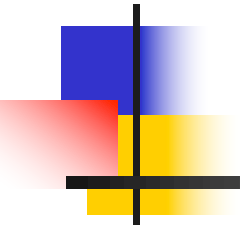
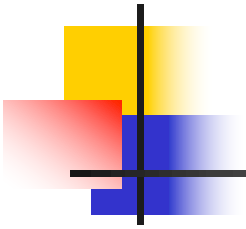


Knowledge Compilers

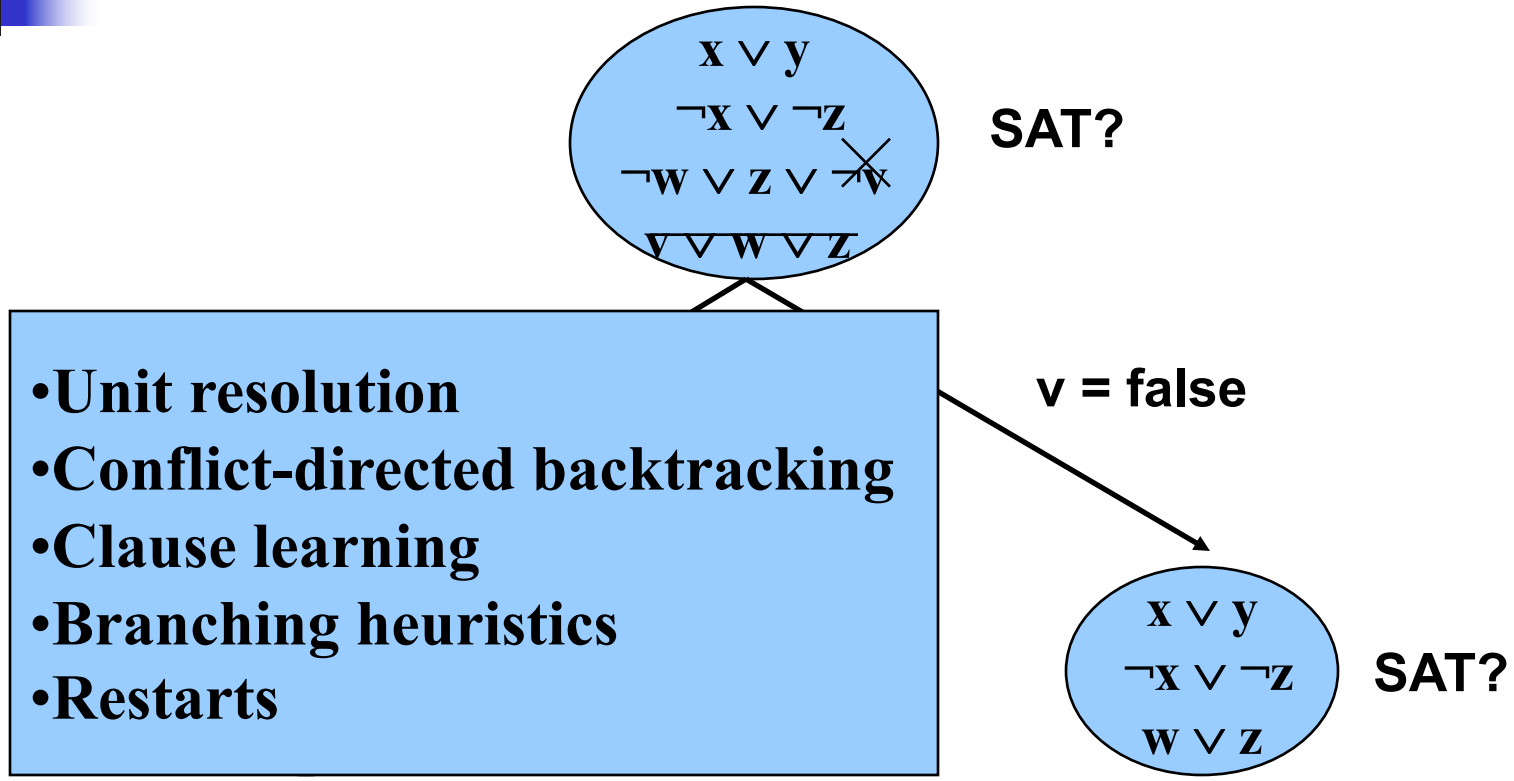
Adnan Darwiche
Computer Science Department, UCLA

Building Compilers



- To-down approaches:
  - Based on exhaustive search
- Bottom-up approaches:
  - Based on transformations

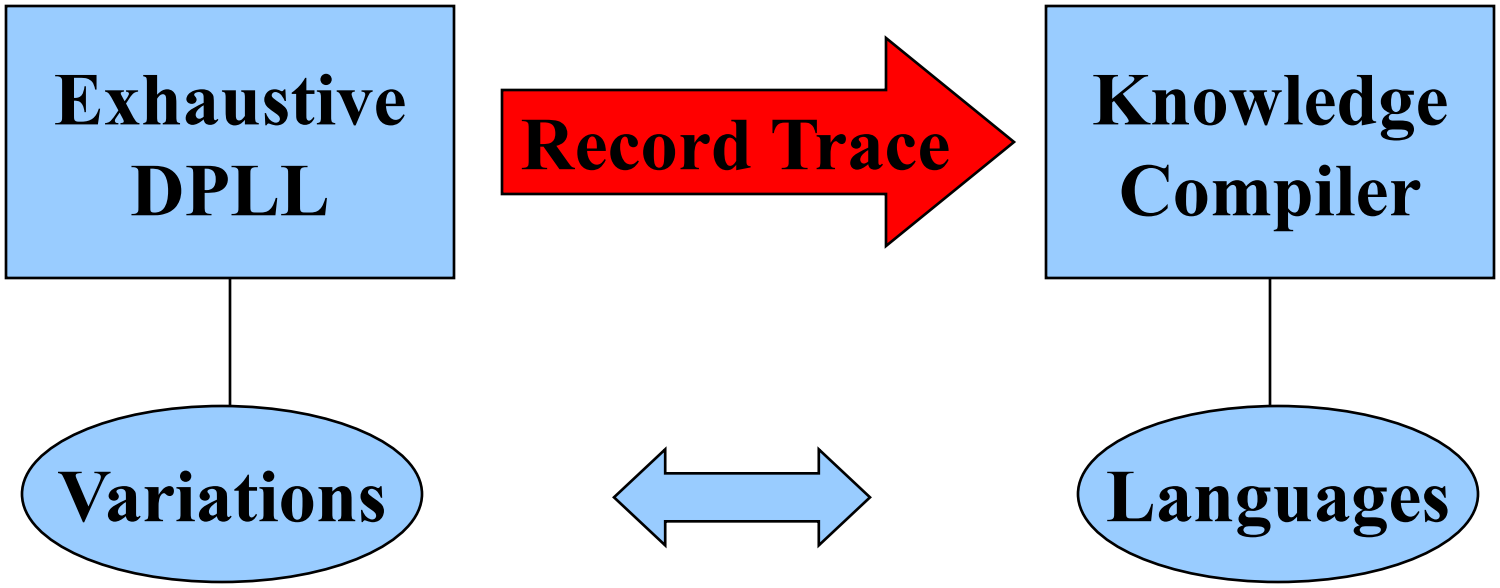
SAT by DPLL Search



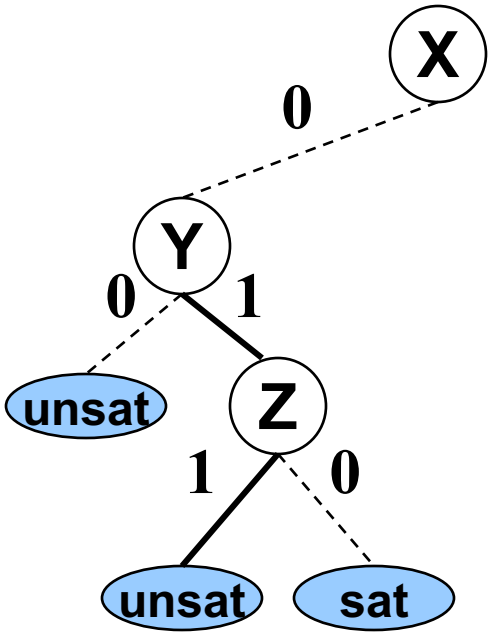
Terminating condition for recursion:

empty set (satisfied), or empty clause (contradiction)

The Language of Search



Trace of DPLL



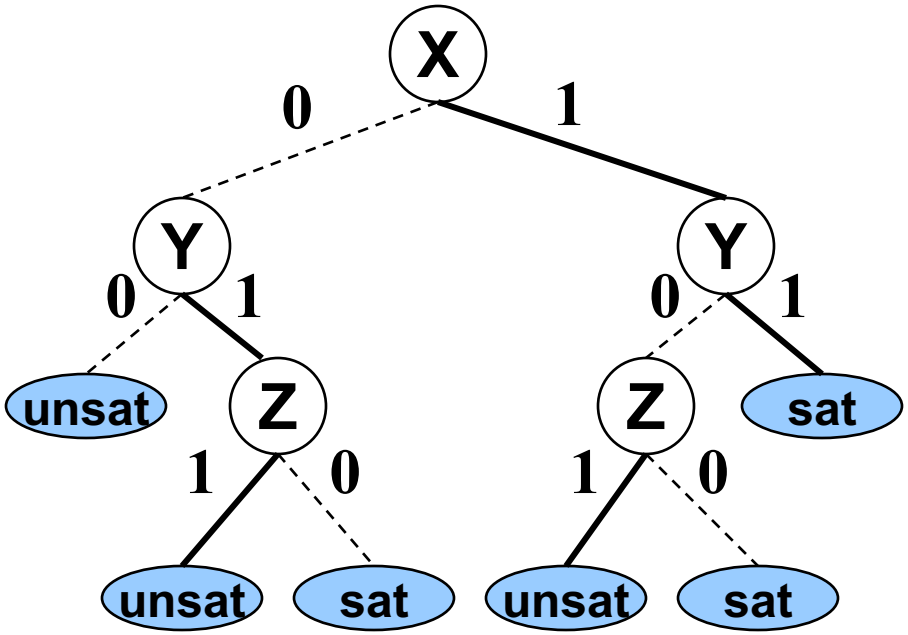
$$X \vee Y$$

$$X \vee \neg Y \vee \neg Z$$

$$\neg X \vee Y \vee \neg Z$$

Exhaustive DPLL

Run to Exhaustion

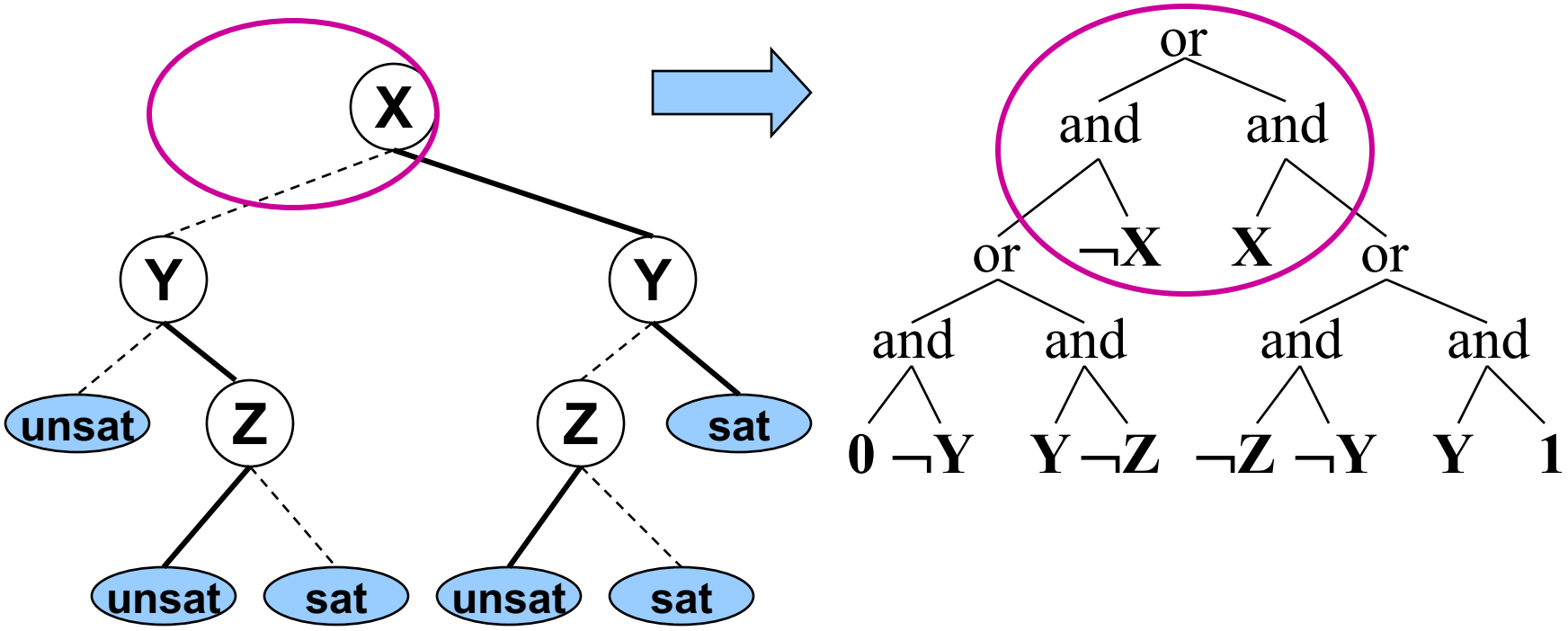


$$X \vee Y$$

$$X \vee \neg Y \vee \neg Z$$

$$\neg X \vee Y \vee \neg Z$$

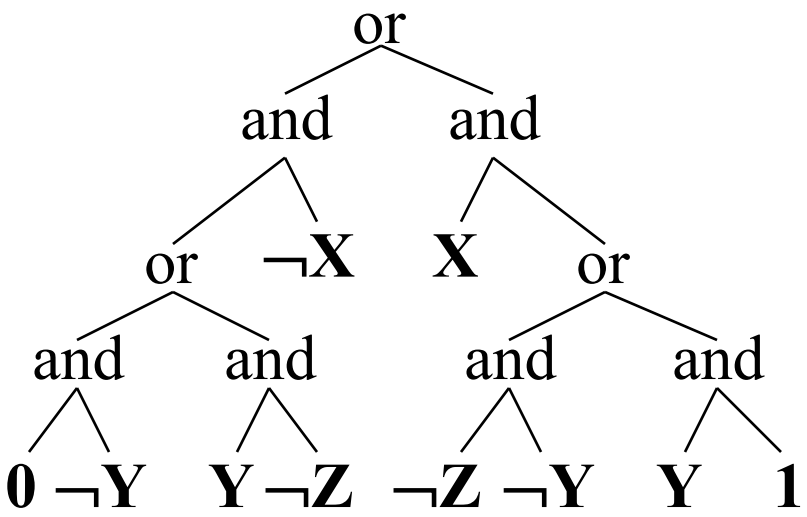
Trace of DPLL: a Formula



Trace of DPLL: a Formula

Equivalent to original CNF

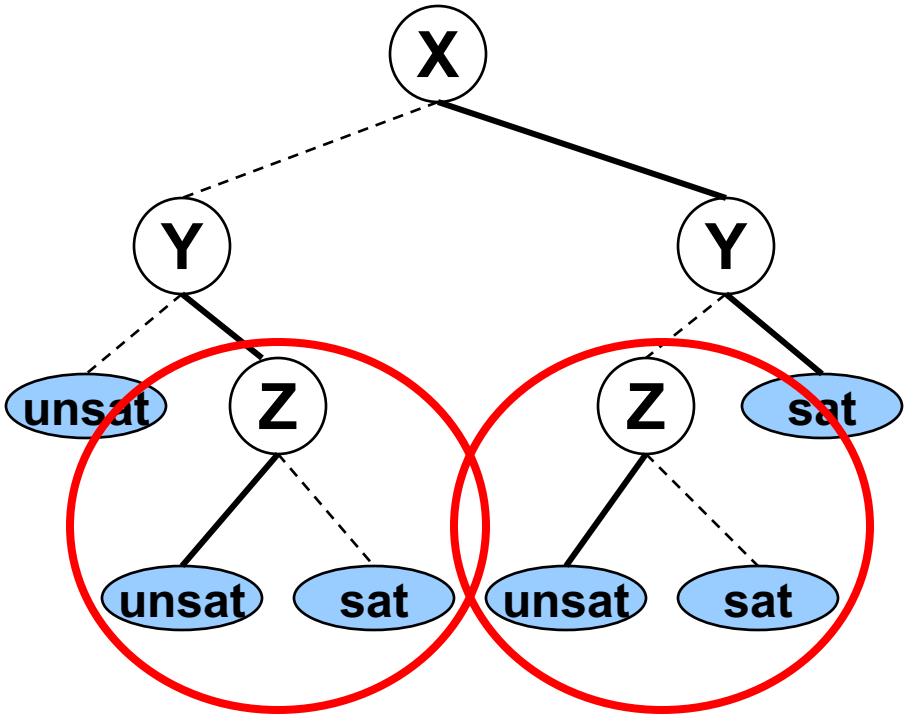
Tractable
(e.g., count models)



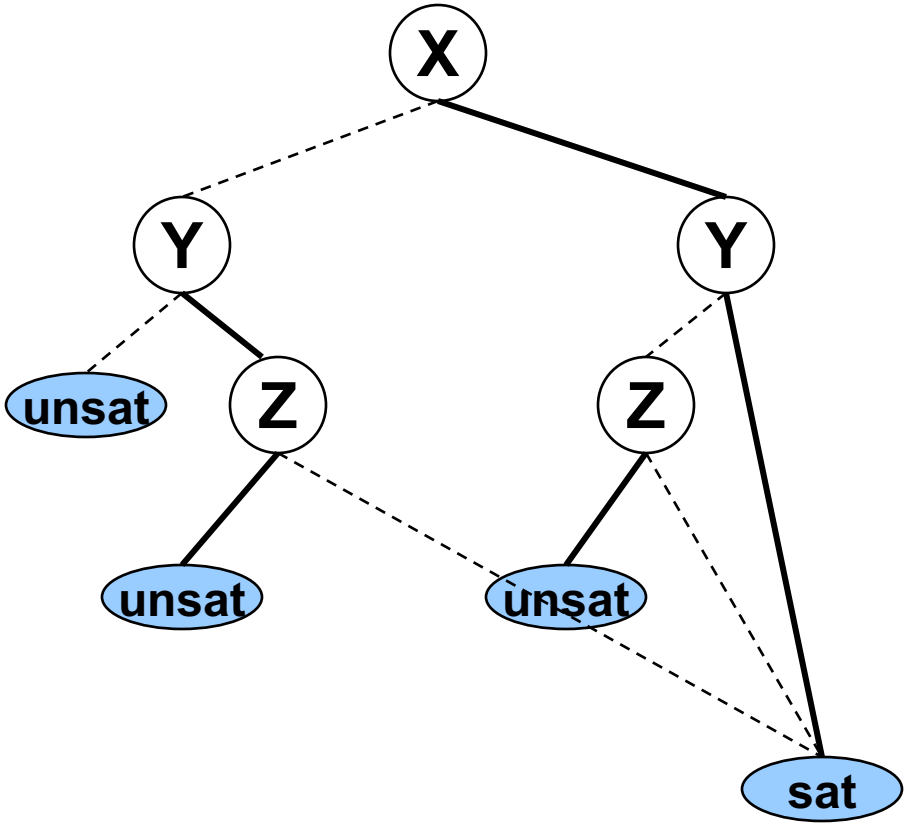
Dealing with Redundancy

Level One: Do not record redundant portions of trace

Level Two: Try not to solve equivalent subproblems

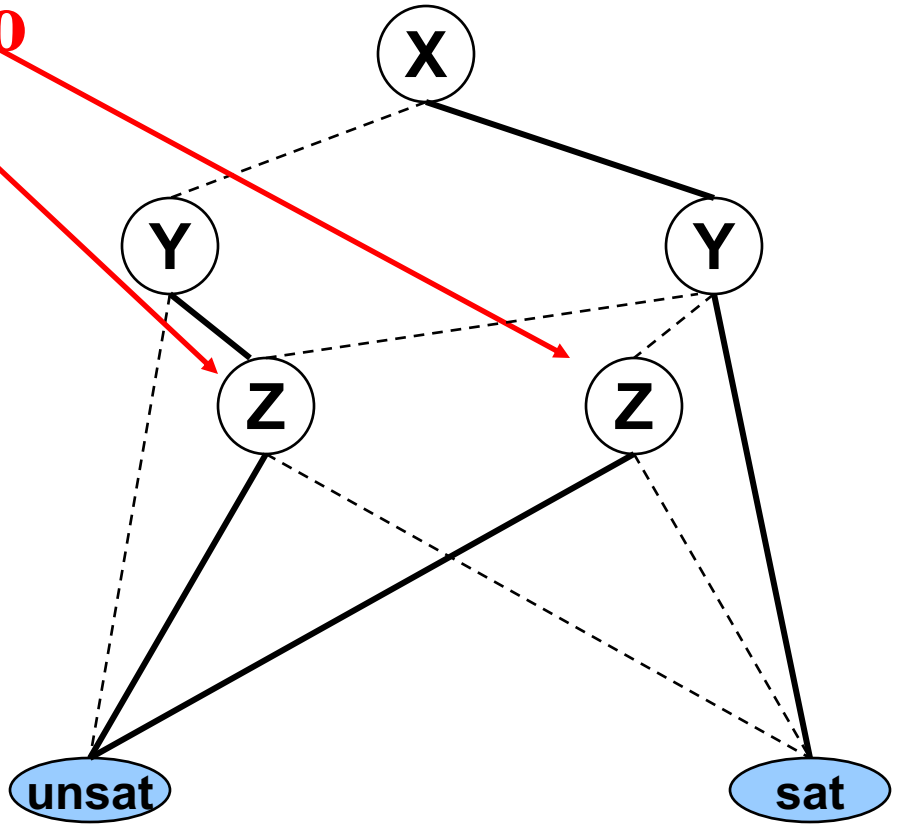


Dealing with Redundancy

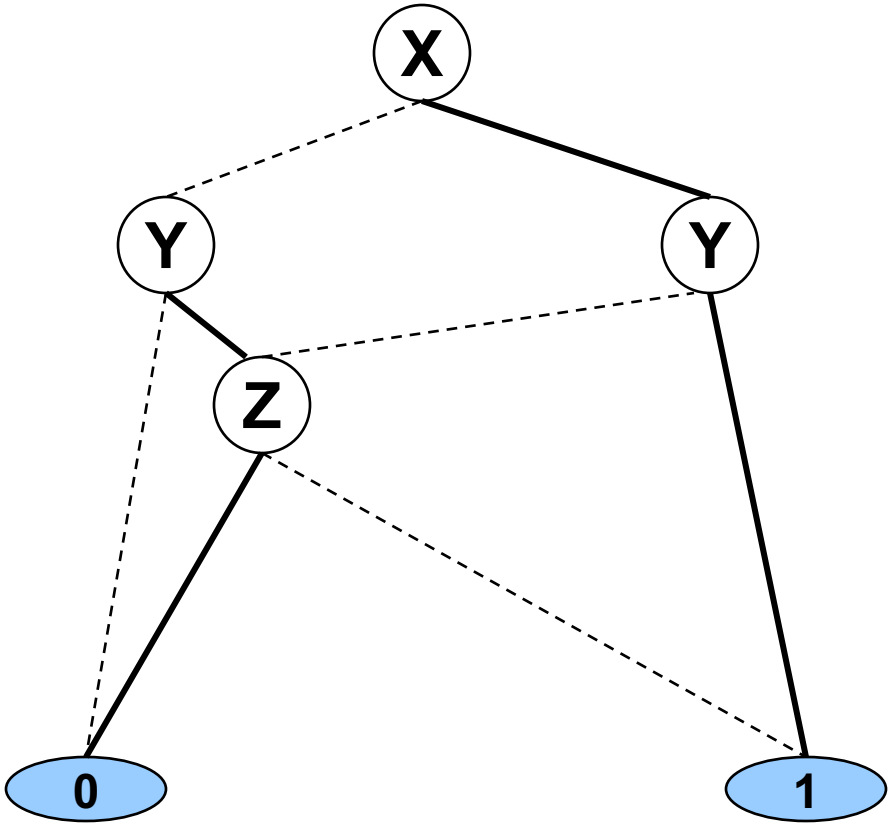


Dealing with Redundancy

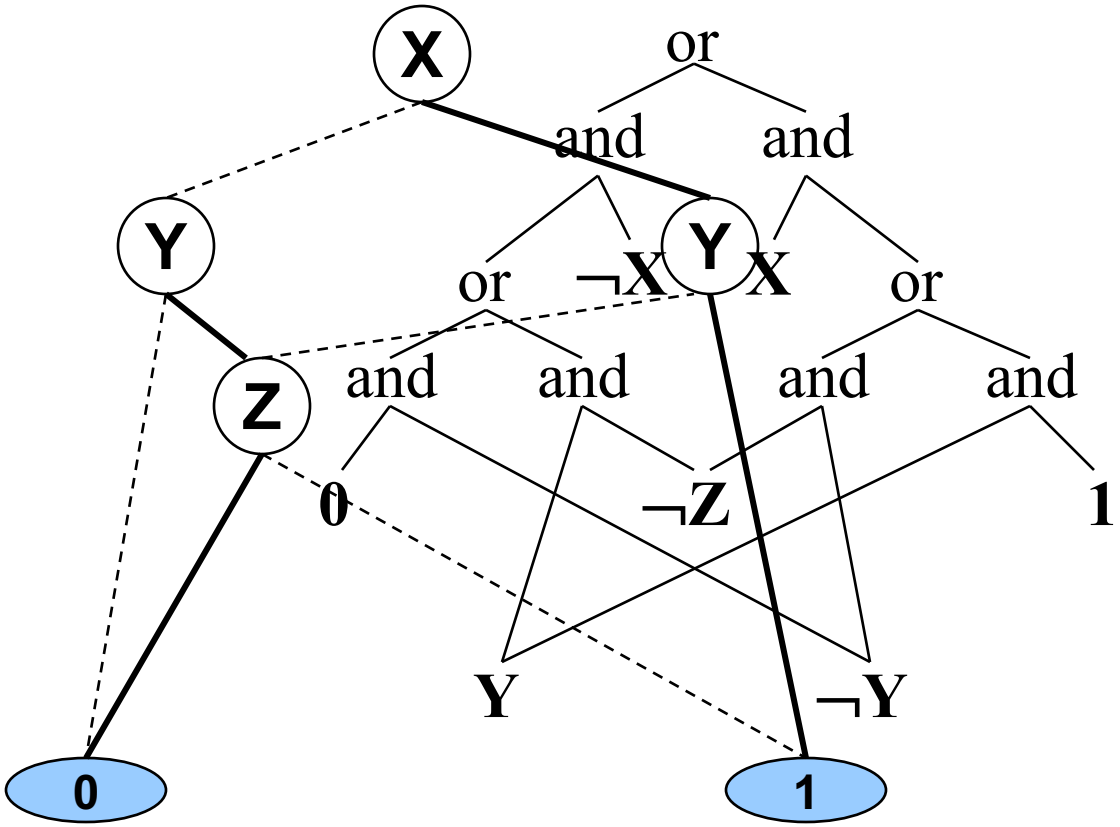
Simply create to existing node



This is an OBDD!



This is an OBDD!



NNF + decision, decomposability, ordering

A Non-traditional OBDD Compiler

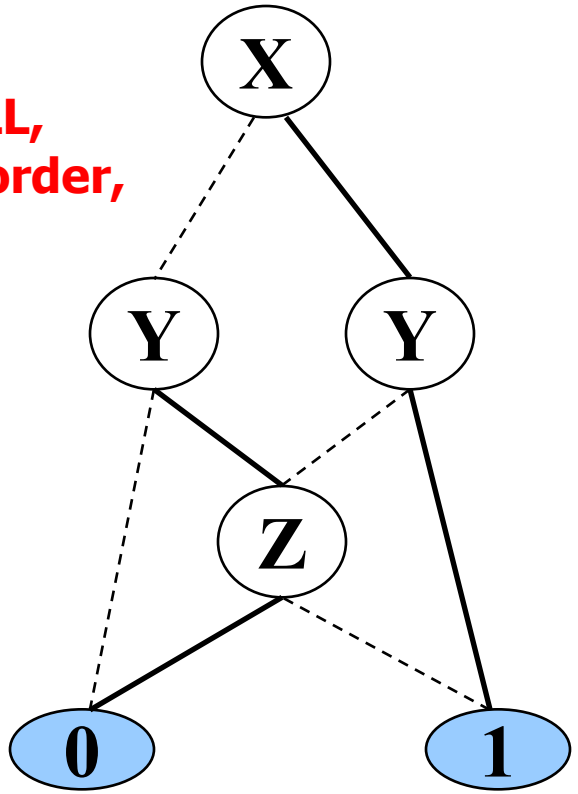
$X \vee Y$

$X \vee \neg Y \vee \neg Z$

$\neg X \vee Y \vee \neg Z$

Exhaustive DPLL,
Fixed variable order,
Unique nodes

Compile



New complexity guarantees

FBDD

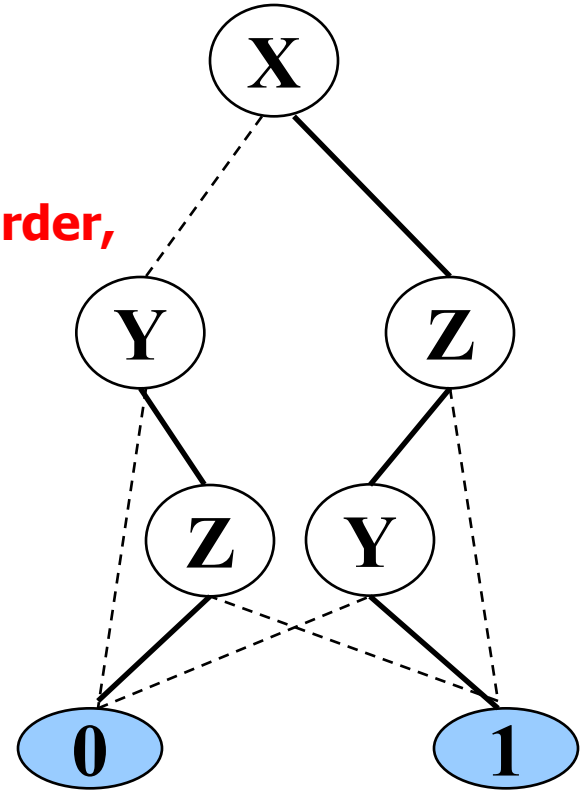
$X \vee Y$

$X \vee \neg Y \vee \neg Z$

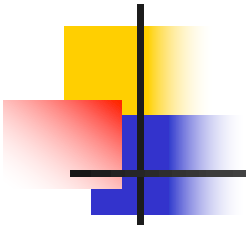
$\neg X \vee Y \vee \neg Z$

Exhaustive DPLL,
Dynamic variable order,
Unique nodes

Compile



NNF + decision, decomposability



Dealing with Redundancy

- **Level One: Unique nodes (done)**
- **Level Two: Avoid redundant compilation (searches)**

Redundant Compilation

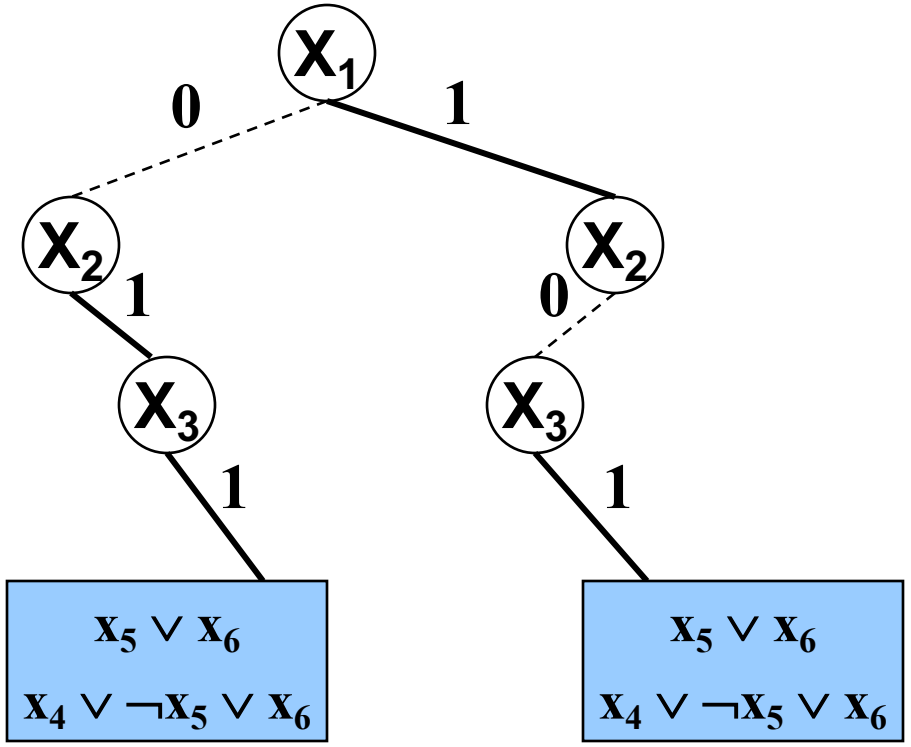
$x_5 \vee x_6$

$x_4 \vee \neg x_5 \vee x_6$

$x_1 \vee x_3 \vee x_4 \vee x_5$

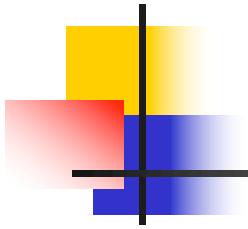
$x_2 \vee x_3$

$x_1 \vee x_2 \vee \neg x_3$



Formula Caching: complexity guarantees

Beyond BDDs…

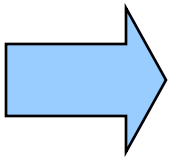


Plain DPLL → **FBDD**

Fixed Variable Ordering → **OBDD**

Decomposition (Component Analysis)

Solve disjoint subproblems independently



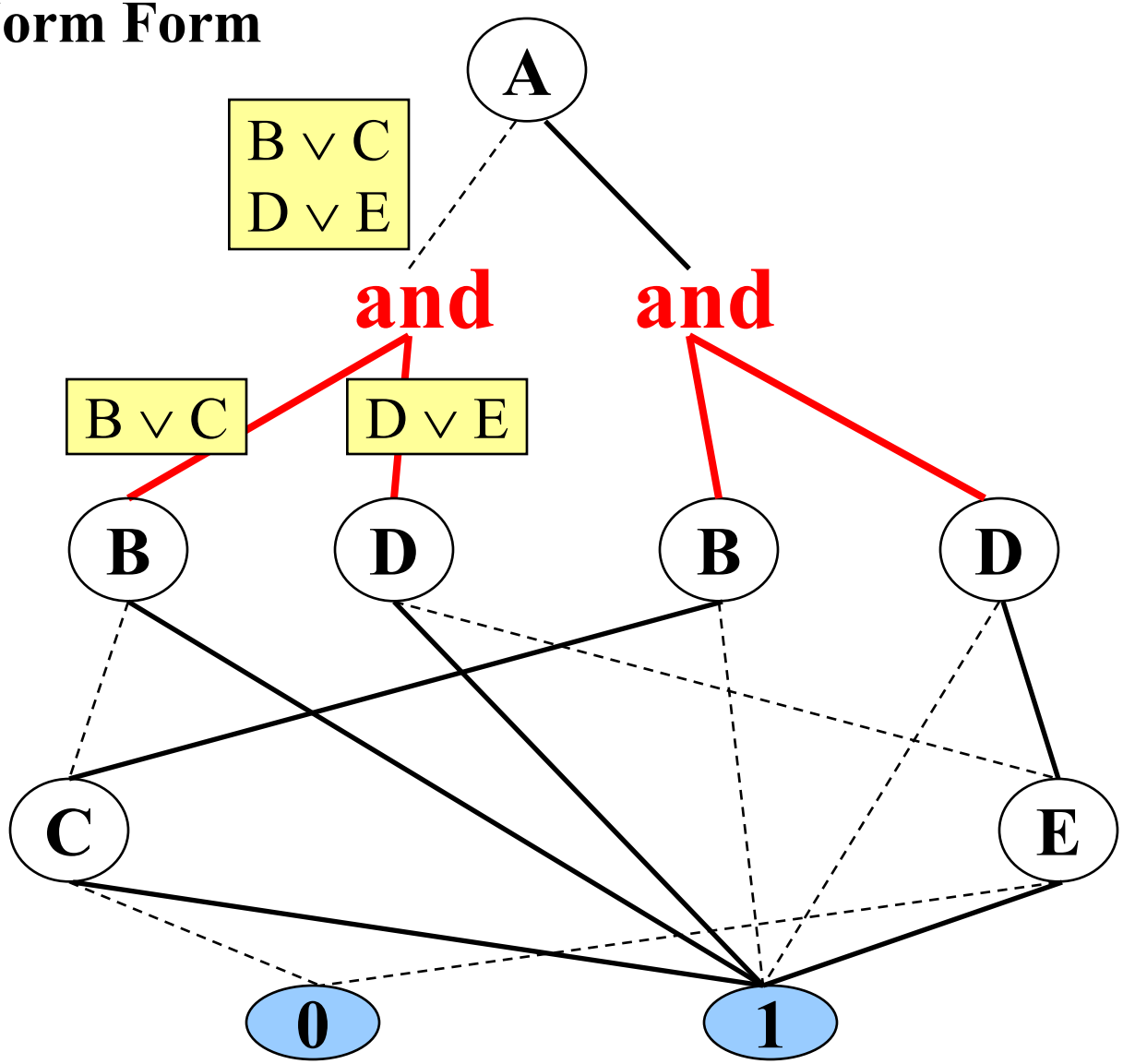
d-DNNF

Combine as AND node

Deterministic
Decomposable
Negation Norm Form

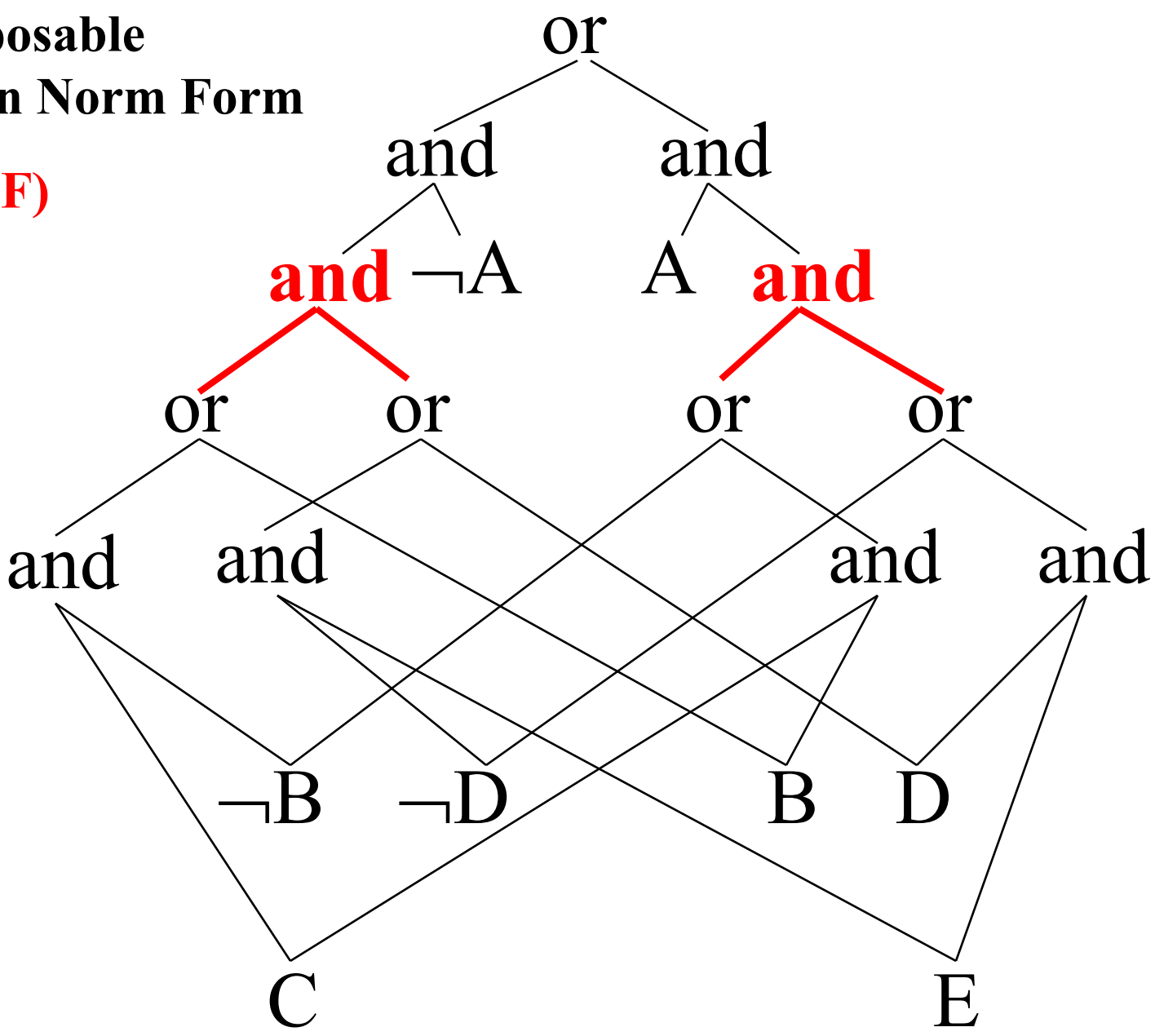
(d-DNNF)

$A \vee B \vee C \qquad \neg A \vee \neg B \vee C$
$A \vee D \vee E \qquad \neg A \vee \neg D \vee E$

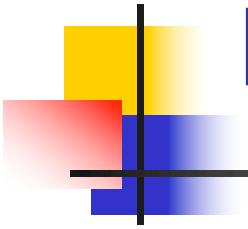


Deterministic
Decomposable
Negation Norm Form

(d-DNNF)

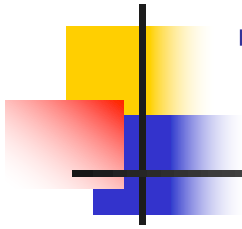


Decomposition Strategies



- Static (c2d)
  - Hypergraph partitioning pre-compilation
  - Strategy captured using a dtree (decomposition tree)
- Dynamic (Dsharp/Cachet)
  - Dynamic variable orderings
  - Lazy detection of decompositions
- Mixed (D4)
  - Hypergraph partitioning during compilation
  - Done selectively (using dtrees)

The Language of Search



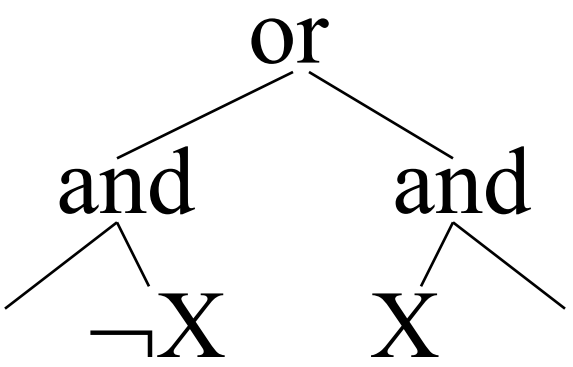
Fixed Variable Ordering → **OBDD**

Plain DPLL → **FBDD**

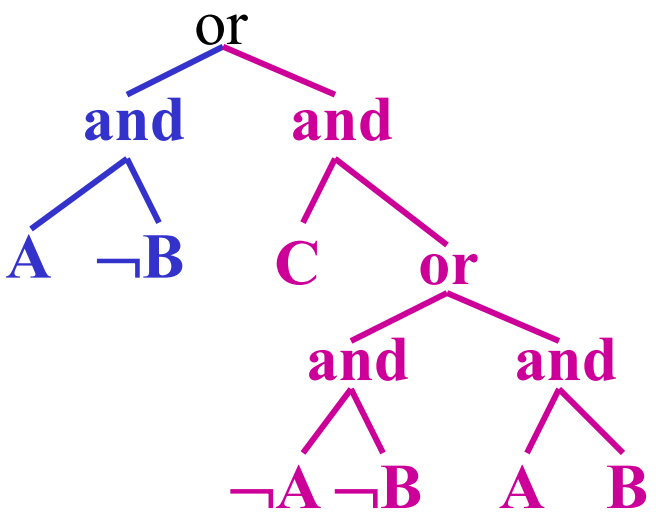
Allowing Decomposition → **d-DNNF**

Limitation of DPLL: General determinism

Decision nodes (Decision-DNNF)

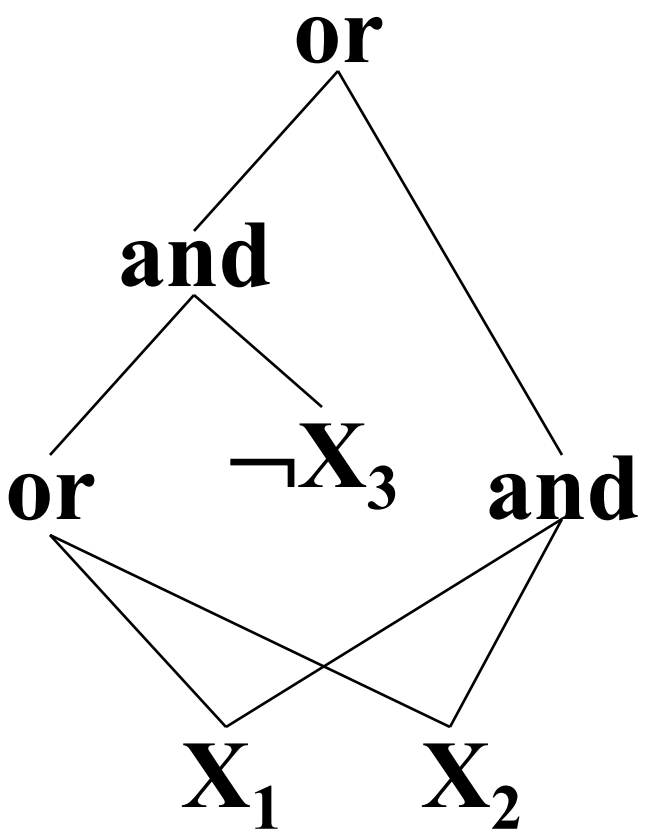


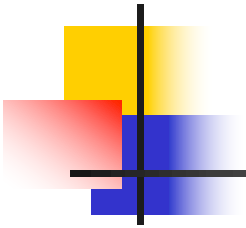
Deterministic nodes (d-DNNF)



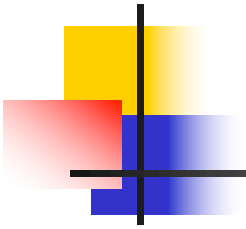
Beyond DPLL:
Decomposability (D) without determinism (d)

DNNF:
CO, CE, ME, exist quantification





Bottom-up Compilation



Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators

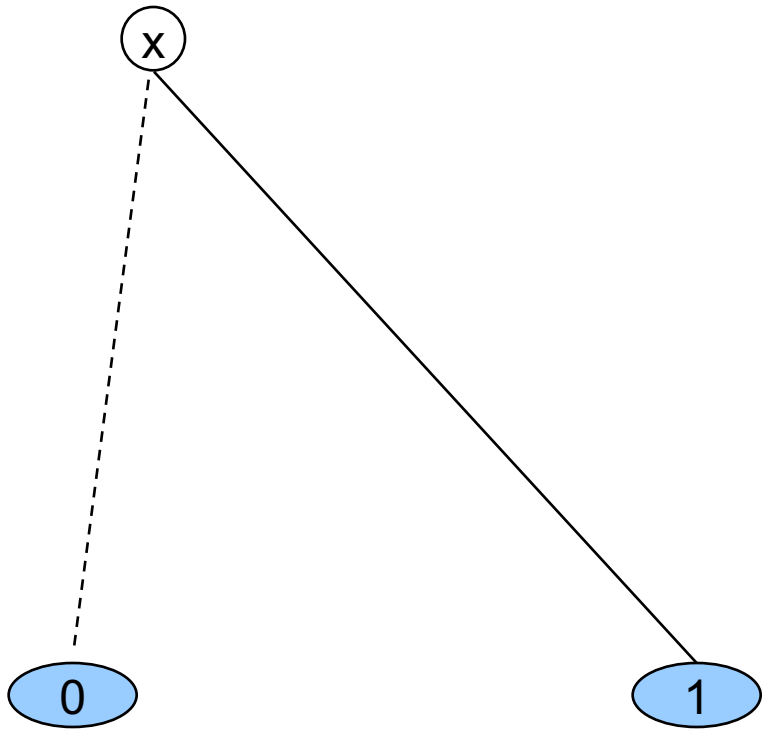
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators



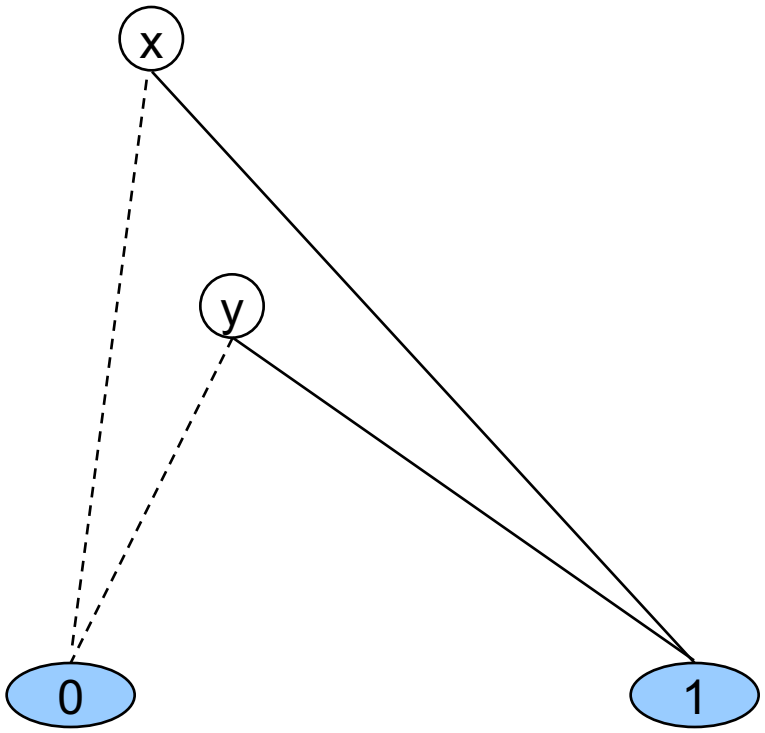
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators



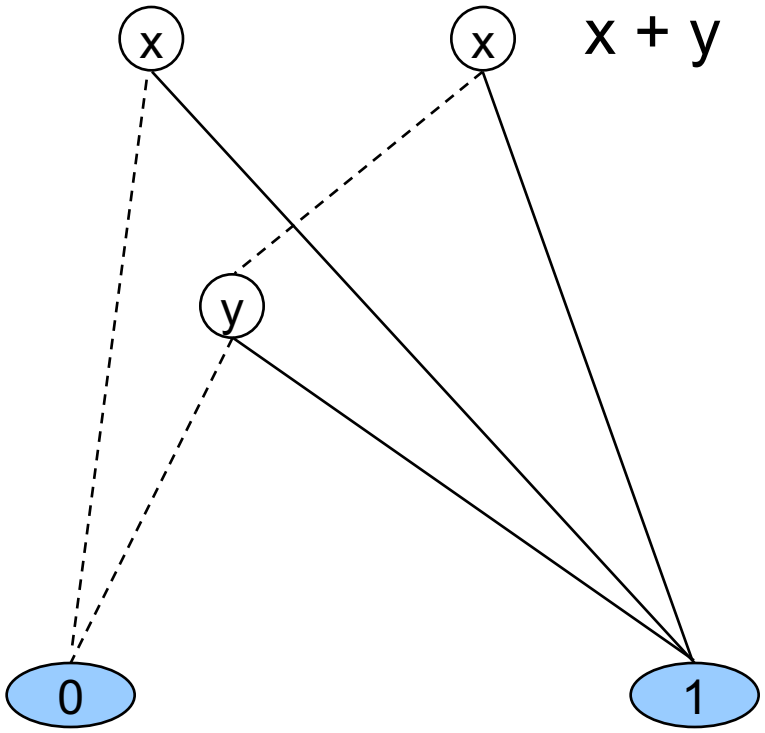
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators



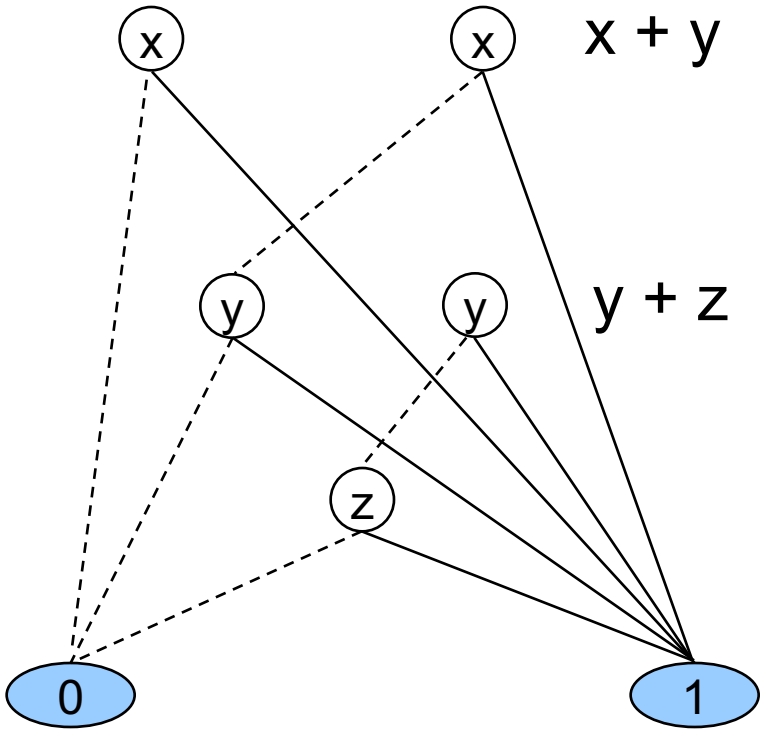
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators



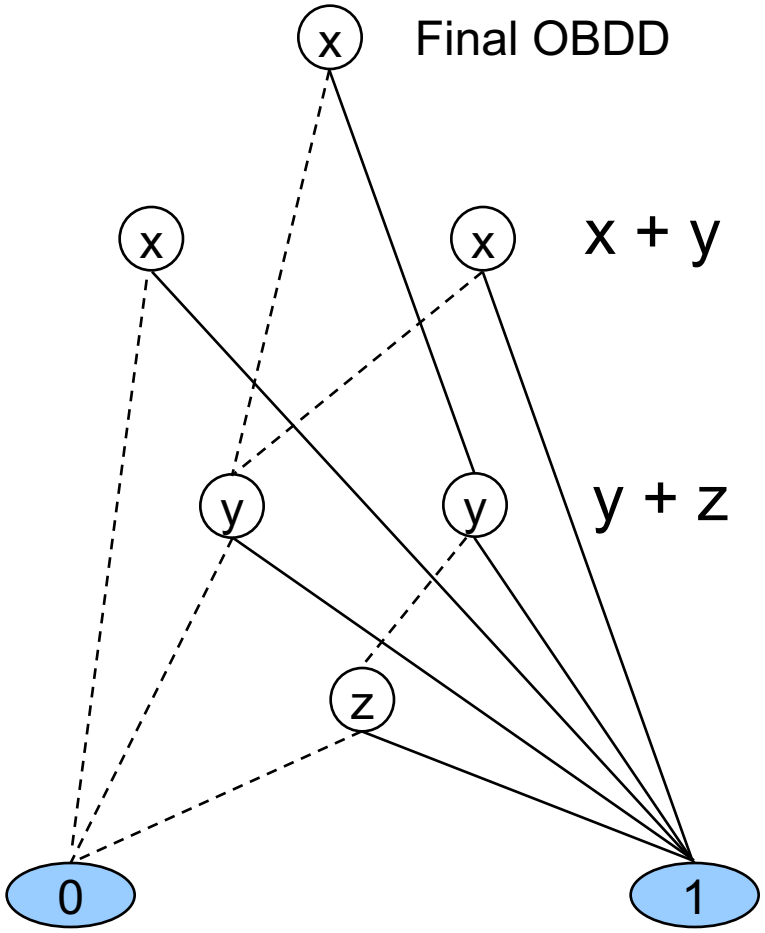
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators



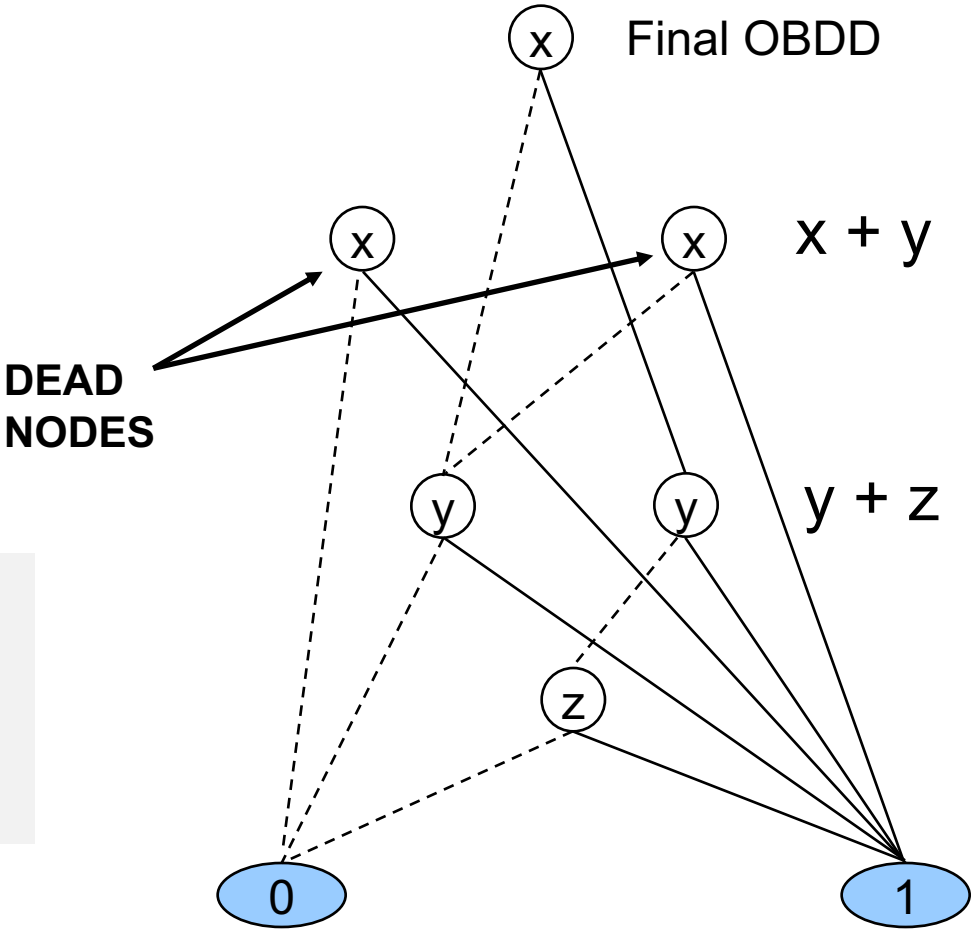
Bottom-up Compilation

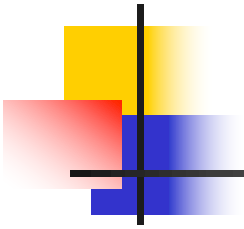
CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators

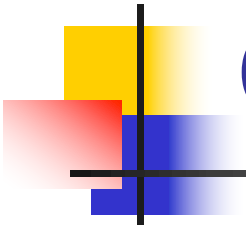




Bottom-up Compilation

- Requires:
  - *Apply* (conjoin, disjoin, etc)
  - Garbage collection of dead nodes
- Challenges:
  - Good variable order
  - Good schedule of Apply operations
- uf100-08 (32 models):
  - 176 nodes in final OBDD under MINCE variable order
  - 30,640,582 intermediate nodes using CUDD package

Canonicity in Compilation

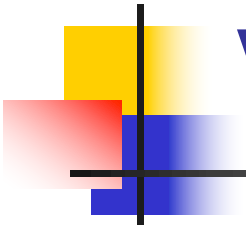


- OBDDs are canonical
  variable order → unique OBDD

  (reduced OBDDs)

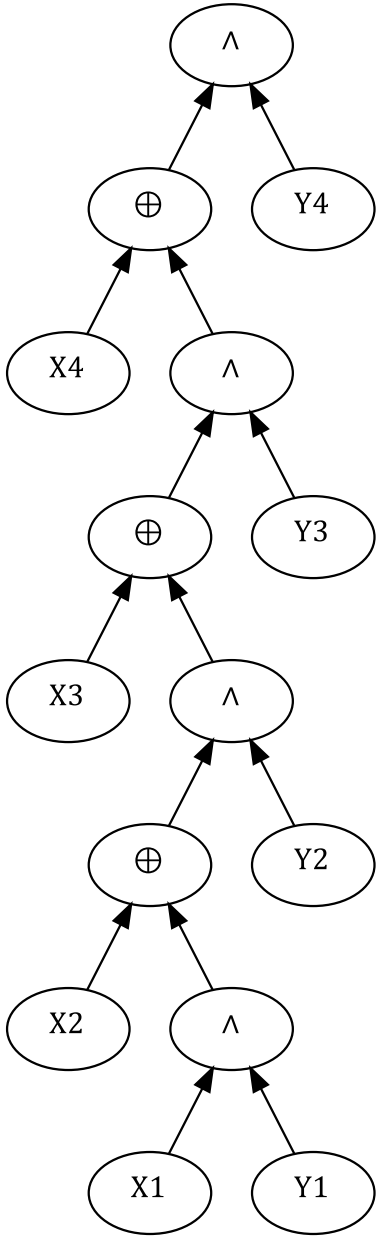
- SDDs are canonical
  vtree → unique SDD

  (trimmed and compressed SDDs)



Vtrees Matter!

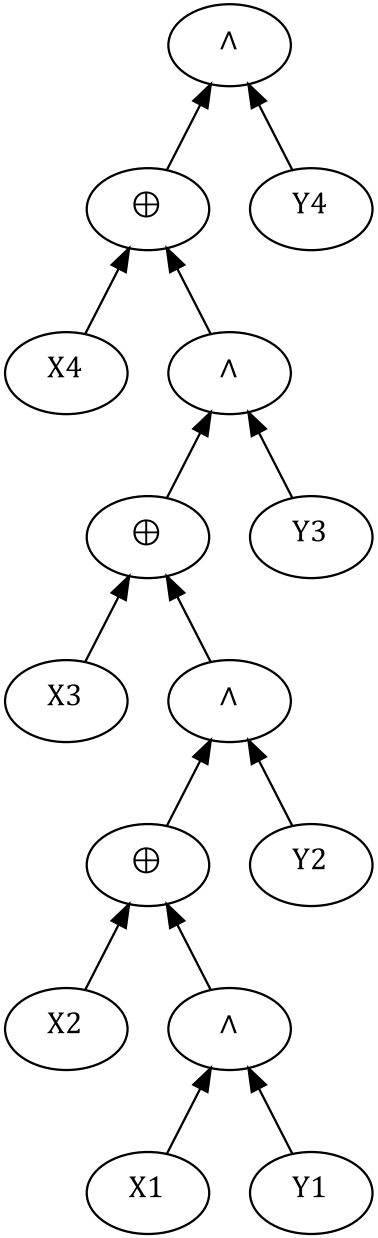
- A vtree can have a significant impact on the size of an SDD
- Good vtrees can be obtained either
  - **Statically**: by analyzing the Boolean function structure before compilation
  - **Dynamically**: by searching for an appropriate vtree during compilation



variables

$\mathbf{X} = \{ X_1\ X_2\ X_3\ \ldots\ X_n \}$

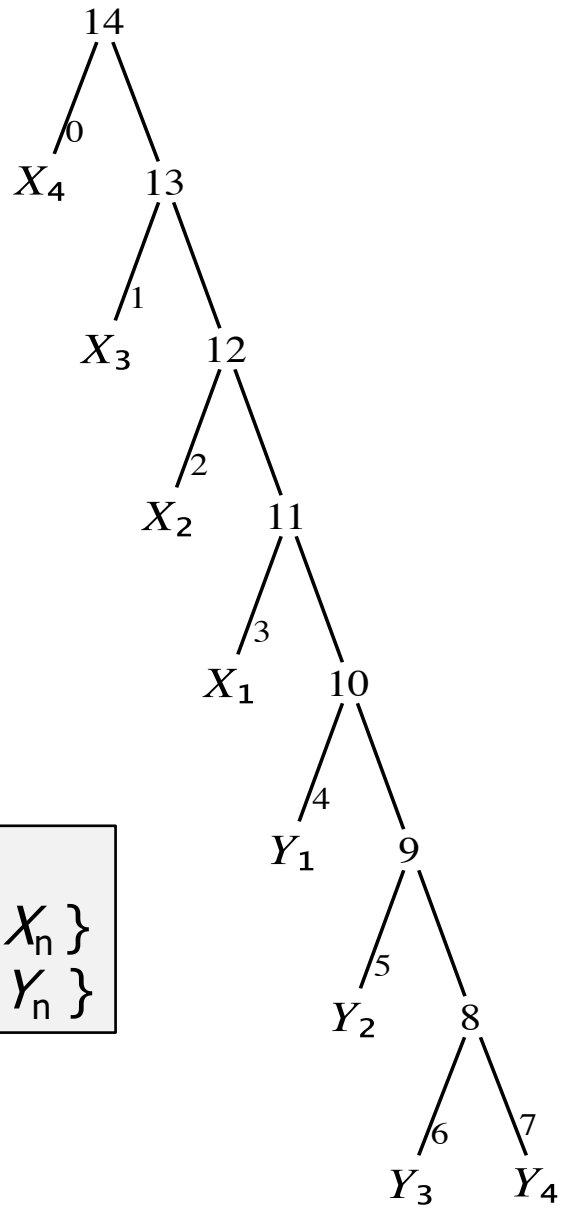
$\mathbf{Y} = \{ Y_1\ Y_2\ Y_3\ \ldots\ Y_n \}$

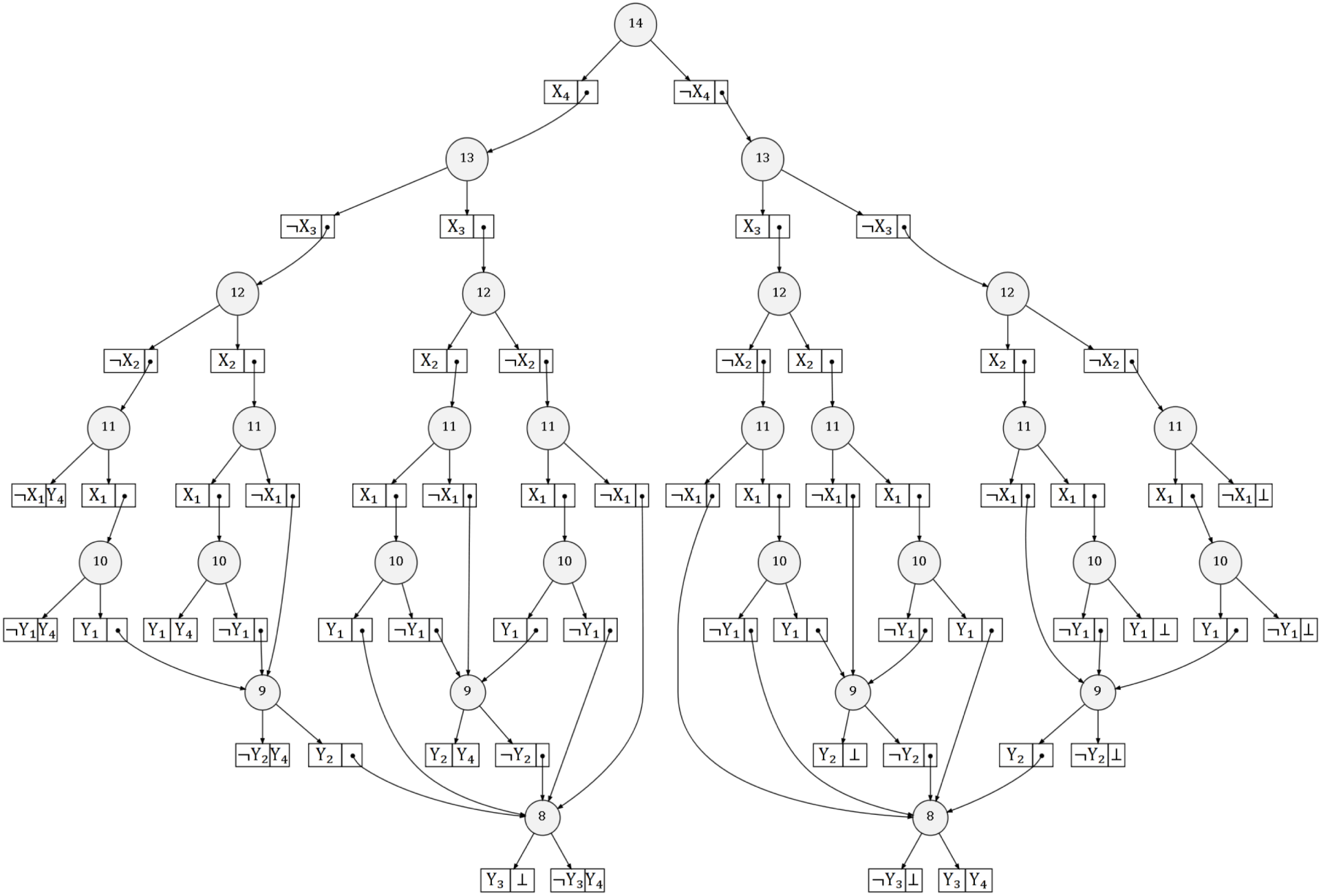


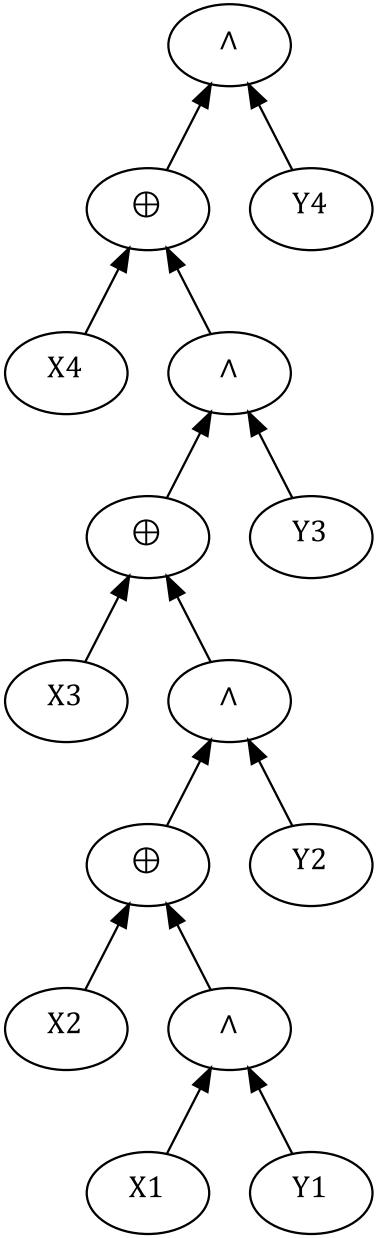
variables

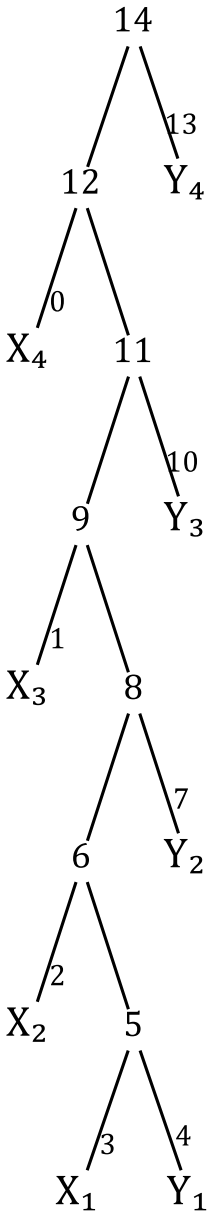
X = { X_1 X_2 X_3 ... X_n }

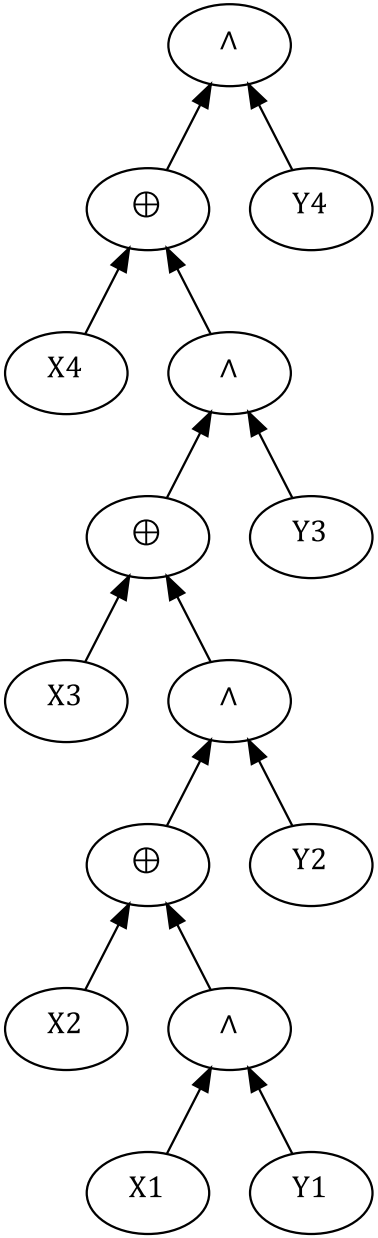
Y = { Y_1 Y_2 Y_3 ... Y_n }

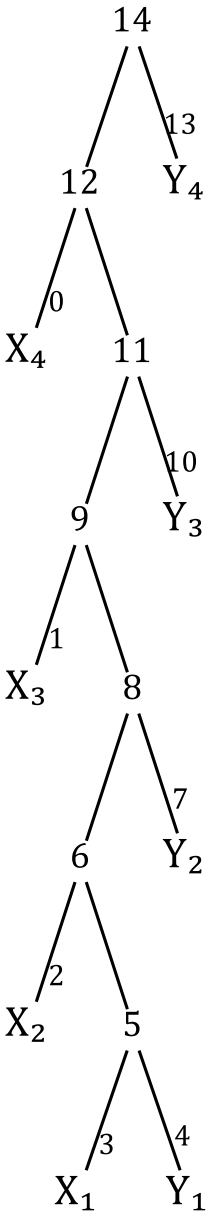


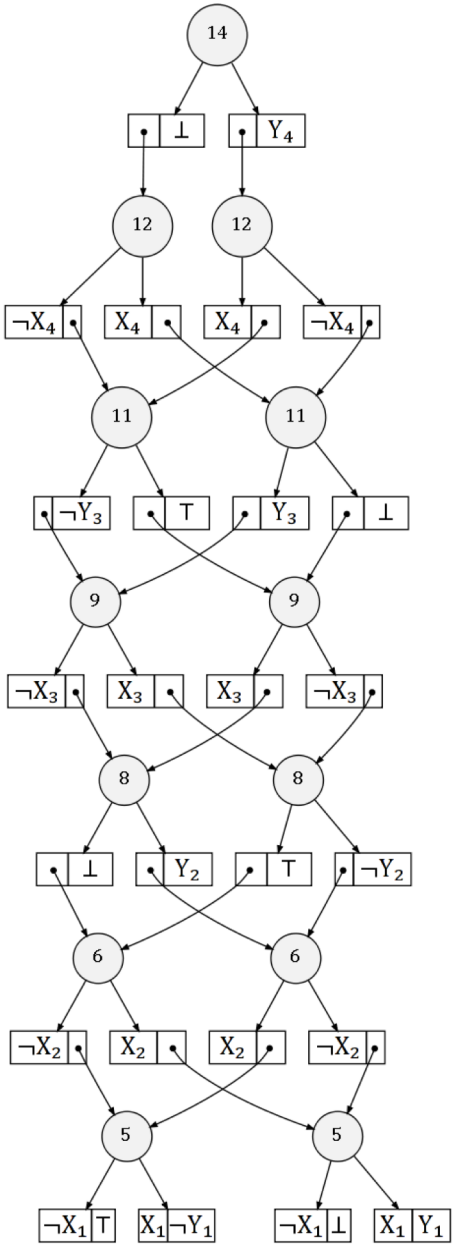

14
X_4
$\neg X_4$
13
13
$\neg X_3$
X_3
X_3
$\neg X_3$
12
12
12
12
$\neg X_2$
X_2
X_2
$\neg X_2$
$\neg X_2$
X_2
X_2
$\neg X_2$
11
11
11
11
11
11
11
11
$\neg X_1$ Y_4
X_1
X_1
$\neg X_1$
X_1
$\neg X_1$
X_1
$\neg X_1$
$\neg X_1$
X_1
$\neg X_1$
X_1
$\neg X_1$
X_1
$\neg X_1$ $\bot$
X_1
10
10
10
10
10
10
10
10
$\neg Y_1$ Y_4
Y_1
Y_1 Y_4
$\neg Y_1$
Y_1
$\neg Y_1$
Y_1
$\neg Y_1$
$\neg Y_1$
Y_1
$\neg Y_1$
Y_1
$\neg Y_1$
Y_1
$\neg Y_1$ $\bot$
Y_1
$\neg Y_1$ $\bot$
9
9
9
9
$\neg Y_2$ Y_4
Y_2
Y_2 Y_4
$\neg Y_2$
Y_2 $\bot$
$\neg Y_2$
Y_2
$\neg Y_2$ $\bot$
8
8
Y_3 $\bot$
$\neg Y_3$ Y_4
$\neg Y_3$ $\bot$
Y_3 Y_4

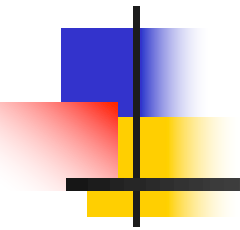

∧
⊕
Y4
X4
∧
⊕
Y3
X3
∧
⊕
Y2
X2
∧
X1
Y1


14
13
12
Y_4
0
X_4
11
10
9
Y_3
1
X_3
8
7
6
Y_2
2
X_2
5
3
4
X_1
Y_1


∧
⊕
Y4
X4
∧
⊕
Y3
X3
∧
⊕
Y2
X2
∧
X1
Y1

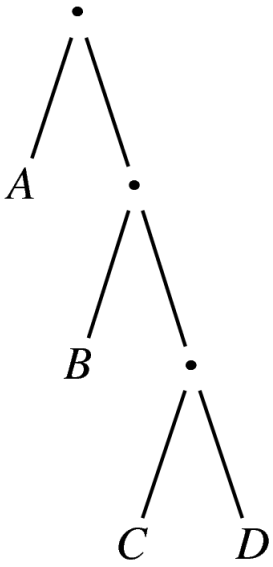

14
12
13
Y4
0
X4
11
9
10
Y3
1
X3
8
6
7
Y2
2
X2
5
3
4
X1
Y1


14
⊥
Y4
12
12
¬X4
X4
X4
¬X4
11
11
¬Y3
⊤
Y3
⊥
9
9
¬X3
X3
X3
¬X3
8
8
⊥
Y2
⊤
¬Y2
6
6
¬X2
X2
X2
¬X2
5
5
¬X1
⊤
X1
¬Y1
¬X1
⊥
X1
Y1

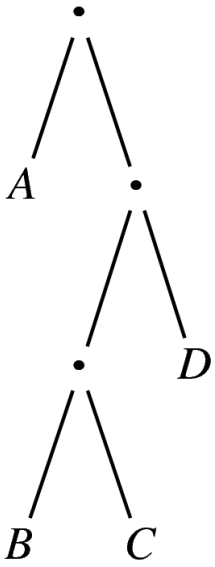


Vtree Search: Minimizing SDD Size

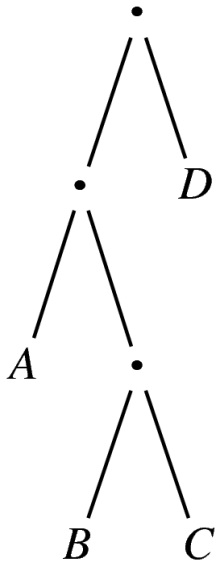
Dissecting Variable Orders



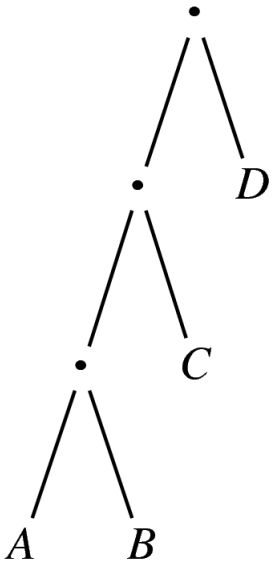
ABCD



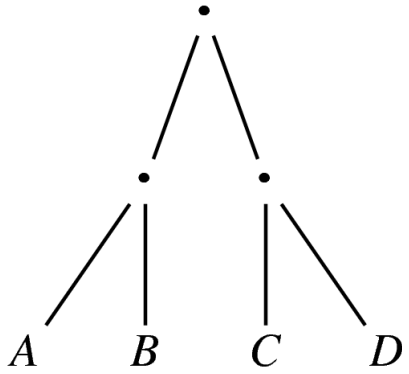
ABCD



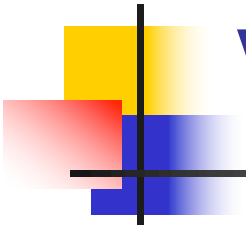
ABCD



ABCD

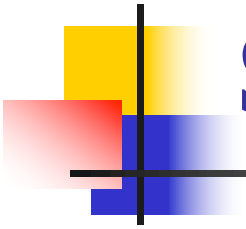


ABCD



Vtrees Matter!

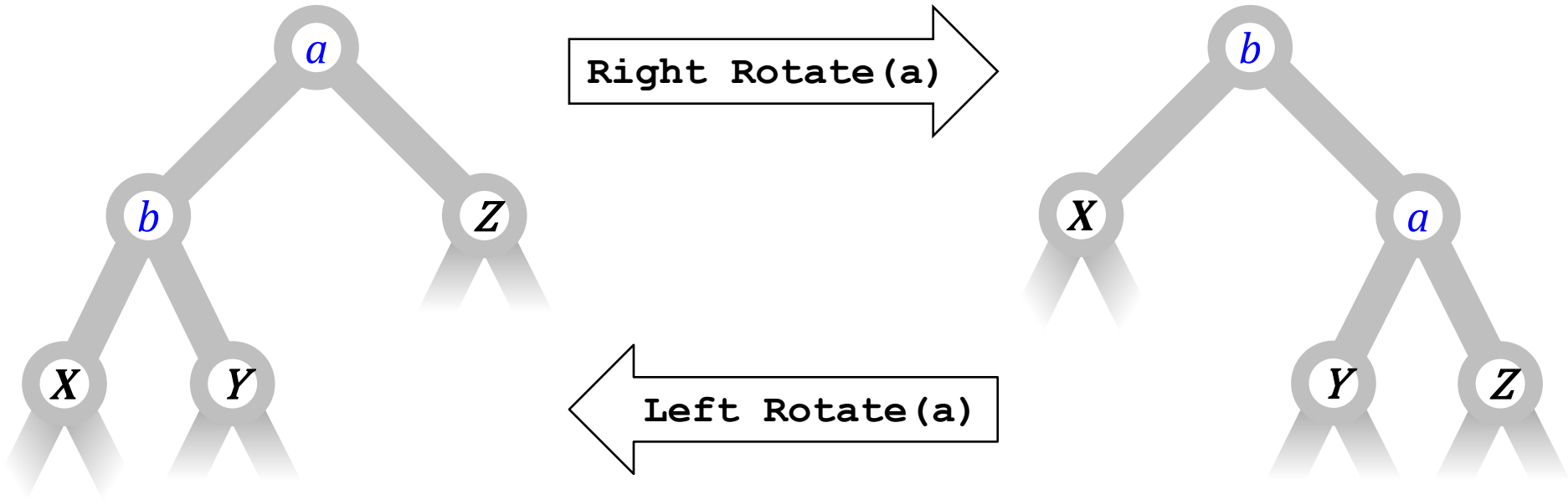
- The choice of a vtree can lead to exponential difference in the size of an SDD
- The choice of a dissection can also lead to exponential differences in the size of an SDD



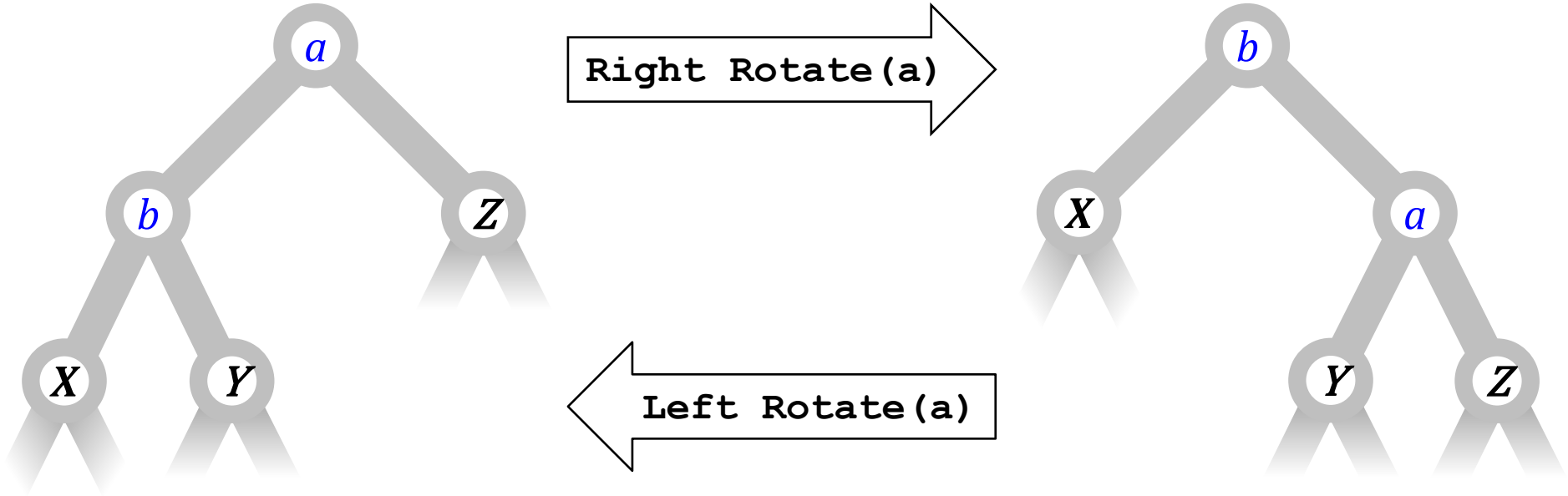
Searching Over Vtrees

- Double search problem:
  - Find variable order
  - Find dissection
- Tree operations:
  - Rotation (left, right)
  - Swapping
- Can enumerate all vtrees

Tree Rotations

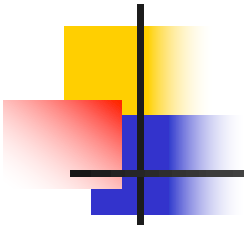


Rotation Preserves Order



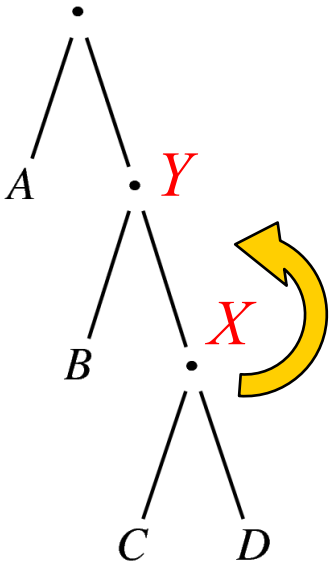
X, Y, Z

X, Y, Z

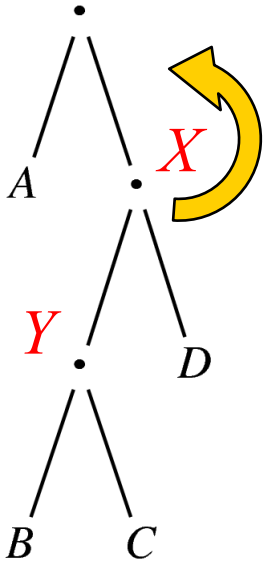


Enumerating Dissections

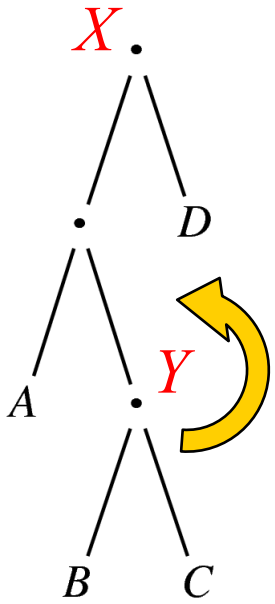
- Rotations can enumerate all dissection of a given variable order
- Systematic methods exist for this purpose
- See, e.g., Knuth's
  Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees



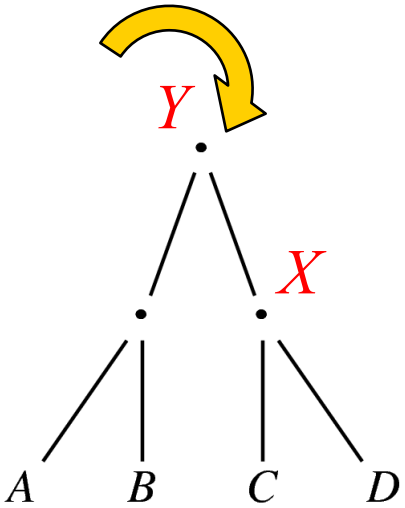
ABCD



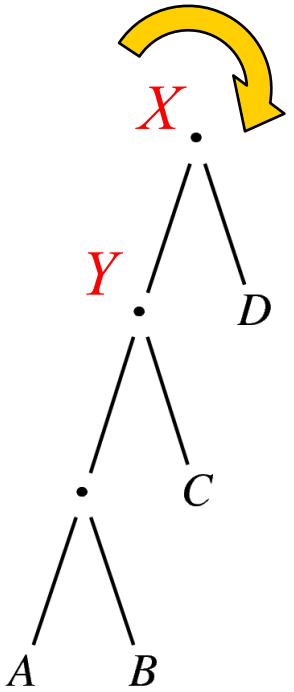
ABCD



ABCD

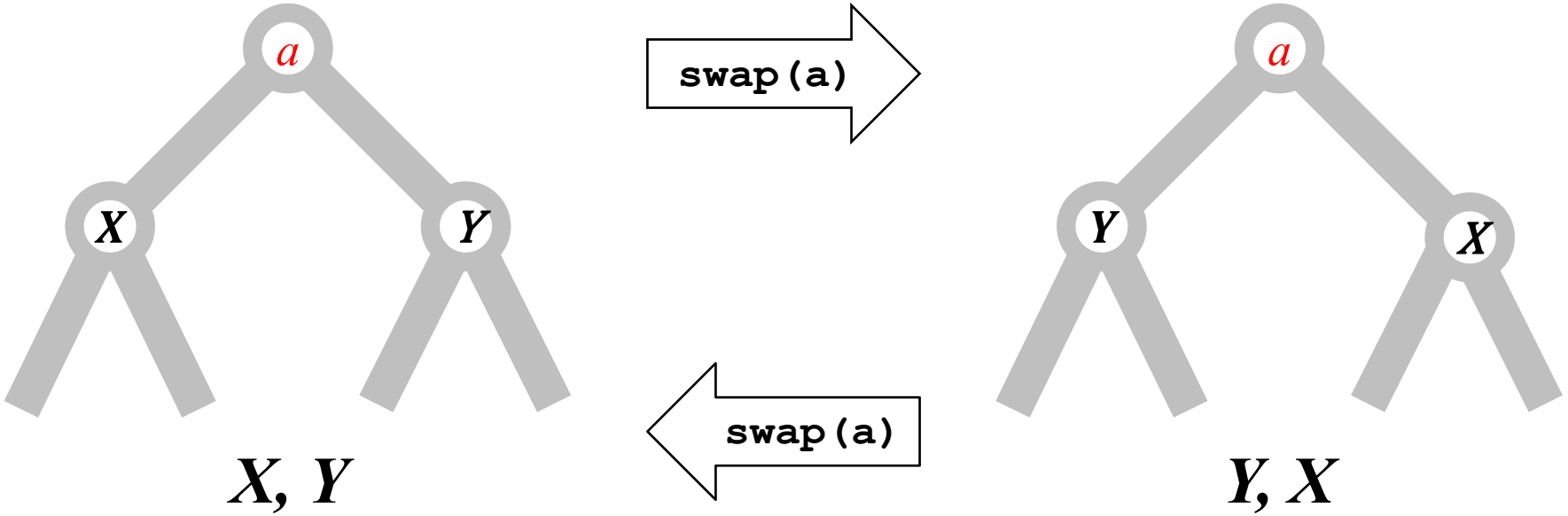


ABCD

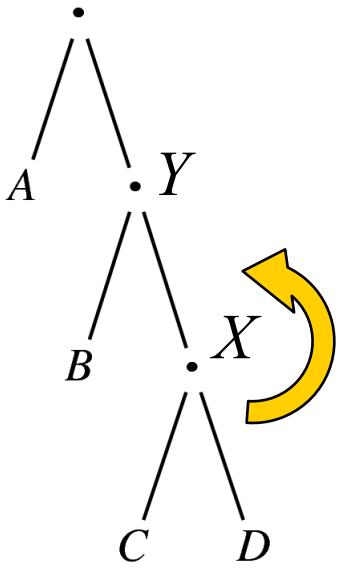


ABCD

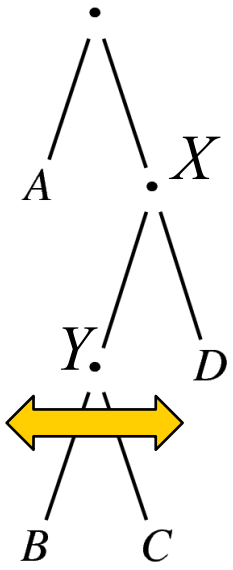
Swapping Changes Order



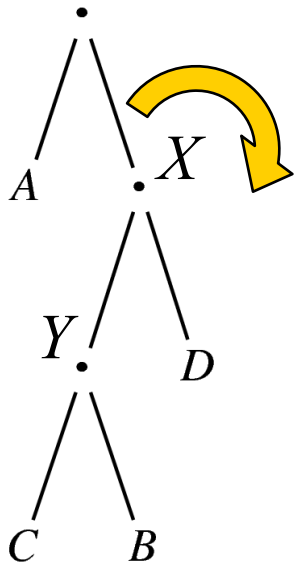
Rotate + Swap



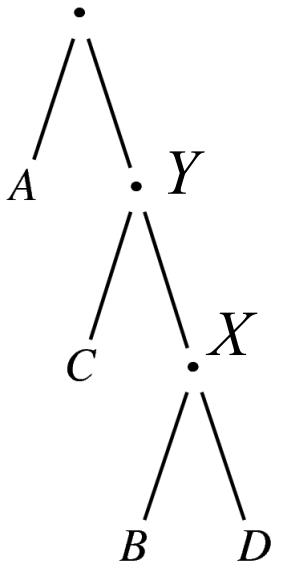
ABCD



ABCD

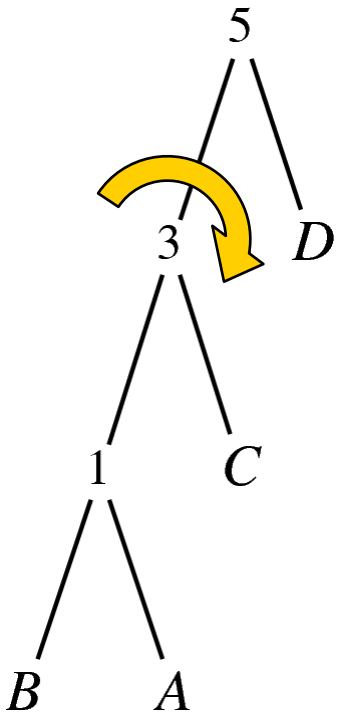


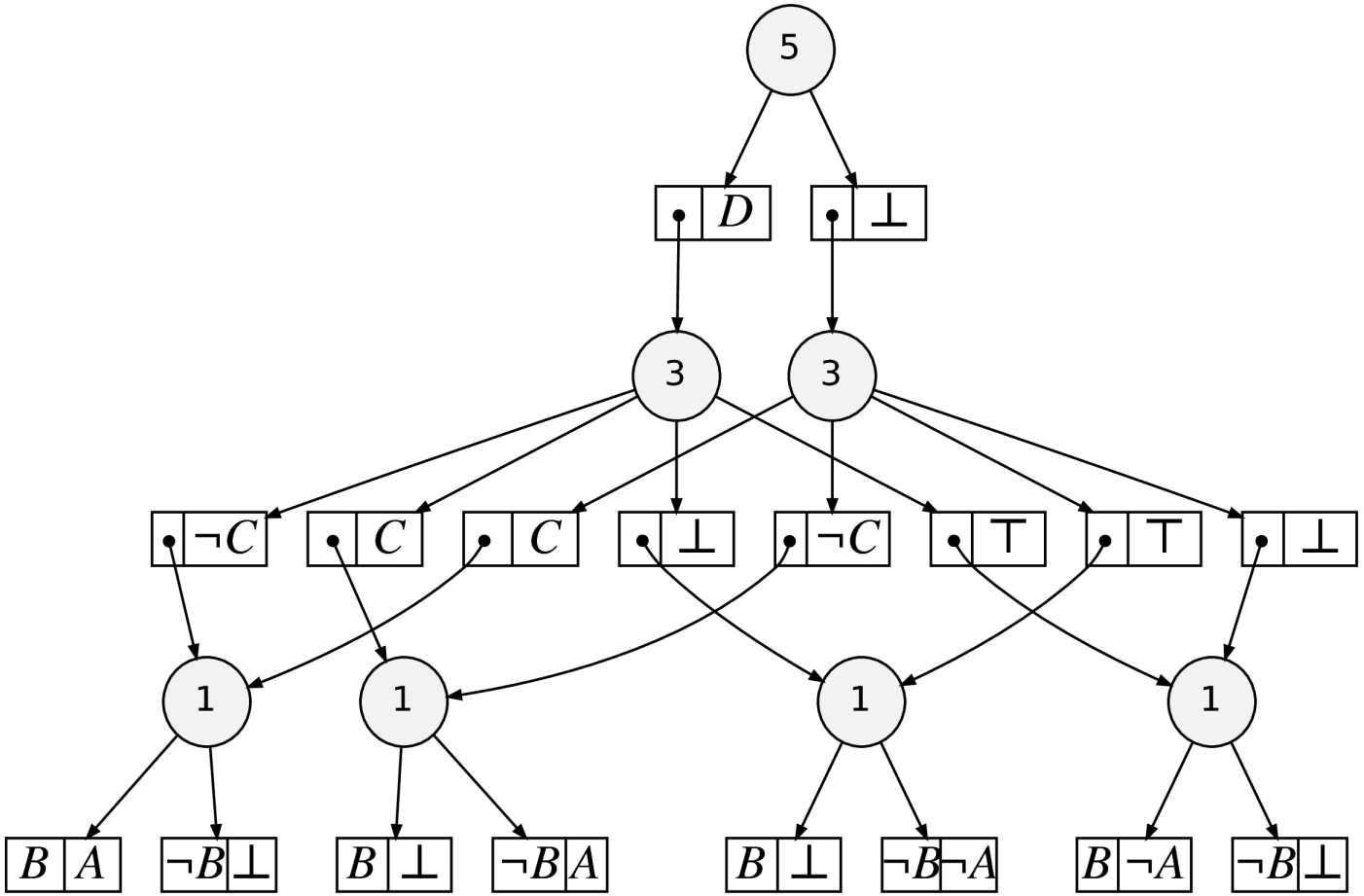
ACBD



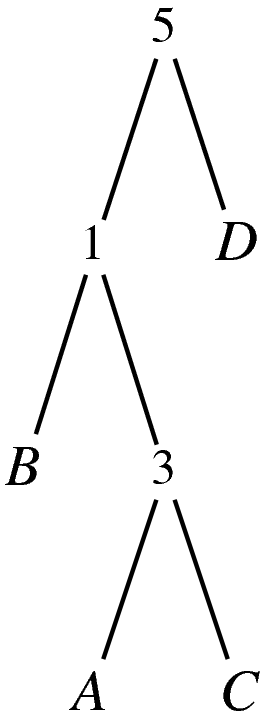
ACBD

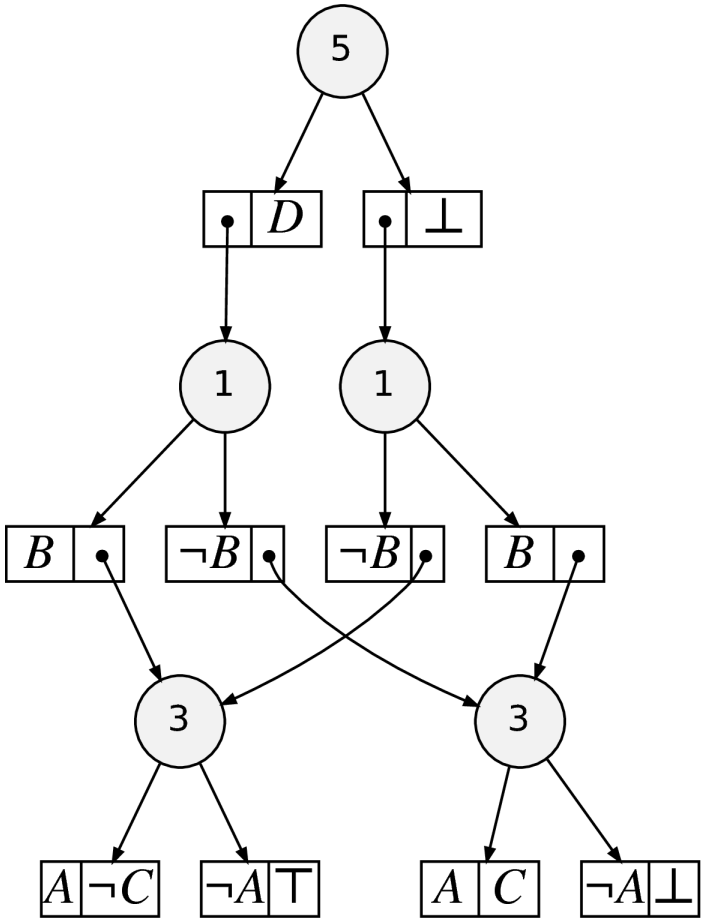
The SDD Package

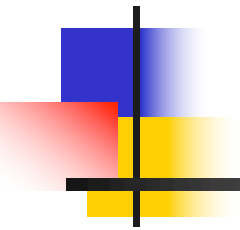




The SDD Package

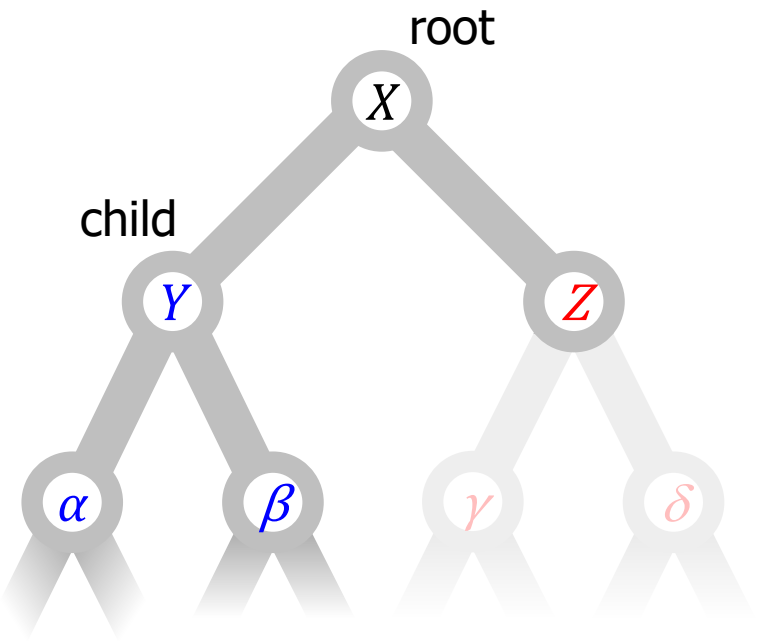






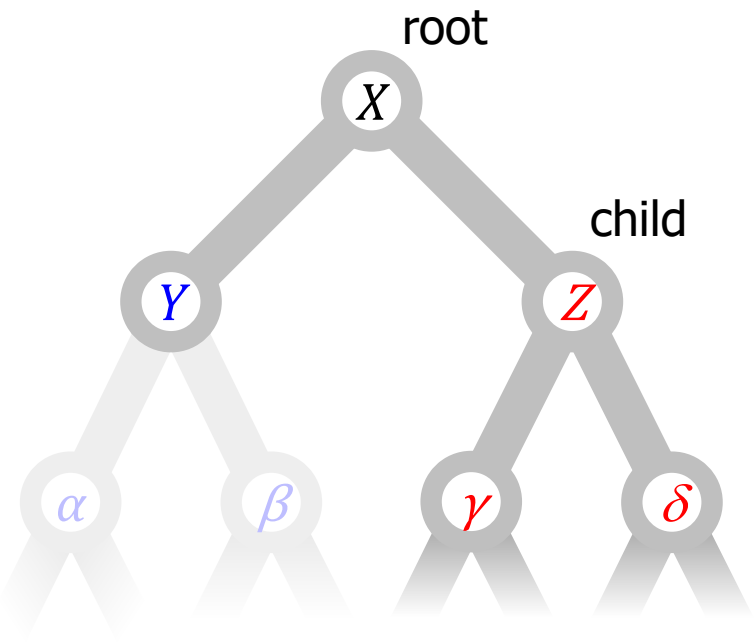
The Fragment Abstraction

Vtree Fragments

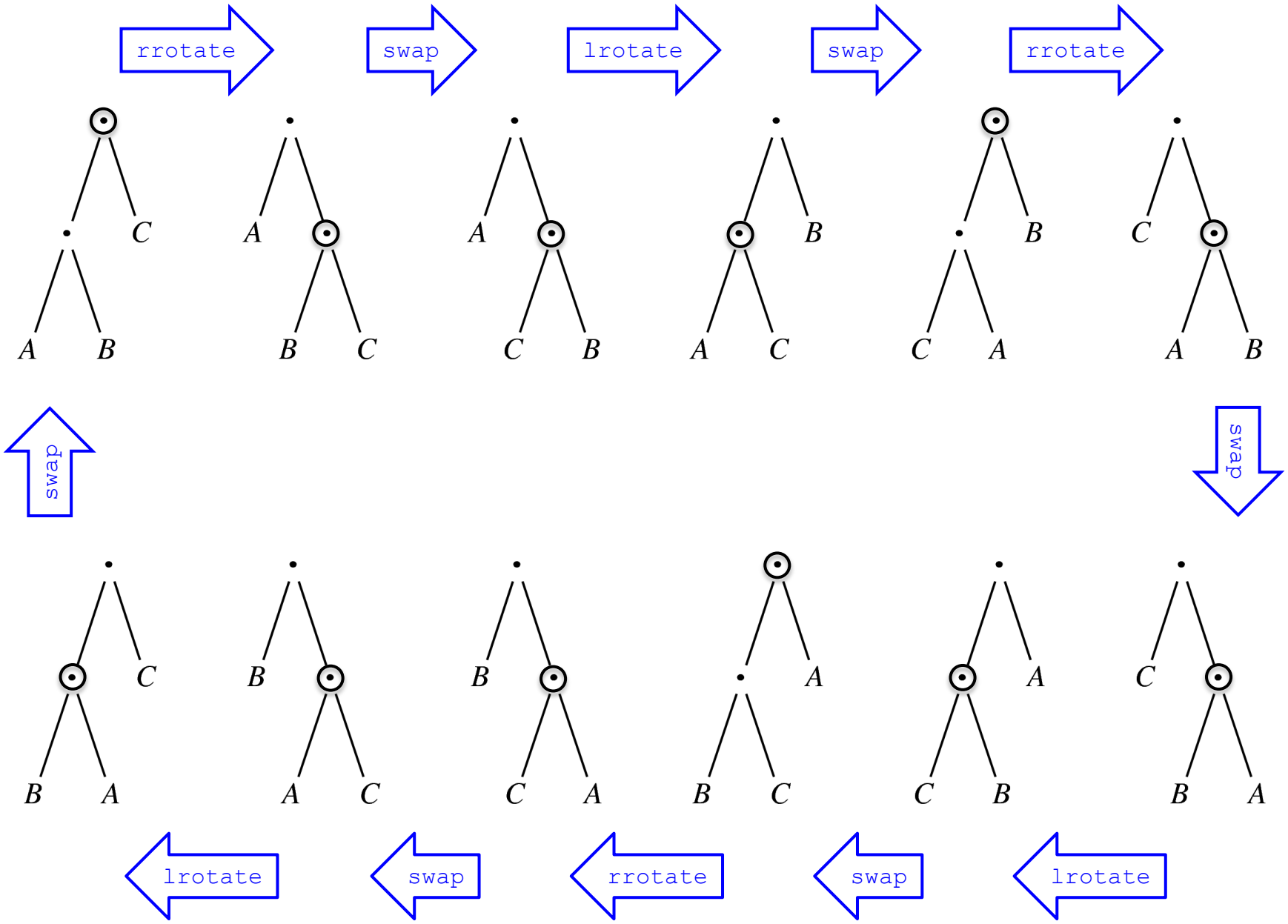


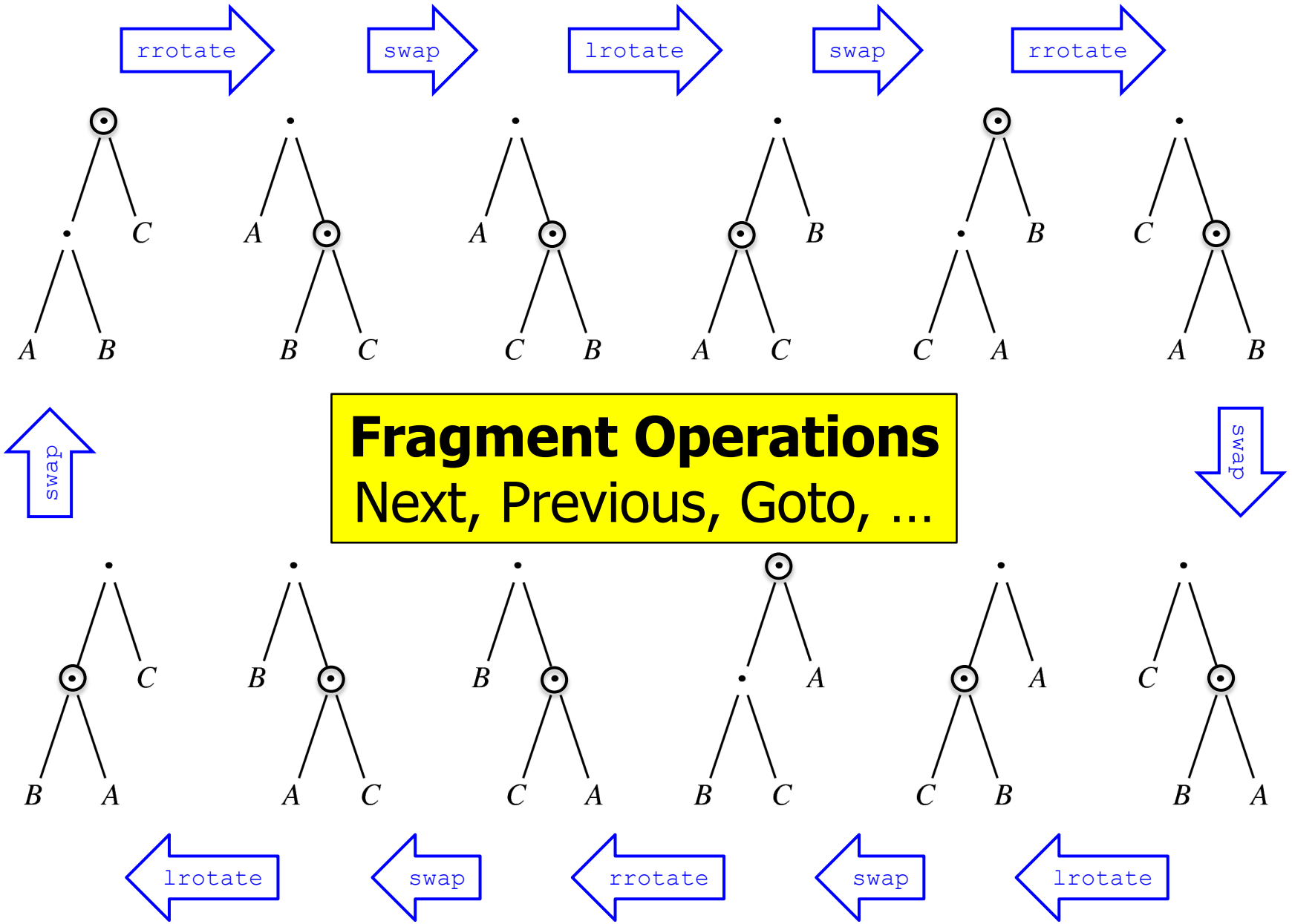
left-linear fragment

Vtree Fragments



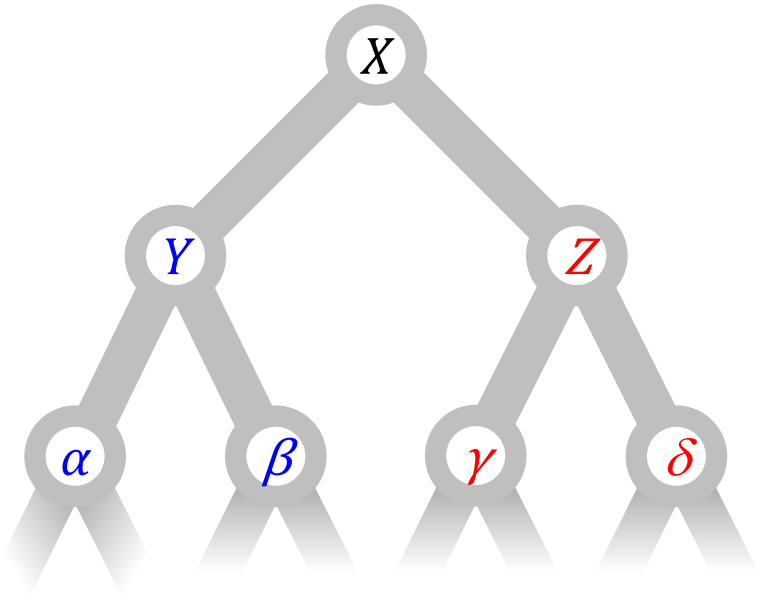
right-linear fragment


rrotate
swap
lrotate
swap
rrotate
A
B
C
swap
swap
lrotate
swap
rrotate
swap
lrotate


rrotate
swap
lrotate
swap
rrotate
A
B
C
swap
swap
Fragment Operations
Next, Previous, Goto, ...
lrotate
swap
rrotate
swap
lrotate

Greedy Vtree Search

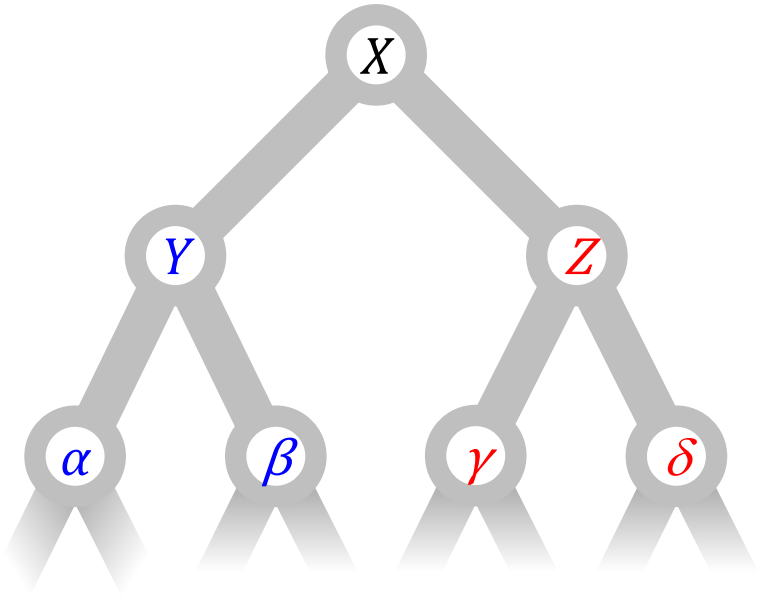
- Traverse vtree bottom-up
- Try enumerating 12 vtrees of left-linear fragment
- Try enumerating 12 vtrees of right-linear fragment
- 24 vtrees in total
- Greedily accept best vtree
- Prune parts of search space

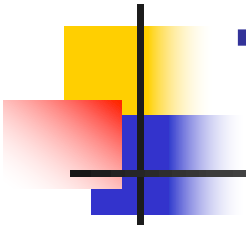


The SDD Package

Opportunities

- vtree search algorithms
- Triggers for vtree search
- Static methods for constructing vtrees
- Heuristics for scheduling Apply operations





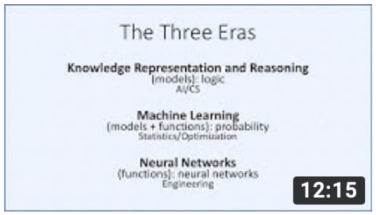
The SDD Package

- http://reasoning.cs.ucla.edu/sdd
- Written in C
- Available as open-source
- Code for compiling CNFs/DNFs into SDDs (includes heuristic for scheduling Apply)
- Two manuals: beginner and advanced

UCLA Automated Reasoning Group

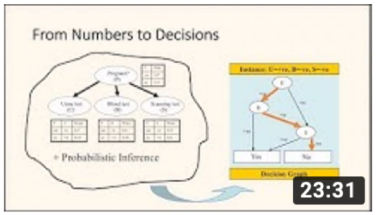
CACM Oct. 2018 - Human-Level Intelligence or Animal…

Association for Computing Ma…
2.7K views • 4 months ago



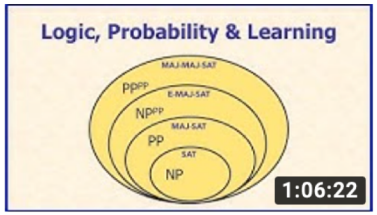
Adnan Darwiche – On AI Education

UCLA Automated Reasoning G…
1.4K views • 5 months ago



Adnan Darwiche – Explaining and Verifying AI Systems

UCLA Automated Reasoning G…
464 views • 4 months ago



Adnan Darwiche – On the Role of Logic in Probabilisti…

UCLA Automated Reasoning G…
2K views • 1 year ago

Adnan Darwiche – On Model-Based versus Model-Blind…

UCR School of Public Policy
3.3K views • 1 year ago

Learning and Reasoning with Bayesian Networks ▶ PLAY ALL

Lectures by Adnan Darwiche for his UCLA course on Bayesian Networks.

Subscribe!

1a. Course Overview with a Historical Perspective on AI

UCLA Automated Reasoning Group • 2.2K views • 1 year ago

Adnan Darwiche's UCLA course: Learning and Reasoning with Bayesian Networks.