Exhaustive DPLL for Model Counting and Knowledge Compilation

Umut Oztok and Adnan Darwiche

University of California, Los Angeles

June 4, 2015

11 years ago…

11 years ago…

Exhaustive DPLL for

- model counting [Sang et al., 2004]: **cachet**
- knowledge compilation [Darwiche 2004]: **c2d**

Using techniques:

- clause learning
- component caching
- component analysis

Today…

Today…

Exhaustive DPLL for

- model counting [Sang et al., 2004]: **cachet**
- knowledge compilation [Darwiche 2004]: **c2d**

Using techniques:

- clause learning
- component caching
- component analysis

Today…

Exhaustive DPLL for

- model counting [Sang et al., 2004]: **cachet**
- knowledge compilation [Darwiche 2004]: **c2d**

Using techniques:

- clause learning
- component caching
- component analysis

What's the problem?

1. No (friendly) source code
2. No clear semantics
3. No proof of correctness

Contributions

Contributions

What are the contributions?

- An exhaustive DPLL algorithm for
  - model counting
  - knowledge compilation
- Clear semantics
- Proof of correctness
- Open source package
  - simple code, almost like a pseudocode

Contributions

What are the contributions?

- An exhaustive DPLL algorithm for
  - model counting
  - knowledge compilation
- Clear semantics
- Proof of correctness
- Open source package
  - simple code, almost like a pseudocode

How is it possible?

- Abstraction of SAT primitives
- Simplified component analysis (using decision vtrees)
- New control structure

Contributions

What are the contributions?

- An exhaustive DPLL algorithm for
  - model counting
  - knowledge compilation
- Clear semantics
- Proof of correctness
- Open source package
  - simple code, almost like a pseudocode

How is it possible?

- Abstraction of SAT primitives
- Simplified component analysis (using decision vtrees)
- New control structure

Current price?

- Trace of c2d & cachet: Decision-DNNF
- Trace of our algorithm: Decision-SDD
  - s*ubset* of Decision-DNNF

Contributions

What are the contributions?

- An exhaustive DPLL algorithm for
  - model counting
  - knowledge compilation
- Clear semantics
- Proof of correctness
- Open source package
  - simple code, almost like a pseudocode

How is it possible?

- Abstraction of SAT primitives
- Simplified component analysis (using decision vtrees)
- New control structure

Current price?

- Trace of c2d & cachet: Decision-DNNF
- Trace of our algorithm: Decision-SDD
  - s*ubset* of Decision-DNNF

[Oztok & Darwiche, IJCAI, 2015]
Top-down compiler for SDDs
(as opposed to the bottom-up compiler based on the SDD package)

SAT by DPLL Search

SAT by DPLL Search

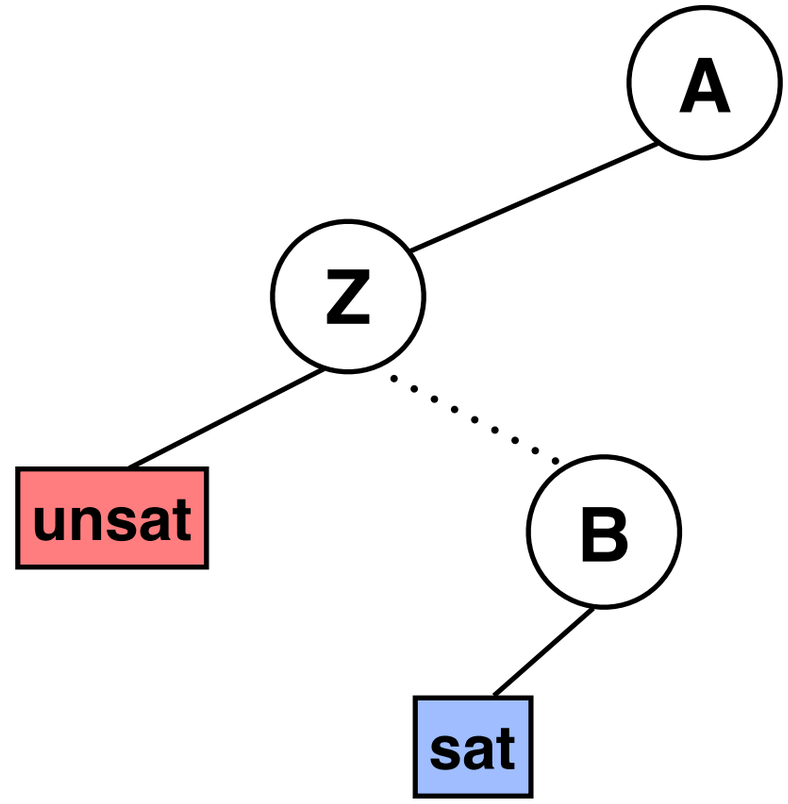
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

SAT by DPLL Search

$\Delta =$

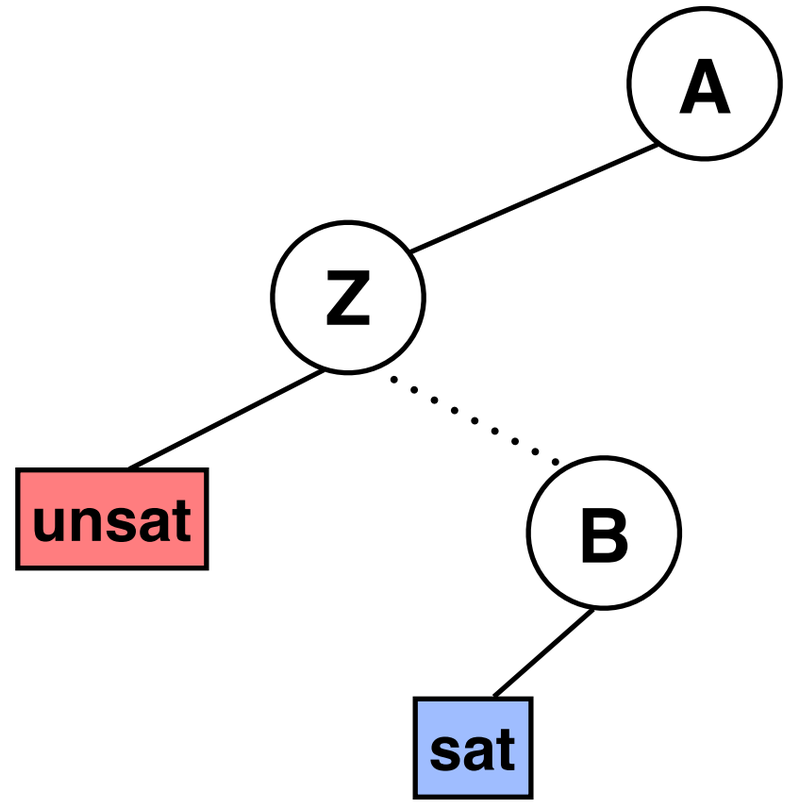
1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}



Exhaustive DPLL

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

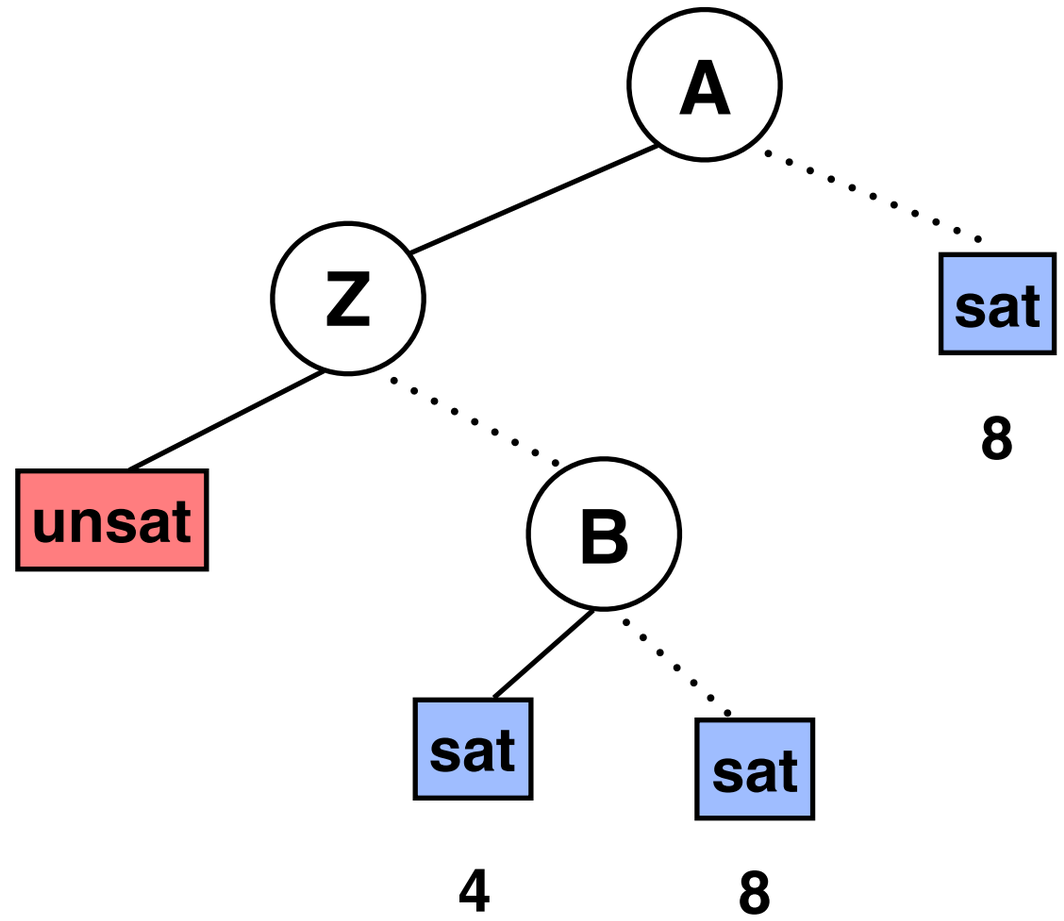


Model Count:

Exhaustive DPLL

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}



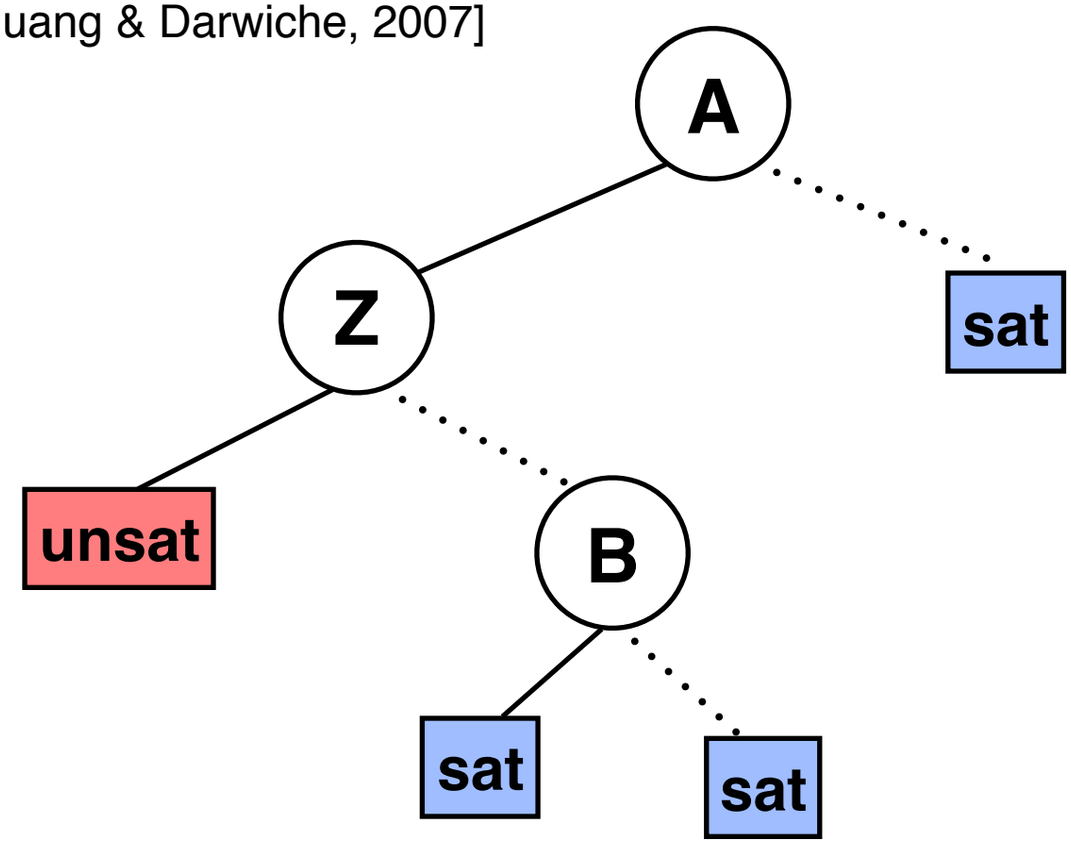
Model Count:

4 + 8 + 8 = 20

Trace of Exhaustive DPLL

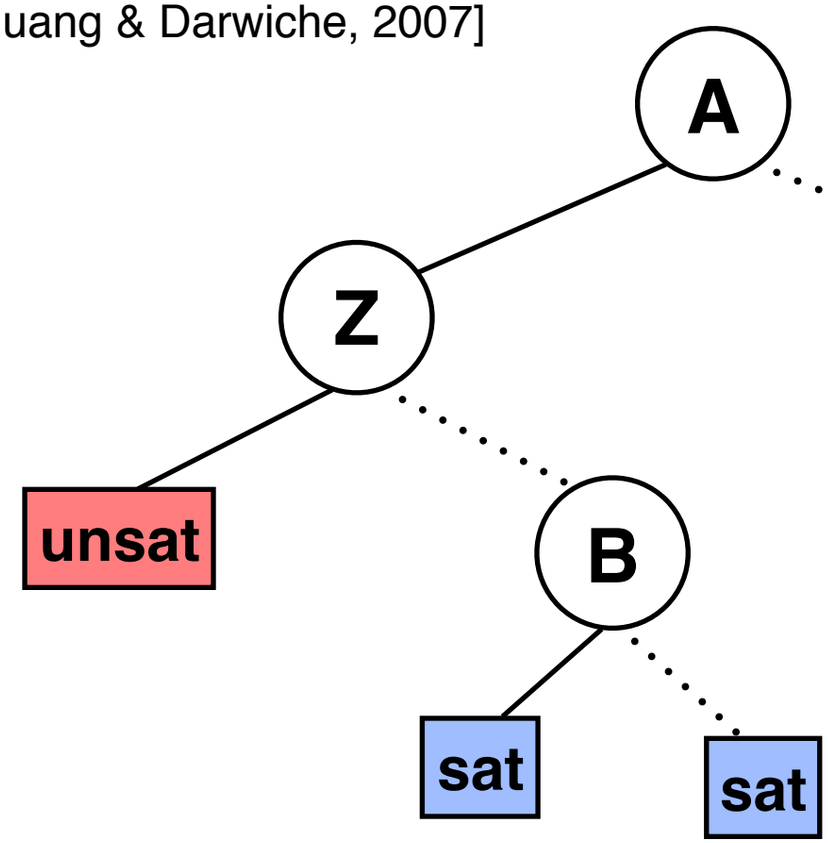
Trace of Exhaustive DPLL

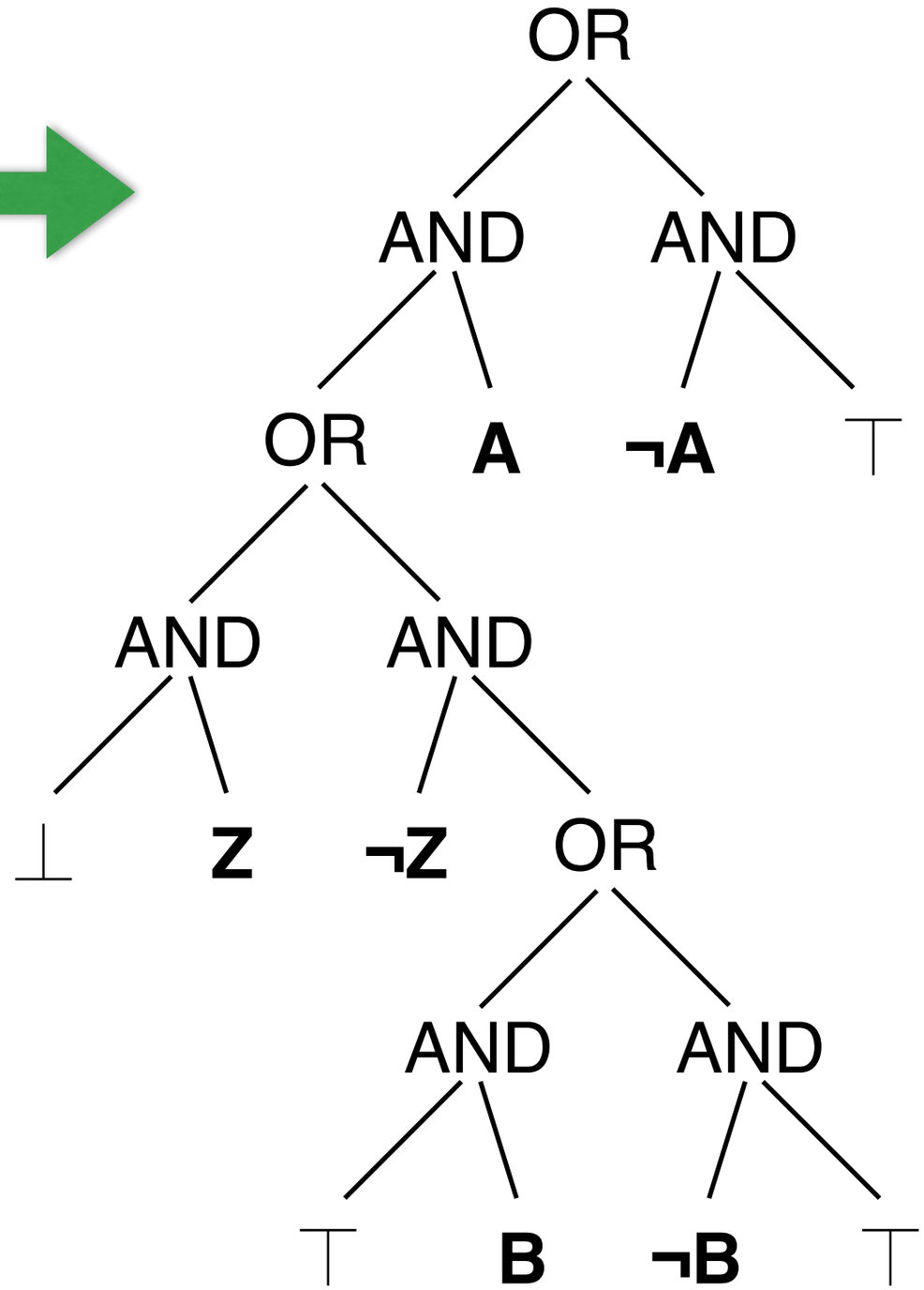
[Huang & Darwiche, 2007]



Trace of Exhaustive DPLL

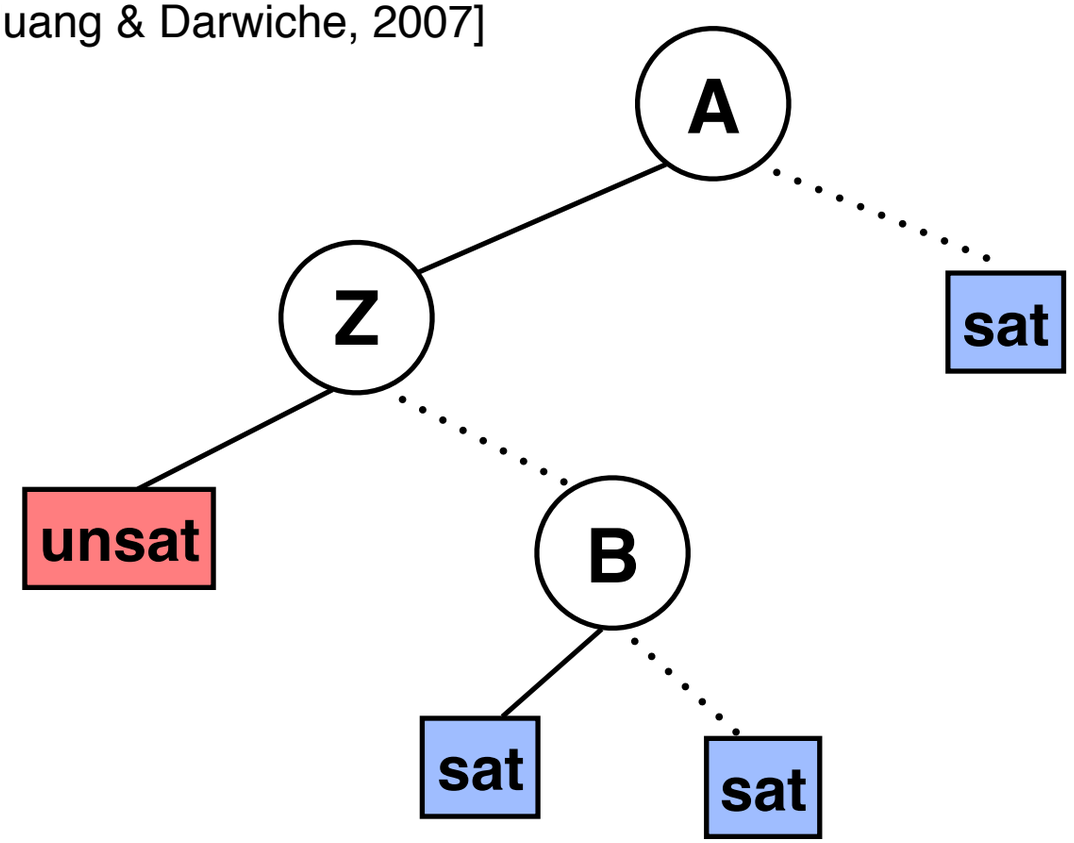
[Huang & Darwiche, 2007]

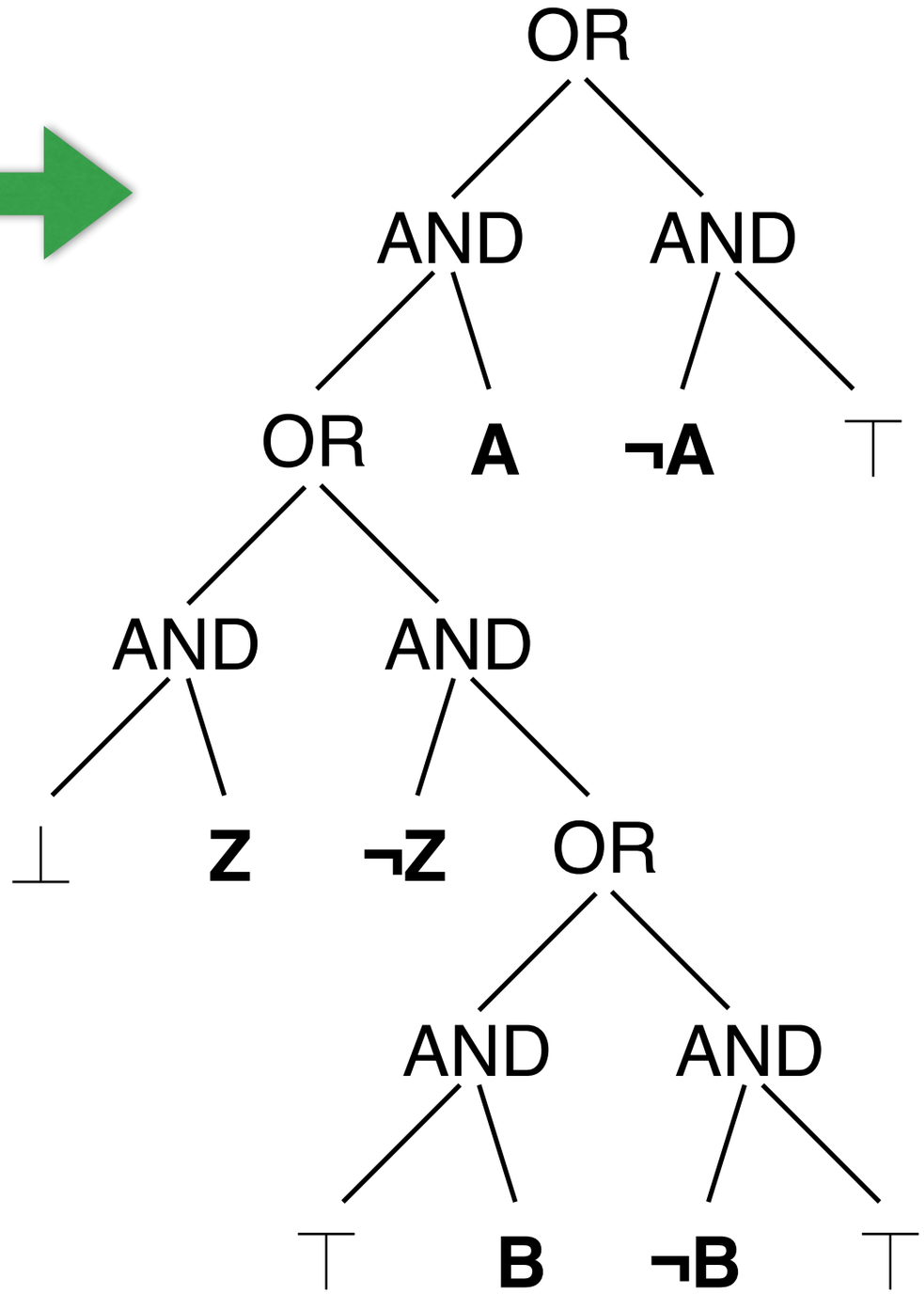




Trace of Exhaustive DPLL

[Huang & Darwiche, 2007]





Fixed variable order	→	OBDD
Dynamic variable order	→	FBDD
Dynamic variable order + Component analysis	→	Decision-DNNF

Ingredients of the New Algorithm

Ingredients of the New Algorithm

- SAT abstraction
- Component analysis
- Component caching

Ingredient 1
SAT Abstraction

- Clause learning
- Conflict-driven backtracking

Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

- Clause learning
- Conflict-driven backtracking

Ingredient 1
SAT Abstraction

L=1 **A**

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

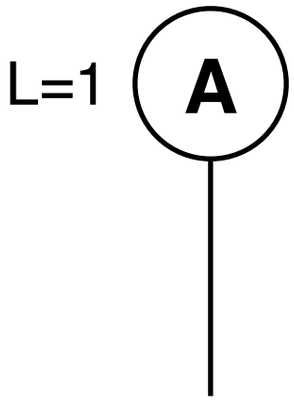
- Clause learning
- Conflict-driven backtracking

Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



1. -
2. {¬B, C}
3. {¬Z, ¬X}
4. {¬Z, X, Y}
5. {¬Z, X, ¬Y}

- Clause learning
- Conflict-driven backtracking

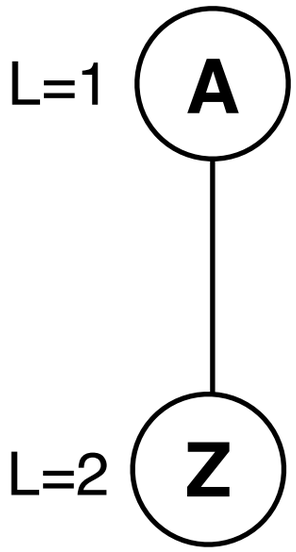
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

1. -
2. {¬B, C}
3. {¬Z, ¬X}
4. {¬Z, X, Y}
5. {¬Z, X, ¬Y}



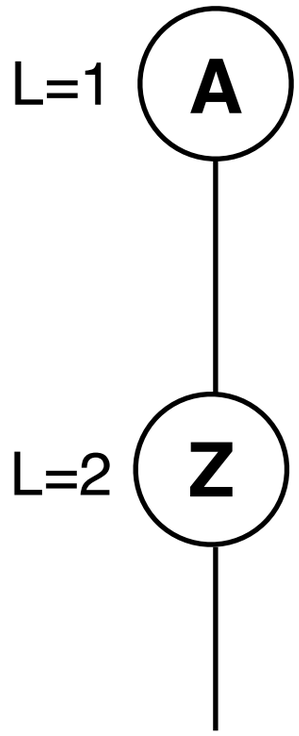
- Clause learning
- Conflict-driven backtracking

Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



1. -
2. {¬B, C}
3. {¬X}
4. {X, Y}
5. {X, ¬Y}

- Clause learning
- Conflict-driven backtracking

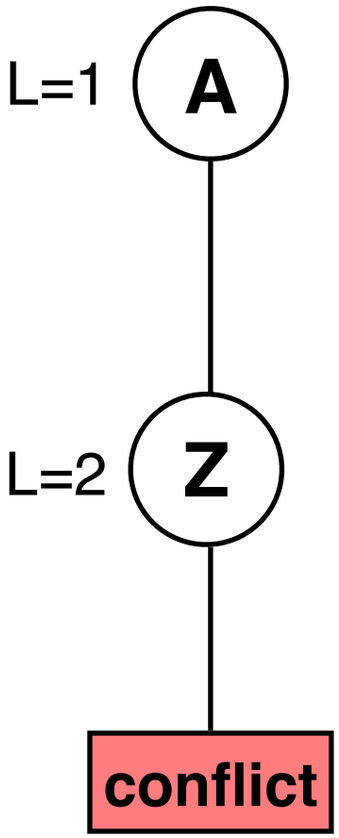
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

1. -
2. {¬B, C}
3. {¬X}
4. {X, Y}
5. {X, ¬Y}



- Clause learning
- Conflict-driven backtracking

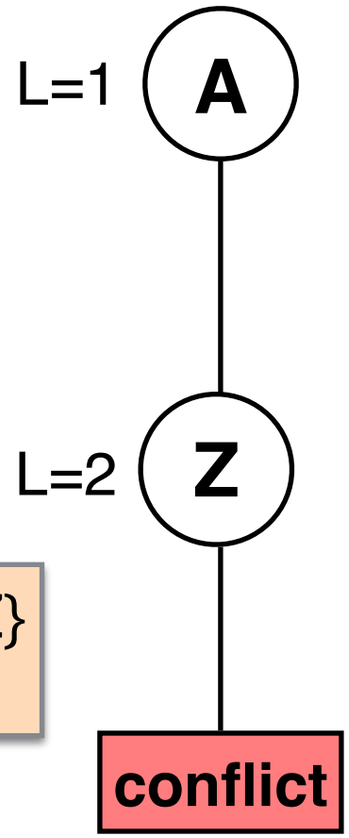
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

1. -
2. {¬B, C}
3. {¬X}
4. {X, Y}
5. {X, ¬Y}



Asserting clause: {¬A, ¬Z}
Assertion level: 1

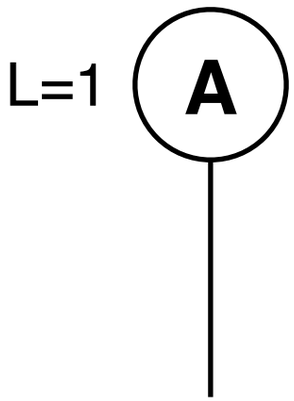
- Clause learning
- Conflict-driven backtracking

Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



Asserting clause: {¬A, ¬Z}
Assertion level: 1

1. -
2. {¬B, C}
3. {¬Z, ¬X}
4. {¬Z, X, Y}
5. {¬Z, X, ¬Y}

- Clause learning
- Conflict-driven backtracking

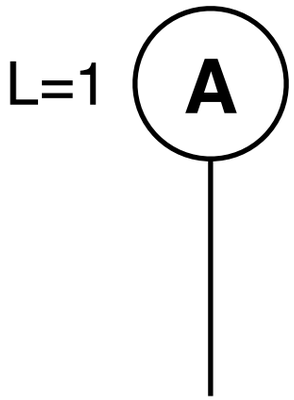
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}



1. -
2. {¬B, C}
3. {¬Z, ¬X}
4. {¬Z, X, Y}
5. {¬Z, X, ¬Y}

- Clause learning
- Conflict-driven backtracking

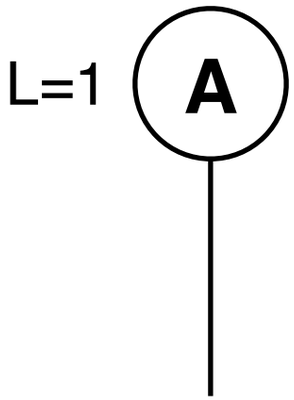
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}



1. -
2. {¬B, C}
3. {¬Z, ¬X}
4. {¬Z, X, Y}
5. {¬Z, X, ¬Y}

6. {¬Z}

- Clause learning
- Conflict-driven backtracking

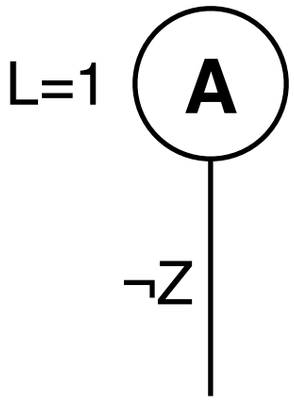
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}



1. -
2. {¬B, C}
3. -
4. -
5. -

6. -

- Clause learning
- Conflict-driven backtracking

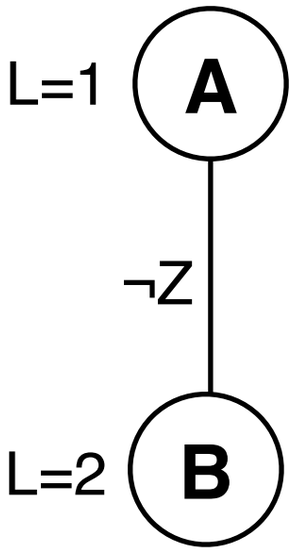
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}



1. -
2. {¬B, C}
3. -
4. -
5. -

6. -

- Clause learning
- Conflict-driven backtracking

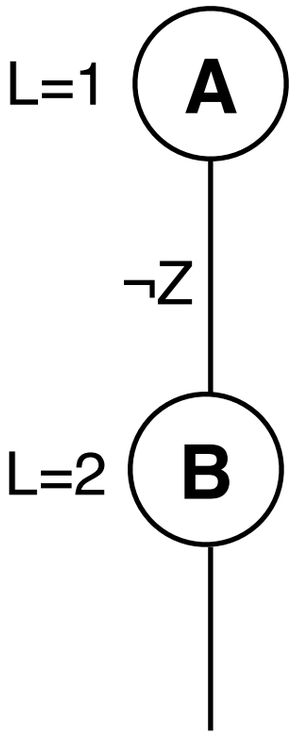
Ingredient 1
SAT Abstraction

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}



1. -
2. {C}
3. -
4. -
5. -

6. -

- Clause learning
- Conflict-driven backtracking

Ingredient 1
SAT Abstraction

Δ =

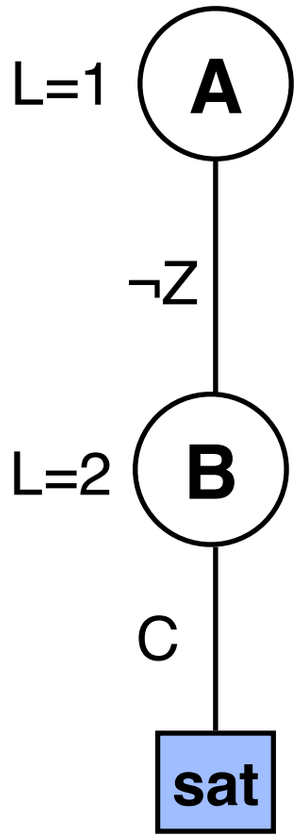
1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}

1. -
2. -
3. -
4. -
5. -

6. -



- Clause learning
- Conflict-driven backtracking

Ingredient 1
SAT Abstraction

SAT State: (Δ, Γ, D, I)

Δ	Input CNF
Γ	Learned clauses
D	Decision sequence
I	Implied literals

Macro: *decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$
$D \leftarrow D; \ell$ `// add a new decision to` D
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

Macro: *undo_decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$
erase the last decision ℓ from D
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$

Macro: *at_assertion_level*$(\alpha, S = (\Delta, \Gamma, D, I))$
$m \leftarrow$ assertion level of α
if *there are* m *literals in* D **then return** true
else return false

Macro: *assert_clause*$(\alpha, S = (\Delta, \Gamma, D, I))$
$\Gamma \leftarrow \Gamma \cup \{\alpha\}$ `// add learned clause to` Γ
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

SAT Solver using the Primitives

SAT(S)

Input: S : a SAT state (Δ, Γ, D, I)
Output: *success* or $\bot$

find a literal ℓ where neither ℓ nor $\neg\ell$ are implied in S
if *there is no such literal* ℓ **then return** *success*
$ret \leftarrow$ *decide_literal*(ℓ, S)
if ret *is a success* **then** $ret \leftarrow$ SAT(S)
undo_decide_literal(ℓ, S)
if ret *is a learned clause* **then**
  if *at_assertion_level*(ret, S) **then**
    $ret \leftarrow$ *assert_clause*(ret, S)
    if ret *is a success* **then return** SAT(S)
    else return ret
  else return ret
else return *success*

Ingredient 2
Component Analysis

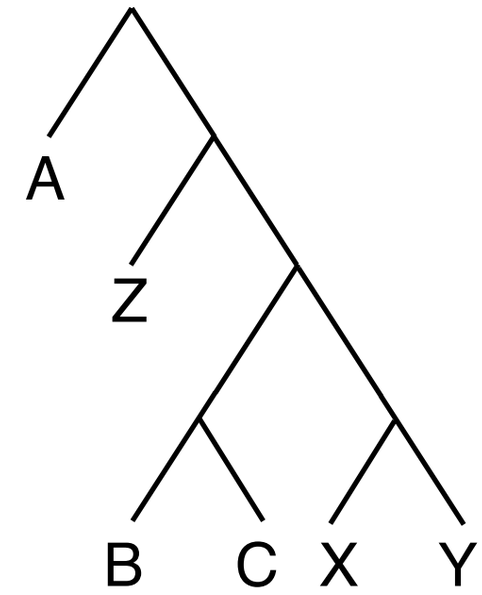
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}

Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}

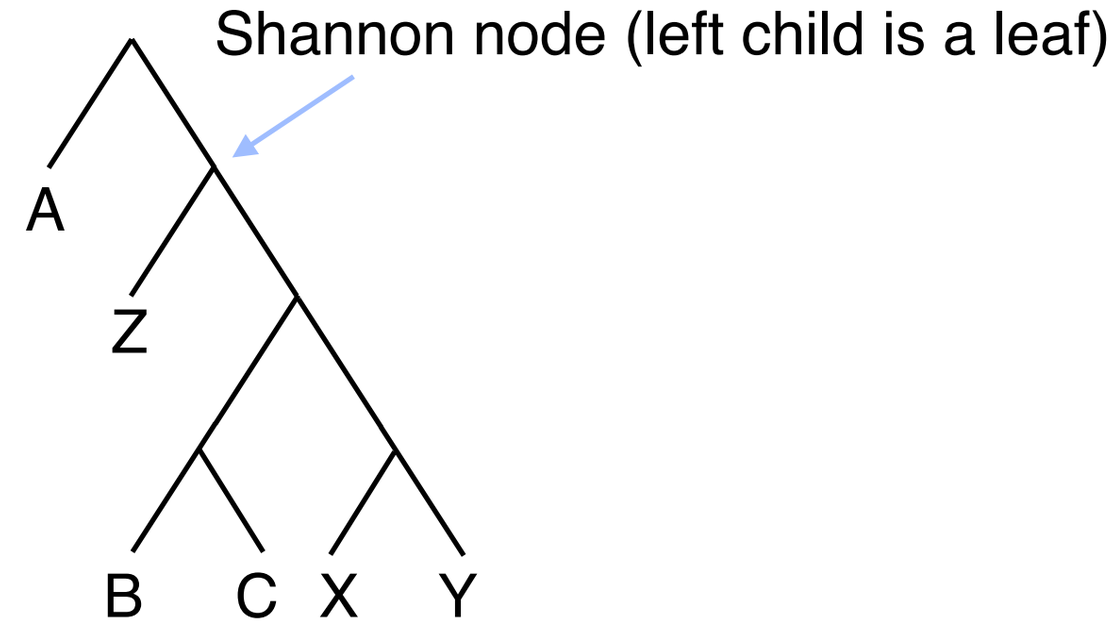


Decision Vtree [Oztok & Darwiche, 2014]

Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}

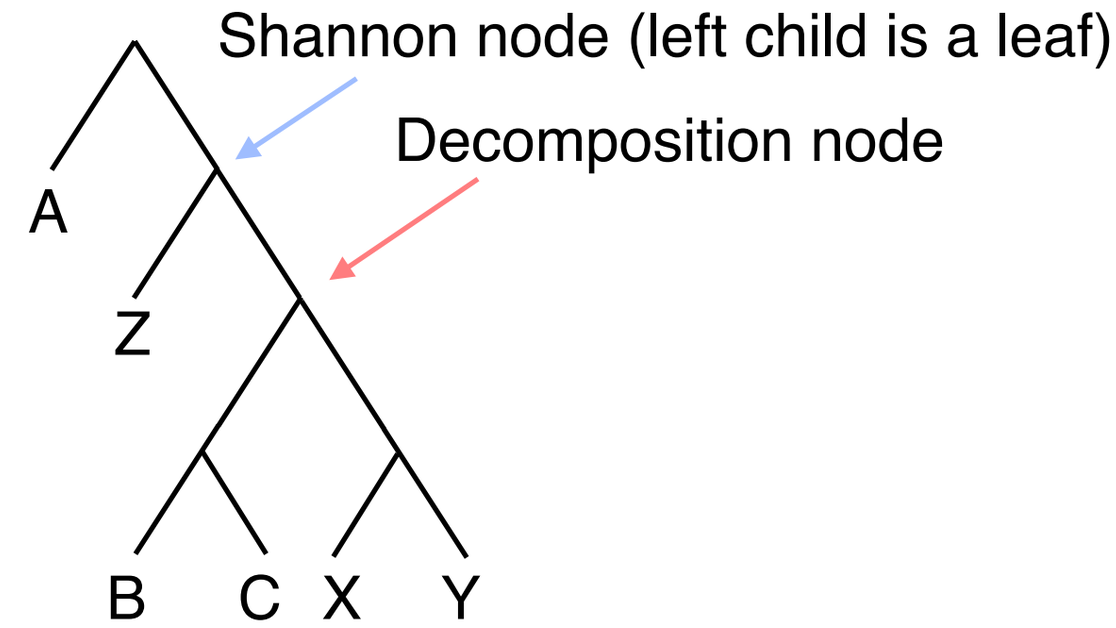


Decision Vtree [Oztok & Darwiche, 2014]

Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}

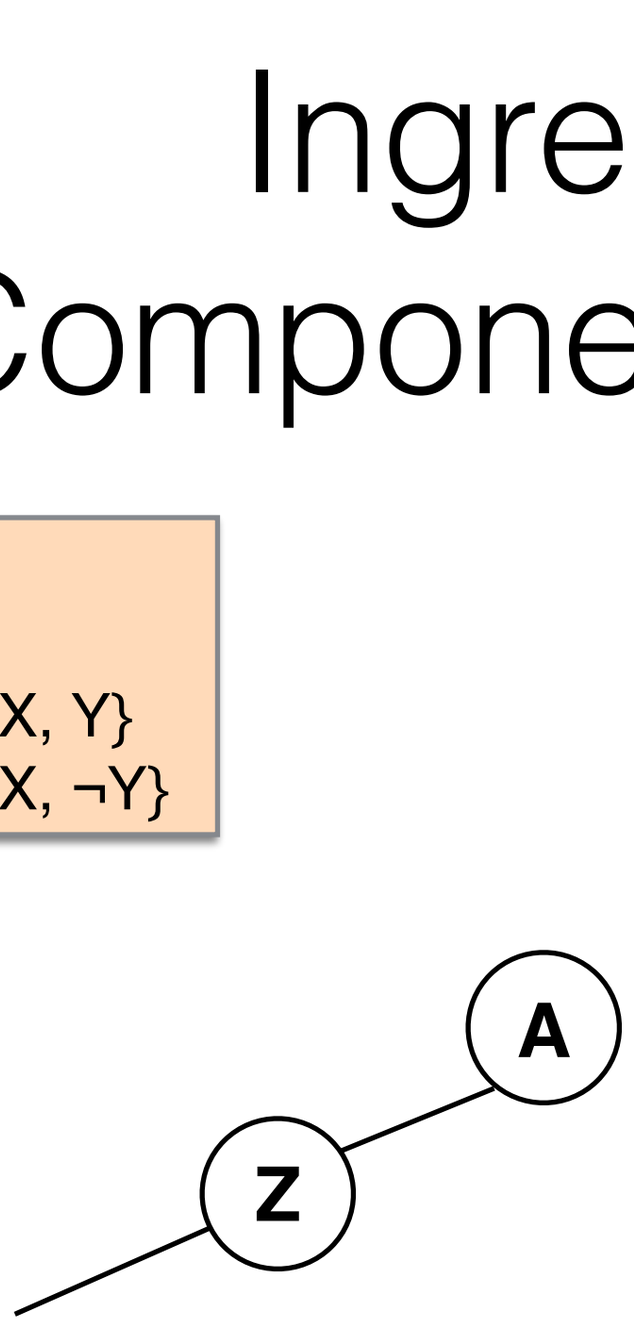


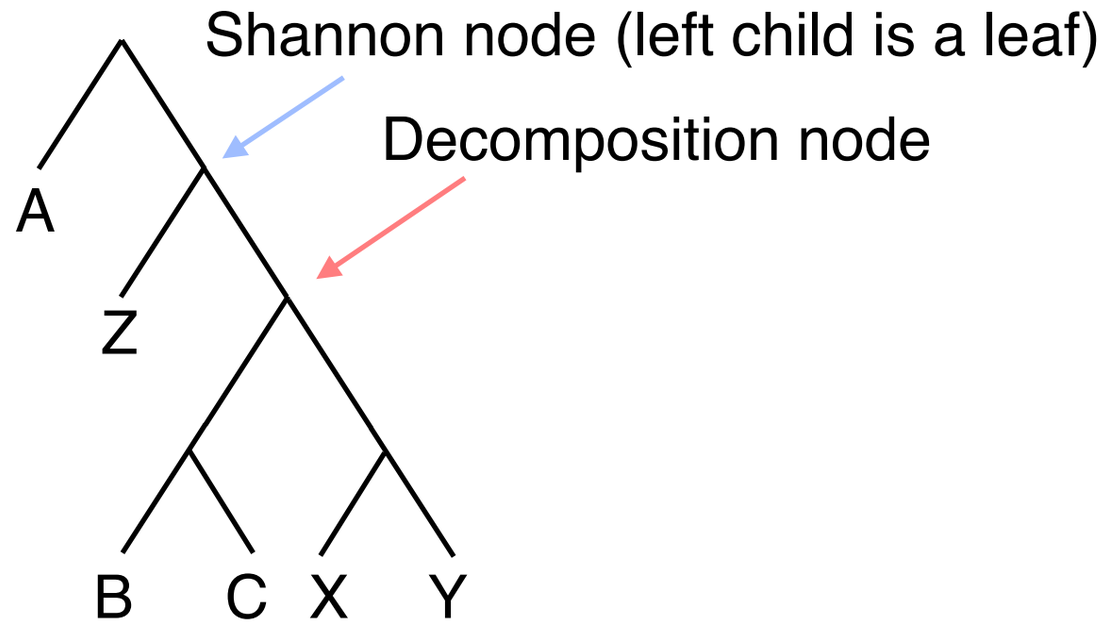
Decision Vtree [Oztok & Darwiche, 2014]

Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}



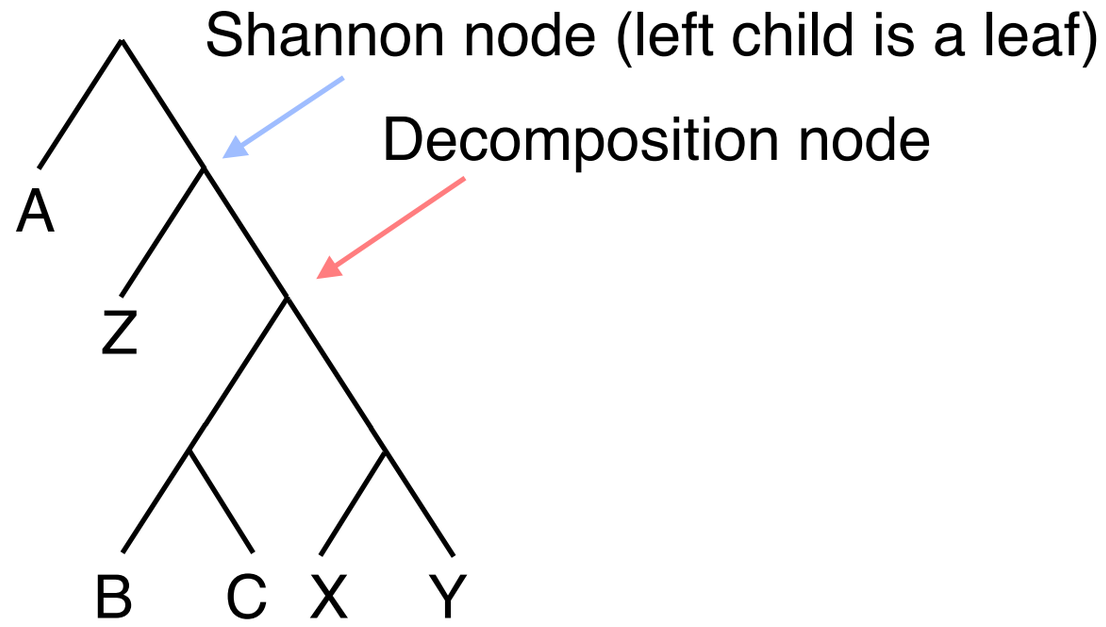


Decision Vtree [Oztok & Darwiche, 2014]

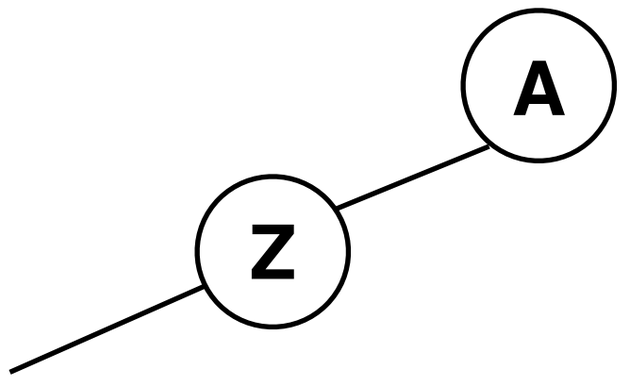
Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}



Decision Vtree [Oztok & Darwiche, 2014]



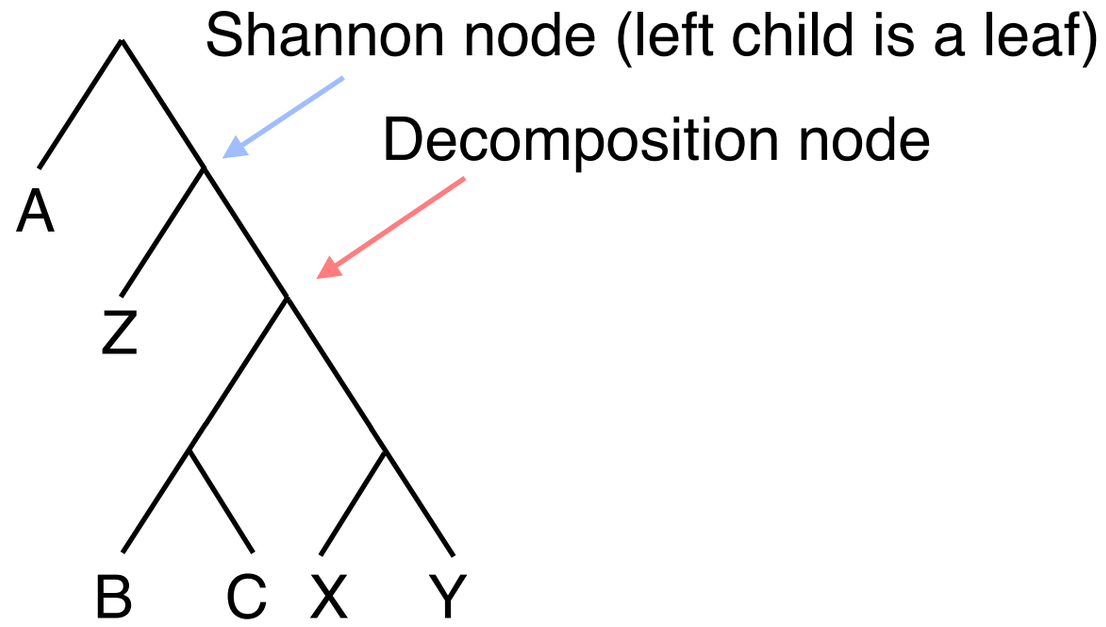
{¬B, C}

{X, Y}, {X, ¬Y}

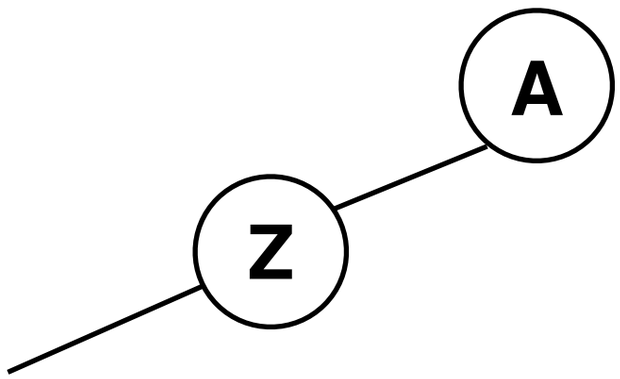
Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}



Decision Vtree [Oztok & Darwiche, 2014]



{¬B, C}

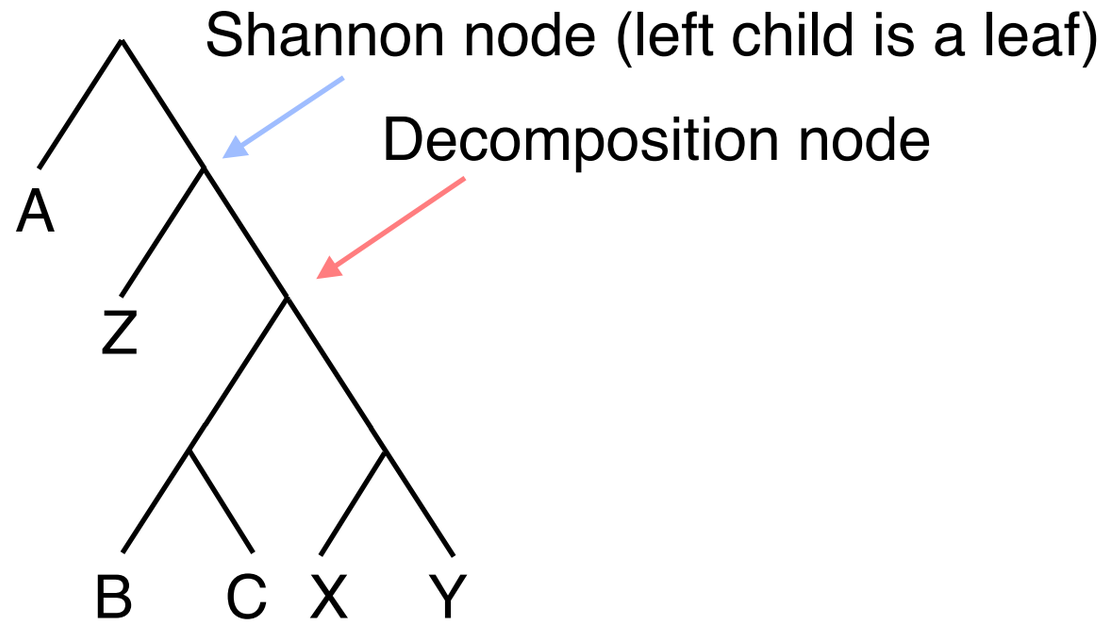
{X, Y}, {X, ¬Y}

3

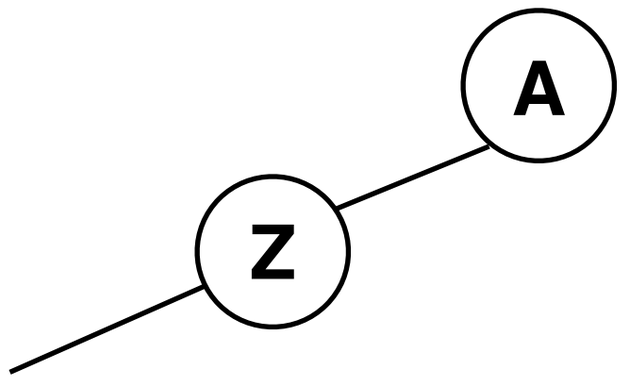
Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}



Decision Vtree [Oztok & Darwiche, 2014]



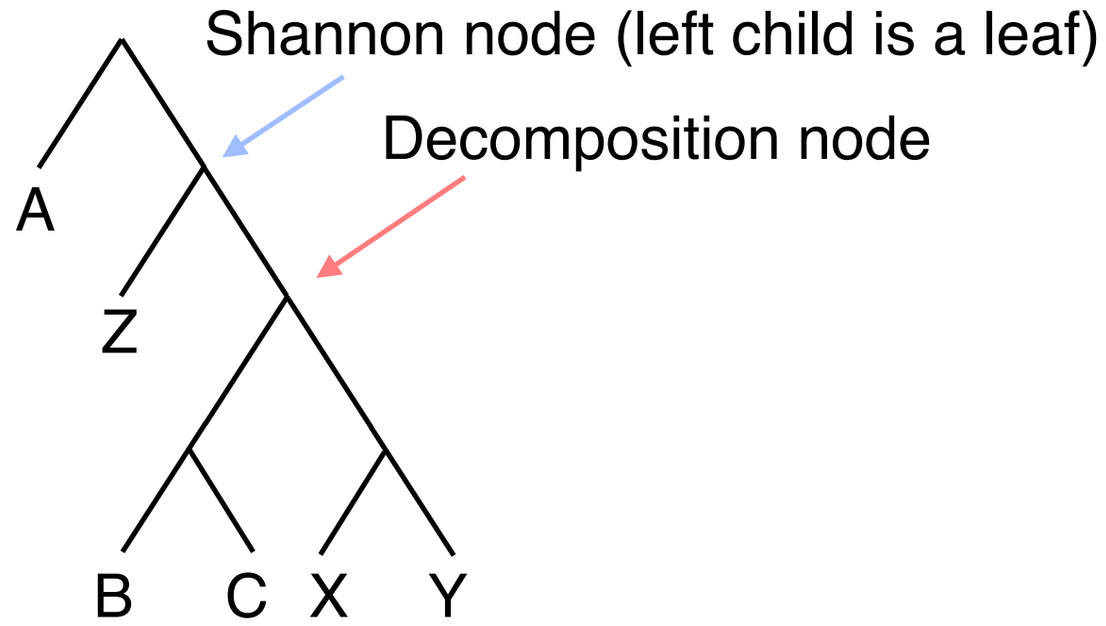
{¬B, C} {X, Y}, {X, ¬Y}

3 2

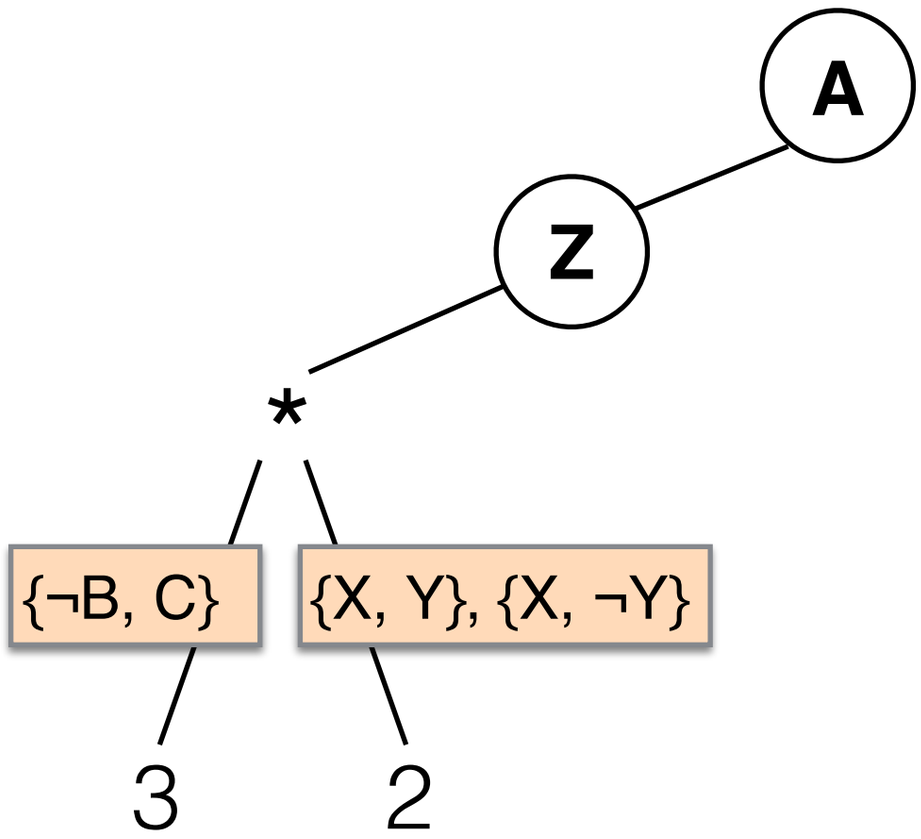
Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}



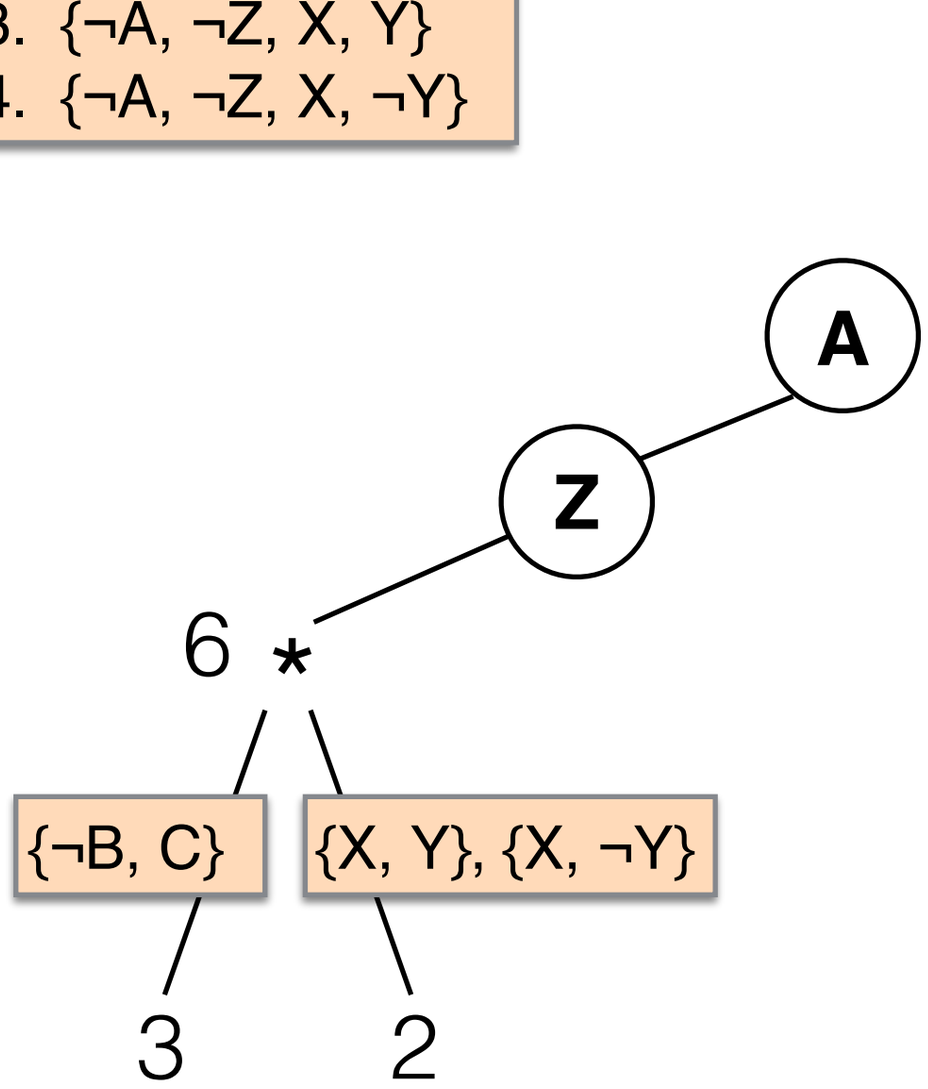
Decision Vtree [Oztok & Darwiche, 2014]

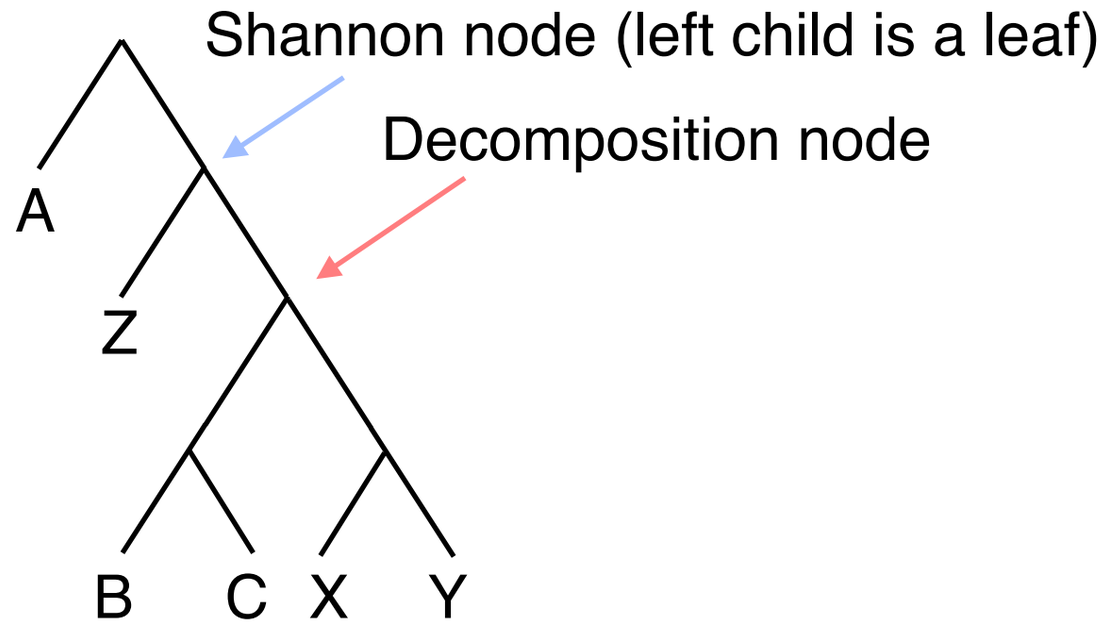


Ingredient 2
Component Analysis

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, X, Y}
4. {¬A, ¬Z, X, ¬Y}



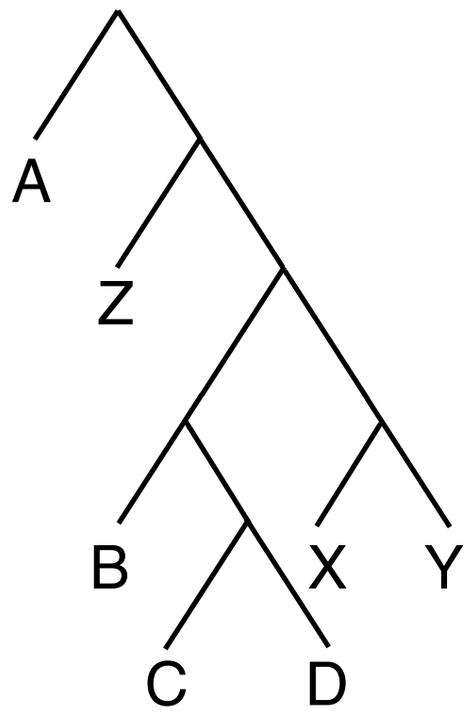


Decision Vtree [Oztok & Darwiche, 2014]

Ingredient 3
Component Caching

$\Delta =$

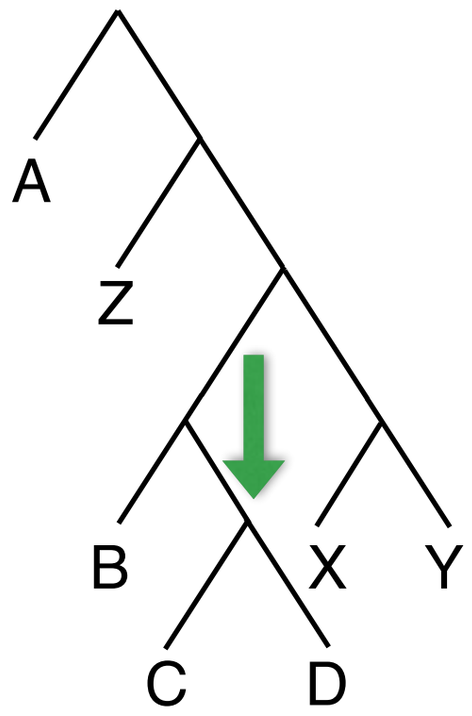
1. {A, B}
2. {¬B, C,D}
3. {C, ¬D}
4. {¬A, ¬Z, ¬X}
5. {¬A, ¬Z, X, Y}
6. {¬A, ¬Z, X, ¬Y}



Ingredient 3
Component Caching

$\Delta =$

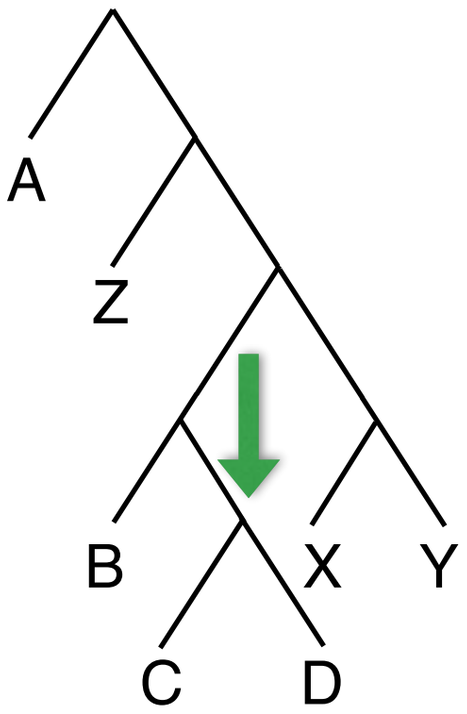
1. {A, B}
2. {¬B, C,D}
3. {C, ¬D}
4. {¬A, ¬Z, ¬X}
5. {¬A, ¬Z, X, Y}
6. {¬A, ¬Z, X, ¬Y}



Ingredient 3
Component Caching

$\Delta =$

1. {A, B}
2. {¬B, C,D}
3. {C, ¬D}
4. {¬A, ¬Z, ¬X}
5. {¬A, ¬Z, X, Y}
6. {¬A, ¬Z, X, ¬Y}

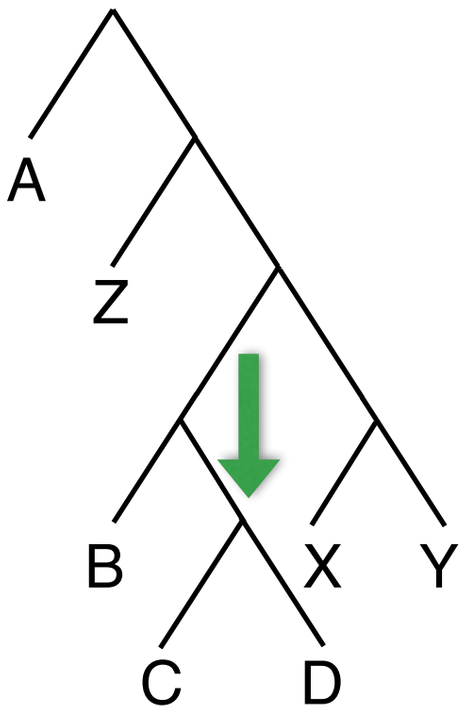


A Z B	Δ'
0 0 0	contradiction
0 0 1	{C,D}, {C,¬D}
0 1 0	contradiction
0 1 1	{C,D}, {C,¬D}
1 0 0	{C,¬D}
1 0 1	{C,D}, {C,¬D}
1 1 0	{C,¬D}
1 1 1	{C,D}, {C,¬D}

Ingredient 3
Component Caching

$\Delta =$

1. {A, B}
2. {¬B, C,D}
3. {C, ¬D}
4. {¬A, ¬Z, ¬X}
5. {¬A, ¬Z, X, Y}
6. {¬A, ¬Z, X, ¬Y}



A Z B	Δ'
0 0 0	contradiction
0 0 1	{C,D}, {C,¬D}
0 1 0	contradiction
0 1 1	{C,D}, {C,¬D}
1 0 0	{C,¬D}
1 0 1	{C,D}, {C,¬D}
1 1 0	{C,¬D}
1 1 1	{C,D}, {C,¬D}

Cache key:

- Bit vector of clauses/variables
  - 1 bit per clause to check clause subsumption
  - 2 bits per variable to capture the state of variable

The New Algorithm

The New Algorithm

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

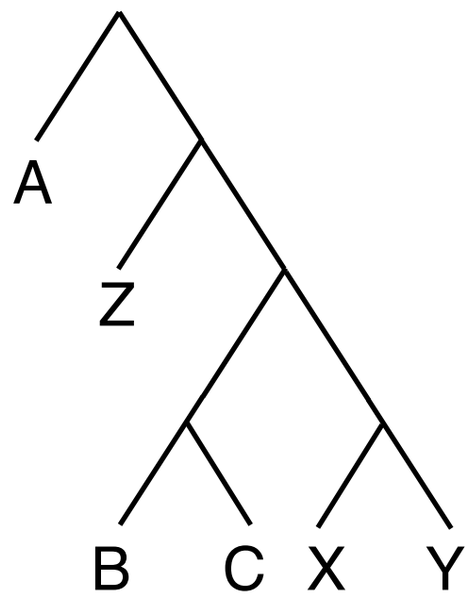
$\Gamma =$

The New Algorithm

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

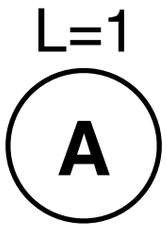


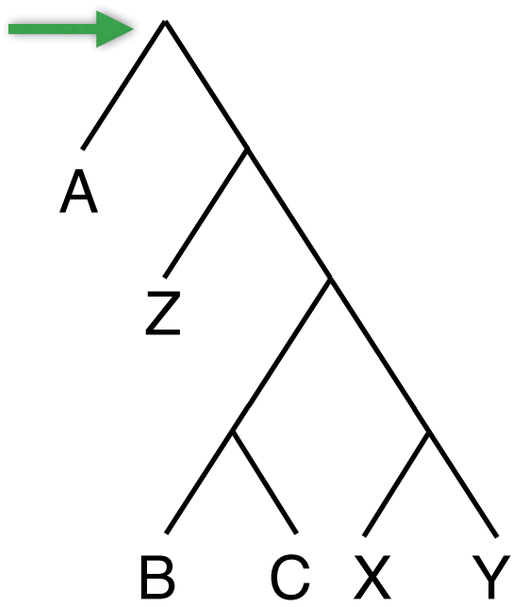
The New Algorithm

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



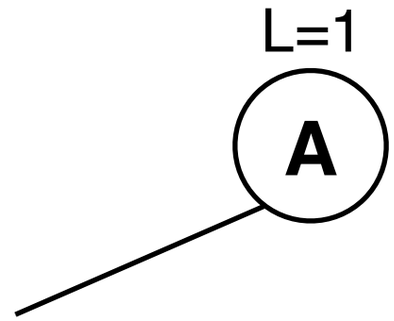


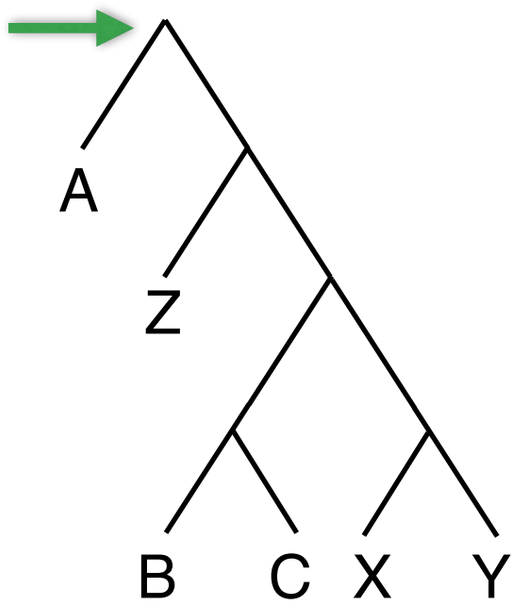
The New Algorithm

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



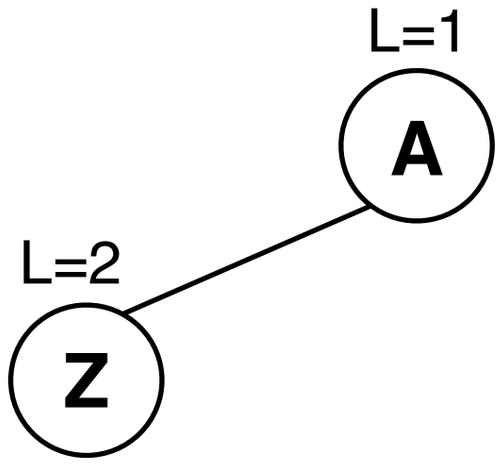


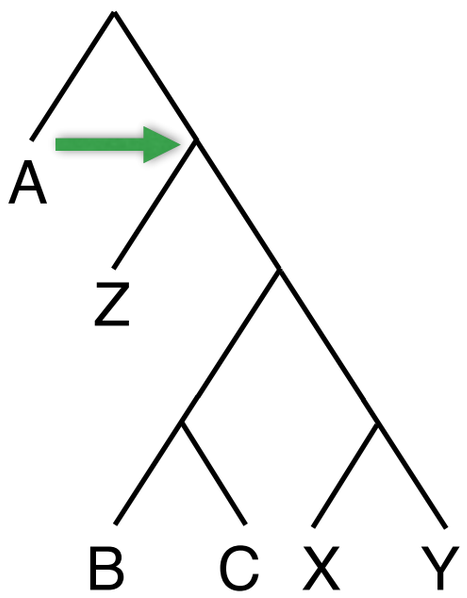
The New Algorithm

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$



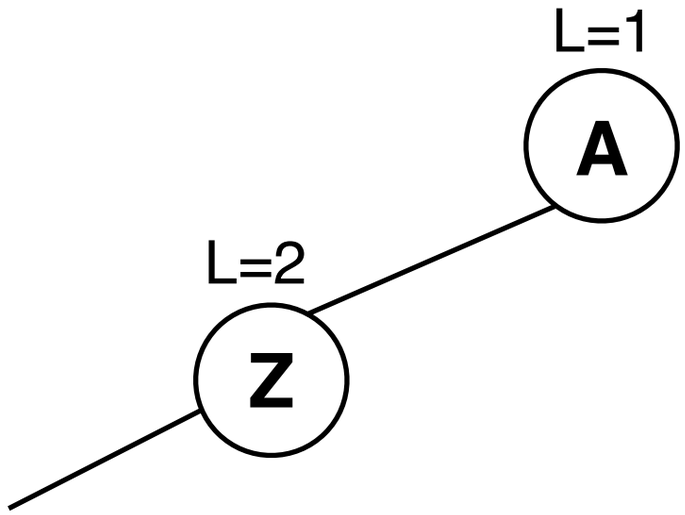


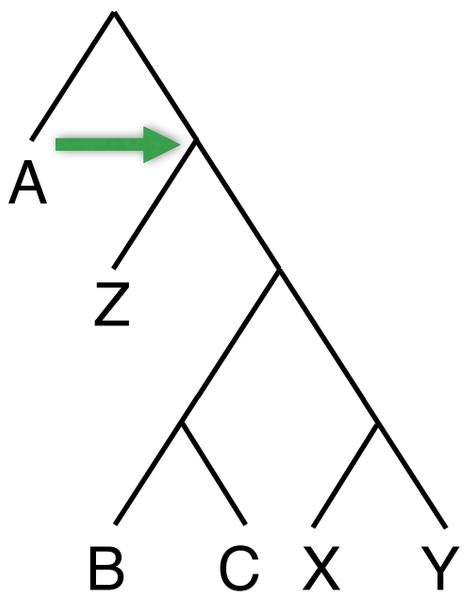
The New Algorithm

$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$



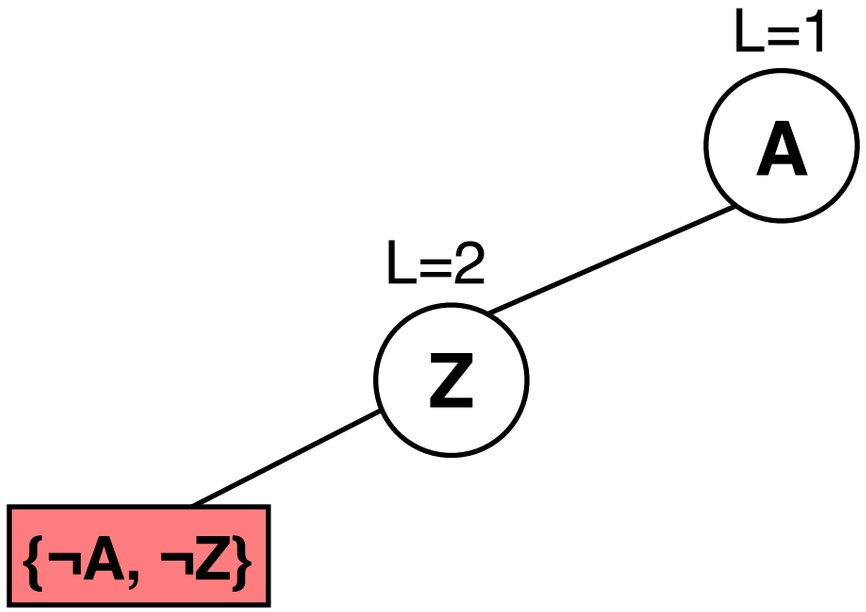


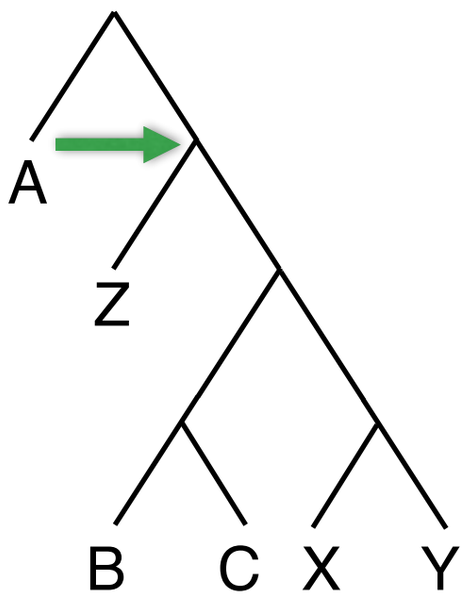
The New Algorithm

Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



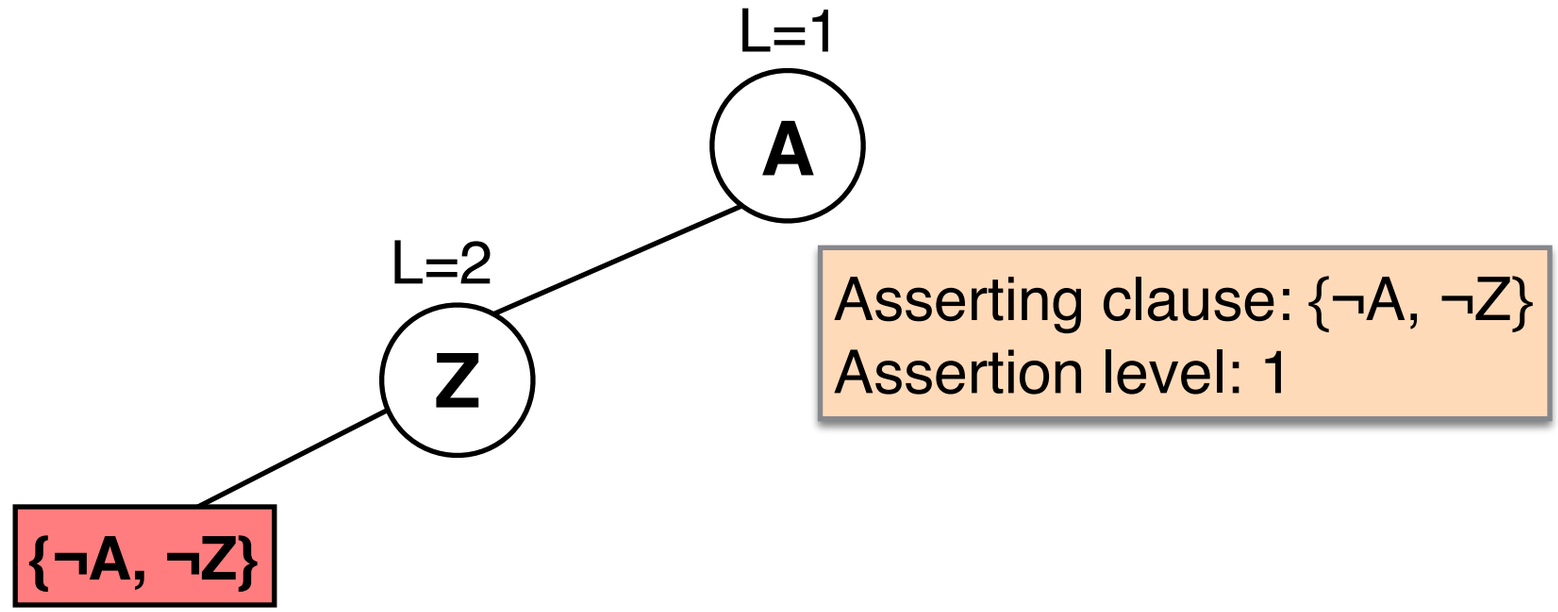


The New Algorithm

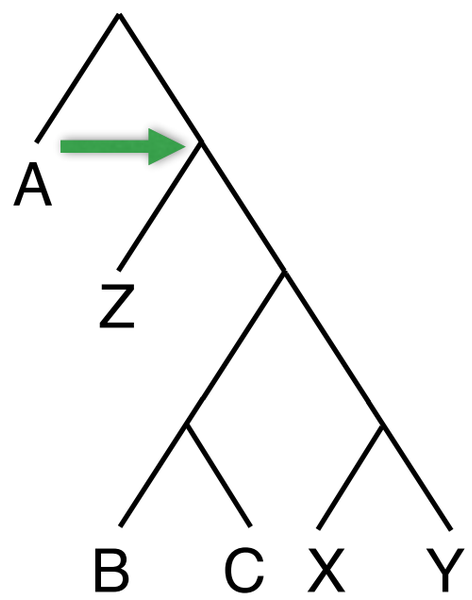
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



Asserting clause: {¬A, ¬Z}
Assertion level: 1

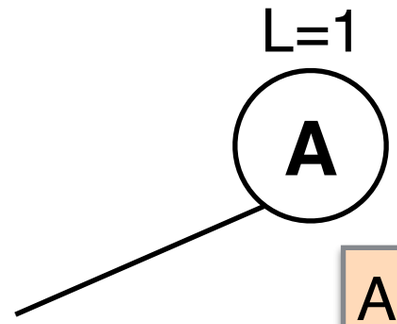


The New Algorithm

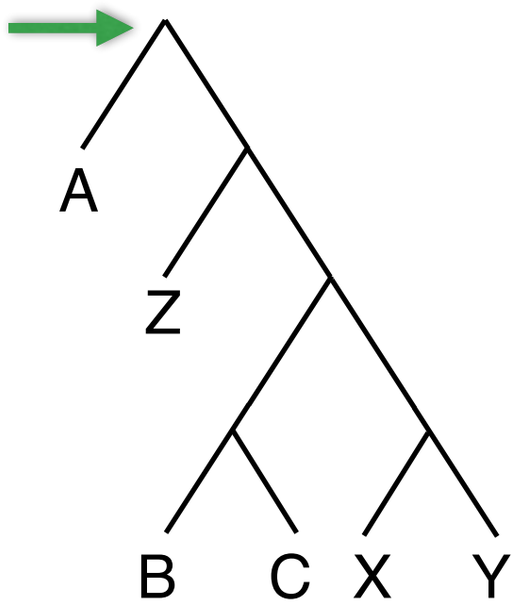
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =



Asserting clause: {¬A, ¬Z}
Assertion level: 1



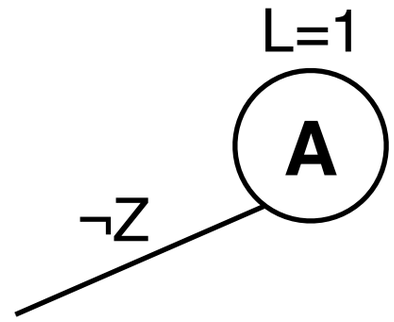
The New Algorithm

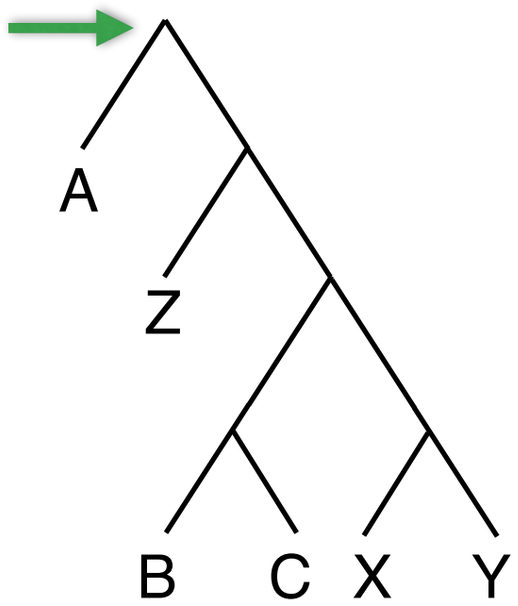
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}





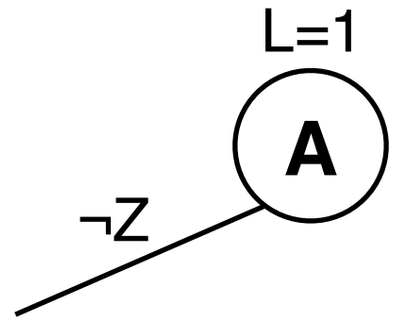
The New Algorithm

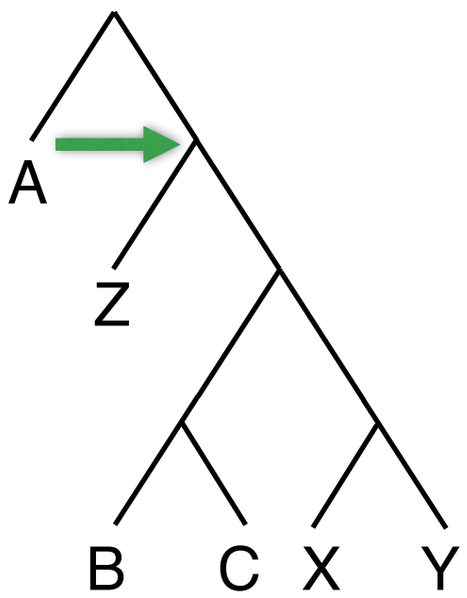
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}





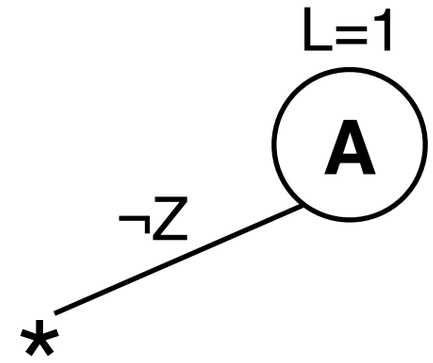
The New Algorithm

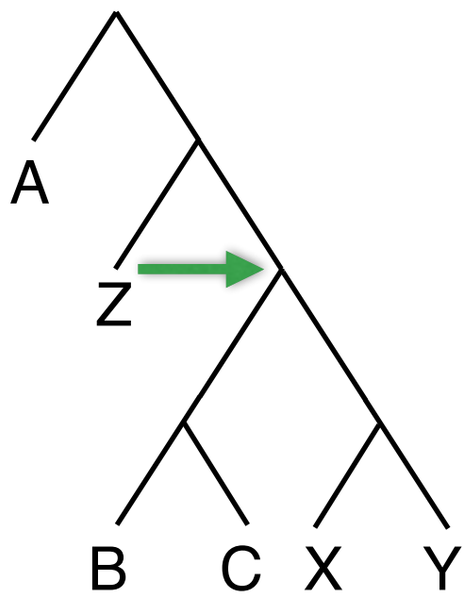
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}





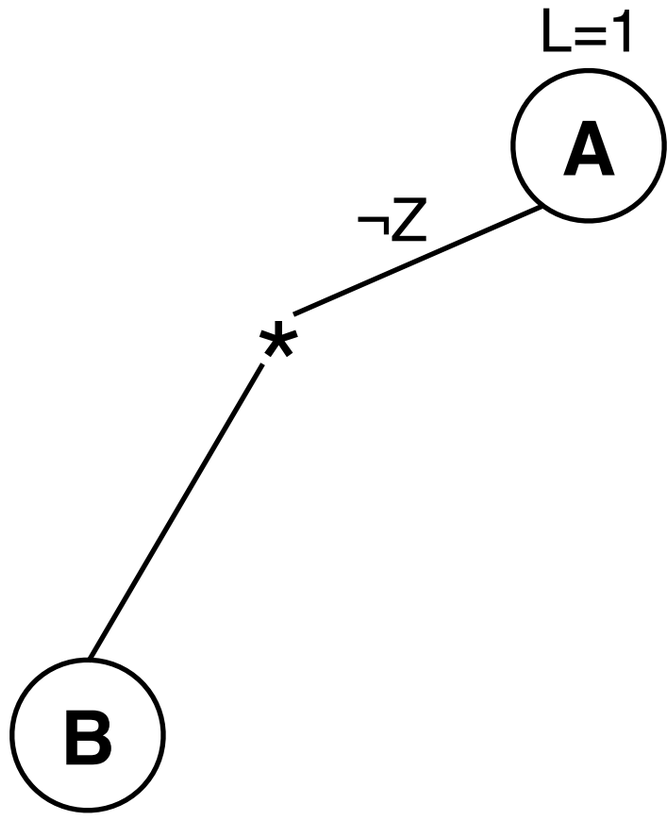
The New Algorithm

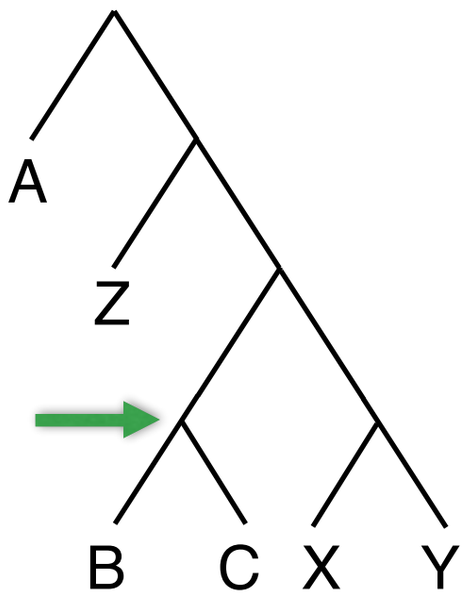
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}





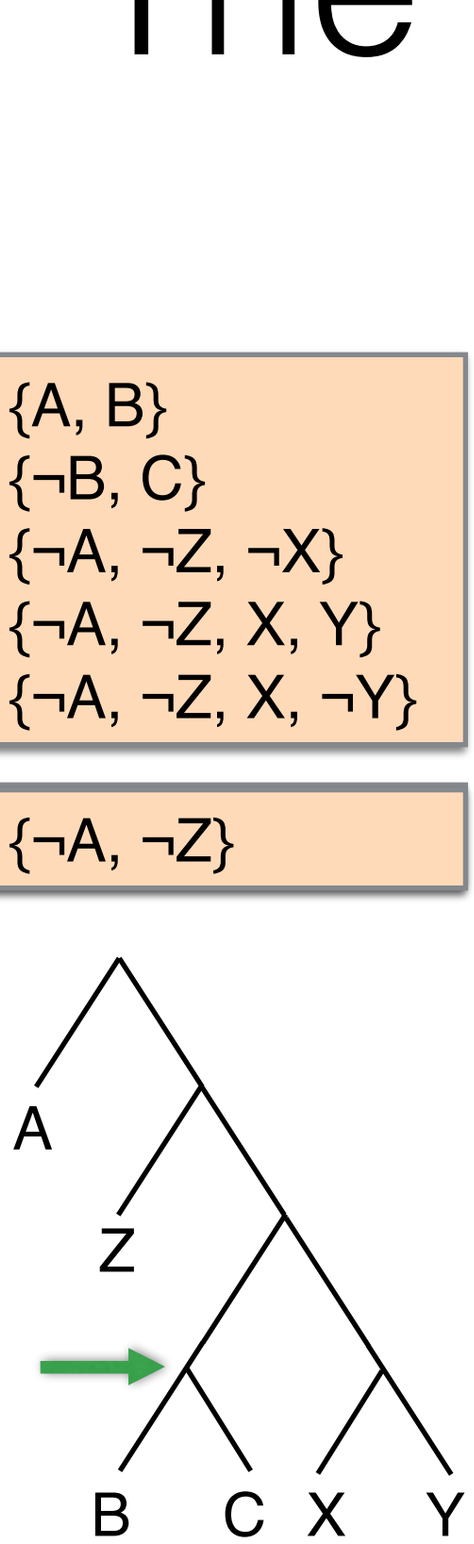
The New Algorithm

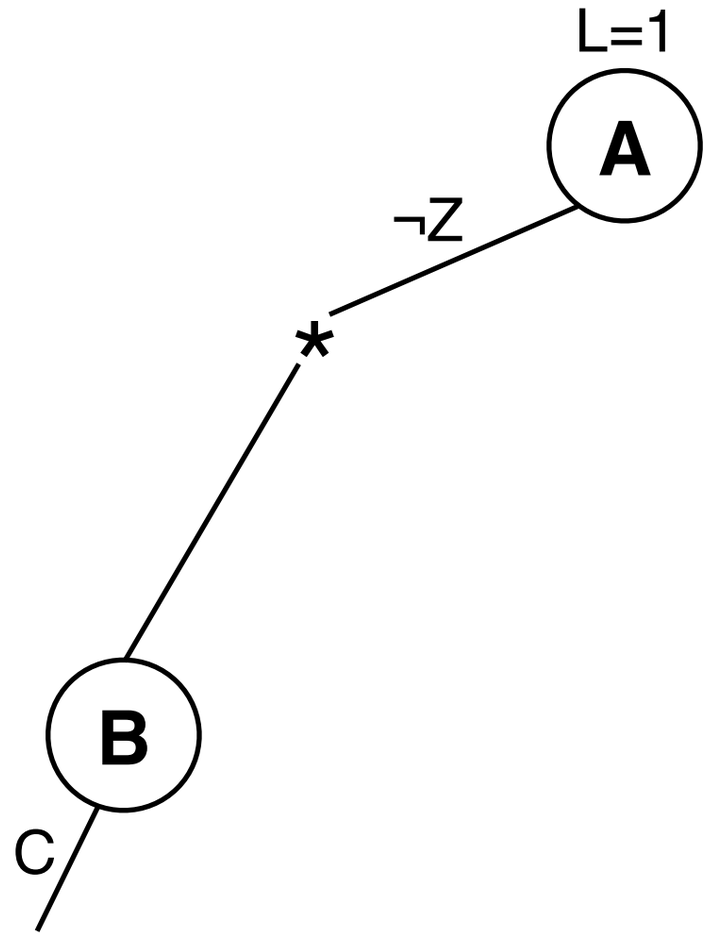
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





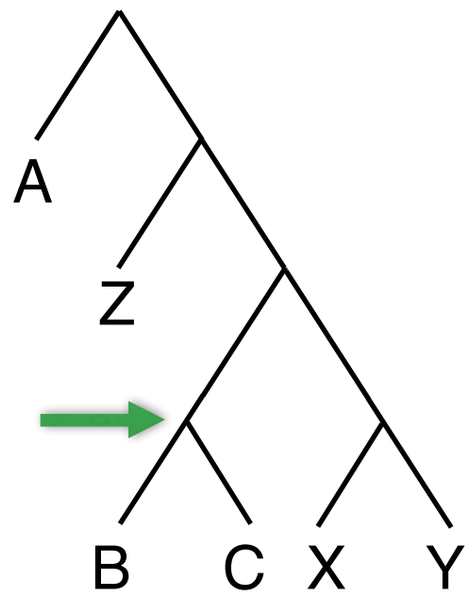
The New Algorithm

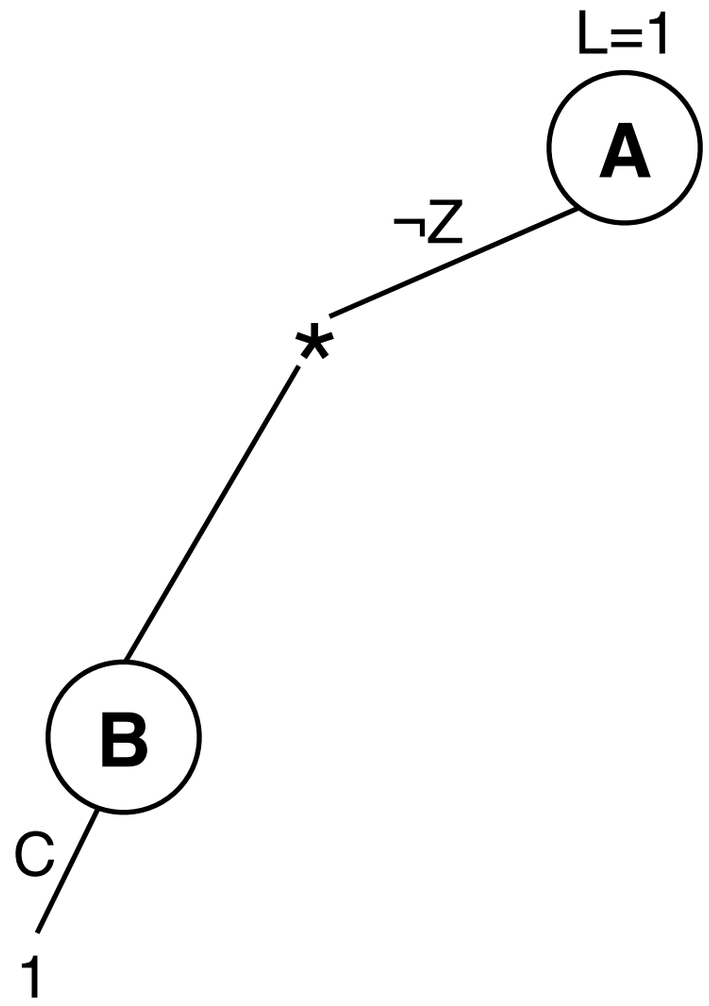
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





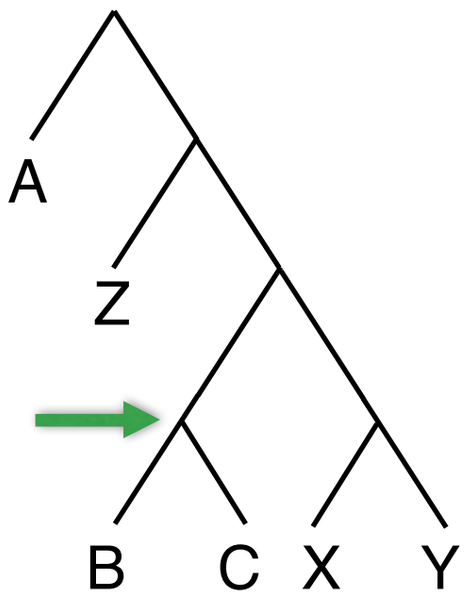
The New Algorithm

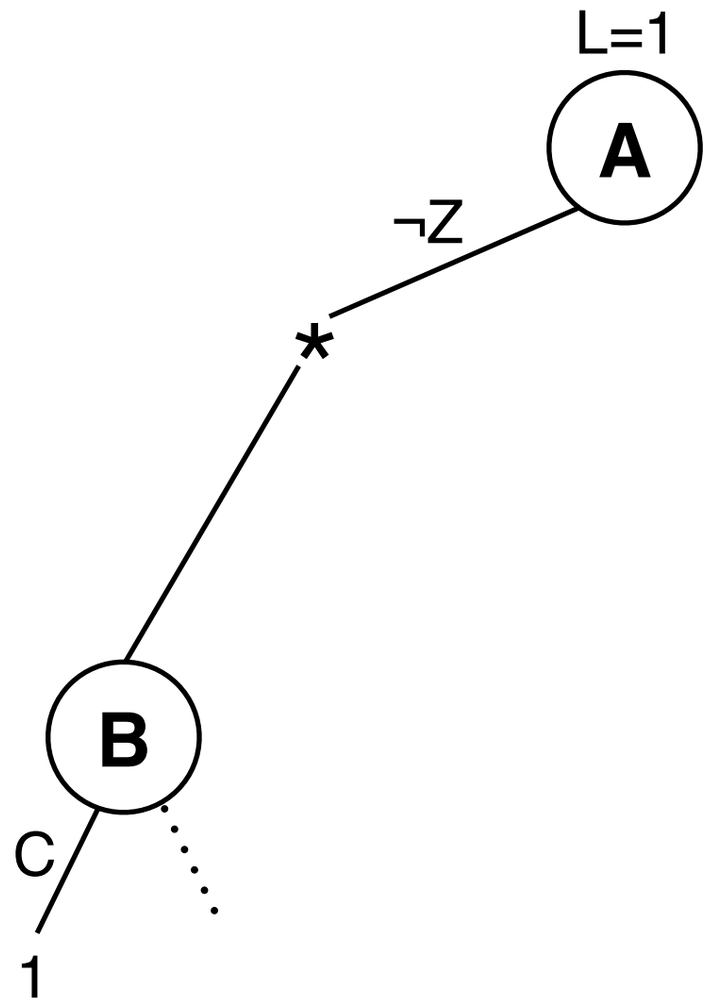
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





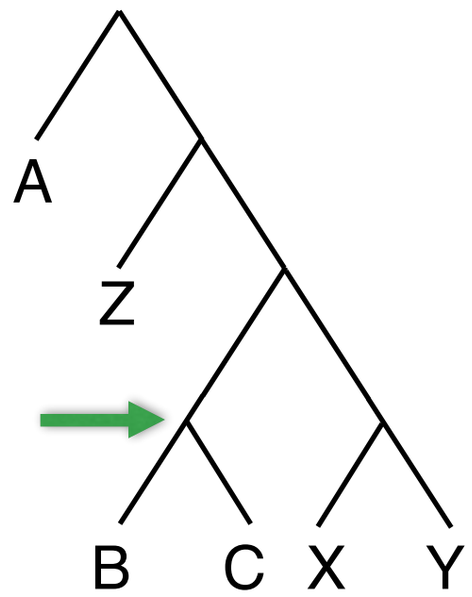
The New Algorithm

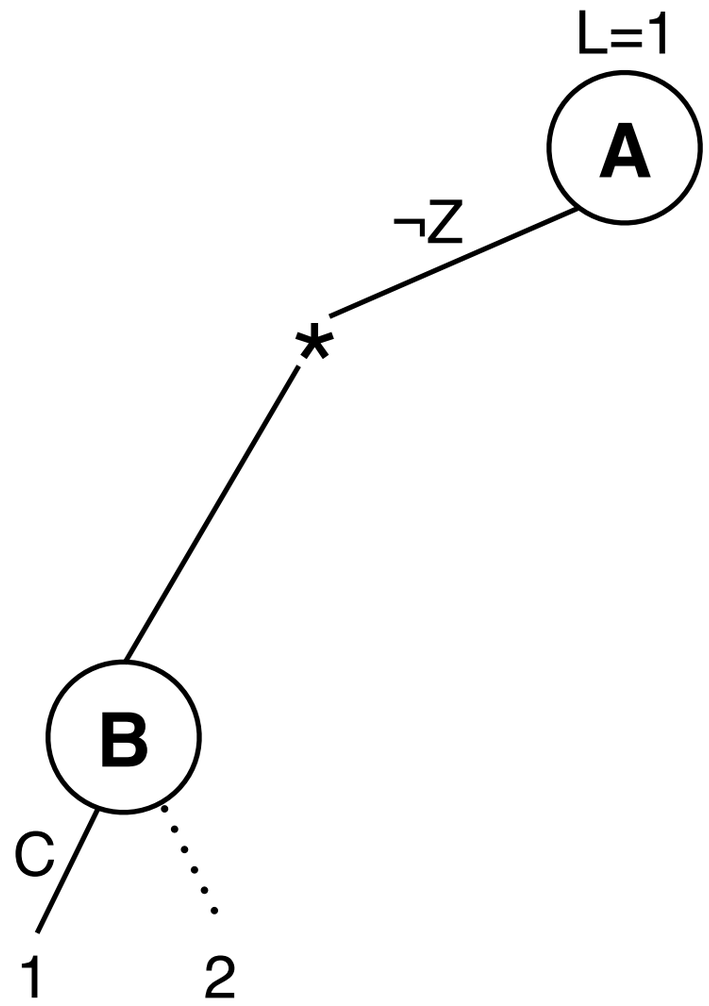
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





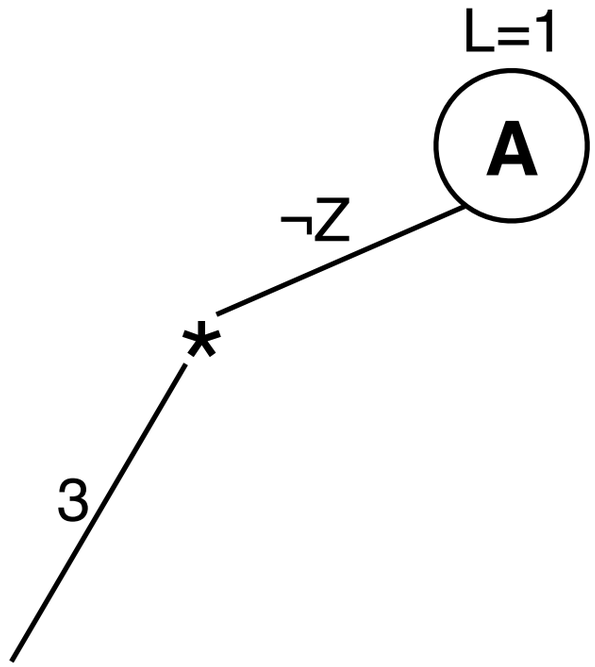
The New Algorithm

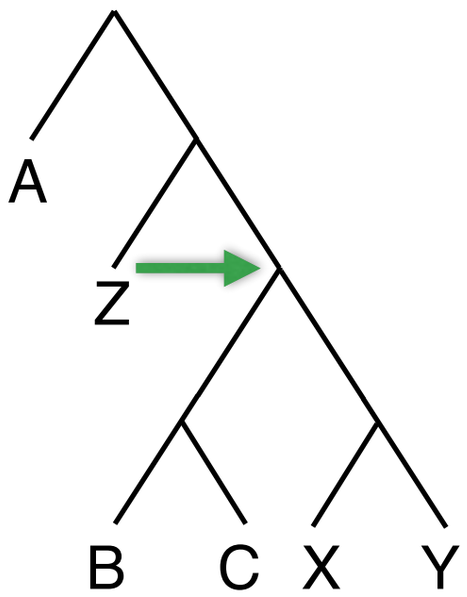
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}





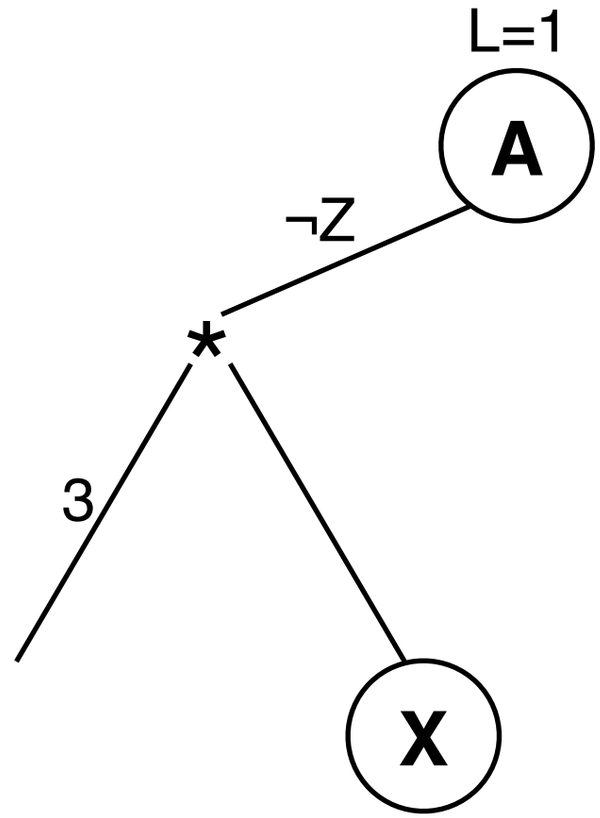
The New Algorithm

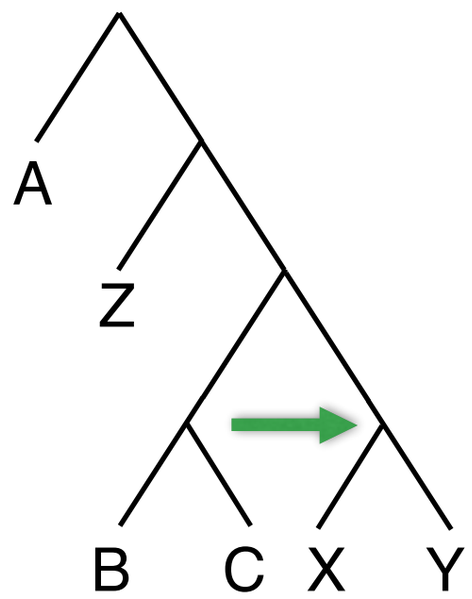
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}





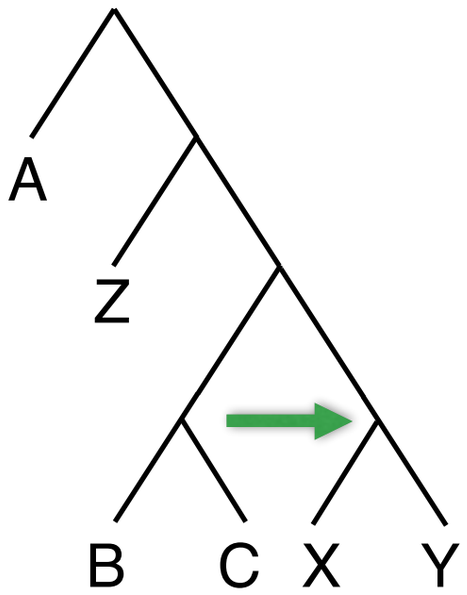
The New Algorithm

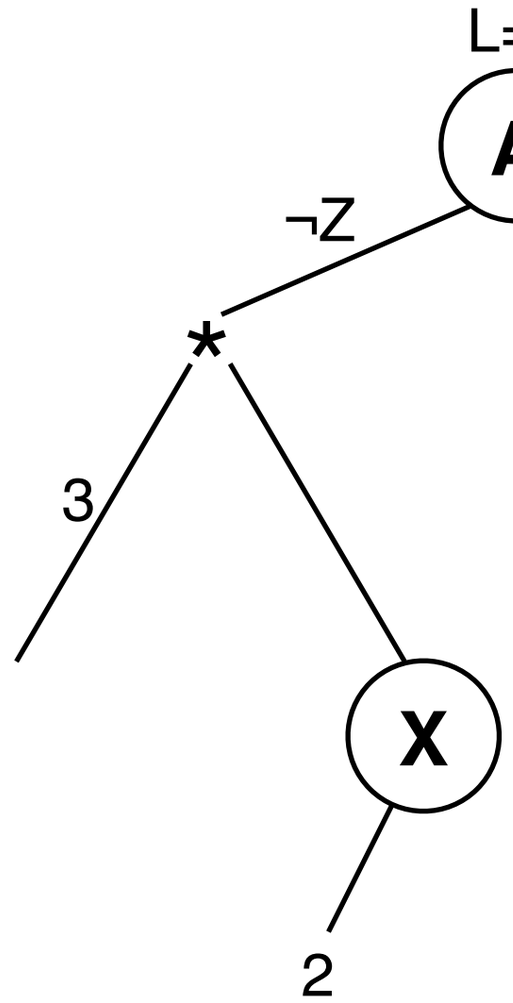
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





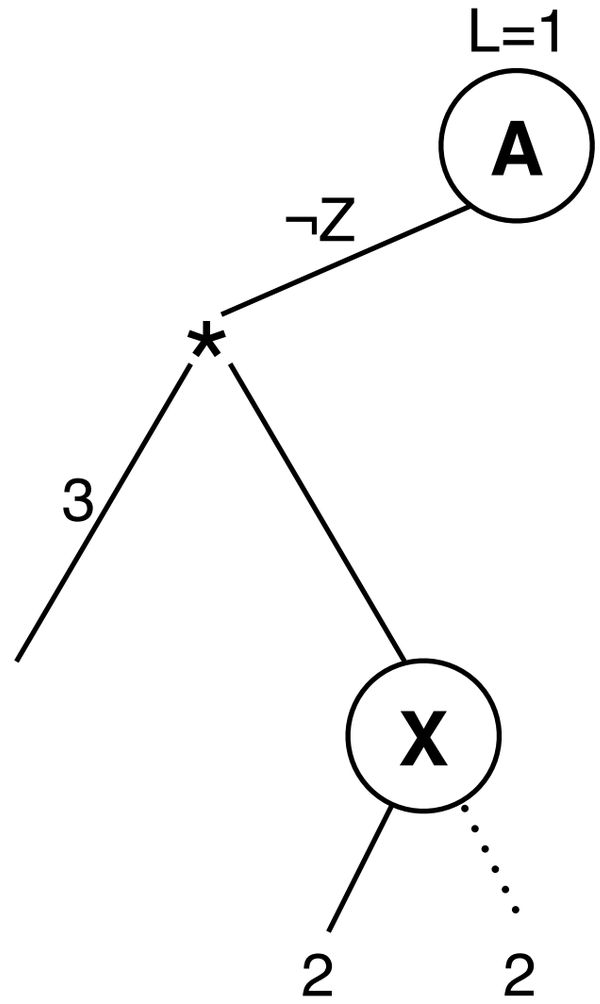
The New Algorithm

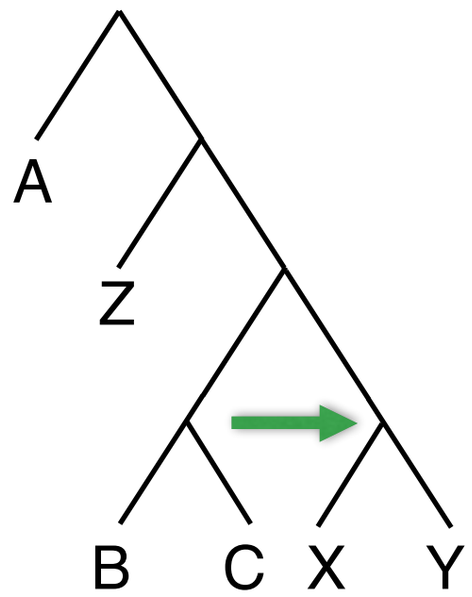
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





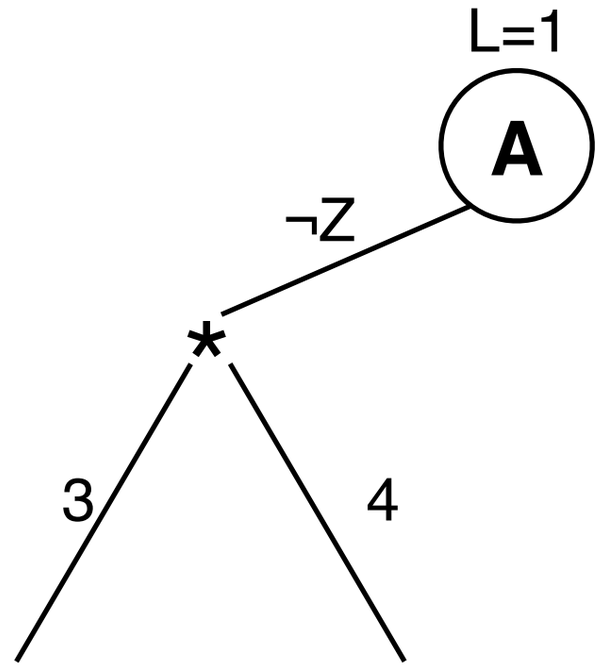
The New Algorithm

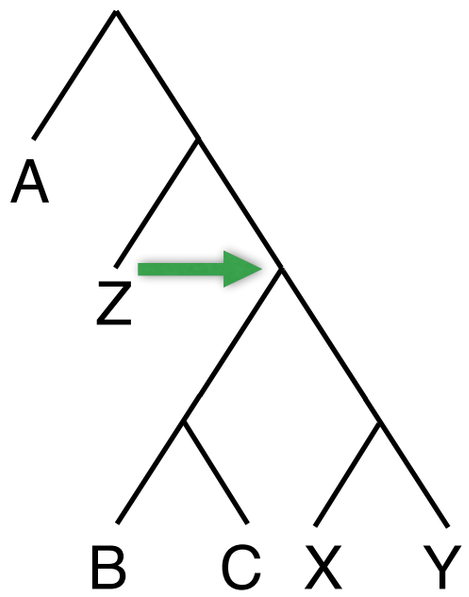
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





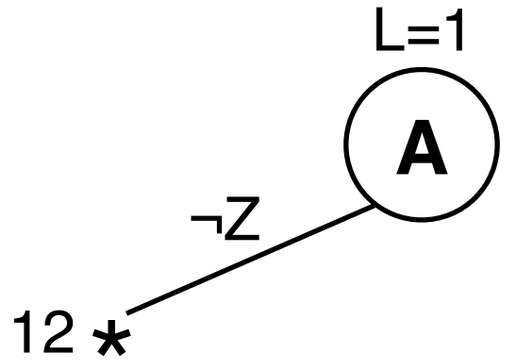
The New Algorithm

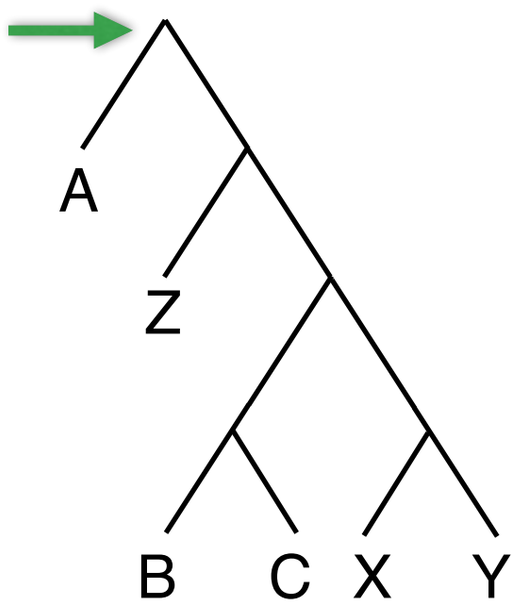
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

6. {¬A, ¬Z}





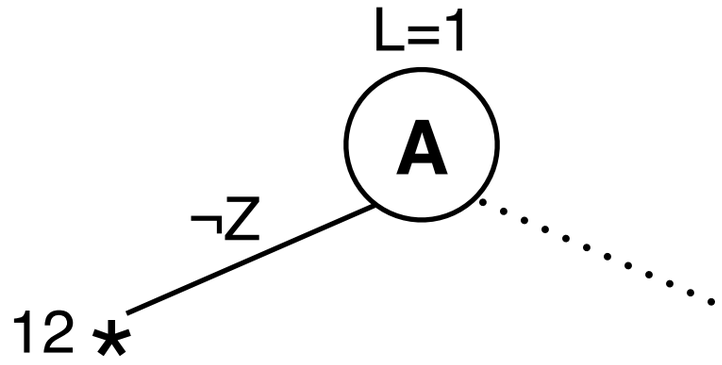
The New Algorithm

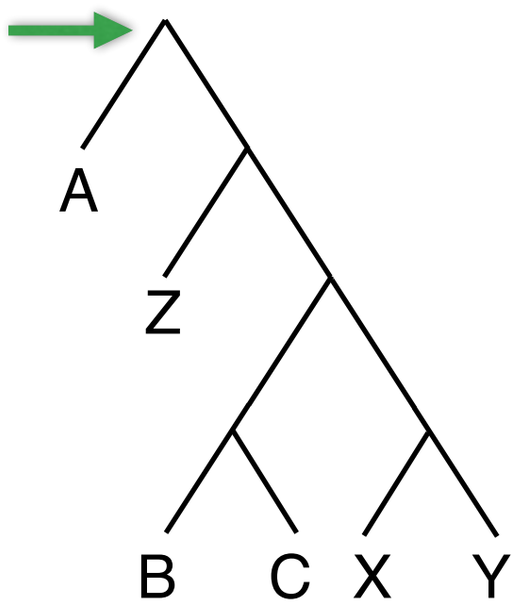
$\Delta =$

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

$\Gamma =$

6. {¬A, ¬Z}





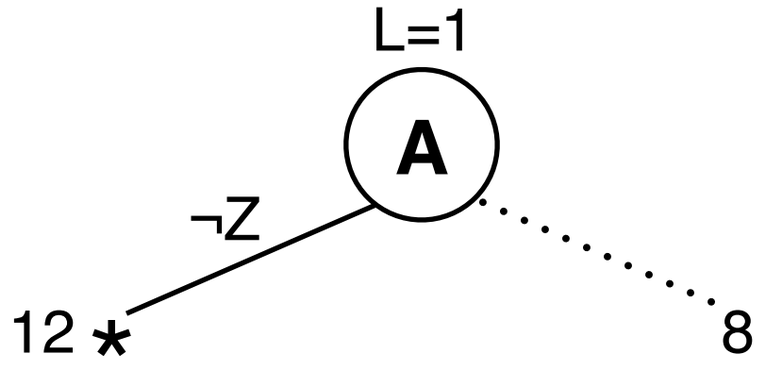
The New Algorithm

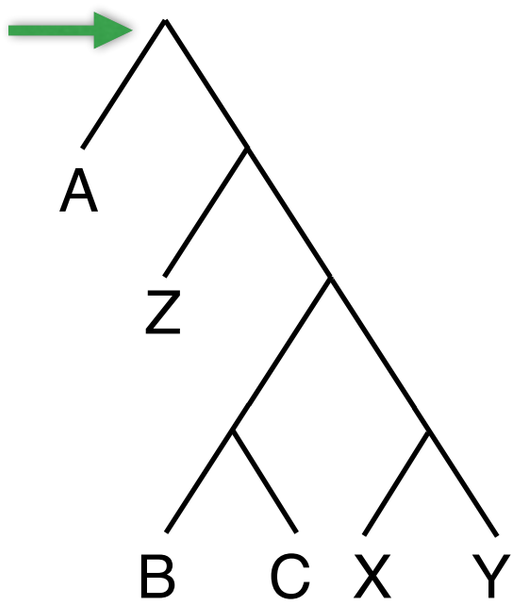
Δ =

1. {A, B}
2. {¬B, C}
3. {¬A, ¬Z, ¬X}
4. {¬A, ¬Z, X, Y}
5. {¬A, ¬Z, X, ¬Y}

Γ =

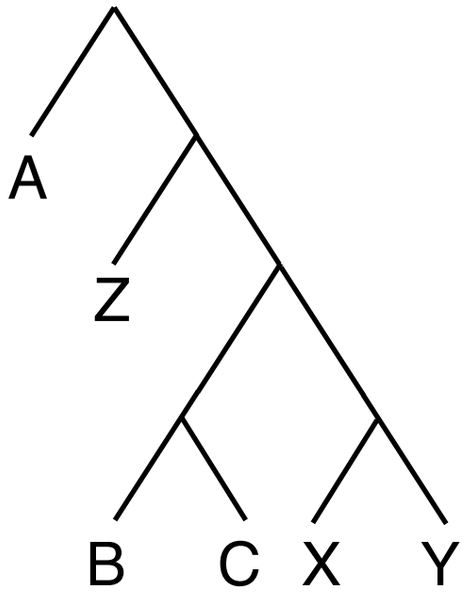
6. {¬A, ¬Z}



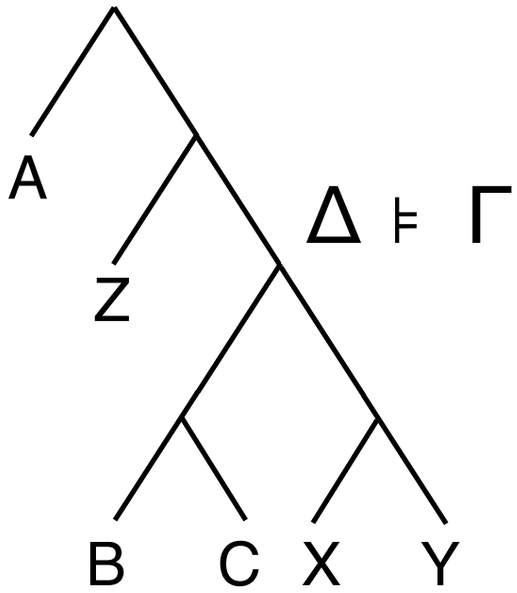


Caching in the New Algorithm

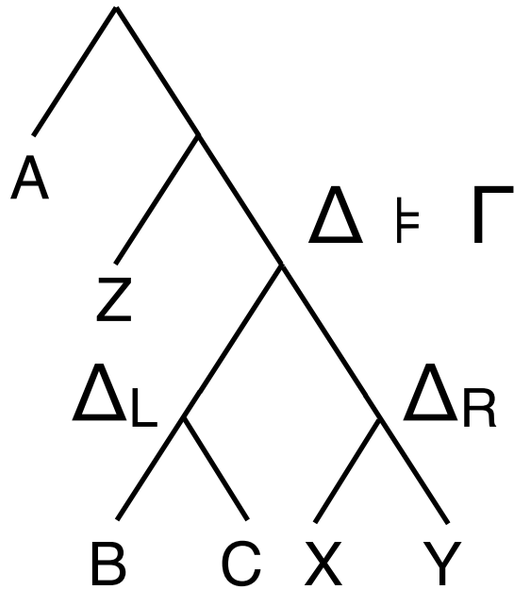
Caching in the New Algorithm



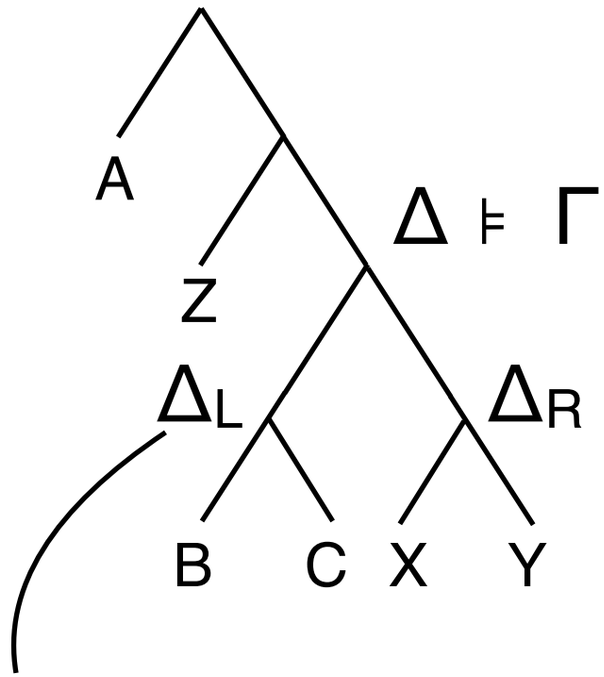
Caching in the New Algorithm



Caching in the New Algorithm



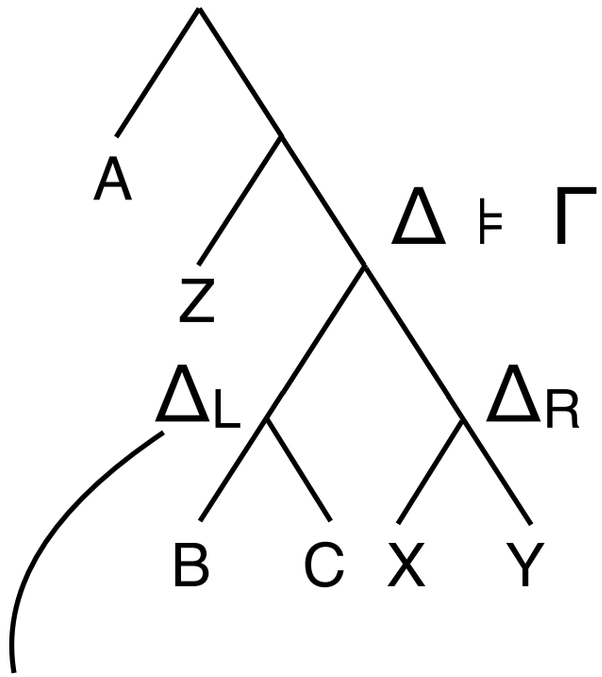
Caching in the New Algorithm



Counting $\Delta_L \wedge I_L$

I_L: literals implied by unit resolution from $\Delta_L \wedge \Gamma$

Caching in the New Algorithm



Counting $\Delta_L \wedge I_L$

I_L: literals implied by unit resolution from $\Delta_L \wedge \Gamma$

$\Delta_L \wedge I_L \equiv \Delta_L$ only if Δ is satisfiable

[Sang et al., 2004]

Proof Sketch of Correctness

Proof Sketch of Correctness

A SAT state S=(Δ, Γ, D, I) is satisfiable if $\Delta \wedge D$ is satisfiable

Proof Sketch of Correctness

A SAT state S=(Δ, Γ, D, I) is satisfiable if $\Delta \wedge D$ is satisfiable

CNF(v,S): clauses of S that mention variables inside v

Proof Sketch of Correctness

A SAT state S=(Δ, Γ, D, I) is satisfiable if $\Delta \wedge D$ is satisfiable

CNF(v,S): clauses of S that mention variables inside v

Theorem: A call #SAT(v,S) with a satisfiable state S will return either

— the model count of component CNF(v,S); or

—a learned clause.

Moreover, if v is the root vtree node, then a model count will be returned.

Proof Sketch of Correctness

A SAT state S=(Δ, Γ, D, I) is satisfiable if $\Delta \wedge D$ is satisfiable

CNF(v,S): clauses of S that mention variables inside v

Theorem: A call #SAT(v,S) with a satisfiable state S will return either

— the model count of component CNF(v,S); or

—a learned clause.

Moreover, if v is the root vtree node, then a model count will be returned.

Theorem: A call #SAT(v,S) with an unsatisfiable state S will either

— return a learned clause; or

—one of its ancestral calls #SAT(u,S) will return a learned clause

where u is a decomposition node.

Proof Sketch of Correctness

A SAT state S=(Δ, Γ, D, I) is satisfiable if $\Delta \wedge D$ is satisfiable

CNF(v,S): clauses of S that mention variables inside v

Theorem: A call #SAT(v,S) with a satisfiable state S will return either
— the model count of component CNF(v,S); or
—a learned clause.
Moreover, if v is the root vtree node, then a model count will be returned.

Theorem: A call #SAT(v,S) with an unsatisfiable state S will either
— return a learned clause; or
—one of its ancestral calls #SAT(u,S) will return a learned clause
where u is a decomposition node.

Corollary: If v is the root vtree node, then the call #SAT(v,(Δ,{},⟨⟩,{})) returns:
— the model count of Δ if Δ is satisfiable;
— an empty clause if Δ is unsatisfiable.

$\#\texttt{SAT}(v, S)$

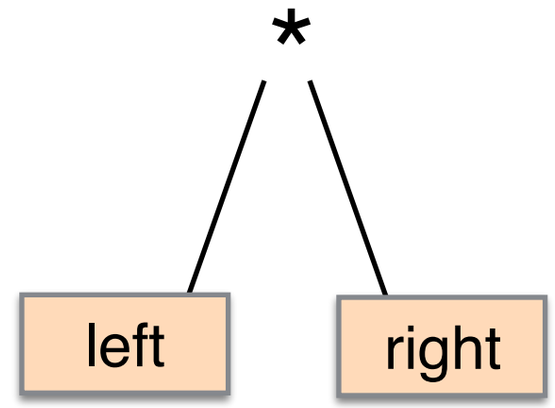
Input: v : a vtree node, S : a SAT state (Δ, Γ, D, I)
Output: A model count or a clause

if *v is a leaf node* **then**
  $X \leftarrow$ variable of v
  if X *or* $\neg X$ *belongs to* I **then return** *1*
  else return *2*

if *v is a decomposition vtree node* **then**
  $left \leftarrow \#\texttt{SAT}(v^l, S)$
  if $left$ *is a learned clause* **then**
    $clean\_cache(v^l)$
    return $left$
  $right \leftarrow \#\texttt{SAT}(v^r, S)$
  if $right$ *is a learned clause* **then**
    $clean\_cache(v)$
    return $right$
  return $left \times right$

Base case

Component analysis



Shannon node (next slide)

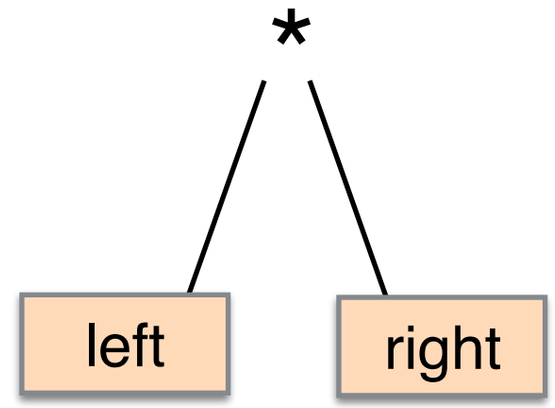
#SAT(v, S)

Input: v : a vtree node, S : a SAT state (Δ, Γ, D, I)
Output: A model count or a clause

```
if v is a leaf node then
    X ← variable of v
    if X or ¬X belongs to I then return 1
    else return 2

if v is a decomposition vtree node then
    left ← #SAT(v^l, S)
    if left is a learned clause then
        clean_cache(v^l)
        return left

    right ← #SAT(v^r, S)
    if right is a learned clause then
        clean_cache(v)
        return right

    return left × right
```

Base case

Component analysis



Shannon node (next slide)

Pseudocode: ~45 lines
C code: ~90 lines

#SAT(v, S)

Input: v : a vtree node, S : a SAT state (Δ, Γ, D, I)
Output: A model count or a clause

if *v is a Shannon node* **then**
  $key \leftarrow Key(v, S)$
  if $cache(key) \neq nil$ **then return** $cache(key)$

  $X \leftarrow$ Shannon variable of v
  if *either* X *or* $\neg X$ *belongs to* I **then return** #SAT(v^r, S)

  $h \leftarrow decide\_literal(X, S)$
  if h *is success* **then** $h \leftarrow$ #SAT(v^r, S)
  $undo\_decide\_literal(X, S)$
  if h *is a learned clause* **then**
    if $at\_assertion\_level(h, S)$ **then**
      $h \leftarrow assert\_clause(h, S)$
      if h *is success* **then return** #SAT(v, S)
      else return h
    else return h

  $l \leftarrow decide\_literal(\neg X, S)$
  if l *is success* **then** $l \leftarrow$ #SAT(v^r, S)
  $undo\_decide\_literal(\neg X, S)$
  if l *is a learned clause* **then**
    if $at\_assertion\_level(l, S)$ **then**
      $l \leftarrow assert\_clause(l, S)$
      if l *is success* **then return** #SAT(v, S)
      else return l
    else return l

  $count \leftarrow h + l$
  $cache(key) \leftarrow count$
  return $count$

Component caching

Try with X

Try with ¬X

Component caching

Open Source Package

Open Source Package

- miniC2D, version 1.00

- Implements the framework discussed
  - simple code, just like the shown pseudocode!
  - knowledge compiler
  - model counter (weighted)

- To be released, this summer in 2015

- Will be posted on Beyond NP!