ICAPS 2014 Tutorial

Decision Diagrams in Automated Planning and Scheduling

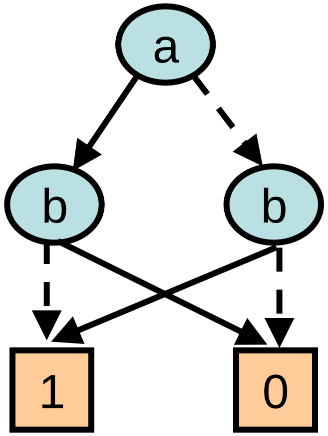
Scott Sanner

NICTA

THE AUSTRALIAN NATIONAL UNIVERSITY

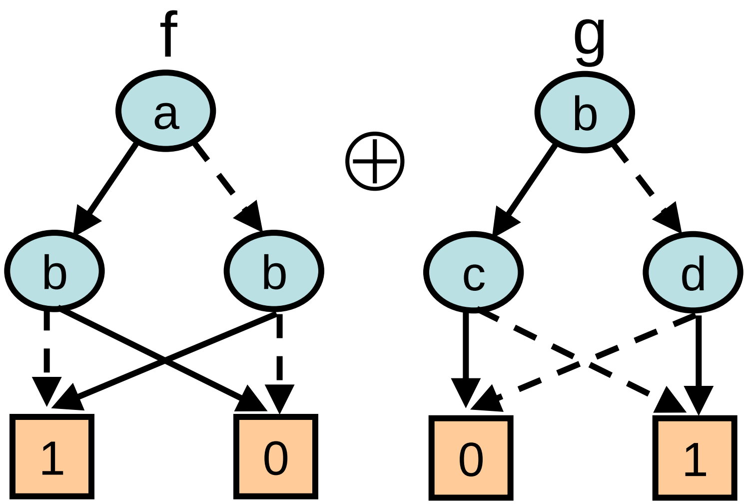
DD Definition

- Decision diagrams (DDs):
  - DAG variant of decision tree
  - Decision tests ordered
  - Used to represent:
    - f: $B^n \rightarrow B$ (boolean – BDD, set of subsets {{a,b},{a}} – ZDD)
    - f: $B^n \rightarrow Z$ (integer – MTBDD / ADD)
    - f: $B^n \rightarrow R$ (real – ADD)



more expressive domains / ranges possible – @ end

What's the Big Deal?



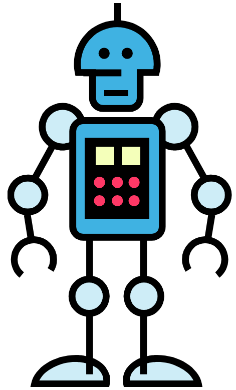
- More than compactness
  - Ordered decision tests in DDs support efficient operations
    - ADD: -f, f $\oplus$ g, f $\otimes$ g, max(f, g)
    - BDD: $\neg$f, f $\wedge$ g, f $\vee$ g
    - ZDD: f \ g, f $\cap$ g, f $\cup$ g
  - Efficient operations key to planning / inference

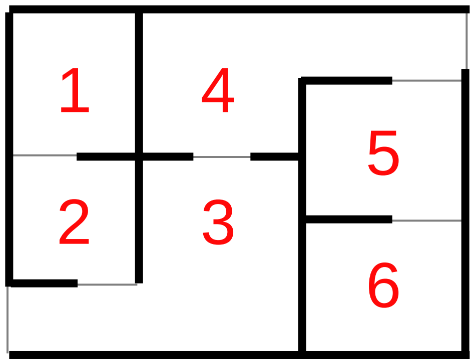
Tutorial Outline

- Need for $B^n \rightarrow B$ / Z / R & operations in planning
- DDs for representing $B^n \rightarrow B$ / Z / R
  - Why important?
  - What can they represent compactly?
  - How to do efficient operations?
- Extensions and Software
  - ZDDs, AADDs, (F)EVBDDs …
- DDs vs. Compilation (d-DNNF)

Factored Representations

- Natural state representations in planning





- State is inherently factored
  - Room location: R = {1,2,3,4,5,6}
  - Door status: D_i={*closed/0,open/1*}; i=1..7
- Relational fluents, e.g., At(r_1,6), (STRIPS) are ground variable templates: at-r1-6

For simplicity we will assume all state vars are boolean {0,1} – all DD ideas generalize to multi-valued case

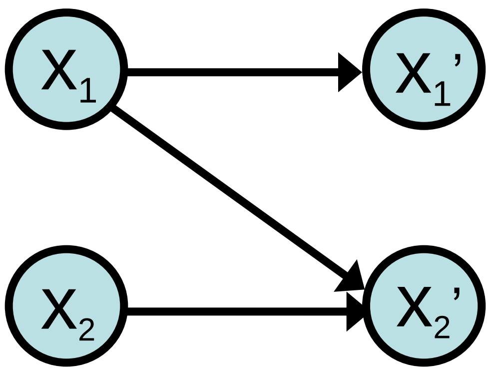
Using Factored State in Planning

- Classical planning
  - State given by variable assignments
    - $(R=1, D_1=o, D_2=c, \ldots, D_7=o)$
  - Planning operators efficiently update state
  - Satisficing tracks dominated by search-based algorithms
    - But representation of $B^n \rightarrow B / Z / R$ important for optimal tracks

- Non-det. / probabilistic planning, temporal verification
  - To compute *progressions* and *regressions*, often need:
    - State sets: $B^n \rightarrow B$ (states satisfying condition)
    - Policies: $B^n \rightarrow Z$ (action ids $\rightarrow$ Z)
    - Value functions: $B^n \rightarrow R$
  - And operations on these functions

Factored Transition Systems I

- If have factored state
  - exploit factored transition systems with *graphical model* (arcs encode dependences)



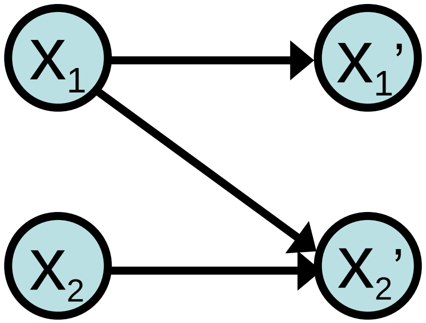
x_1	x_2	x_2'	T/P
0	0	0	1
0	0	1	0
0	1	0	0
	…		…

- Can represent
  - (Non-)deterministic transitions
    - $T(x_1' \mid x_1, x_2): (x_1', x_1, x_2) \rightarrow B$
  - Probabilistic transitions
    - $P(x_1' \mid x_1, x_2): (x_1', x_1, x_2) \rightarrow R$ (really [0,1])

How is table different for det / non-det cases?

Factored Transition Systems II

- (Non-)det. transition systems
  - Forward reachability (FR) / backward reachability (BR)



- **Progression:**
  - given a single state $x_1=0, x_2=1$
    - » $FR(x_1', x_2') = T(x_1' | x_1=0, x_2=1) \wedge T(x_2' | x_2=1)$
  - given a set of possible states S: $(x_1, x_2) \rightarrow B$
    - » $FR(x_1', x_2') = \exists x_1 \exists x_2\ T(x_1' | x_1, x_2) \wedge T(x_2' | x_2) \wedge S(x_1, x_2)$
  - Note: $\exists x\ F(x, \ldots) = F(x=1, \ldots) \vee F(x=0, \ldots)$

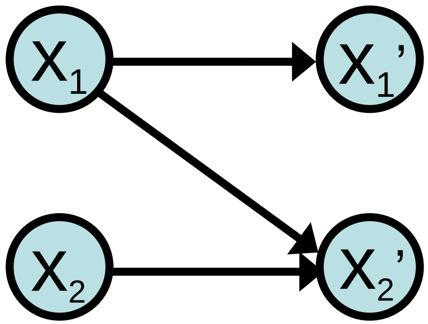
Conformant planning

When use $\forall$?

- **Regression:** given goal function G: $(x_1', x_2') \rightarrow B$
  - $BR(x_1, x_2) = \exists x_1' \exists x_2'\ T(x_1' | x_1, x_2) \wedge T(x_2' | x_2) \wedge G(x_1', x_2')$

Factored Transition Systems III

- Probabilistic transition systems



P(x_1, x_2) can be {0,1} if prev. state known

  - **State updates:** given $P(x_1, x_2)$
    - State sample: $x_1' \sim P(x_1')$: $\sum_{x1} \sum_{x2} P(x_1' | x_1, x_2) \otimes P(x_1, x_2)$
      $x_2' \sim P(x_2')$: $\sum_{x1} \sum_{x2} P(x_2' | x_2) \otimes P(x_1, x_2)$
    - Note: $\sum_x F(x, \ldots) = F(x=1, \ldots) \oplus F(x=0, \ldots)$
    - State belief update:
      $$P(x_1', x_2') = \sum_{x1} \sum_{x2} P(x_1' | x_1, x_2) \otimes P(x_2' | x_2) \otimes P(x_1, x_2)$$

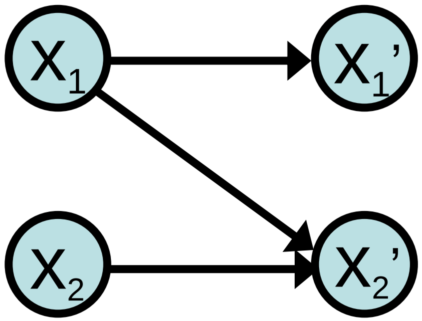
Decision-theoretic regression

  - **DTR:** given value $V'(x_1', x_2')$, compute $E[V](x_1, x_2)$
    - $V(x_1, x_2) = \sum_{x1'} \sum_{x2'} P(x_1' | x_1, x_2) \otimes P(x_2' | x_2) \otimes V'(x_1', x_2')$

Avoids state enum

Factored Transition Systems IV

- Adversarial transition systems

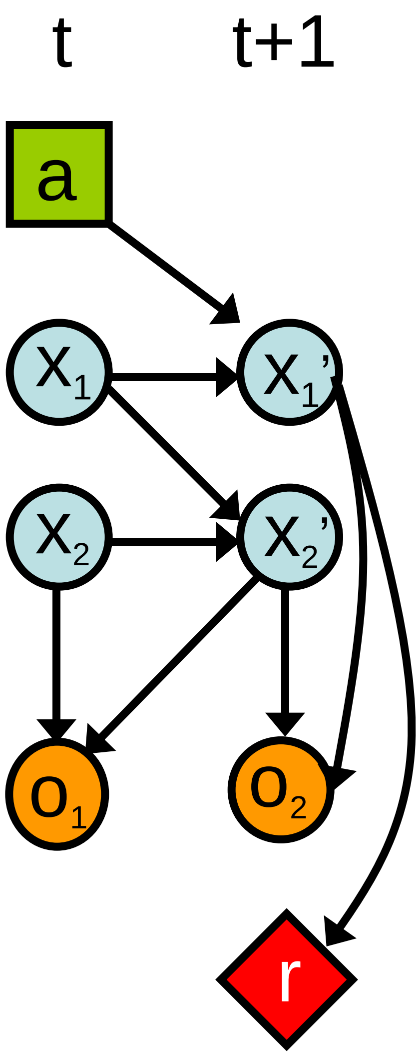


  - **Adversarial DTR**
    - Given value $V'(x_1', x_2')$, compute $E[V](x_1, x_2)$
    - Opponent chooses *non-det.* transitions to minimize V
      - $V(x_1, x_2) = \min_{x1'} \min_{x2'} T(x_1' | x_1, x_2) \otimes T(x_2' | x_2) \otimes V'(x_1', x_2')$
    - Note: $\min_x F(x, \ldots) = \min(F(x{=}1, \ldots), F(x{=}0, \ldots))$

In a zero-sum setting

  - Many other multi-agent formalizations
    - Often alternating turns with action variables…

Factored / Symbolic Planning Approaches



- Classical and Adversarial planning
  - Classical: recent work by Torralba, Alcázar, *et al*
  - Games: Gamer (*Kissmann, Edelkamp*)
    http://www.tzi.de/~kissmann/publications/
- (Non-det) planning
  - Planning as model checking
  - Conformant planning
  - Temporal verification, e.g., x_1 Until x_2?
    (*Bertoli, Cimatti, Pistore, Roveri, Traverso, ...*)
    see refs @ http://mbp.fbk.eu/AIPS02-tutorial.html
- Probabilistic planning
  - MDPs: SPUDD (*Hoey, Boutilier et al*)
    http://www.cs.uwaterloo.ca/~jhoey/research/spudd/index.php
  - POMDPs: Symbolic Perseus (*Poupart et al*)
    http://www.cs.uwaterloo.ca/~ppoupart/software.html

All use of $B^n \rightarrow B$ / Z / R in representation
All planning as operations on these functions

OK, we need $B^n \rightarrow B / Z / R$ for Planning

But why Decision Diagrams?

Function Representation (Tables)

- How to represent functions: $B^n \rightarrow R$?
- How about a fully enumerated table…
- …OK, how to do operations?

a	b	c	F(a,b,c)
0	0	0	0.00
0	0	1	0.00
0	1	0	0.00
0	1	1	1.00
1	0	0	0.00
1	0	1	1.00
1	1	0	0.00
1	1	1	1.00

Manipulating Discrete Distributions

- Marginalization

$$\sum_b P(A,b) \quad = \quad P(A)$$

$$\sum_b$$

A	*B*	*Pr*
0	*0*	*.1*
0	*1*	*.3*
1	*0*	*.4*
1	*1*	*.2*

$=$

A	*Pr*
0	*.4*
1	*.6*

Manipulating Discrete Distributions

- Maximization

$$\max_b P(A, b) \quad = \quad P(A)$$

$\max_b$

A	B	Pr
0	0	.1
0	1	.3
1	0	.4
1	1	.2

=

A	Pr
0	.3 (B=1)
1	.4 (B=0)

Manipulating Discrete Distributions

- Binary Multiplication

$$P(A|B) \quad \cdot \quad P(B|C) \quad = \quad P(A,B|C)$$

A	B	Pr
0	0	.1
0	1	.9
1	0	.2
1	1	.8

$\cdot$

B	C	Pr
0	0	.1
0	1	.9
1	0	.2
1	1	.8

$=$

A	B	C	Pr
0	0	0	.01
0	0	1	.09
0	1	0	.18
0	1	1	.72
...	...	...	...

- Same principle holds for all binary ops
  - +, -, /, max, etc…

Discrete Inference & Optimization

- **Observation 1:** all discrete functions can be tables

P(A,B) =

A	*B*	*Pr*
0	*0*	*.1*
0	*1*	*.3*
1	*0*	*.4*
1	*1*	*.2*

- **Observation 2:** all operations computable in closed-form
  - $f_1 \oplus f_2$, $f_1 \otimes f_2$
  - $\max(f_1 , f_2)$, $\min(f_1, f_2)$
  - $\sum_x f(x)$
  - $(\arg)\max_x f(x)$, $(\arg)\min_x f(x)$

Are we done?
Why do we need DDs?

Why DDs for Planning?

- Reason 1: Space considerations
  - V(Door-1-open, … , Door-40-open) requires ~1 terabyte if all states enumerated
- Reason 2: Time considerations
  - With 1 gigaflop/s. computing power, binary operation on above function requires ~1000 seconds

Function Representation (Tables)

- How to represent functions: $B^n \rightarrow R$?
- How about a fully enumerated table…
- …OK, but can we be more compact?

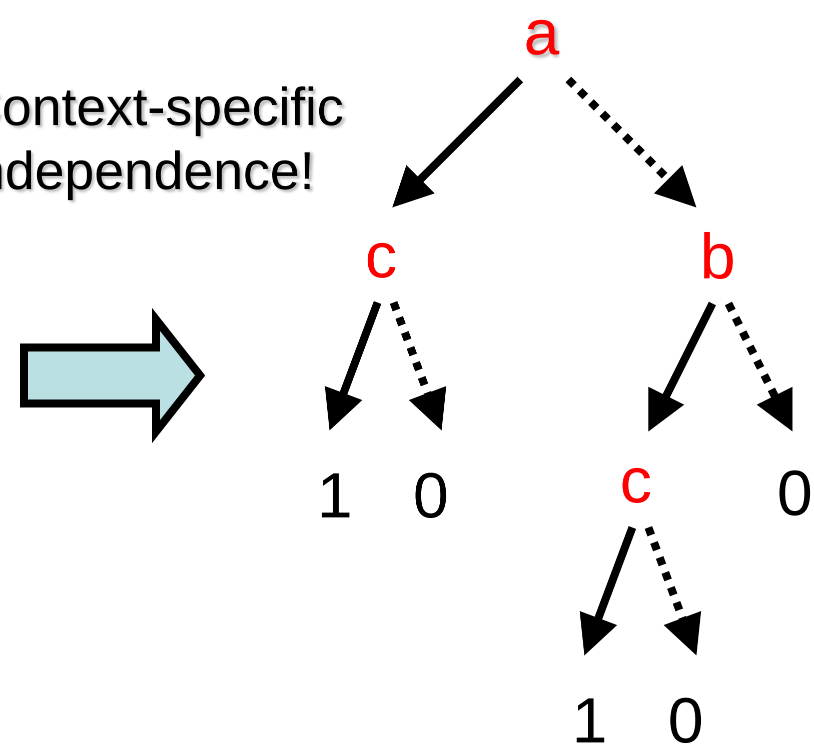
a	b	c	F(a,b,c)
0	0	0	0.00
0	0	1	0.00
0	1	0	0.00
0	1	1	1.00
1	0	0	0.00
1	0	1	1.00
1	1	0	0.00
1	1	1	1.00

Function Representation (Trees)

- How about a tree? Sure, can simplify.

a	b	c	F(a,b,c)
0	0	0	0.00
0	0	1	0.00
0	1	0	0.00
0	1	1	1.00
1	0	0	0.00
1	0	1	1.00
1	1	0	0.00
1	1	1	1.00

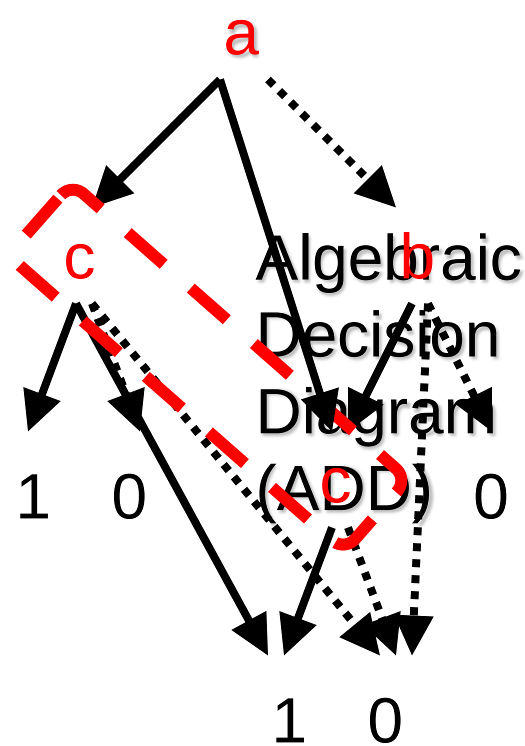
Context-specific independence!



Function Representation (ADDs)

- Why not a directed acyclic graph (DAG)?

a	b	c	F(a,b,c)
0	0	0	0.00
0	0	1	0.00
0	1	0	0.00
0	1	1	1.00
1	0	0	0.00
1	0	1	1.00
1	1	0	0.00
1	1	1	1.00



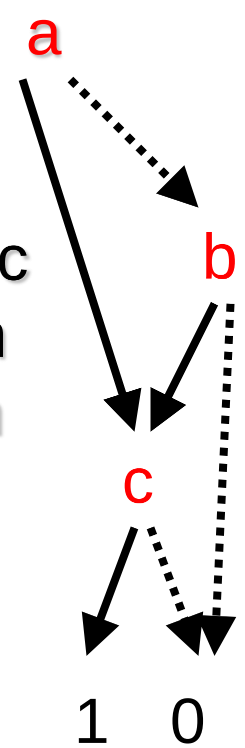
Think of BDDs as {0,1}
subset of ADD range

Function Representation (ADDs)

- Why not a directed acyclic graph (DAG)?

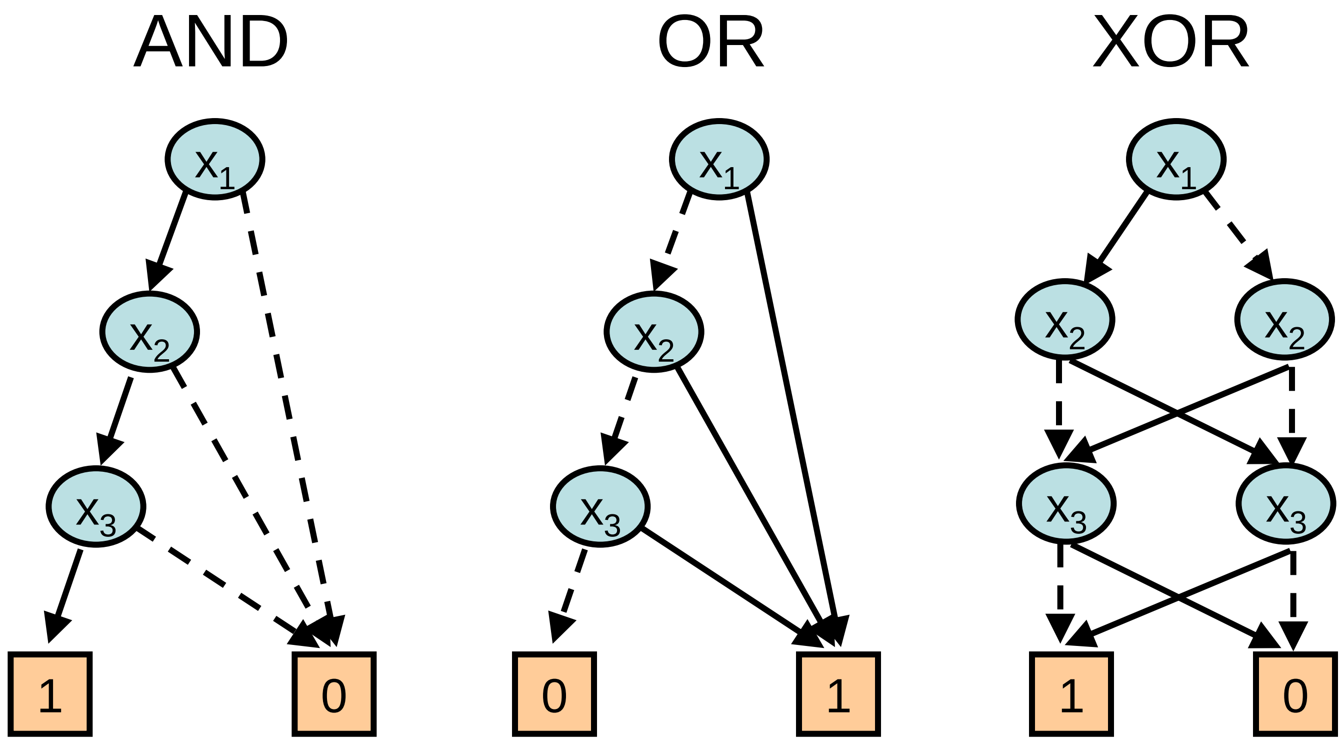
a	b	c	F(a,b,c)
0	0	0	0.00
0	0	1	0.00
0	1	0	0.00
0	1	1	1.00
1	0	0	0.00
1	0	1	1.00
1	1	0	0.00
1	1	1	1.00

Algebraic Decision Diagram (ADD)



Think of BDDs as {0,1} subset of ADD range

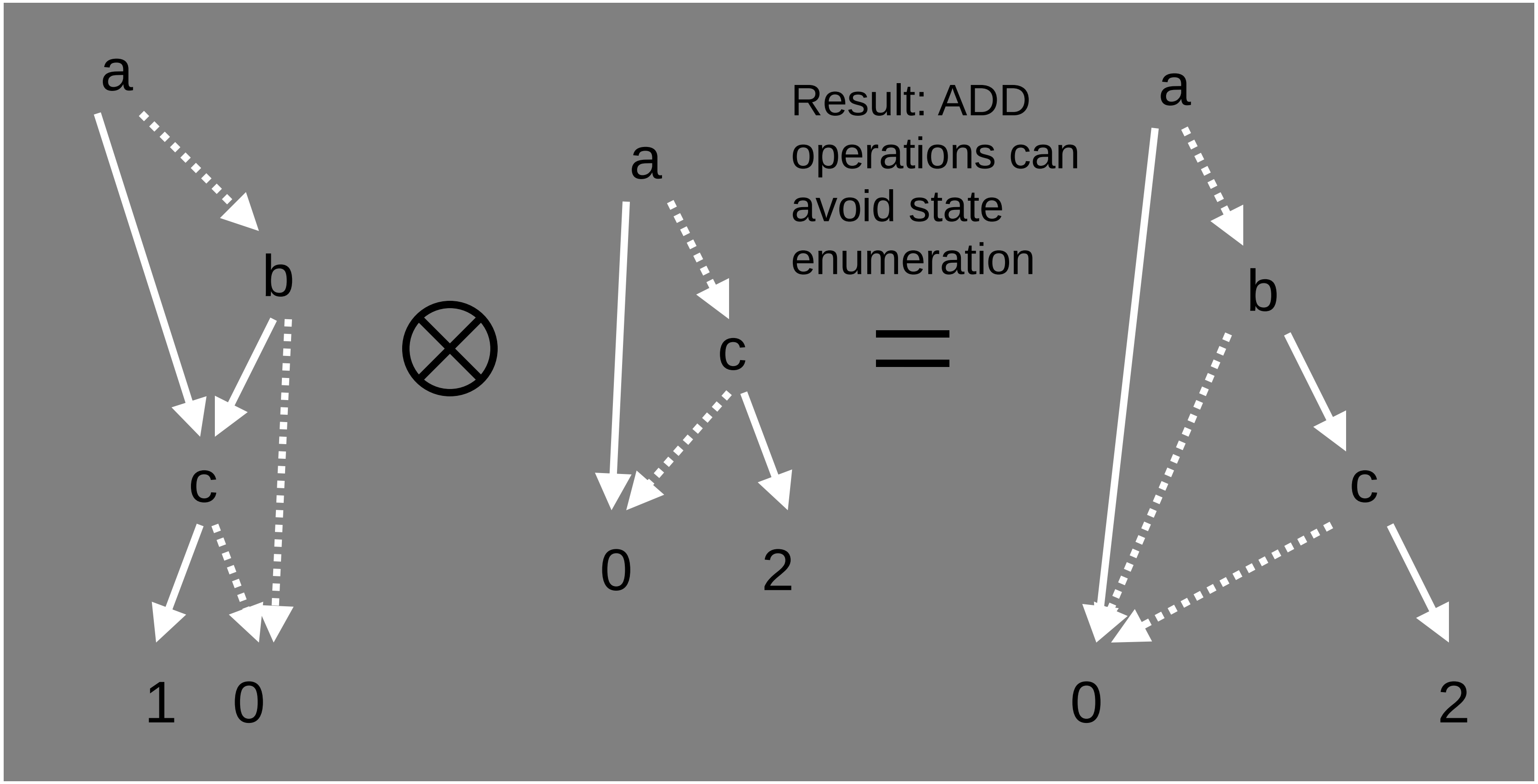
Trees vs. ADDs

- 

- Trees can compactly represent AND / OR
  - But not XOR (linear as ADD, exponential as tree)
  - Why? Trees must represent every path

Binary Operations (ADDs)

- Why do we order variable tests?
- Enables us to do efficient binary operations…



Summary

- We need $B^n \rightarrow B / Z / R$
  - We need compact representations
  - We need efficient operations

$\rightarrow$ DDs are a promising candidate

Not claiming DDs solve all problems… but often better than tabular approach

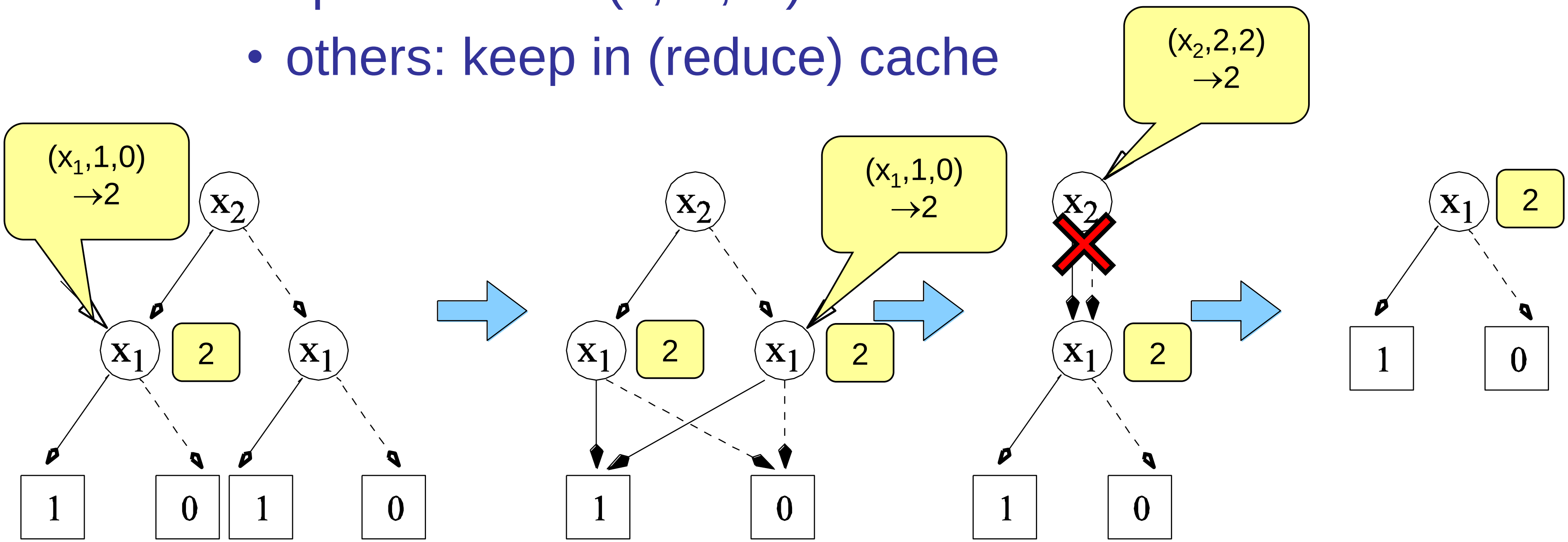
- Great, tell me all about DDs…
  - OK ☺

Decision Diagrams: Reduce

(how to build canonical DDs)

How to Reduce Ordered Tree to ADD?

- Recursively build bottom up
  - Hash terminal nodes R $\rightarrow$ ID
    - leaf cache
  - Hash non-terminal functions (v, ID_0, ID_1) $\rightarrow$ ID
    - special case: (v,ID,ID) $\rightarrow$ ID
    - others: keep in (reduce) cache



Reduce Algorithm

Algorithm 1: $Reduce(F) \longrightarrow F_r$

input : F : Node id
output: F_r : Canonical node id for reduced ADD
begin
  // Check for terminal node
  if *(F is terminal node)* **then**
    return canonical terminal node for value of F;

  // Check reduce cache
  if *($F \to F_r$ is not in reduce cache)* **then**
    // Not in cache, so recurse
    $F_h := Reduce(F_h)$;
    $F_l := Reduce(F_l)$;

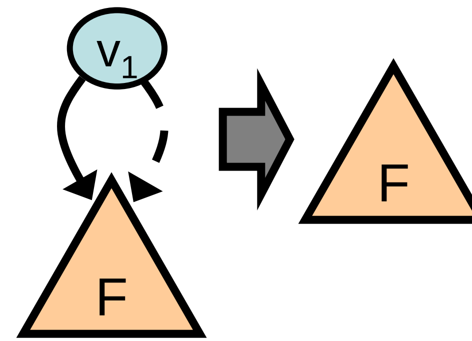
    // Retrieve canonical form
    $F_r := GetNode(F^{var}, F_h, F_l)$;

    // Put in cache
    insert $F \to F_r$ in reduce cache;

  // Return canonical reduced node
  return F_r;
end

GetNode

- Returns unique ID for internal nodes
- Removes redundant branches



Algorithm 1: $GetNode(v, F_h, F_l\rangle) \longrightarrow F_r$

input : v, F_h, F_l : Var and node ids for high/low branches
output: F_r : Return values for offset,
multiplier, and canonical node id

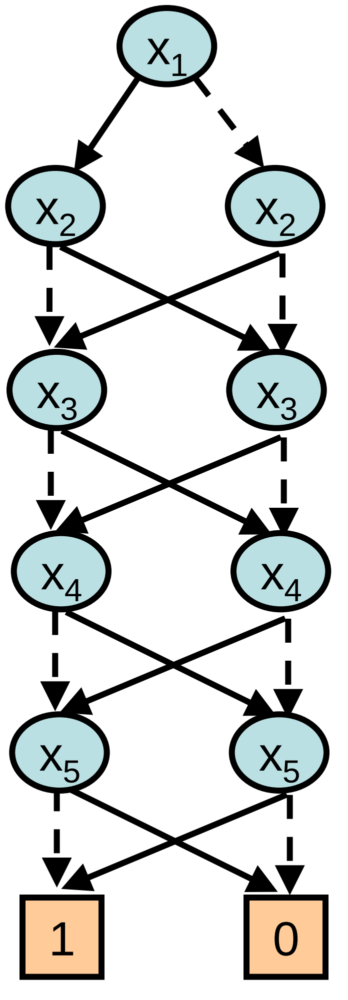
begin
  // If branches redundant, return child
  if $(F_l = F_h)$ **then**
    return F_l;

  // Make new node if not in cache
  if $(\langle v, F_h, F_l \rightarrow id$ *is not in node cache*) **then**
    id := currently unallocated id;
    insert $\langle v, F_h, F_l\rangle\rangle \rightarrow id$ in cache;

  // Return the cached, canonical node
  return id ;
end

Reduce Complexity

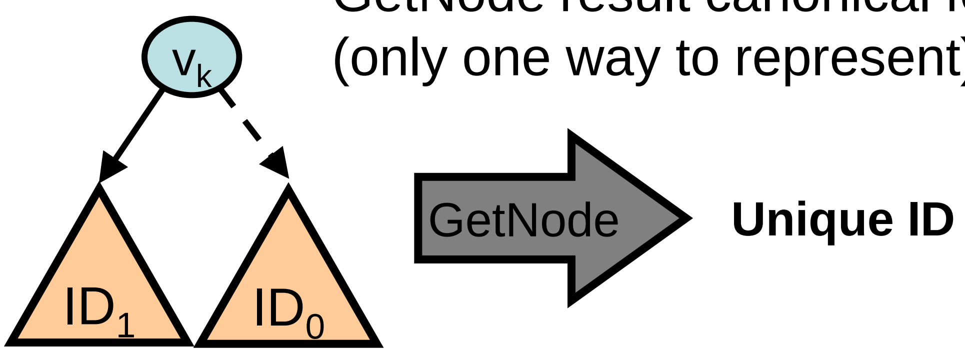
- Linear in size of input
  - Input can be tree or DAG
- Because of caching
  - Explores each node once
  - Does not need to explore all branches

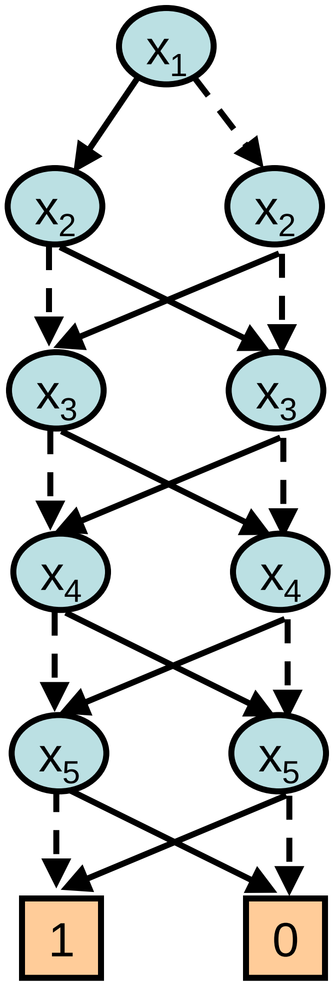


Canonicity of ADDs via Reduce

- Claim: *if two functions are identical, Reduce will assign both functions same ID*

- By induction on var order
  - Base case:
    - Canonical for 0 vars: terminal nodes
  - Inductive:
    - Assume canonical for k-1 vars
    - GetNode result canonical for k^{th} var (only one way to represent)

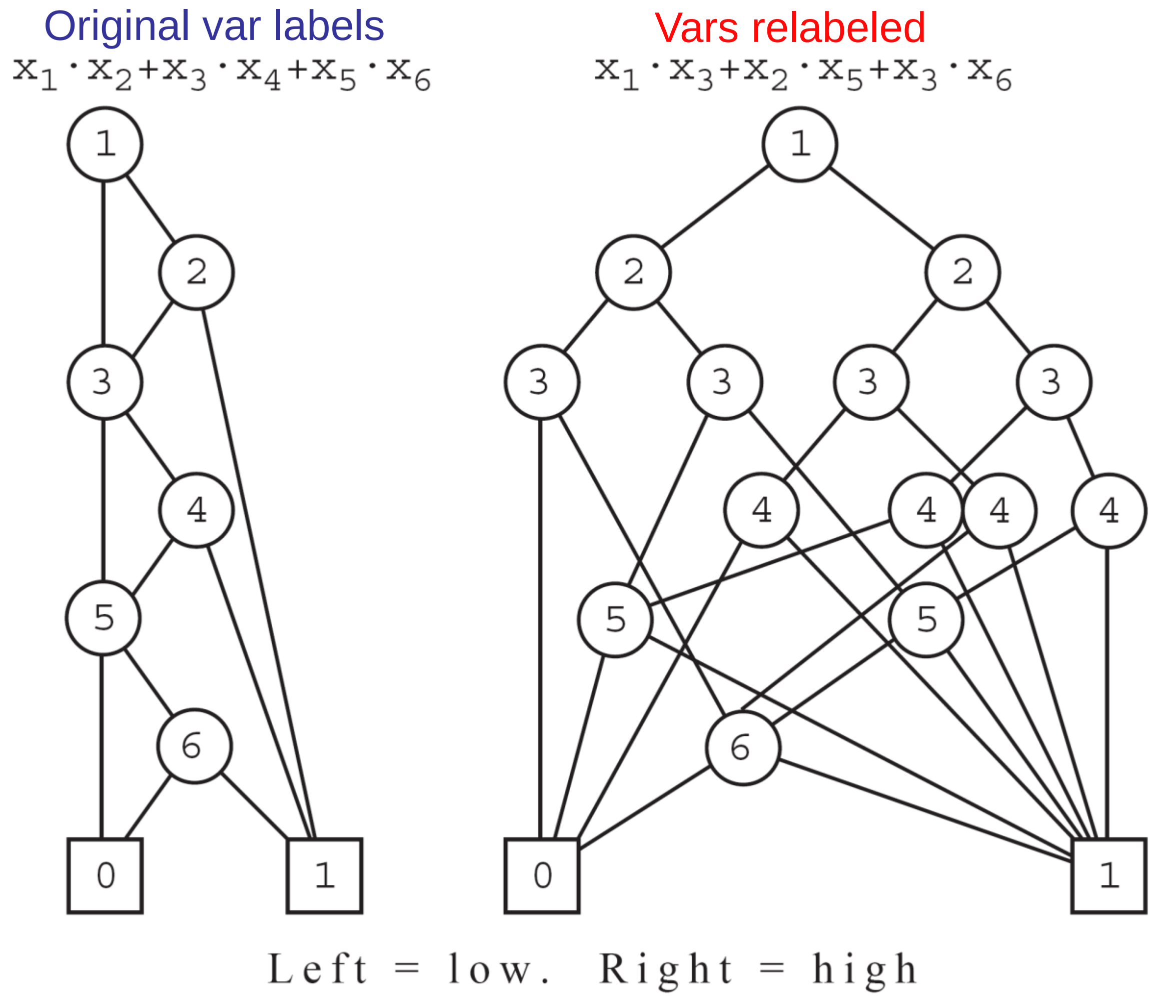




Impact of Variable Orderings

- Good orders can matter

- Good orders typically have related vars together
  - e.g., low tree-width order in transition graphical model

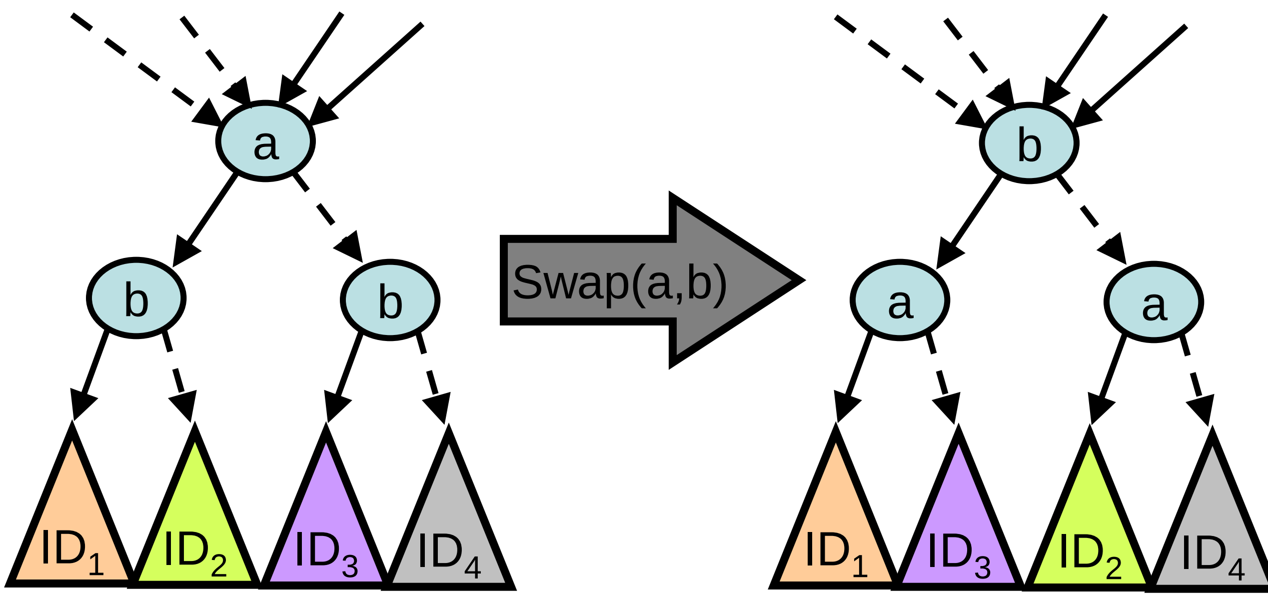


Graph-Based Algorithms for Boolean Function Manipulation
Randal E. Bryant; IEEE Transactions on Computers 1986.

Reordering

- Rudell's sifting algorithm
  - Global reordering as pairwise swapping
  - Only need to redirect arcs
    - Helps to use pointers

      $\rightarrow$ then don't need to redirect parents, e.g.,



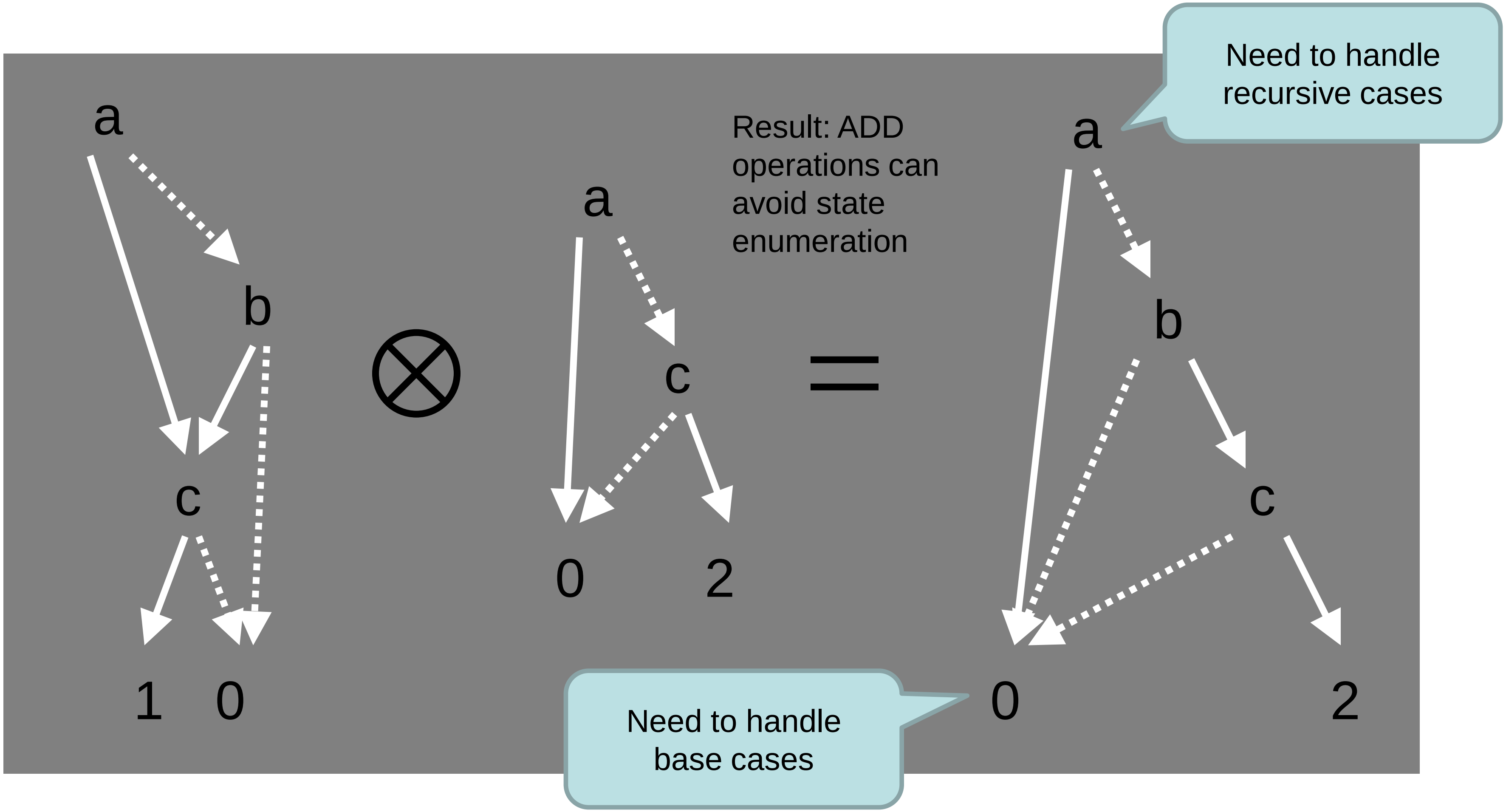
Can also do reorder using Apply… later

Decision Diagrams: Apply

(how to do efficient operations on DDs)

Recap

- Recall the Apply recursion



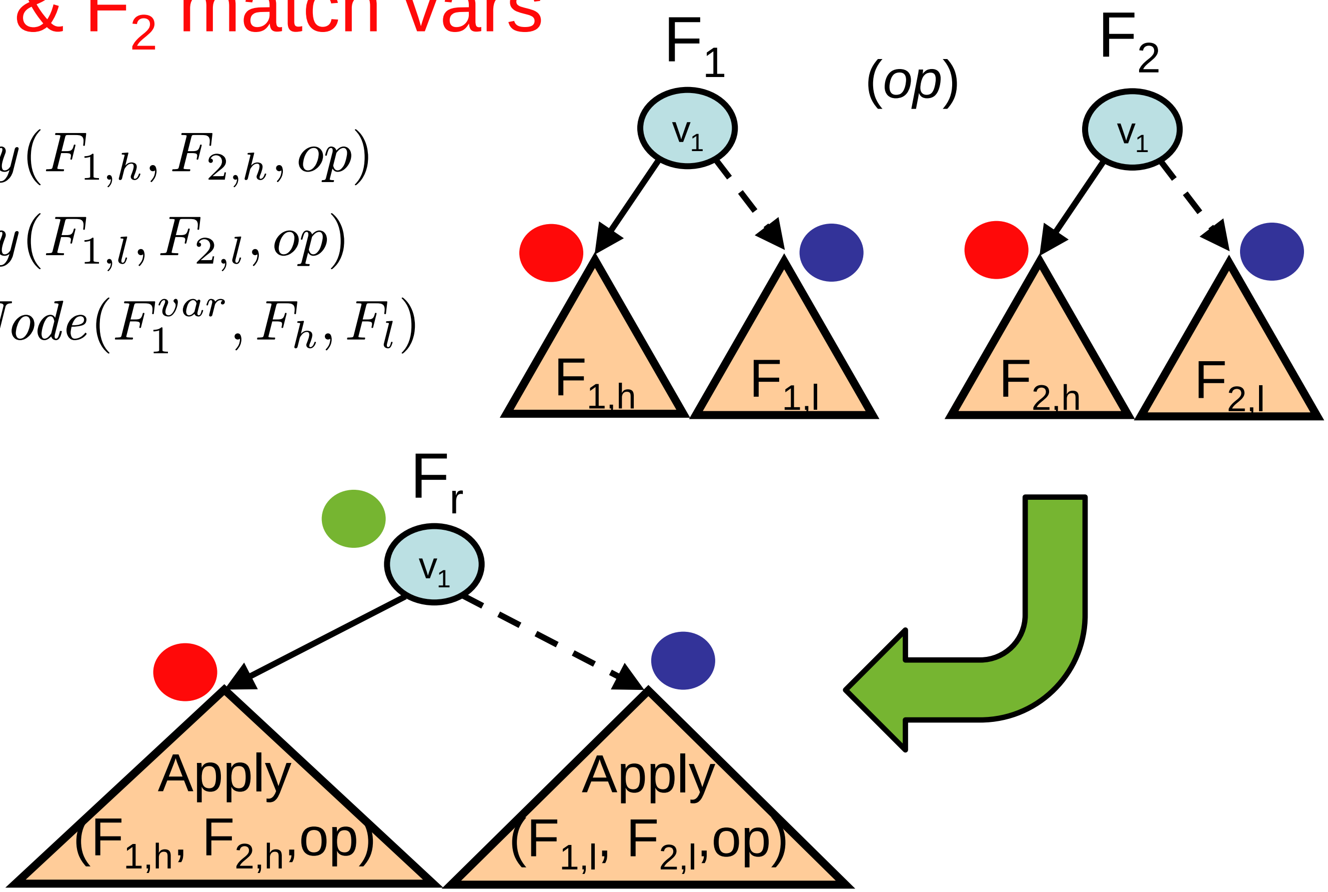
Apply Recursion

- Need to compute F_1 *op* F_2
  - e.g., *op* $\in \{\oplus,\otimes,\wedge,\vee\}$
- Case 1: F_1 & F_2 match vars

$$F_h = Apply(F_{1,h}, F_{2,h}, op)$$

$$F_l = Apply(F_{1,l}, F_{2,l}, op)$$

$$F_r = GetNode(F_1^{var}, F_h, F_l)$$



Apply Recursion

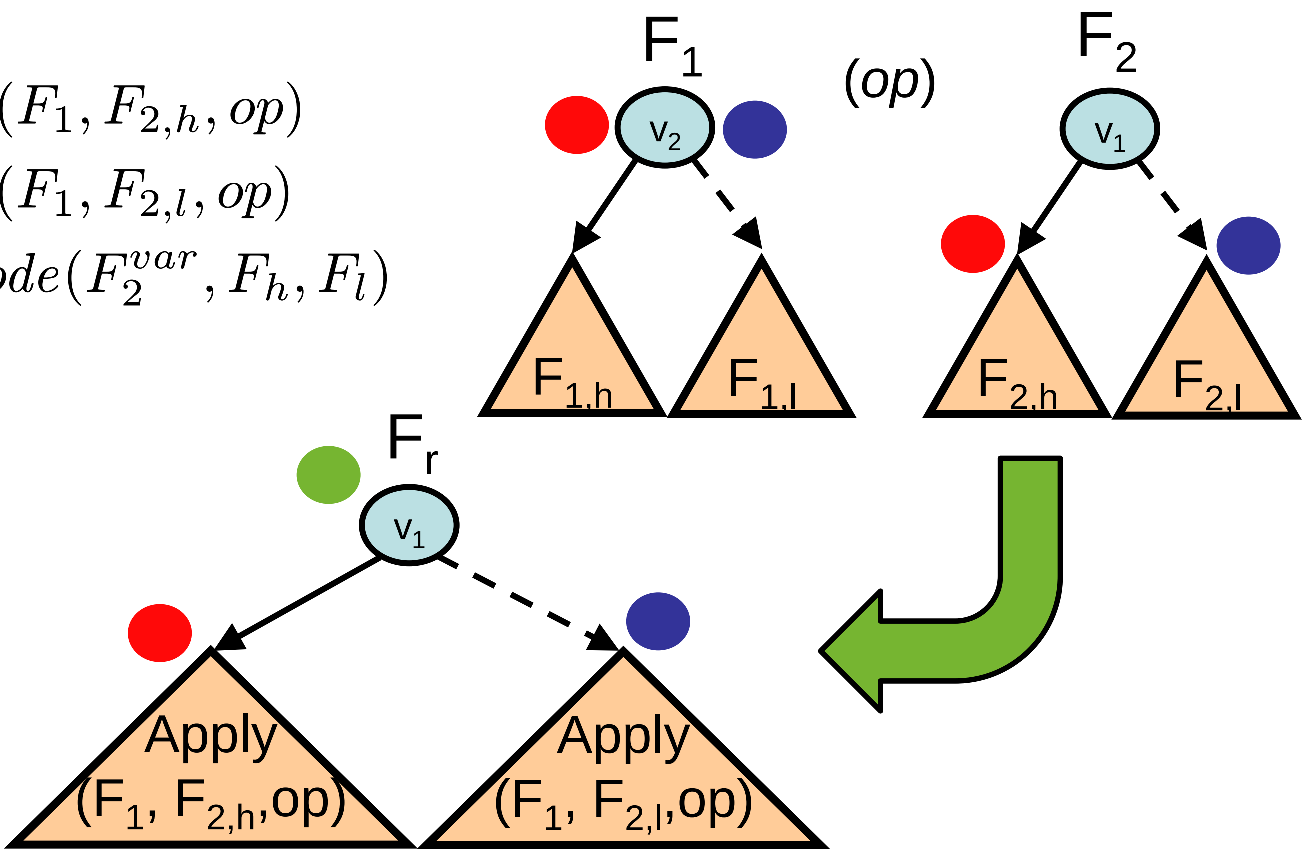
- Need to compute F_1 *op* F_2
  - e.g., *op* $\in \{\oplus, \otimes, \wedge, \vee\}$

- Case 2: Non-matching var: $v_1 \prec v_2$

$$F_h = Apply(F_1, F_{2,h}, op)$$

$$F_l = Apply(F_1, F_{2,l}, op)$$

$$F_r = GetNode(F_2^{var}, F_h, F_l)$$



Apply Base Case: ComputeResult

F_1 (*op*) F_2

- Constant (terminal) nodes and some other cases can be computed without recursion

$ComputeResult(F_1, F_2, op) \longrightarrow F_r$	
Operation and Conditions	**Return Value**
$F_1 \; op \; F_2; \; F_1 = C_1; \; F_2 = C_2$	$C_1 \; op \; C_2$
$F_1 \oplus F_2; \; F_2 = 0$	F_1
$F_1 \oplus F_2; \; F_1 = 0$	F_2
$F_1 \ominus F_2; \; F_2 = 0$	F_1
$F_1 \quad F_2; \; F_2 = 1$	F_1
$F_1 \quad F_2; \; F_1 = 1$	F_2
$F_1 \oslash F_2; \; F_2 = 1$	F_1
$\min(F_1, F_2); \; \max(F_1) \cdot \; \min(F_2)$	F_1
$\min(F_1, F_2); \; \max(F_2) \cdot \; \min(F_1)$	F_2
similarly for max	
other	*null*

Table 1: Input and output summaries of *ComputeResult*.

Apply Algorithm

Note: Apply works for *any* binary operation!

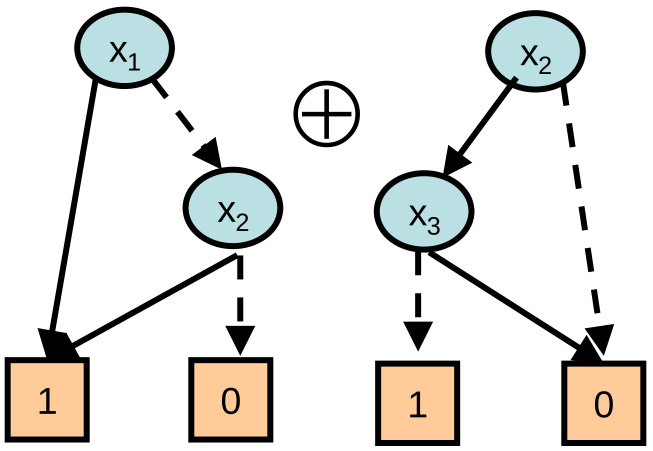
Why?

Algorithm 1: $Apply(F_1, F_2, op) \longrightarrow F_r$

```
input  : F_1, F_2, op : ADD nodes and op
output : F_r : ADD result node to return
begin
    // Check if result can be immediately computed
    if (ComputeResult(F_1, F_2, op) → F_r is not null ) then
        return F_r;
    // Check if result already in apply cache
    if ( ⟨F_1, F_2, op⟩ → F_r is not in apply cache) then
        // Not terminal, so recurse
        var := GetEarliestVar(F_1^var, F_2^var);

        // Set up nodes for recursion
        if (F_1 is non-terminal ∧ var = F_1^var) then
            F_l^v1 := F_{1,l};   F_h^v1 := F_{1,h};
        else
            F_{l/h}^v1 := F_1;

        if (F_2 is non-terminal ∧ var = F_2^var) then
            F_l^v2 := F_{2,l};   F_h^v2 := F_{2,h};
        else
            F_{l/h}^v2 := F_2;

        // Recurse and get cached result
        F_l := Apply(F_l^v1, F_l^v2, op);
        F_h := Apply(F_h^v1, F_h^v2, op);
        F_r := GetNode(var, F_h, F_l);

        // Put result in apply cache and return
        insert ⟨F_1, F_2, op⟩ → F_r into apply cache;
    return F_r;
end
```

Apply Properties

- Apply uses *Apply cache*
  - $(F_1,F_2,op) \rightarrow F_R$
- Complexity
  - Quadratic: $O(|F_1| \cdot |F_2|)$
    - $|F|$ measured in node count
  - Why?
    - Cache implies touch every pair of nodes at most once!

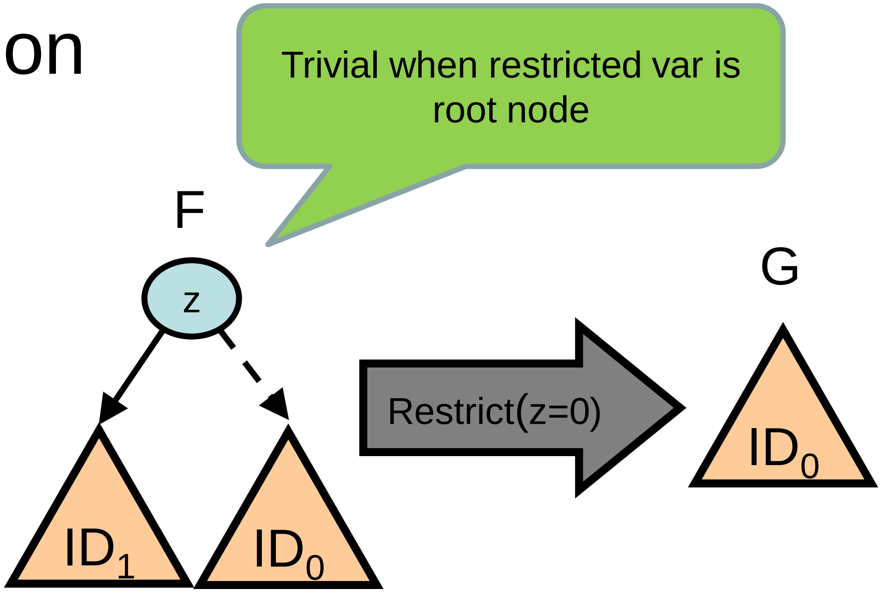


- Canonical?
  - Same inductive argument as Reduce

Reduce-Restrict

- Important operation

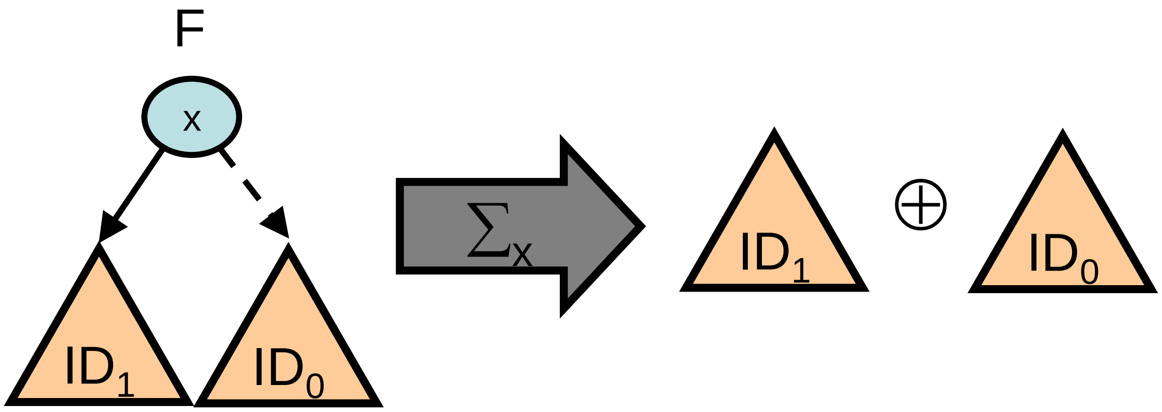
- Have
  - F(x,y,z)
- Want
  - $G(x,y) = F|_{z=0}$



- Restrict $F|_{v=value}$ operation performs a *Reduce*
  - Just returns branch for v=value whenever v reached
  - Need *Restrict-Reduce cache* for O(|F|) complexity

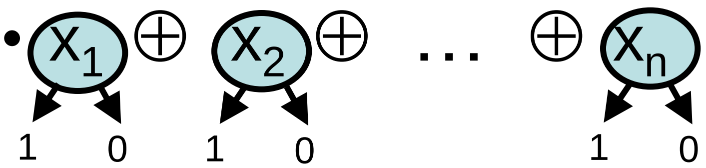
Marginalization, etc.

- Use Apply + Reduce-Restrict
  - $\sum_x F(x, \ldots) = F|_{x=0} \oplus F|_{x=1}$, e.g.



  - Likewise for similar operations…
    - ADD: $\min_x F(x, \ldots) = \min(F|_{x=0}, F|_{x=1})$
    - BDD: $\exists x\, F(x, \ldots) = F|_{x=0} \vee F|_{x=1}$
    - BDD: $\forall x\, F(x, \ldots) = F|_{x=0} \wedge F|_{x=1}$

Apply Tricks I

- Build $F(x_1, \ldots, x_n) = \sum_{i=1..n} x_i$
  - Don't build a tree and then call Reduce!
  - Just use indicator DDs and Apply to compute
    - 

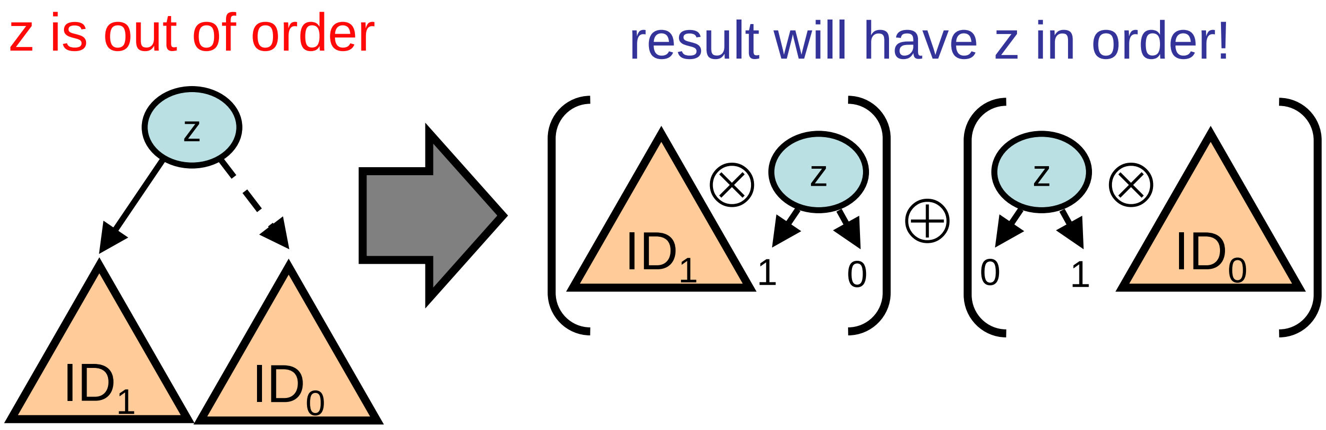
  - In general:
    - Build *any* arithmetic expression bottom-up using Apply!

$x_1 + (x_2 + 4x_3) * (x_4)$

$\rightarrow x_1 \oplus (x_2 \oplus (4 \otimes x_3)) \otimes (x_4)$

Apply Tricks II

- Build *ordered* DD from *unordered* DD



ZDDs

(zero-suppressed BDDs)

Represent sets of subsets

ZDDs for Sets of Subsets

- Example BDD and ZDD

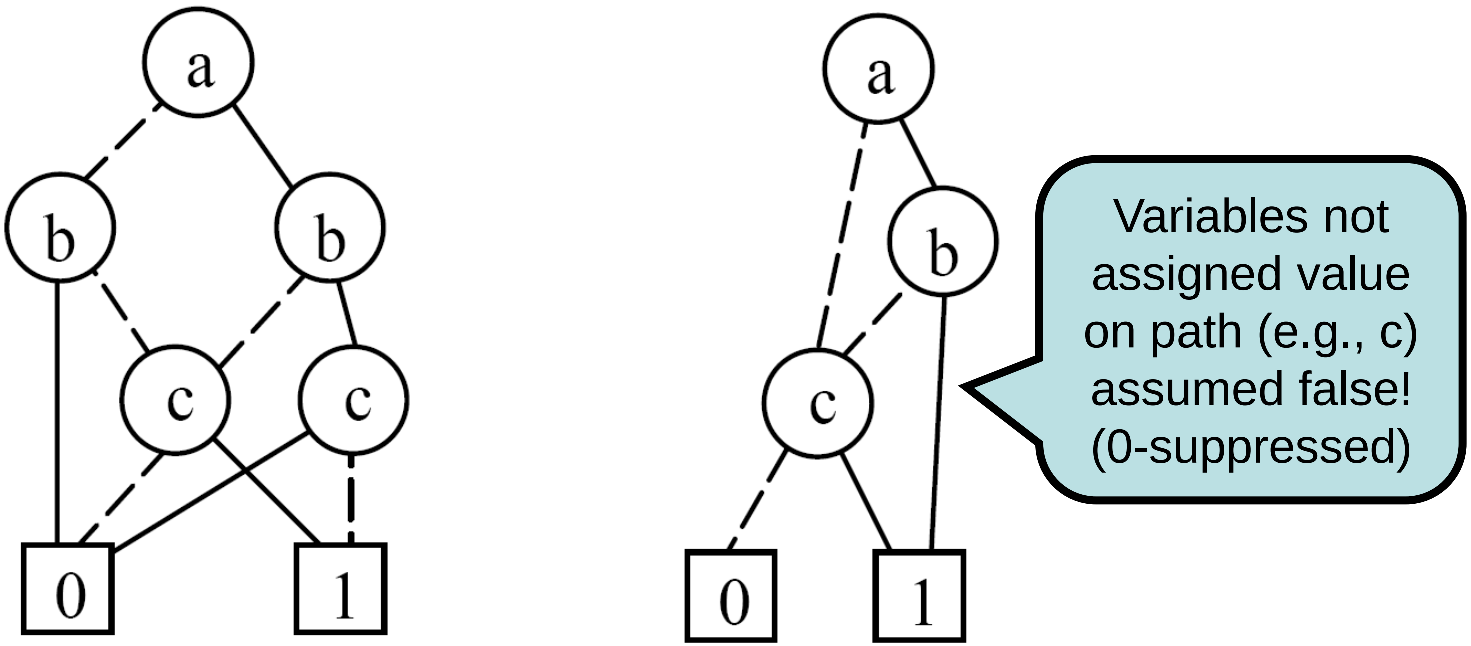


Figure 2. The BDD and the ZDD for the set of subsets $\{\{a,b\}, \{a,c\}, \{c\}\}$.

ZDDs vs. BDDs

- But ZDDs not universal replacement for BDDs…

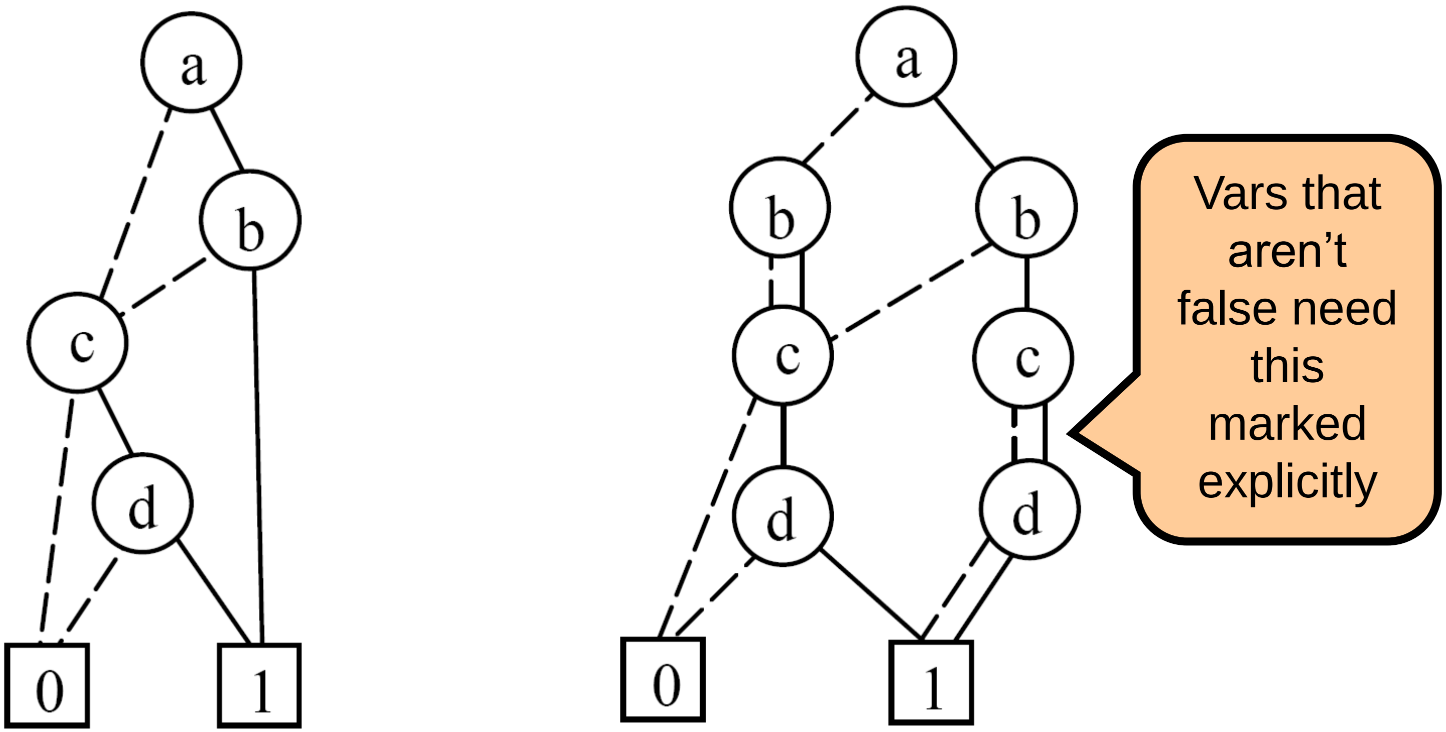
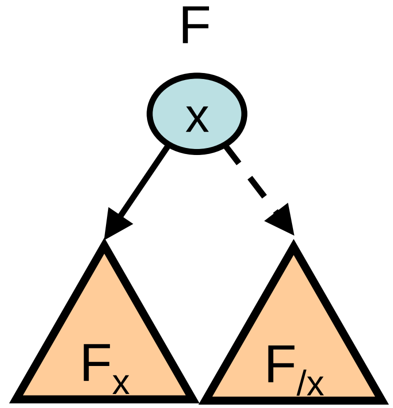


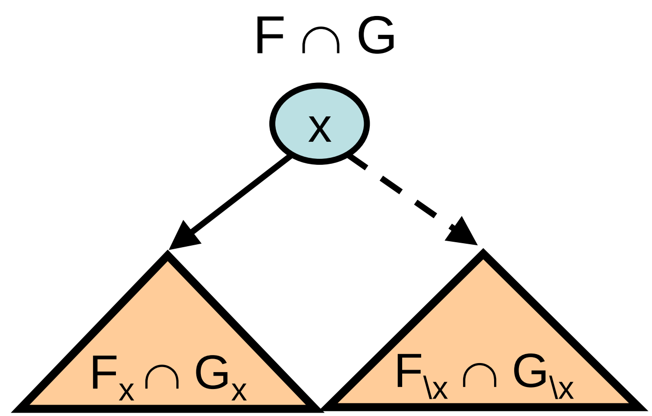
Figure 1. BDD and ZDD for F = ab + cd.

An Introduction to Zero-Suppressed Binary Decision Diagrams
Alan Mishchenko

How to Modify Apply for ZDDs?



- Simple
  - F_x is sub-ZDD for set *with* x
  - $F_{\backslash x}$ is sub-ZDD for set *without* x

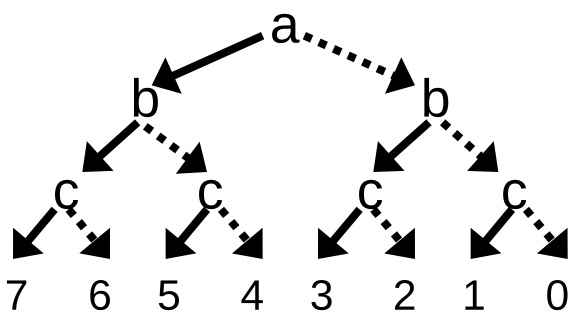


- $F \cap G$:
  - if (x in set)
    - then $F_x \cap G_x$
    - else $F_{\backslash x} \cap G_{\backslash x}$
- This is just standard *Apply*
  - With properly defined GetNode, leaf ops: $\cap = \wedge$, $\cup = \vee$

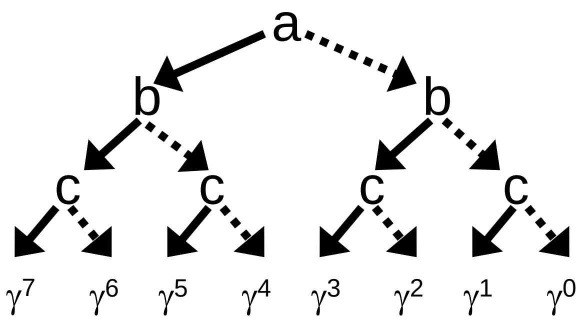
Affine ADDs

ADD Inefficiency

- Are ADDs enough?
- Or do we need more compactness?
- Ex. 1: Additive reward/utility functions
  - $R(a,b,c) = R(a) + R(b) + R(c)$
    $= 4a + 2b + c$



- Ex. 2: Multiplicative value functions
  - $V(a,b,c) = V(a) \cdot V(b) \cdot V(c)$
    $= \gamma^{(4a + 2b + c)}$

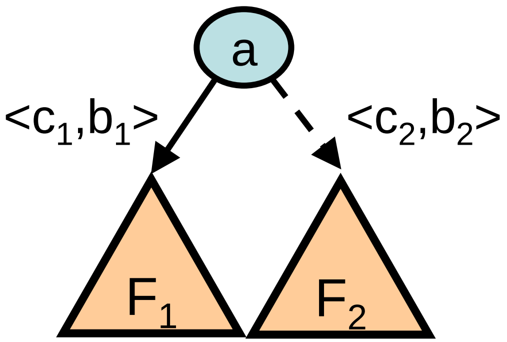


(Sanner, McAllester, IJCAI-05)

Affine ADD (AADD)

- Define a new decision diagram – **Affine ADD**

- Edges labeled by **offset (c)** and **multiplier (b)**:



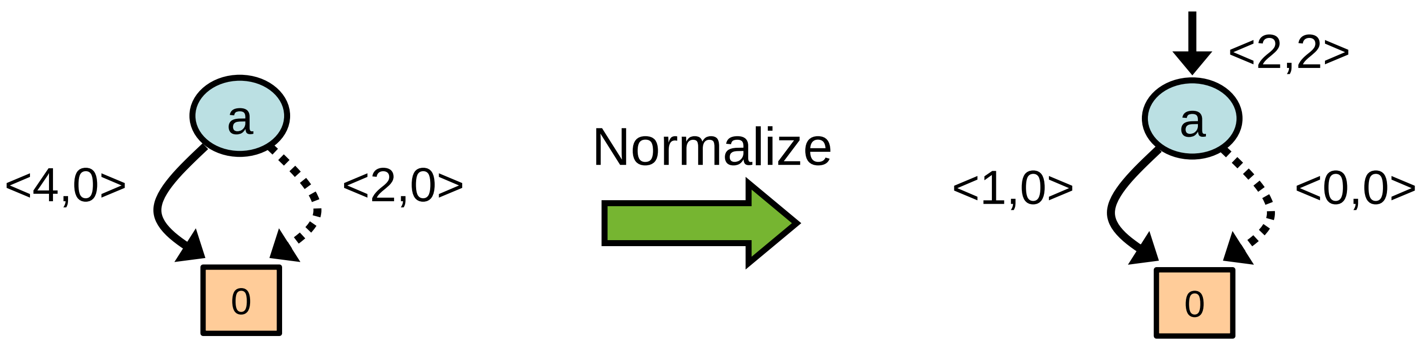
- **Semantics:** if (a) then $(c_1+b_1F_1)$ else $(c_2+b_2F_2)$

Affine ADD (AADD)

- Maximize sharing by **normalizing** nodes [0,1]

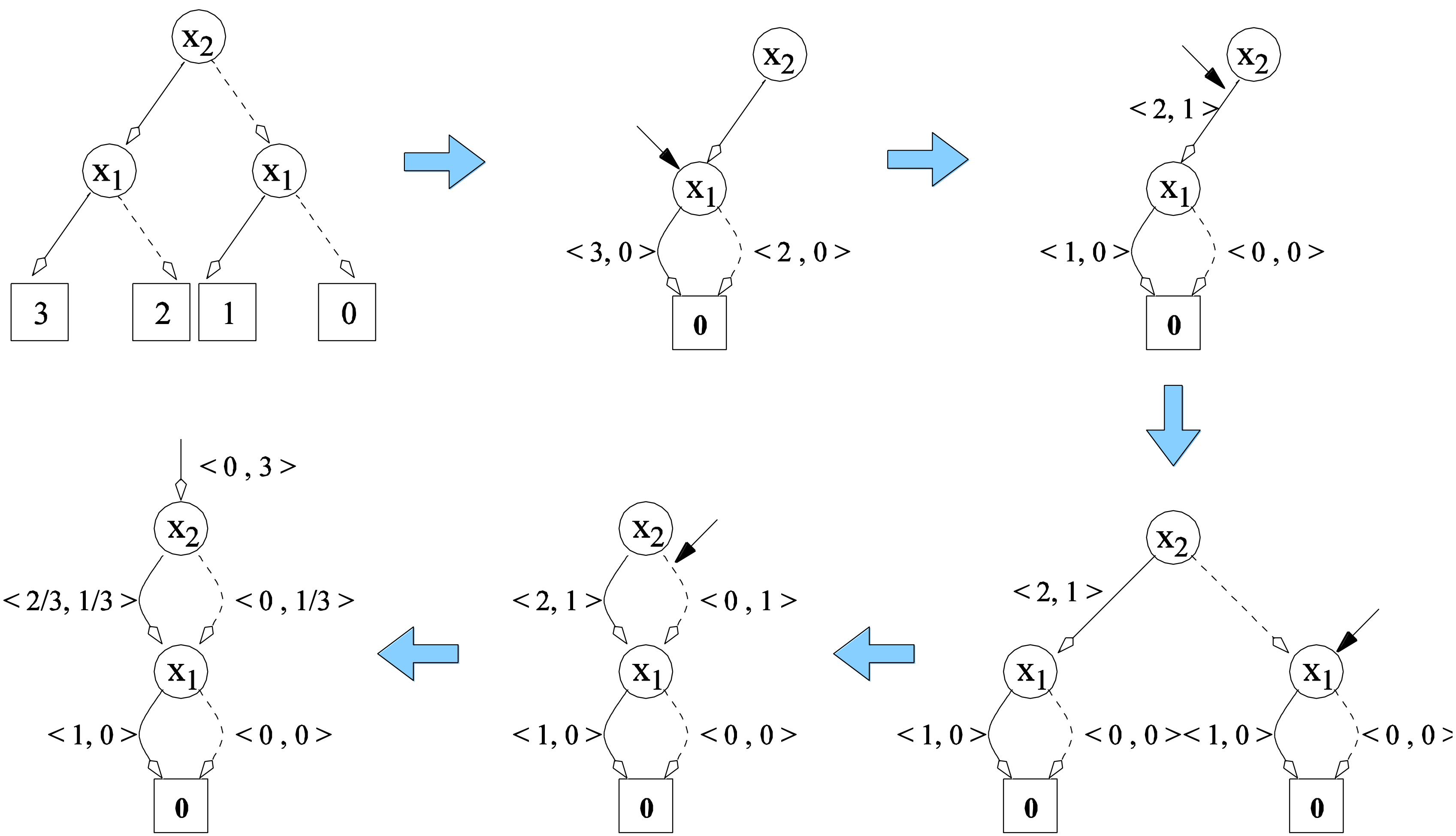
- Example: if (a) then (4) else (2)

Need top-level affine transform to recover original range



AADD Reduce

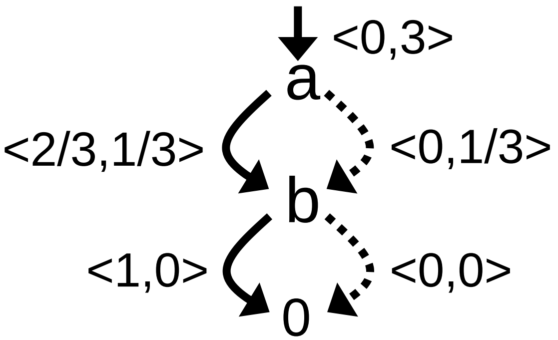
Key point:
automatically finds
additive structure



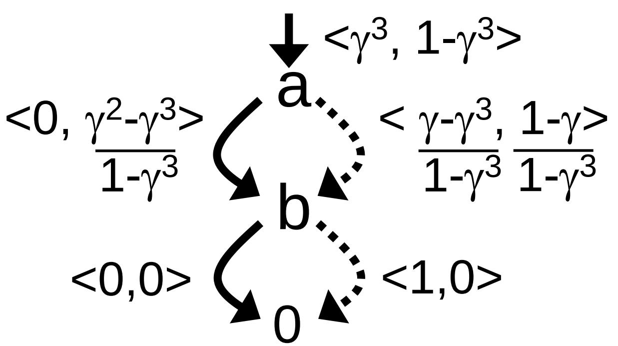
AADD Examples

Automatically Constructed!

- Back to our previous examples…
- Ex. 1: Additive reward/utility functions
  - $R(a,b) = R(a) + R(b)$
    $= 2a + b$



- Ex. 2: Multiplicative value functions
  - $V(a,b) = V(a) \cdot V(b)$
    $= \gamma^{(2a + b)}; \gamma<1$



AADD Apply & Normalized Caching

- Need to normalize Apply cache keys, e.g., given

$$\langle 3 + 4F_1 \rangle \oplus \langle 5 + 6F_2 \rangle$$

- before lookup in Apply cache, normalize

$$8 + 4 \cdot \langle 0 + 1F_1 \rangle \oplus \langle 0 + 1.5F_2 \rangle$$

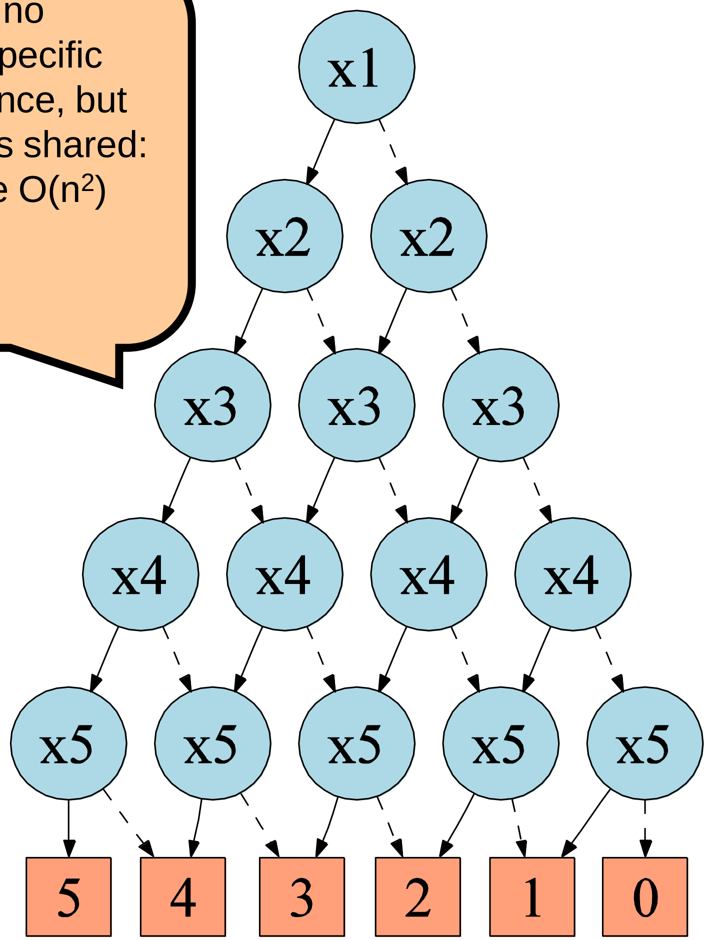
$\mathit{GetNormCacheKey}(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, \mathit{op}) \longrightarrow \langle\langle c_1', b_1'\rangle\langle c_2', b_2'\rangle\rangle$ **and** $\mathit{ModifyResult}(\langle c_r, b_r, F_r\rangle) \longrightarrow \langle c_r', b_r', F_r'\rangle$

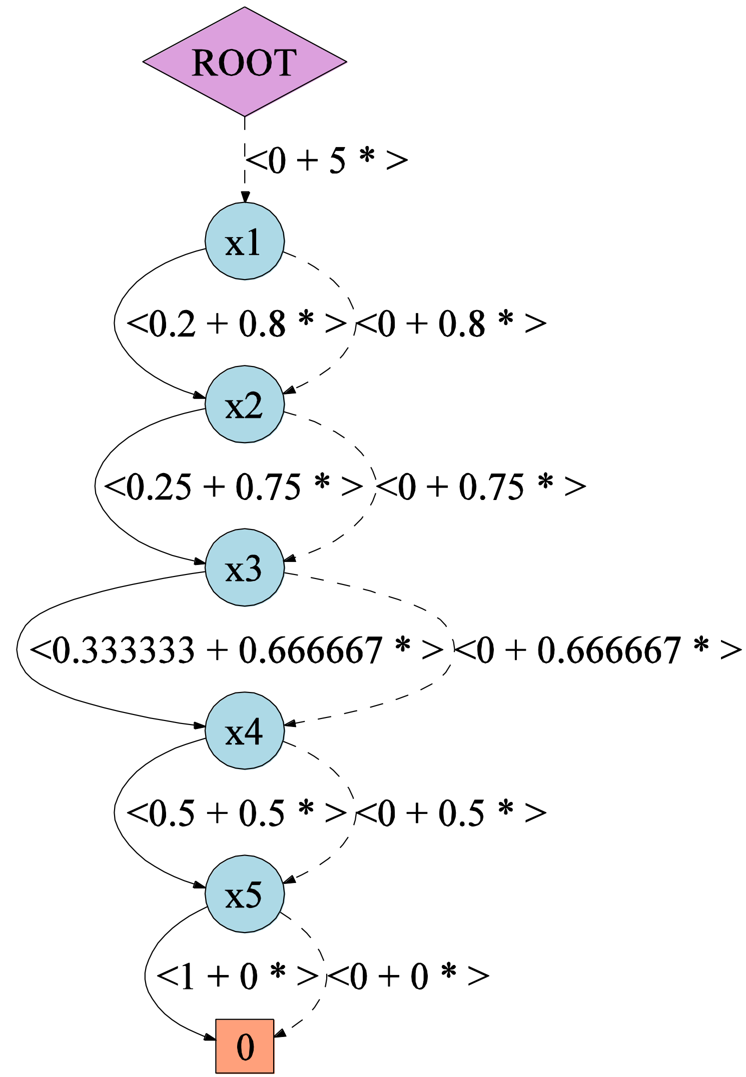
Operation and Conditions	**Normalized Cache Key and Computation**	**Result Modification**
$\langle c_1 + b_1F_1\rangle \oplus \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle 0 + 1F_1\rangle \oplus \langle 0 + (b_2/b_1)F_2\rangle$	$\langle (c_1 + c_2 + b_1c_r) + b_1b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \ominus \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle 0 + 1F_1\rangle \ominus \langle 0 + (b_2/b_1)F_2\rangle$	$\langle (c_1 - c_2 + b_1c_r) + b_1b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \quad \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle (c_1/b_1) + F_1\rangle \quad \langle (c_2/b_2) + F_2\rangle$	$\langle b_1b_2c_r + b_1b_2b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \oslash \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle (c_1/b_1) + F_1\rangle \oslash \langle (c_2/b_2) + F_2\rangle$	$\langle (b_1/b_2)c_r + (b_1/b_2)b_rF_r\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_2\rangle)$; $F_1 \neq 0$, Note: same for min	$\langle c_r + b_rF_r\rangle = \max(\langle 0 + 1F_1\rangle, \langle (c_2 - c_1)/b_1 + (b_2/b_1)F_2\rangle)$	$\langle (c_1 + b_1c_r) + b_1b_rF_r\rangle$
any $\langle \mathit{op}\rangle$ not matching above: $\langle c_1 + b_1F_1\rangle\ \langle \mathit{op}\rangle\ \langle c_2 + b_2F_2\rangle$	$\langle c_r + b_rF_r\rangle = \langle c_1 + b_1F_1\rangle\ \langle \mathit{op}\rangle\ \langle c_2 + b_2F_2\rangle$	$\langle c_r + b_rF_r\rangle$

ADDs vs. AADDs

- Additive functions: $\sum_{i=1..n} x_i$

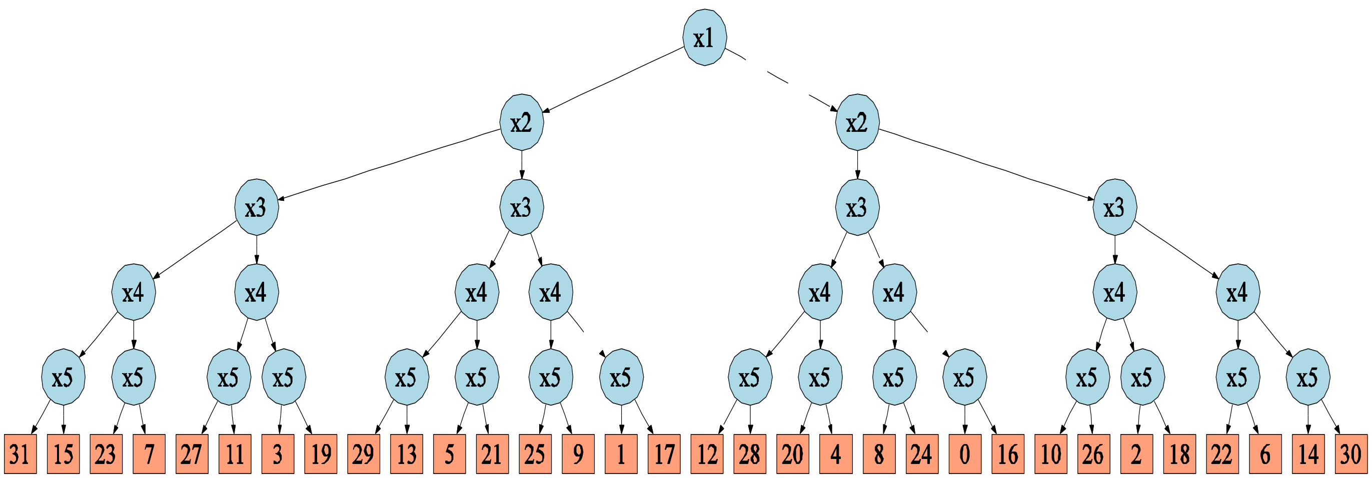
Note: no context-specific independence, but subdiagrams shared: result size O(n²)

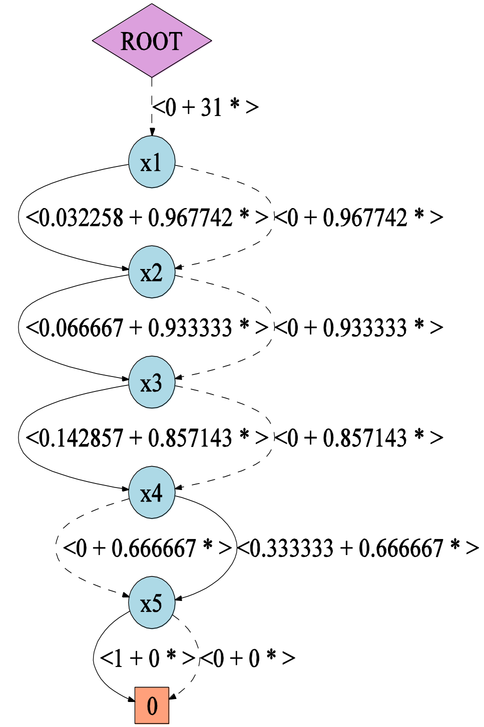




ADDs vs. AADDs

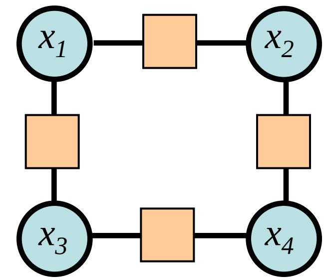
- Additive functions: $\sum_i 2^i x_i$
  - Best case result for ADD (exp.) vs. AADD (linear)

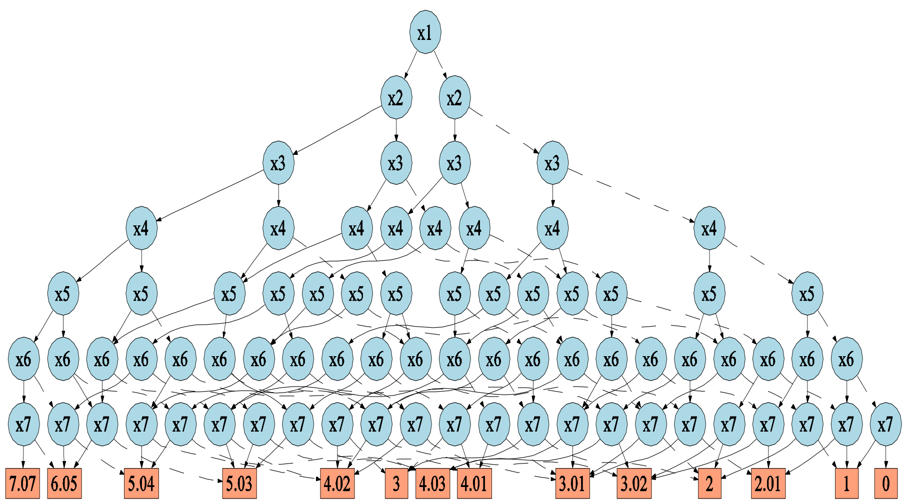




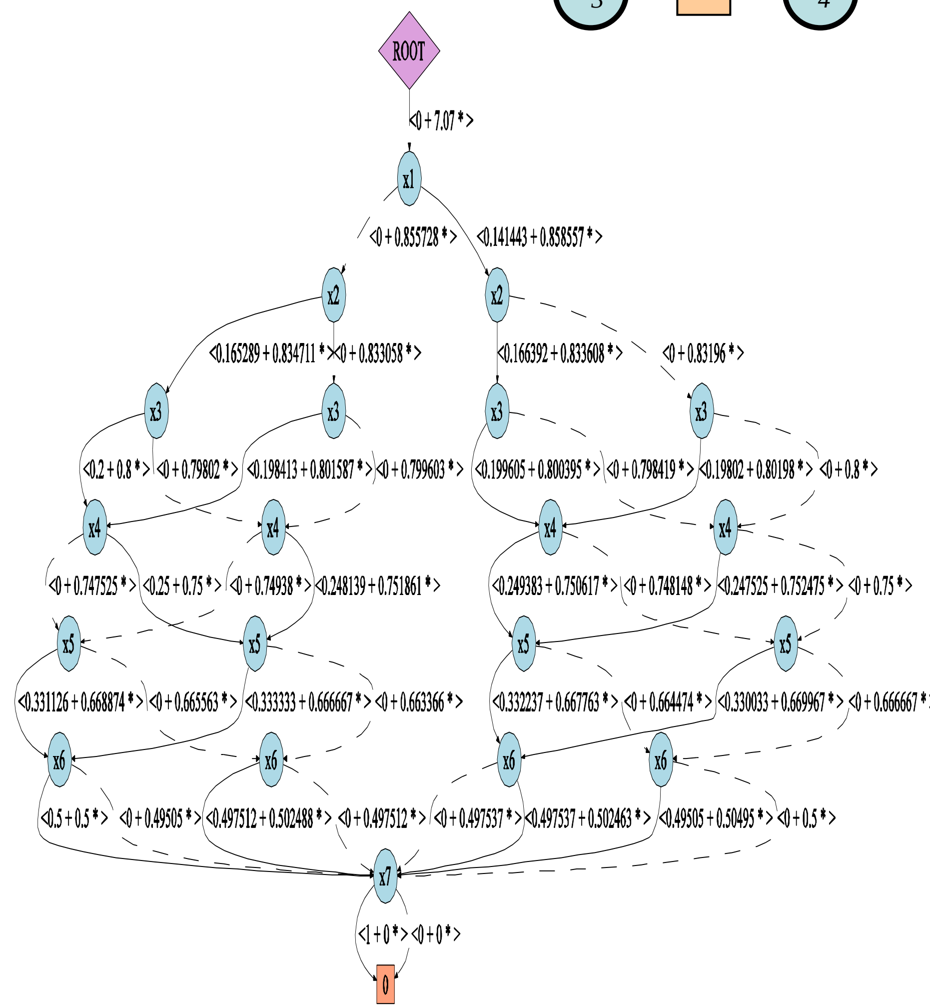
ADDs vs. AADDs

- Additive functions: $\sum_{i=0..n-1} F(x_i, x_{(i+1)\ \%\ n})$



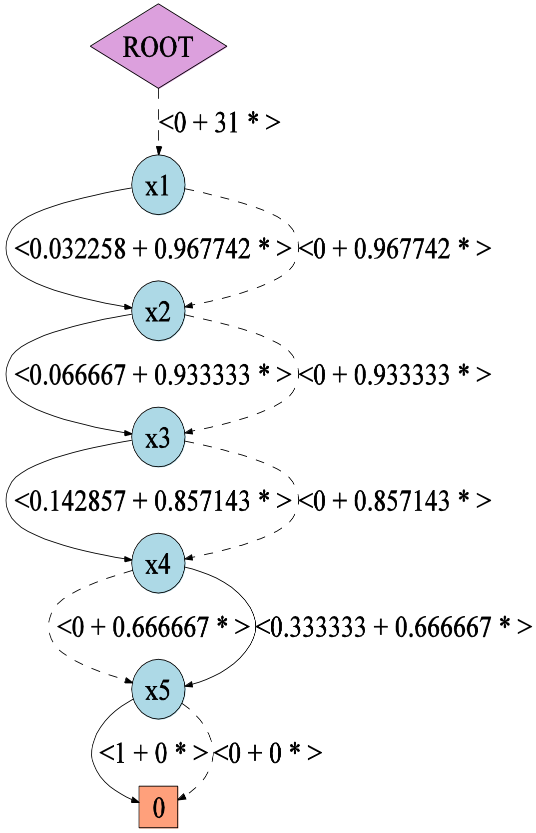


Pairwise factoring evident in AADD structure



Main AADD Theorem

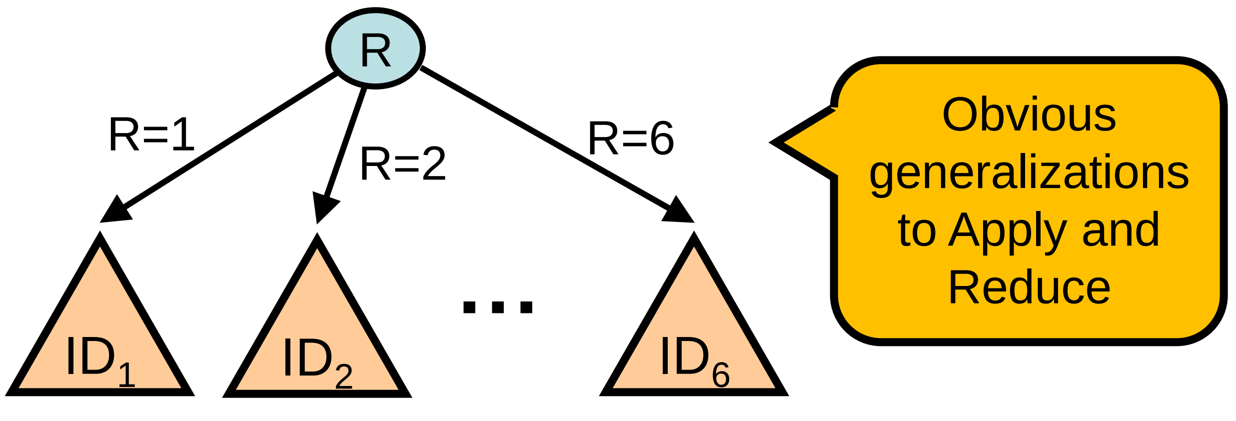
- **AADD can yield exponential time/space improvement over ADD**
  - **and never performs worse!**
- But…
  - Apply operations on AADDs can be exponential
  - Why?
    - Reconvergent diagrams possible in AADDs (edge labels), but not ADDs →
    - Sometimes Apply explores all paths if no hits in normalized Apply cache



Other DDs

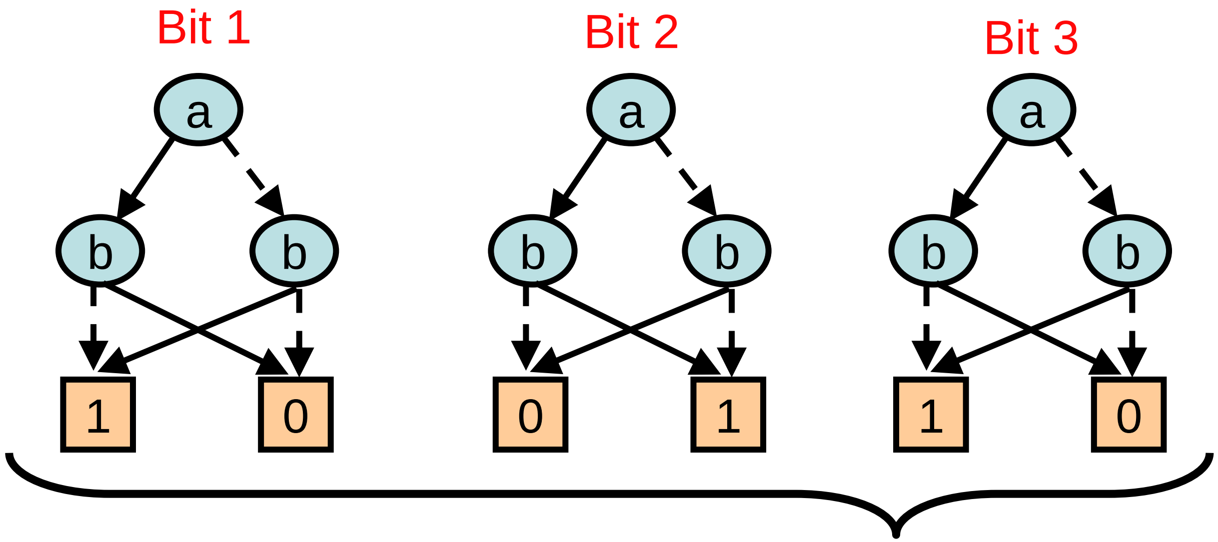
Multivalued (MV-)DD

- A DD with multivalued variables
  - straightforward k-branch extension
  - e.g., k=6

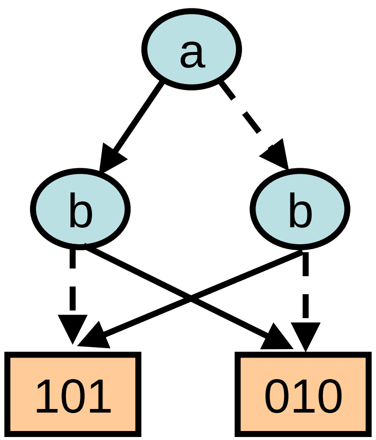


Multi-terminal (MT-)BDD

- Imagine terminal is 3 bits… use 3 BDDs



- MT-BDD – combine into single diagram
  - **Same as ADD** using bit vector (integer) leaves



(F)EV-BDDs

- EdgeValue-BDD is like AADD where only additive constant substracted
  - Not a full affine transform
  - **Better numerical precision properties than AADD**
    - **Additive, but no multiplicative compression like AADD**

- Factor-EVBDD is for integer leaves Z
  - Instead of dividing by range…
    factors out largest prime factor as multiplier

Further Afield

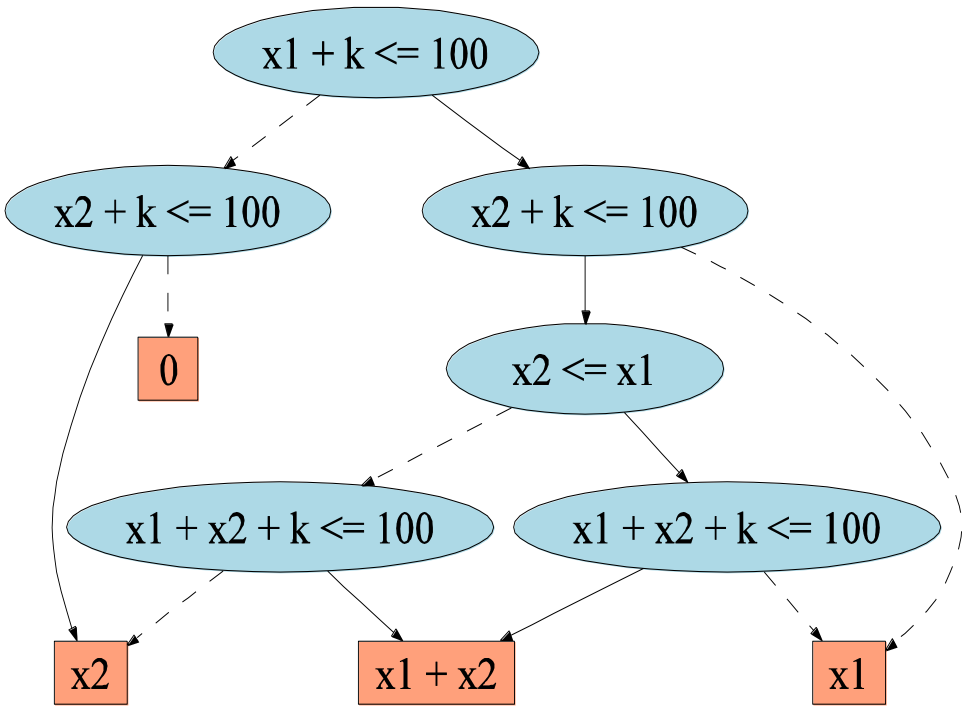
- K*DDs, BMDs, K*BMDs
  - Like ZDD, different ways to do decomposition
  - Mainly used in digitial verification literature

- FODDs, FOADDs
  - Support first-order logical decision tests

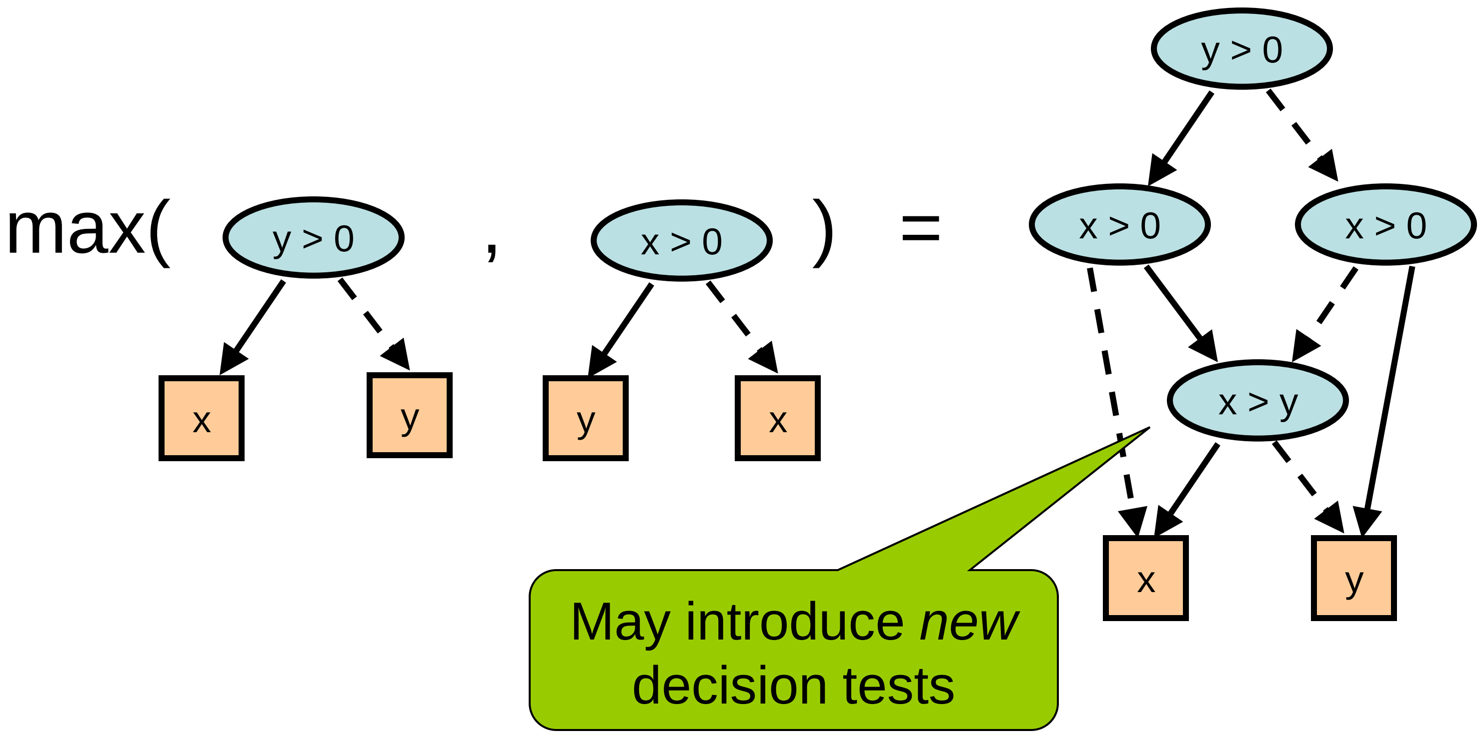
  (Wang, Joshi, Khardon, JAIR-08)

  (Sanner, Boutilier, AIJ-09)

- XADDs: continuous variables →
  (Sanner, UAI-11)



Teaser: XADD Maximization



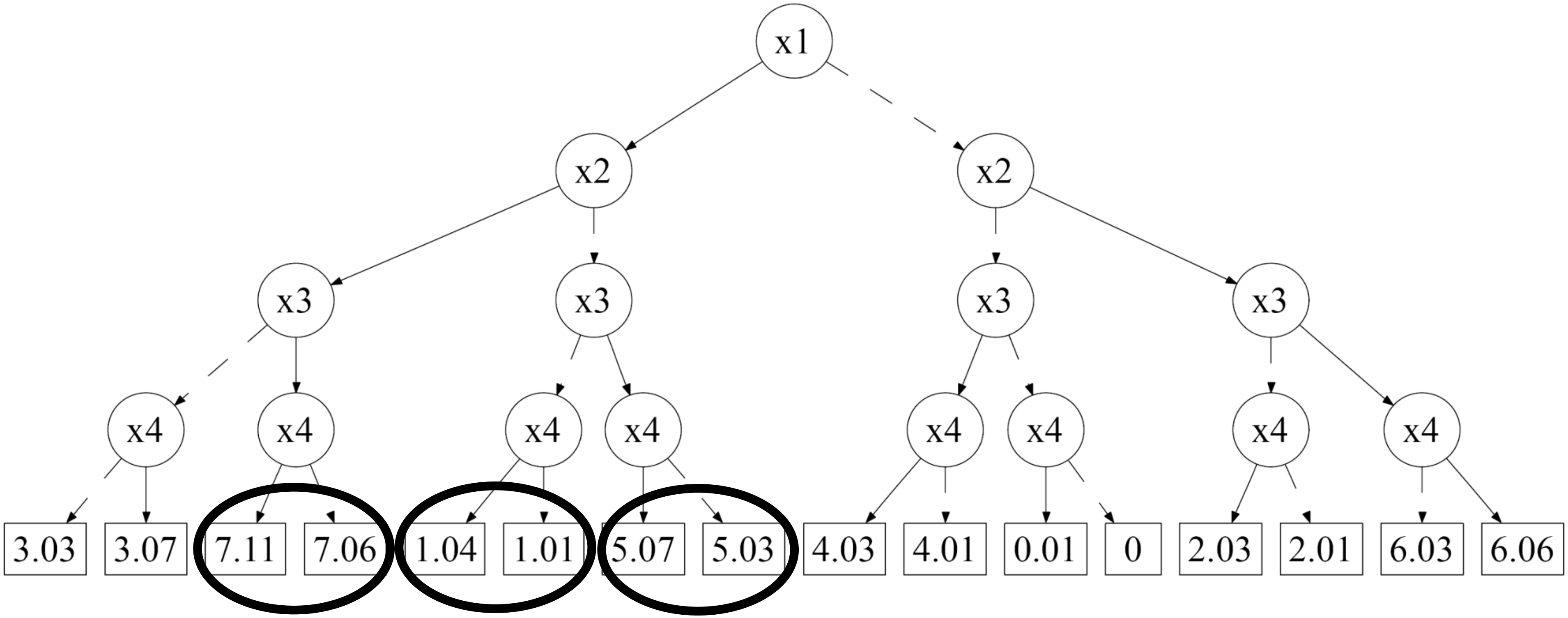
Approximation

Sometimes no DD is compact,

but bounded approximation is…

Problem: Value ADD Too Large

- Sum: $(\sum_{i=1..3} 2^i \cdot x_i) + x_4 \cdot \varepsilon\text{-}Noise$

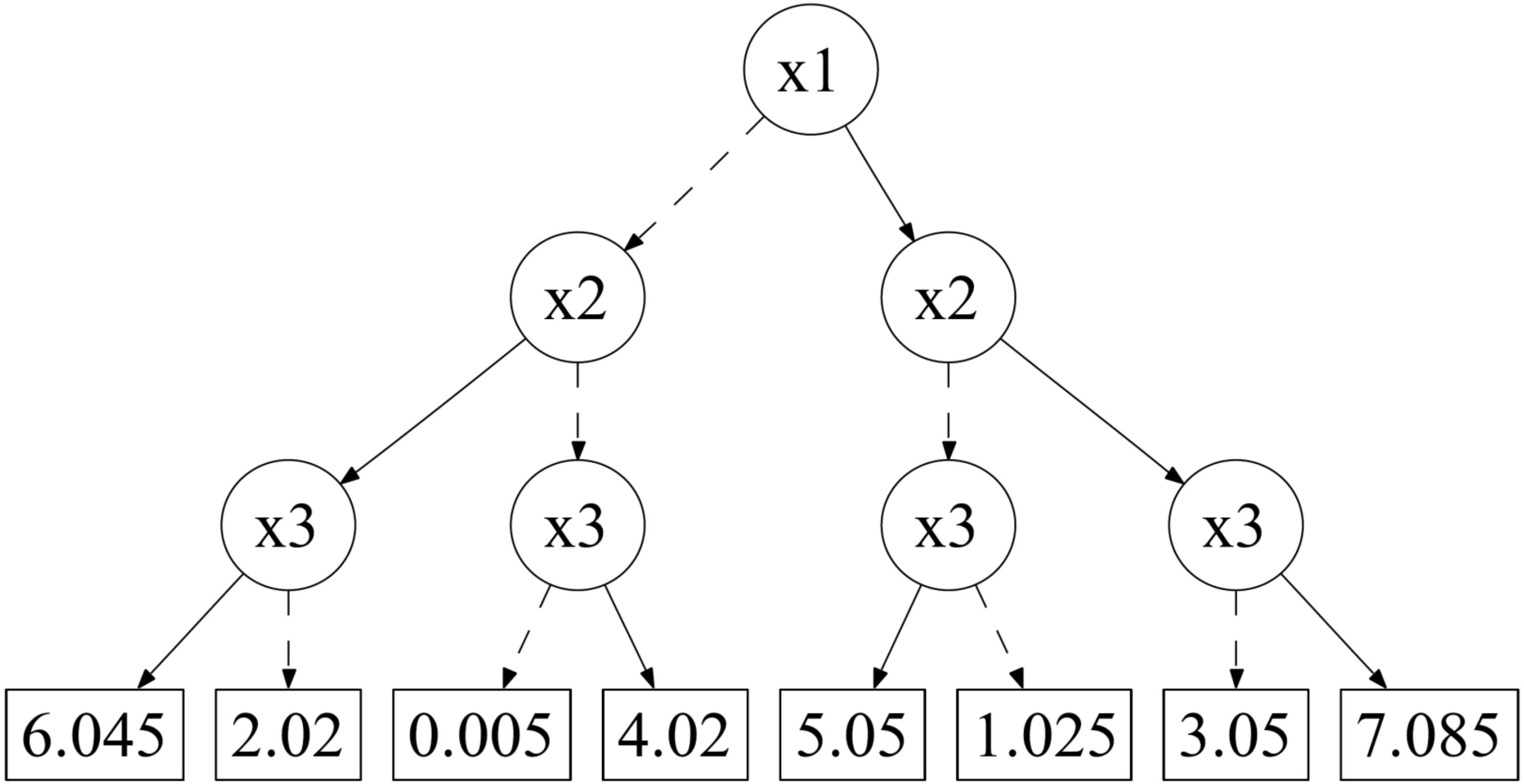


- How to approximate?

(St-Aubin, Hoey, Boutilier, NIPS-00)

Solution: APRICODD Trick

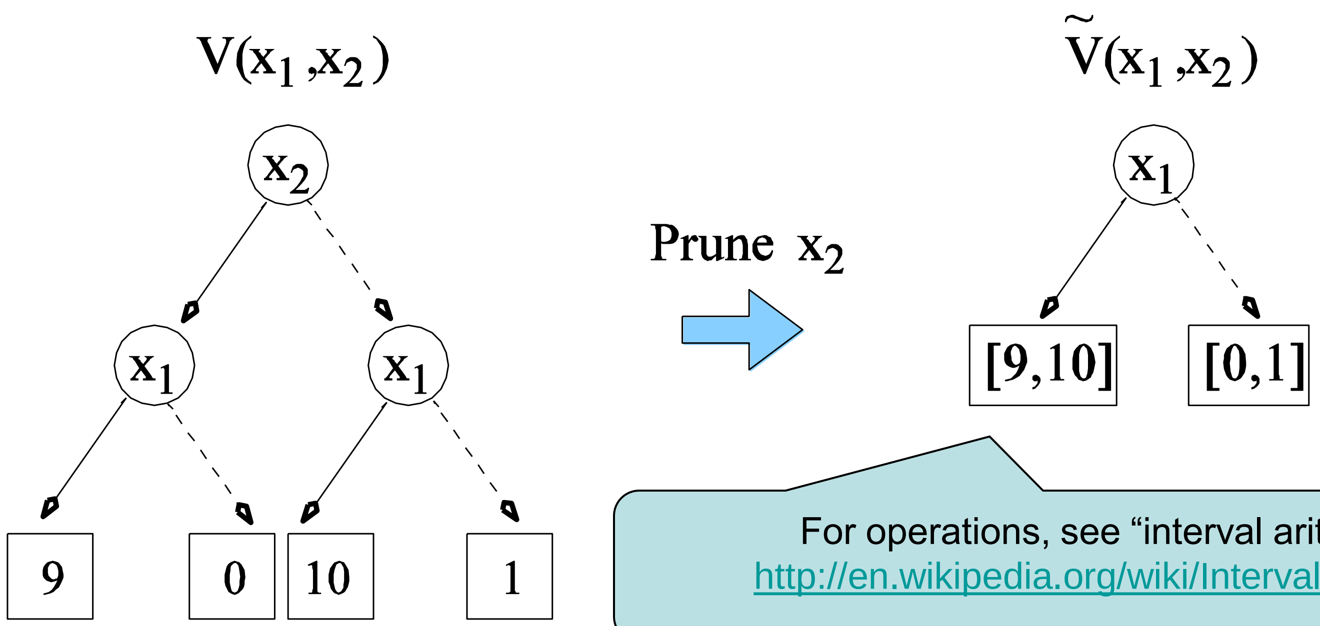
- Merge ≈ leaves and reduce:



- Error is bounded!

Can use ADD to Maintain Bounds!

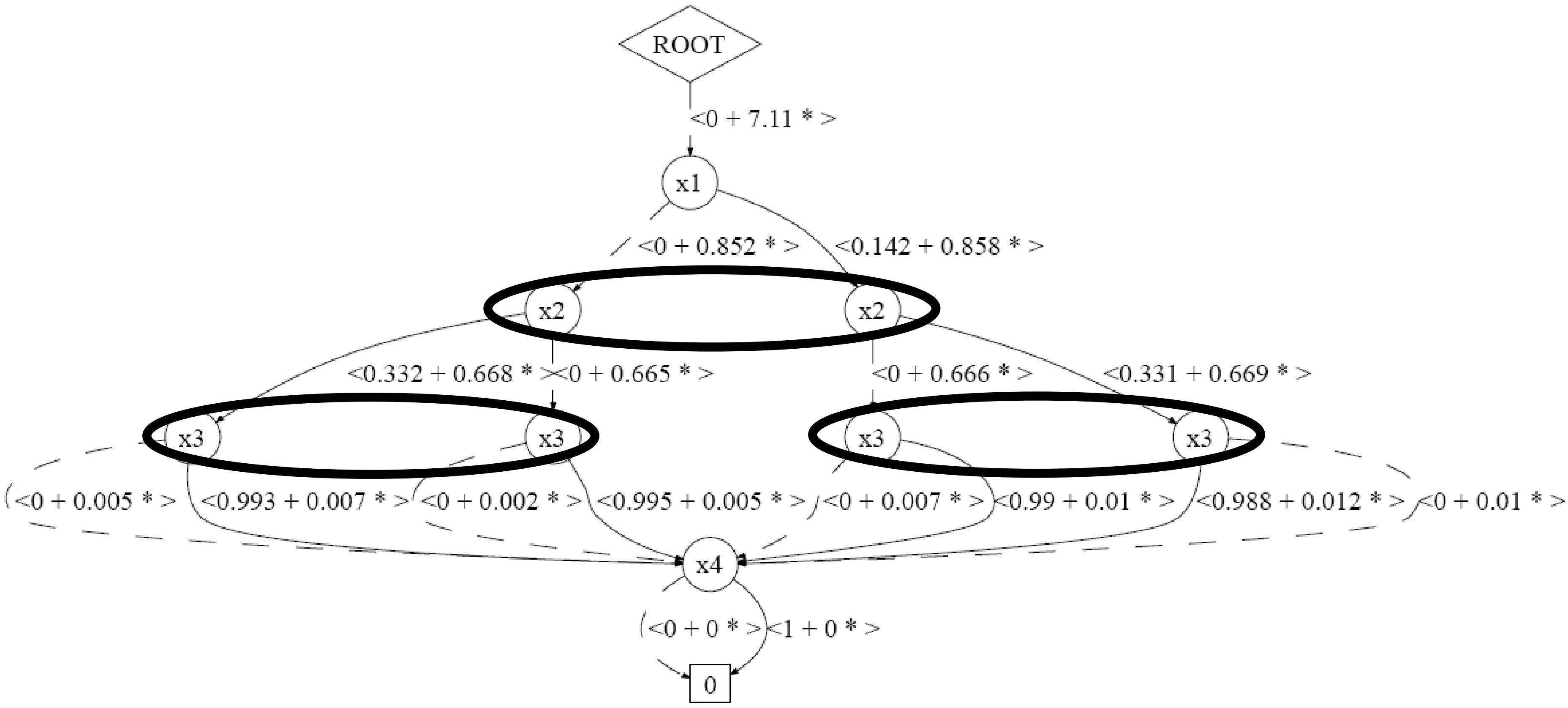
- Change leaf to represent range [L,U]
  - Normal leaf is like [V,V]
  - When merging leaves…
    - keep track of min and max values contributing



For operations, see "interval arithmetic":
http://en.wikipedia.org/wiki/Interval_arithmetic

More Compactness? AADDs?

- Sum: $(\sum_{i=1..3} 2^i \cdot x_i) + x_4 \cdot \varepsilon$-*Noise*

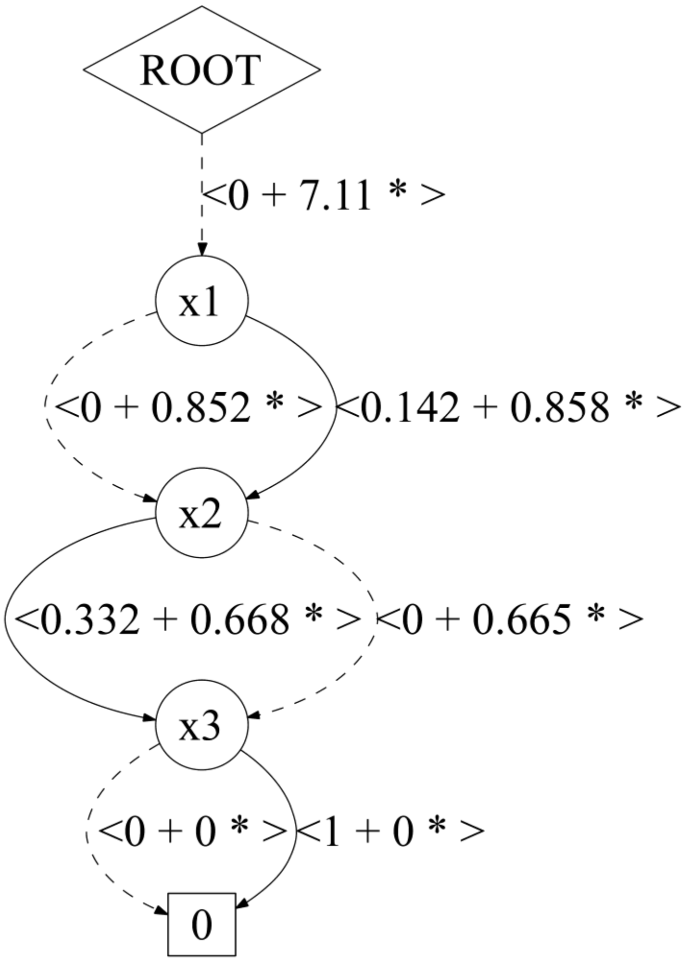


- How to approximate? Error-bounded merge

(Sanner, Uther, Delgado, AAMAS-10)

Solution: MADCAP Trick

- Merge ≈ nodes from bottom up:



Decision Diagram Software

Work with decision diagrams
in < 1 hour!

Software Packages

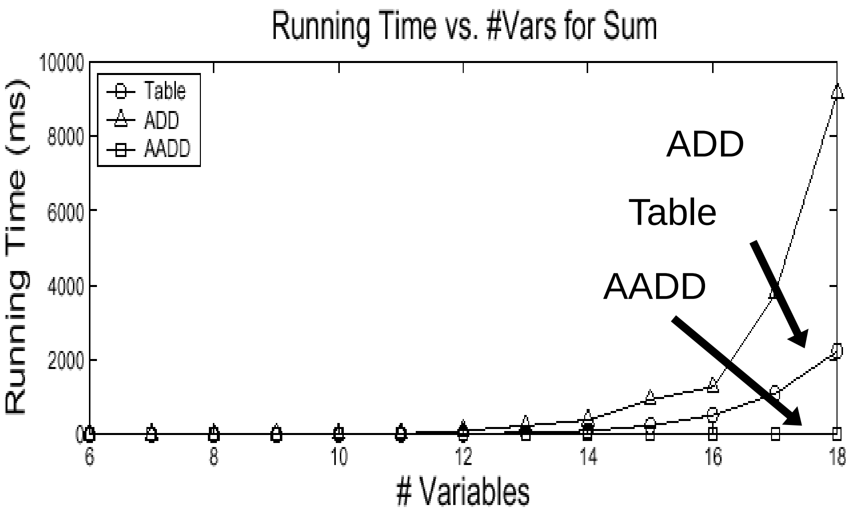
- CUDD
  - BDD / ADD / ZDD
  - http://vlsi.colorado.edu/~fabio/CUDD/
  - Hands down, the best package available
- JavaBDD (native interface to CUDD / others):
  - http://javabdd.sourceforge.net/
- NuSMV – Model Based Planner (MBP)
  - http://mbp.fbk.eu/
- SPUDD – ADD-based value iteration for MDPs
  - http://www.computing.dundee.ac.uk/staff/jessehoey/spudd/index.html
- Symbolic Perseus – Matlab / Java ADD version of value PBVI for POMDPs
  - http://www.cs.uwaterloo.ca/~ppoupart/software.html
- Java BDDs / ADDs / AADDs
  - https://code.google.com/p/dd-inference/
  - Scott's code, not high performance, but functional
  - Includes Java version of SPUDD factored MDP solver & variable elimination

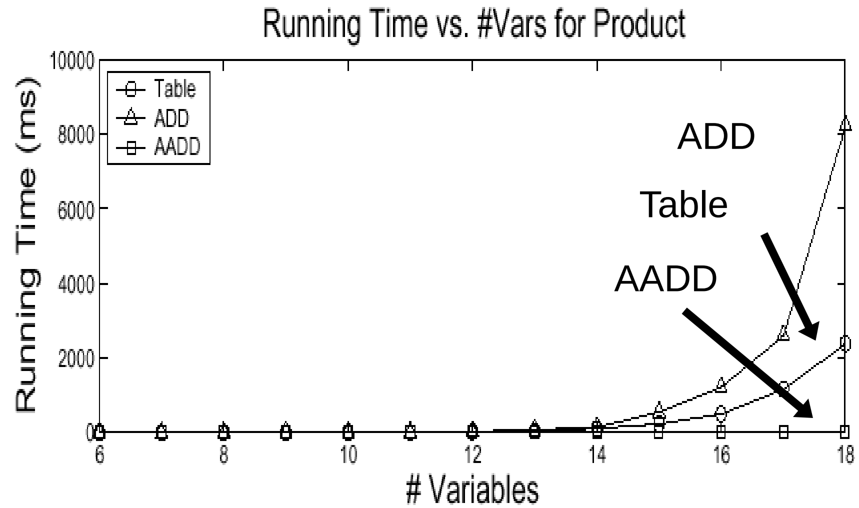
Example Applications using Decision Diagrams

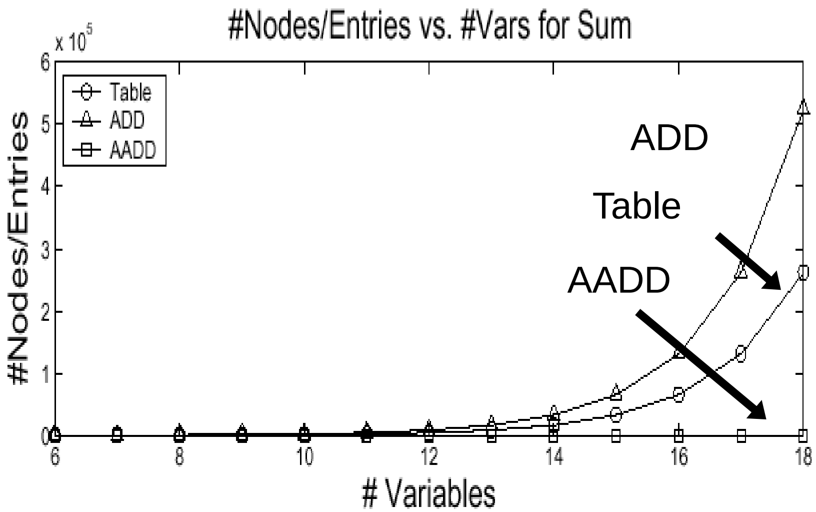
Do they really work well?

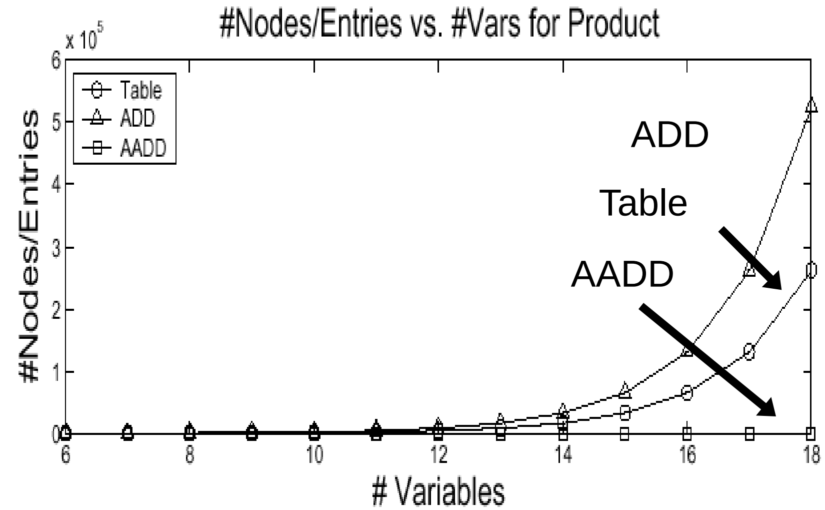
Empirical Comparison: Table/ADD/AADD

- Sum: $\sum_{i=1}^{n} 2^{i} \cdot x_i \oplus \sum_{i=1}^{n} 2^{i} \cdot x_i$
- Prod: $\prod_{i=1}^{n} \gamma\text{^}(2^{i} \cdot x_i) \otimes \prod_{i=1}^{n} \gamma\text{^}(2^{i} \cdot x_i)$









Application: Bayes Net Inference

- Use variable elimination
  - Replace CPTs with ADDs or AADDs
  - Could do same for clique/junction-tree algorithms
- Exploits
  - Context-specific independence
    - Probability has logical structure: $P(a|b,c) = \text{if } b \;?\; 1 : \text{if } c \;?\; .7 : .3$
  - Additive CPTs
    - Probability is discretized linear function: $P(a|b_1 \ldots b_n) = c + b \cdot \sum_i 2^i\, b_i$
  - Multiplicative CPTs
    - Noisy-or (multiplicative AADD): $P(e|c_1 \ldots c_n) = 1 - \prod_i (1 - p_i)$
- If factor has above compact form, AADD exploits it

Bayes Net Results: Various Networks

Bayes Net	Table		ADD		AADD	
	# Entries	Time	# Nodes	Time	# Nodes	Time
Alarm	1,192	2.97s	689	2.42s	**405**	**1.26s**
Barley	470,294	EML*	139,856	EML*	**60,809**	**207m**
Carpo	636	0.58s	955	0.57s	**360**	**0.49s**
Hailfinder	9,045	26.4s	4,511	9.6s	**2,538**	**2.7s**
Insurance	2,104	278s	1,596	116s	**775**	**37s**
Noisy-Or-15	65,566	27.5s	125,356	50.2s	**1,066**	**0.7s**
Noisy-Max-15	131,102	33.4s	202,148	42.5s	**40,994**	**5.8s**

*EML: Exceeded Memory Limit (1GB)

Application: MDP Solving

- SPUDD Factored MDP Solver (Hoey et al, 99)
  - Originally uses ADDs, can use AADDs
  - Implements factored value iteration…

DD

DD

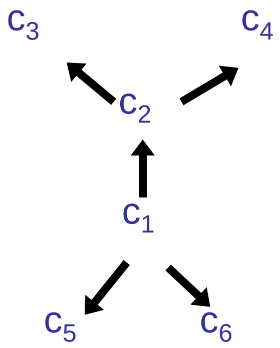
DD

DD

$$V^{n+1}(x_1 \ldots x_i) = R(x_1 \ldots x_i) + \gamma \cdot \max_a \sum_{x1' \ldots xi'} \prod_{F1 \ldots Fi} P(x_1' | _{\ldots} x_i) \ldots P(x_i' | _{\ldots} x_i) \, V^n(x_1' \ldots x_i')$$

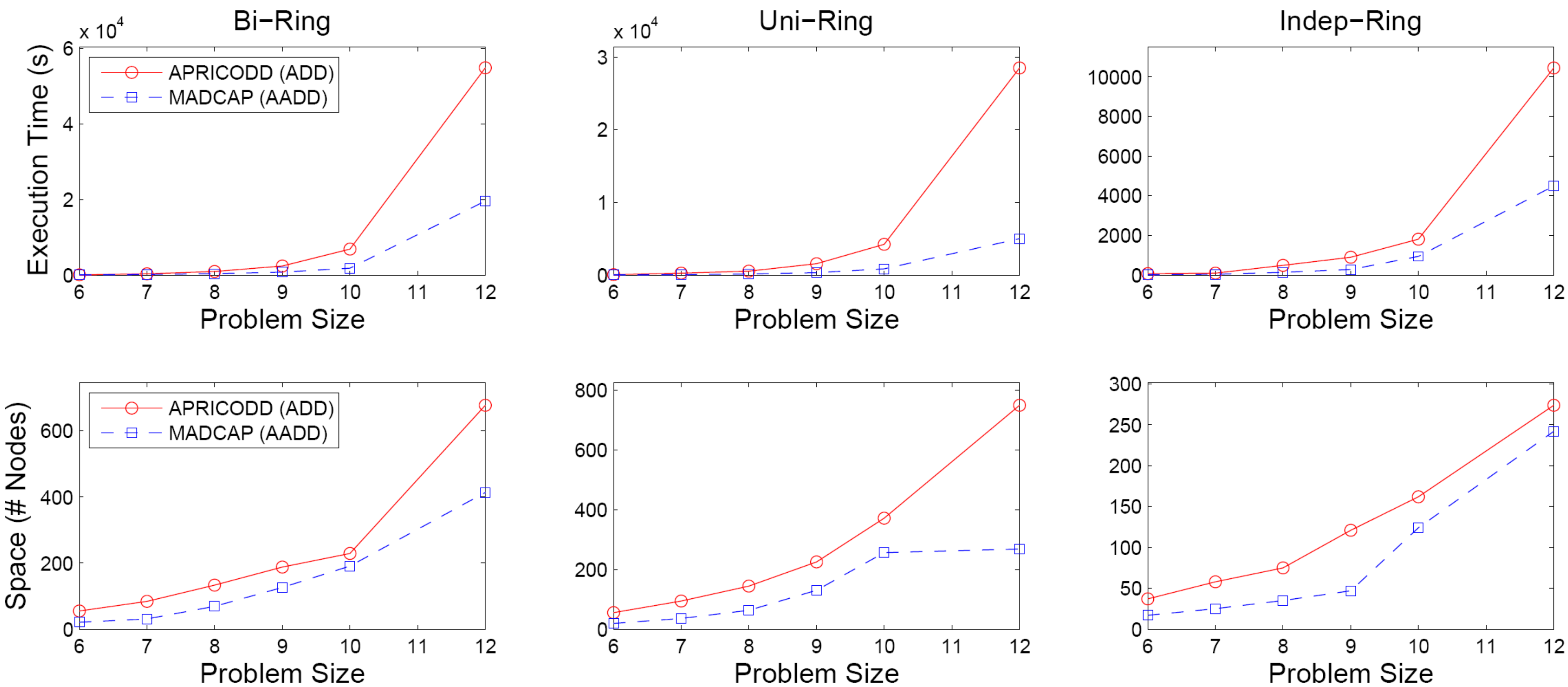
DD

Application: SysAdmin

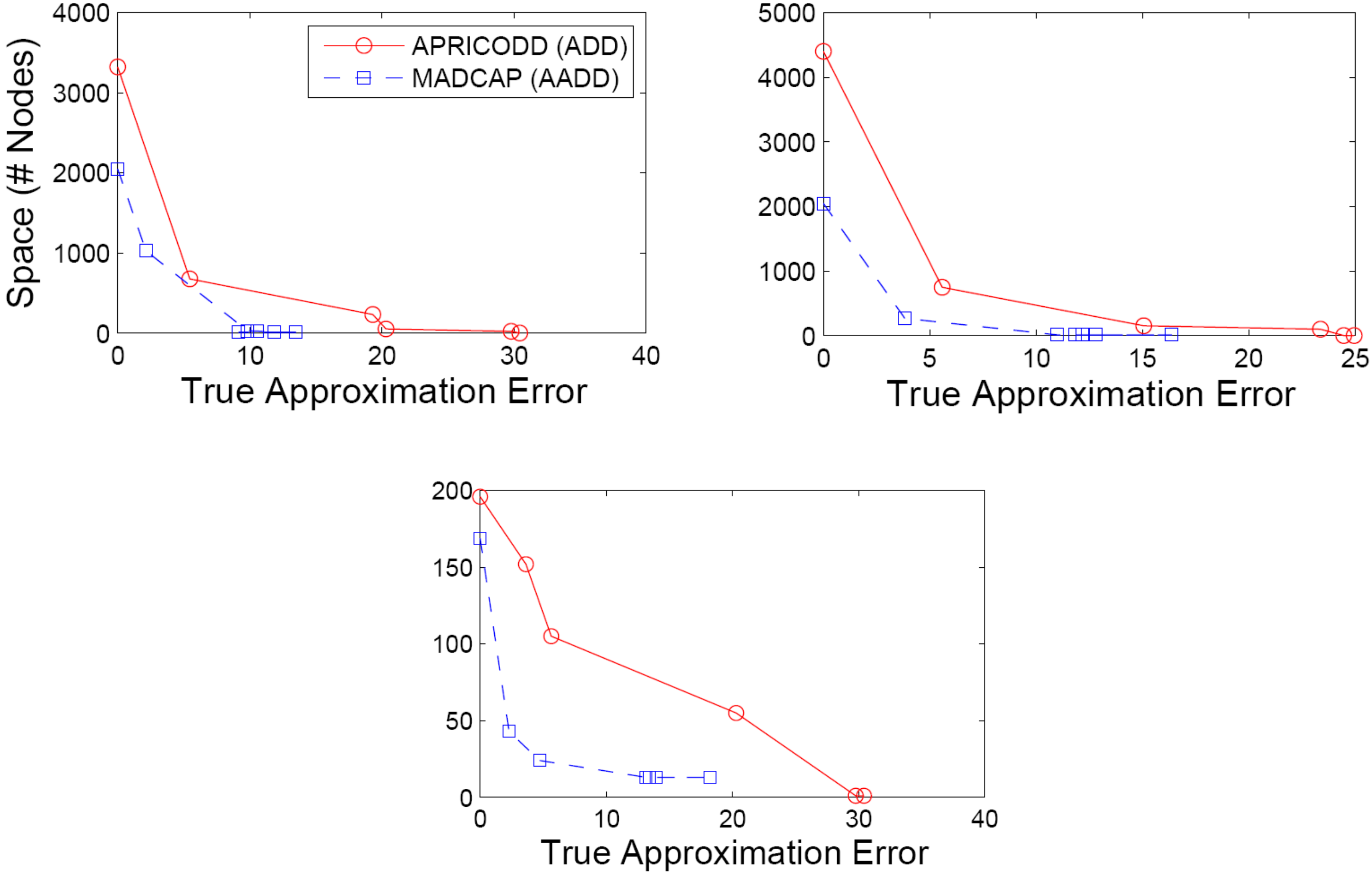


- SysAdmin MDP (GKP, 2001)
  - Network of computers: $c_1, \ldots, c_k$
  - Various network topologies
  - Every computer is running or crashed
  - At each time step, status of c_i affected by
    - Previous state status
    - Status of incoming connections in previous state
  - Reward: +1 for every computer running (additive)

Results I: SysAdmin (10% Approx)

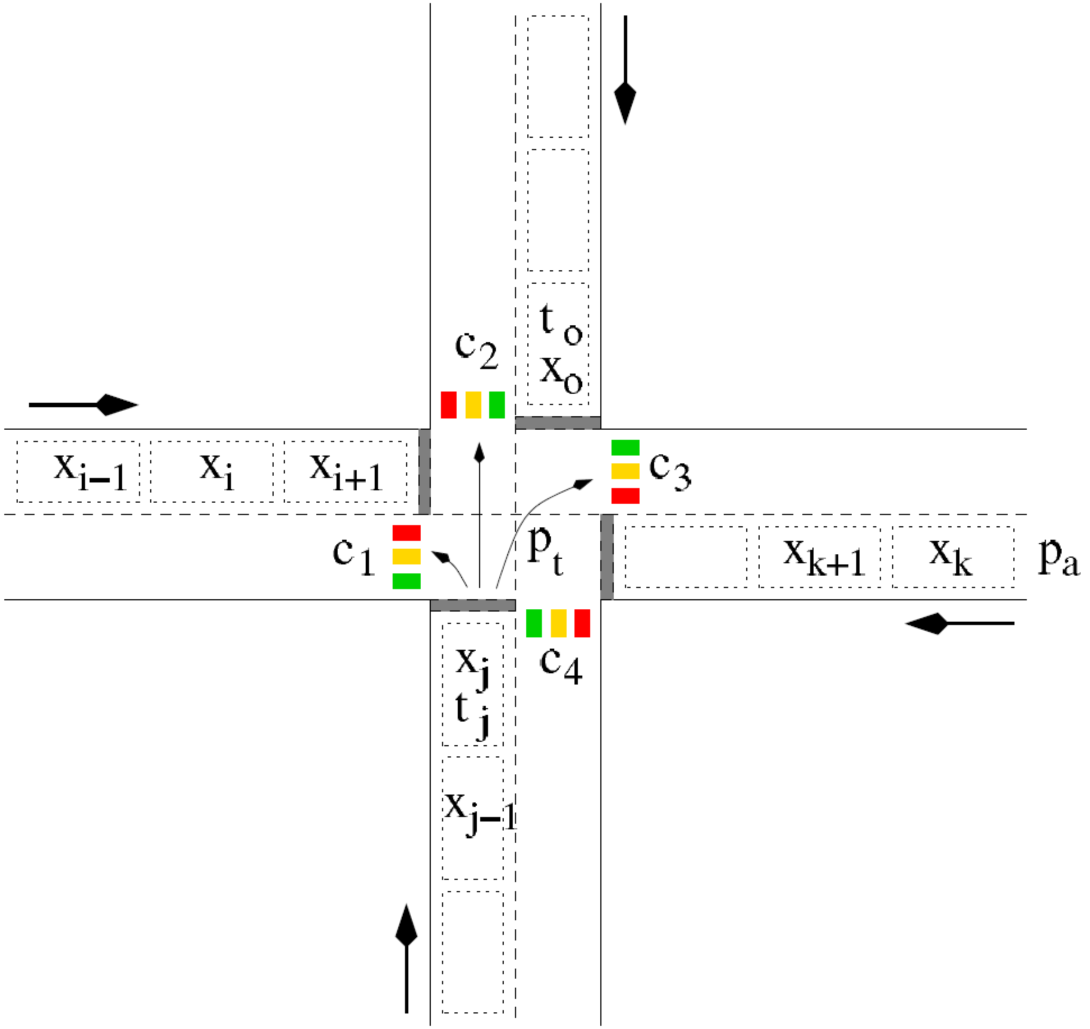


Results II: SysAdmin

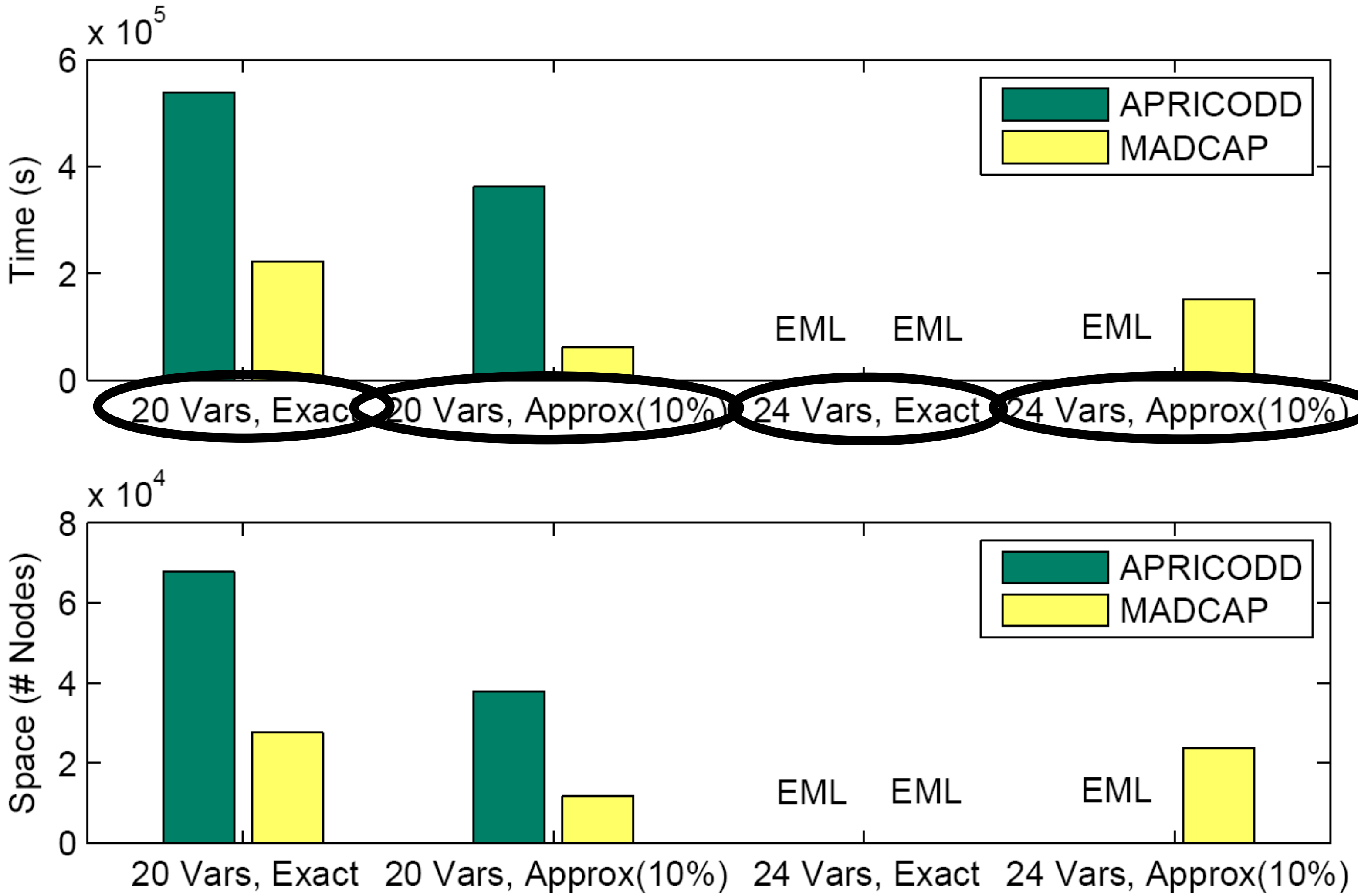


Traffic Domain

- Binary **cell transmission model (CTM)**
- Actions
  - Light changes
- Objective:
  - Maximize empty cells in network



Results Traffic



Application: POMDPs

- Provided an AADD implementation for Guy Shani's factored POMDP solver

- Final value function size results:

	ADD	**AADD**
Network Management	7000	92
Rock Sample	189	34

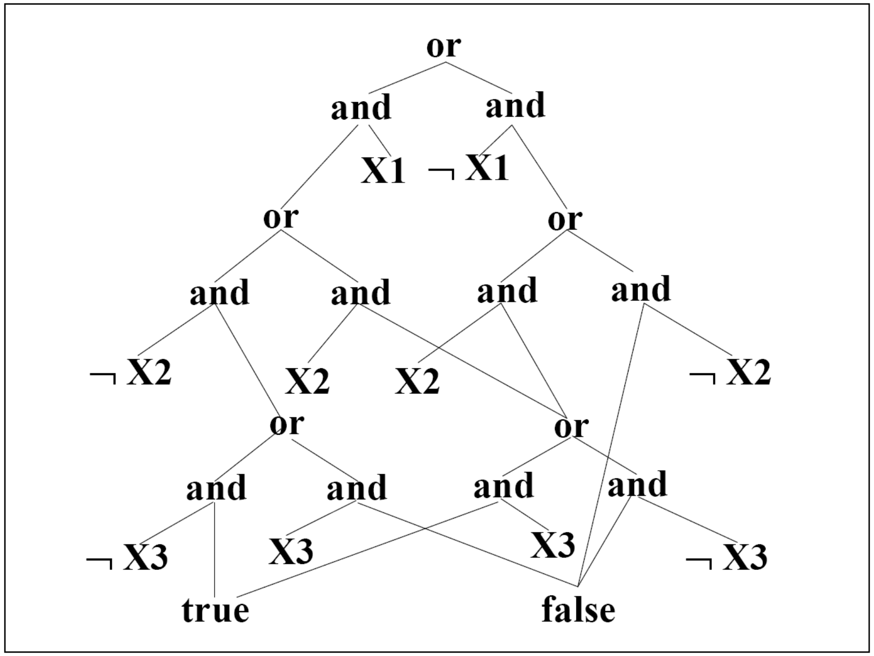
Inference with Decision Diagrams vs. Compilations (d-DNNF, etc.)

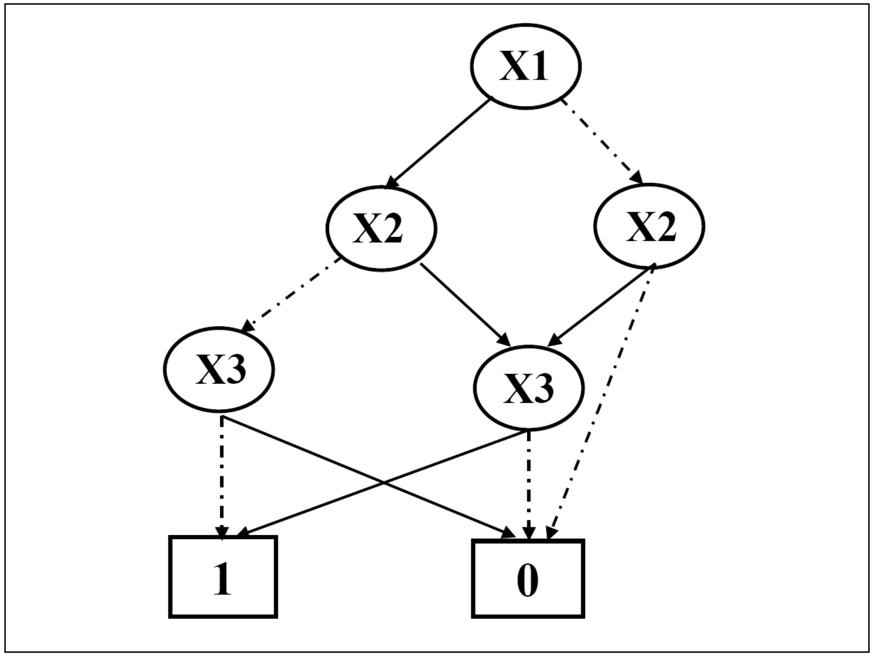
Important Distinctions

Defintions / Diagrams from
"A Knowledge Compilation Map",
Darwiche and Marquis. JAIR 02

BDDs in NNF

- Can express BDD as NNF formula
- Can represent NNF diagrammatically

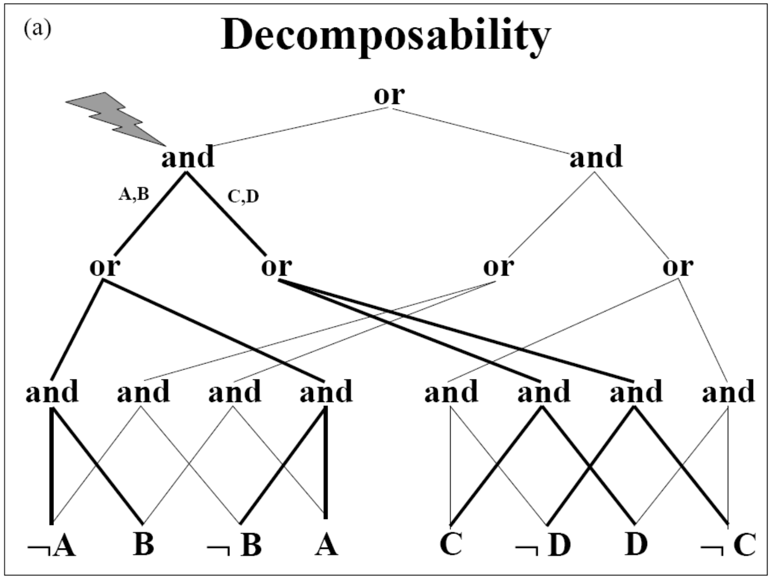




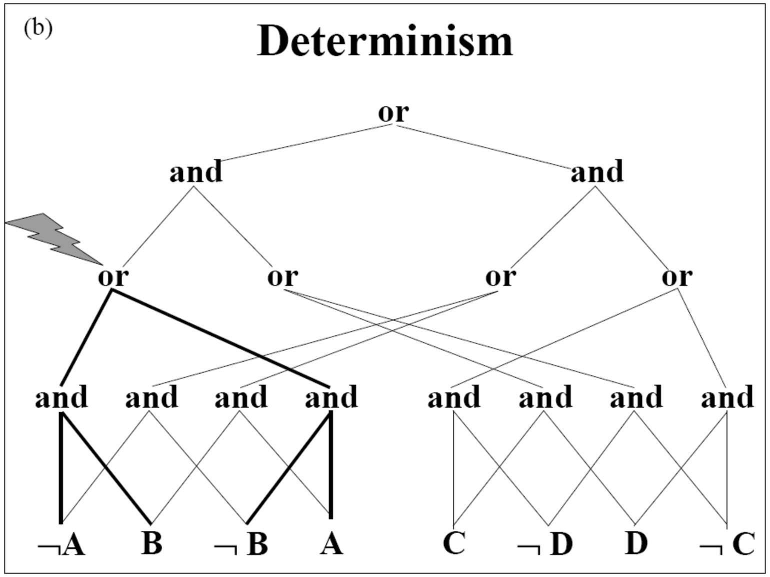
d-DNNF

Defintions / Diagrams from "A Knowledge Compilation Map", Darwiche and Marquis. JAIR 02

- Decomposable NNF: sets of leaf vars of conjuncts are disjoint



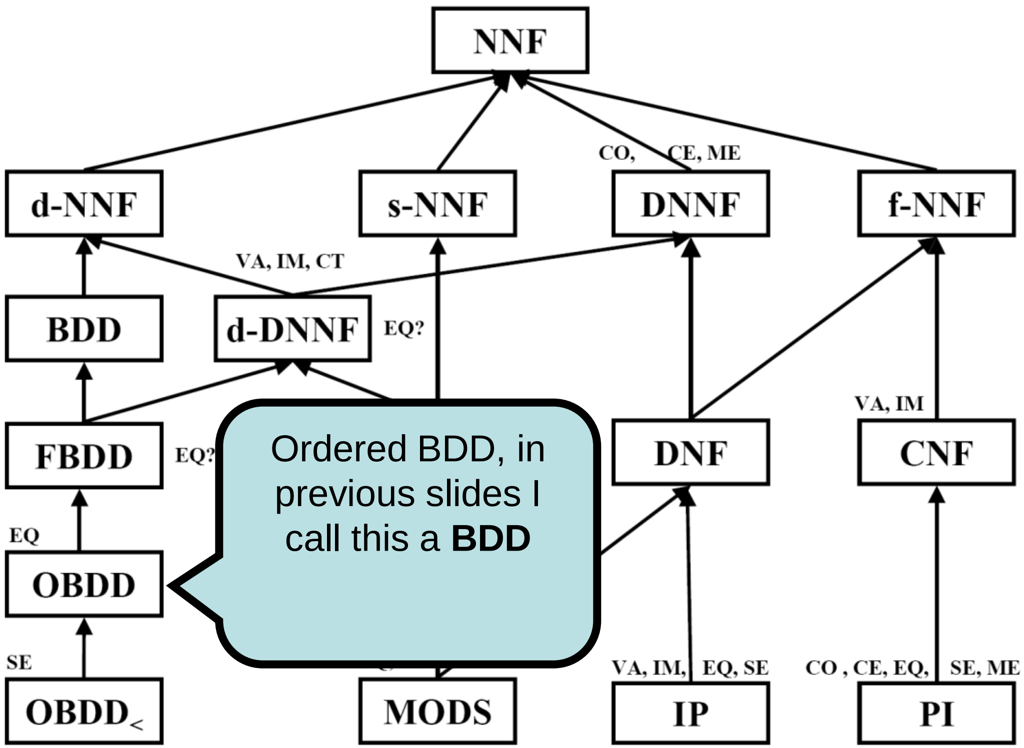
- Deterministic NNF: formula for disjuncts have disjoint models (conjunction is unsatisfiable)



Defintions / Diagrams from "A Knowledge Compilation Map", Darwiche and Marquis. JAIR 02

d-DNNF

- D-DNNF used to **compile single formula**
  - **d-DNNF does not support efficient binary operations ($\vee,\wedge,\neg$)**
  - d-DNNF shares some polytime operations with OBDD / ADD
    - (weighted) model counting (CT) – used in many inference tasks
    - $\rightarrow$ Size(d-DNNF) $\leq$ Size(OBDD) so more efficient on d-DNNF



child is subset of $\rightarrow$ parent

Children inherit polytime operations of parents

Size of children $\geq$ parents

Notation	Query
CO	polytime consistency check
VA	polytime validity check
CE	polytime clausal entailment check
IM	polytime implicant check
EQ	polytime equivalence check
SE	polytime sentential entailment check
CT	polytime model counting
ME	polytime model enumeration

Table 4: Notations for queries.

Compilations vs Decision Diagrams

Typically not a good idea in sequential probabilistic inference or decision-making

- Summary
  - **If** you can compile problem into **single formula** then compilation is likely preferable to DDs
    - provided you only need ops that compilation supports
  - Not *all* compilations efficient for *all binary* operations
    - e.g., all ops needed for progression / regression approaches
    - fixed ordering of DDs help support these operations
- Note: other compilations (e.g., arithmetic circuits)
  - Great software: http://reasoning.cs.ucla.edu/

And that’s a crash course in DDs!

Take-home point:

- If your problem is factored
- and you’re currently using a tabular representation
- and you need binary operations on these tables

$\rightarrow$ consider using a DD instead.