Lecture 13

Binary Decision Diagrams

Zvonimir Rakamarić
University of Utah

Last Time

- Complexity
- Complexity classes P, NP, NP-complete
- Proving NP-completeness using reduction

This Time

- Binary decision diagrams (BDDs)

History I

- Lee 1959, Akers 1976
  - Idea of representing Boolean function as BDD
- Hopcroft, Fortune, Schmidt 1978
  - Recognized that ordered BDDs were like finite state machines
  - Polynomial algorithm for equivalence
- Bryant 1986
  - Proposed as useful data structure + efficient algorithms

History II

- McMillan 1993
  - Developed symbolic model checking
  - Method for verifying complex sequential systems
- Bryant 1991
  - Proved that multiplication has exponential BDD
  - No matter how variables are ordered

Binary Decision Diagrams

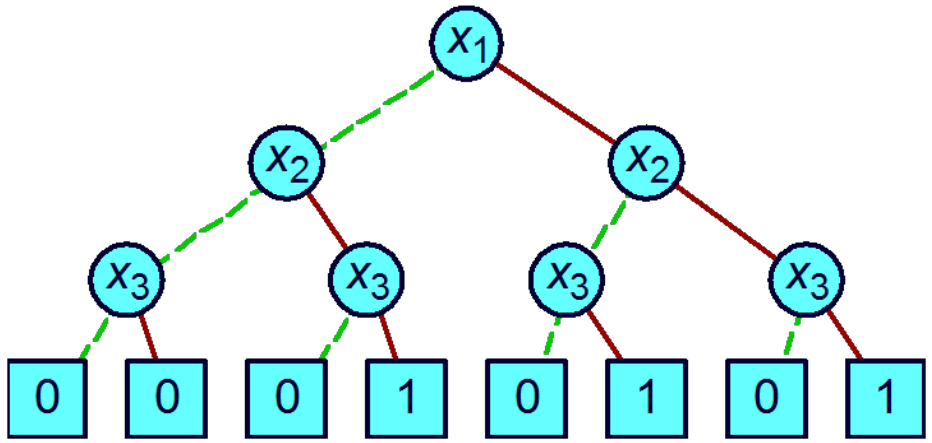
- Canonical data structure for representing quantifier-free boolean formulas
  - Therefore also represent sets!
- Equivalence checking in constant time
- In practice, model checkers spend more than 90% of their time in “pre-image” or “post-image” computation
  - Almost synonymous with “symbolic” model checking

Decision Structures

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Decision Tree



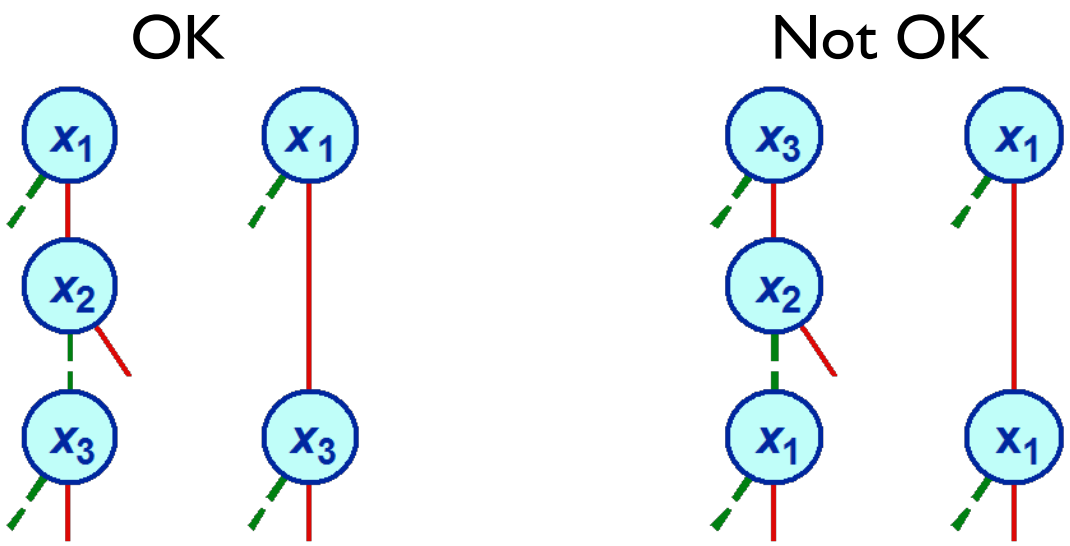
- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

Binary Decision Tree

- Order k boolean variables x1, ..., xk
- Binary tree of height k+1, each leaf labeled 0 or 1
- Leaf of path “left, right, right,...” gives value of boolean formula if x1=0, x2=1, x3=1, etc.

Variable Ordering

- Assign arbitrary total ordering to variables
  - e.g., $x_1 < x_2 < x_3$
- Variables must appear in ascending order along all paths



Properties

- No conflicting variable assignments along path
- Simplifies manipulation

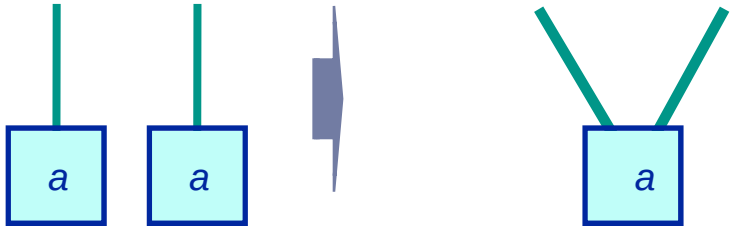
(Reduced Ordered) Binary Decision Diagram

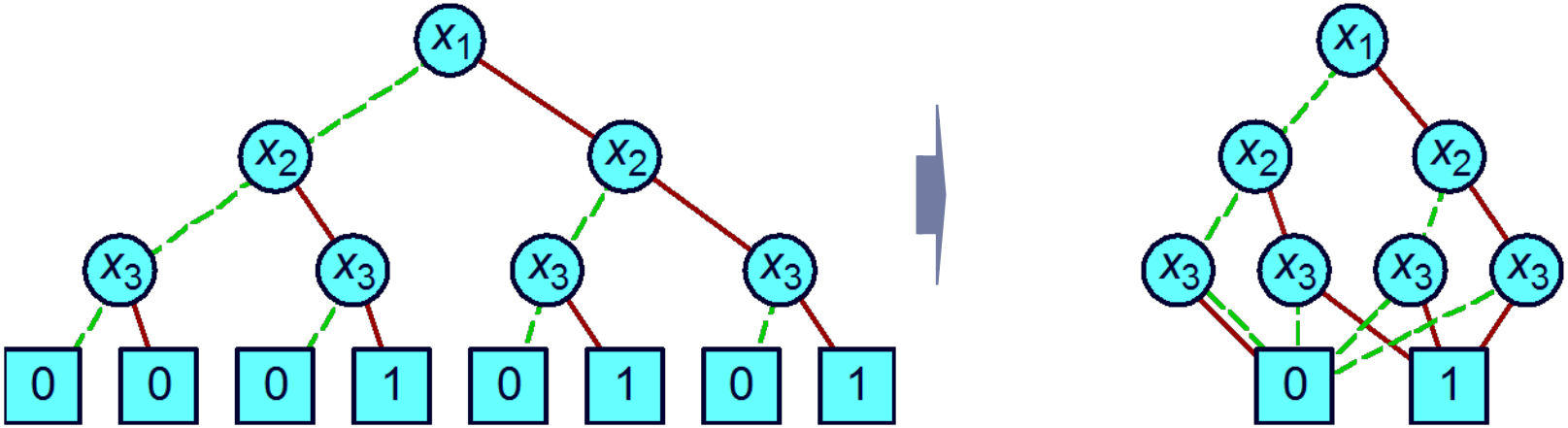
- Perform the following steps
  - Identify isomorphic subtrees (this gives a dag)
  - Eliminate nodes with identical left and right successors
  - Eliminate redundant tests

- For a given boolean formula and variable order, the result is unique
  - The choice of variable order may make an exponential difference!

Reduction Rule #1

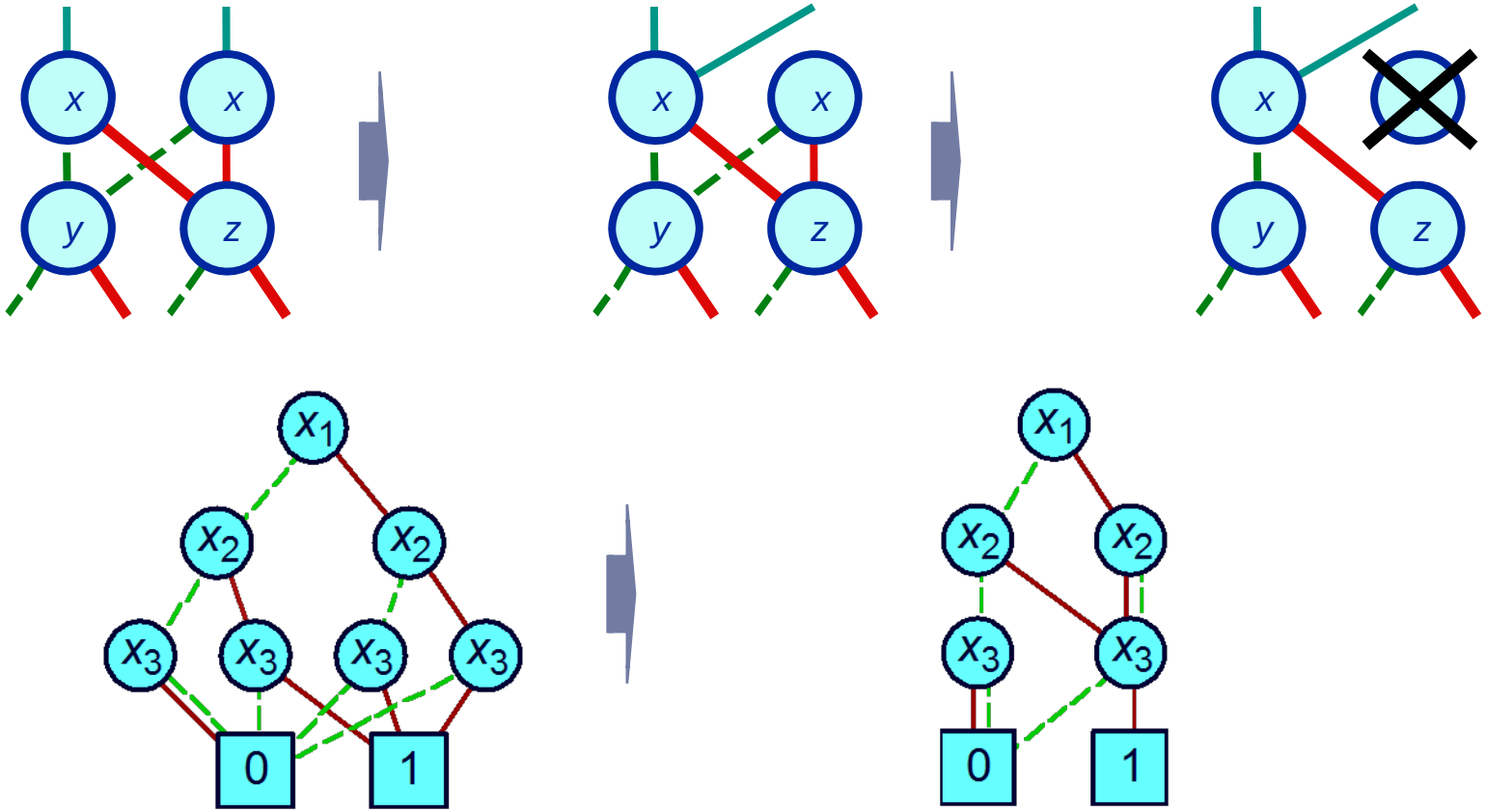
Merge equivalent leaves





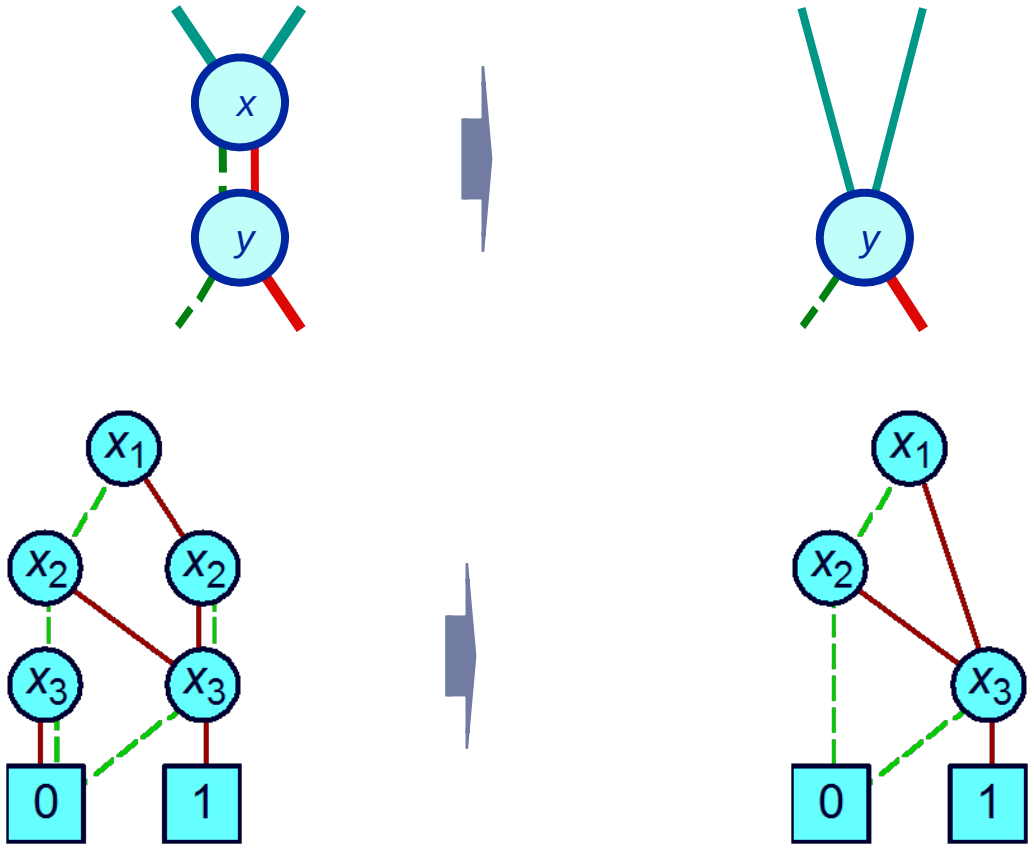
Reduction Rule #2

Merge isomorphic nodes

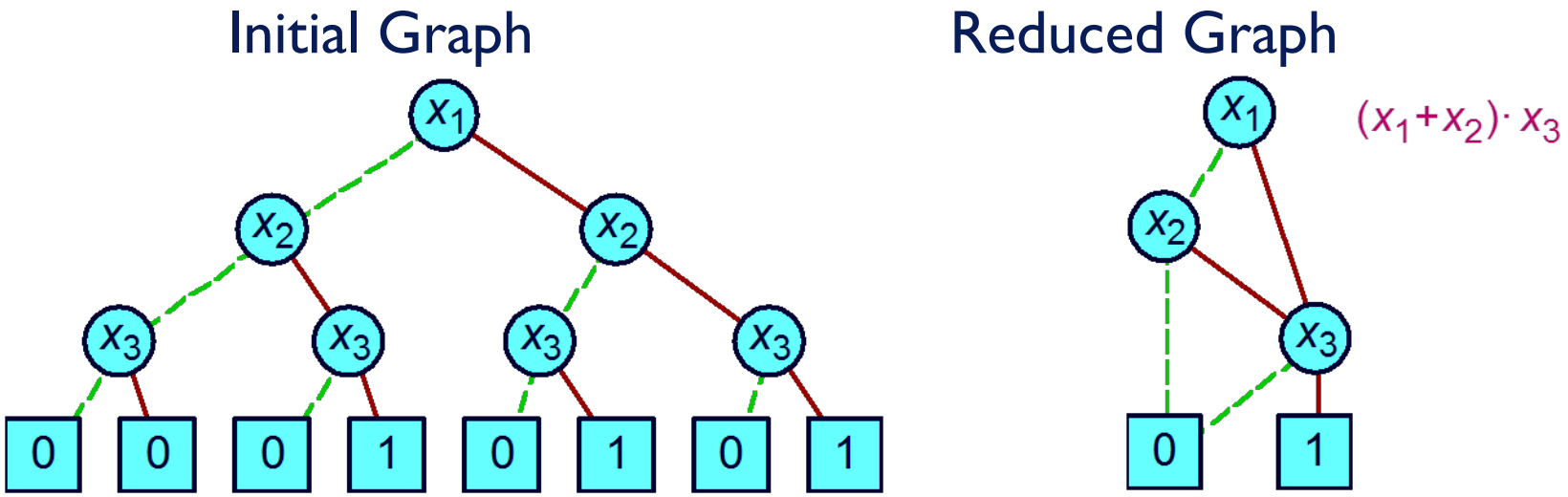


Reduction Rule #3

Eliminate Redundant Tests



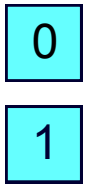
Example OBDD



- Canonical representation of Boolean function
  - For given variable ordering
  - Two functions equivalent if and only if graphs isomorphic
    - Can be tested in linear time
  - Desirable property: *simplest form is canonical*.

Example Functions

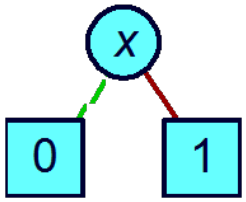
Constants



Unique unsatisfiable function

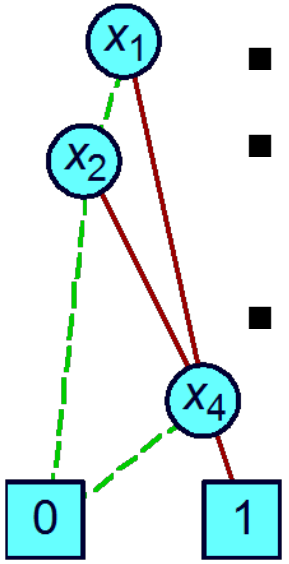
Unique tautology

Variable



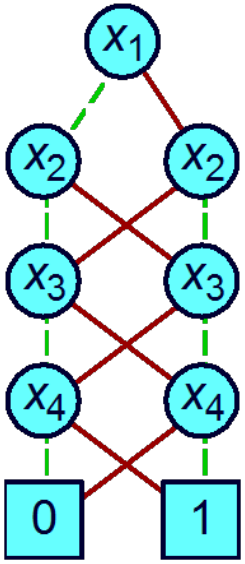
Treat variable as function

Typical Function



- $(x_1 \vee x_2) \wedge x_4$
- No vertex labeled x_3
  - independent of x_3
- Many subgraphs shared

Odd Parity

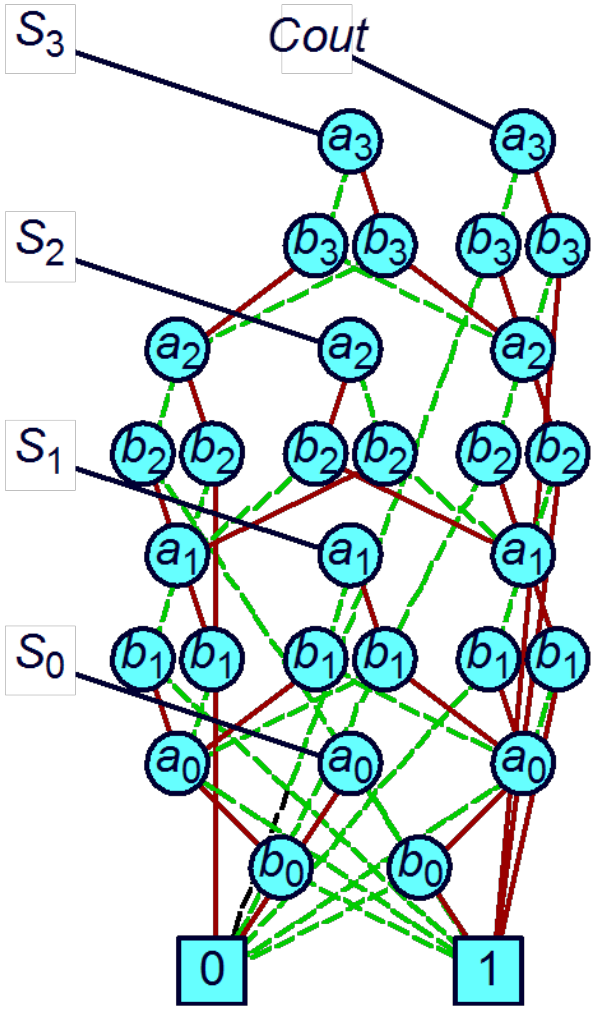


Linear representation

More Complex Functions

- Functions
  - Add 4-bit words a and b
  - Get 4-bit sum s
  - Carry output bit `Cout`

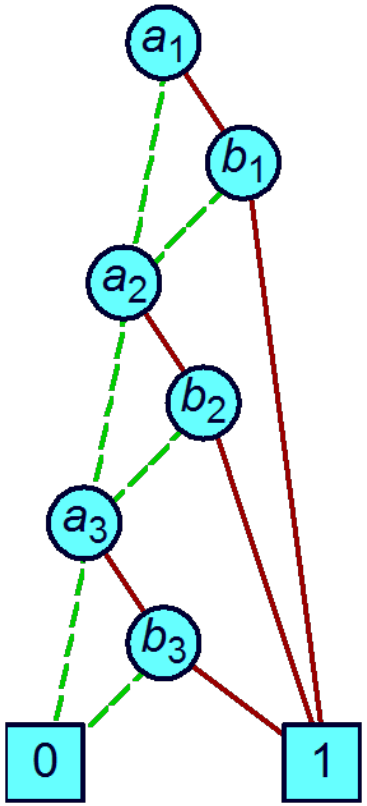
- Shared Representation
  - Graph with multiple roots
  - 31 nodes for 4-bit adder
  - 571 nodes for 64-bit adder
  - Linear growth!



Effect of variable ordering

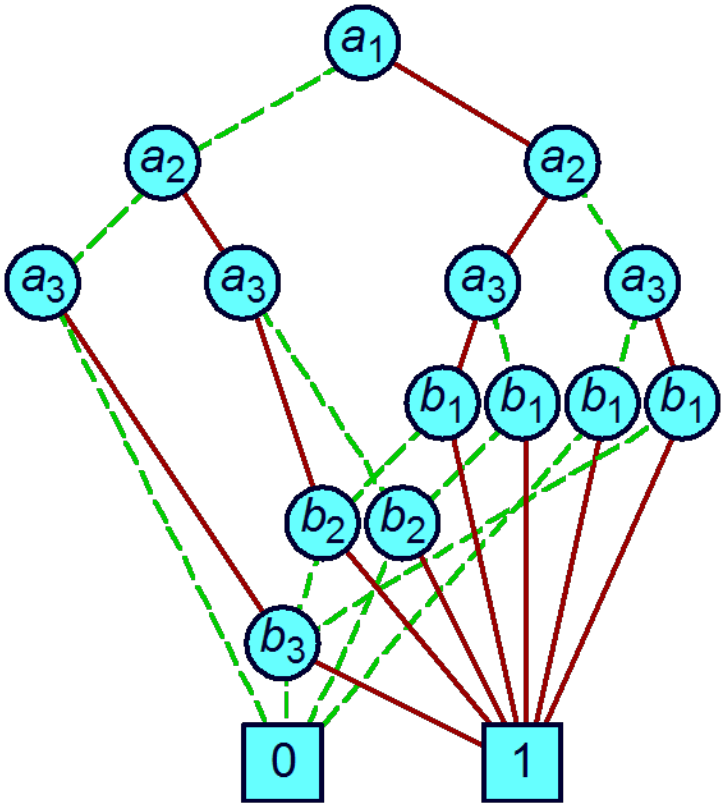
$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

Good ordering



Linear growth

Bad ordering



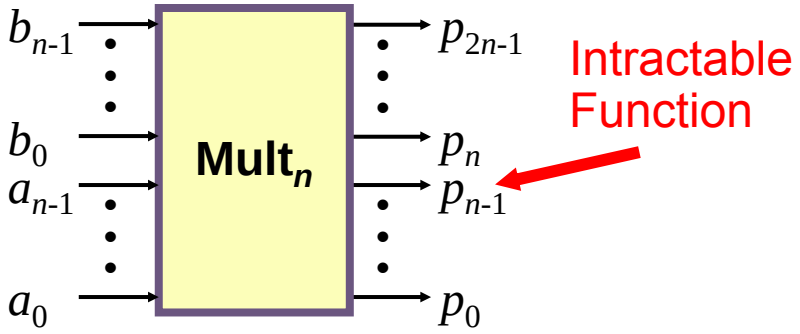
Exponential growth

Dynamic variable reordering

- Invented by Richard Rudell, Synopsys
- Periodically attempt to improve ordering for all BDDs
  - Part of garbage collection
  - Move each variable through ordering to find its best location
- Has proved very successful

Lower bound for multiplication (Bryant 1991)

- Integer multiplier circuit
  - *n*-bit input words *A* and *B*
  - 2*n*-bit output word *P*
- Boolean function
  - Middle bit (*n*-1) of product
- Complexity
  - Exponential BDD for all possible variable orderings



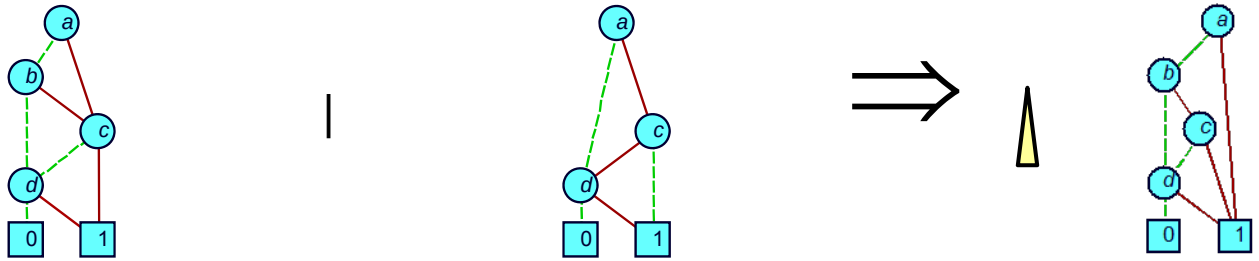
Actual Numbers

- 40,563,945 BDD nodes to represent all outputs of 16-bit multiplier
- Grows 2.86x per bit of word size

Apply Operation

- Concept
  - Basic technique for building OBDD from Boolean formula.

A *op* *B* *A op B*



Arguments *A*, *B*, *op*

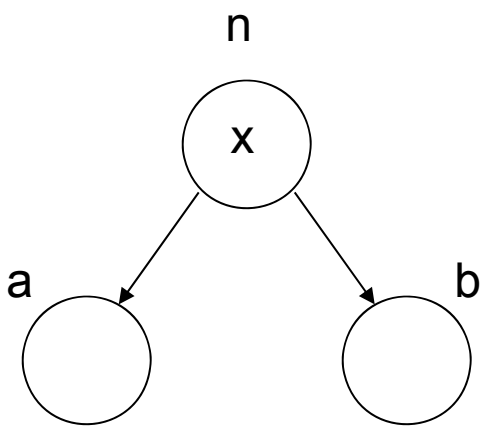
- *A* and *B*: Boolean Functions
  - Represented as OBDDs
- op: Boolean Operation (e.g., ^, &, |)

Result

- OBDD representing composite function
- *A op B*

BDD operations $\neg$, $\wedge$, $\vee$, $\exists$, $\forall$

BDD node



n.var = x
n.false = a
n.true = b

- BDD manager maintains a directed acyclic graph of BDD nodes
- ite(x,a,b) returns a node with variable x, left child a, and right child b.

not(a)

```
if (a = true) return false
if (a = false) return true
return ite(a.var, not(a.false), not(a.true))
```

Complexity: O(|a|)

and(a,b)

```
if (a = false ∨ b = false) return false
if (a = true) return b
if (b = true) return a
if (a = b) return a
if (a.var < b.var)
    return ite(a.var, and(a.false,b), and(a.true,b))
if (b.var < a.var)
    return ite(b.var, and(a,b.false), and(a,b.true))
// a.var = b.var
return ite(a.var, and(a.false,b.false), and(a.true,b.true))
```

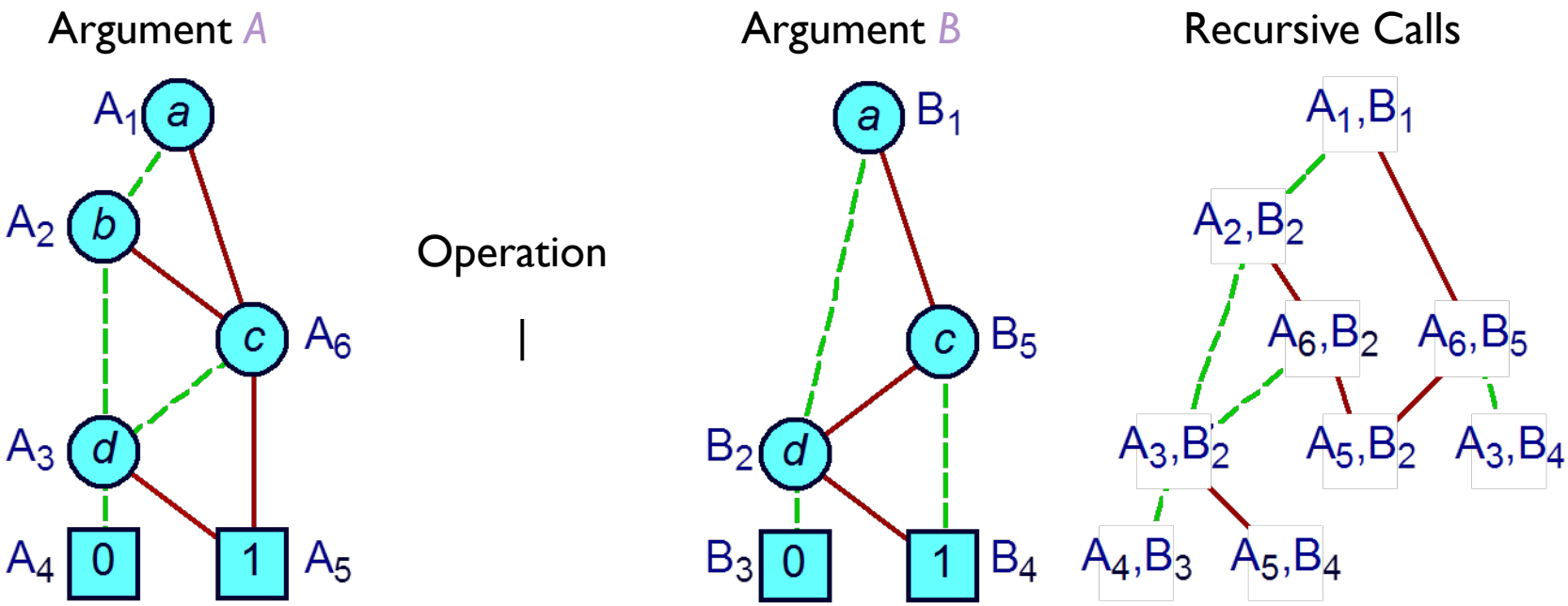
Complexity: $O(|a| \times |b|)$

Derived operations

Operations returning BDD:
or(a,b) ≡ not(and(not(a),not(b)))

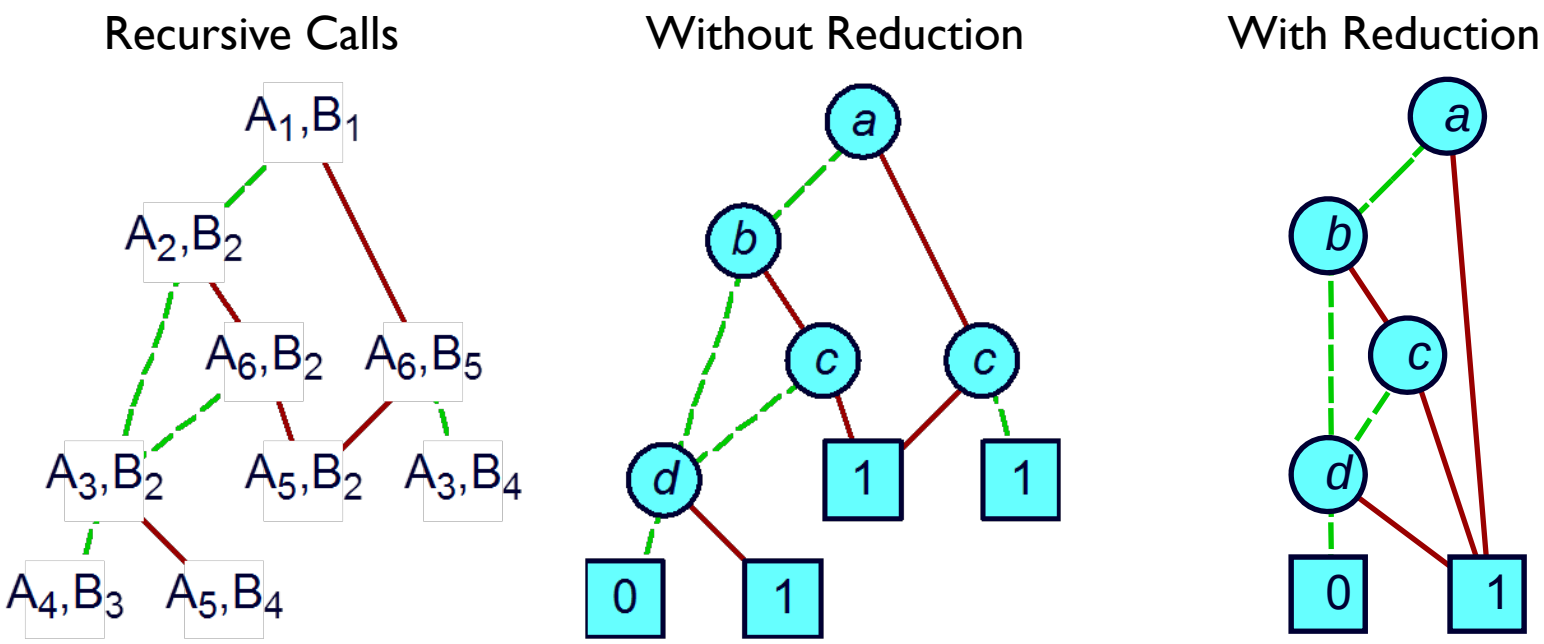
Operations returning boolean:
implies(a,b) ≡ (or(not(a),b) = true)
iff(a,b) ≡ (a = b)

Apply Execution Example



- Optimizations
  - Dynamic programming
  - Early termination rules

Apply Result Generation



- Recursive calling structure implicitly defines unreduced BDD
- Apply reduction rules bottom-up as return from recursive calls

Next Time

- Model checking