Binary Decision Diagrams and Symbolic Model Checking

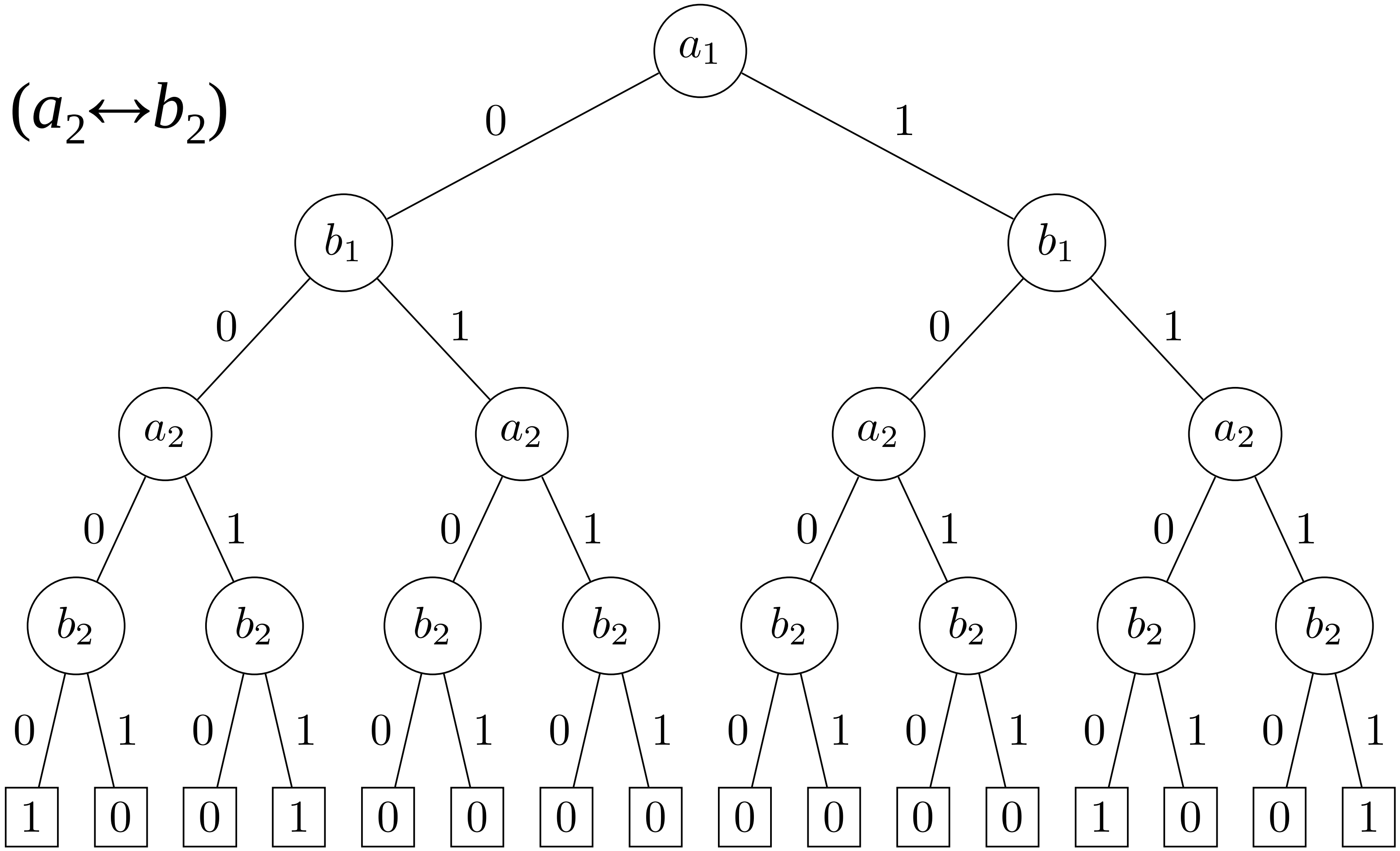
Chapter 8

Binary Decision Diagrams and Symbolic Model Checking

Representing Boolean Formulas

Binary Decision Trees

$f = (a_1 \leftrightarrow b_1) \land (a_2 \leftrightarrow b_2)$



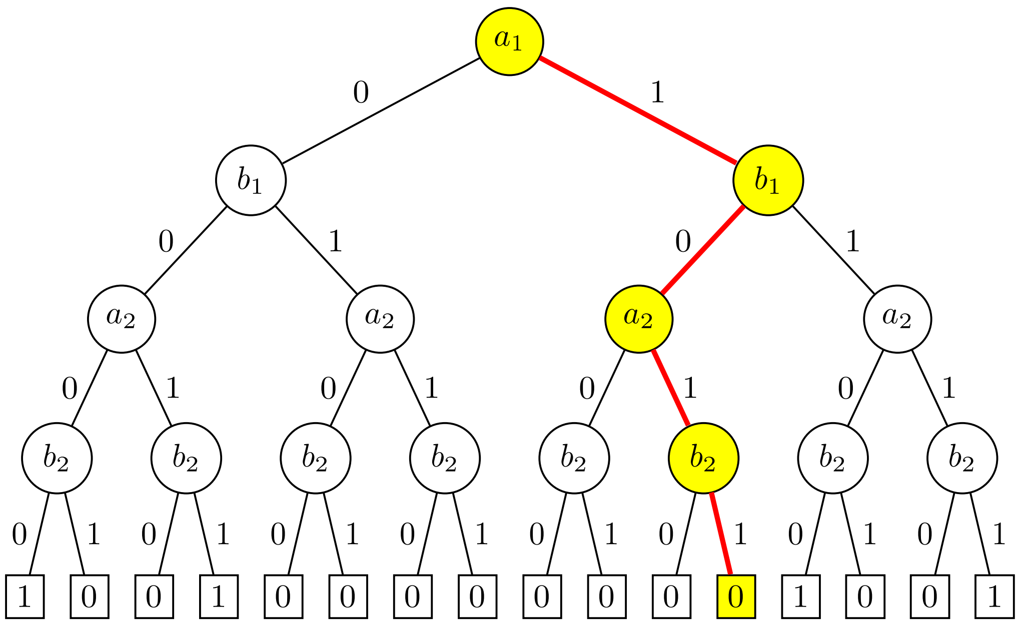
Binary Decision Trees

- Graphical representation for a Boolean formula
- Two kinds of nodes: nonterminals $\textcircled{a_1}$ and terminals $\boxed{1}$
- Nonterminal nodes are labeled with a variable, edges are labeled with 0 or 1
- One node is the root

Semantics of Binary Decision Trees

1. Start at the root
2. Consider the variable in the node
3. Follow the edge that assigns the desired truth value to the variable

The value of the function is given by the terminal



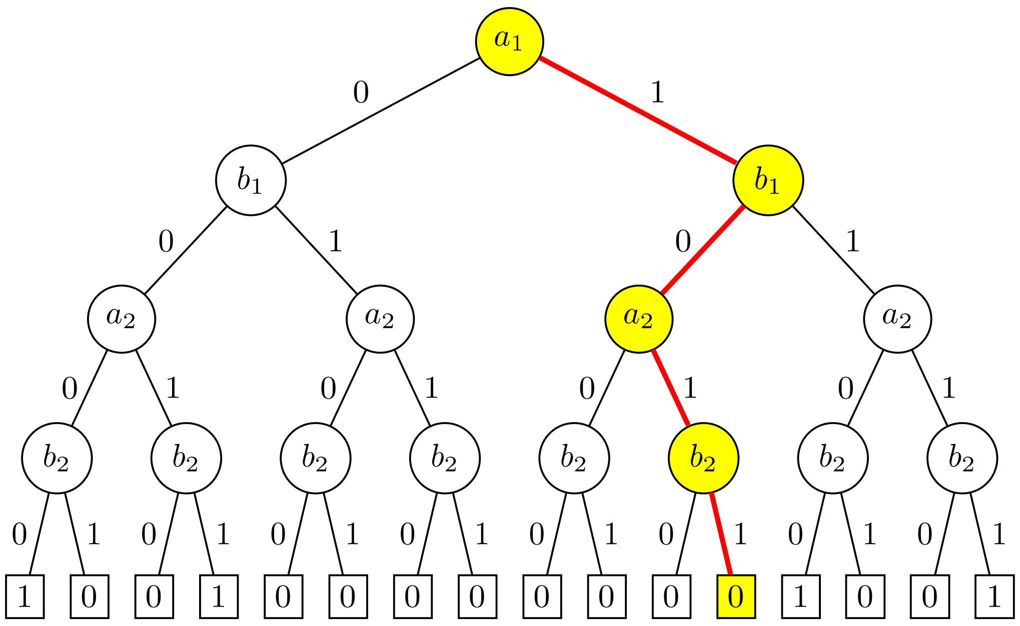
$$f = 0$$

for $a_1 \mapsto 1, b_1 \mapsto 0, a_2 \mapsto 1, b_2 \mapsto 1$

Variable Ordering in Binary Decision Trees

Observe that the ordering of the variables on all paths from the root to a terminal is the same

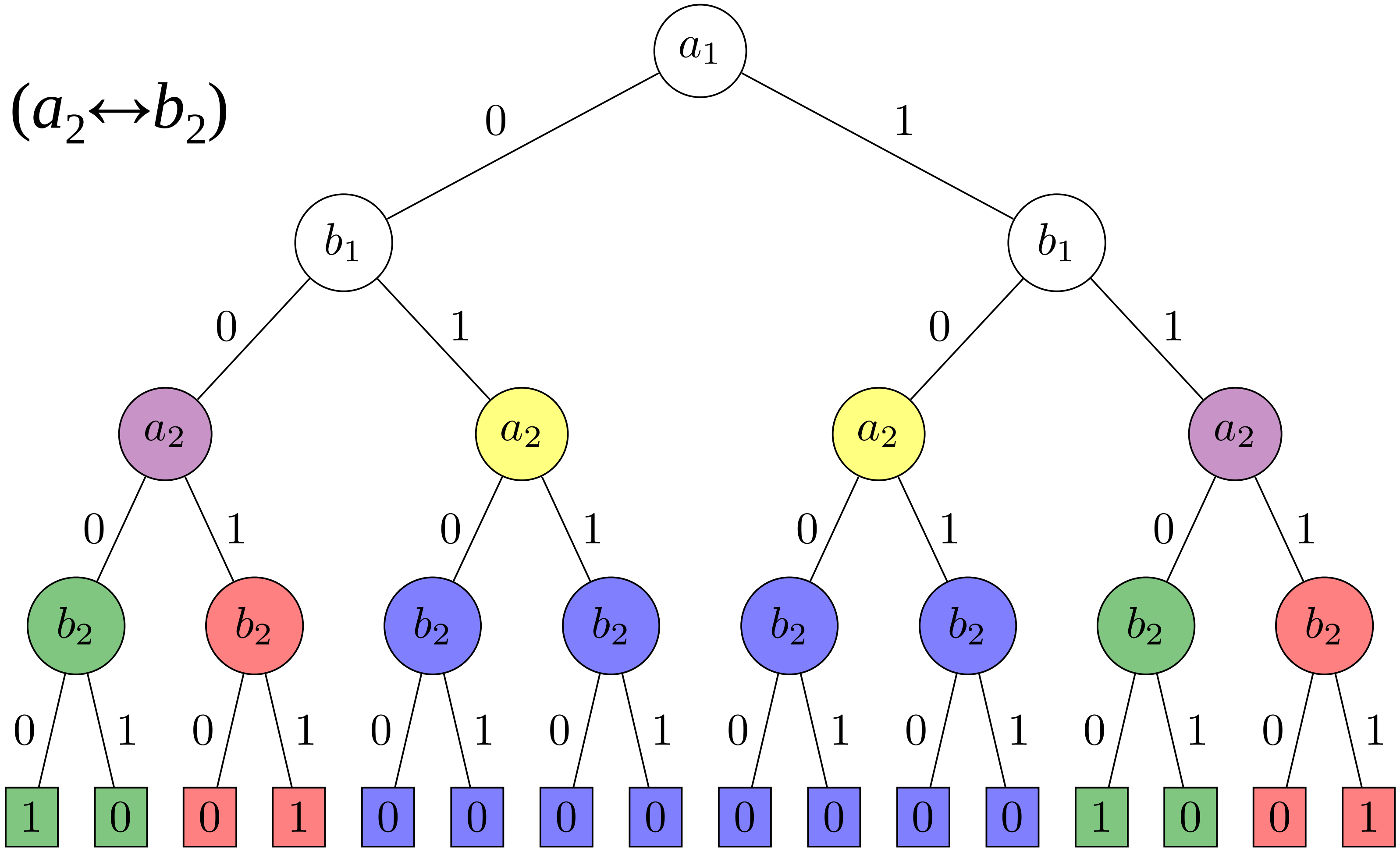
The decision tree is ordered



$$a_1 < b_1 < a_2 < b_2$$

Redundancy in Binary Decision Trees

$f = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$

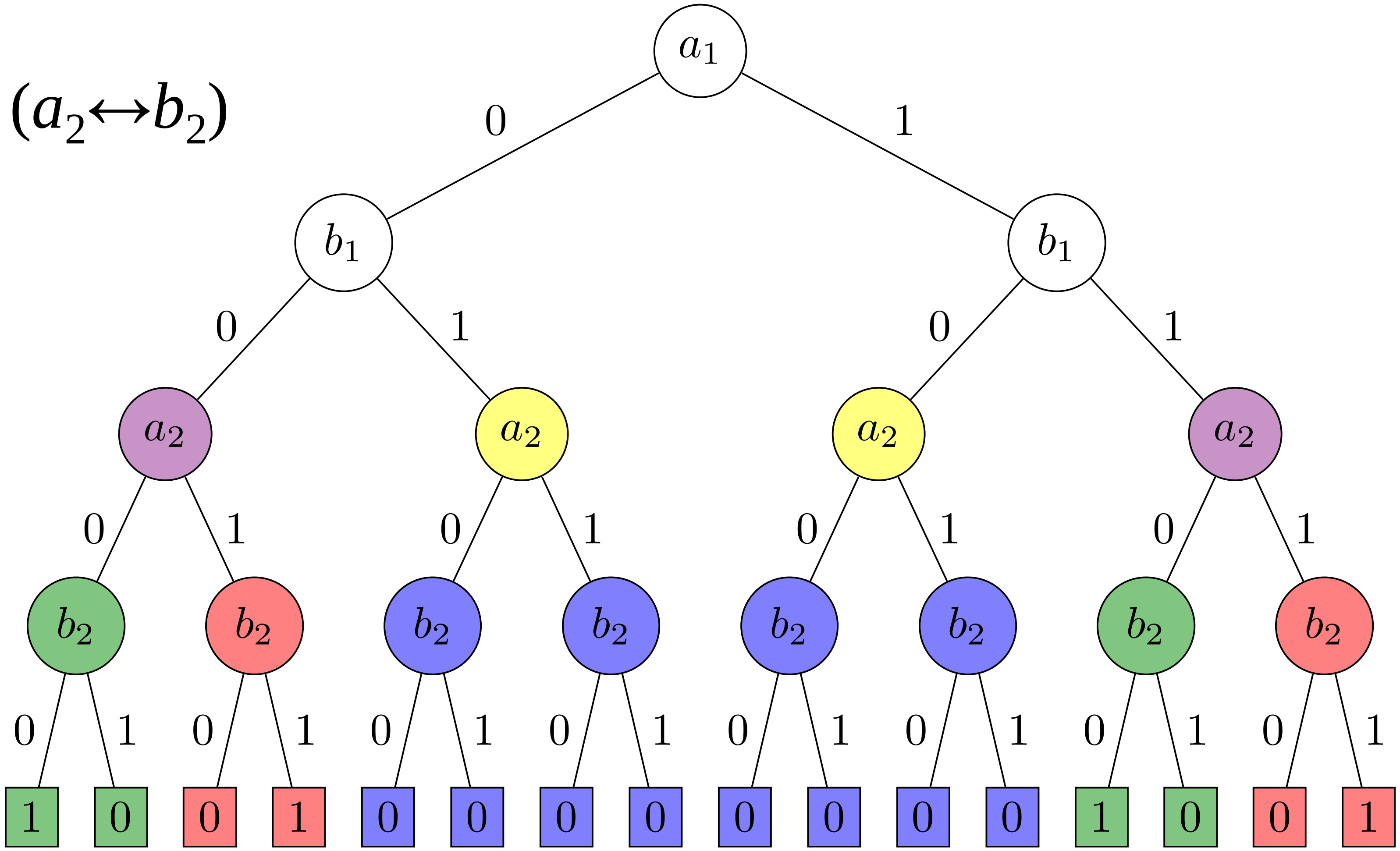


Sharing Tree Nodes

- Decision trees can be highly redundant
- Idea: identify parts of the tree that are identical
- Replace a duplicate by a pointer or reference

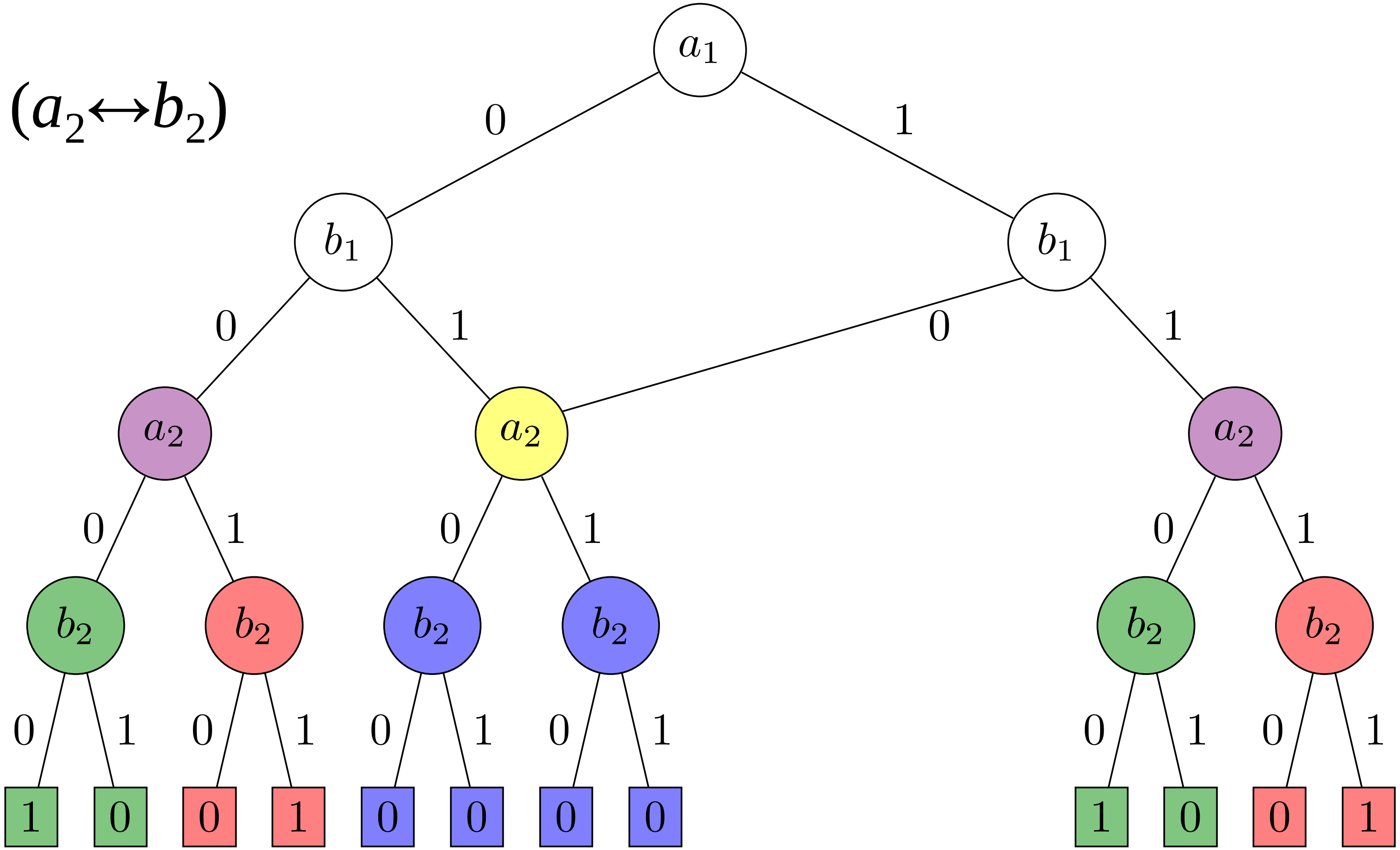
Removing Redundancy

$f = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$



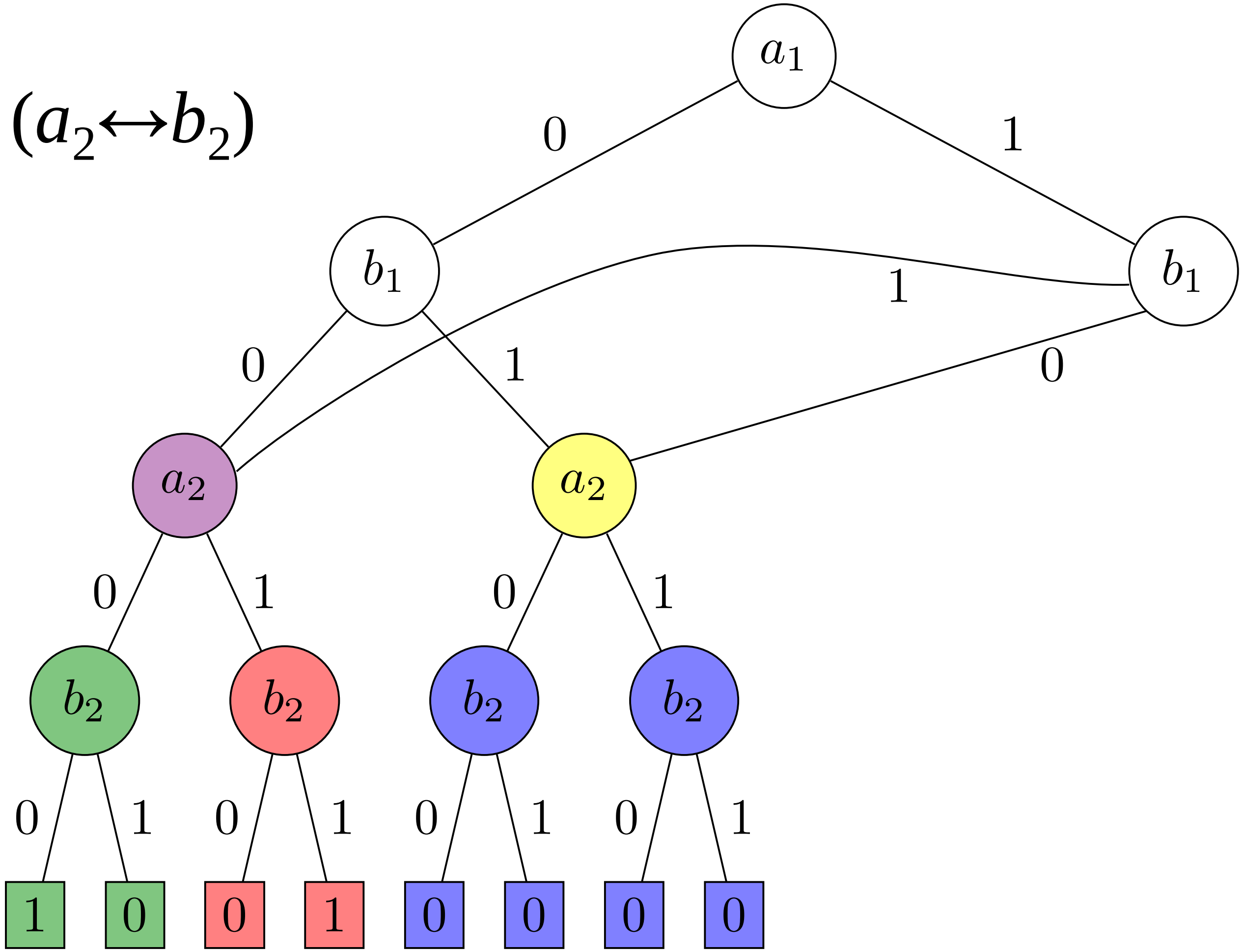
Removing Redundancy

$f = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$



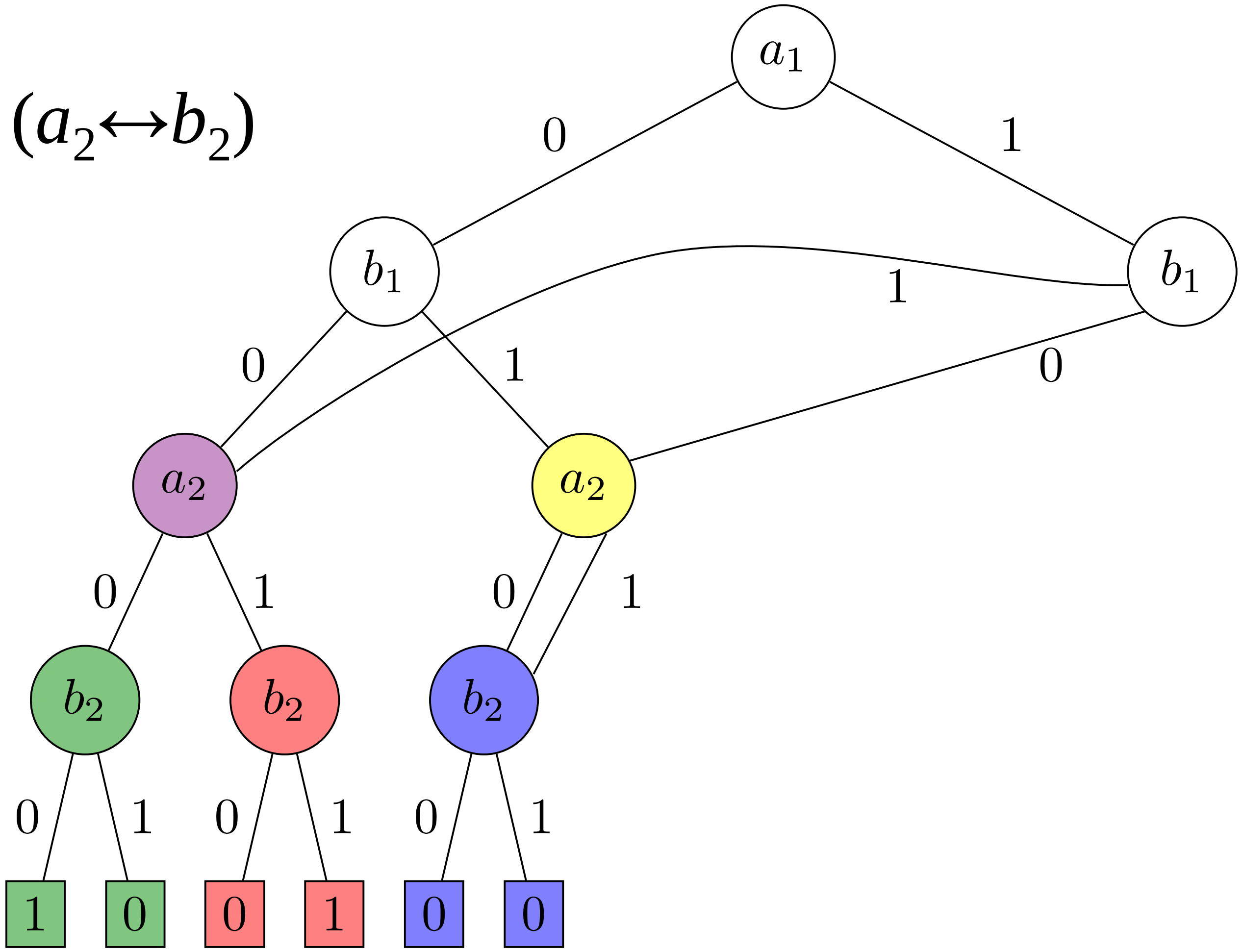
Removing Redundancy

$f = (a_1 \leftrightarrow b_1) \land (a_2 \leftrightarrow b_2)$



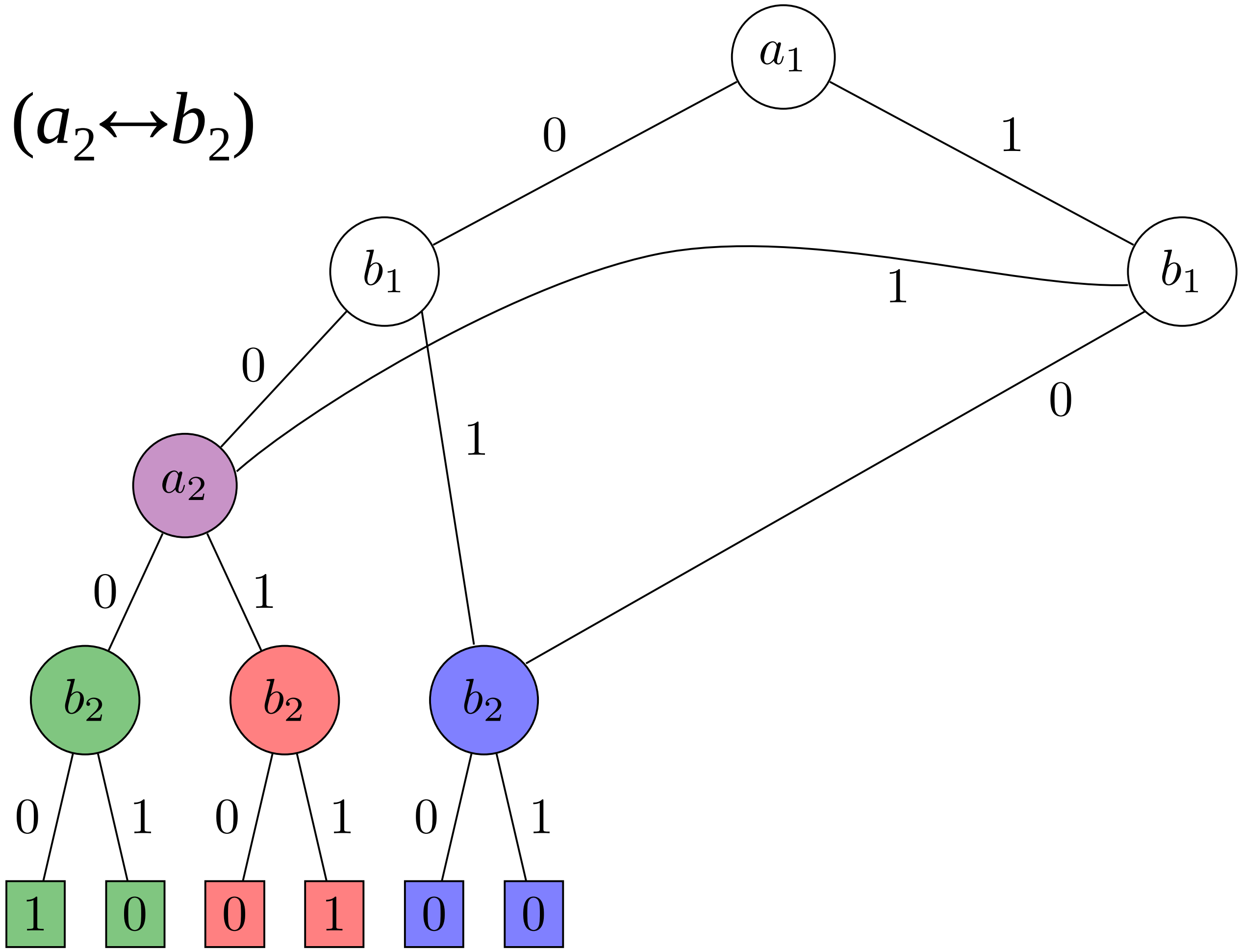
Removing Redundancy

$f = (a_1 \leftrightarrow b_1) \land (a_2 \leftrightarrow b_2)$



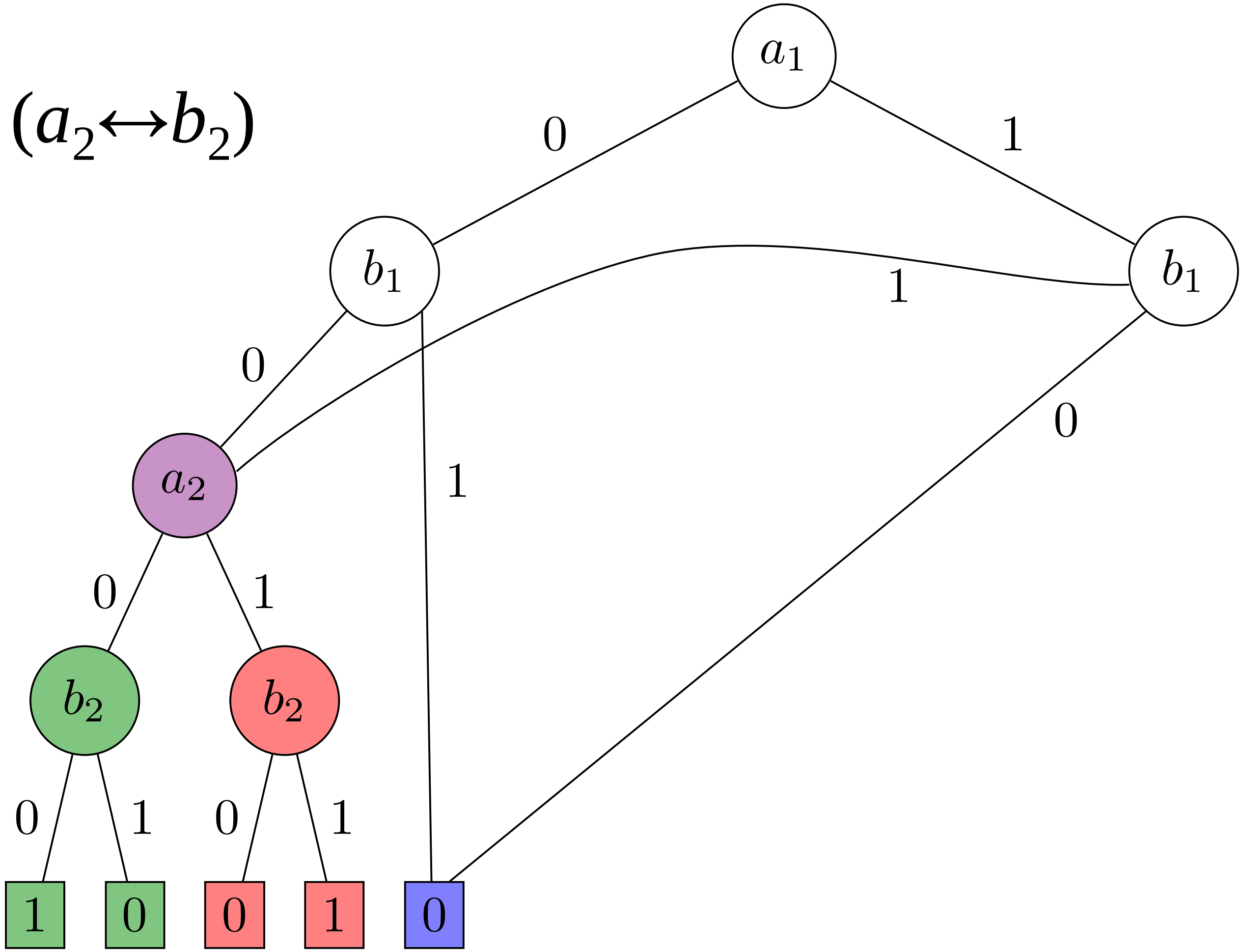
Removing Redundancy

$$f = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$$



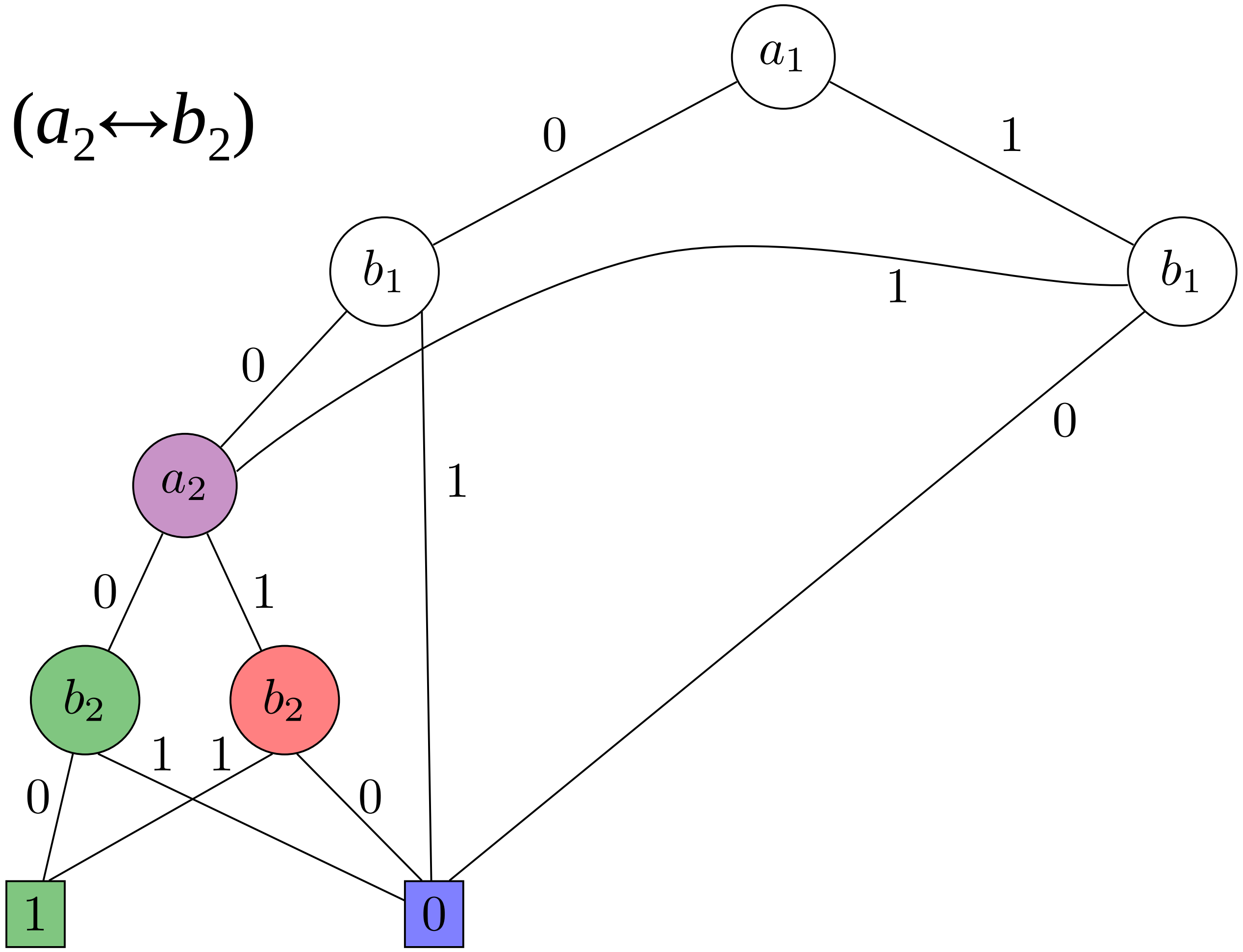
Removing Redundancy

$f = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$



Removing Redundancy

$f = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$



Reducing Binary Decision Diagrams

- Remove duplicate terminals
- Remove duplicate nonterminals
- Remove redundant tests

Reduced BDDs fulfill:

- Each variable appears at most once along every path from root to leaf
- The variables appear in the same order along every path from root to leaf
- The graph does not contain
  - isomorphic sub-graphs
  - Redundant nodes

Reducing Binary Decision Diagrams

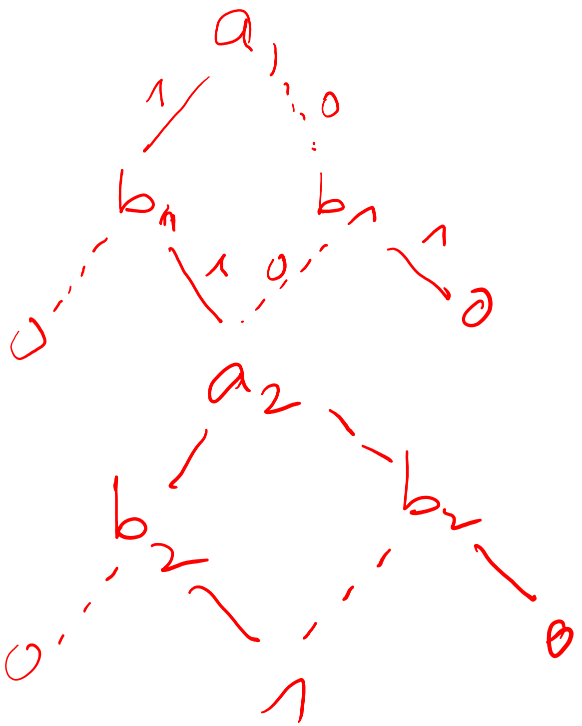
- After removing
- Canonical representation for Boolean formulas
  - ⇒Equivalent formulas have the same representation
  - ⇒Test for equivalence can be done in O(1) time
- Often very compact
- Efficient algorithms for computing pre- and post-images

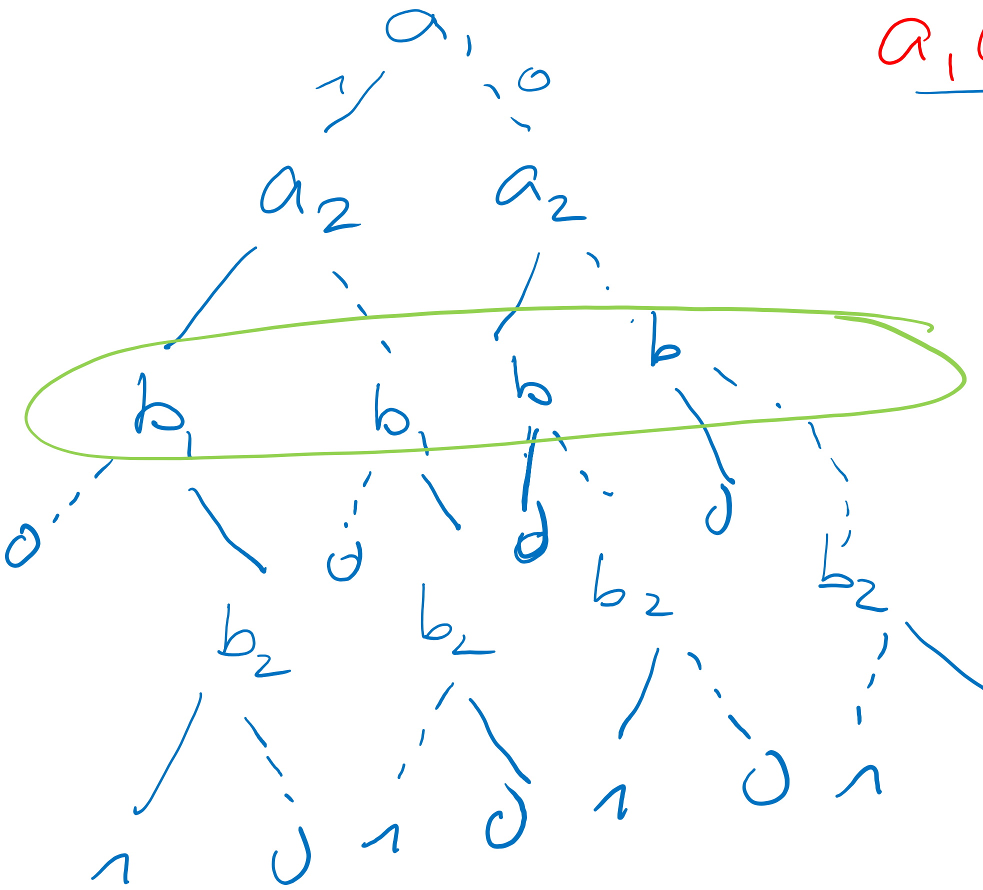
Order of Variables in the BDD

- $f\ (a_1, b_1, a_2, b_2) = \ (a_1 \leftrightarrow b_1) \wedge \ (a_2 \leftrightarrow b_2)$
- Which variable order should we use?

$a_1\ b_1\ a_2\ b_2$ better than

$a_1\ a_2\ b_1\ b_2$



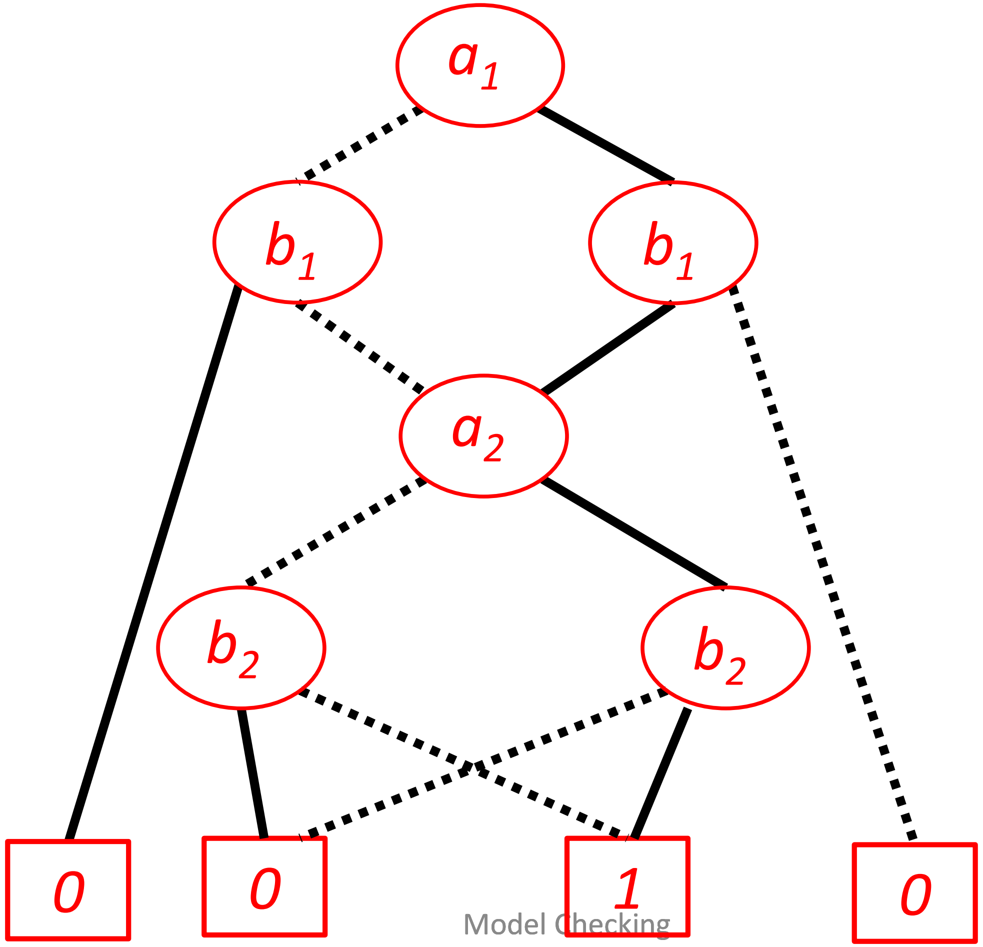


2^n nodes

$\bigwedge_{i=1}^{n} (a_i \leftrightarrow b_i)$

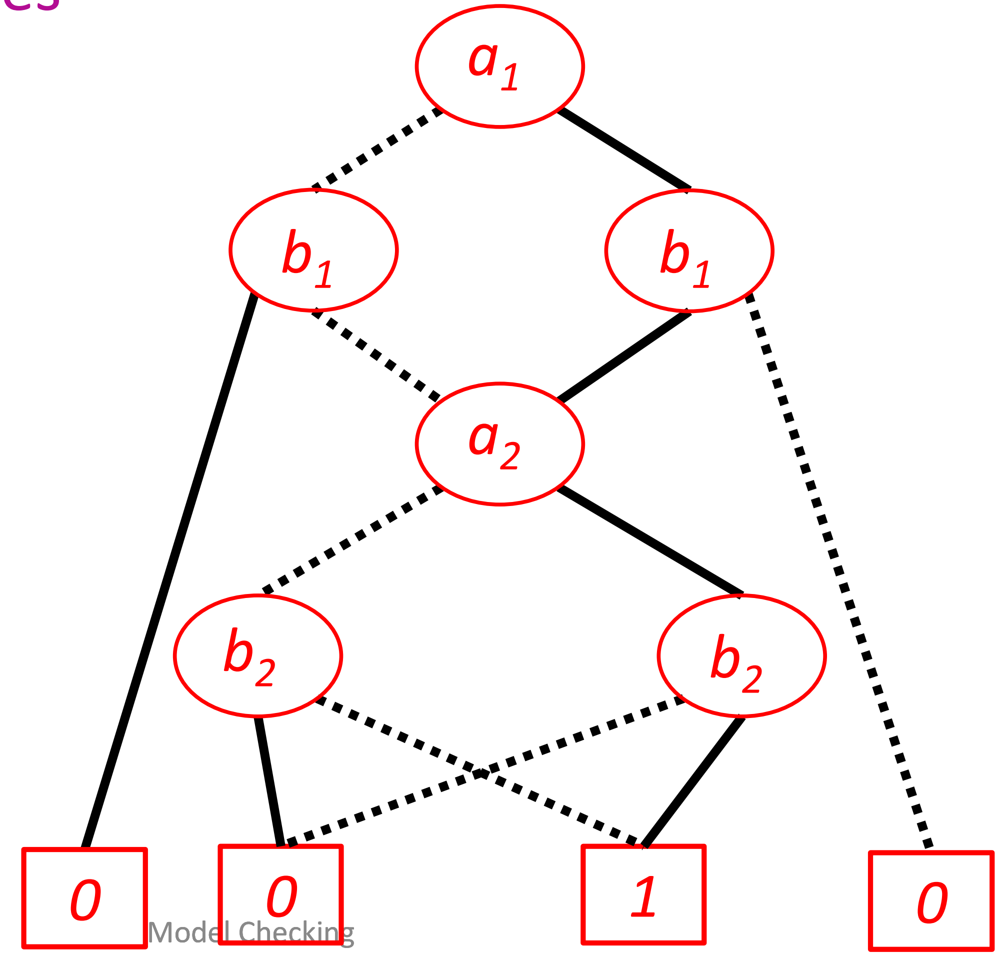
Order of Variables in the BDD

- $f\ (a_1, b_1, a_2, b_2) = \ (a_1 \leftrightarrow b_1) \land \ (a_2 \leftrightarrow b_2)$
- variable order $a_1 < b_1 < a_2 < b_2$:



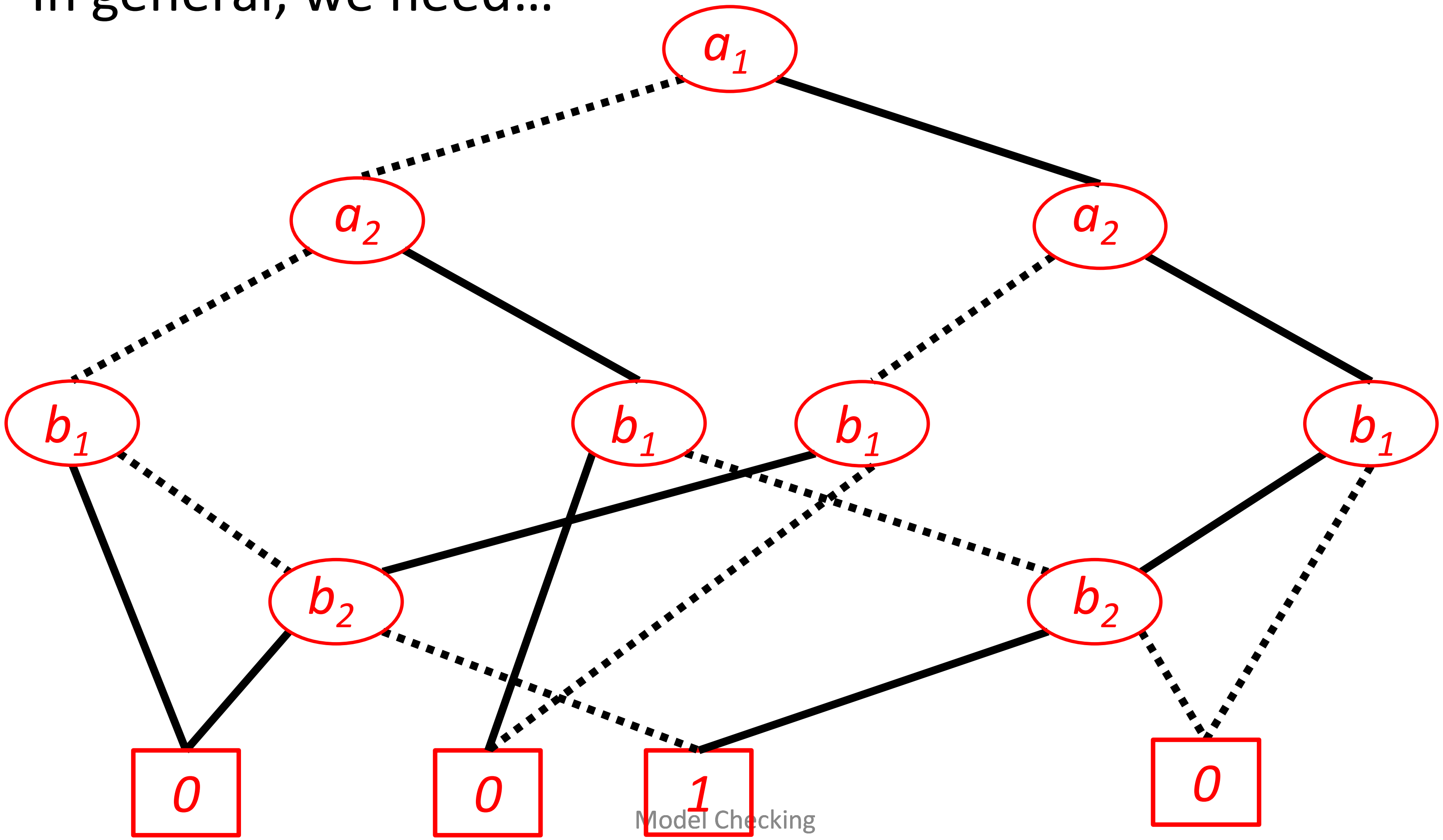
Order of Variables in the BDD

- In the general case: $a_1 < b_1 < a_2 < b_2 < .... < a_n < b_n$
- $f(a_1, b_1, \ldots, a_n, b_n) = (a_1 \wedge b_1) \vee \ldots \vee (a_n \wedge b_n)$
- BDD with 3n+2 nodes



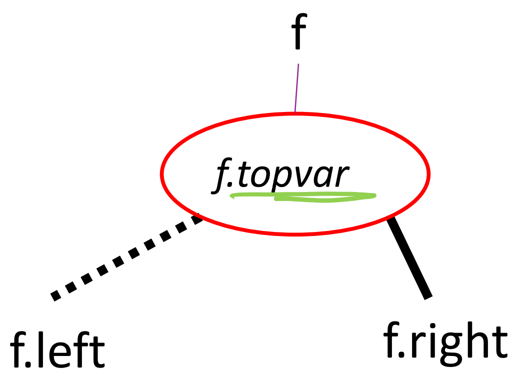
Order of Variables in the BDD

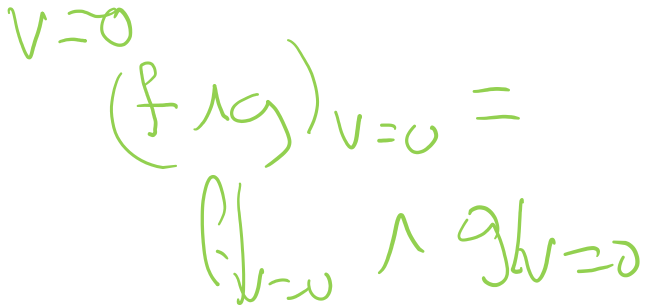
- $f\ (a_1, b_1, a_2, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$
- variable order $a_1 < a_2 < b_1 < b_2$
- In general, we need…



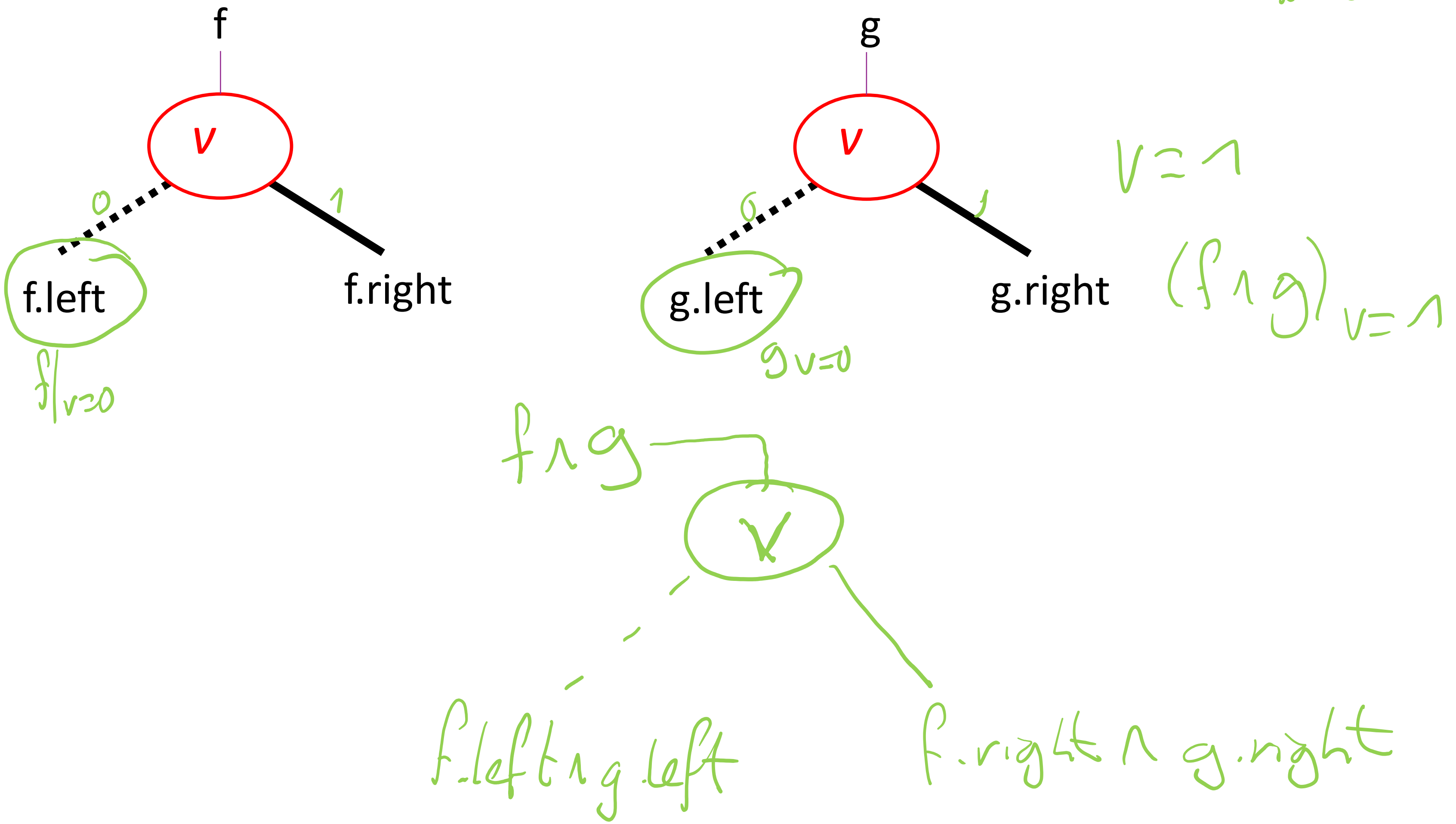
Operations on BDDs - Apply

- Gets two BDDs, representing functions **f** and **f’** and an operation *****
  - **Over the same variable ordering**
- Returns the BDD representing **f*f’**
- ***** can be any of 16 binary operations on two Boolean functions
- Bdds are built recursively
  - 0
  - 1
  - ITE(topvar, left, right)





bddand



Operations on BDDs - And

```
bddand(f,g)
```

Operations on BDDs - And


```
bddand(f,g)
   if f==0 || g==0 return 0
   if f==1 return g
   if g==1 return f
   if f==g return g
   if(f.topvar == g.topvar)
        return bddite(f.topvar, bddand(f.left, g.left),
        bddand(f.right, g.right))
   if(f.topvar < g.topvar)
        return bddite(f.topvar, bddand(f.left, g), bddand(f.right, g))
   if(f.topvar > g.topvar)
        return bddite(f.topvar, bddand(f.left, g), bddand(f.right, g))
```

BDD Operations

- Boolean operations $f * g$ can be performed in time $O(|f| \cdot |g|)$

- Other important operations:
- bddcofactor(f, v, b) = $f|_{v=b}$ (replace all occurrences of var v by value b)
  - $(x \land y \lor \neg x \land z)|_{x=0} = 0 \land y \lor 1 \land z = z$
  - $(x \land y \lor \neg x \land z)|_{x=1} = 1 \land y \lor 0 \land z = y$
- bddexists(v,f) = $\exists v.\, f \;=\; f|_{v=0} \lor f|_{v=1}$
  - $\exists x.\,(x \land y \lor \neg x \land z)|_{v=0} = y \lor z$

BDD Operations

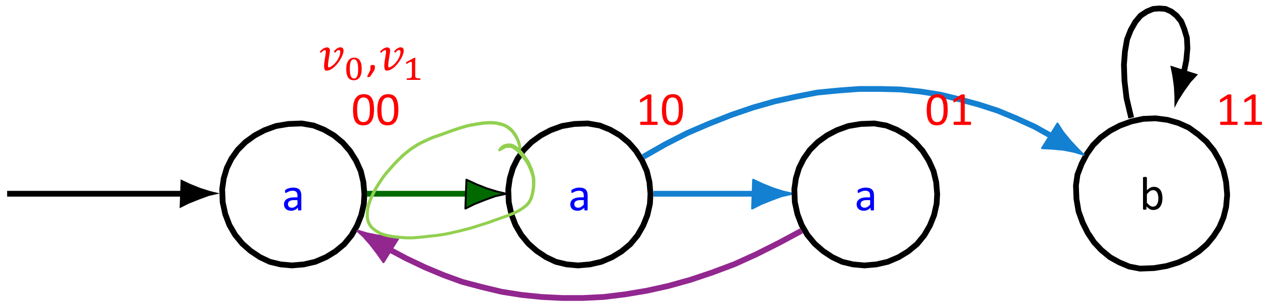
- Boolean operations $f * g$ can be performed in time $O(|f| \cdot |g|)$

- Other important operations:
- bddcofactor(f, v, b) = $f|_{v=b}$ (replace all occurrences of var v by value b)

- bddexists(v,f) = $\exists v. f = f|_{v=0} \vee f|_{v=1}$

Representing Kripke Structures with OBDDs

Example from SAT chapter



$$path_1(s_0, s_1, s_2) = S_0(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2)$$

$$= \neg v_{0,0} \wedge \neg v_{1,0} \wedge$$

$$\begin{pmatrix} \neg v_{00} \wedge \neg v_{10} \wedge v_{01} \wedge \neg v_{11} \\ \vee\, v_{00} \wedge \neg v_{10} \wedge v_{11} \\ \vee\, \neg v_{00} \wedge v_{10} \wedge \neg v_{01} \wedge \neg v_{11} \\ \vee\, v_{00} \wedge v_{10} \wedge v_{01} \wedge v_{11} \end{pmatrix}$$

$$S_0(v_0, v_1) = \neg v_0 \wedge \neg v_1$$

$$R(v_0, v_1, v_0', v_1') = \neg v_0 \wedge \neg v_1 \wedge v_0' \wedge \neg v_1'$$
$$\vee\, v_0 \wedge \neg v_1 \wedge v_1'$$
$$\vee\, \neg v_0 \wedge v_1 \wedge \neg v_0' \wedge \neg v_1'$$
$$\vee\, v_0 \wedge v_1 \wedge v_0' \wedge v_1'$$

$$a(v_0, v_1) = \neg v_0 \vee \neg v_1$$

https://eecs.ceas.uc.edu/~weaversa/BDD_Visualizer.html $\bigvee_{i=0}^{k} \neg a(s_i) = \neg(\neg v_{00} \vee \neg v_{10}) \vee \neg(\neg v_{01} \vee \neg v_{11})$

Symbolic Model Checking for CTL

Symbolic (BDD-based) model checking

- BDD-based model checking manipulates set of states
  - **BDD efficiently represents Boolean function that represents set**
- Implement breadth-first search
- Many algorithms reminiscent of explicit state

Operations on sets

- Union of sets $\Rightarrow \vee$ (or) over their BDDs
- Intersection $\Rightarrow \wedge$ (and)
- Complementation $\Rightarrow \neg$ (not)
- Equality of sets $\Rightarrow \leftrightarrow$ (iff)

BDD-based Model Checking

- Accept: Kripke structre M, CTL formula f
- Returns: S_f - the set of states satisfying f

M is given by:

- BDD R(V,V’), representing the transition relation
- BDD p(V), for every $p \in AP$, representing S_p
  - the set of states satisfying p
- $V = (v_1,\ldots v_n)$

BDD-based Model Checking

- The algorithm works from simpler formulas to more complex ones
- When a formula g is handled, the BDD for S_g is built
- A formula is handled only after all its sub-formulas have been handled

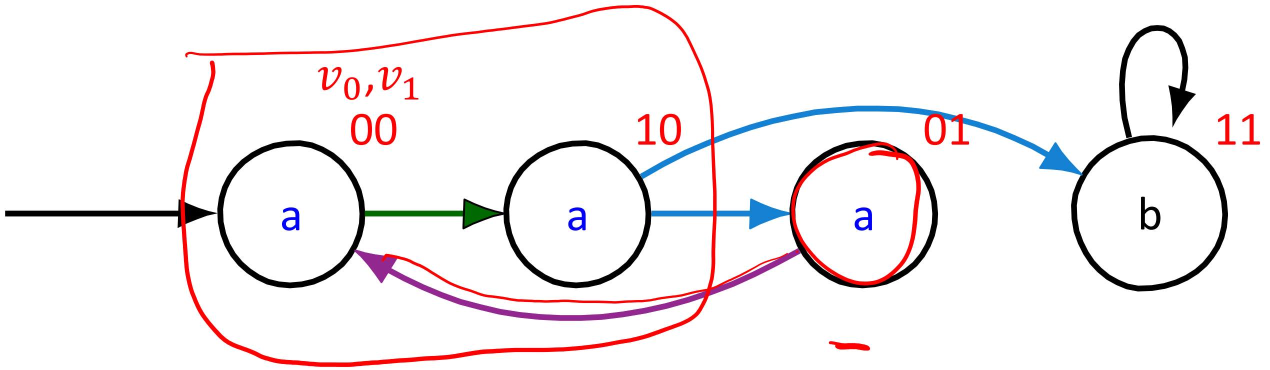
BDD-based Model Checking

- For $p \in AP$, return $f(p)$
- For $f = f_1 \wedge f_2$, return $f_1 \wedge f_2$
- For $f = \neg f_1$, return $\neg f_1$

BDD-based Model Checking

- For $p \in AP$, return $p(V)$
- For $f = f_1 \wedge f_2$, return $f(V) = f_1(V) \wedge f_2(V)$
- For $f = \neg f_1$, return $f(v) = \neg\, f_1(V)$

Example from SAT chapter



$S_0(v_0, v_1) = \neg v_0 \wedge \neg v_1$

$$R(v_0, v_1, v_0', v_1') = \neg v_0 \wedge \neg v_1 \wedge v_0' \wedge \neg v_1' \vee v_0 \wedge \neg v_1 \wedge v_1' \vee \neg v_0 \wedge v_1 \wedge \neg v_0' \wedge \neg v_1' \vee v_0 \wedge v_1 \wedge v_0' \wedge v_1'$$

$a(v_0, v_1) = \neg v_0 \vee \neg v_1$

$\neg v_1$. $EX(\neg v_1)$

For f = EX f_1 return

$\exists v_0' v_1' . [\neg v_1' \wedge R(v, v')]$

BDD-based Model Checking

- For $f = EX\ f_1$ return

  $f(V) = \exists V' \, [\, f_1(V') \wedge R(V,V') \,]$

- This BDD represents all (encoding V of) states that have a successor (with encoding V') in f_1

- Defined as a new BDD operator:
  EX $f_1(V) = \exists V' \, [\, f_1(V') \wedge R(V,V') \,]$

- This operation is also called pre-image or Pred

- Important:
  the formula defines a sequence of BDD operations and therefore is considered as a symbolic algorithm

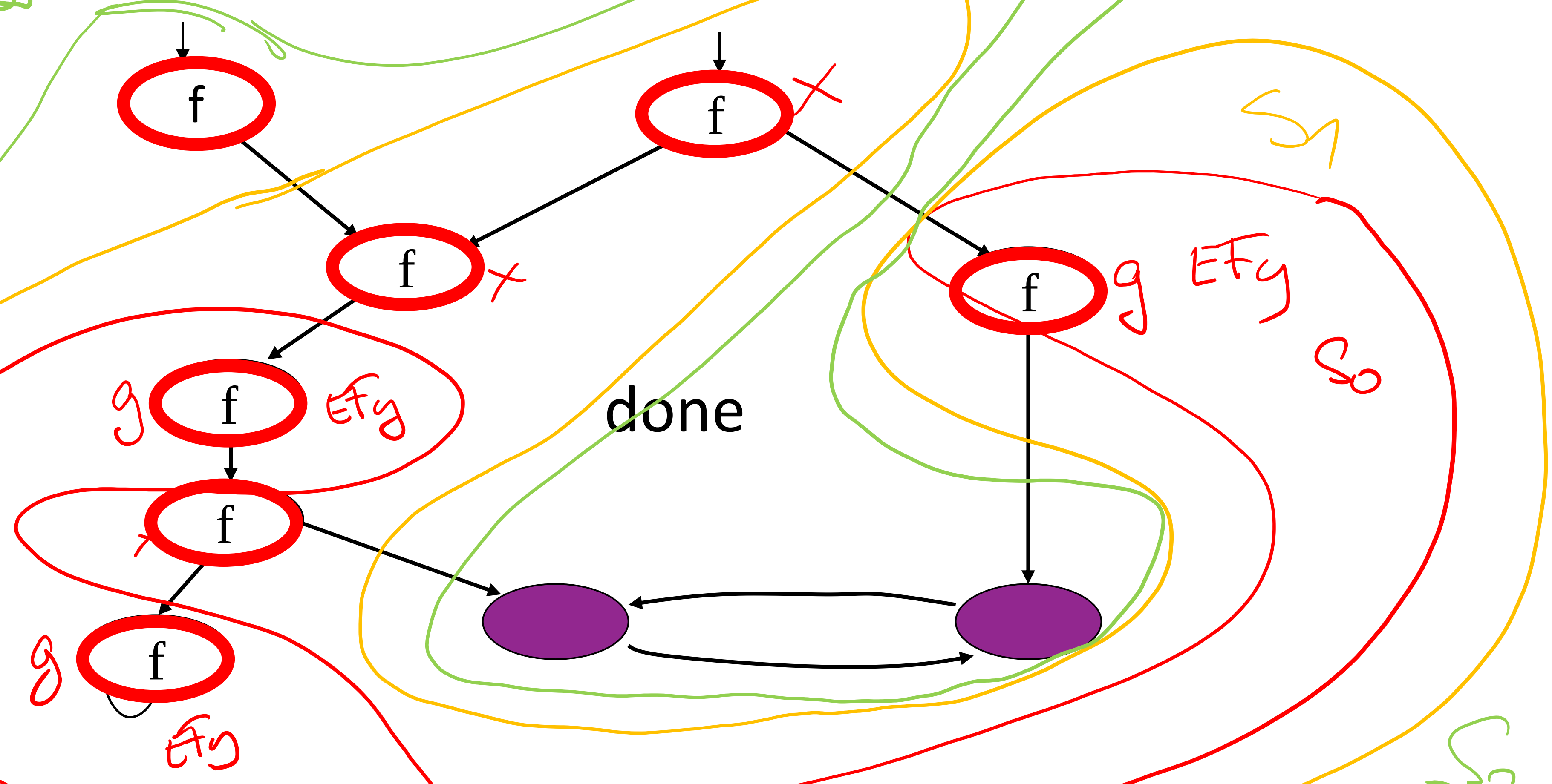
$s \models EF\, p$ iff

1. $s \models p$ OR
2. $s \models EX\, EF\, p$

Example: f = EF g

Note: s ⊨ $EF\ p$ iff

…



BDD-based Model Checking for **f= EF g**

Given: BDDs **R** and $\mathbf{S_g}$ **:**

procedure **CheckEF** (S_g)

Q := $\emptyset$; Q' := S_g ;

While **Q** $\neq$ **Q'** do

Q := Q'

Q' := (EX Q) ∨ Q

end while

S_f := Q ; return(S_f)

BDD-based Model Checking for **f= EF g**

Given: BDDs **R** and **S_g :**

procedure **CheckEF** (S_g)

    Q := emptyset; Q' := S_g ;

    while **Q** $\neq$ **Q'** do

        Q := Q';

        Q' := Q $\vee$ EX(Q)

    end while

    S_f := Q ;   return(S_f)

The algorithm applies

- BDD operations (or $\vee$**)**, and EX
- comparison $Q(V) \neq Q'(V)$ (easy)

Therefore, this is a symbolic algorithm!

Model Checking M |= f (cont.)

- We compute subformula **g** of **f** after all subformulas of **g** have been computed
- For subformula **g**, the algorithm returns the **set of states** that satisfy **g** ($\mathbf{S_g}$)

Model Checking **f = E[g_1 U g_2]**

Given: a model **M** and the sets $\mathbf{S_{g_1}}$ and $\mathbf{S_{g_2}}$ of states satisfying **g_1 and g_2** in M

procedure CheckEU ($\mathbf{S_{g_1}}$, $\mathbf{S_{g_2}}$)

Q := emptyset; Q’ := S_{g2} ;

while Q $\neq$ Q’ do

Q := Q’;

Q’ :=

end while

S_f := Q ; return(S_f)

Least fixpoint

Model Checking $f = E[g_1 \; U \; g_2]$

Given: a model **M** and the sets S_{g_1} and S_{g_2} of states satisfying **g_1 and g_2** in M

procedure CheckEU (S_{g_1}, S_{g_2})

Q := emptyset; Q' := S_{g2} ;

while Q $\neq$ Q' do

Q := Q';

Q' := $Q' \vee (S_{g_1} \wedge EX(Q'))$

end while

S_f := Q ; return(S_f)

Least fixpoint

Example: **f = EG p**

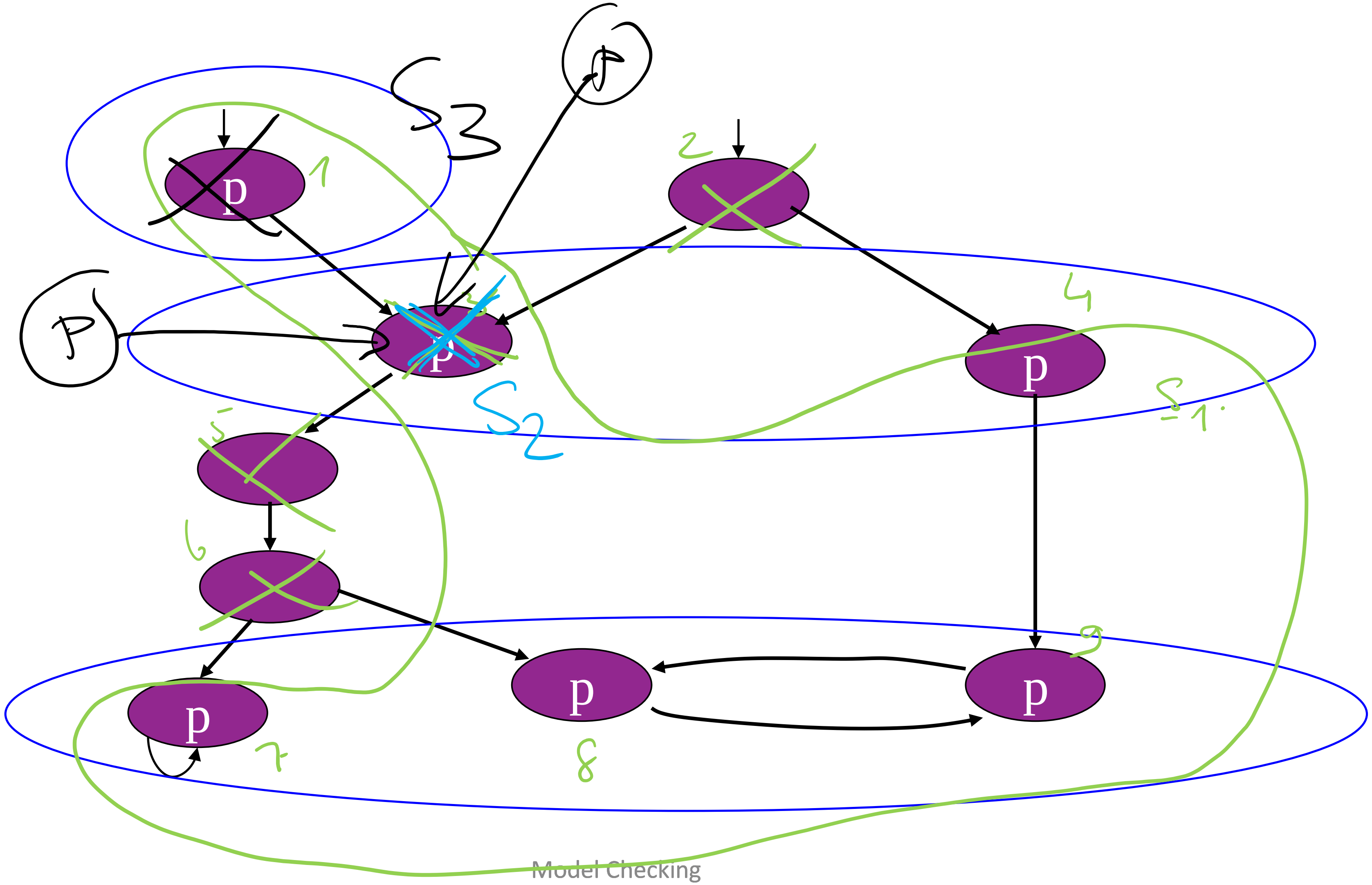
Note: $s \vDash EG\ p$ iff

...

$s \vDash p$ AND

$s \vDash EX\ EG\ p$

EG p?



Model Checking f = EG g

CheckEG gets **M** and $\mathbf{S_g}$ and returns $\mathbf{S_f}$

procedure CheckEG (S_g)

  $Q := S$; $Q' := S_g$;

  while $Q \neq Q'$ do

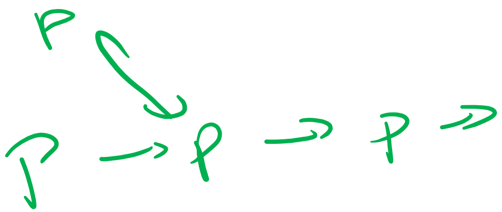
    $Q := Q'$;

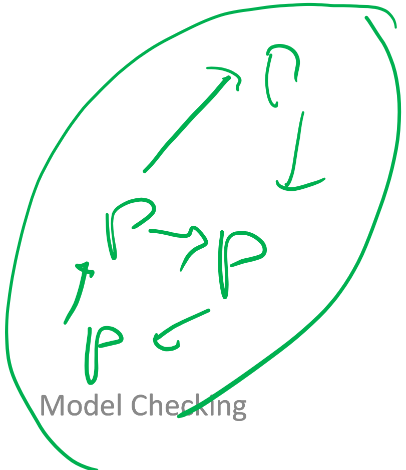
    $Q' := Q \cap Pred(Q)$

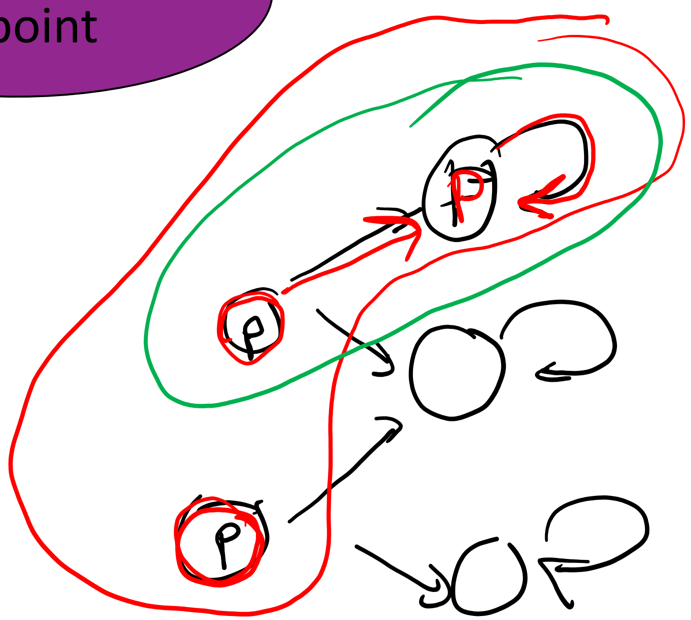
  end while

  $S_f := Q$; return(S_f)

Greatest fixpoint







EF and EG are similar

CheckEG gets **M** and $\mathbf{S_g}$ and returns $\mathbf{S_f}$

procedure CheckEG (S_g)
    Q := S ; Q’ := S_g ;
    while Q $\neq$ Q’ do
        Q := Q’;
        Q’ := $\boldsymbol{Q} \wedge \boldsymbol{EX}(\boldsymbol{Q})$
    end while
    S_f := Q ;   return(S_f)

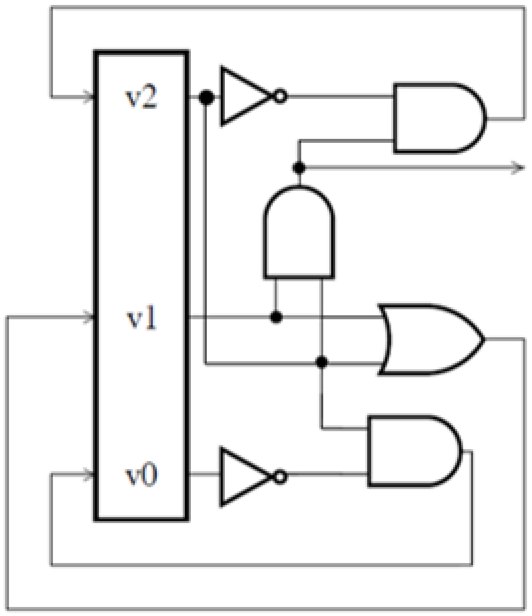
procedure CheckEF (Sg)
    Q := ∅; Q’ := Sg ;
    while Q $\neq$ Q’ do
        Q := Q’;
        Q’ := $\boldsymbol{Q} \vee \boldsymbol{EX}(\ \boldsymbol{Q}\ )$
    end while
    Sf := Q ;   return(Sf)

EG EX EF p

Homework

Deadline: 24.6. 4:00pm
Sent solution to: modelcheckin

Given the following synchronous c



The initial value of the state varia unknown.

Task 1a. [4 Points]

- Show the BDD for the transition relation. Use the variable ordering v2', v2, v1, v1', v0, v0'

Task 1b. [4 Points]

- Draw the Kripke Structure $M = (S, S_0, R, AP, L)$ that represents C. (Hint: see Homework 1.) Show the iterations of the computation of the formula $EG\neg v_2$. (You can show the iterations graphically, or you can give a sequence of sets of states. You don't need to draw any BDDs.)

Task 1c. [2 Points]

- Show which states fulfil the formula $EF\ EG\neg v_2$.

Fairness in Symbolic Model Checking

Counterexamples and Witnesses

Relational Product Computations