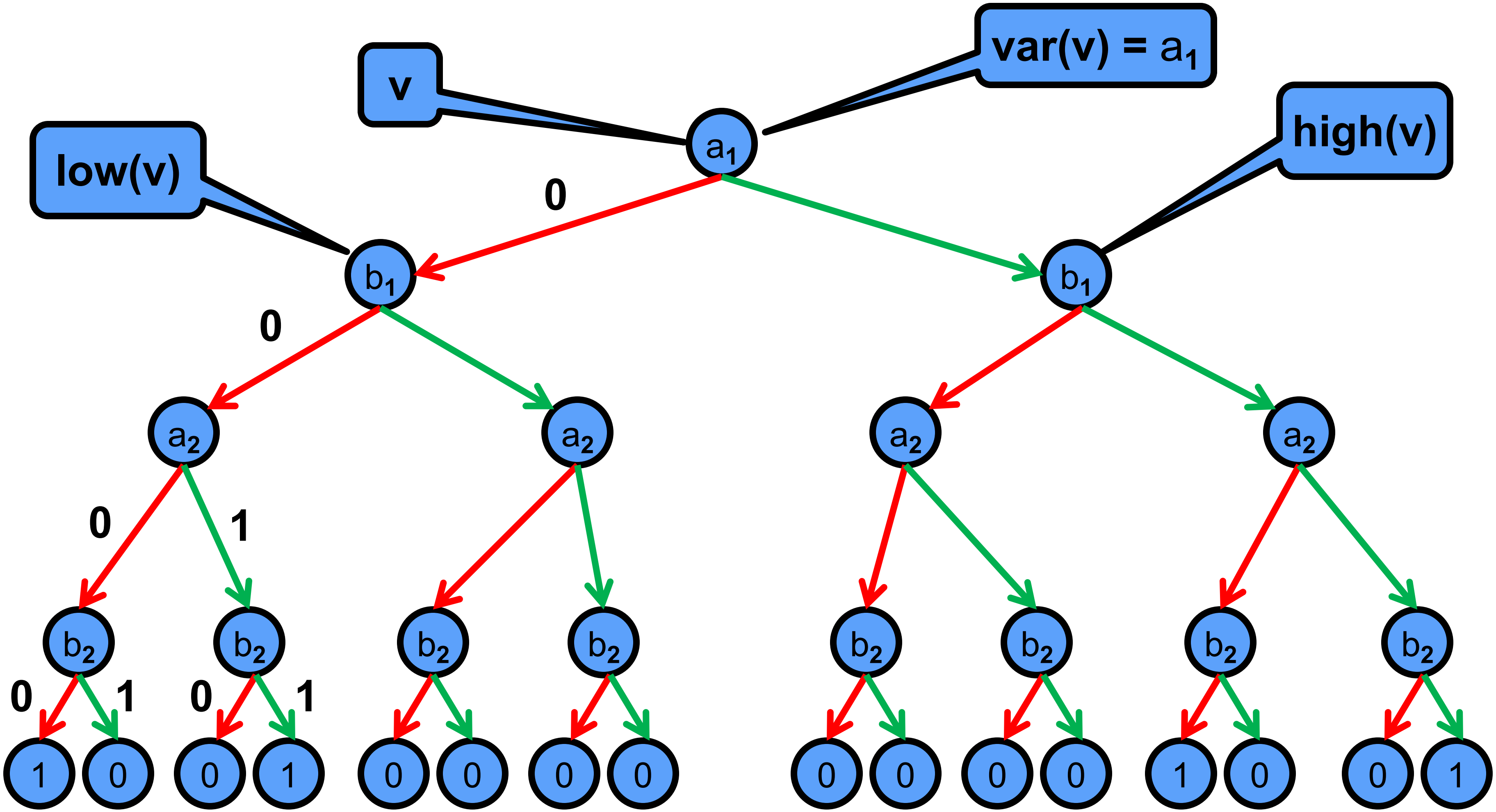
Each path corresponds to a (partial) truth assignment to variables

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

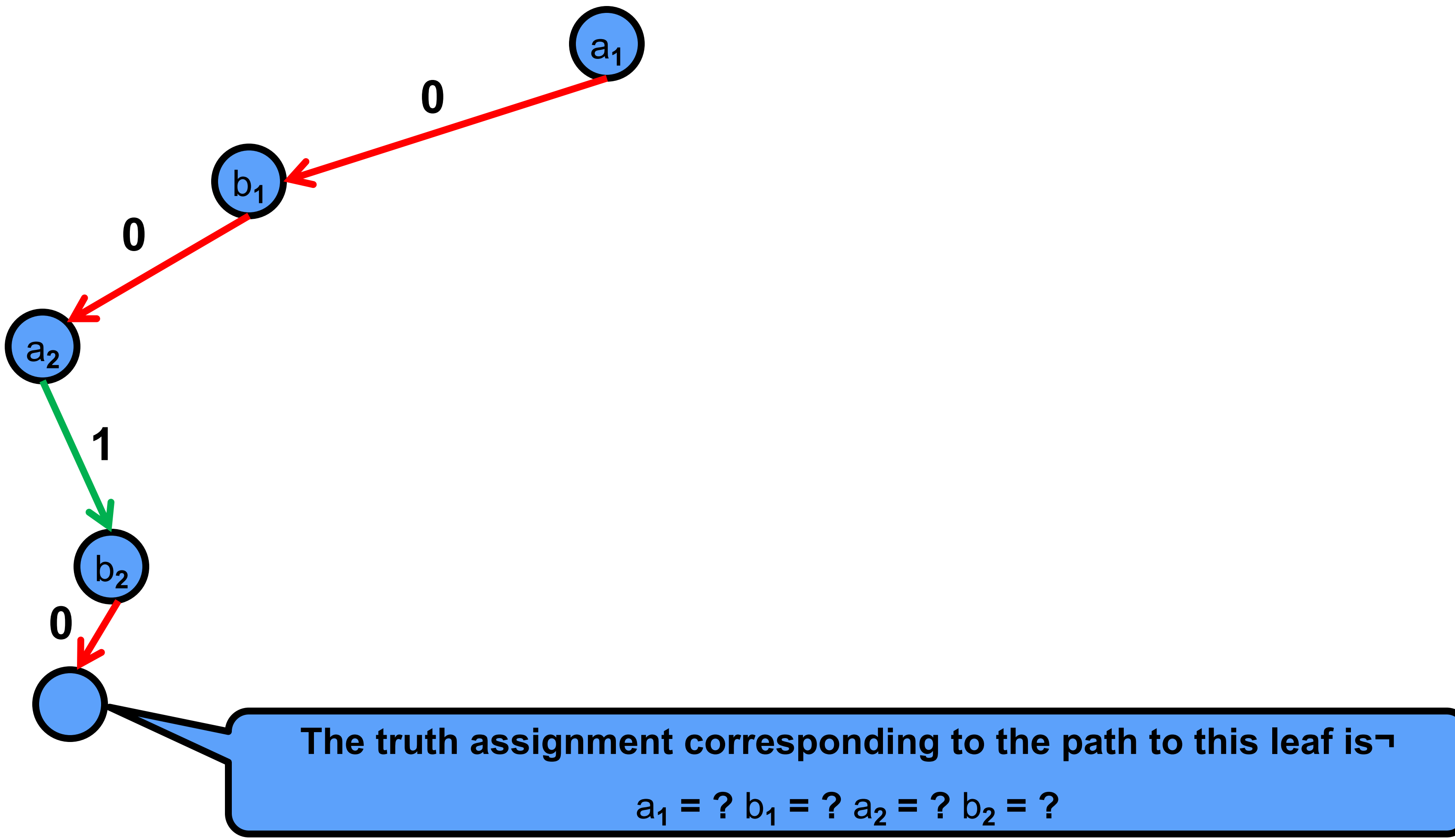
Value of a leaf is determined by:

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

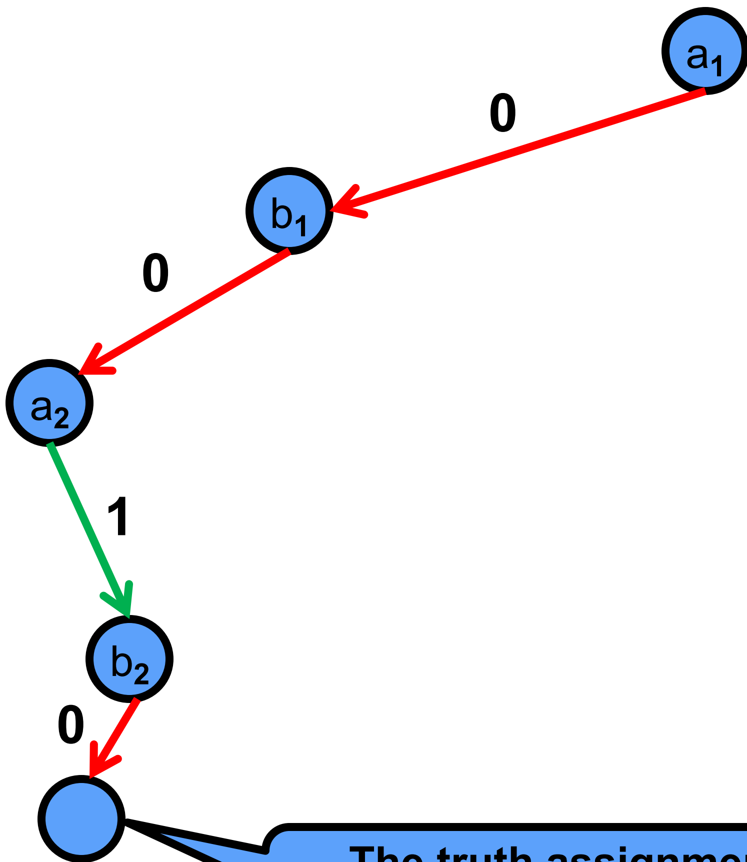
Binary Decision Tree



Binary Decision Tree



Binary Decision Tree

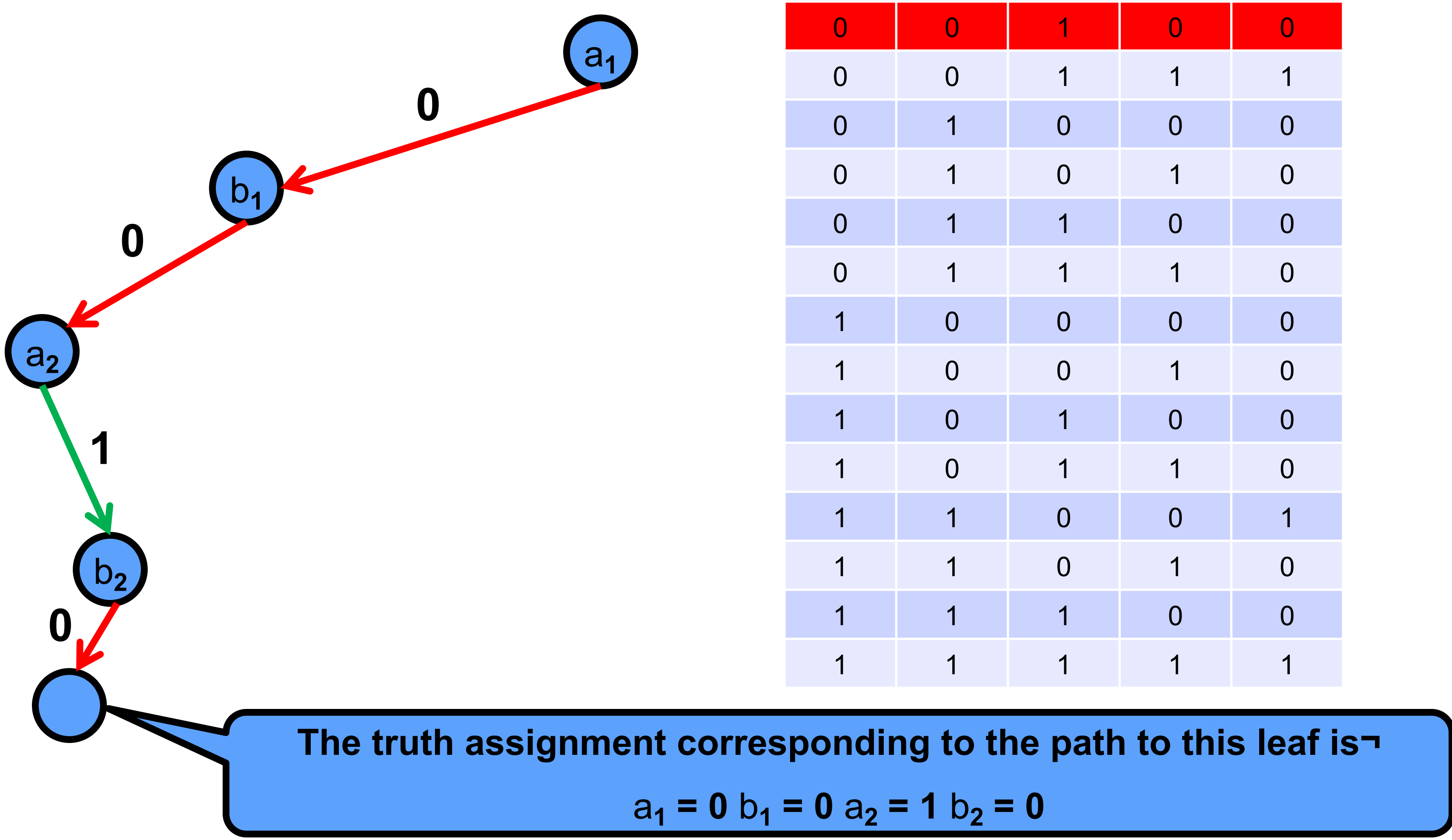


a_1	b_1	a_2	b_2	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

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

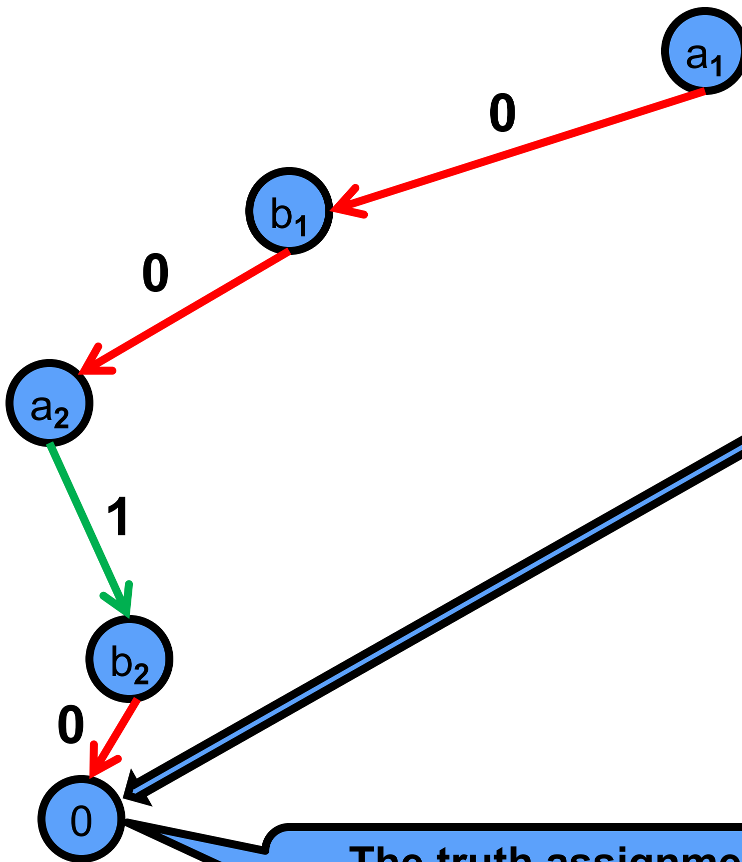
$a_1 = 0$ $b_1 = 0$ $a_2 = 1$ $b_2 = 0$

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

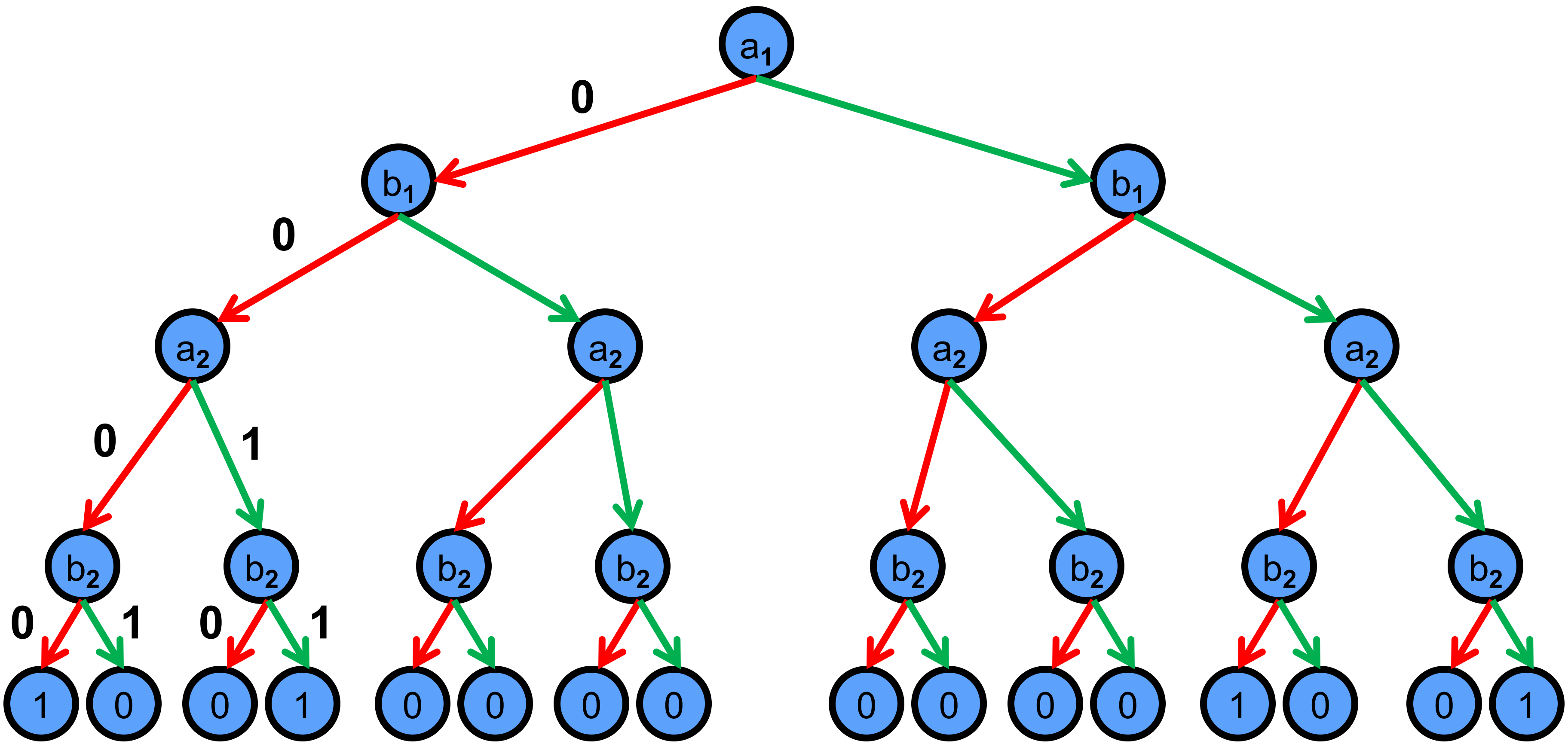


a_1	b_1	a_2	b_2	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

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

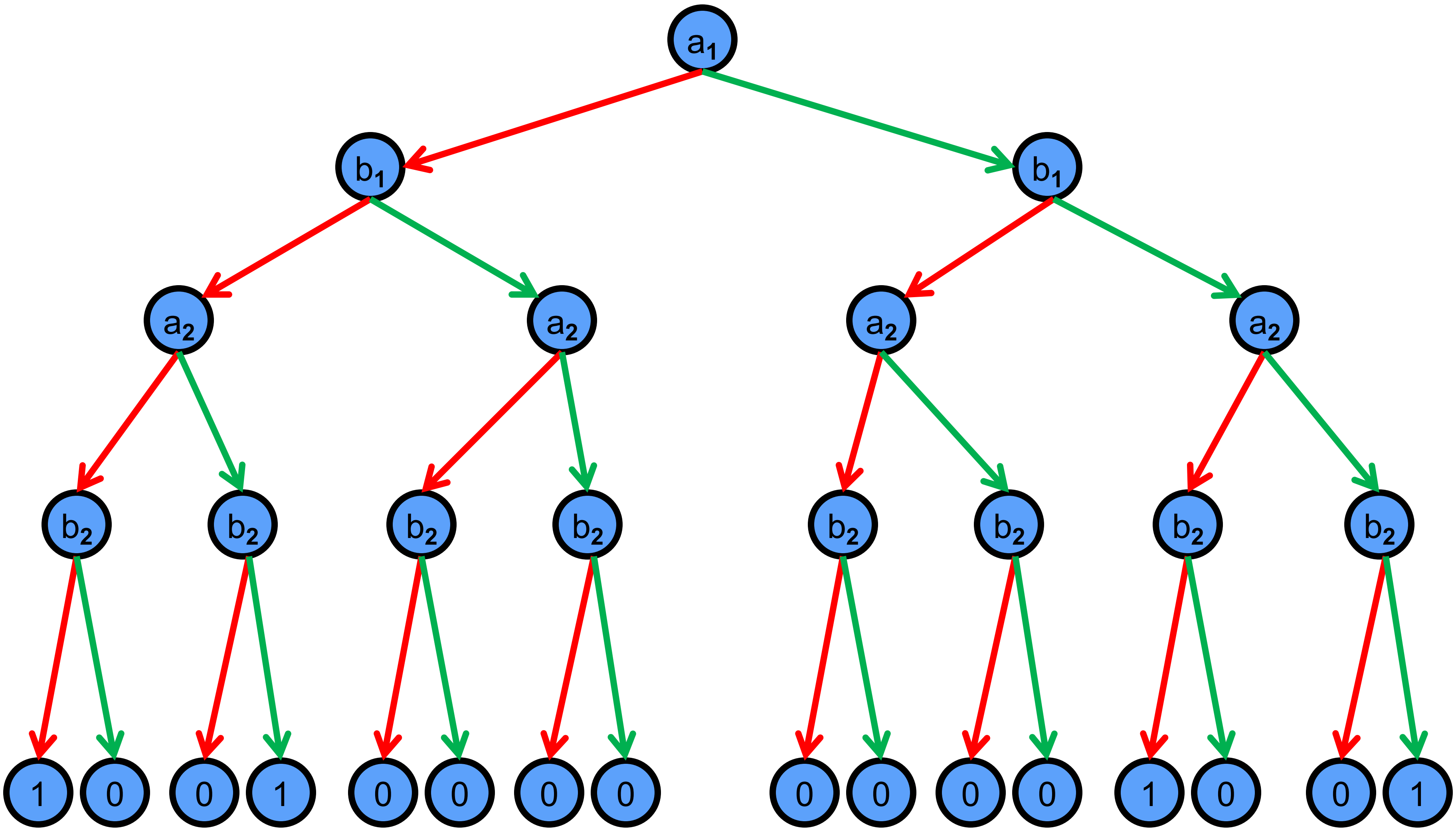
Start with OBDT and repeatedly apply the following two operations as long as possible¬

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?

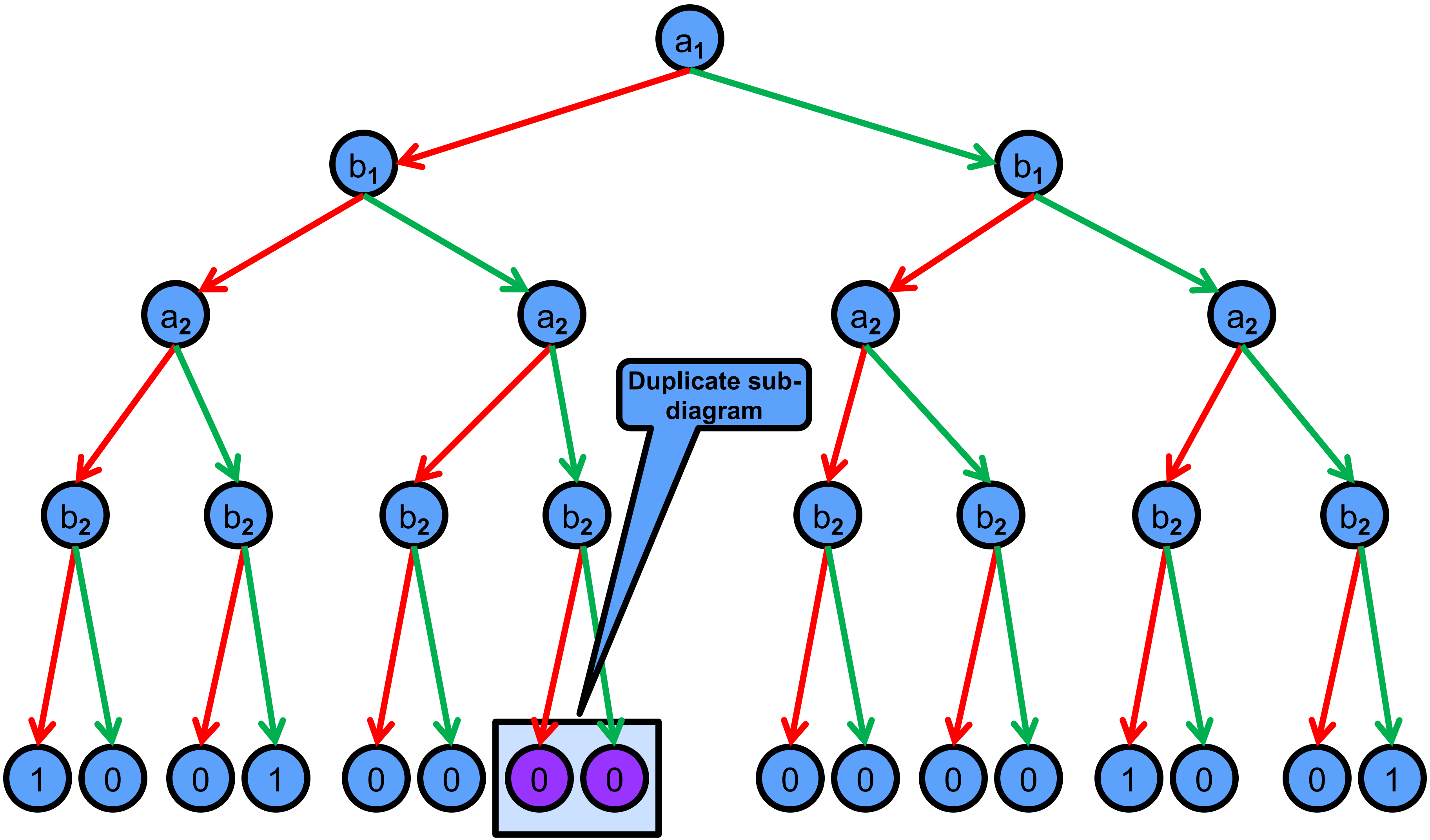
ROBDD is often exponentially smaller than the corresponding OBDT

OBDT to ROBDD



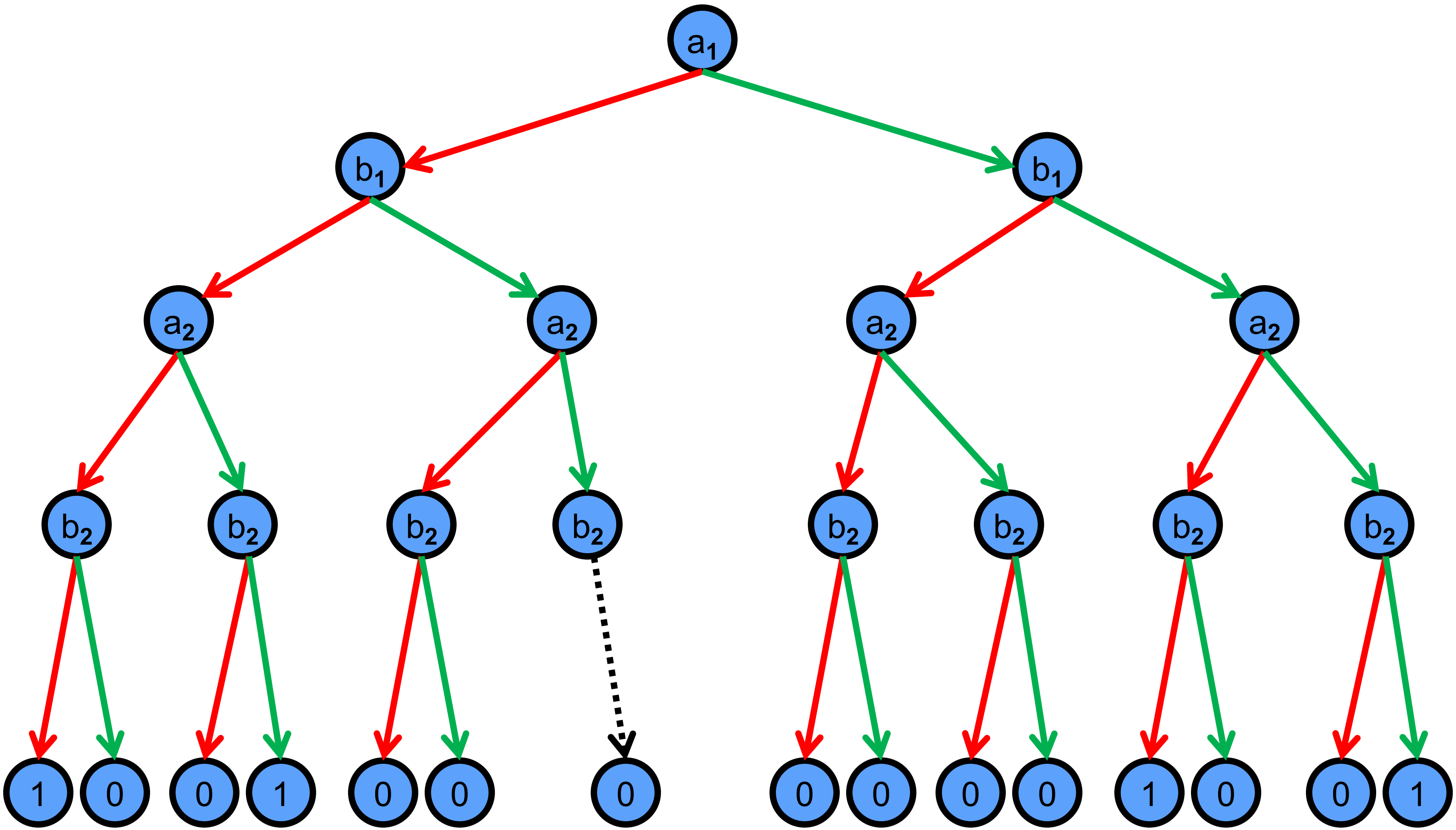
UNIVERSITY OF WATERLOO

OBDT to ROBDD

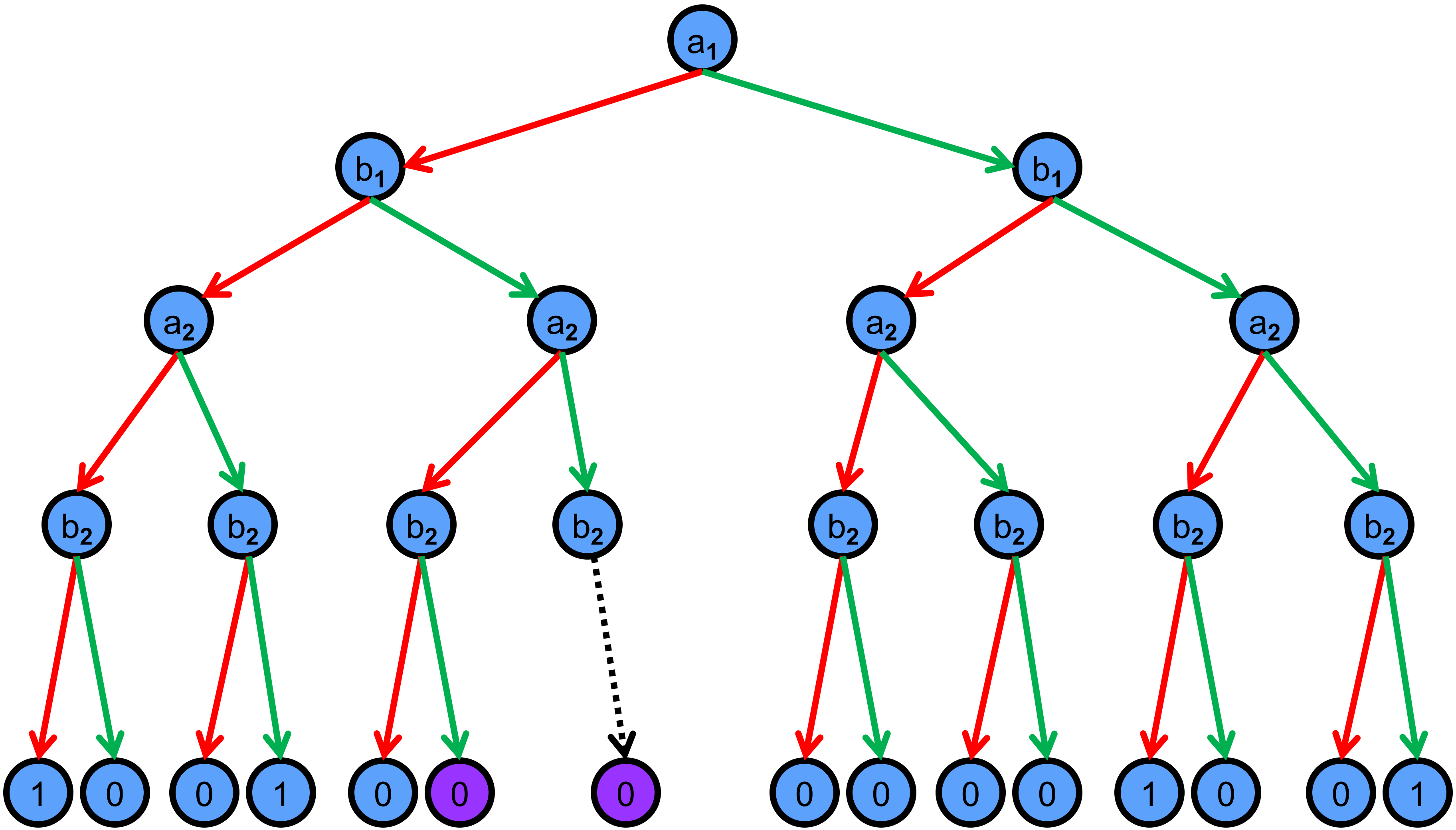


UNIVERSITY OF WATERLOO

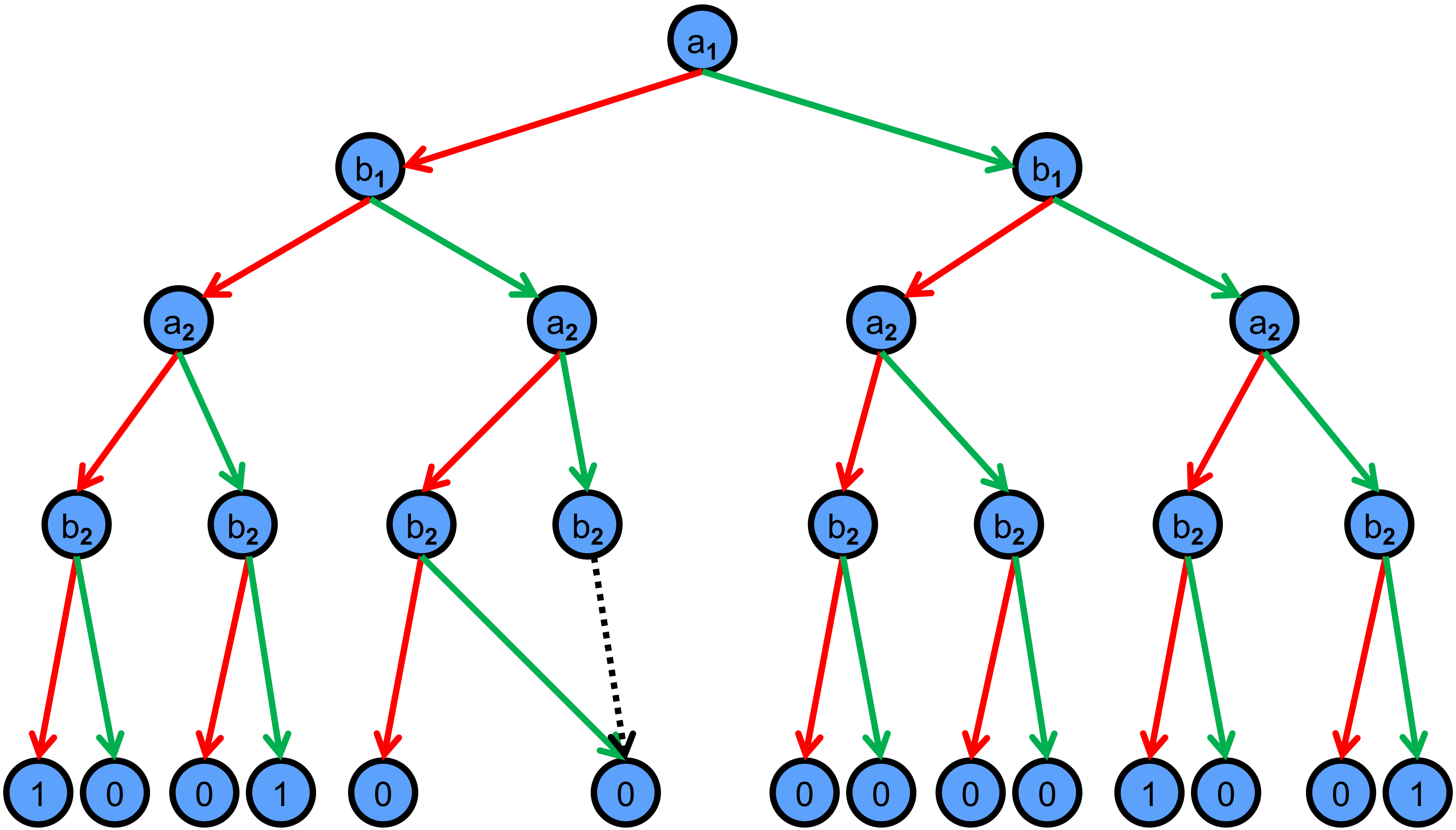
OBDT to ROBDD



OBDT to ROBDD

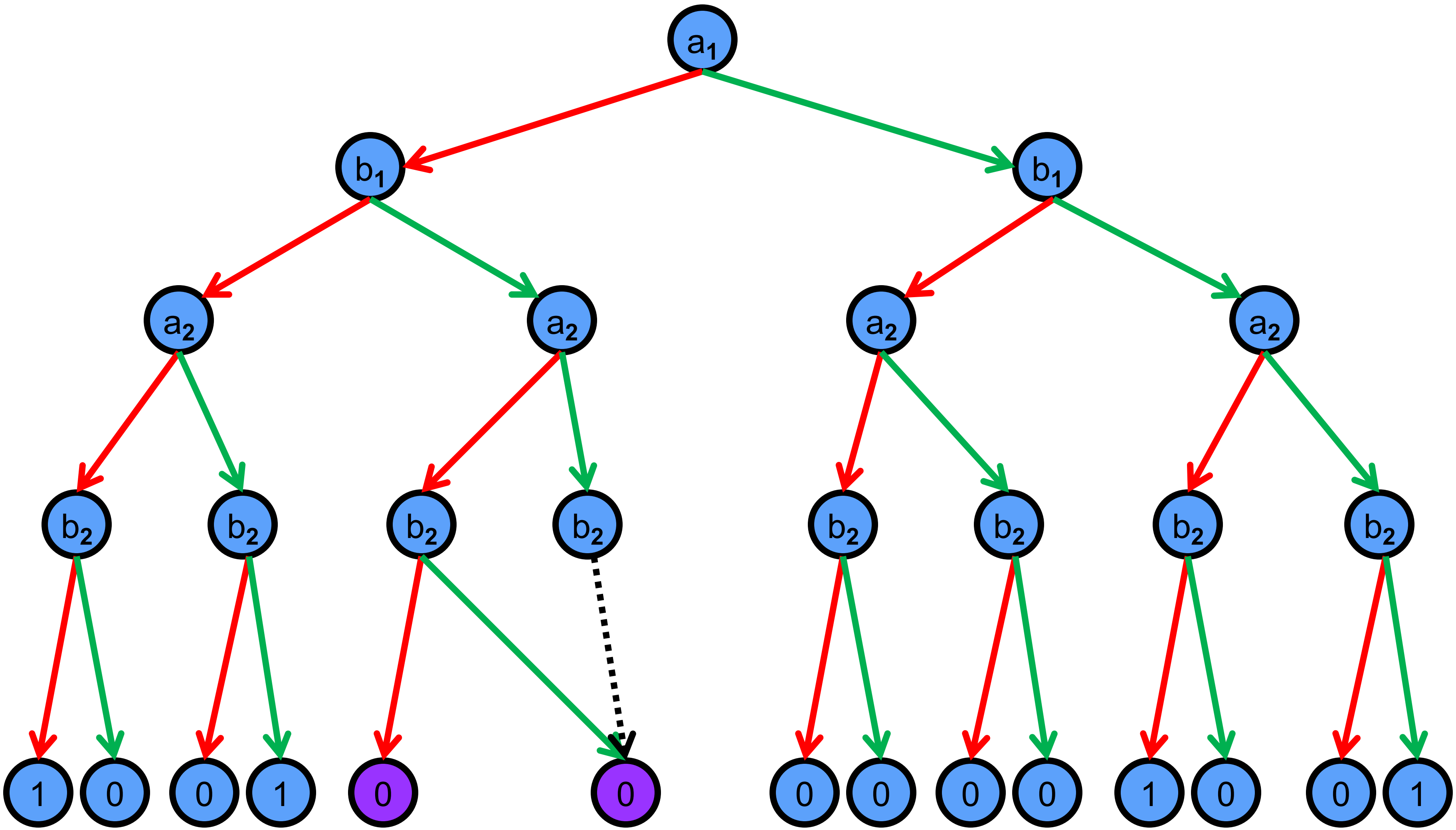


OBDT to ROBDD

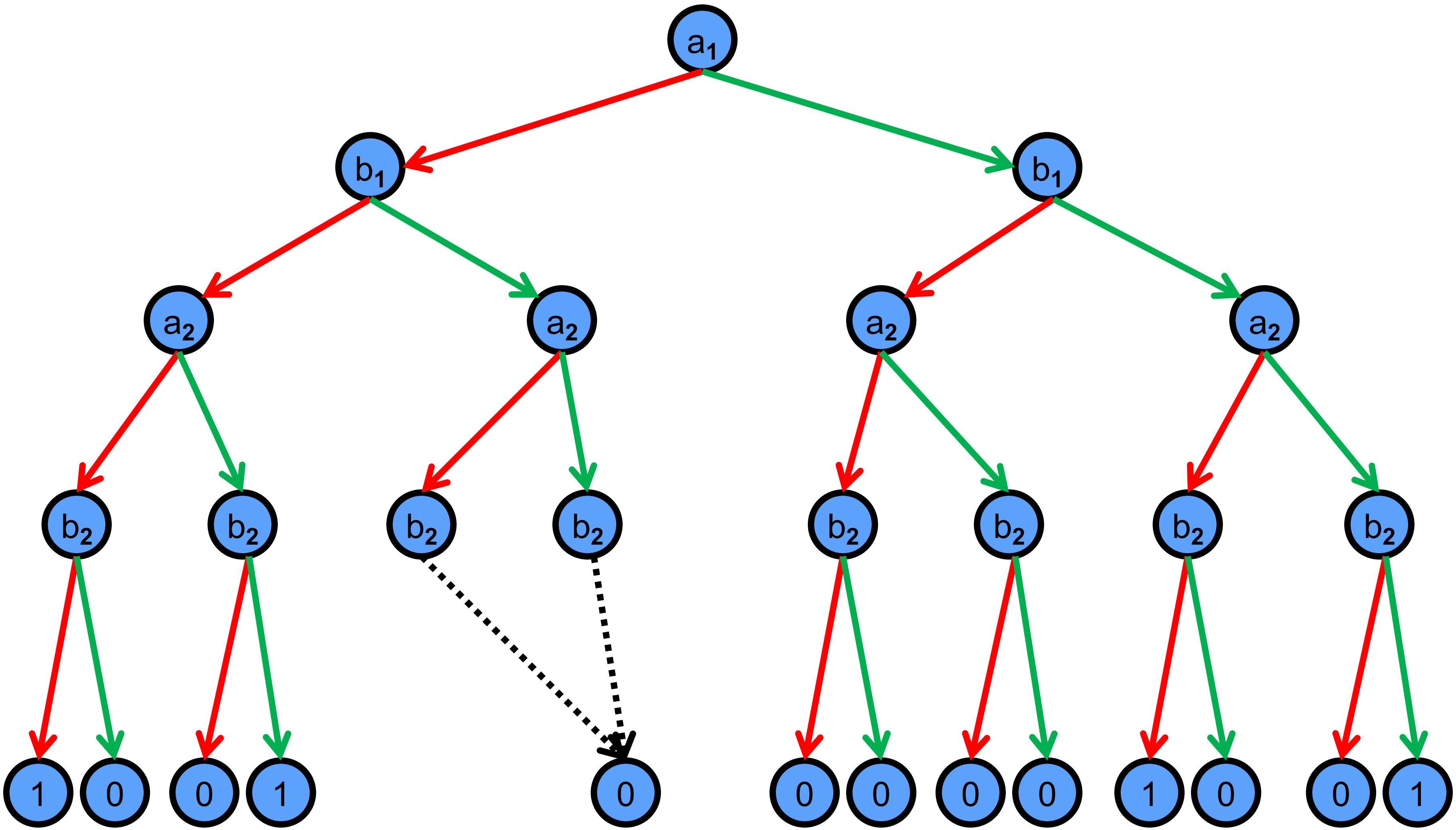


UNIVERSITY OF WATERLOO

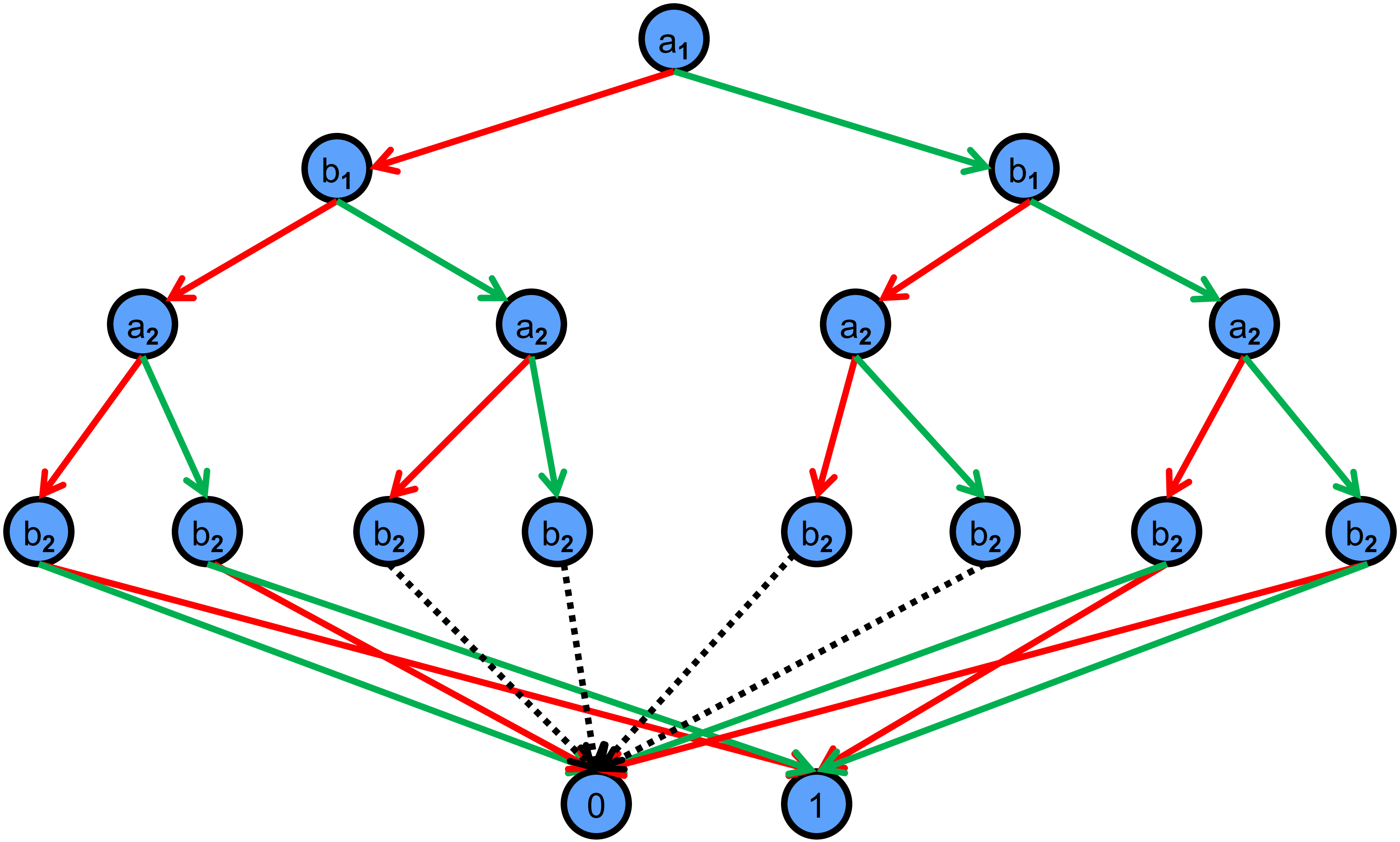
OBDT to ROBDD



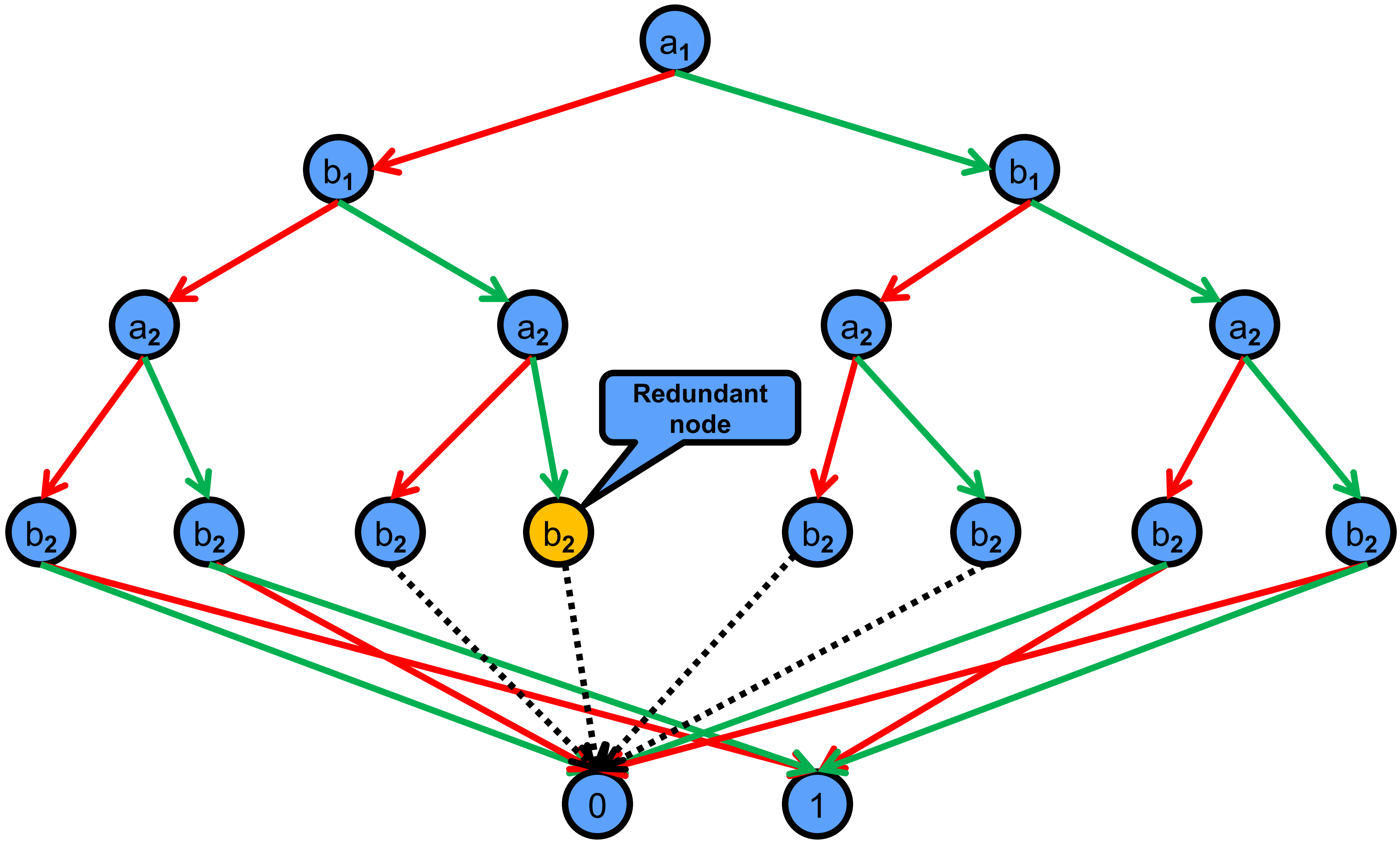
OBDT to ROBDD



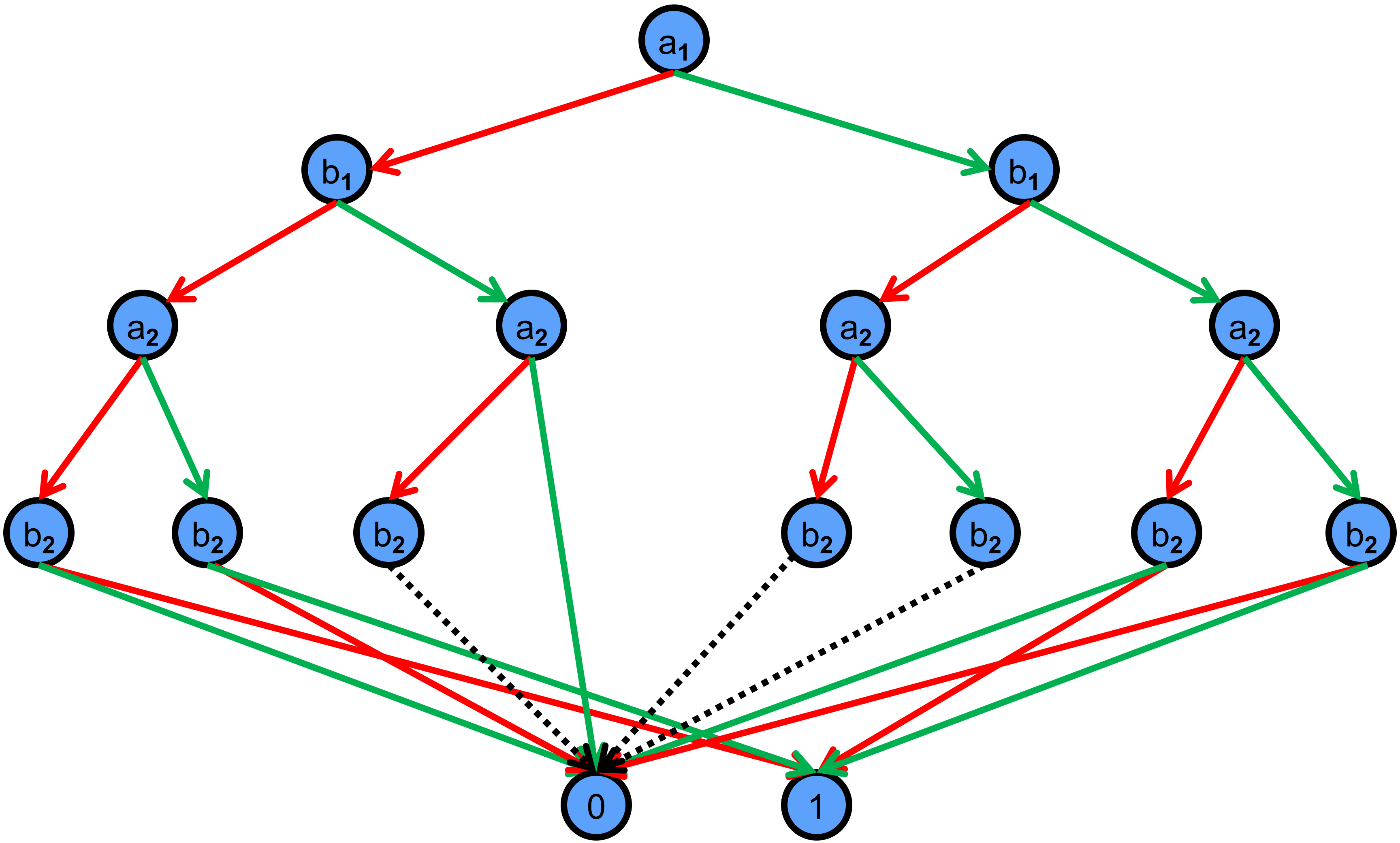
OBDT to ROBDD



OBDT to ROBDD

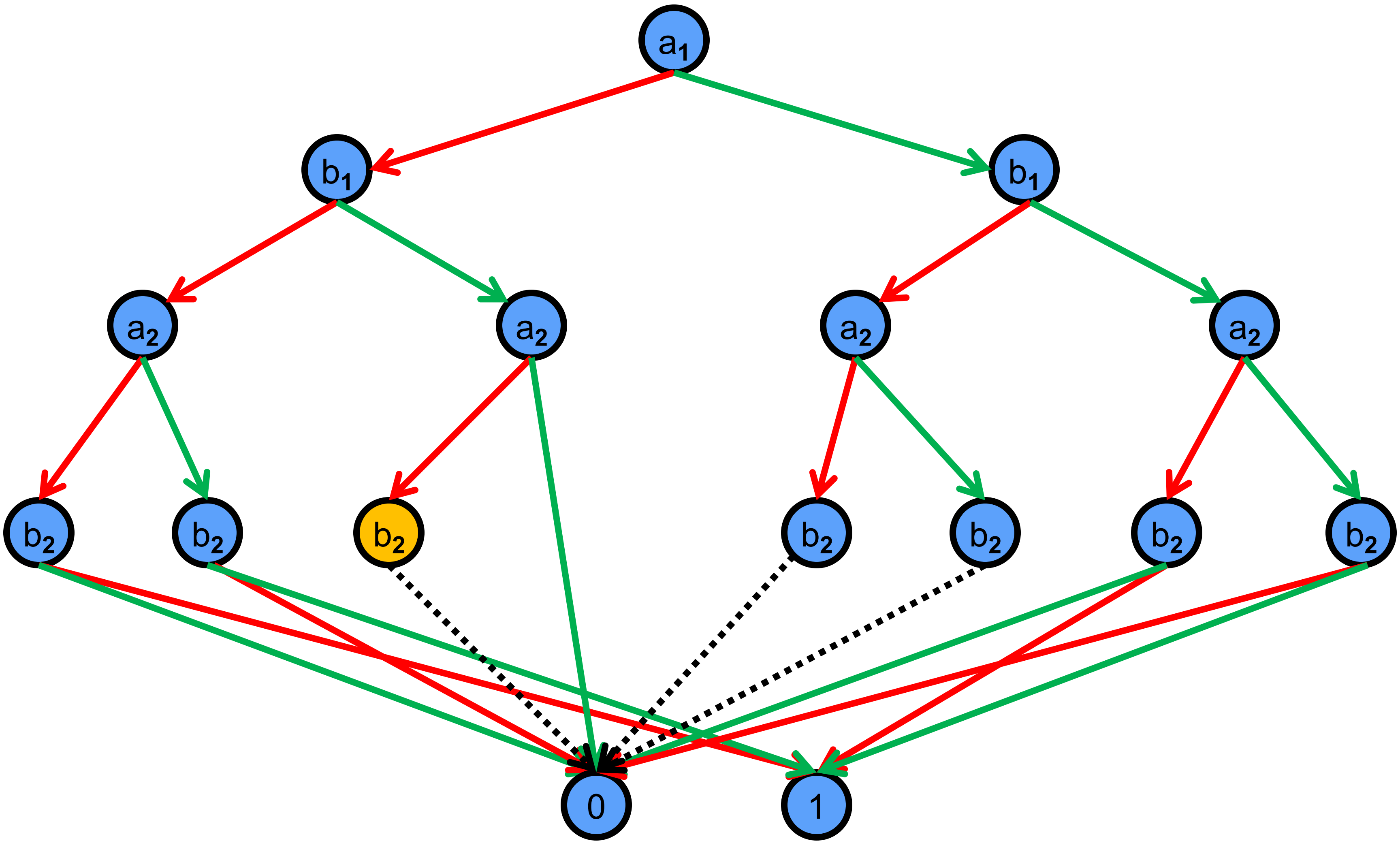


OBDT to ROBDD



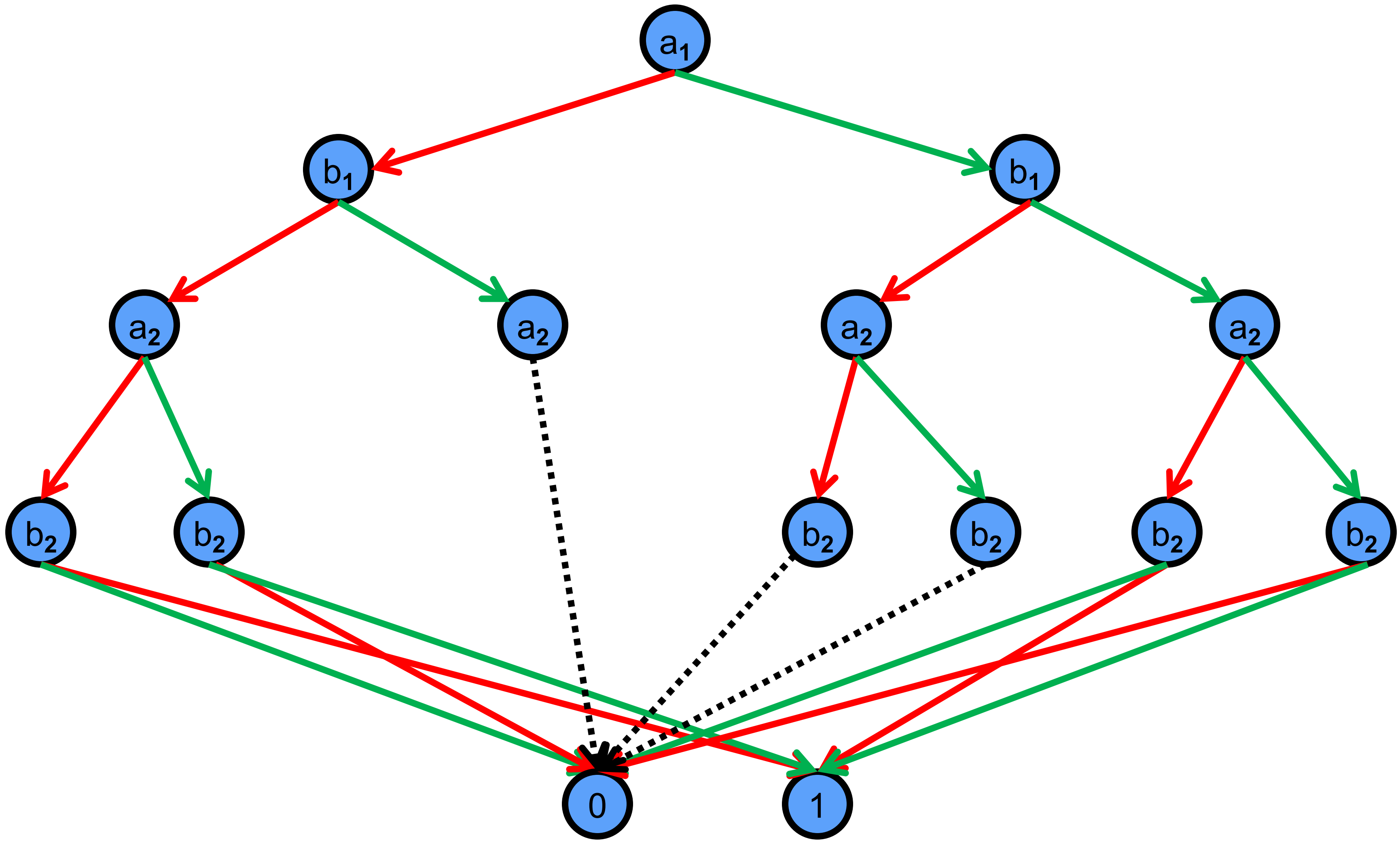
UNIVERSITY OF WATERLOO

OBDT to ROBDD

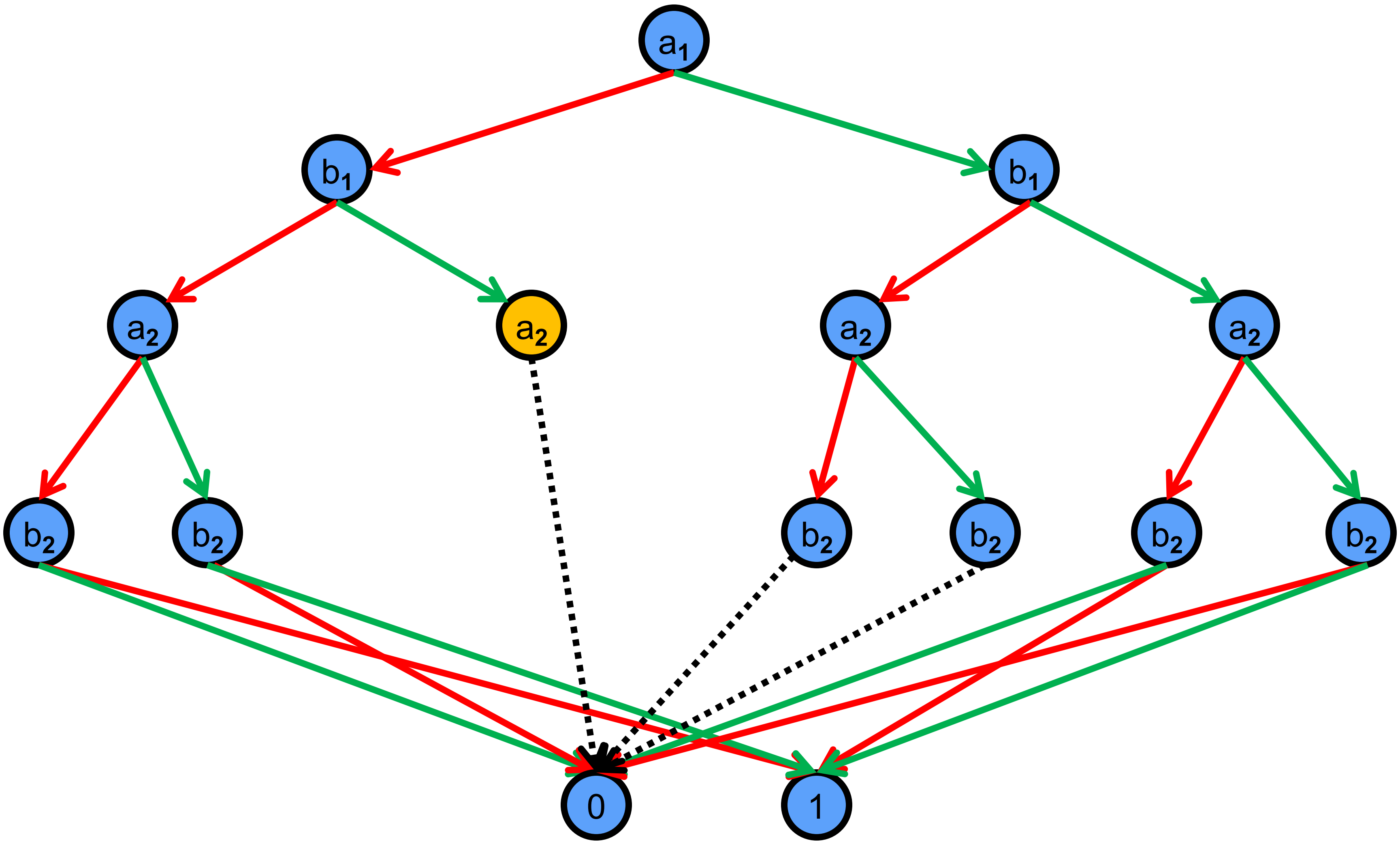


UNIVERSITY OF WATERLOO

OBDT to ROBDD



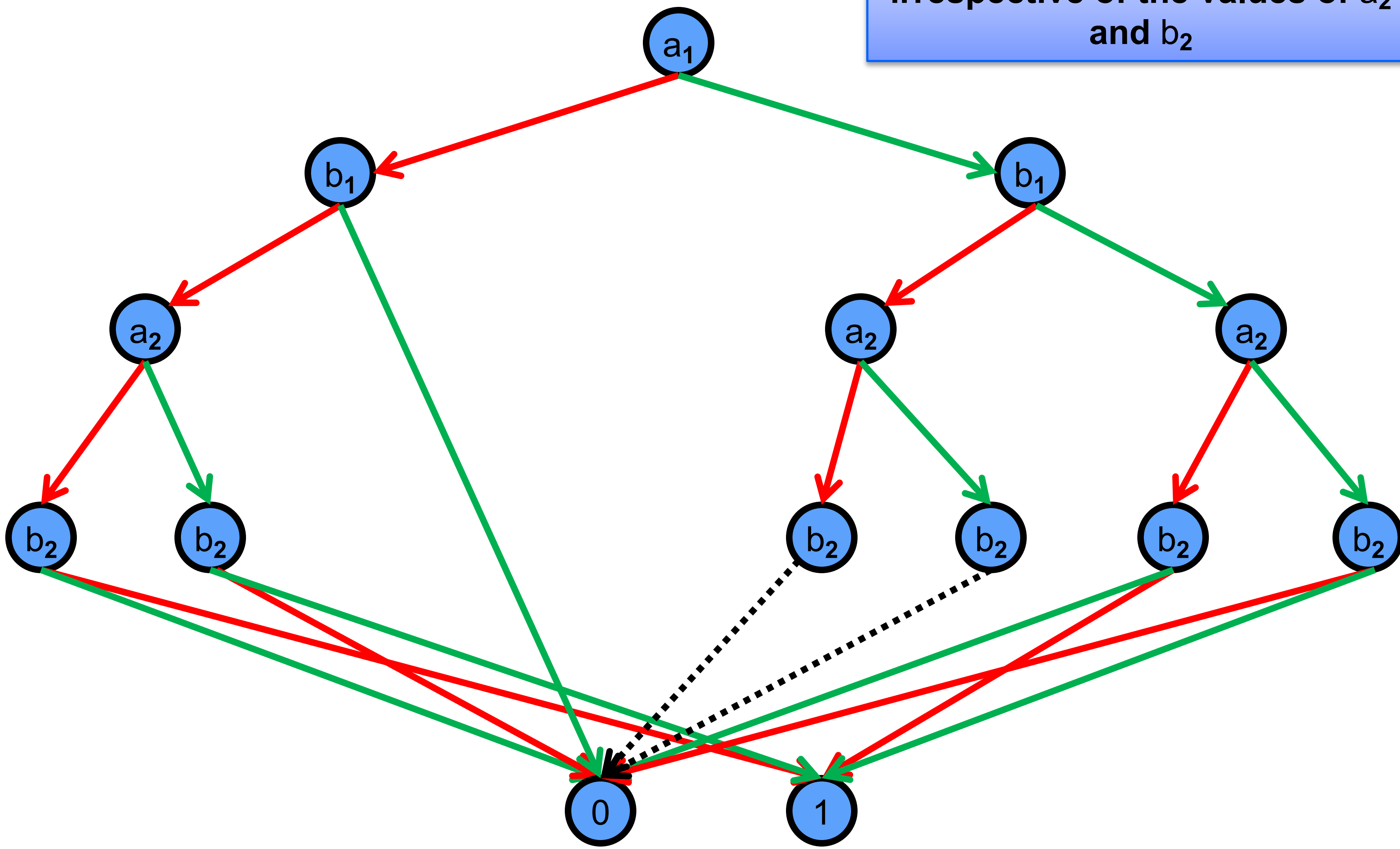
OBDT to ROBDD



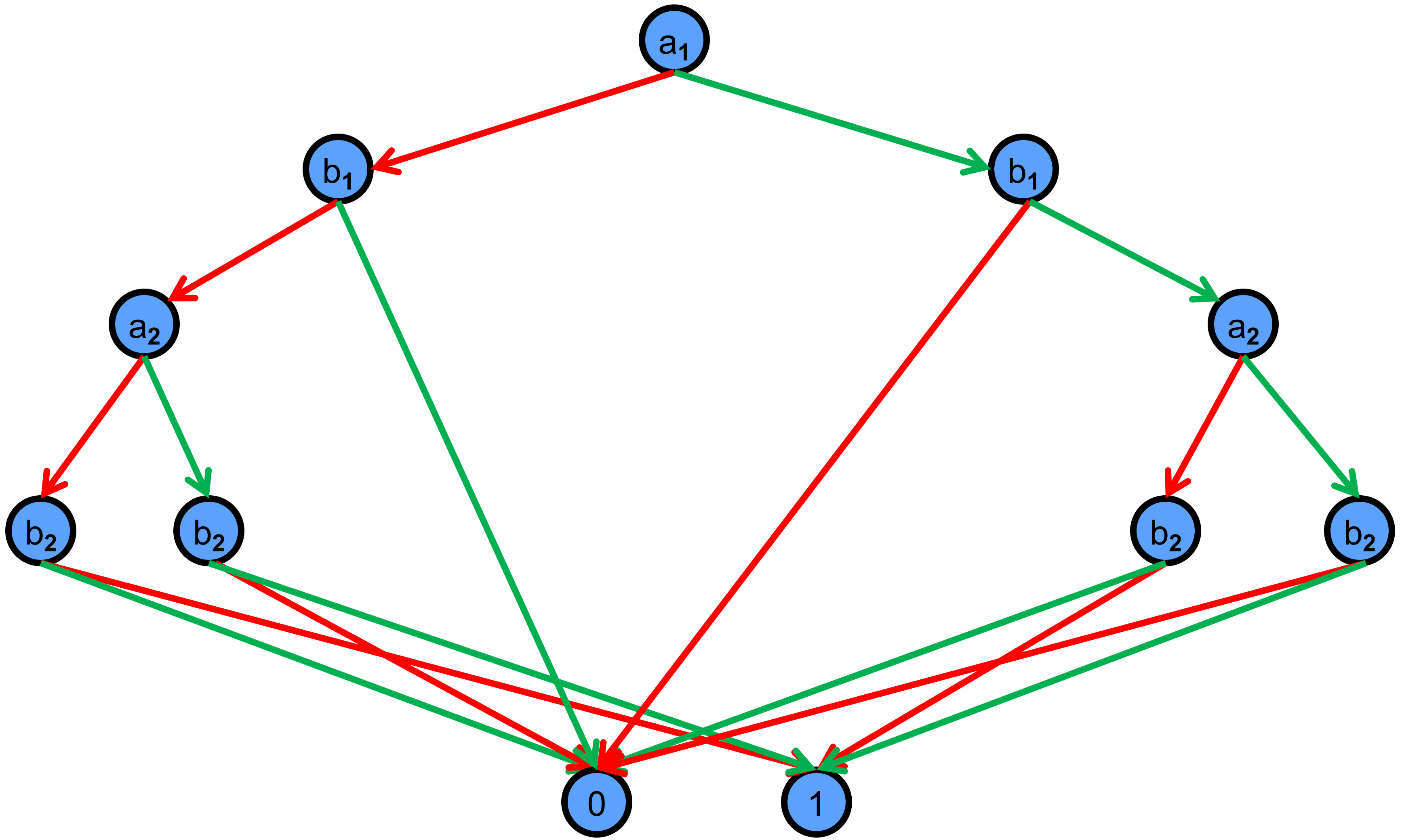
UNIVERSITY OF WATERLOO

OBDT to ROBDD

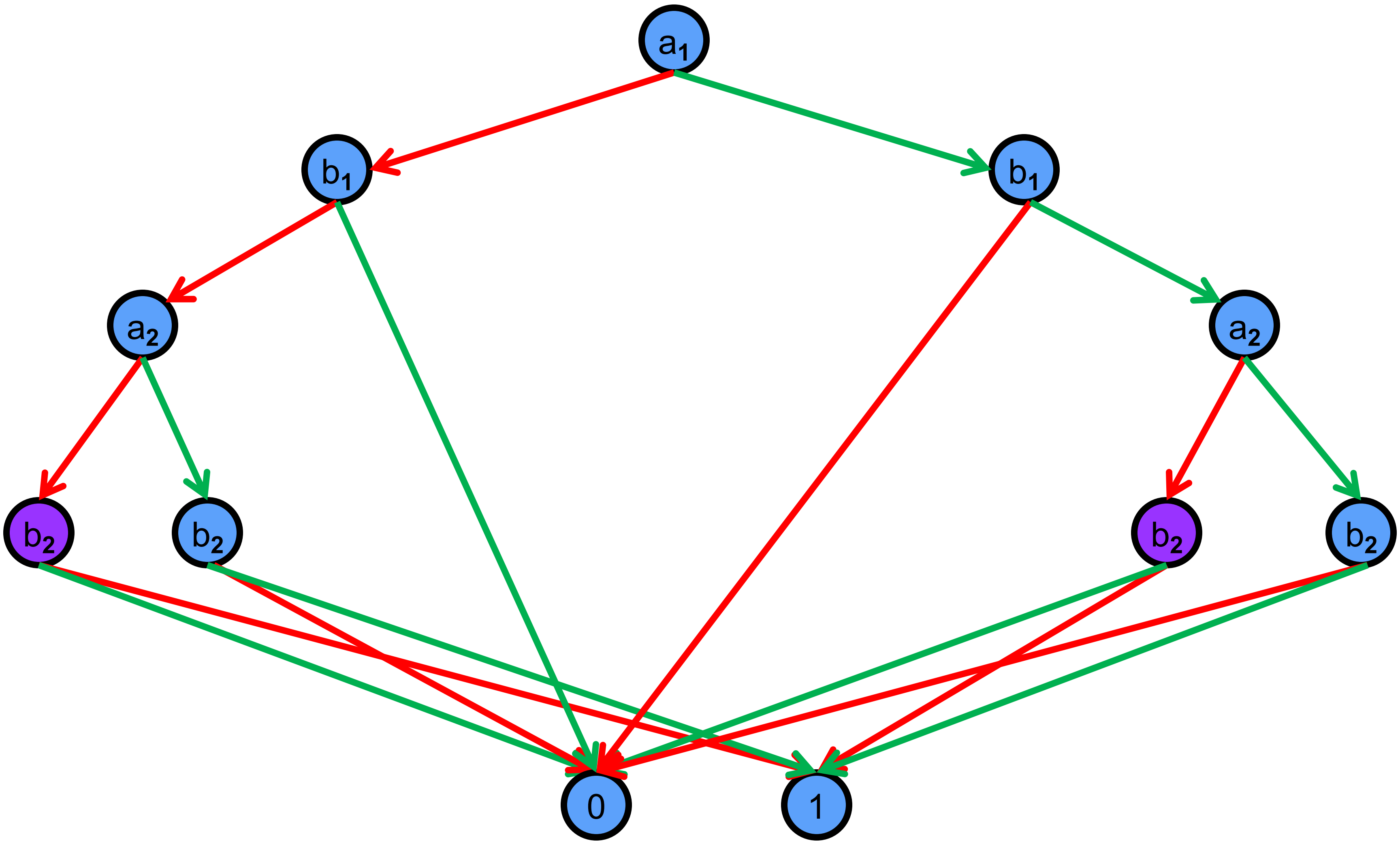
If a_1 = 0 and b_1 = 1 then f = 0 irrespective of the values of a_2 and b_2



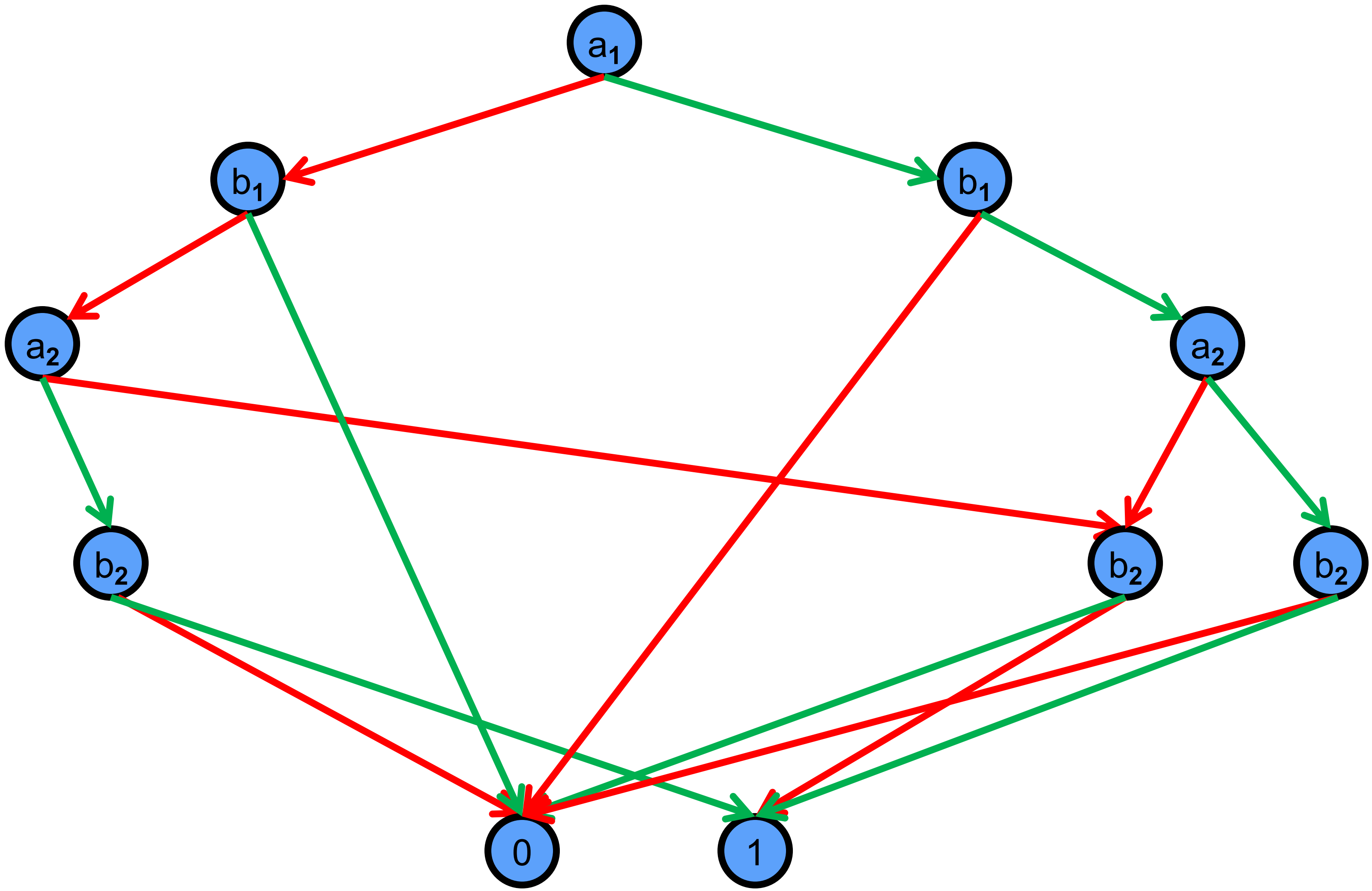
OBDT to ROBDD



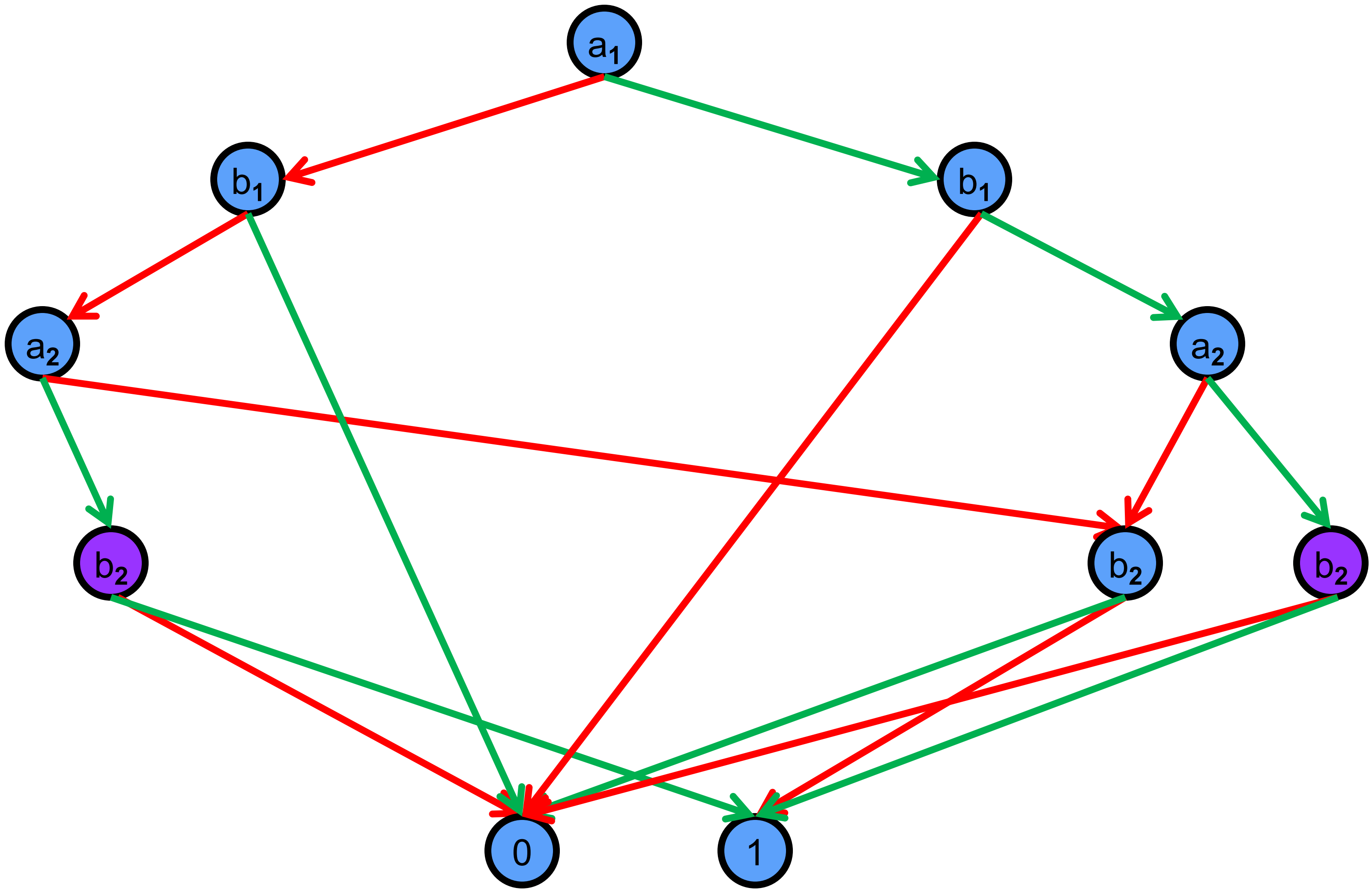
OBDT to ROBDD



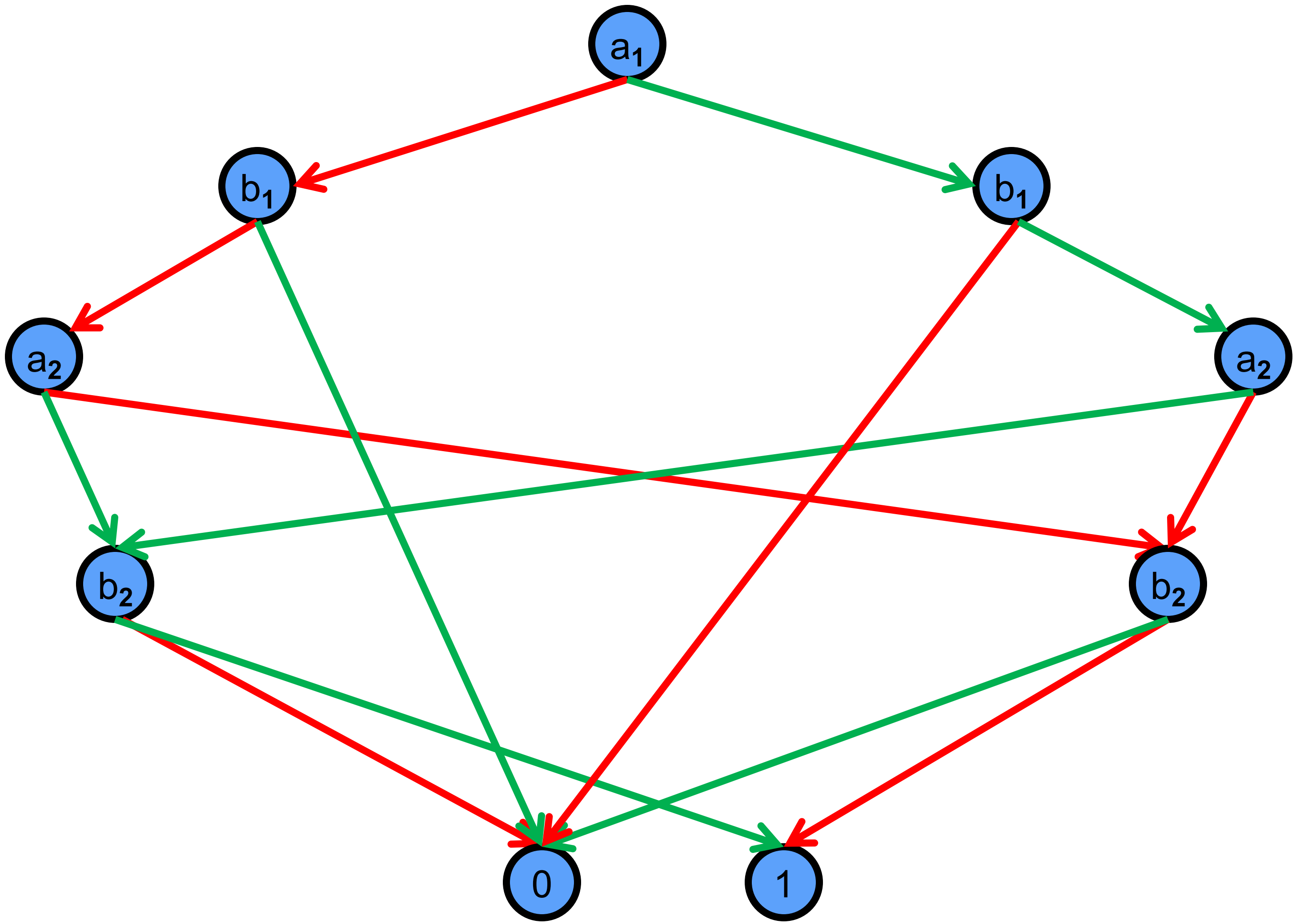
OBDT to ROBDD



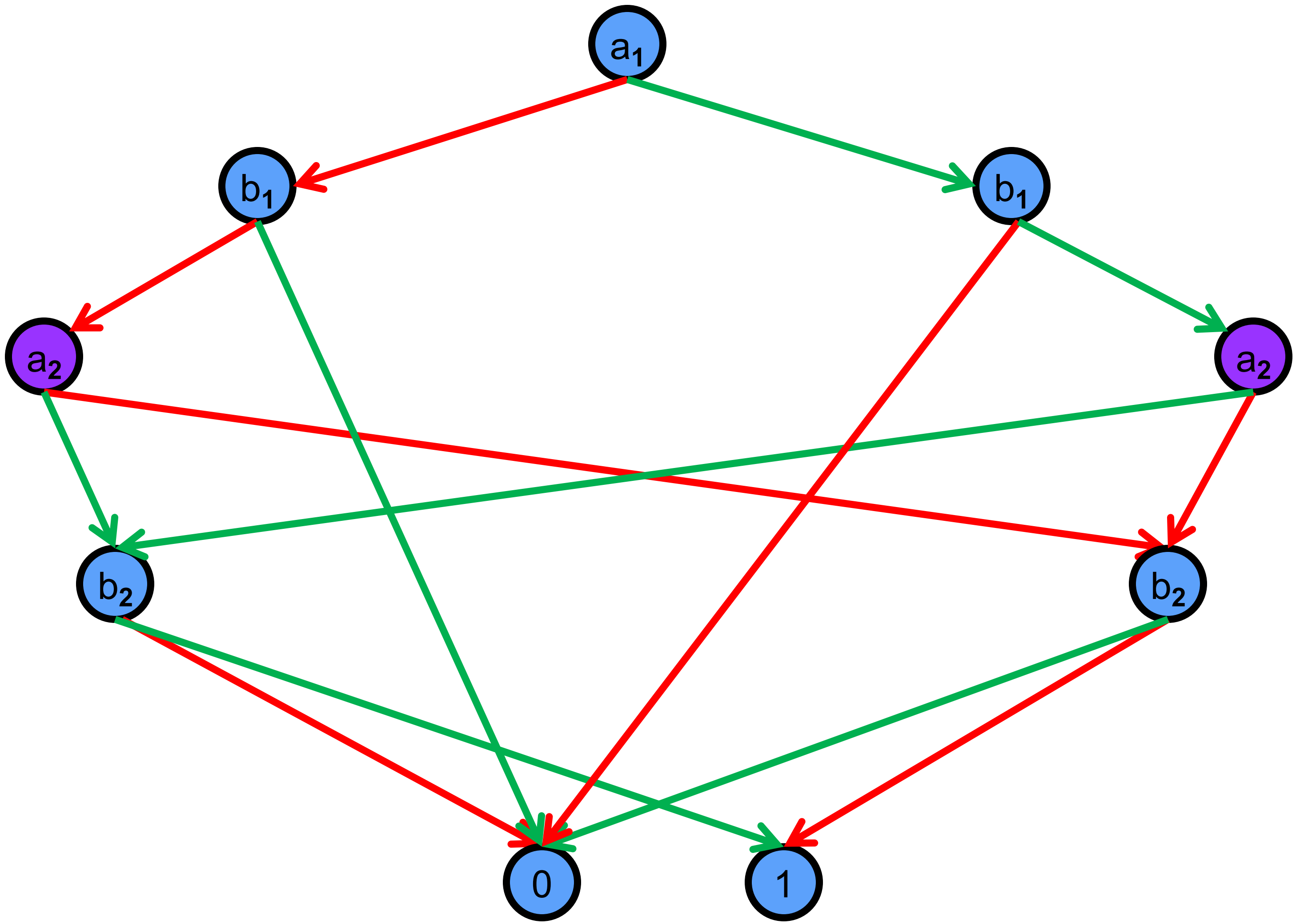
OBDT to ROBDD



OBDT to ROBDD

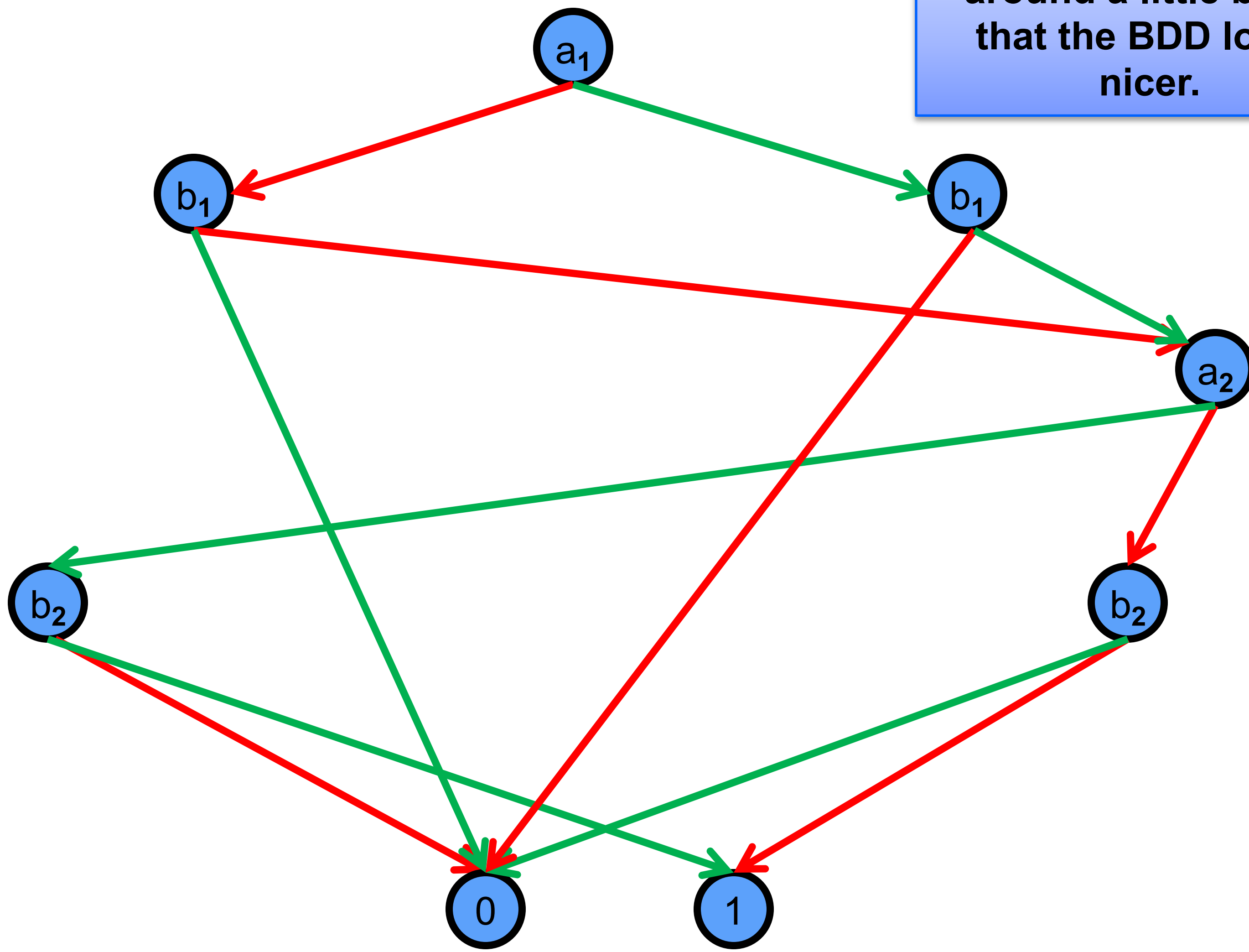


OBDT to ROBDD

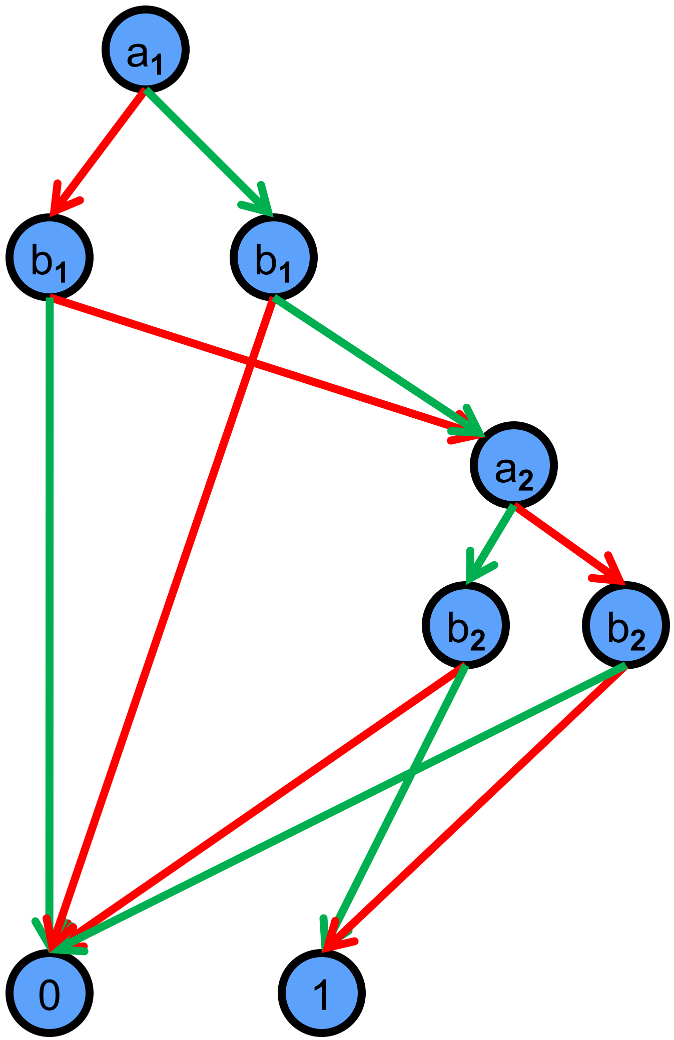


OBDT to ROBDD

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?

UNIVERSITY OF WATERLOO

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

- $f_1 = f_2 \Leftrightarrow$?

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

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

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

- $f_1 = f_2 \Leftrightarrow$ BDD(f_1) and BDD(f_2) are isomorphic
- f is unsatisfiable $\Leftrightarrow$ BDD(f) is the leaf node "0"
- f is valid $\Leftrightarrow$?

ROBDD (a.k.a. BDD) Summary

BDDs are canonical representations of Boolean formulas

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

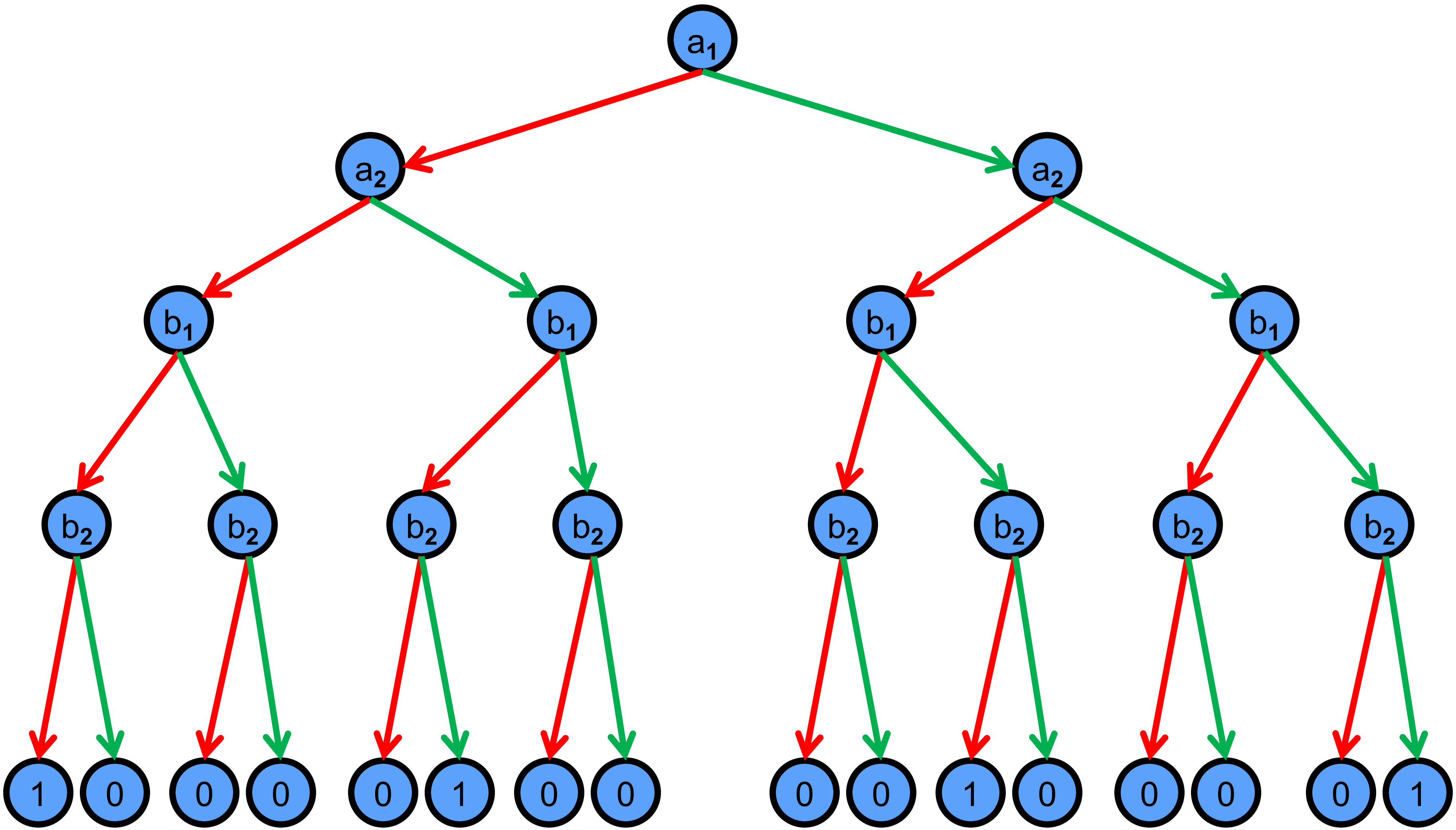
Logical operations can be performed efficiently on BDDs

- Polynomial in argument size

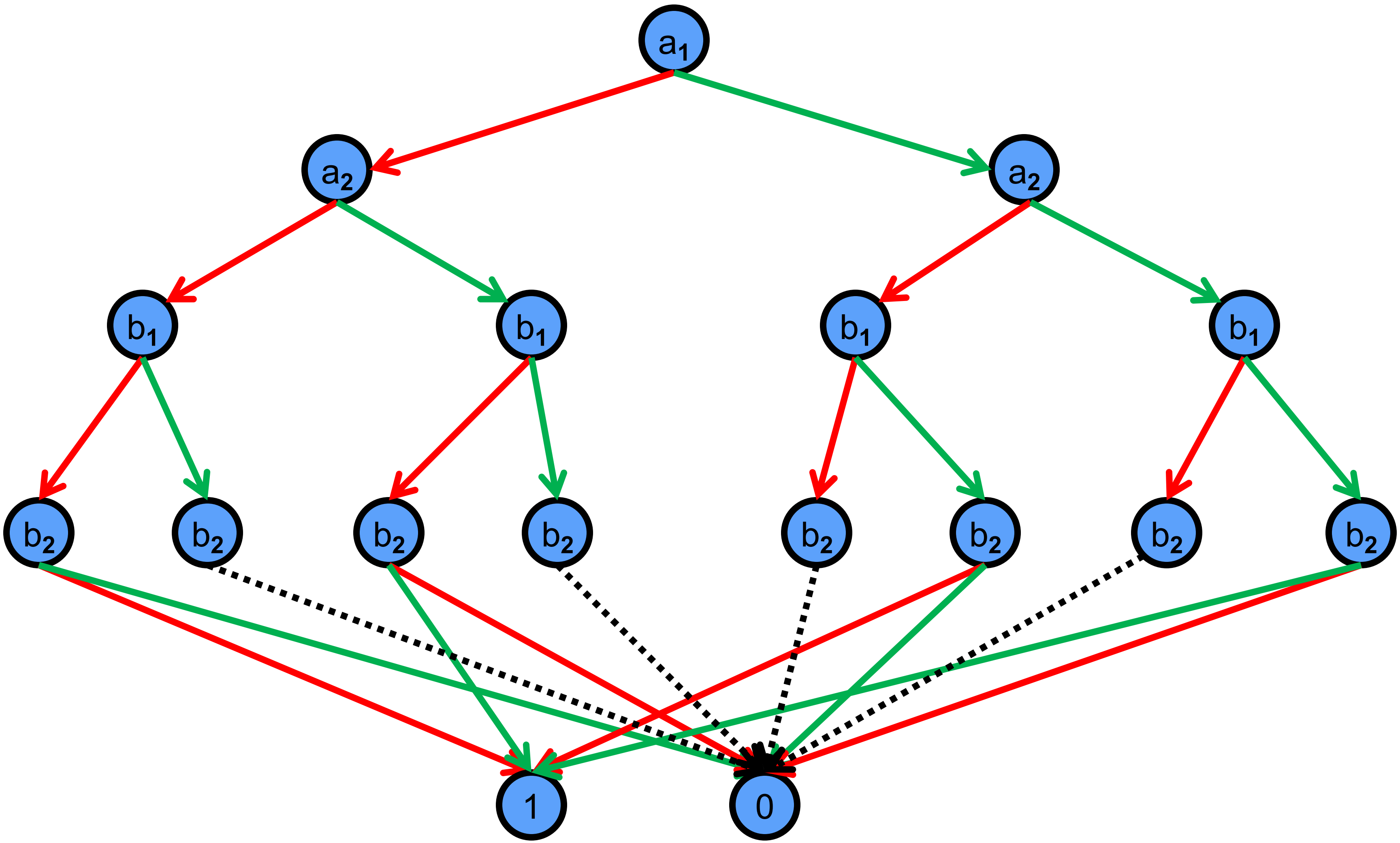
BDD size depends critically on the variable ordering

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

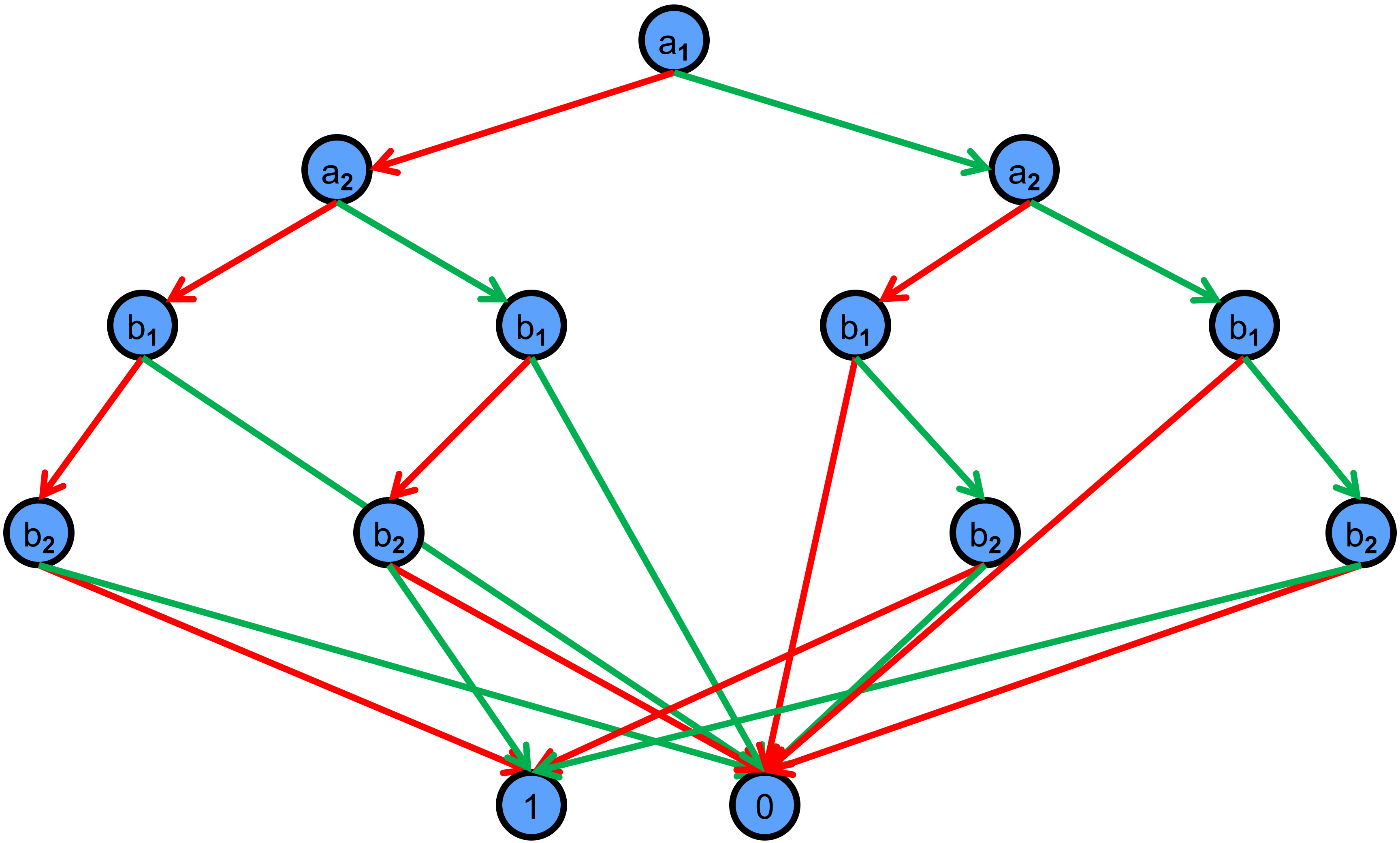
ROBDD and variable ordering



ROBDD and variable ordering

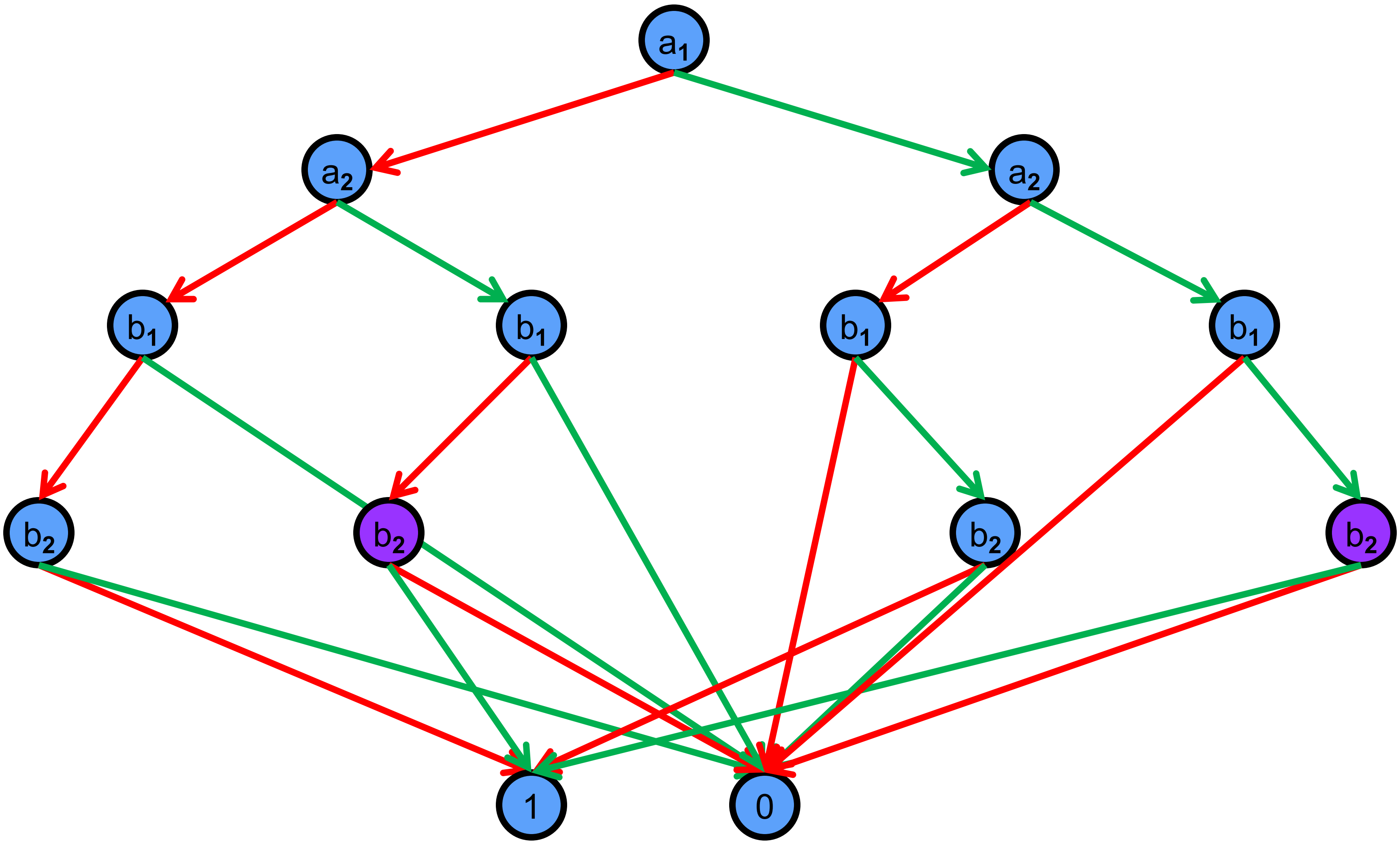


ROBDD and variable ordering

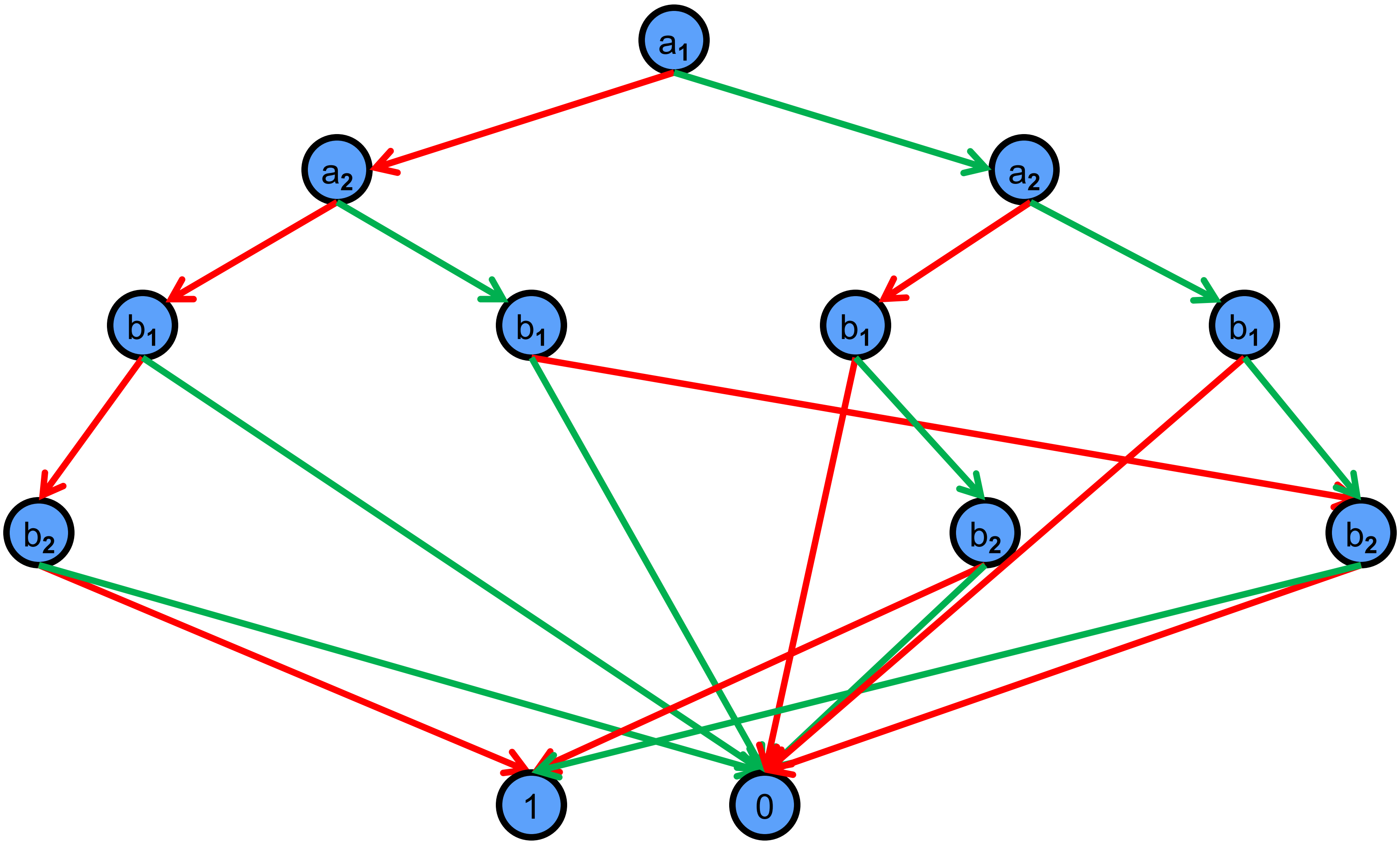


UNIVERSITY OF WATERLOO

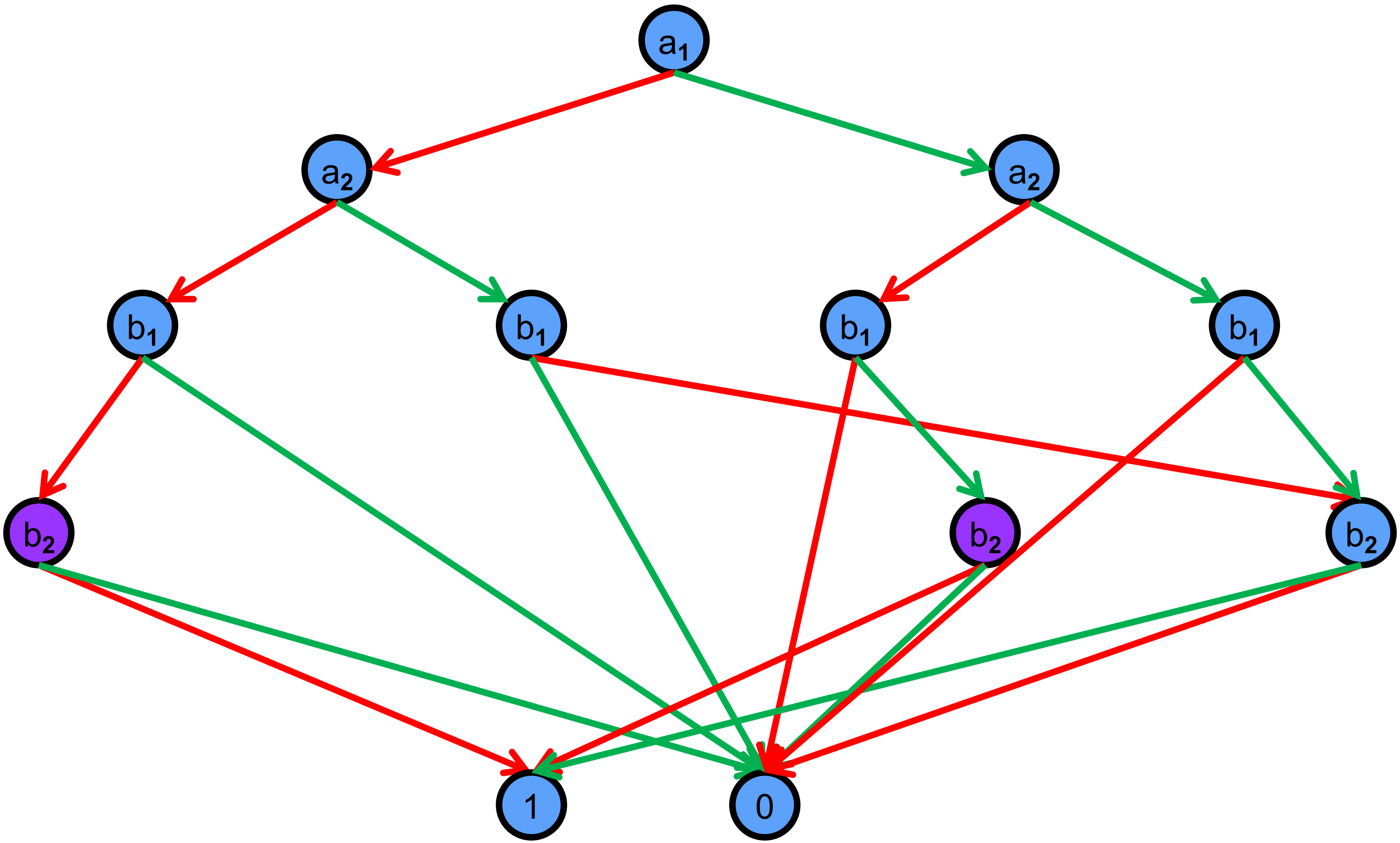
ROBDD and variable ordering



ROBDD and variable ordering

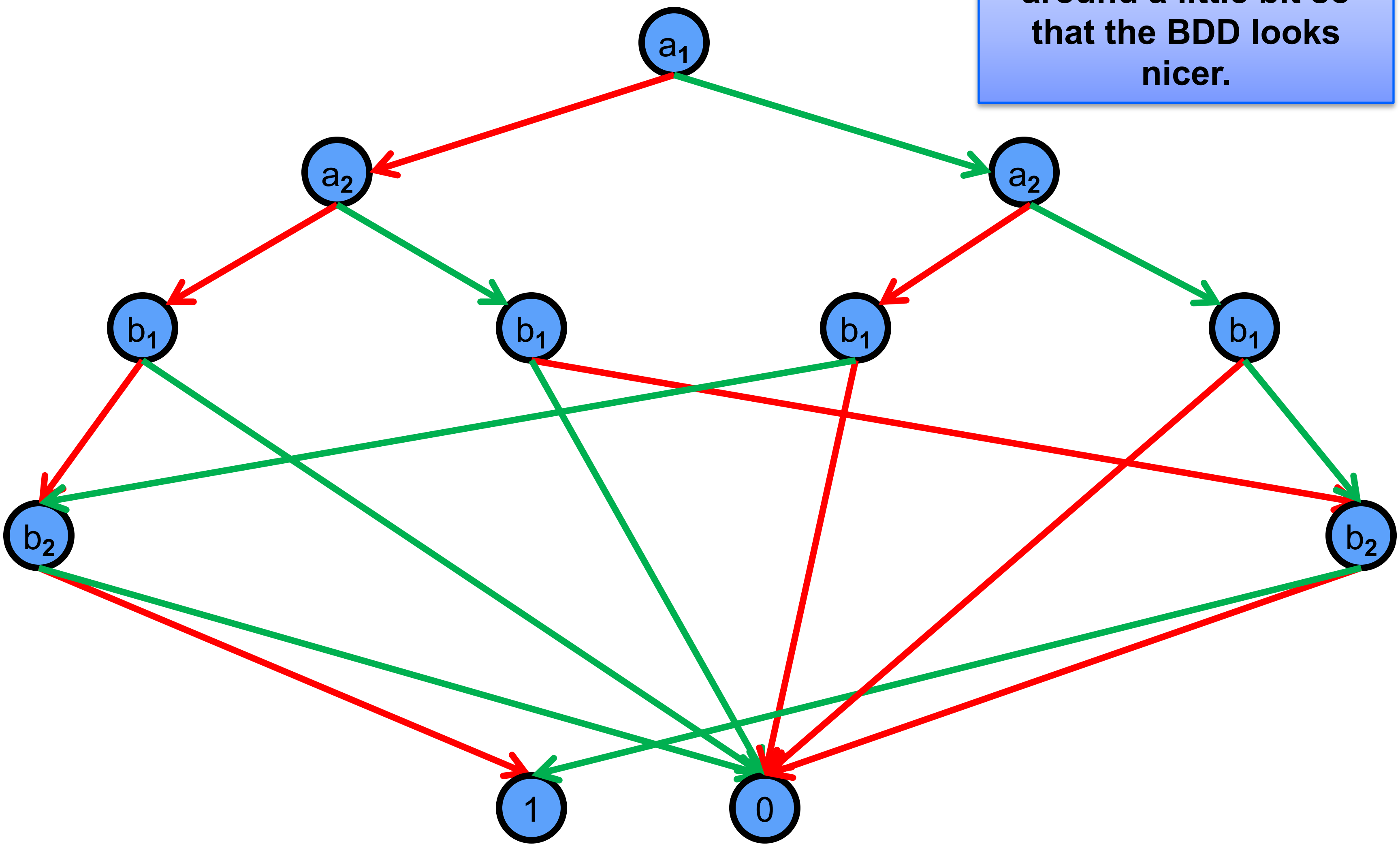


ROBDD and variable ordering



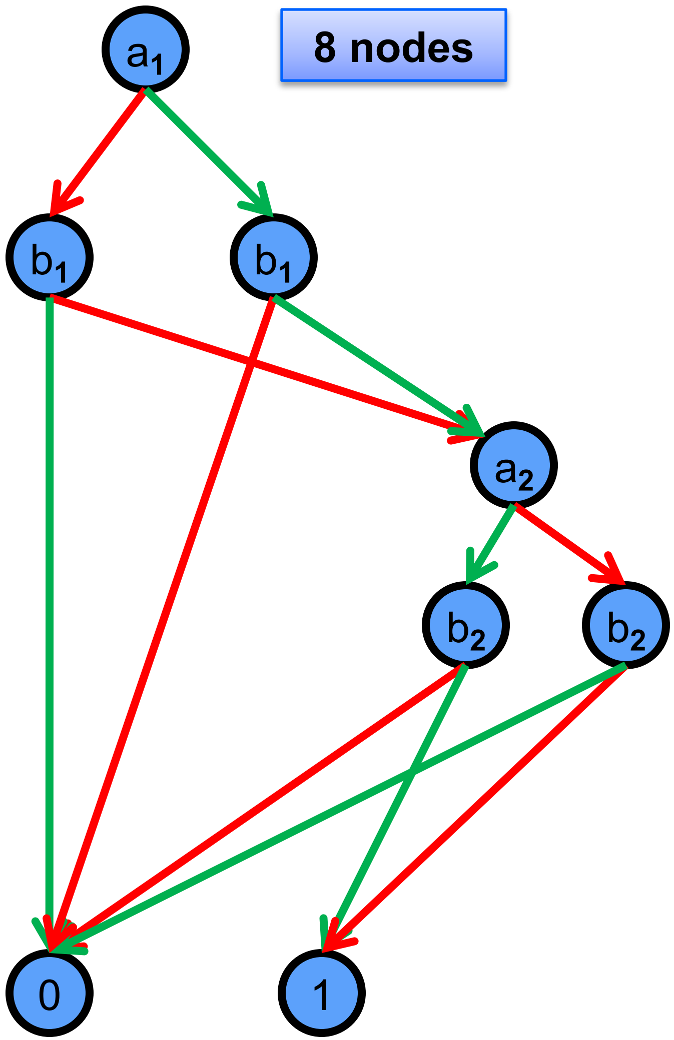
ROBDD and variable ordering

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



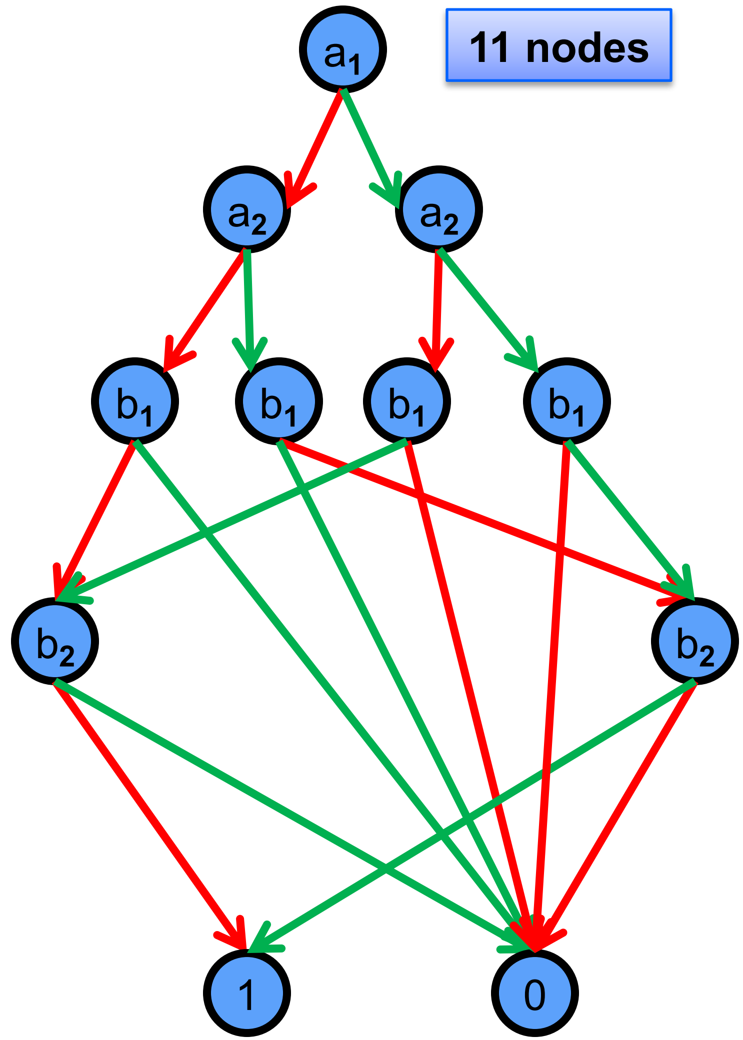
ROBDD and variable ordering

8 nodes



$a_1 < b_1 < a_2 < b_2$

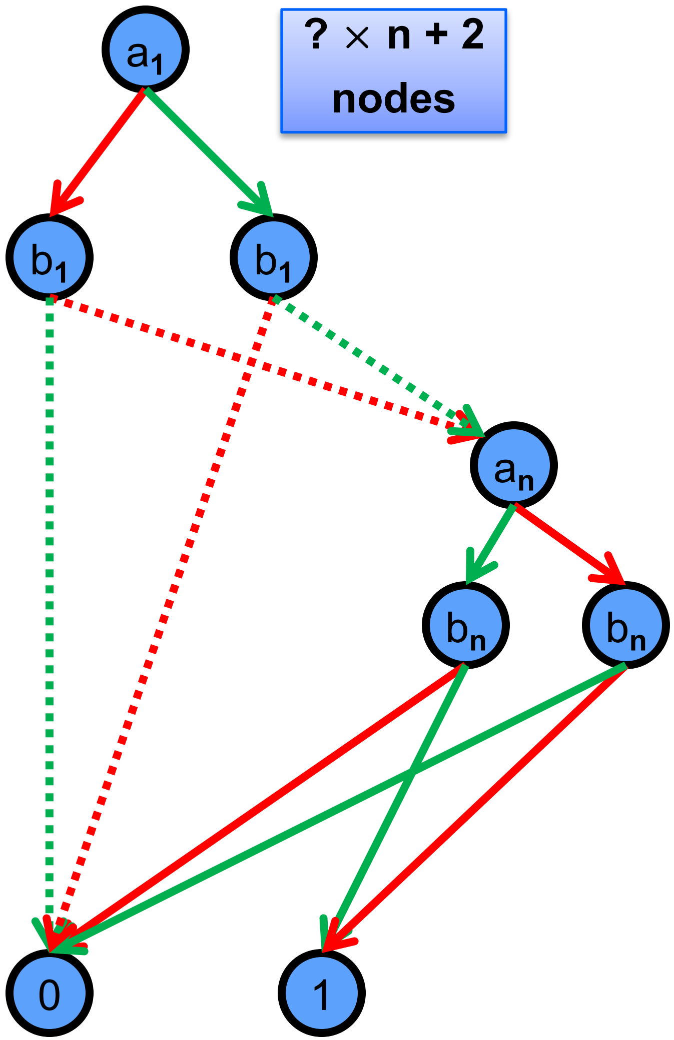
11 nodes



$a_1 < a_2 < b_1 < b_2$

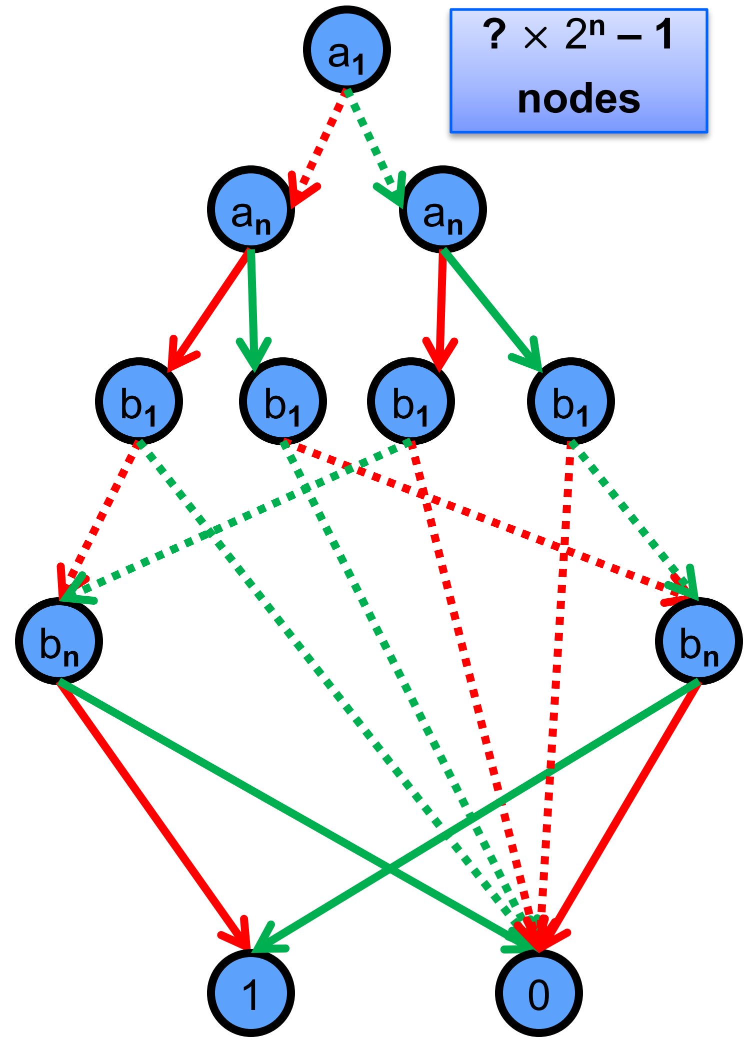
UNIVERSITY OF WATERLOO

ROBDD and variable ordering



? × n + 2 nodes

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



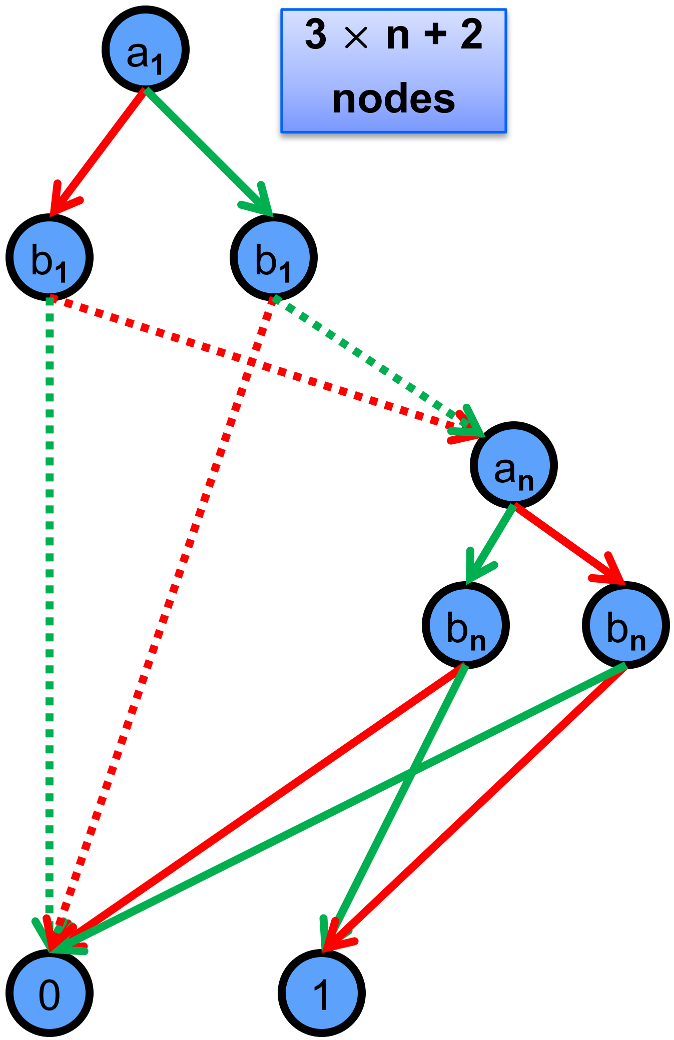
? × 2^n – 1 nodes

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

UNIVERSITY OF WATERLOO

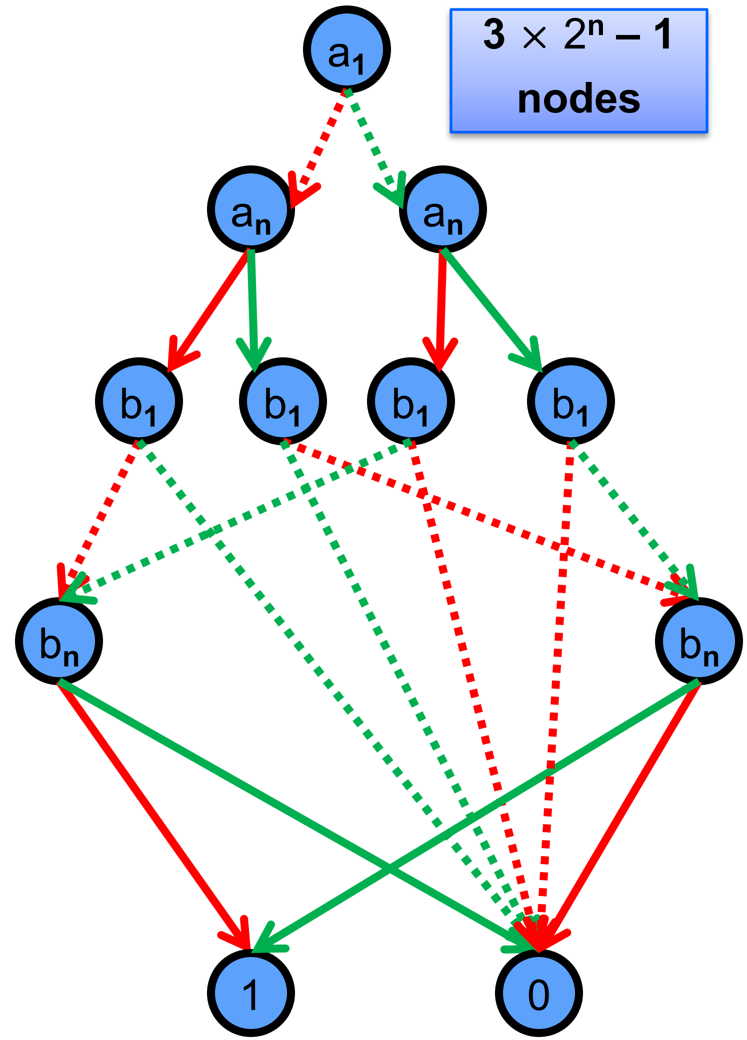
ROBDD and variable ordering

$3 \times n + 2$ nodes



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

$3 \times 2^n - 1$ nodes



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

BDD Operations

True ¬ BDD(TRUE)

False¬ BDD(FALSE)

Var ¬ v $\mapsto$ BDD(v)

Not ¬ BDD(f) $\mapsto$ BDD(¬f)

And ¬ BDD(f_1) $\times$ BDD(f_2) $\mapsto$ BDD($f_1 \wedge f_2$)

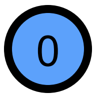
Or ¬ BDD(f_1) $\times$ BDD(f_2) $\mapsto$ BDD($f_1 \vee f_2$)

Exists ¬ BDD(f) $\times$ v $\mapsto$ BDD($\exists$ v. f)

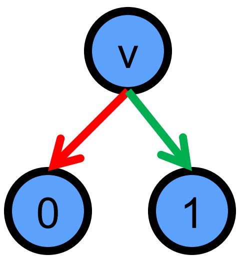
Basic BDD Operations

True

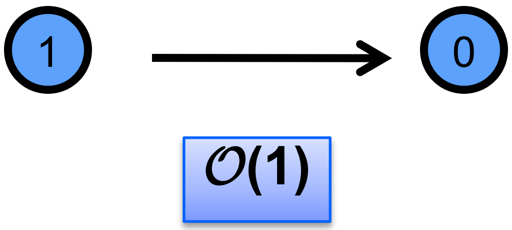
False

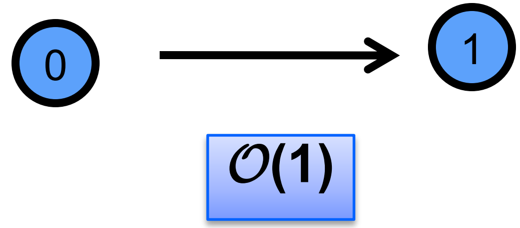


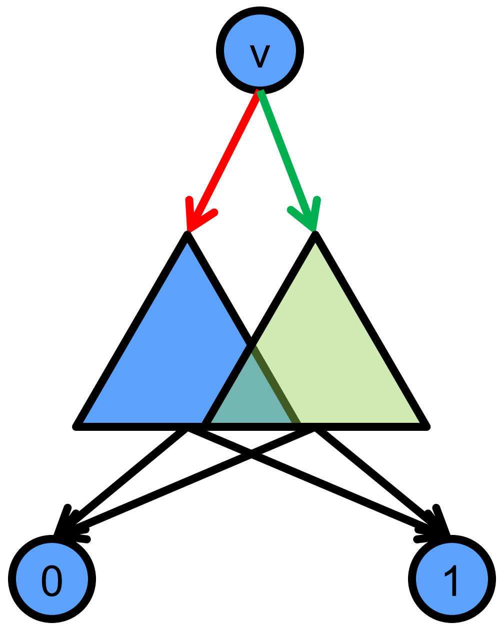
Var(v)



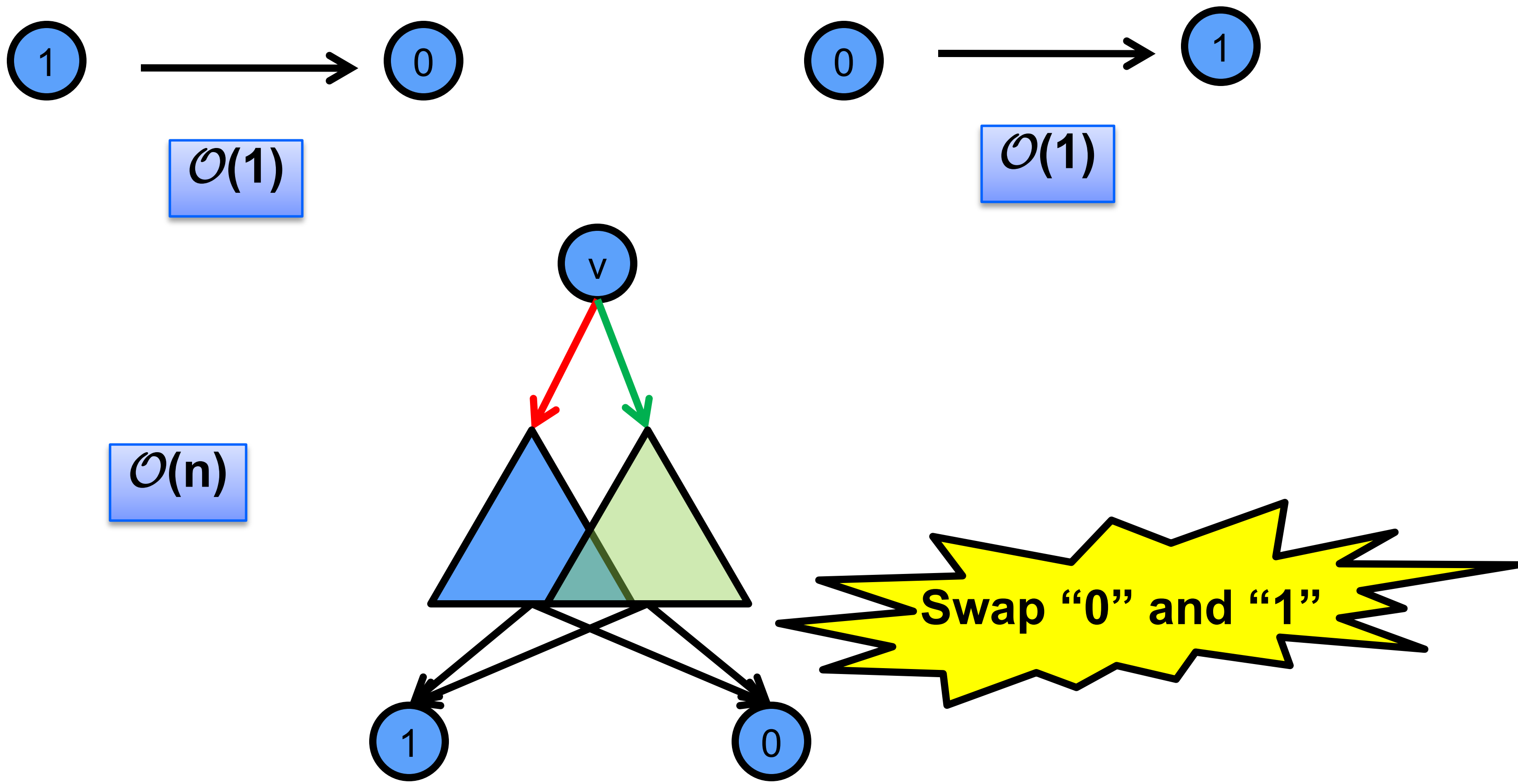
BDD Operations: Not



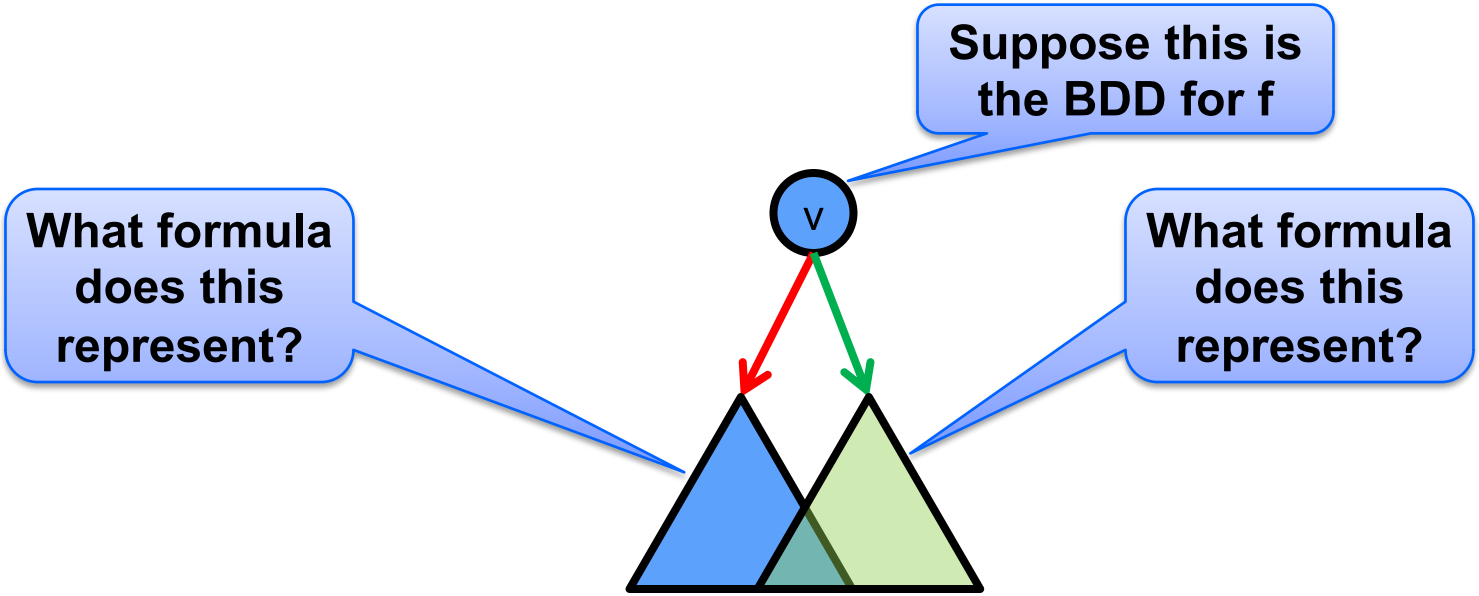




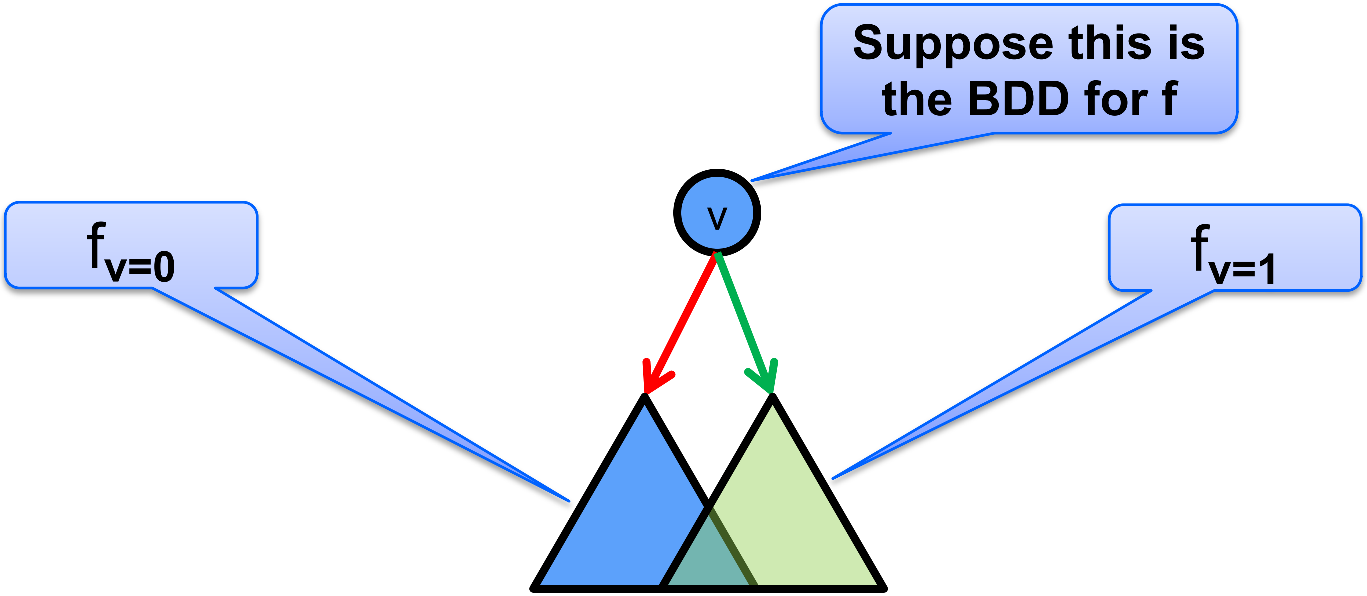
BDD Operations: Not



BDD Operations: And



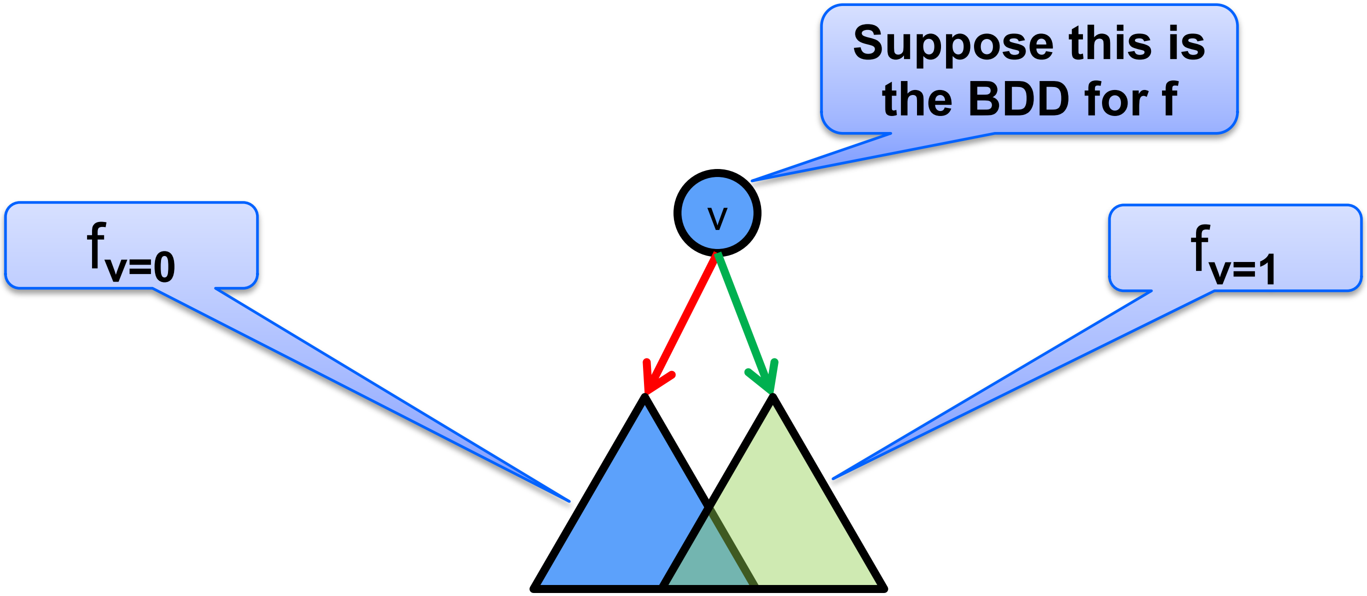
BDD Operations: And



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

$f = (X \wedge f_{v=0}) \vee (Y \wedge f_{v=1})$

BDD Operations: And



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

$f = (\neg v \wedge f_{v=0}) \vee (v \wedge f_{v=1})$

BDD Operations: And (Simple Cases)

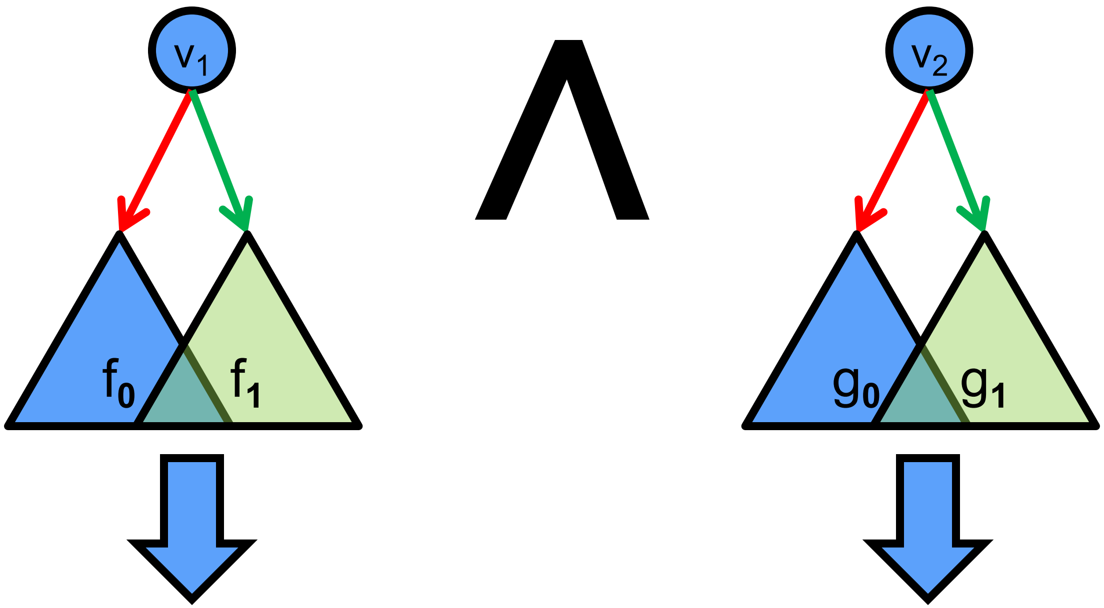
And (f, 0) = 0

And (f, 1) = f

And (1 ,f) = f

And (0 ,f) = 0

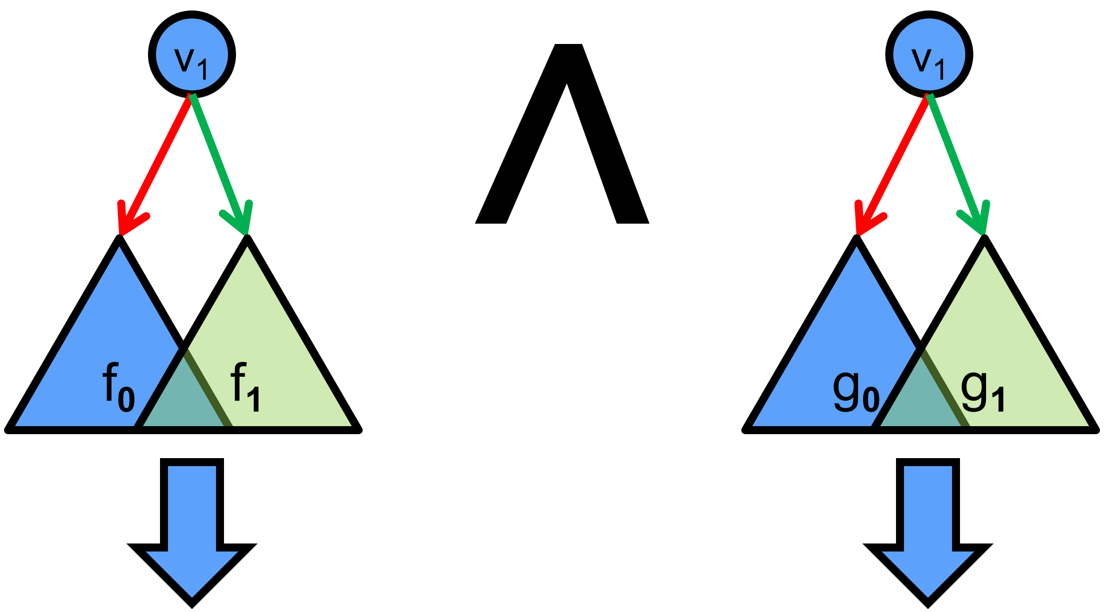
BDD Operations: And (Complex Case)



$(\neg v_1 \land f_0) \lor (v_1 \land f_1)$ $\land$ $(\neg v_2 \land g_0) \lor (v_2 \land g_1)$

BDD Operations: And (Complex Case 1)

$v_1 = v_2$



$(\neg v_1 \wedge f_0) \vee (v_1 \wedge f_1)$ $\wedge$ $(\neg v_1 \wedge g_0) \vee (v_1 \wedge g_1)$

BDD Operations: And (Complex Case 1)

$v_1 = v_2$

$(\neg v_1 \wedge X) \vee (v_1 \wedge Y)$

$(\neg v_1 \wedge f_0) \vee (v_1 \wedge f_1)$ $\wedge$ $(\neg v_1 \wedge g_0) \vee (v_1 \wedge g_1)$

BDD Operations: And (Complex Case 1)

$v_1 = v_2$

Compute recursively

$(\neg v_1 \wedge (f_0 \wedge g_0)) \vee (v_1 \wedge (f_1 \wedge g_1))$

↑

$(\neg v_1 \wedge f_0) \vee (v_1 \wedge f_1)$ $\wedge$ $(\neg v_1 \wedge g_0) \vee (v_1 \wedge g_1)$

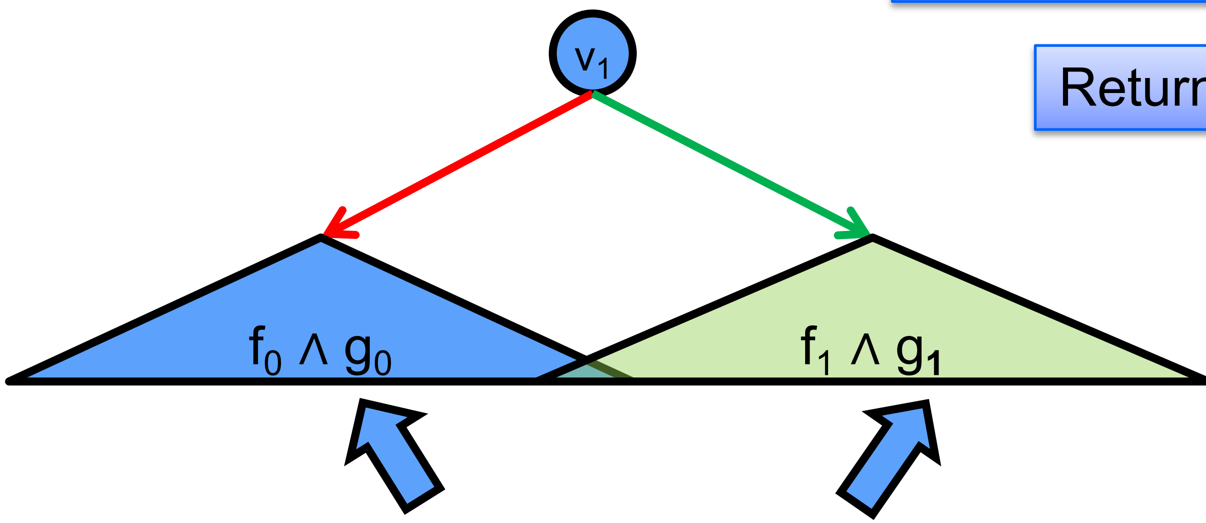
UNIVERSITY OF WATERLOO

BDD Operations: And (Complex Case 1)

$v_1 = v_2$

What if $f_0 \wedge g_0 = f_1 \wedge g_1$?

Return $f_0 \wedge g_0$



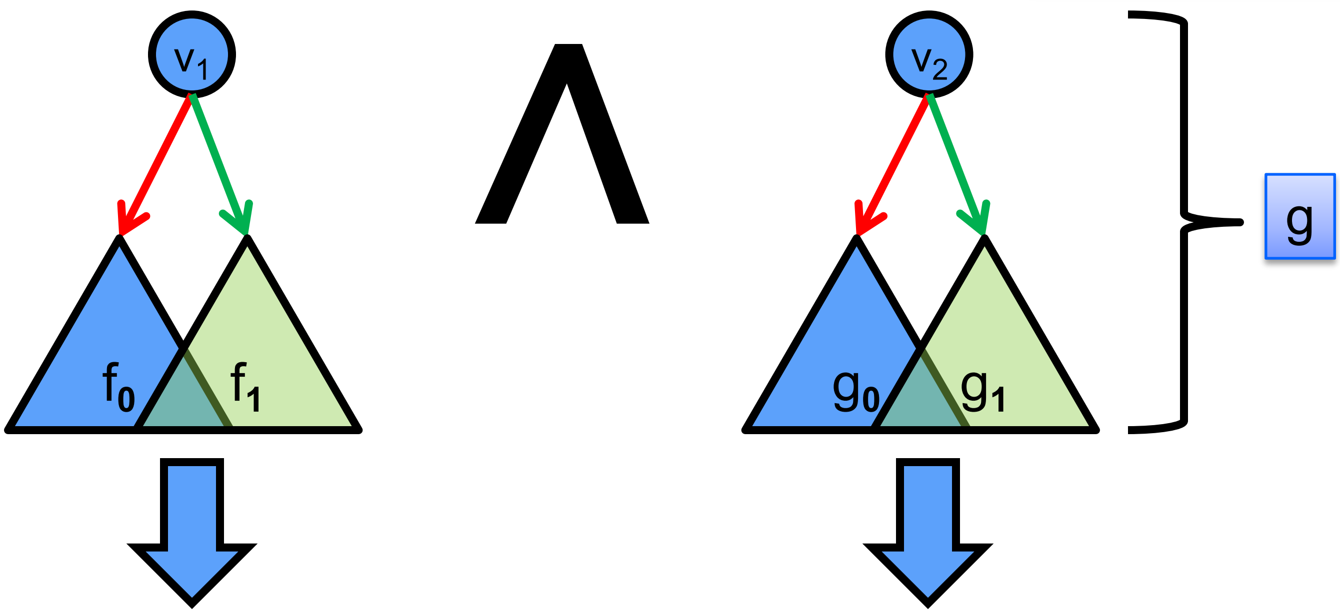
$(\neg v_1 \wedge (f_0 \wedge g_0)) \vee (v_1 \wedge (f_1 \wedge g_1))$

$(\neg v_1 \wedge f_0) \vee (v_1 \wedge f_1)$ $\wedge$ $(\neg v_1 \wedge g_0) \vee (v_1 \wedge g_1)$

BDD Operations: And (Complex Case 2)

$v_1 < v_2$

v_1 appears before v_2 in the variable ordering



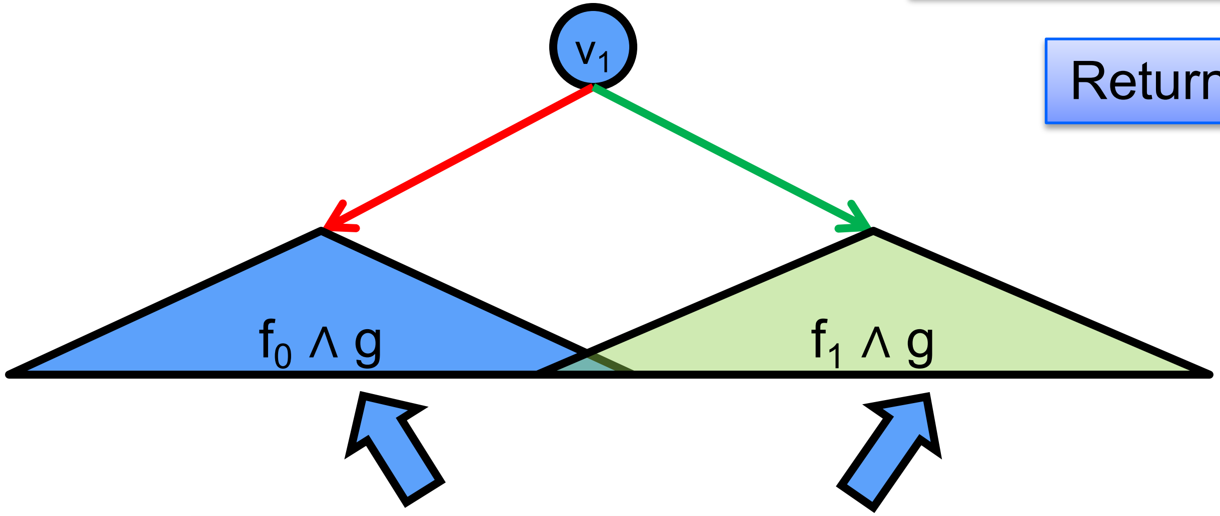
$(\neg v_1 \land f_0) \lor (v_1 \land f_1) \quad \land \quad (\neg v_2 \land g_0) \lor (v_2 \land g_1)$

BDD Operations: And (Complex Case 2)

$v_1 < v_2$

What if $f_0 \wedge g = f_1 \wedge g$?

Return $f_0 \wedge g$



$(\neg v_1 \wedge (f_0 \wedge g)) \vee (v_1 \wedge (f_1 \wedge g))$

$(\neg v_1 \wedge f_0) \vee (v_1 \wedge f_1)$ $\wedge$ g

$\mathcal{O}(n_1 \times n_2)$

UNIVERSITY OF WATERLOO

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**)))**

$\mathcal{O}(n_1 \times n_2)$

BDD Operations: Exists

Exists(“0”,v) = ?

UNIVERSITY OF WATERLOO

BDD Operations: Exists

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

Exists("1",v) = ?

BDD Operations: Exists

$$\text{Exists}(\text{“0”}, v) = \text{“0”}$$

$$\text{Exists}(\text{“1”}, v) = \text{“1”}$$

$$\text{Exists}((\neg v \land f) \lor (v \land g), v) = ?$$

BDD Operations: Exists

$$\text{Exists}(``0", v) = ``0"$$

$$\text{Exists}(``1", v) = ``1"$$

$$\text{Exists}((\neg v \wedge f) \vee (v \wedge g)\ ,\ v) = \text{Or}(f,g)$$

$$\text{Exists}((\neg v' \wedge f) \vee (v' \wedge g)\ ,\ v) = ?$$

BDD Operations: Exists

$\mathcal{O}(n^2)$

$$\text{Exists}(\text{“0”},v) = \text{“0”}$$

$$\text{Exists}(\text{“1”},v) = \text{“1”}$$

$$\text{Exists}((\neg v \wedge f) \vee (v \wedge g), v) = \text{Or}(f,g)$$

$$\text{Exists}((\neg v' \wedge f) \vee (v' \wedge g), v) =$$

$$(\neg v' \wedge \text{Exists}(f,v)) \vee (v' \wedge \text{Exists}(g,v))$$

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

Because the BDD size changes!

UNIVERSITY OF WATERLOO

BDD Applications

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

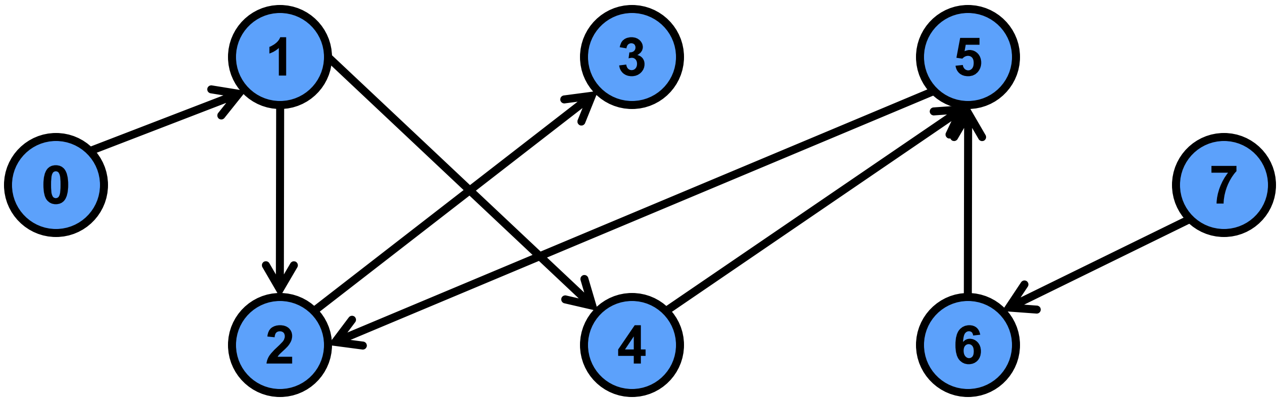
BDDs are great if you are interested in the set of all solutions

- 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

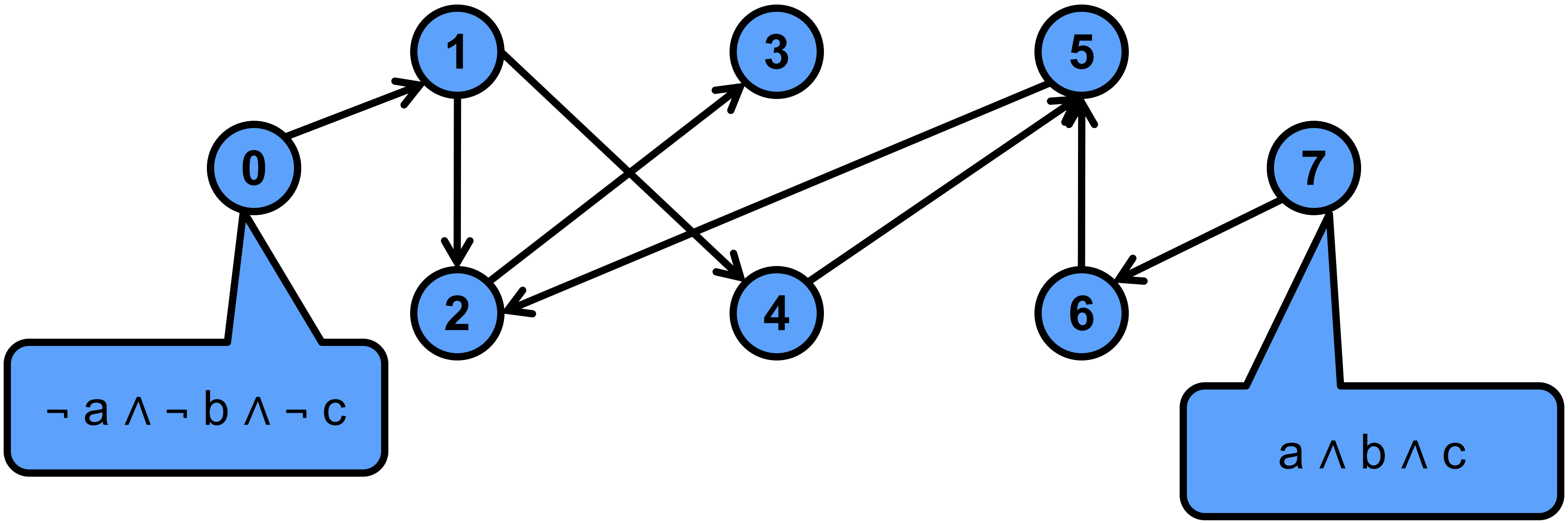


Which nodes are reachable from “7”?

{2,3,5,6,7}

But what if the graph has trillions of nodes?

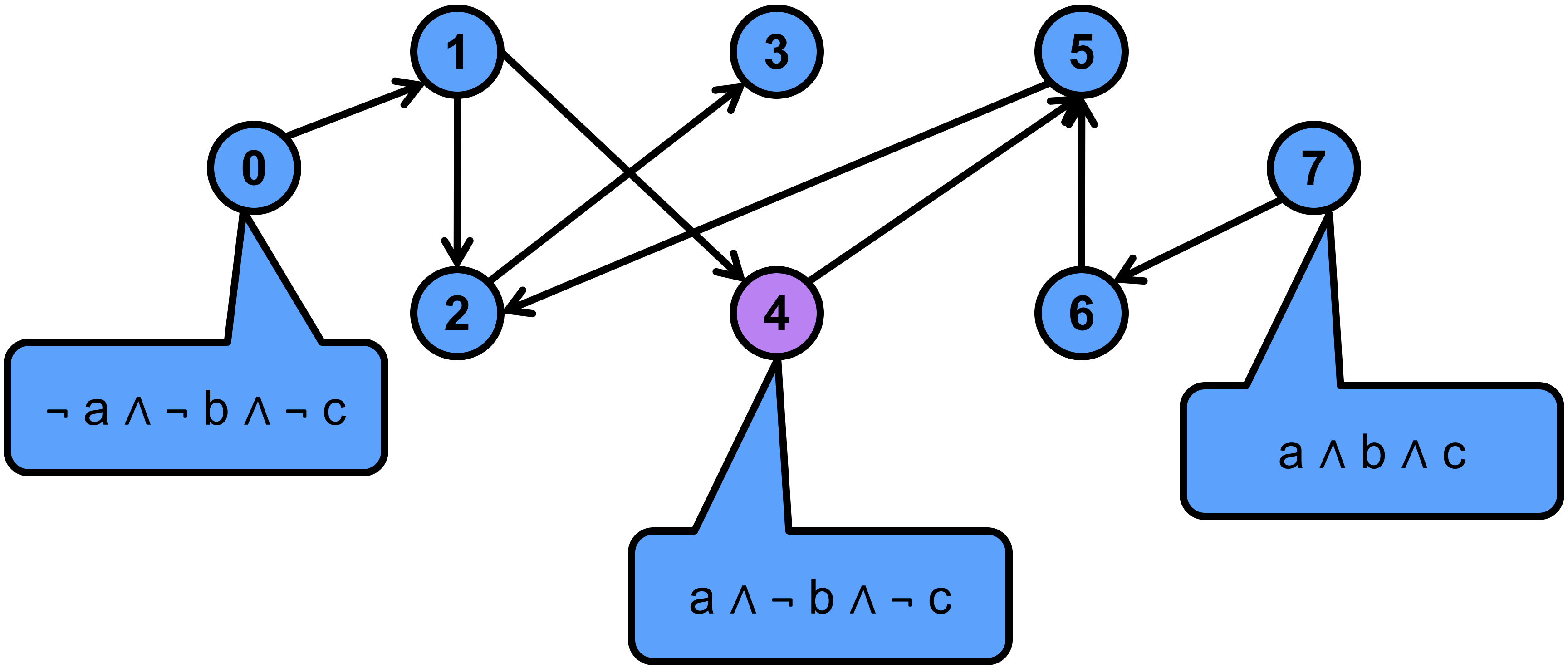
Graph Reachability



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

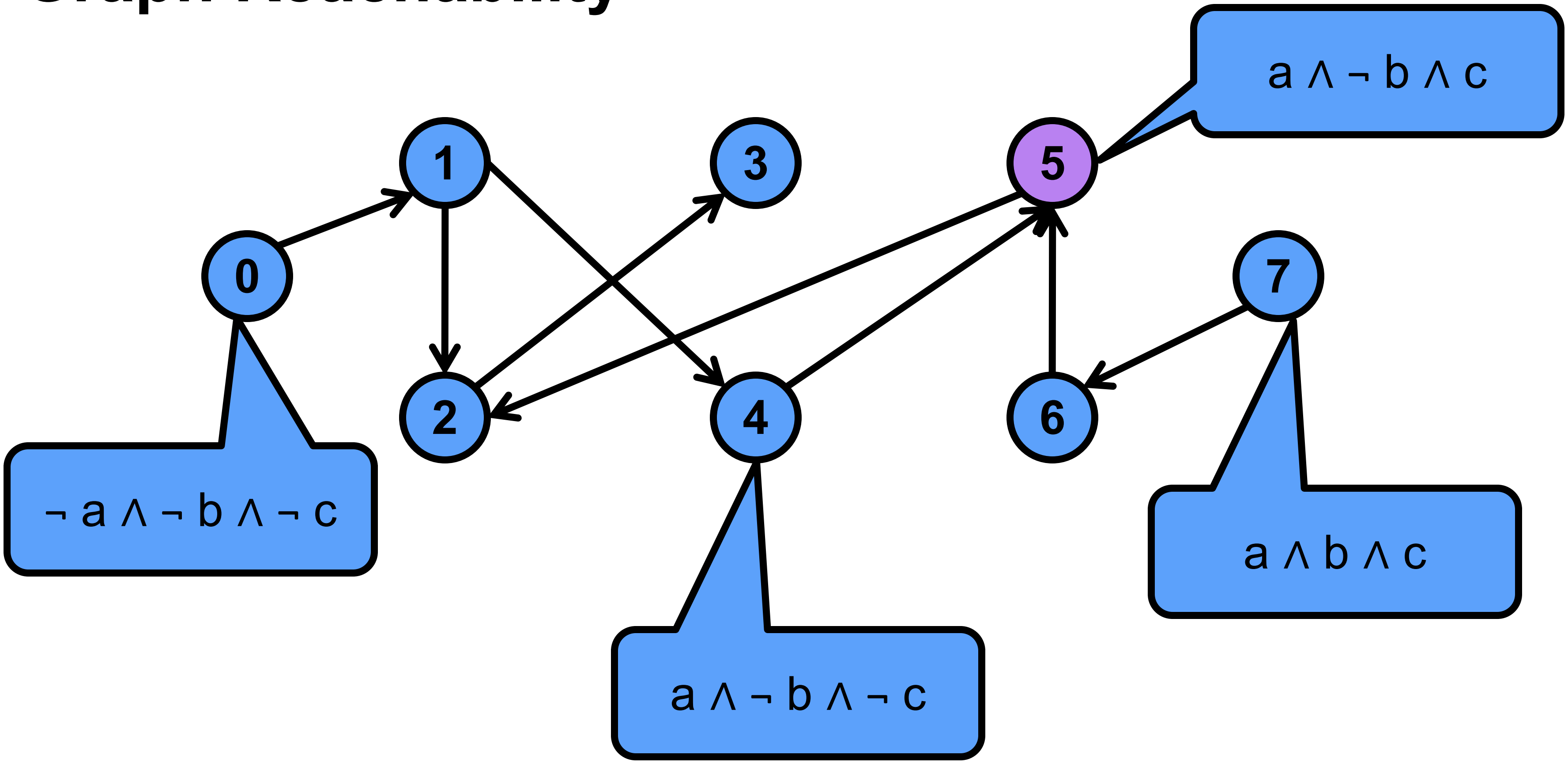
UNIVERSITY OF WATERLOO

Graph Reachability



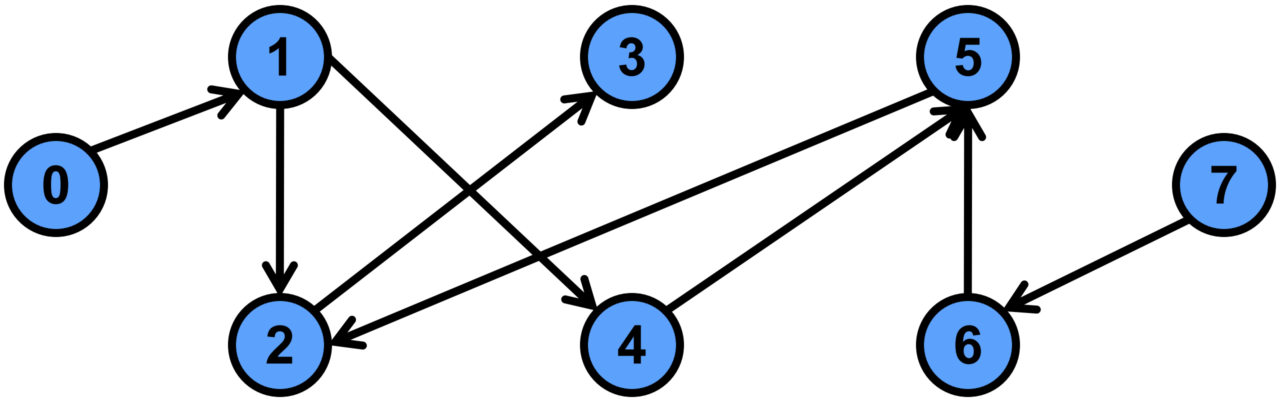
Use three Boolean variables (a,b,c) to encode each node?

Graph Reachability



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

Graph Reachability

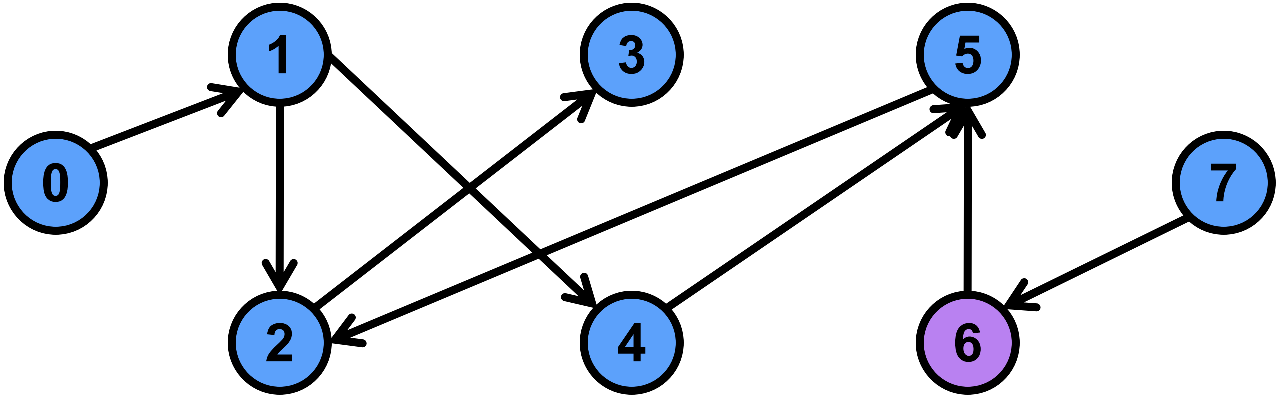


a ∧ b ∧ ¬ c = ?

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

The nodes whose encodings satisfy the formula.

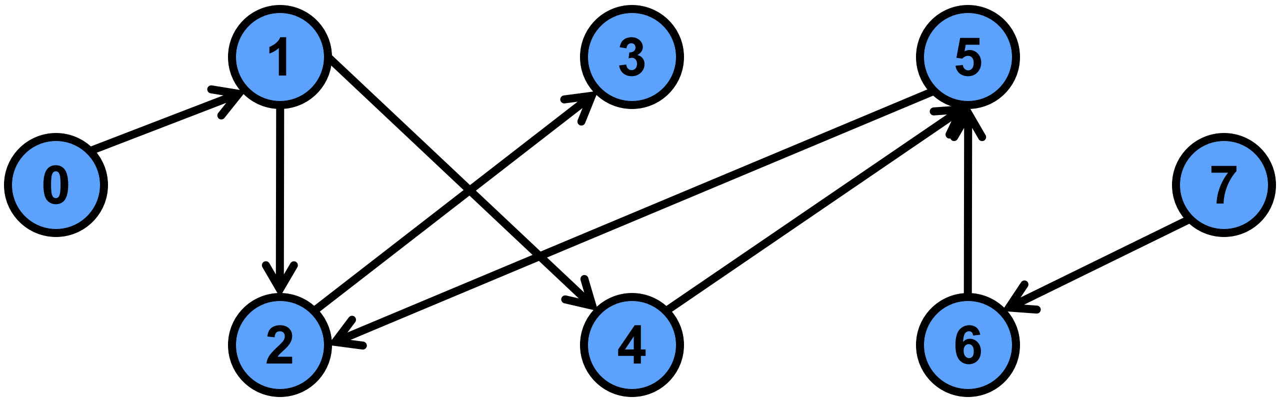
Graph Reachability



$a \wedge b \wedge \neg c = \{6\}$

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

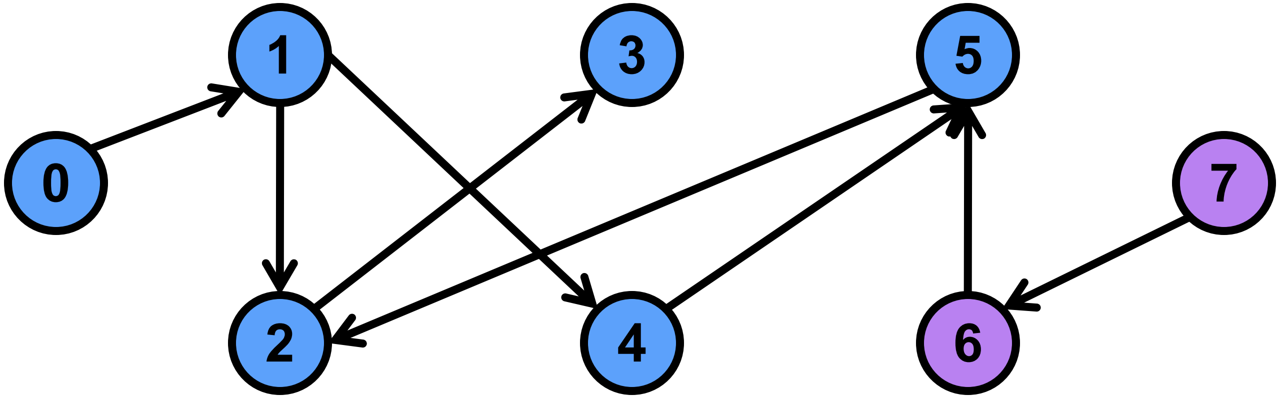
Graph Reachability



$a \wedge b = \ ?$

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

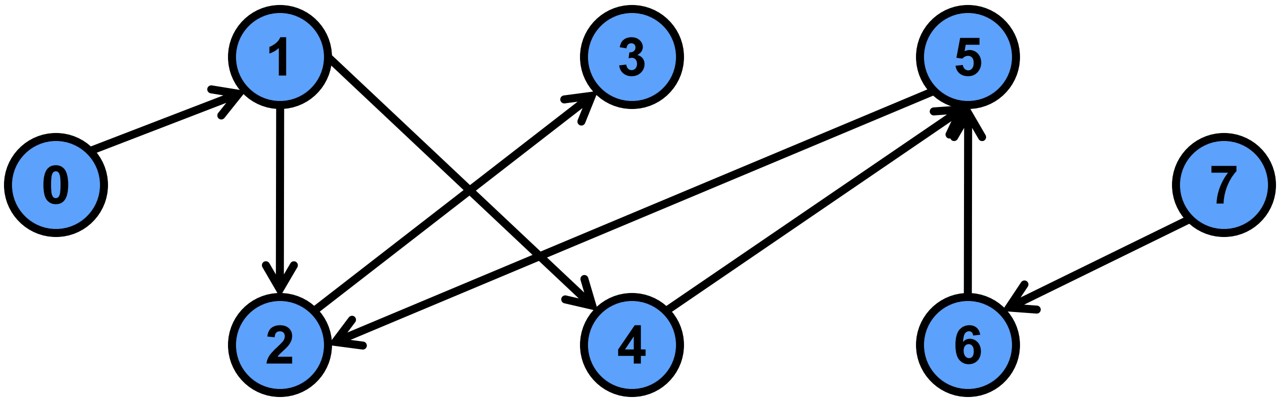
Graph Reachability



a ∧b = {6,7}

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

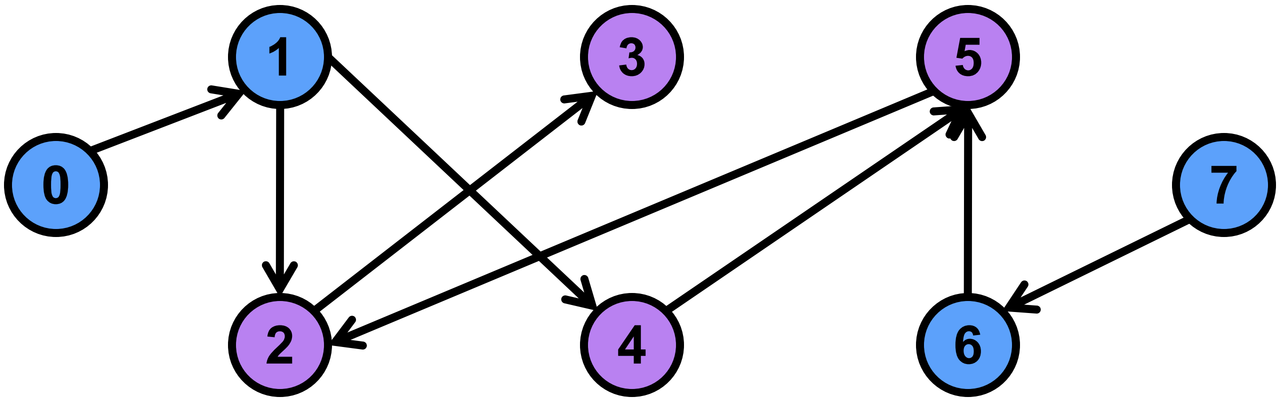
Graph Reachability



a xor b = ?

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

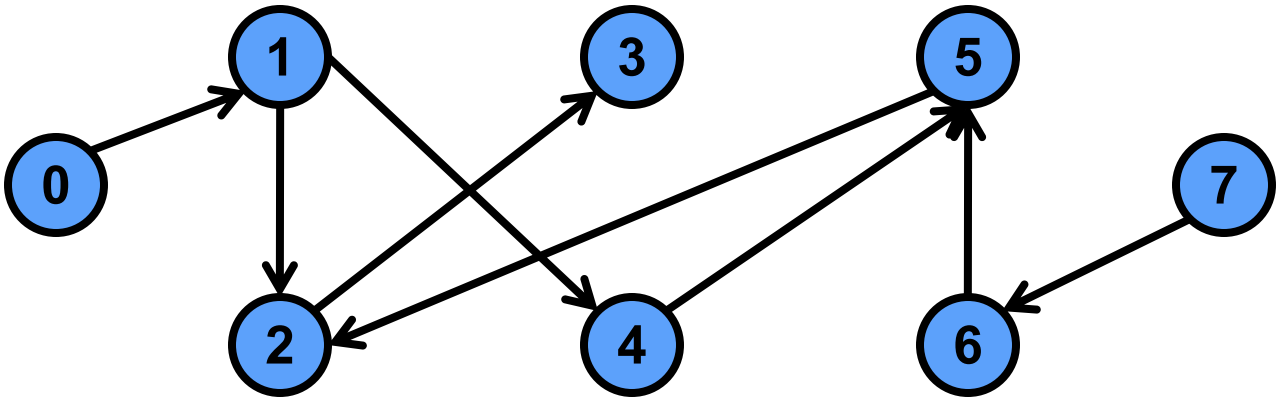
Graph Reachability



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

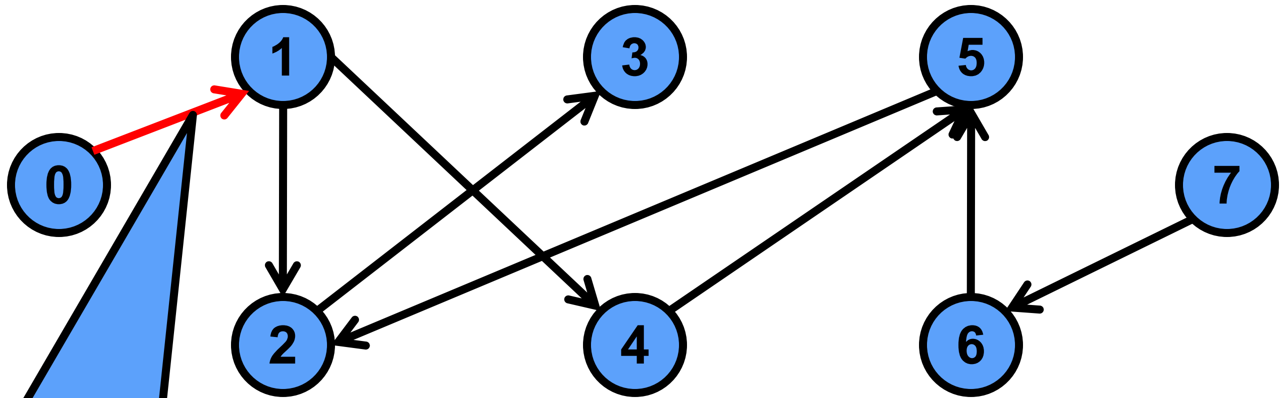
Key Idea 1: Every Boolean formula represents a set of nodes!

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 *Φ* represents all pairs of nodes (n,n') that satisfy *Φ* when n is encoded using (a,b,c) and n' is encoded using (a',b',c')

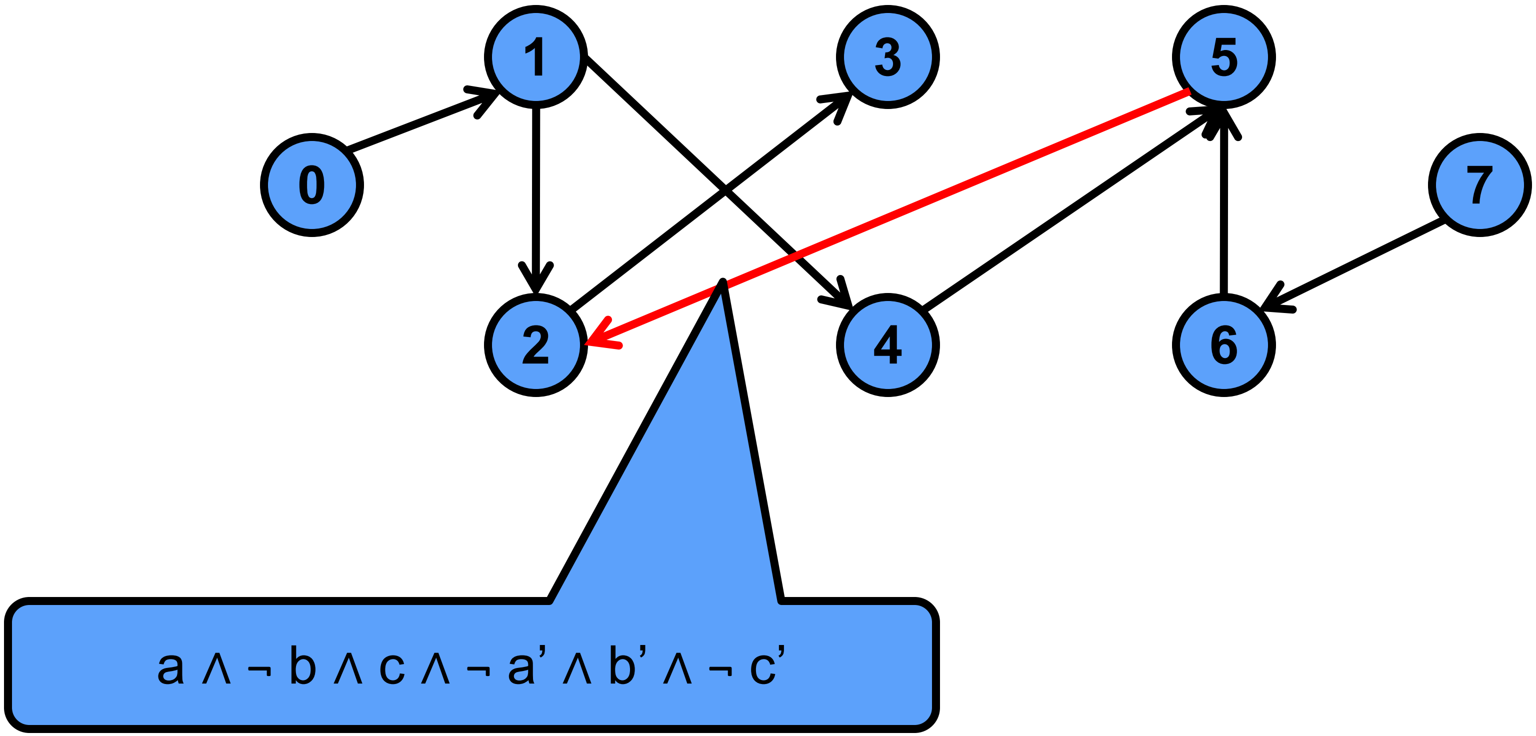
Graph Reachability



$\neg a \land \neg b \land \neg c \land \neg a' \land \neg b' \land c'$

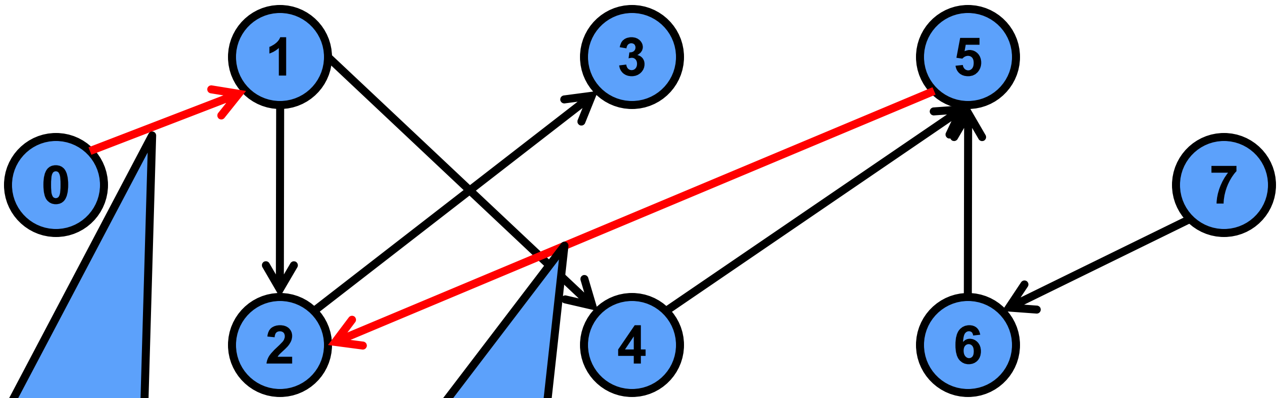
Key Idea 2: Edges can also be represented by Boolean formulas

Graph Reachability



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

Graph Reachability



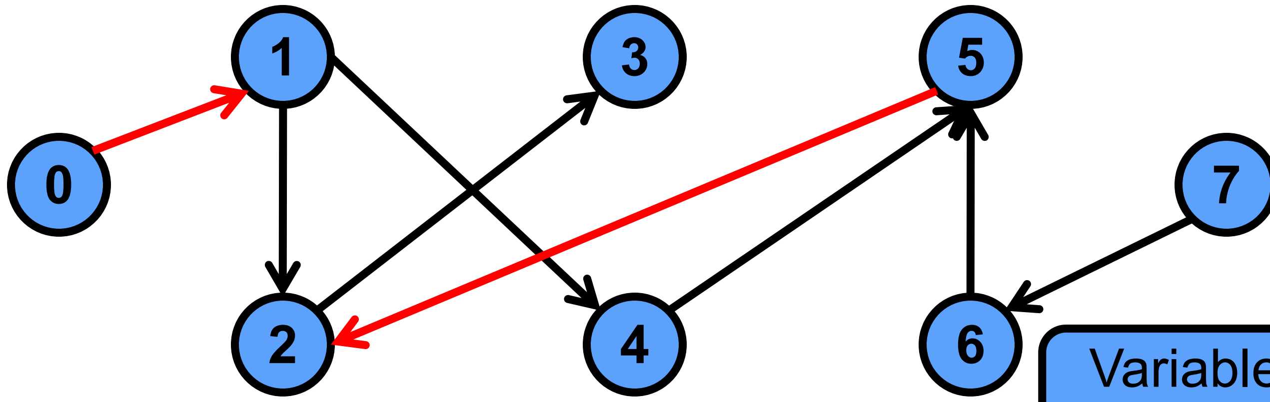
$a \wedge \neg b \wedge c \wedge \neg a' \wedge b' \wedge \neg c'$

$\vee$

$\neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge \neg b' \wedge c'$

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

Graph Reachability



Variable renaming ¬ replace a' with a

Image(S,R) =

$$(\exists\, a,b,c \,.\, (S \wedge R))\,[\, a \backslash a',\ b \backslash b',\ c \backslash c' \,]$$

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

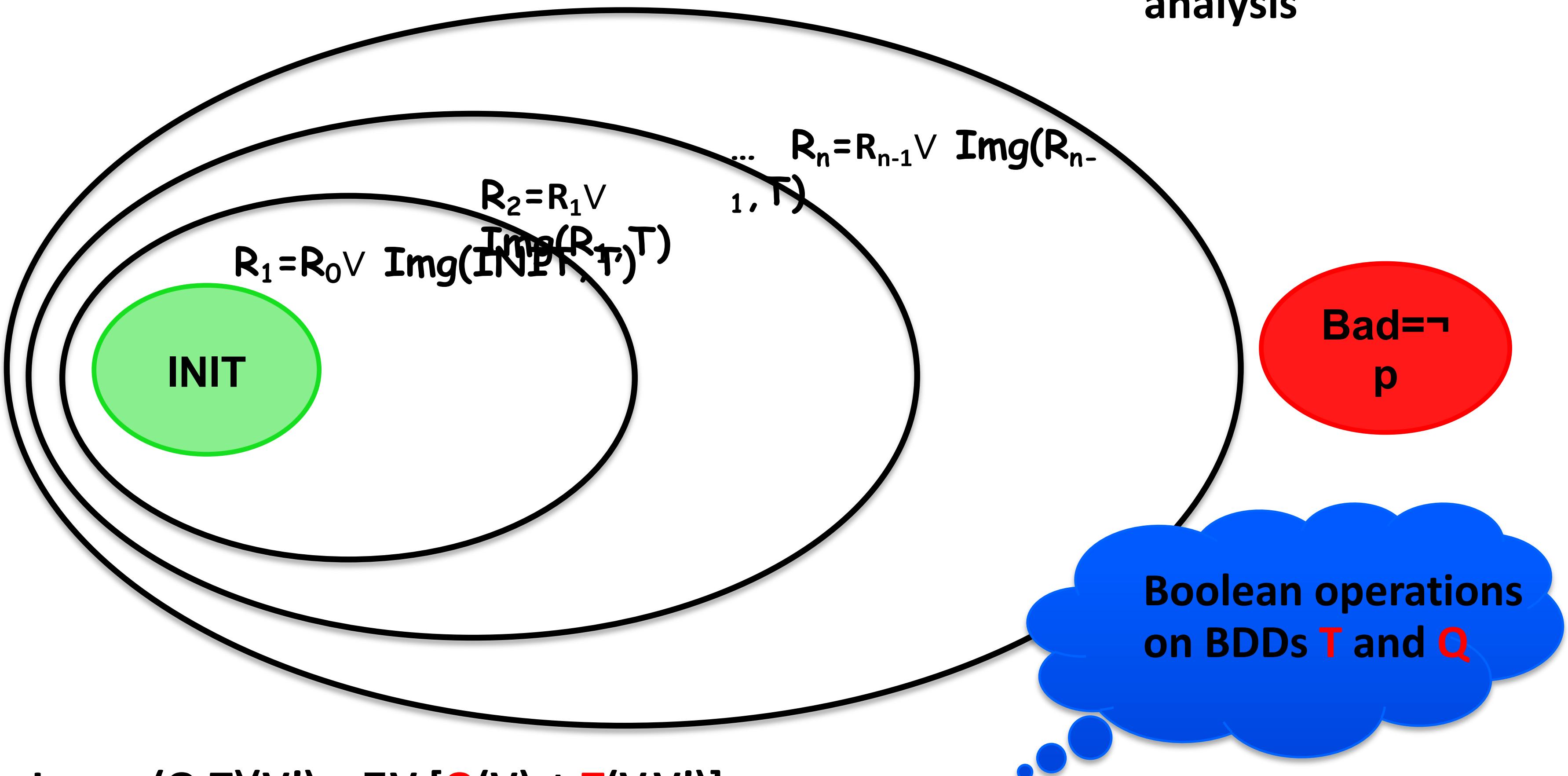
```
S = BDD for initial set of nodes;
R = BDD for all the edges of the graph;

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.

Forward Reachability Analysis with BDDs

Does AG p hold?

All safety properties reduce to reachability analysis



$$Image(Q,T)(V') = \exists V\,[Q(V) \wedge T(V,V')]$$