# Knowledge Compilation: Theory, Practice, and Applications

Jean-Marie Lagniez[1] Johannes Fichte[2]

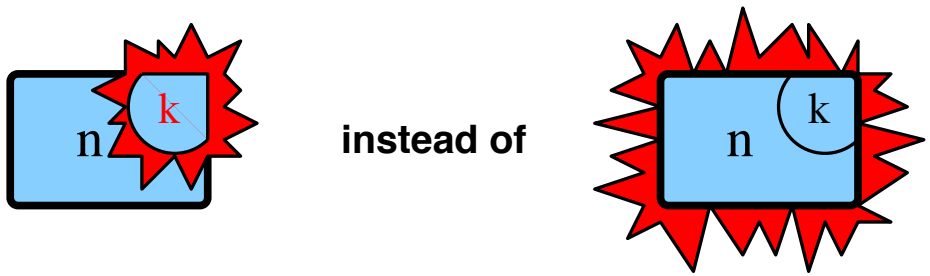
[1] CRIL, Univ. Artois & CNRS, France
[2] Linköping University, Sweden

# Knowledge Compilation

# Outline

## Treewidth $k$ — Bertelè and Brioschi (1973); Robertson and Seymour (1983)

- renders large variety of NP-hard problems *fixed-parameter tractable (FPT)*

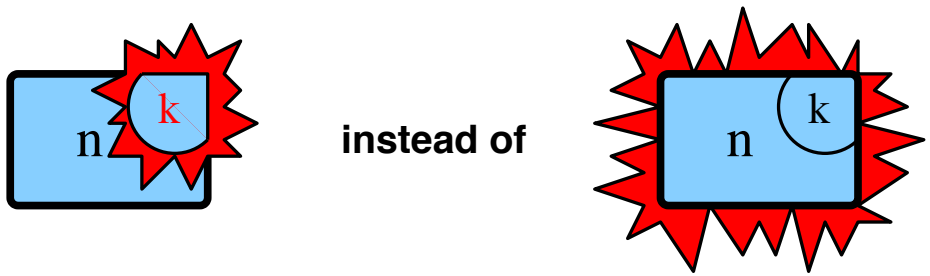



$f(k) \cdot poly(n)$ vs. $2^n$

# Excursion into Parameterized Complexity

## Treewidth $k$ — Bertelè and Brioschi (1973); Robertson and Seymour (1983)

- renders large variety of NP-hard problems *fixed-parameter tractable (FPT)*




$f(k) \cdot poly(n)$ vs. $2^n$

## Algorithms

- Dynamic Programming

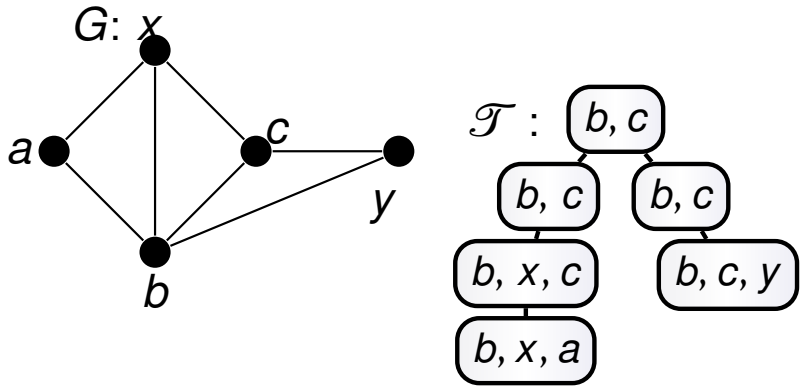
$\Rightarrow$ on Tree Decompositions (incidence) construct d-DNNFs by Bova et al. (2015)

$\Rightarrow$ on Path Decompositions construct OBDDs by Amarilli et al. (2020)

$\Rightarrow$ (indirect results) about treewidth and SDDs by Bova and Szeider (2017)

# Tree Decompositions (TDs)

## Tree Decomposition $\mathcal{T}$ of $G$




## Definition

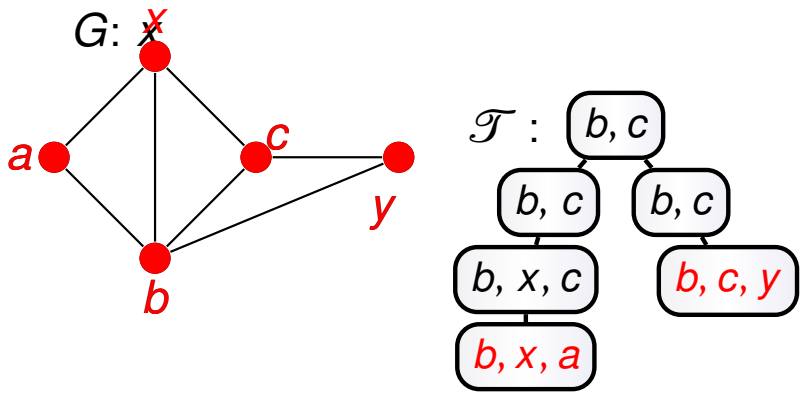
Robertson and Seymour (1983)

A tree decomposition is a tree of *bags* obtained from a graph s.t.

1. Every vertex must occur in some *bag*
2. Every edge must be covered by some *bag*
3. *Connected*: Tree "restricted" to any vertex must be connected

# Tree Decompositions (TDs)

## Tree Decomposition $\mathcal{T}$ of $G$




## Definition

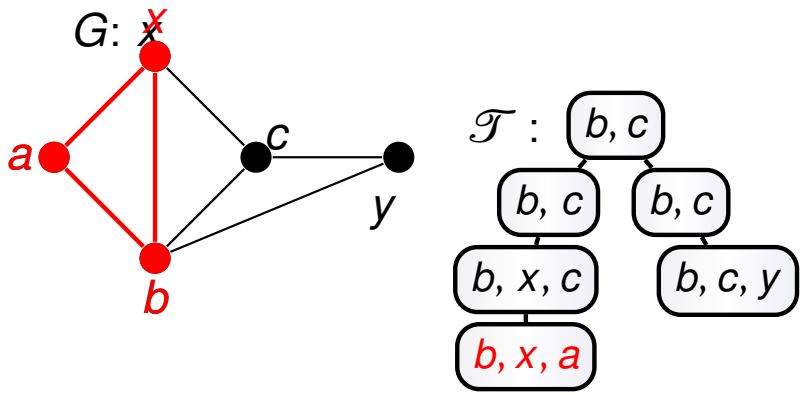
Robertson and Seymour (1983)

A tree decomposition is a tree of *bags* obtained from a graph s.t.

1. Every vertex must occur in some *bag*
2. Every edge must be covered by some *bag*
3. *Connected*: Tree "restricted" to any vertex must be connected

# Tree Decompositions (TDs)

## Tree Decomposition $\mathcal{T}$ of $G$




## Definition

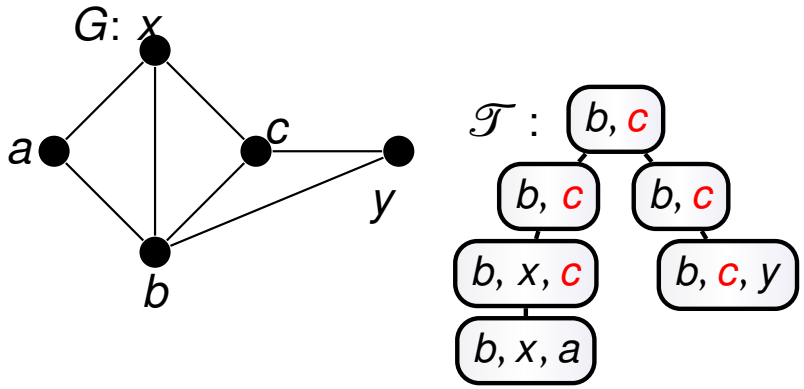
Robertson and Seymour (1983)

A tree decomposition is a tree of *bags* obtained from a graph s.t.

1. Every vertex must occur in some *bag*
2. Every edge must be covered by some *bag*
3. *Connected*: Tree "restricted" to any vertex must be connected

# Tree Decompositions (TDs)

## Tree Decomposition $\mathcal{T}$ of $G$




## Definition

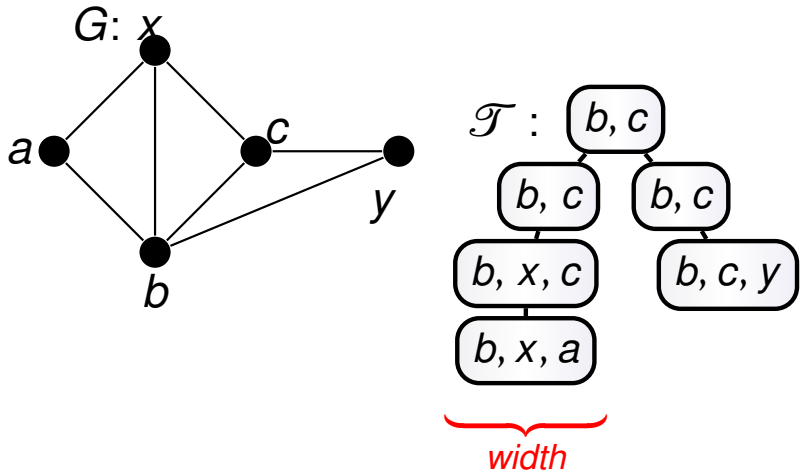
Robertson and Seymour (1983)

A tree decomposition is a tree of *bags* obtained from a graph s.t.

1. Every vertex must occur in some *bag*
2. Every edge must be covered by some *bag*
3. *Connected*: Tree "restricted" to any vertex must be connected

# Tree Decompositions (TDs)

## Tree Decomposition $\mathcal{T}$ of $G$




## Definition

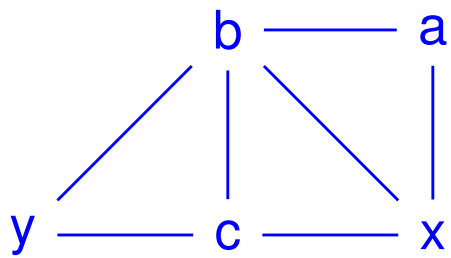
Robertson and Seymour (1983)

A tree decomposition is a tree of *bags* obtained from a graph s.t.

1. Every vertex must occur in some *bag*
2. Every edge must be covered by some *bag*
3. *Connected*: Tree "restricted" to any vertex must be connected

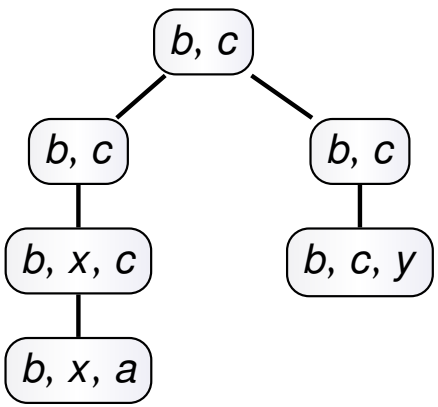
$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation



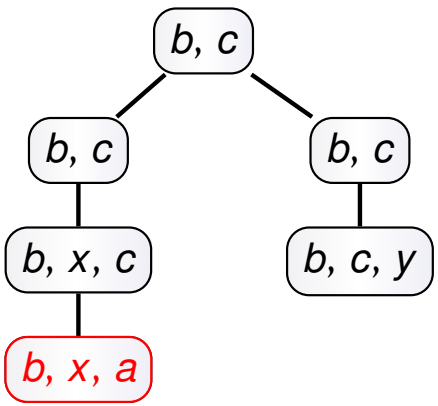
$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

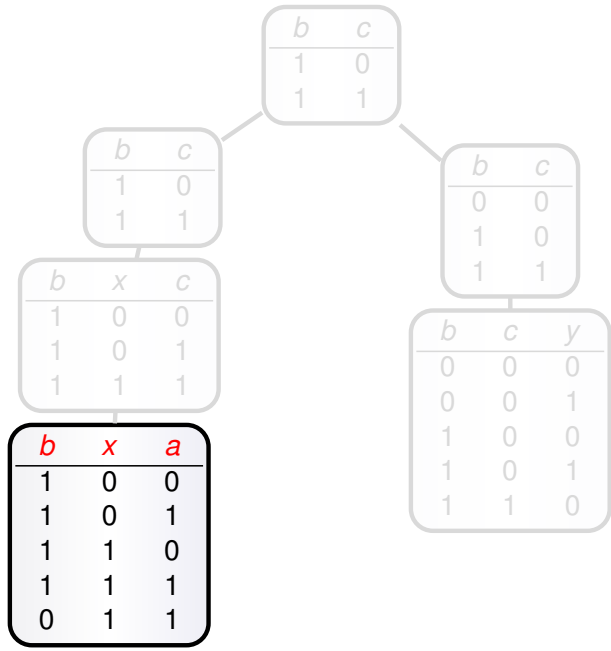
1. Create graph representation
2. Decompose graph



$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve subproblems

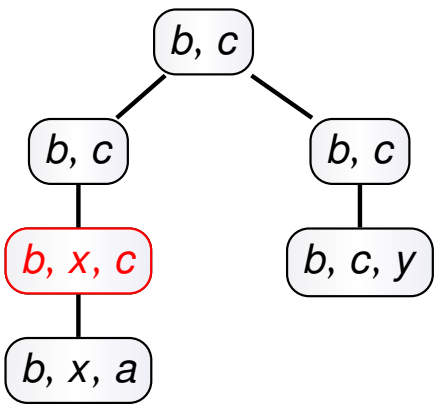


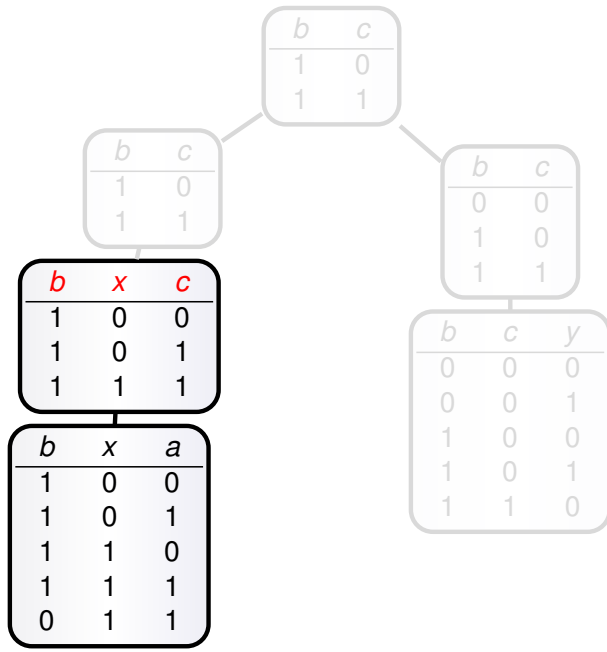




"Local formula" $F_t$ clauses whose variables are contained in the bag (colored in red above)

$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

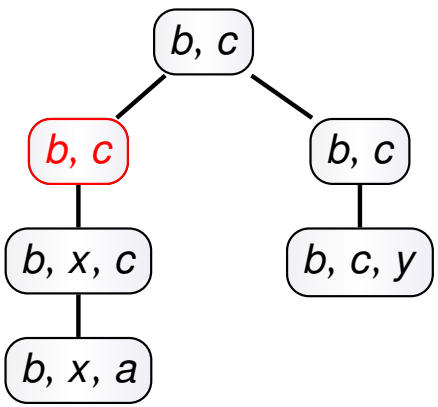
1. Create graph representation
2. Decompose graph
3. Solve subproblems

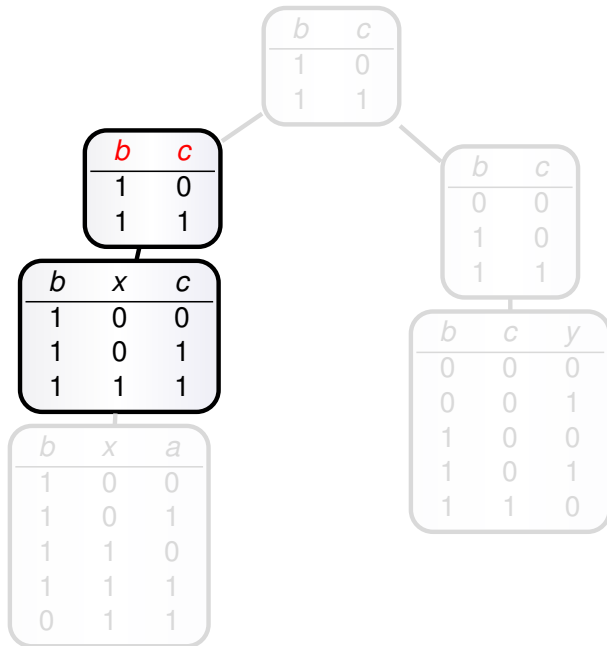


$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

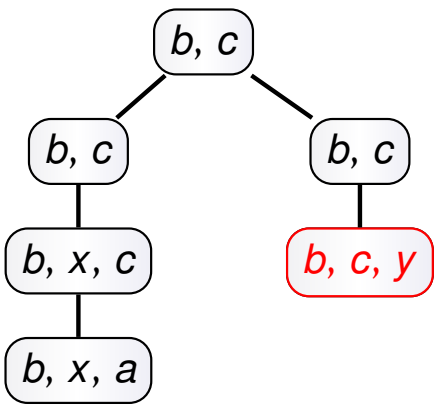
1. Create graph representation
2. Decompose graph
3. Solve subproblems

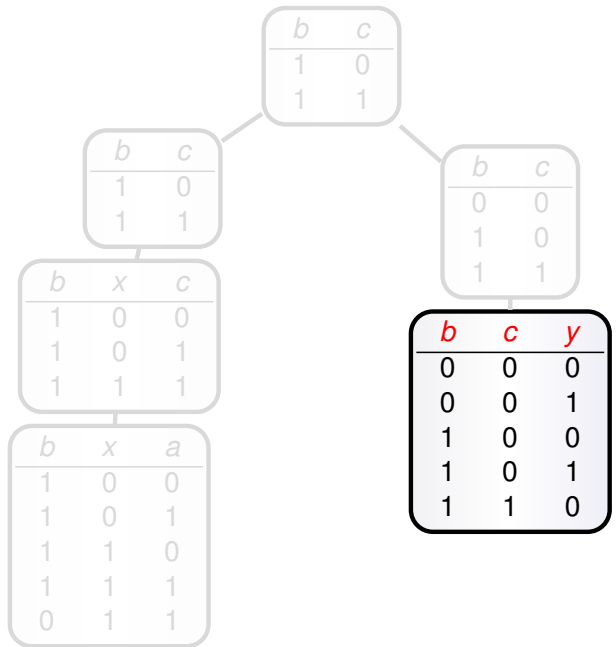


$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

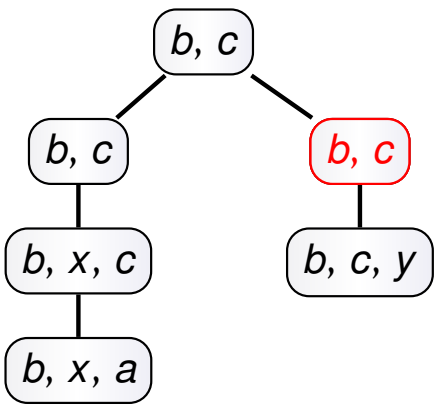
1. Create graph representation
2. Decompose graph
3. Solve subproblems

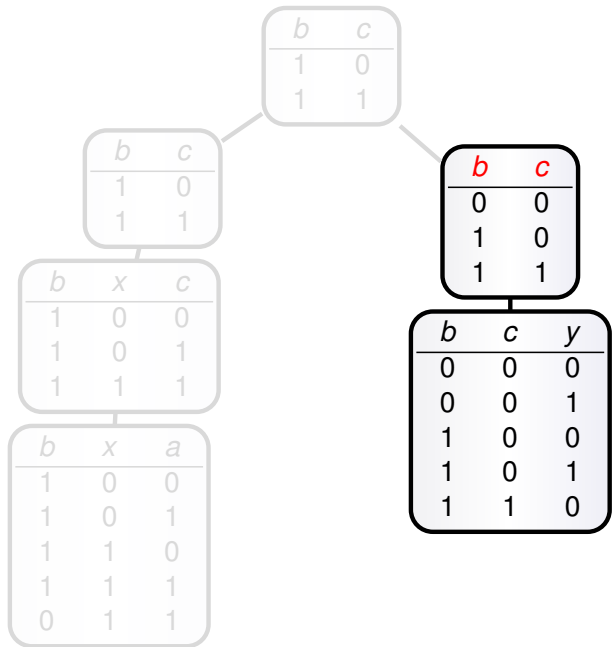


$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

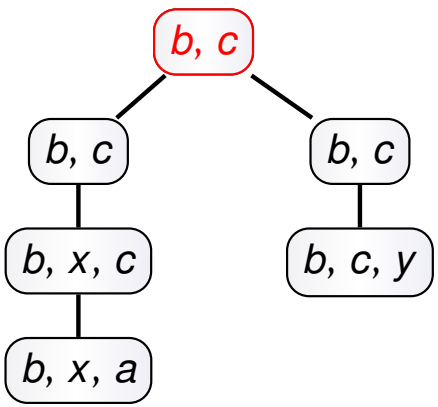
1. Create graph representation
2. Decompose graph
3. Solve subproblems

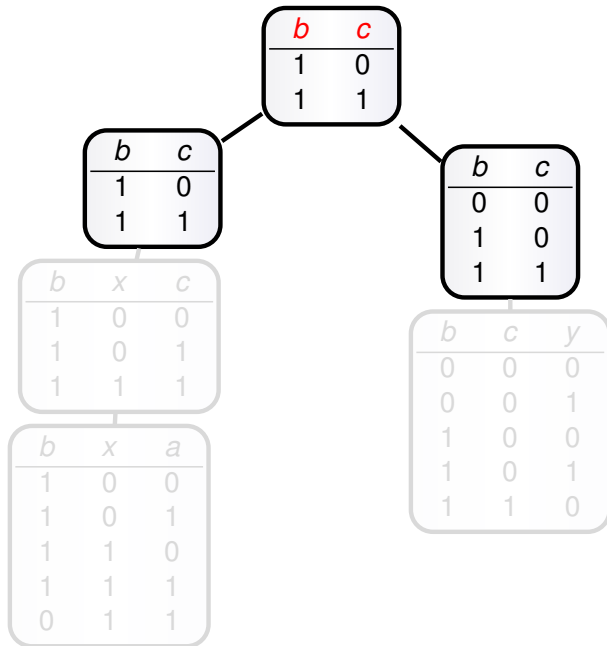


$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

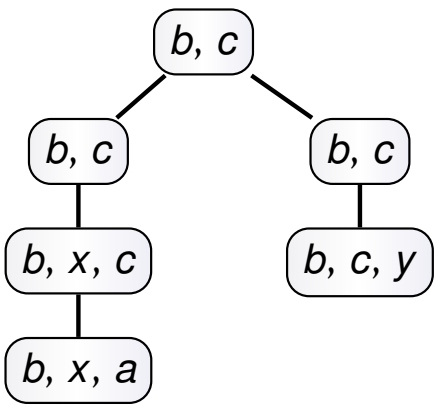
1. Create graph representation
2. Decompose graph
3. Solve subproblems

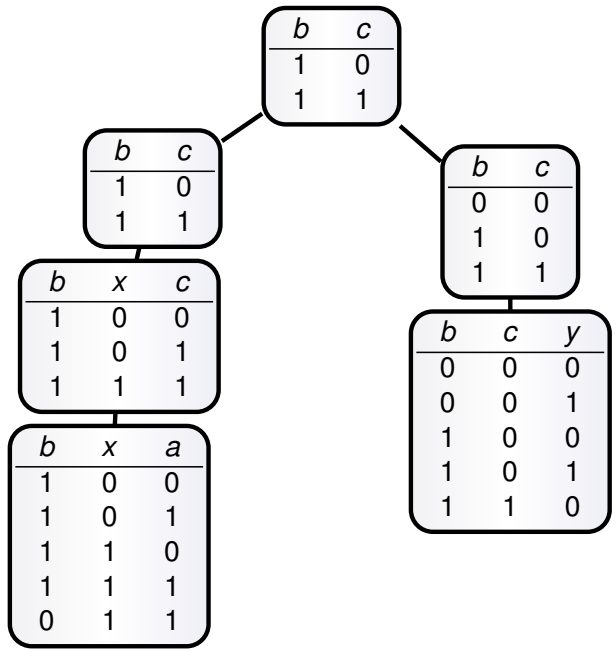


$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

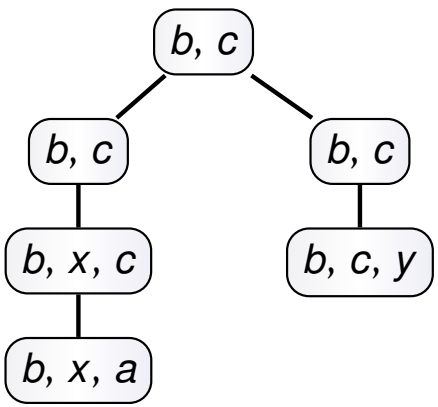
1. Create graph representation
2. Decompose graph
3. Solve subproblems

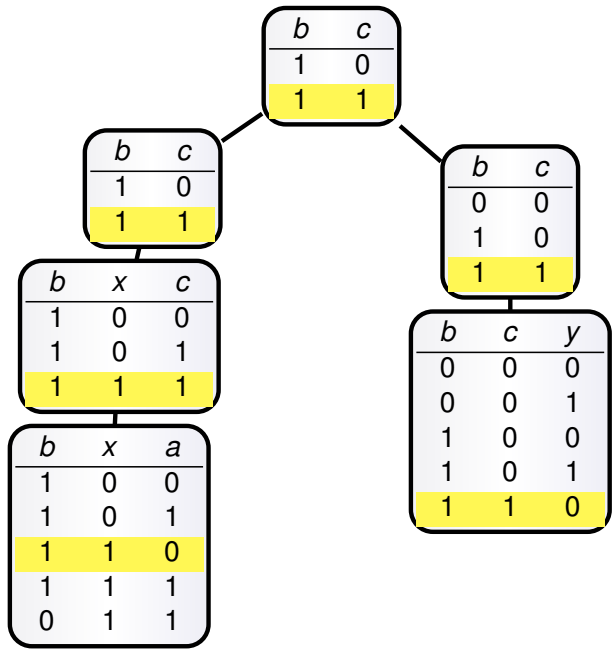


$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

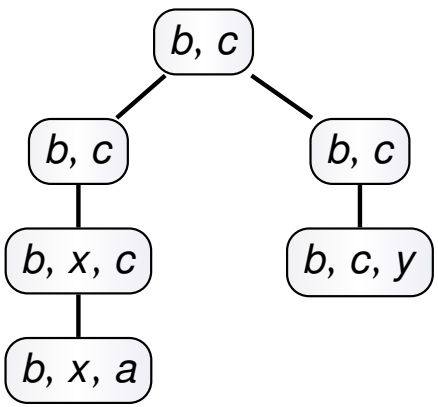
1. Create graph representation
2. Decompose graph
3. Solve subproblems
4. Combine rows

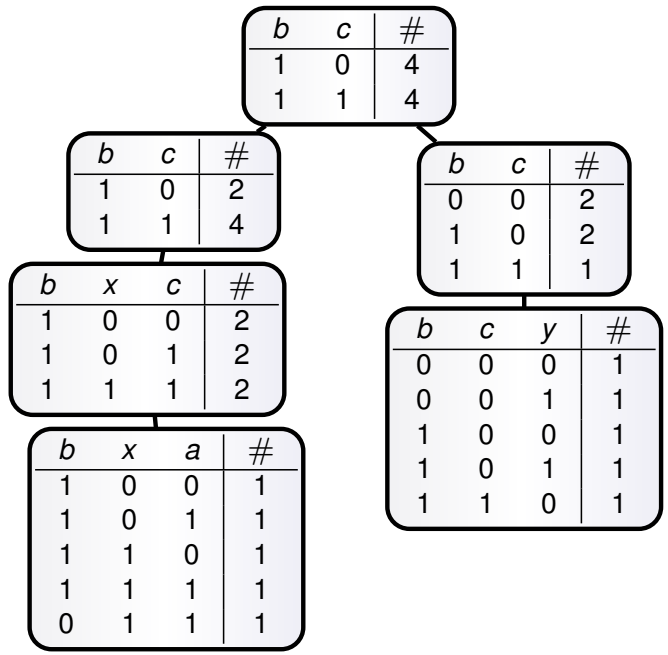


$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve subproblems
4. Combine rows



| b | c | # |
|---|---|---|
| 1 | 0 | 4 |
| 1 | 1 | 4 |

| b | c | # |
|---|---|---|
| 1 | 0 | 2 |
| 1 | 1 | 4 |

| b | x | c | # |
|---|---|---|---|
| 1 | 0 | 0 | 2 |
| 1 | 0 | 1 | 2 |
| 1 | 1 | 1 | 2 |

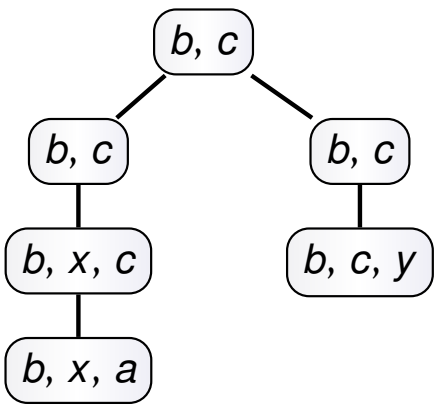
| b | x | a | # |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |

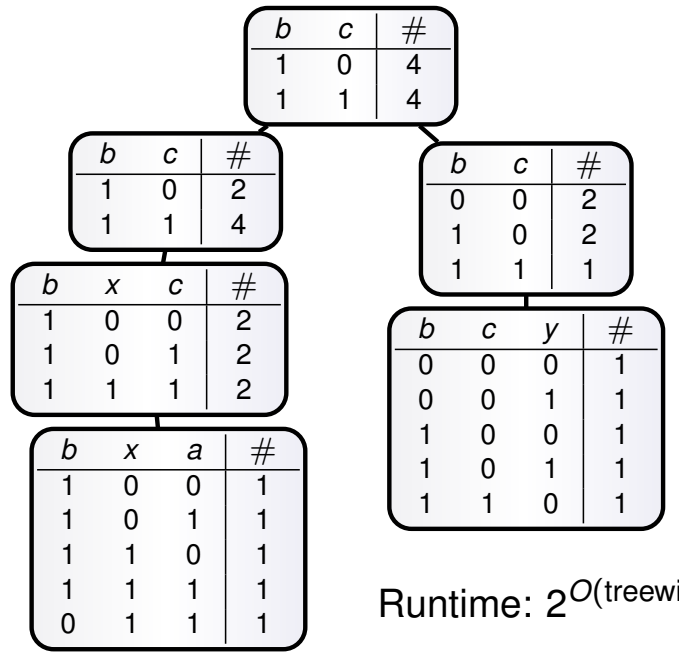
| b | c | # |
|---|---|---|
| 0 | 0 | 2 |
| 1 | 0 | 2 |
| 1 | 1 | 1 |

| b | c | y | # |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |

$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve subproblems
4. Combine rows



| b | c | # |
|---|---|---|
| 1 | 0 | 4 |
| 1 | 1 | 4 |

| b | c | # |
|---|---|---|
| 1 | 0 | 2 |
| 1 | 1 | 4 |

| b | x | c | # |
|---|---|---|---|
| 1 | 0 | 0 | 2 |
| 1 | 0 | 1 | 2 |
| 1 | 1 | 1 | 2 |

| b | x | a | # |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |

| b | c | # |
|---|---|---|
| 0 | 0 | 2 |
| 1 | 0 | 2 |
| 1 | 1 | 1 |

| b | c | y | # |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |

Runtime: $2^{O(\text{treewidth})} \cdot \text{poly}(|F|)$

# Outline

# A New Paradigm: Bottom-Up Compilation

- So far, we have looked at **Top-Down** compilers: we take a massive `CNF` formula, run a SAT solver, and trace the search space.
- **Bottom-Up** compilers work completely differently:
  1. Compile the simplest base elements (individual variables/literals).
  2. Recursively combine them using a magical function called `Apply`.

## The `Apply` Function

`Apply`$(\Phi, \Psi, \circ)$ takes two compiled representations and a Boolean operator (e.g., $\wedge, \vee$) and returns the exact, minimized compiled representation of $(\Phi \circ \Psi)$.

- To make this work efficiently, the `Apply` function must run in polynomial time. This requires the target language to be **Canonical** (every logic function has exactly one unique representation).

# Outline

# Ordered Binary Decision Diagrams ($\texttt{OBDD}_<$) (Bryant, 2018)

- An $\texttt{OBDD}_<$ is simply an FBDD (each variable tested at most once) with one strict, unyielding rule: **Variables must be tested in the exact same order ($\pi$) on every single path.**

## The Superpower of Strict Ordering

- Because of this strict ordering, $\texttt{OBDD}_<$s are strongly **Canonical**.
- This makes the `Apply` function strictly polynomial ($\mathcal{O}(|\Phi| \cdot |\Psi|)$)!
- We can instantly test if two formulas are equivalent (**EQ**) just by checking if they point to the exact same node in memory!

# Compiling an $\texttt{OBDD}_<$: The Bottom-Up Strategy

- Let's compile our full 4-clause `CNF` formula without any pre-simplification:

$$\Sigma = C_1 \wedge C_2 \wedge C_3 \wedge C_4$$

  - $C_1 = (\neg x \vee y \vee \neg z)$
  - $C_2 = (x \vee y \vee z)$
  - $C_3 = (y \vee t \vee u)$
  - $C_4 = (\neg y \vee t \vee u)$
- Variable ordering: $\pi = x \prec y \prec z \prec t \prec u$.

## The Algorithm

A bottom-up compiler builds the final graph by sequentially merging the clauses using the `Apply` function:

1. Compute $OBDD_{12} = \texttt{Apply}(C_1, C_2)$
2. Compute $OBDD_{34} = \texttt{Apply}(C_3, C_4)$
3. Compute $Final = \texttt{Apply}(OBDD_{12}, OBDD_{34})$

# The Formal `Apply` Algorithm

- We want to compile $\Sigma = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ bottom-up.
- We use the recursive `Apply` algorithm, defined by the Shannon expansion:

## Recursive Characterization

$$\texttt{Apply}(f, g) = \neg x \cdot \texttt{Apply}(f_{|x=0}, g_{|x=0}) + x \cdot \texttt{Apply}(f_{|x=1}, g_{|x=1})$$

*Where $x$ is the highest variable in the ordering $\pi$ currently present in $f$ or $g$.*

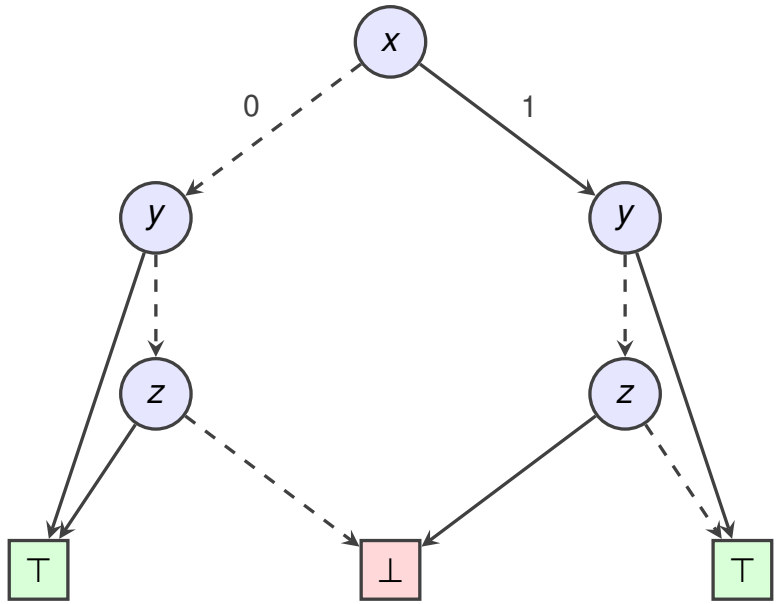
- **The Reduce Rule:** During this bottom-up construction, if the algorithm computes that the 0-branch and 1-branch point to the **exact same node**, it skips creating the $x$ node and simply returns the child node.
- **Base Cases:** $\texttt{Apply}(\top, g) = g$ and $\texttt{Apply}(\bot, g) = \bot$.

## Step 1: $\texttt{Apply}(C_1, C_2, \wedge)$

- $C_1 = (\neg x \vee y \vee \neg z)$ and $C_2 = (x \vee y \vee z)$.
- Top variable is $x$. Let's trace the recursion:

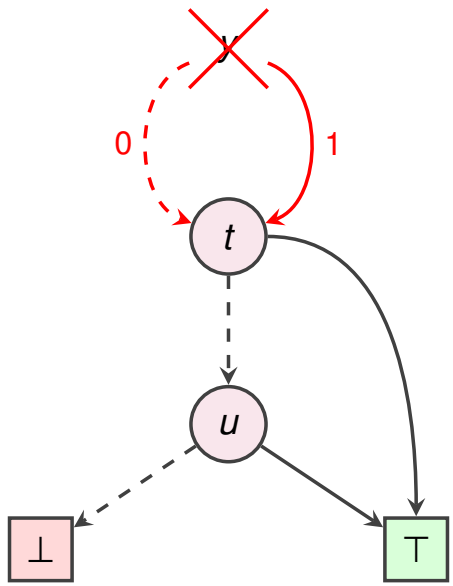
## Branching on $x$

- **If $x = 0$:** $C_1$ evaluates to $\top$. $C_2$ evaluates to $(y \vee z)$.
  $\texttt{Apply}(\top, y \vee z, \wedge) = \mathbf{y} \vee \mathbf{z}$
- **If $x = 1$:** $C_1$ evaluates to $(y \vee \neg z)$. $C_2$ evaluates to $\top$.
  $\texttt{Apply}(y \vee \neg z, \top, \wedge) = \mathbf{y} \vee \neg\mathbf{z}$



## Step 2: $\texttt{Apply}(C_3, C_4)$

- $C_3 = (y \vee t \vee u)$ and $C_4 = (\neg y \vee t \vee u)$.
- Top variable is $y$. We expand:
  - **If $y = 0$:** $C_3 \to (t \vee u)$, and $C_4 \to \top$.
    $\texttt{Apply}(t \vee u, \top) = \mathbf{t} \vee \mathbf{u}$
  - **If $y = 1$:** $C_3 \to \top$, and $C_4 \to (t \vee u)$.
    $\texttt{Apply}(\top, t \vee u) = \mathbf{t} \vee \mathbf{u}$

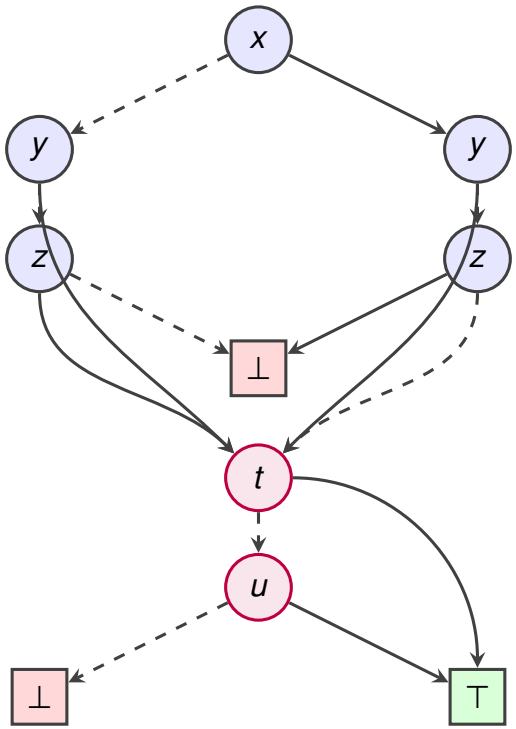



## The Reduce Rule Triggers

Because `Apply` returns the exact same subgraph $(t \vee u)$ for both branches, the $y$ node is strictly redundant. The algorithm eliminates it entirely, automatically discovering the Boolean resolution!

# Step 3: $\texttt{Apply}(OBDD_{12}, OBDD_{34})$

- We now compute the final graph. The recursion descends through $OBDD_{12}$.
- Because the variables $t, u$ in $OBDD_{34}$ appear strictly lower in $\pi$ than $x, y, z$, the algorithm simply passes $OBDD_{34}$ down the calls until it hits the leaves of $OBDD_{12}$:
  - $\texttt{Apply}(\top, OBDD_{34}) = \mathbf{OBDD_{34}}$
  - $\texttt{Apply}(\bot, OBDD_{34}) = \bot$



# KC Map for $\mathtt{OBDD}_<$: Queries

| $\mathscr{L}$ | CO | VA | CE | IM | EQ | SE | CT | ME |
|---|---|---|---|---|---|---|---|---|
| Circ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| CNF | ○ | √ | ○ | √ | ○ | ○ | ○ | ○ |
| DNF | √ | ○ | √ | ○ | ○ | ○ | ○ | √ |
| ODNF | √ | √ | √ | √ | √ | √ | √ | √ |
| MODS | √ | √ | √ | √ | √ | √ | √ | √ |
| decision − DNNF | √ | √ | √ | √ | **?** | ○ | √ | √ |
| FBDD | √ | √ | √ | √ | ○ | ○ | √ | √ |
| DT | √ | √ | √ | √ | √ | √ | √ | √ |
| EADT | √ | √ | √ | √ | **?** | ○ | √ | √ |
| ADT | √ | √ | √ | √ | √ | √ | √ | √ |
| $\mathtt{OBDD}_<$ | √ | √ | √ | √ | √ | √ | √ | √ |

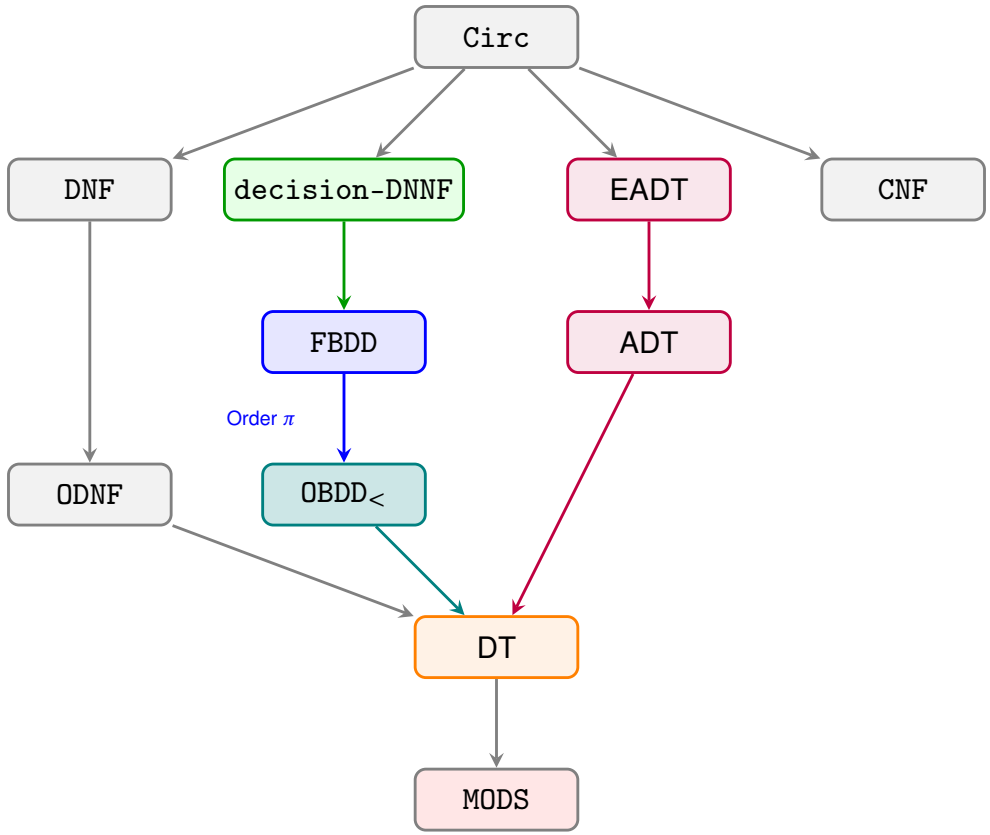
- **Absolute Perfection:** The strict ordering $\pi$ gives $\mathtt{OBDD}_<$ a perfect score on the Query map, rescuing the **EQ** and **SE** queries that standard DAGs (FBDD, decision-DNNF) lose.

# KC Map for $\mathtt{OBDD}_<$: Transformations

| $\mathscr{L}$ | **CD** | **FO** | **SFO** | **$\wedge$C** | **$\wedge$BC** | **$\vee$C** | **$\vee$BC** | **$\neg$C** |
|---|---|---|---|---|---|---|---|---|
| Circ | $\surd$ | $\circ$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| CNF | $\surd$ | $\circ$ | $\surd$ | $\surd$ | $\surd$ | $\circ$ | $\surd$ | $\circ$ |
| DNF | $\surd$ | $\surd$ | $\surd$ | $\circ$ | $\surd$ | $\surd$ | $\surd$ | $\circ$ |
| ODNF | $\surd$ | $\circ$ | $\circ$ | $\circ$ | $\surd$ | $\circ$ | $\circ$ | $\circ$ |
| MODS | $\surd$ | $\surd$ | $\surd$ | $\circ$ | $\surd$ | $\circ$ | $\surd$ | $\circ$ |
| decision$-$DNNF | $\surd$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | **?** |
| FBDD | $\surd$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\surd$ |
| DT | $\surd$ | $\circ$ | $\surd$ | $\circ$ | $\surd$ | $\circ$ | $\surd$ | $\surd$ |
| EADT | $\surd$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\circ$ | $\surd$ |
| ADT | $\surd$ | $\circ$ | $\surd$ | $\circ$ | $\surd$ | $\circ$ | $\surd$ | $\surd$ |
| $\mathtt{OBDD}_<$ | $\surd$ | $\circ$ | $\surd$ | $\circ$ | $\surd$ | $\circ$ | $\surd$ | $\surd$ |

- **The Benefit of `Apply`:** Because the `Apply` function is polynomial, Bounded Conjunction ($\wedge$**BC**) and Bounded Disjunction ($\vee$**BC**) are guaranteed $\surd$.

# Succinctness (Up to $\mathtt{OBDD}_<$)




## The Flaw of $\mathtt{OBDD}_<$

$\mathtt{FBDD} <_S \mathtt{OBDD}_<$: Forcing variables to be tested in the **exact same order** on every branch destroys our ability to dynamically adapt to sub-problems.

# Outline

# The Foundation of SDD (Darwiche, 2011)

- $\texttt{OBDD}_<$s force a strict, linear variable ordering ($\pi$).
- **Sentential Decision Diagrams (SDD)** generalize this by replacing the linear order with a **vtree**.
- Branching moves from literals to sentences

## What is a vtree?

A vtree is a full binary tree whose leaves are the variables of the formula. Every internal node recursively partitions the variables into two disjoint sets: $X$ (left child) and $Y$ (right child).

- By structuring the compilation according to a vtree, SDDs can cleanly split independent variables (just like `decision-DNNF`) while maintaining a strict enough structure to remain **Canonical** (like $\texttt{OBDD}_<$).

# Bottom-Up Compilation of SDDs

- Just like with $\texttt{OBDD}_<$, we compile an SDD from the bottom up:
  1. Compile the base variables into trivial SDDs.
  2. Combine them using an $\texttt{Apply}(\Phi, \Psi)$ function.

## The SDD Apply Function

Instead of branching on a single variable $x$, the SDD Apply function is guided by the vtree. At each node, a function is represented as $\bigvee_i (p_i \wedge s_i)$:

- $p_i$ (**primes**) are functions over the left variables $X$ (mutually exclusive + exhaustive).
- $s_i$ (**subs**) are functions over the right variables $Y$.

The algorithm recursively intersects the primes and applies the conjunction to the subs.

## Garbage Collection

During these recursive Apply operations, the compiler generates many intermediate "dead nodes" that are not part of the final result. The compiler must continuously run a **Garbage Collector** to free memory!

# Example: CNF to SDD Compilation

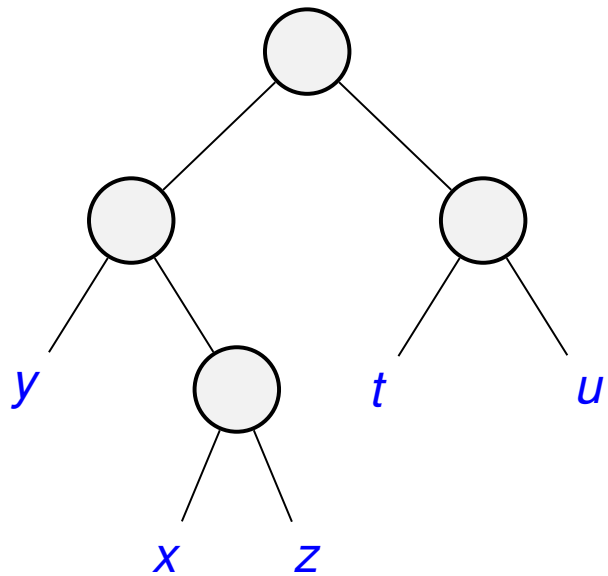
## The CNF Formula $\Sigma$

Let's compile a full 4-clause CNF formula:

$$\Sigma = C_1 \wedge C_2 \wedge C_3 \wedge C_4$$

- $C_1 = (\neg x \vee y \vee \neg z)$
- $C_2 = (x \vee y \vee z)$
- $C_3 = (y \vee t \vee u)$
- $C_4 = (\neg y \vee t \vee u)$

**The Variable Tree (vtree)**

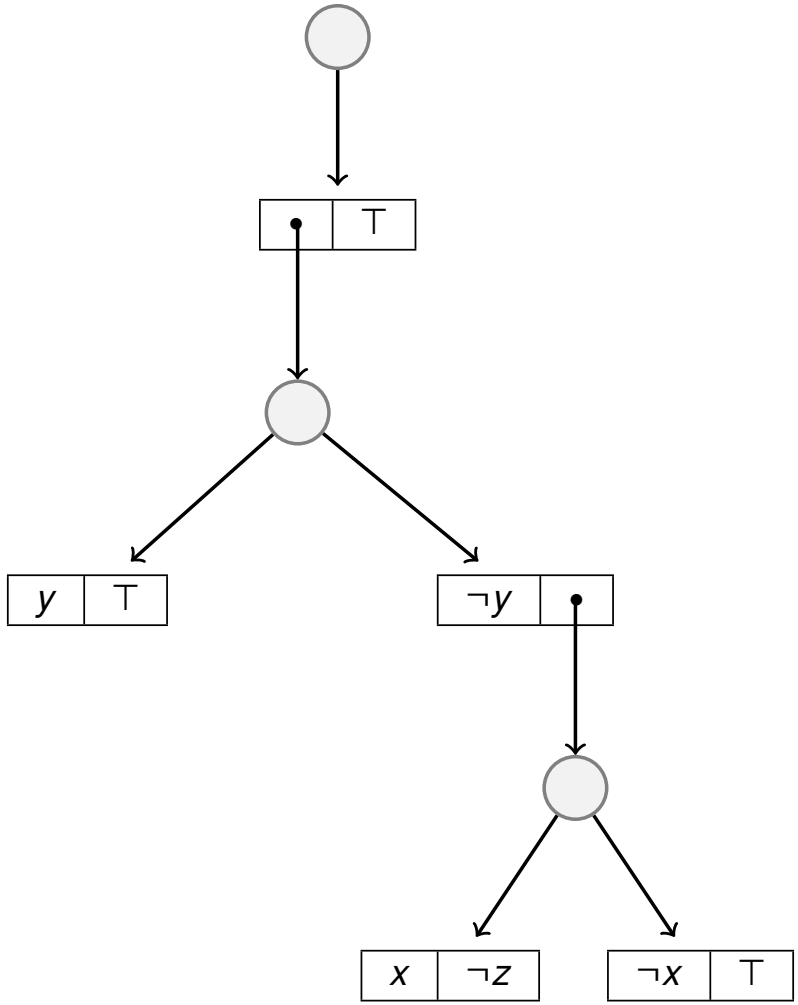



## Step 1: Bottom-Up Construction

First, we compile each clause ($C_1$ to $C_4$) separately into an SDD strictly respecting the partitions of the vtree above.

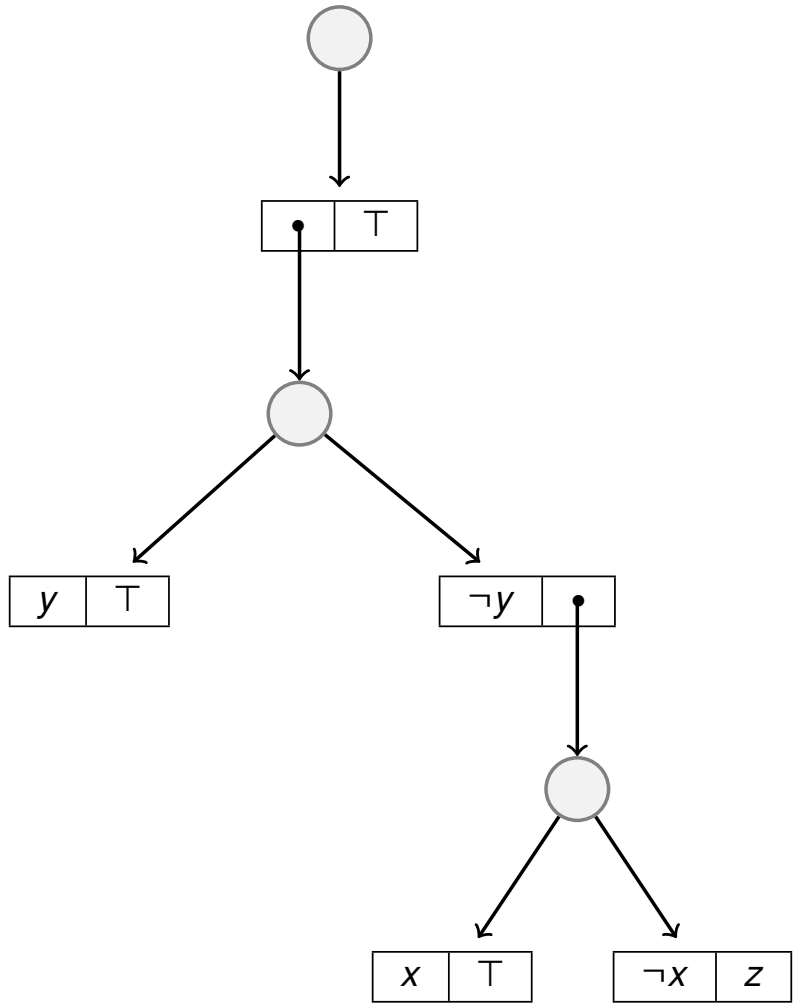
# Compiling $C_1$ to an SDD

For the clause $\neg x \vee y \vee \neg z$ we get the following SDD:



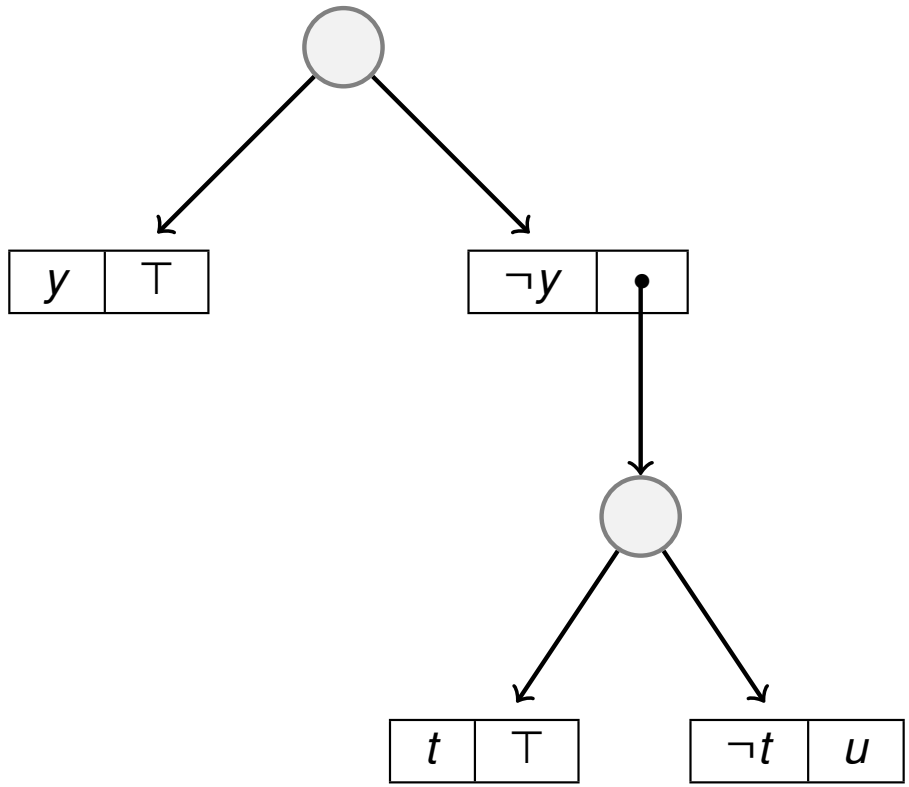
# Compiling $C_2$ to an SDD

For the clause $x \vee y \vee z$ we get the following SDD:



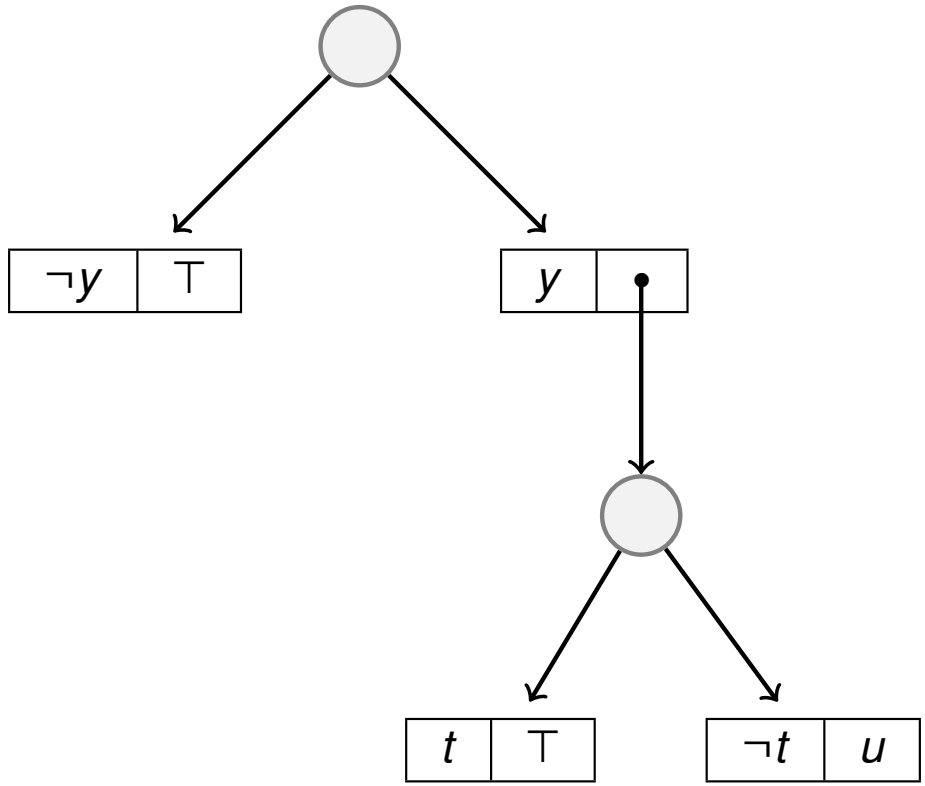
# Compiling $C_3$ to an SDD

For the clause $y \vee t \vee u$ we get the following SDD:



# Compiling $C_4$ to an SDD

For the clause $\neg y \vee t \vee u$ we get the following SDD:



# The APPLY Operator: $C_1 \wedge C_2$

## The Core Logic (Cross-Product)

Given two SDDs normalized for the same vtree decision node:

- $C_1 = \bigvee_i (p_i, s_i)$
- $C_2 = \bigvee_j (p'_j, s'_j)$

Their conjunction is the Cartesian product of their elements:

$$C_1 \wedge C_2 = \bigvee_{i,j} (p_i \wedge p'_j, s_i \wedge s'_j)$$

**Efficient Bottom-Up Execution:**

# The APPLY Operator: $C_1 \wedge C_2$

## The Core Logic (Cross-Product)

Given two SDDs normalized for the same vtree decision node:

- $C_1 = \bigvee_i (p_i, s_i)$
- $C_2 = \bigvee_j (p'_j, s'_j)$

Their conjunction is the Cartesian product of their elements:

$$C_1 \wedge C_2 = \bigvee_{i,j} (p_i \wedge p'_j, s_i \wedge s'_j)$$

**Efficient Bottom-Up Execution:**

- **1. Intersect Primes:** Compute $p_i \wedge p'_j$. Because both respect the exact same left vtree branch, this is purely structural.

# The APPLY Operator: $C_1 \wedge C_2$

## The Core Logic (Cross-Product)

Given two SDDs normalized for the same vtree decision node:

- $C_1 = \bigvee_i (p_i, s_i)$
- $C_2 = \bigvee_j (p'_j, s'_j)$

Their conjunction is the Cartesian product of their elements:

$$C_1 \wedge C_2 = \bigvee_{i,j} (p_i \wedge p'_j, s_i \wedge s'_j)$$

**Efficient Bottom-Up Execution:**

- **1. Intersect Primes:** Compute $p_i \wedge p'_j$. Because both respect the exact same left vtree branch, this is purely structural.
- **2. Filter Inconsistencies:** Discard any pair where $p_i \wedge p'_j \equiv \bot$. (By definition, SDD primes must be satisfiable).

# The APPLY Operator: $C_1 \wedge C_2$

## The Core Logic (Cross-Product)

Given two SDDs normalized for the same vtree decision node:

- $C_1 = \bigvee_i (p_i, s_i)$
- $C_2 = \bigvee_j (p'_j, s'_j)$

Their conjunction is the Cartesian product of their elements:

$$C_1 \wedge C_2 = \bigvee_{i,j} (p_i \wedge p'_j, s_i \wedge s'_j)$$

**Efficient Bottom-Up Execution:**

- **1. Intersect Primes:** Compute $p_i \wedge p'_j$. Because both respect the exact same left vtree branch, this is purely structural.
- **2. Filter Inconsistencies:** Discard any pair where $p_i \wedge p'_j \equiv \bot$. (By definition, SDD primes must be satisfiable).
- **3. Recursive Apply:** Recursively compute the new subs, $s_i \wedge s'_j$, down the right branch of the vtree.

# The APPLY Operator: $C_1 \wedge C_2$

## The Core Logic (Cross-Product)

Given two SDDs normalized for the same vtree decision node:

- $C_1 = \bigvee_i (p_i, s_i)$
- $C_2 = \bigvee_j (p'_j, s'_j)$

Their conjunction is the Cartesian product of their elements:

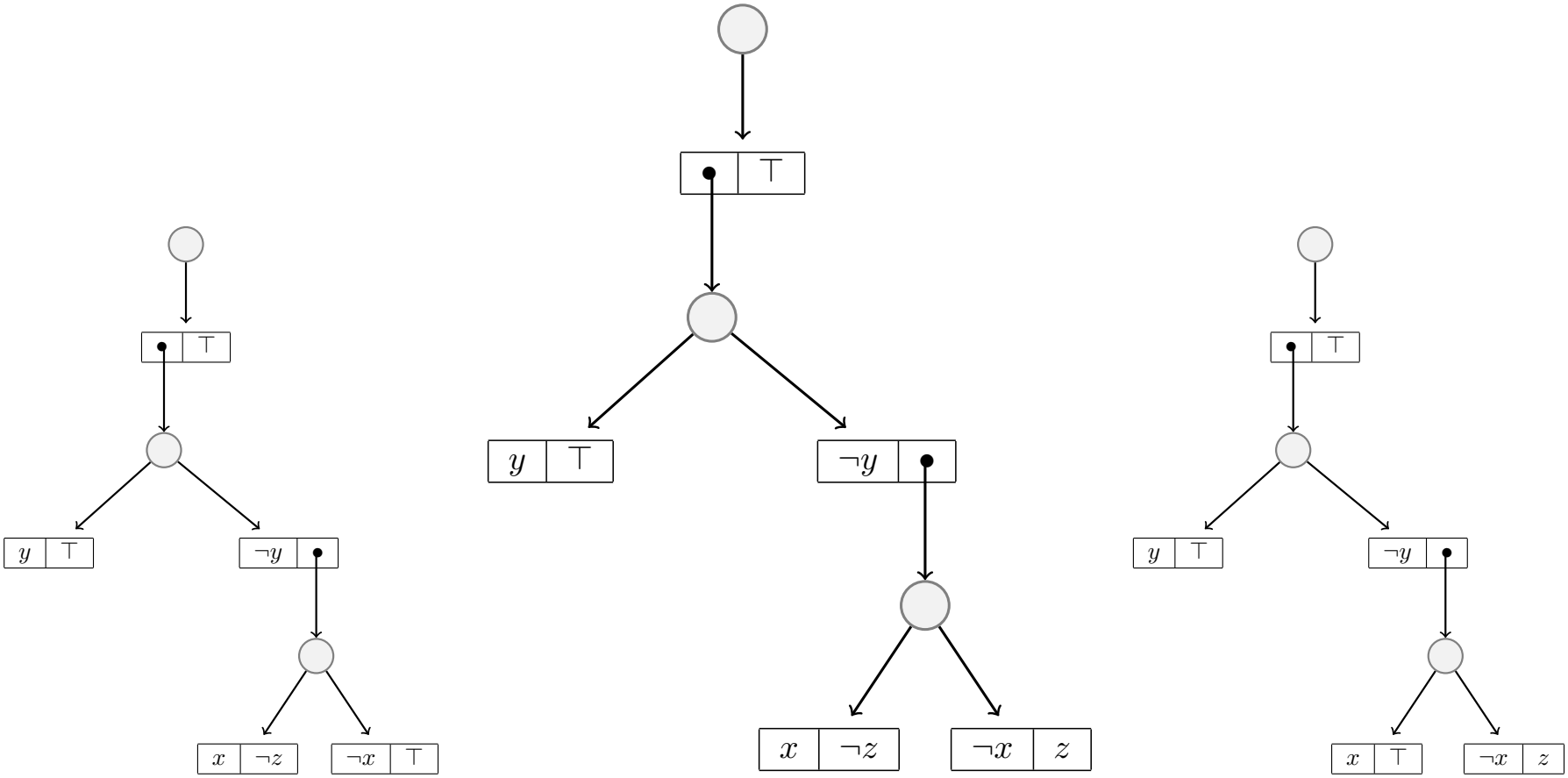
$$C_1 \wedge C_2 = \bigvee_{i,j} (p_i \wedge p'_j, s_i \wedge s'_j)$$

**Efficient Bottom-Up Execution:**

- **1. Intersect Primes:** Compute $p_i \wedge p'_j$. Because both respect the exact same left vtree branch, this is purely structural.
- **2. Filter Inconsistencies:** Discard any pair where $p_i \wedge p'_j \equiv \bot$. (By definition, SDD primes must be satisfiable).
- **3. Recursive Apply:** Recursively compute the new subs, $s_i \wedge s'_j$, down the right branch of the vtree.
- **4. Compress:** To maintain the canonical property, if multiple combinations yield the exact same sub $s$, merge their primes via $\vee$.

# Result: $C_1 \wedge C_2$

**Applying the cross-product:** $C_1 = (\neg x \vee y \vee \neg z)$ and $C_2 = (x \vee y \vee z)$



# KC Map for SDD: Queries

| $\mathscr{L}$ | CO | VA | CE | IM | EQ | SE | CT | ME |
|---|---|---|---|---|---|---|---|---|
| Circ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| CNF | ○ | √ | ○ | √ | ○ | ○ | ○ | ○ |
| DNF | √ | ○ | √ | ○ | ○ | ○ | ○ | √ |
| ODNF | √ | √ | √ | √ | √ | √ | √ | √ |
| MODS | √ | √ | √ | √ | √ | √ | √ | √ |
| decision − DNNF | √ | √ | √ | √ | **?** | ○ | √ | √ |
| FBDD | √ | √ | √ | √ | ○ | ○ | √ | √ |
| DT | √ | √ | √ | √ | √ | √ | √ | √ |
| EADT | √ | √ | √ | √ | **?** | ○ | √ | √ |
| ADT | √ | √ | √ | √ | √ | √ | √ | √ |
| $\text{OBDD}_<$ | √ | √ | √ | √ | √ | √ | √ | √ |
| SDD (Canon.) | √ | √ | √ | √ | √ | √ | √ | √ |
| SDD | √ | √ | √ | √ | √ | √ | √ | √ |

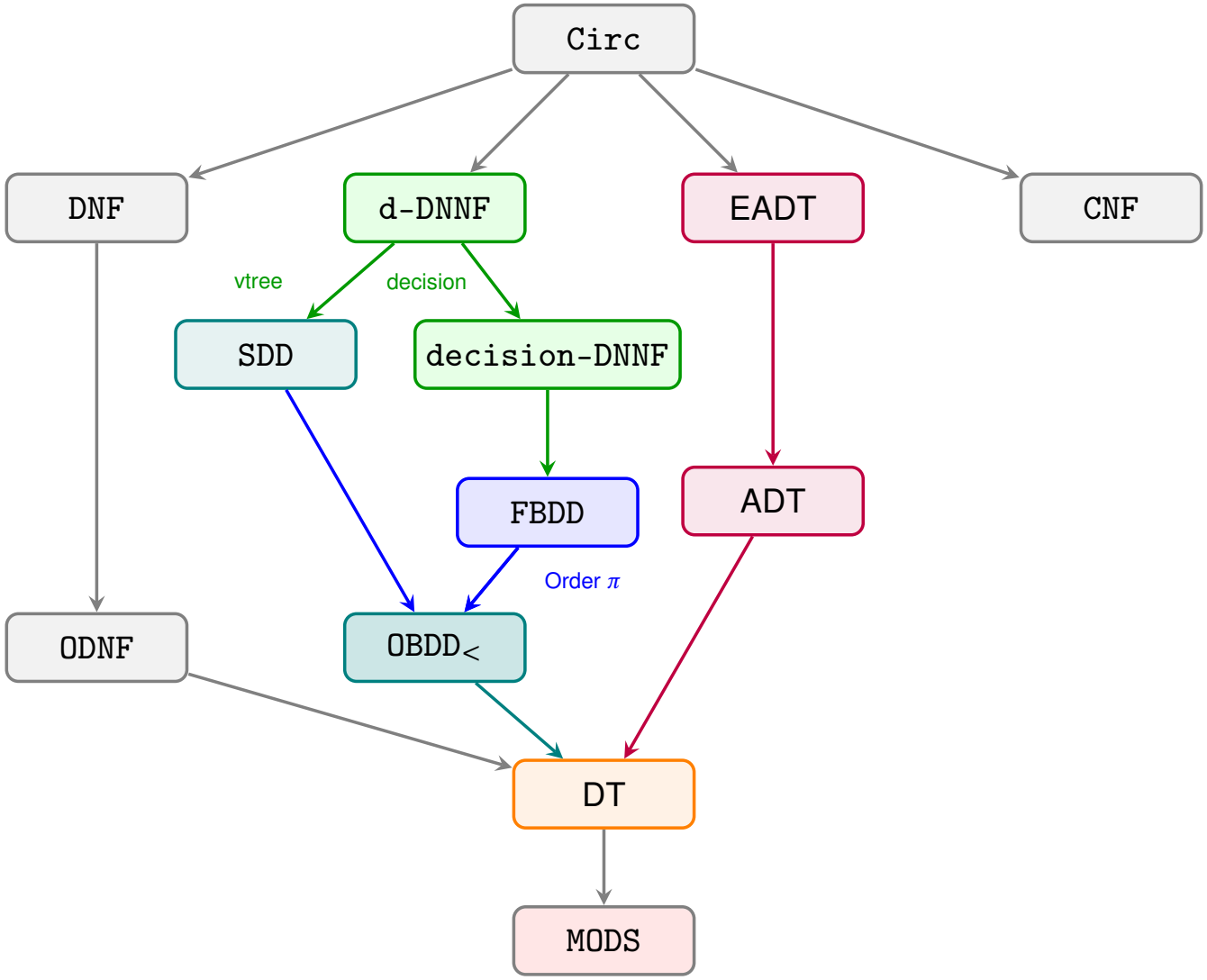
- **The Ultimate Query Language:** Because SDDs structured on the same vtree are Canonical, they easily inherit the perfect query profile of $\text{OBDD}_<$.
- They successfully answer the open **EQ** question that plagues decision-DNNF.

# KC Map for SDD: Transformations

| $\mathscr{L}$ | CD | FO | SFO | $\wedge$C | $\wedge$BC | $\vee$C | $\vee$BC | $\neg$C |
|---|---|---|---|---|---|---|---|---|
| Circ | √ | ○ | √ | √ | √ | √ | √ | √ |
| CNF | √ | ○ | √ | √ | √ | ○ | √ | ○ |
| DNF | √ | √ | √ | ○ | √ | √ | √ | ○ |
| ODNF | √ | ○ | ○ | ○ | √ | ○ | ○ | ○ |
| MODS | √ | √ | √ | ○ | √ | ○ | √ | ○ |
| decision − DNNF | √ | ○ | ○ | ○ | ○ | ○ | ○ | **?** |
| FBDD | √ | ○ | ○ | ○ | ○ | ○ | ○ | √ |
| DT | √ | ○ | √ | ○ | √ | ○ | √ | √ |
| EADT | √ | ○ | ○ | ○ | ○ | ○ | ○ | √ |
| ADT | √ | ○ | √ | ○ | √ | ○ | √ | √ |
| $\text{OBDD}_{<}$ | √ | ○ | √ | ○ | √ | ○ | √ | √ |
| SDD (Canon.) | ○ | ○ | ○ | ○ | ○ | ○ | ○ | √ |
| SDD | √ | ○ | √ | ○ | √ | ○ | √ | √ |

- Applying a Boolean operator to two canonical SDDs can cause an **exponential blow-up** when attempting to re-establish canonicity.

# Succinctness Hierarchy




- **The Core Takeaway:** The more rigid the structure (vtree, linear ordering), the better the queries, but the exponentially larger the file size!

# Alternative KC Languages

## The Tractability vs. Succinctness Trade-off

While this talk focuses on languages that support both **conditioning** and **model counting** in polynomial time (like `d-DNNF`), the broader Knowledge Compilation map includes many other target languages.

These alternatives often trade strict query tractability for greater succinctness, supporting only satisfiability or restricted forms of conditioning.

**Notable Examples:**

- **Symmetric KC:** Exploits symmetries in the formula to compress the representation, which requires adapting the standard query algorithms (Bart et al., 2014).
- **Weak variants of `d-DNNF`:** Relaxes strict structural properties (like decomposability), leading to smaller circuits but limiting the scope of polynomial-time queries (Illner and Kucera, 2024).

# Outline

# d-DNNF **Queries**

# The Versatility of d-DNNF

## The d-DNNF Representation: definitions

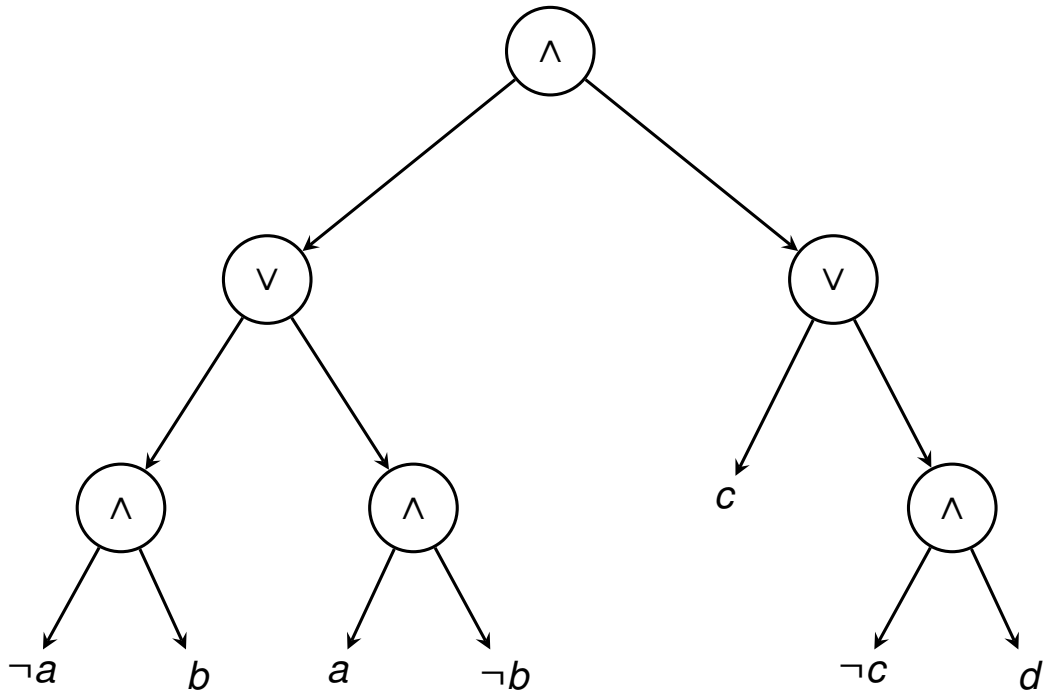
- **d-DNNF:** Deterministic Decomposable Negation Normal Form (a rooted DAG).
- **Decomposability ($\wedge$):** Children's sub-circuits have pairwise disjoint variables.
- **Determinism ($\vee$):** Children are pairwise logically inconsistent.

## A Tractable Framework

- **Polynomial Tractability:** The d-DNNF language is highly expressive and allows many hard reasoning tasks to be executed in time linear to the circuit size.
- **Core Queries:** It efficiently supports Satisfiability, Model Counting (#SAT), Model Enumeration (AllSAT), Direct Access, and Uniform Sampling.
- **Conditioning ($\Phi[E]$):** Practical applications often require executing these queries multiple times under varying contexts (evidence sets $E$).
- **Lazy Evaluation:** Rather than explicitly rewriting the DAG structure for each new query, conditioning is achieved *implicitly*. We assign a weight of 0 to any literal leaf falsified by $E$.

# Running Examples

Consider the CNF formula: $\Psi = \{a \vee b, \neg a \vee \neg b, a \vee c \vee d, b \vee c \vee d\}$
With variables: $Var(\Psi) = \{a, b, c, d\}$.

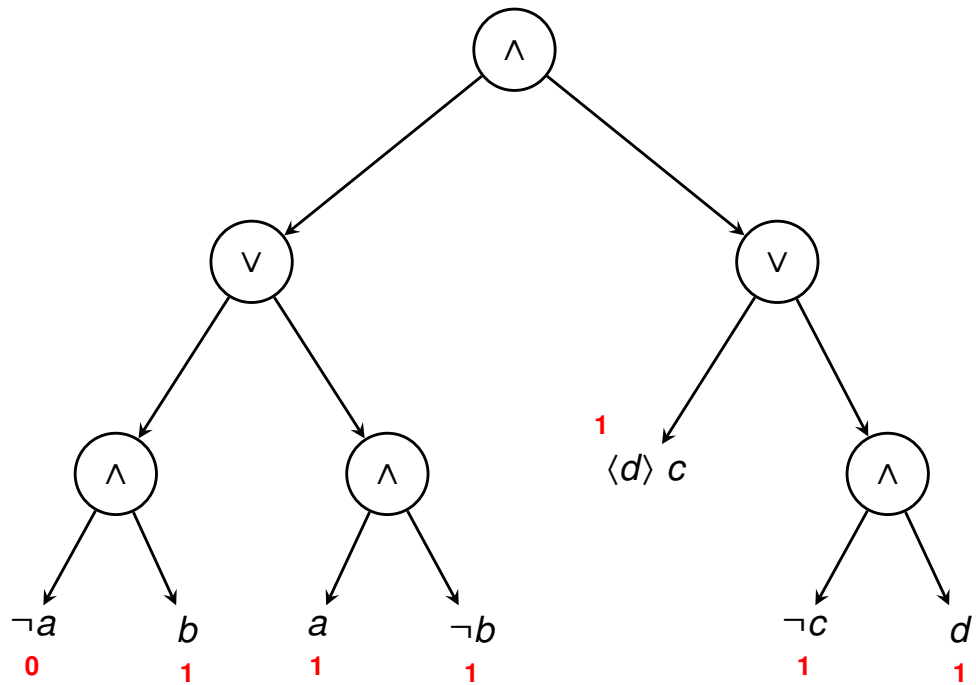


# Outline

# Efficient Querying via Conditioning

- **Implicit Conditioning:** Instead of altering the DAG structure, falsified leaves under evidence $E$ are dynamically assigned a weight of 0.
- **Algebraic Evaluation:** Zero weights propagate bottom-up, pruning invalid branches efficiently.
- **Dynamic Smoothness:** Missing variables in a node's subgraph are tracked on-the-fly, avoiding memory overhead.

Let us consider $E = \{a\}$:



# Outline

# Satisfiability (SAT)

## Definitions

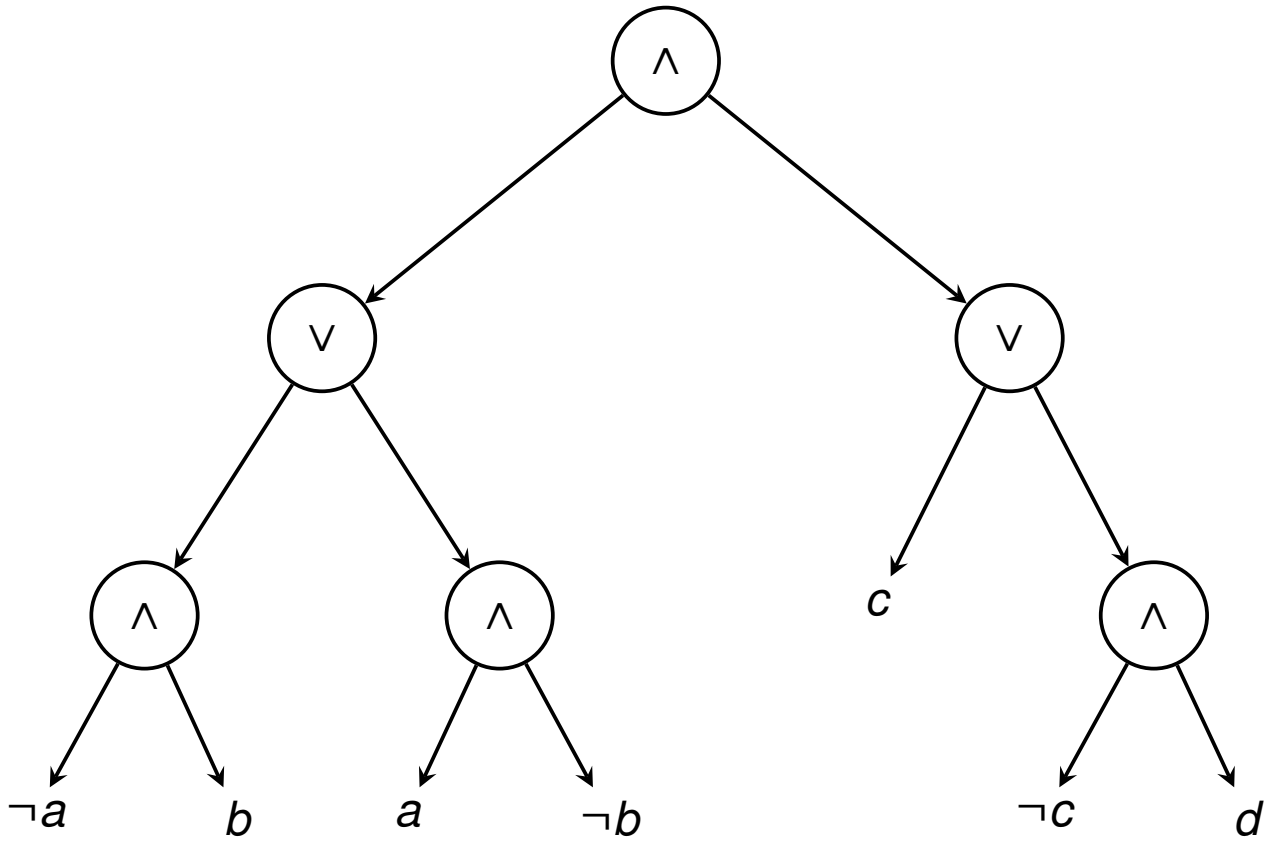
- **Goal:** Check if a formula is consistent under context $E$ ($\mathcal{M}(\Phi[E]) \neq \varnothing$).
- **Mechanism:** Single top-down traversal. Falsified leaves evaluate to 0 ($\bot$), consistent ones to 1 ($\top$).
- **Complexity:** Strictly linear relative to the circuit size $\mathcal{O}(|\Phi|)$.

## Algorithm: SAT($\Phi, E$)

```
1  SAT(Φ, E):
2    if Φ == ⊥:
3      return false
4    if Φ is a literal ℓ:
5      if ¬ℓ ∈ E
6        return false
7      return true
8    if Φ == Φ1 ∨ Φ2:
9      return SAT(Φ1, E) ∨ SAT(Φ2, E)
10   if Φ == Φ1 ∧ Φ2:
11     return SAT(Φ1, E) ∧ SAT(Φ2, E)
12   return true   // Φ == ⊤
```

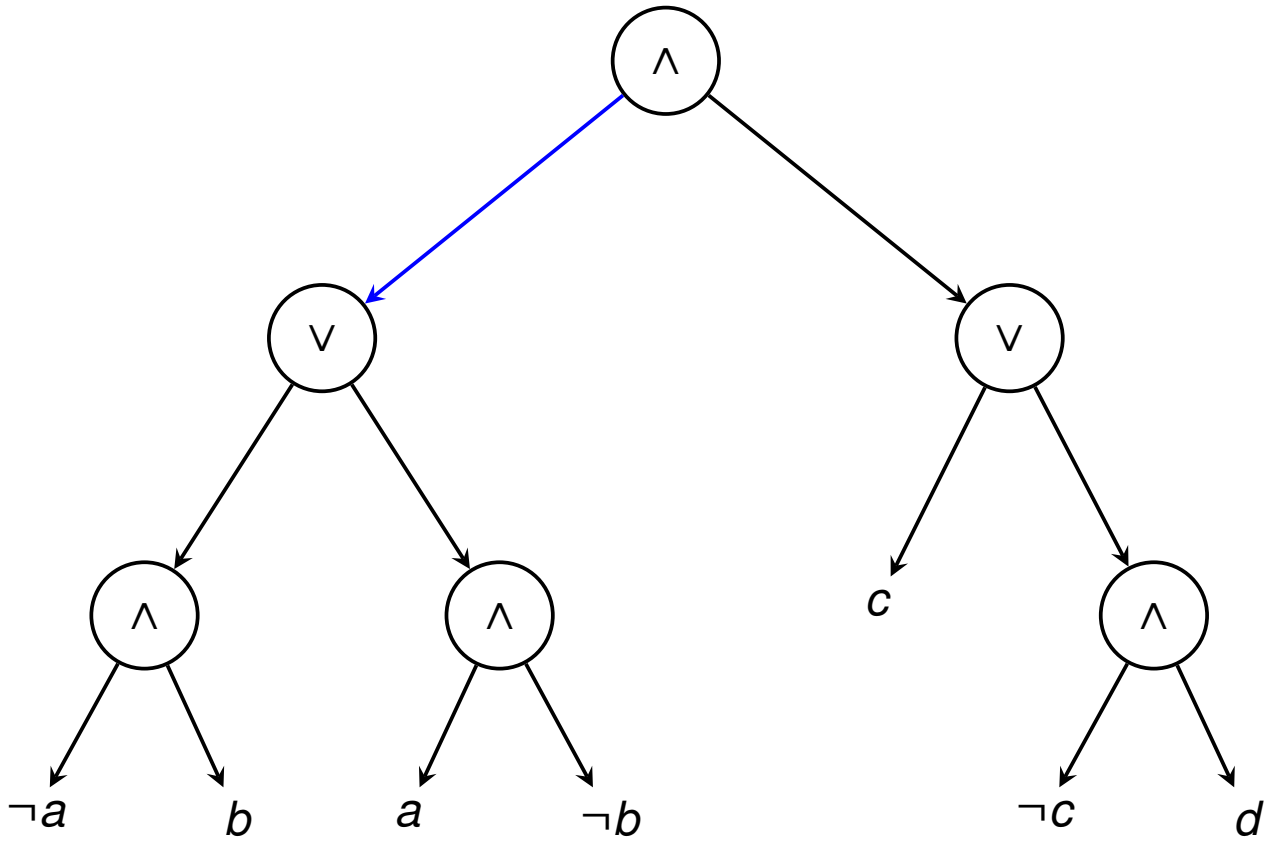
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



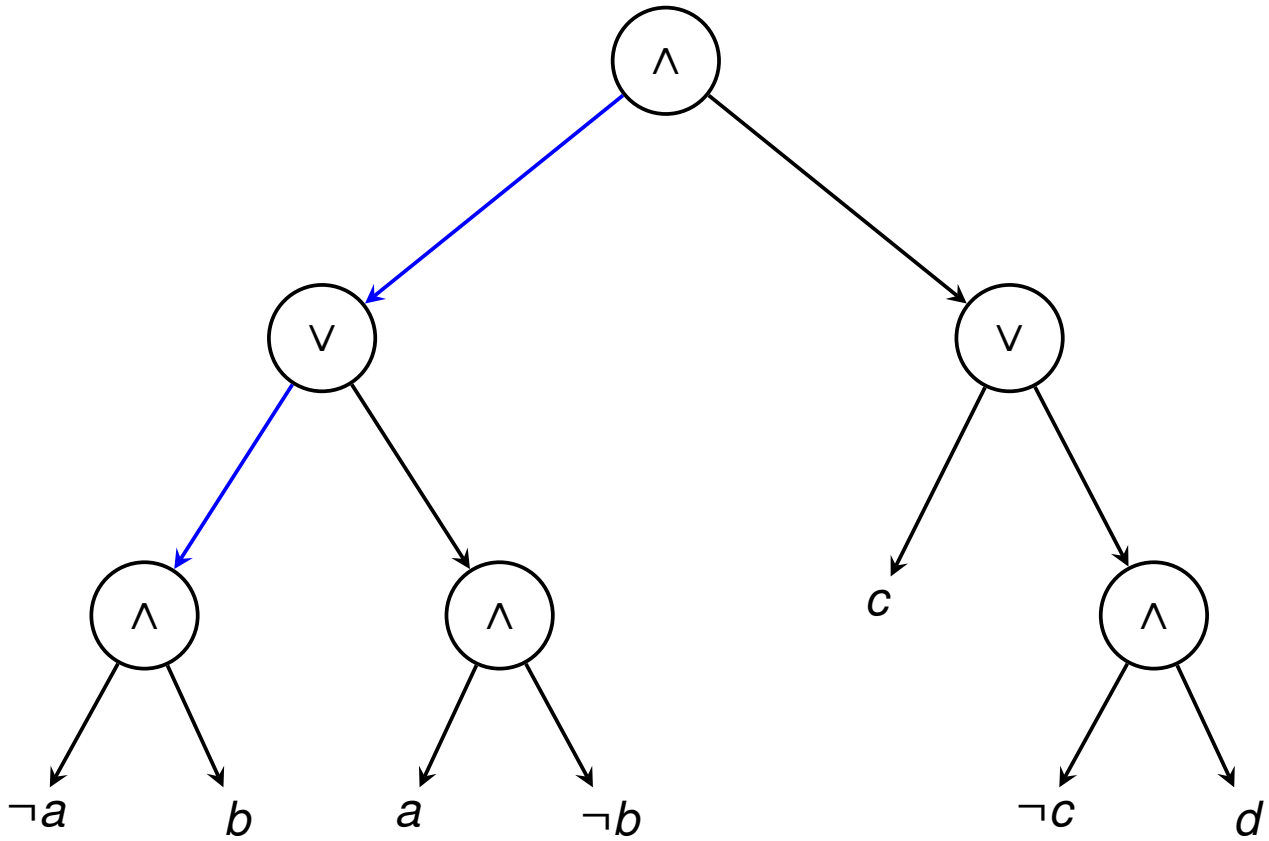
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



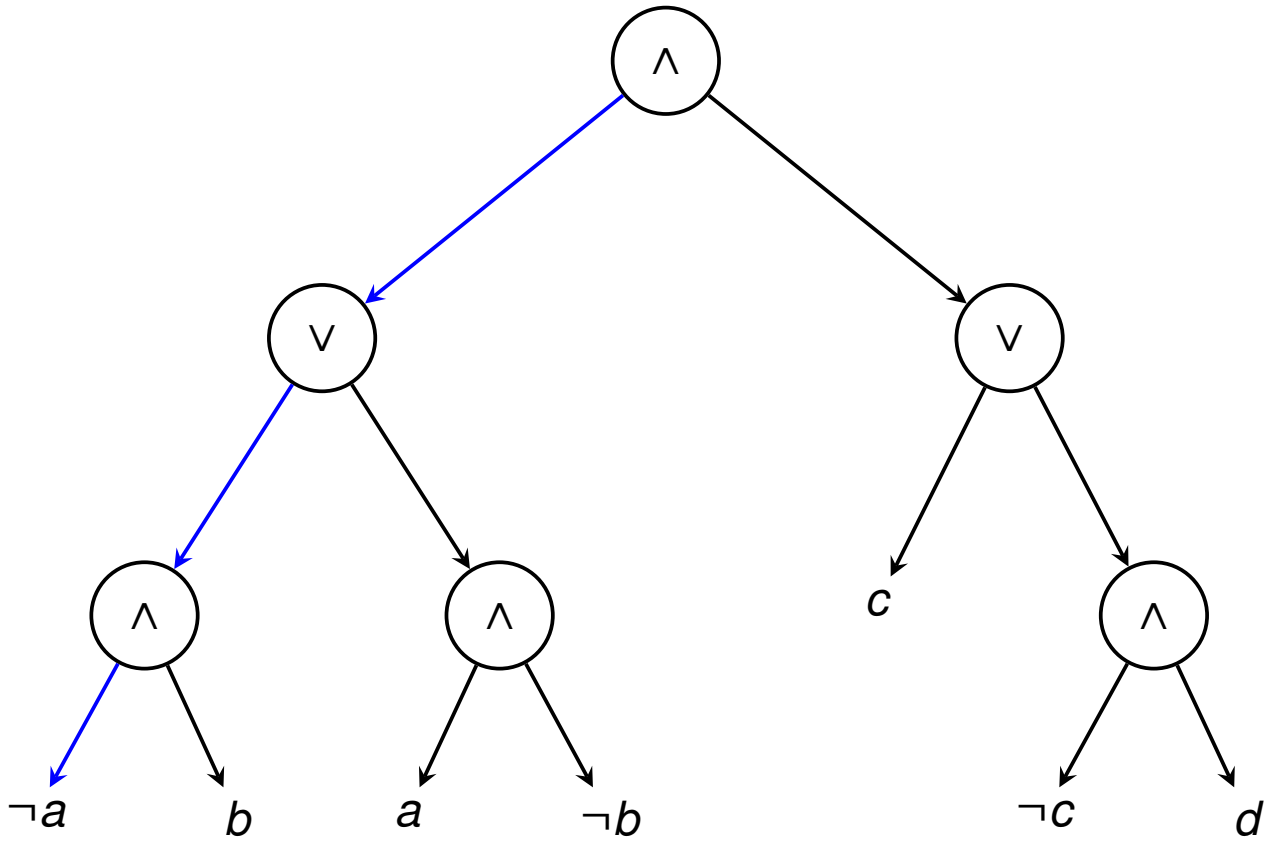
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



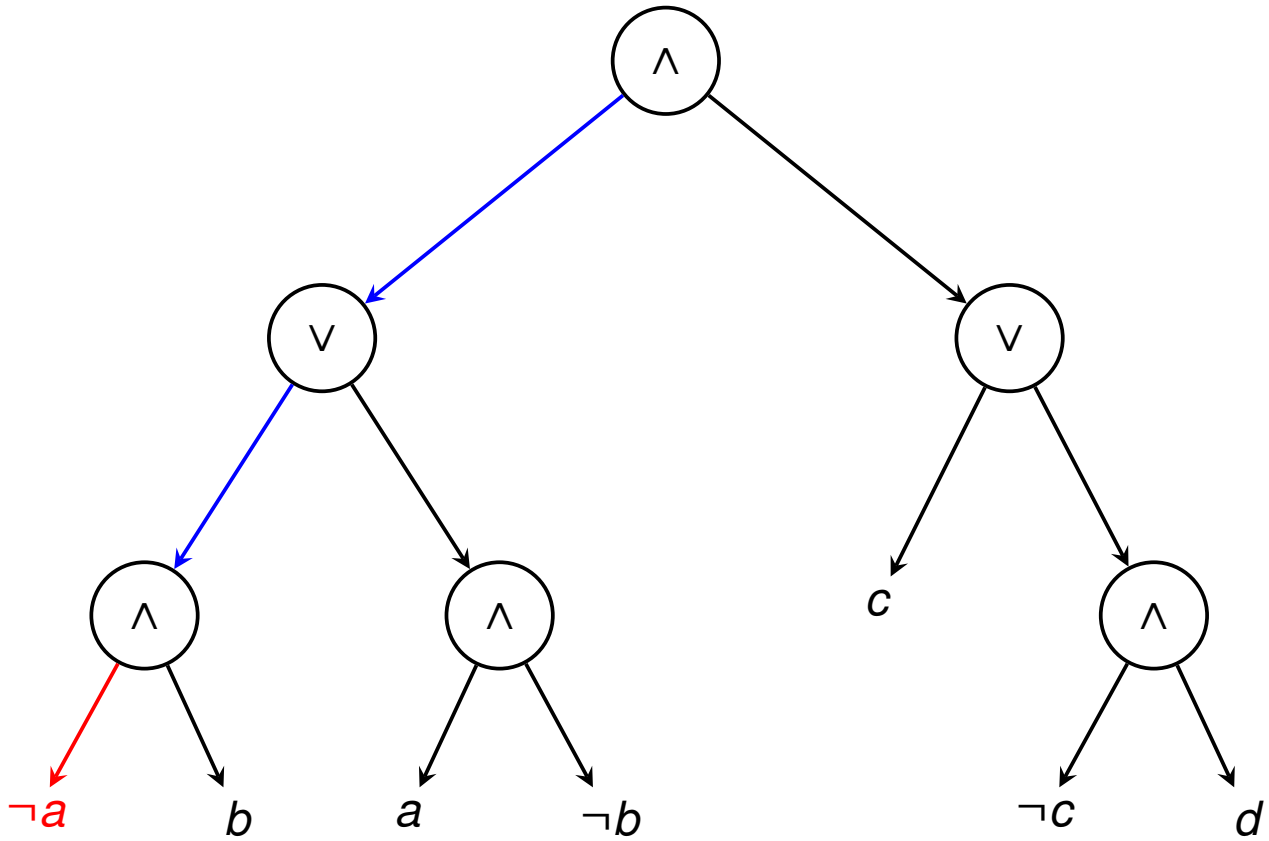
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



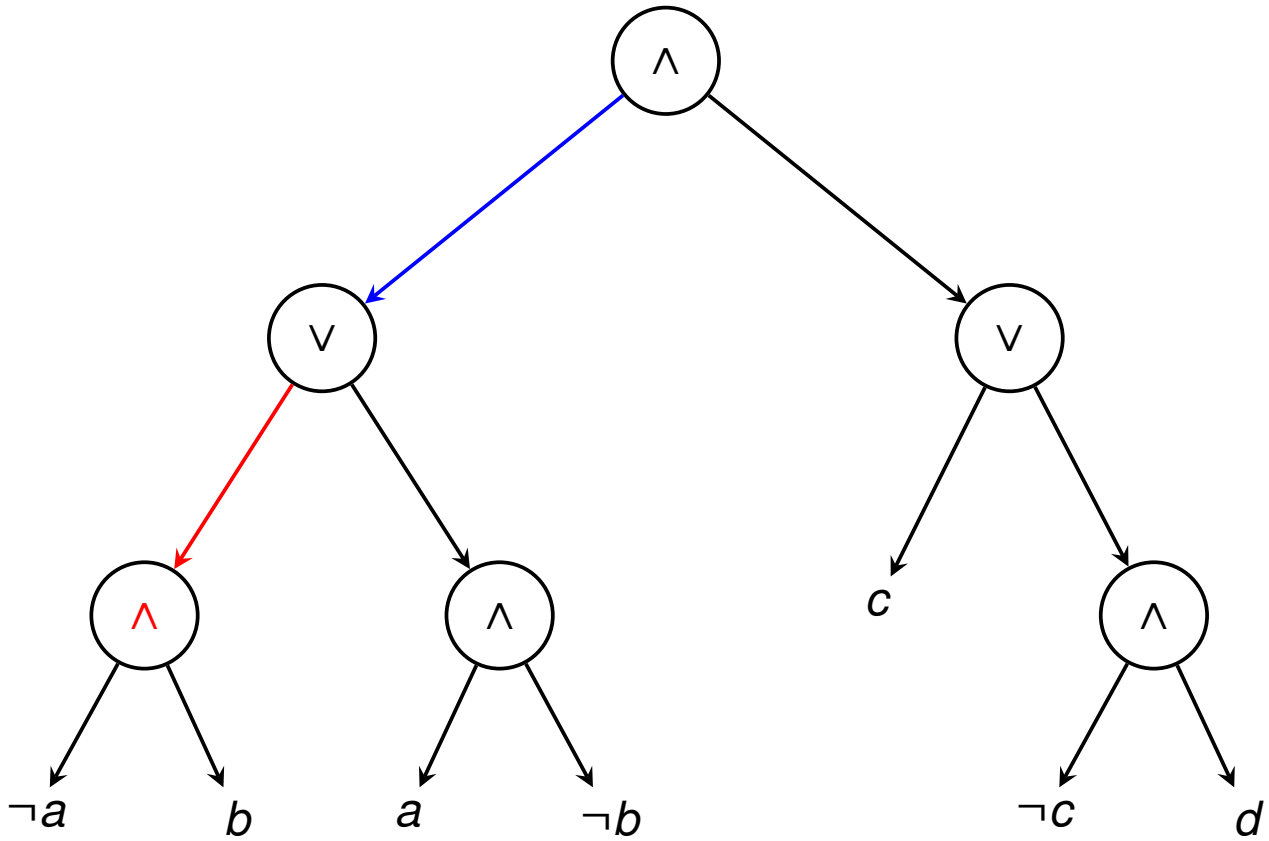
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



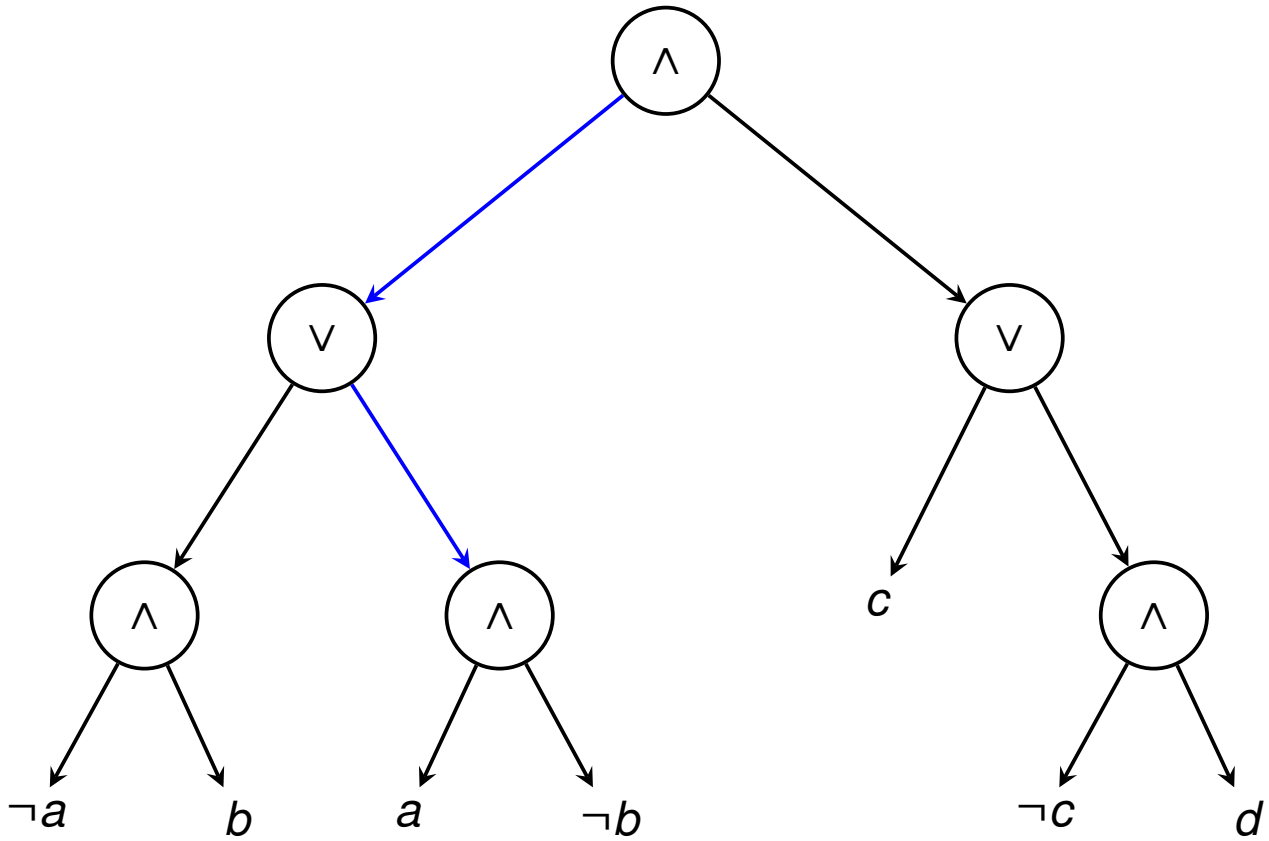
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



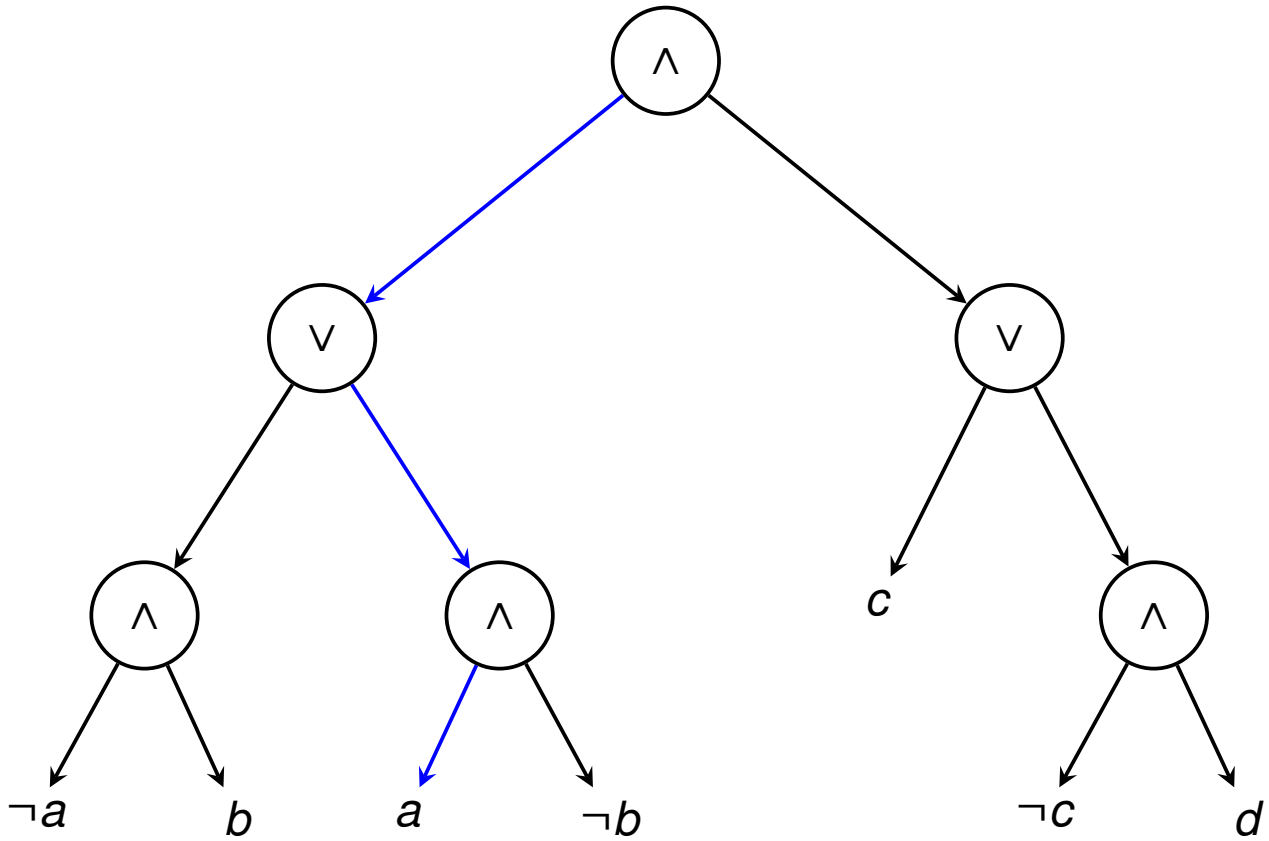
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



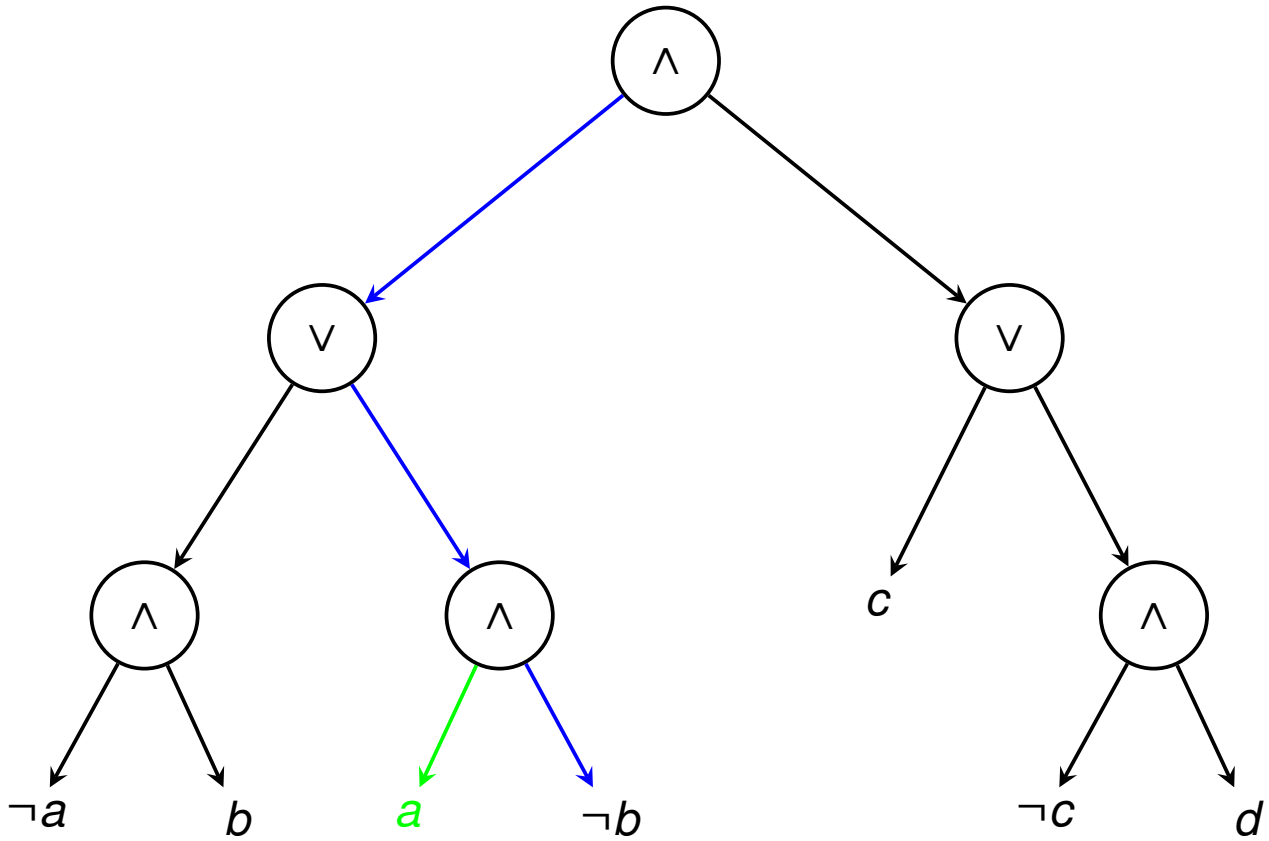
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



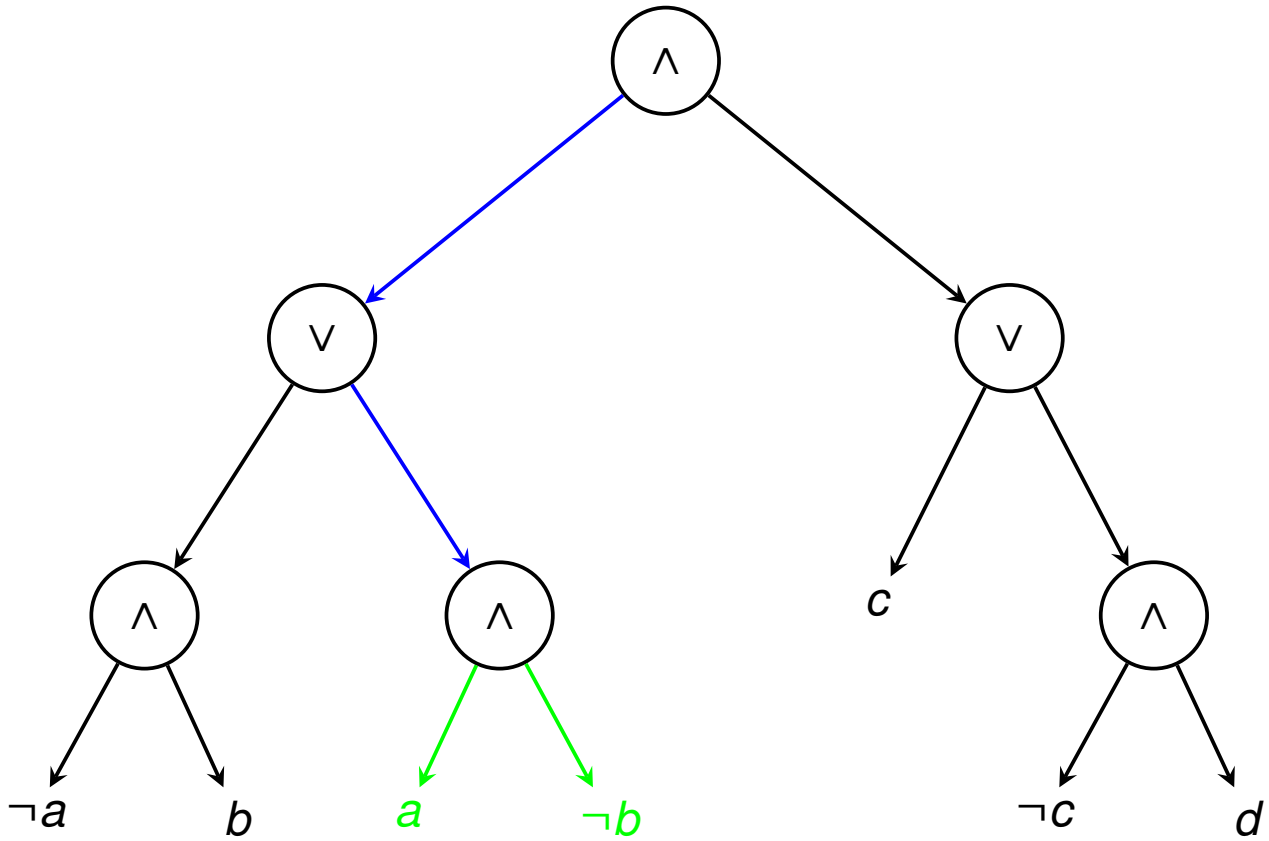
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



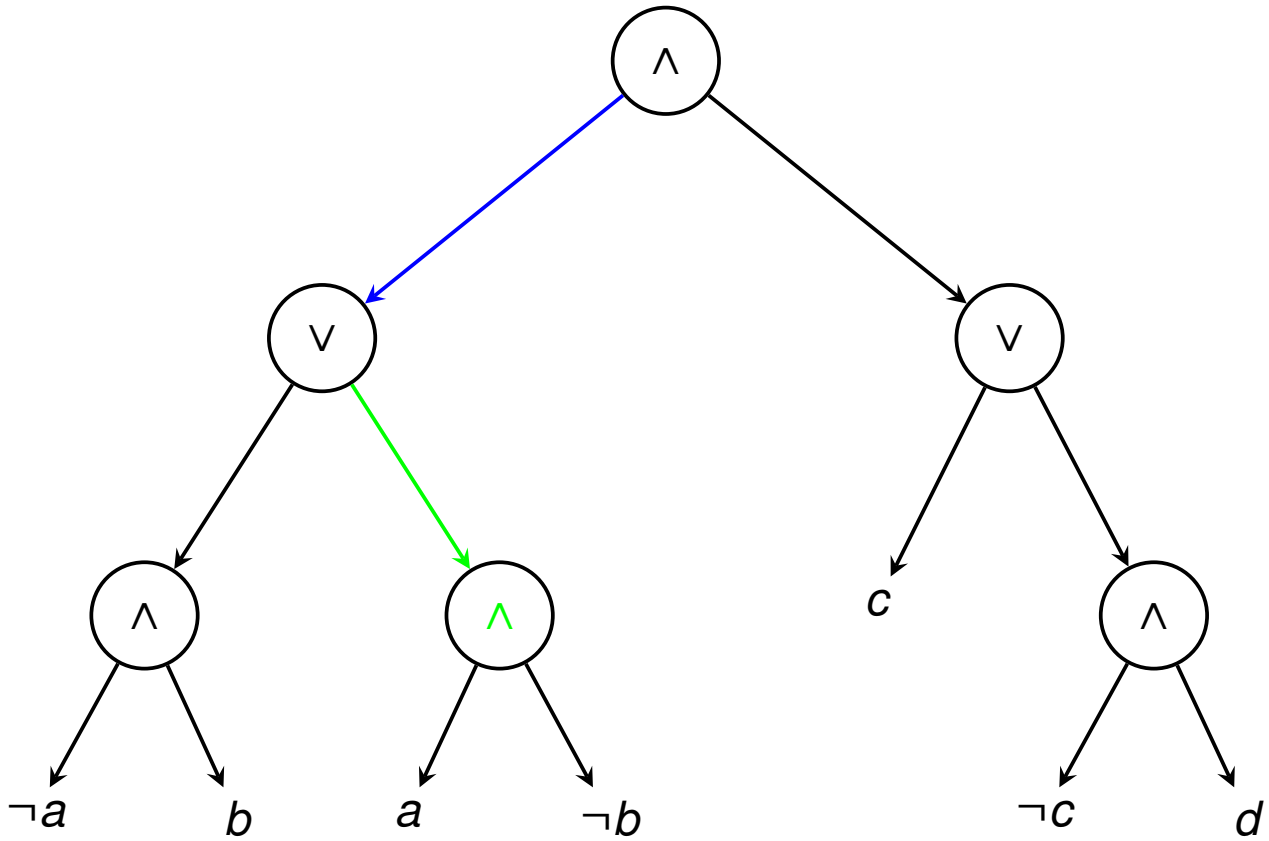
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



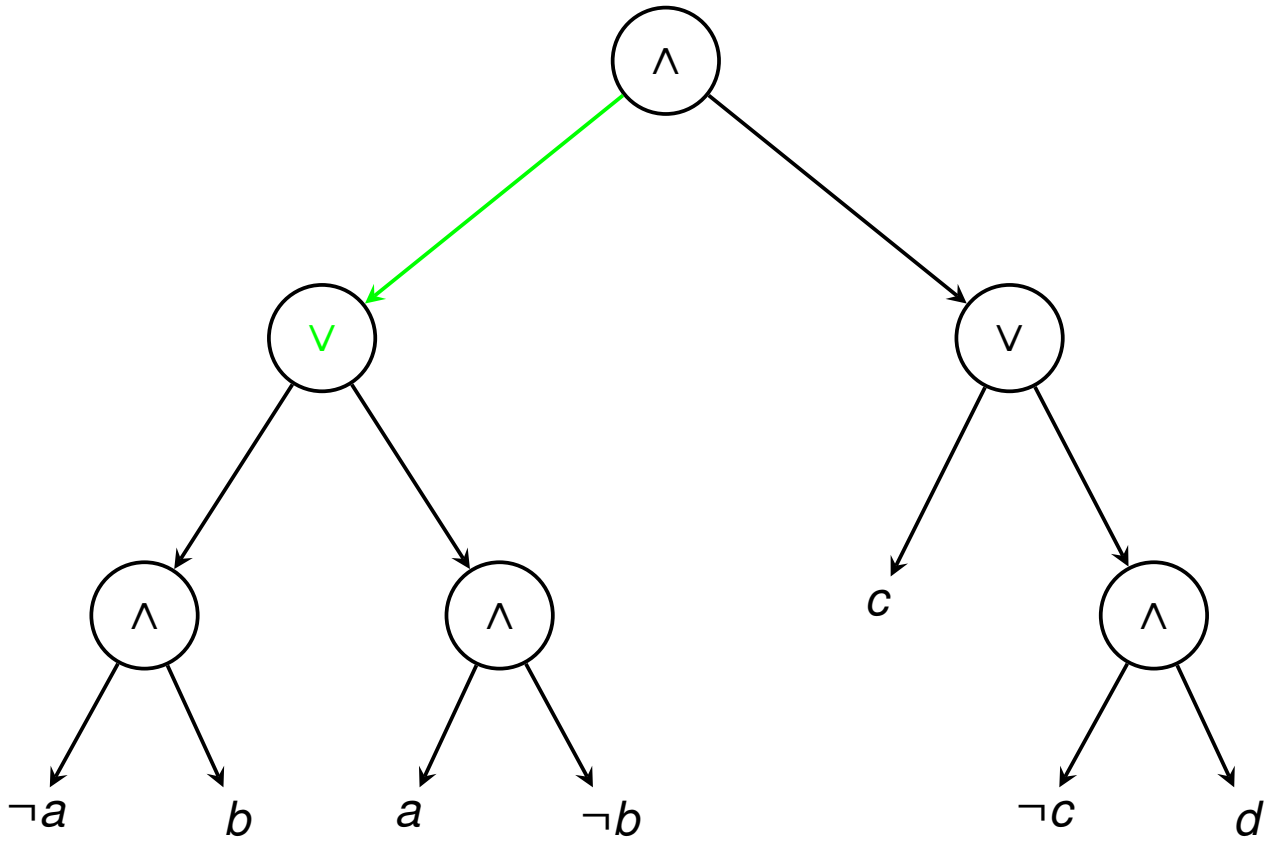
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



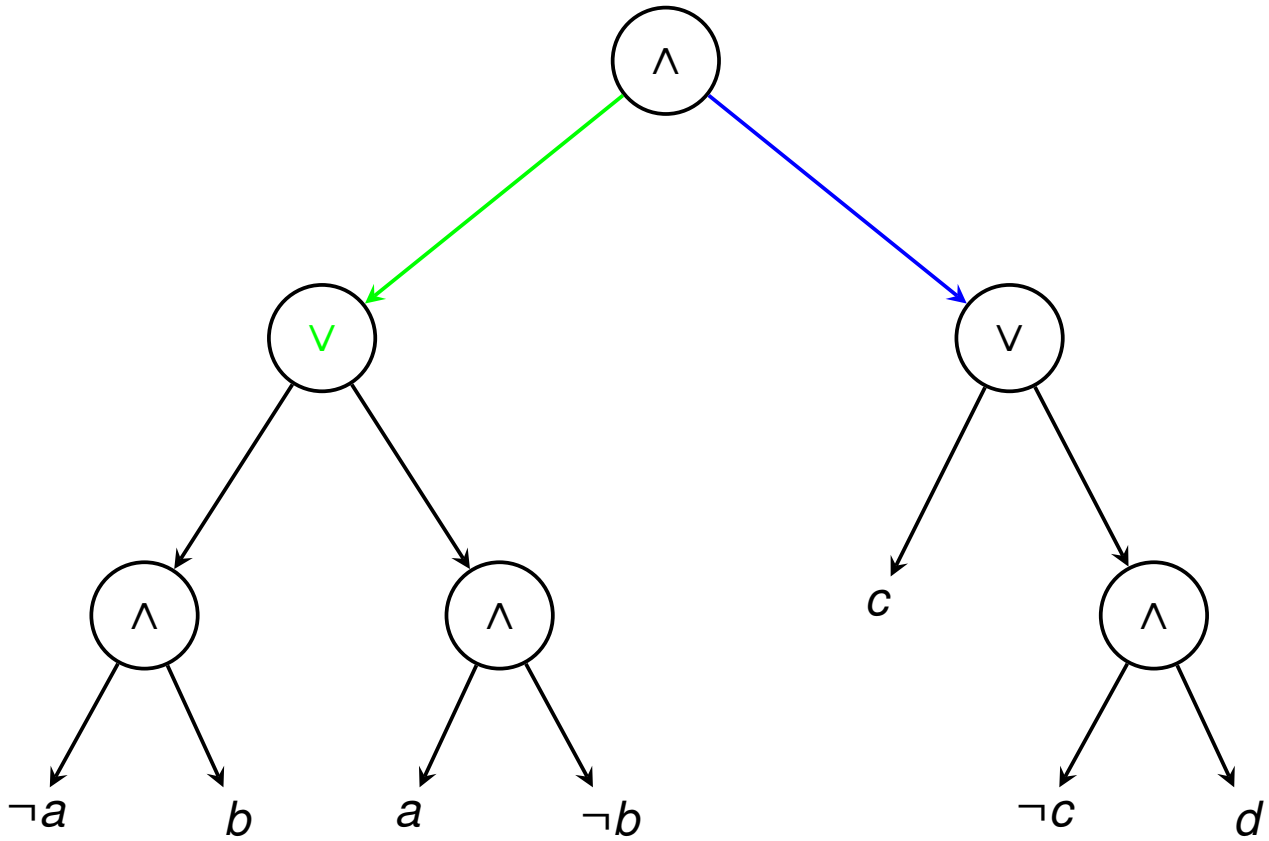
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



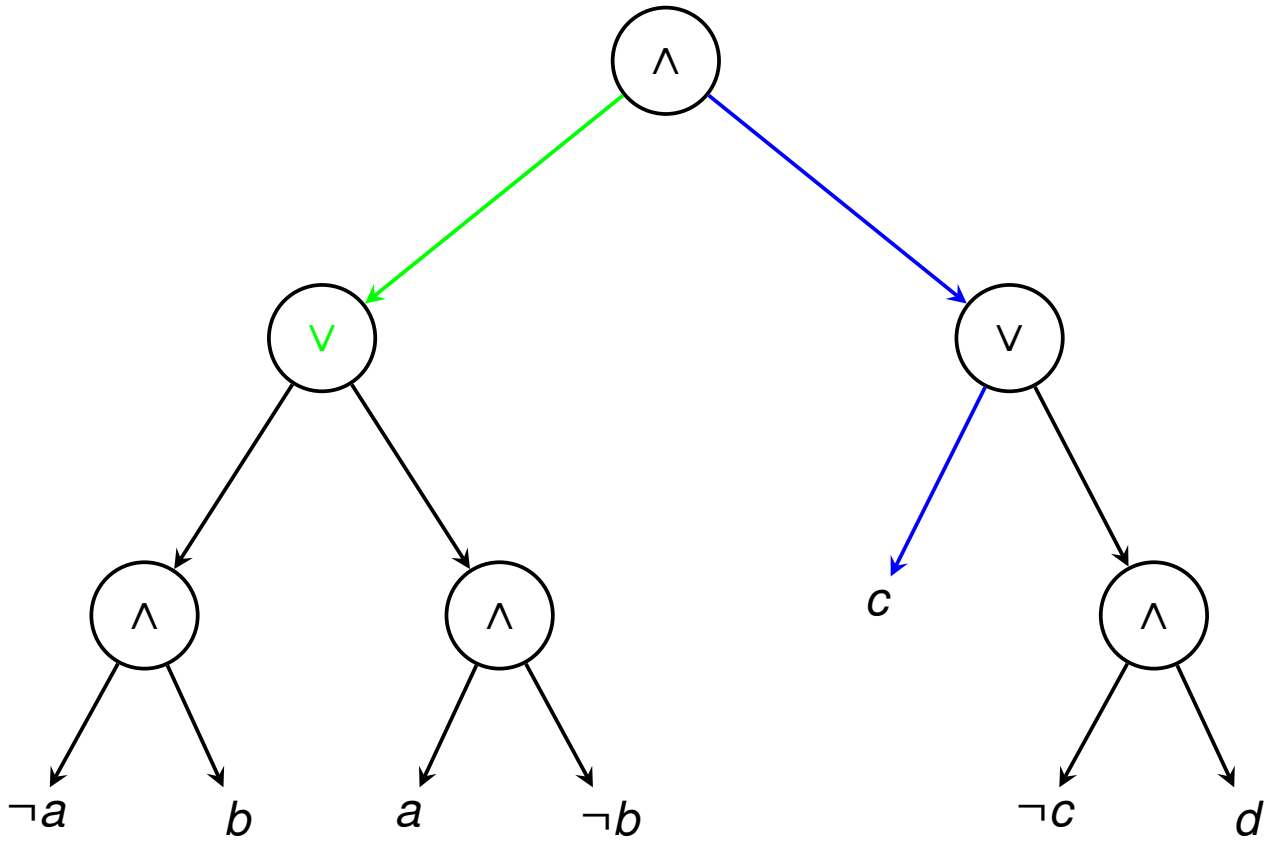
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



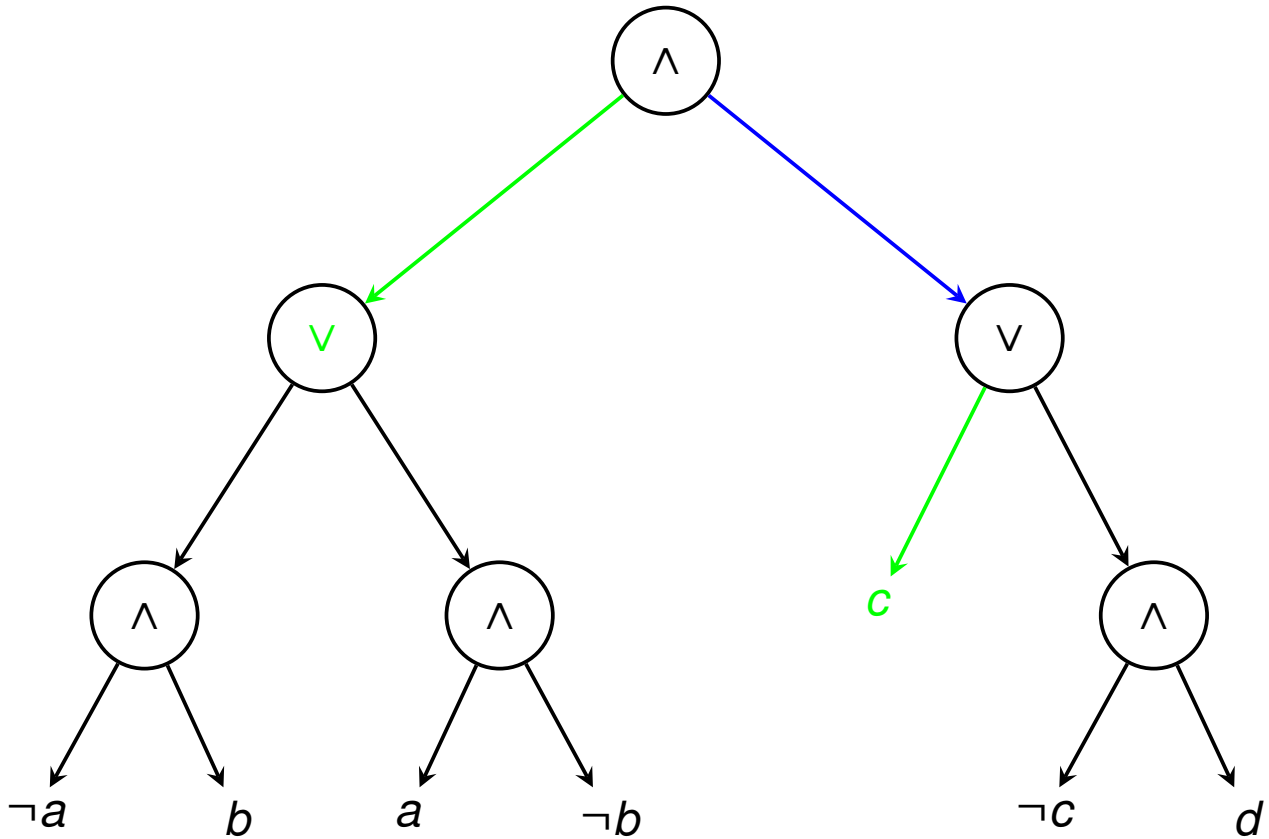
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



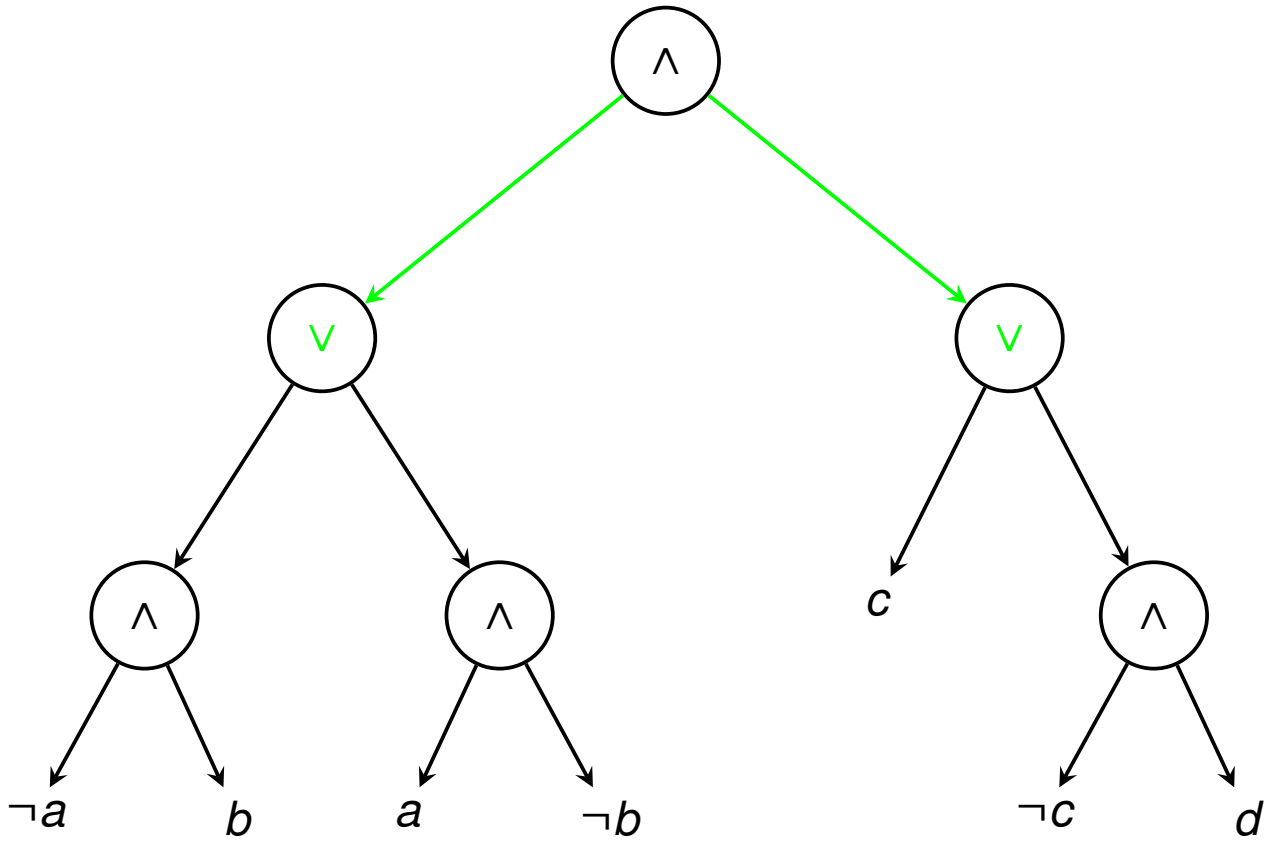
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



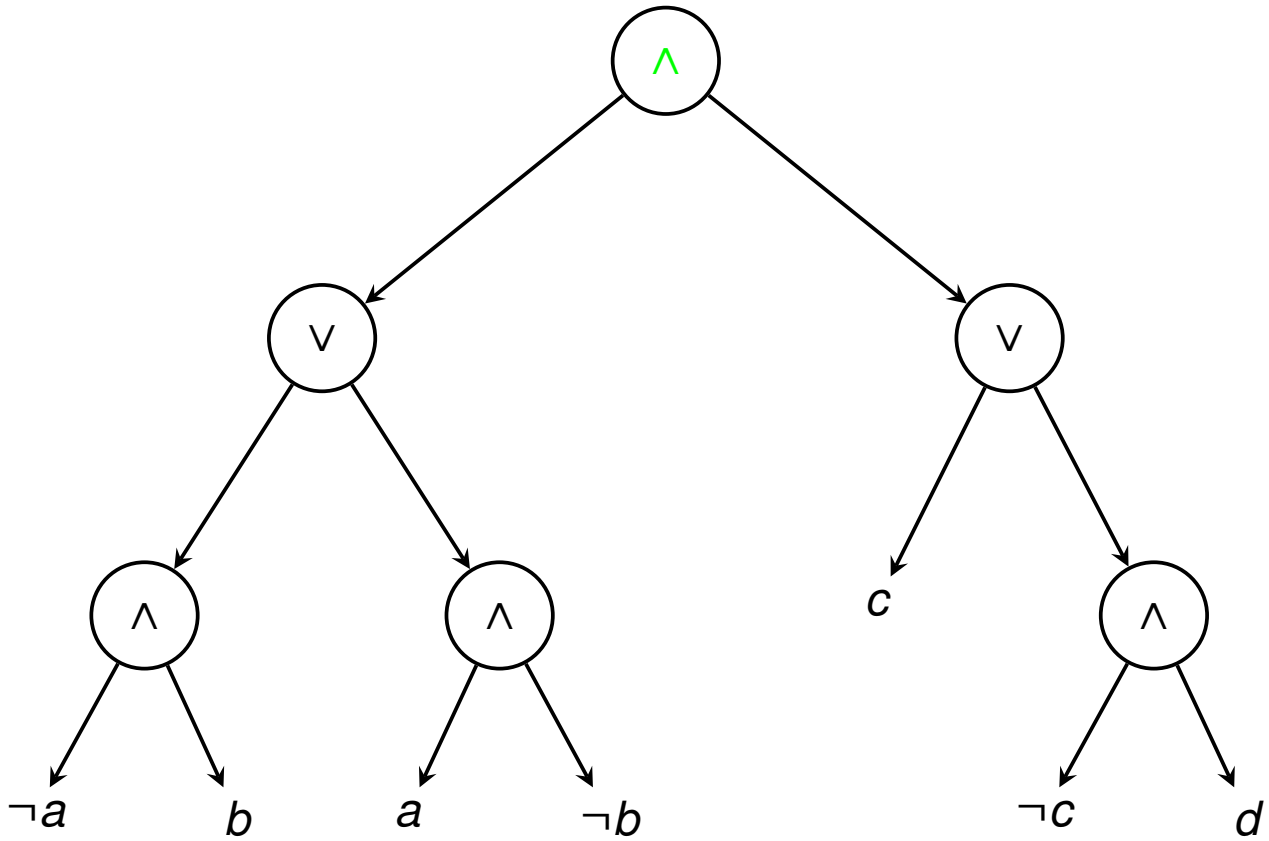
# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



# Running Example: SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



# Outline

# Model Enumeration (AllSAT)

## Definitions

- **Goal:** Generate all satisfying assignments without redundancy.
- **Complete vs. Partial:** Complete models assign a value to every variable. Partial models (cubes) leave some unassigned, compactly representing $2^{|Var(\Phi)|-|X|}$ complete models.
- **Mechanism:** Extracts disjoint paths from root to consistent leaves.

## Algorithm: `baseline-enum`$(\Sigma, \mu)$

```
1  // Input: a d-DNNF Σ, μ a partial assignment
2  // Output: Δ the collected set of models
3    if Σ ≡ ⊥:
4      return ∅
5    if Σ ≡ ⊤:
6      return {μ}
7    else:
8      Let x ∈ Var(Σ)
9      return baseline-enum(Σ[x], μ∪{x}) ∪ baseline-enum(Σ[¬x], μ∪{¬x})
```

- The algorithm complexity is $O(|Var(\Sigma)| \times |\Sigma| \times |Mod(\Sigma)|)$
- Performance is limited due to frequent querying and transformations of the `d-DNNF` formula

# Model Graph Enumeration (Lagniez and Lonca, 2024)

- Objective: Develop an algorithm that operates within the size constraints of the `d-DNNF` and outputs models with a polynomial delay
- A bottom-up algorithm is impractical, it requires memoizing all models, which may exceed the space constraints of the `d-DNNF` circuit
- A Depth First Search (DFS) approach, is a viable alternative:
  - Start at the root and propagate to both children if the node is a conjunction ($\wedge$)
  - For disjunctions ($\vee$), propagate to only one child
- Literals found during traversal form the model
- The DFS trace represents a sub-graph of the d-DNNF:
  $\Rightarrow$ **model graph**

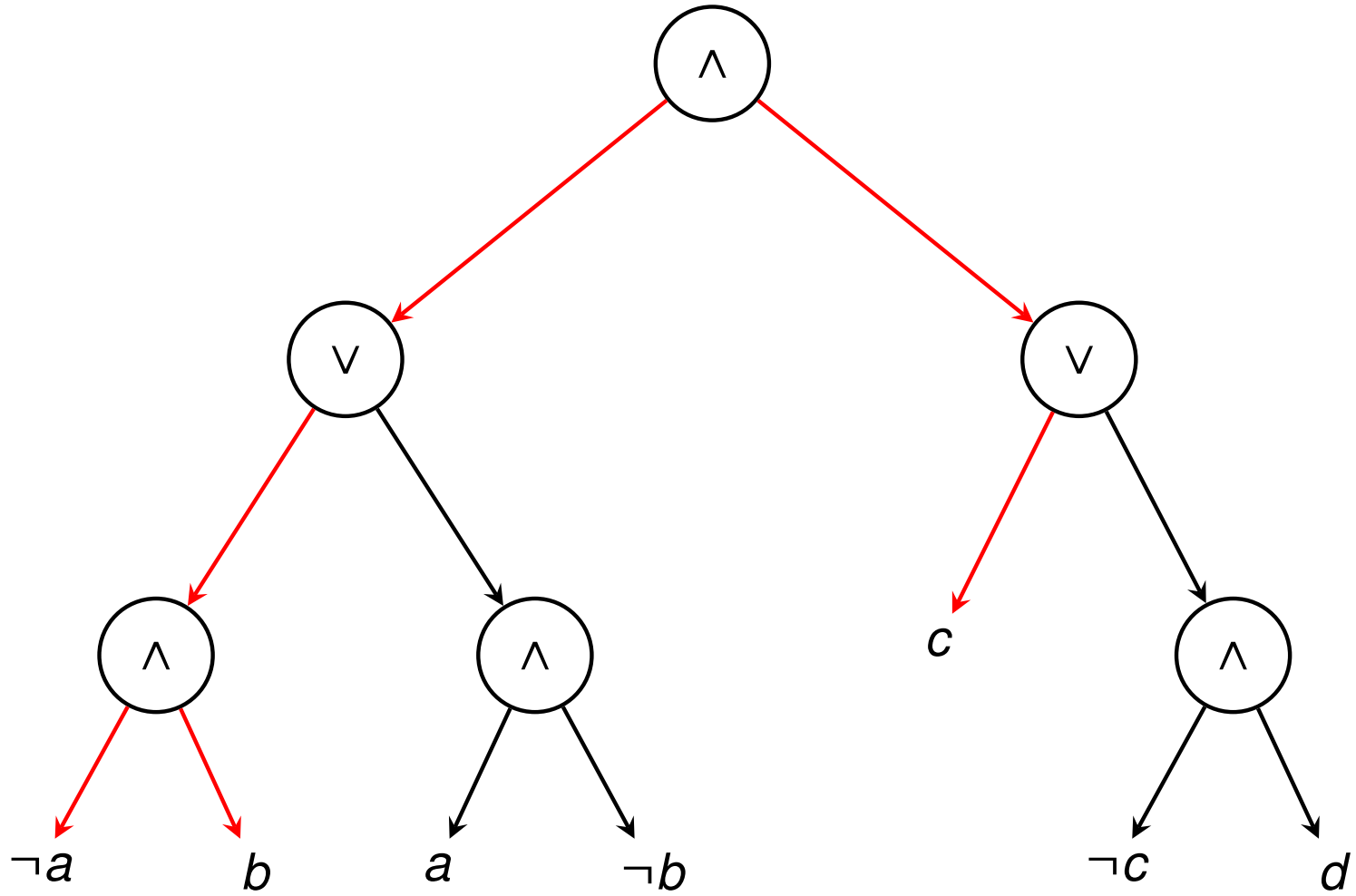
# Definition: Model Graph

Let $\Sigma = \langle V, E \rangle$ be a (smooth) d-DNNF rooted at $v_{\text{root}} \in V$. A model graph $\omega = \langle V', E' \rangle$ is a subgraph of $\Sigma$ that forms a DAG, rooted at $v_{\text{root}}$, with the following conditions:

- For each $v \in V'$, let $E_v \subseteq E$ be the set of edges where $v$ is the source: $E_v = \{e_i = (v, v_i) \mid (v, v_i) \in E\}$
- If $v \in V'$ is labelled with $\wedge$, then $E_v \subseteq E'$ and $\{v_i\} \subseteq V'$
- If $v \in V'$ is labelled with $\vee$, there exists exactly one $(v, v_i) \in E_v$ where $E_v \cap E' = \{(v, v_i)\}$ and $v_i \in V'$

**Note:** Nodes labelled $\wedge$ have exactly two children, nodes labelled $\vee$ have one child, and literal nodes have none.

# Example: Model Graph




$\{a, \neg b, c\}$

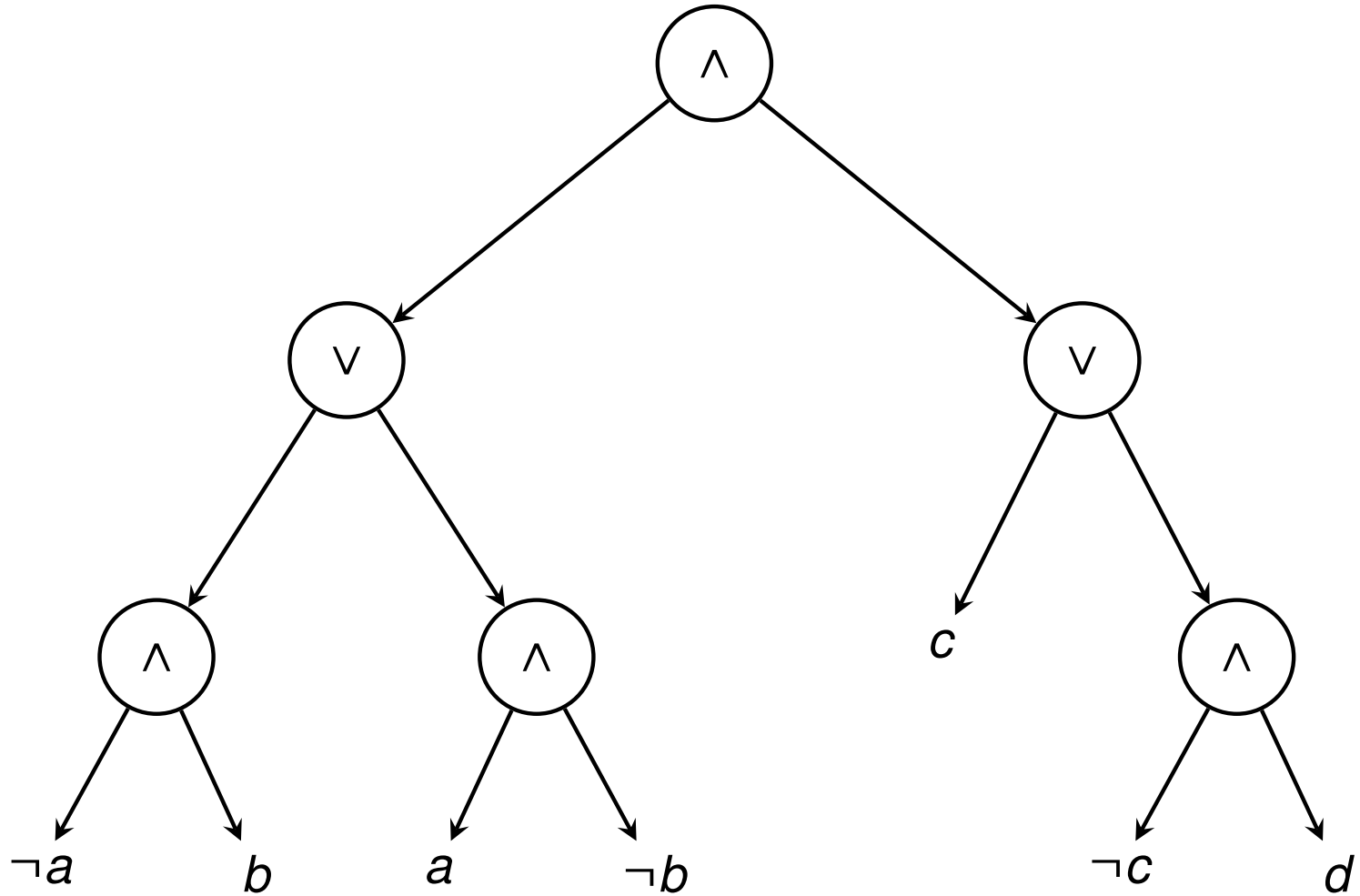
# Begin and End of Model Graph Enumeration

- We introduce the notion of model graph to enumerate in a disjoint manner all models of a `d-DNNF` $\Sigma$ rooted at $n$
- The model graphs are ordered and iterated through this order using two recursive functions:
  - **begin**$(\Sigma)$: Determines the first model graph
  - **end**$(\Sigma)$: Determines the last model graph
- If $n$ is a leaf node, both **begin**$(\Sigma)$ and **end**$(\Sigma)$ return $\langle n, \varnothing \rangle$ (only one model graph)
- For or-nodes, **begin**$(\Sigma)$ starts with the leftmost subgraph, and **end**$(\Sigma)$ with the rightmost one:
  - **begin**$(\Sigma) = \langle \{n\}, \{(n, \text{left}(n))\} \rangle \cup \text{begin}(\text{sub}(\Sigma, \text{left}(n)))$
  - **end**$(\Sigma) = \langle \{n\}, \{(n, \text{right}(n))\} \rangle \cup \text{end}(\text{sub}(\Sigma, \text{right}(n)))$
- For and-nodes, both functions traverse the left and right subgraphs

# Defining the Next Model

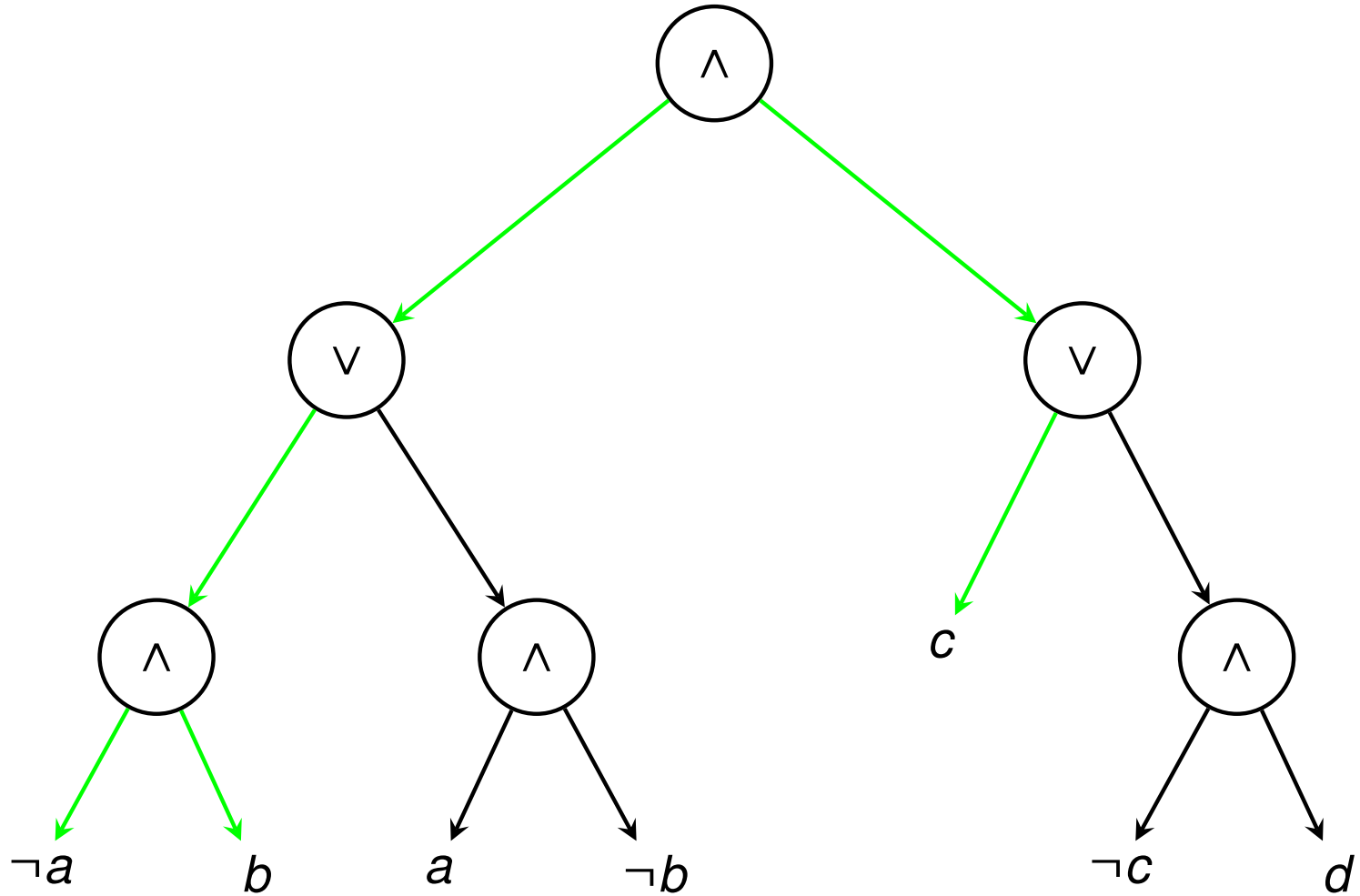
- The function **next** computes the next model graph given a d-DNNF $\Sigma = \langle N, E \rangle$ and current model $\omega = \langle N_\omega, E_\omega \rangle$
- If $n$ is a leaf node, $\textbf{next}(\omega)$ returns $\omega$, as there are no further models to enumerate
- For or-nodes:
  - If $n = (n, m) \in E$ and $\textbf{end}(\textbf{sub}(\Sigma, \textbf{m})) \neq \textbf{sub}(\omega, \textbf{m})$, recursively call **next** on $sub(\omega, m)$.
  - If $\textbf{end}(\textbf{sub}(\Sigma, \textbf{m})) = \textbf{sub}(\omega, \textbf{m})$, check if $m = \text{left}(n)$ or $m = \text{right}(n)$:
    - If $m = \text{left}(n)$, begin enumerating models from the right branch: $\textbf{begin}(\textbf{sub}(\Sigma, \textbf{right}(\textbf{n})))$.
    - If $m = \text{right}(n)$, we have enumerated all models and return $\omega$.
- For and-nodes, the function traverses both left and right subgraphs similarly

# Example: Model Graph Enumeration



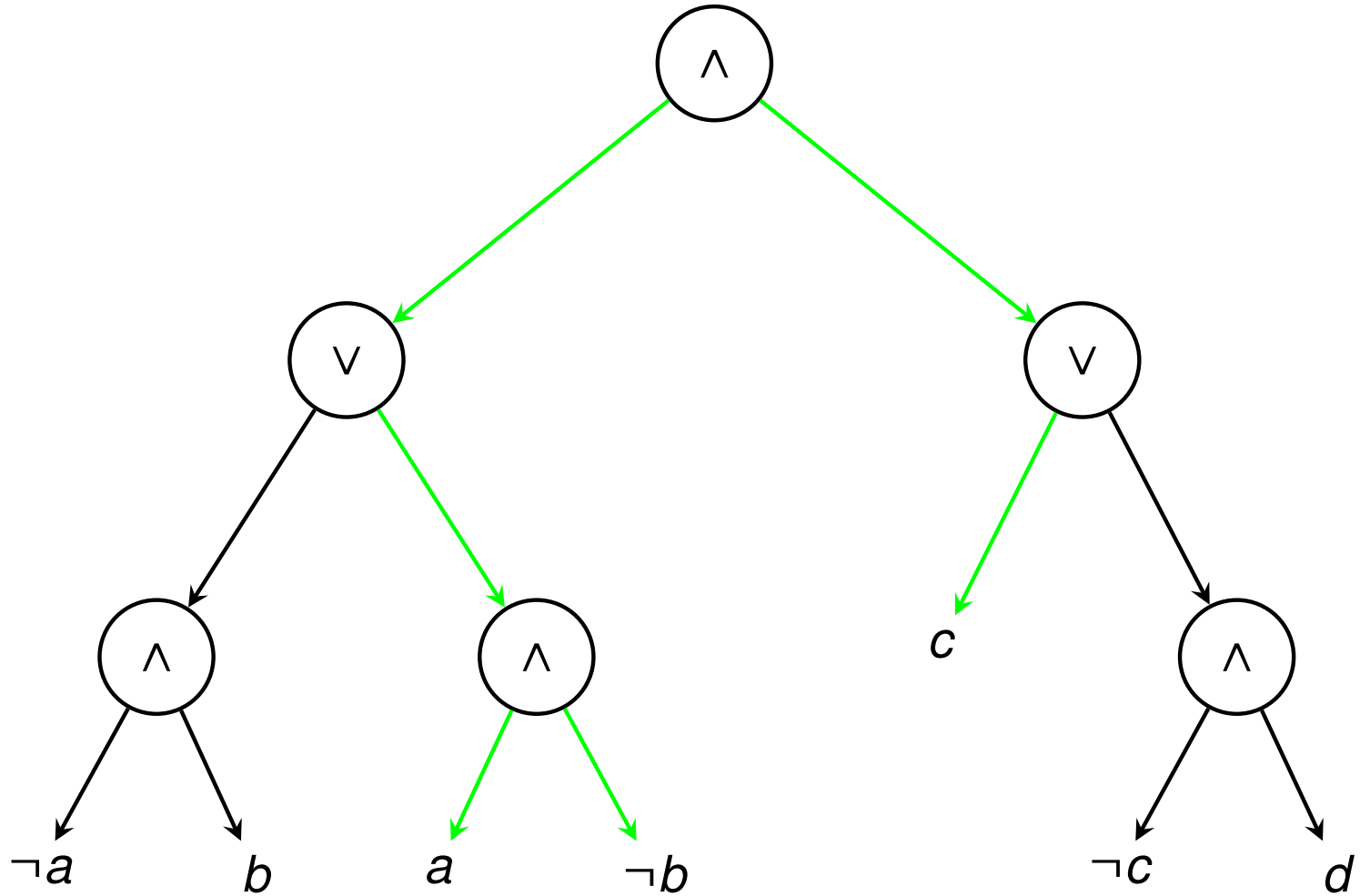

$\Gamma = \{$ $\}$

# Example: Model Graph Enumeration



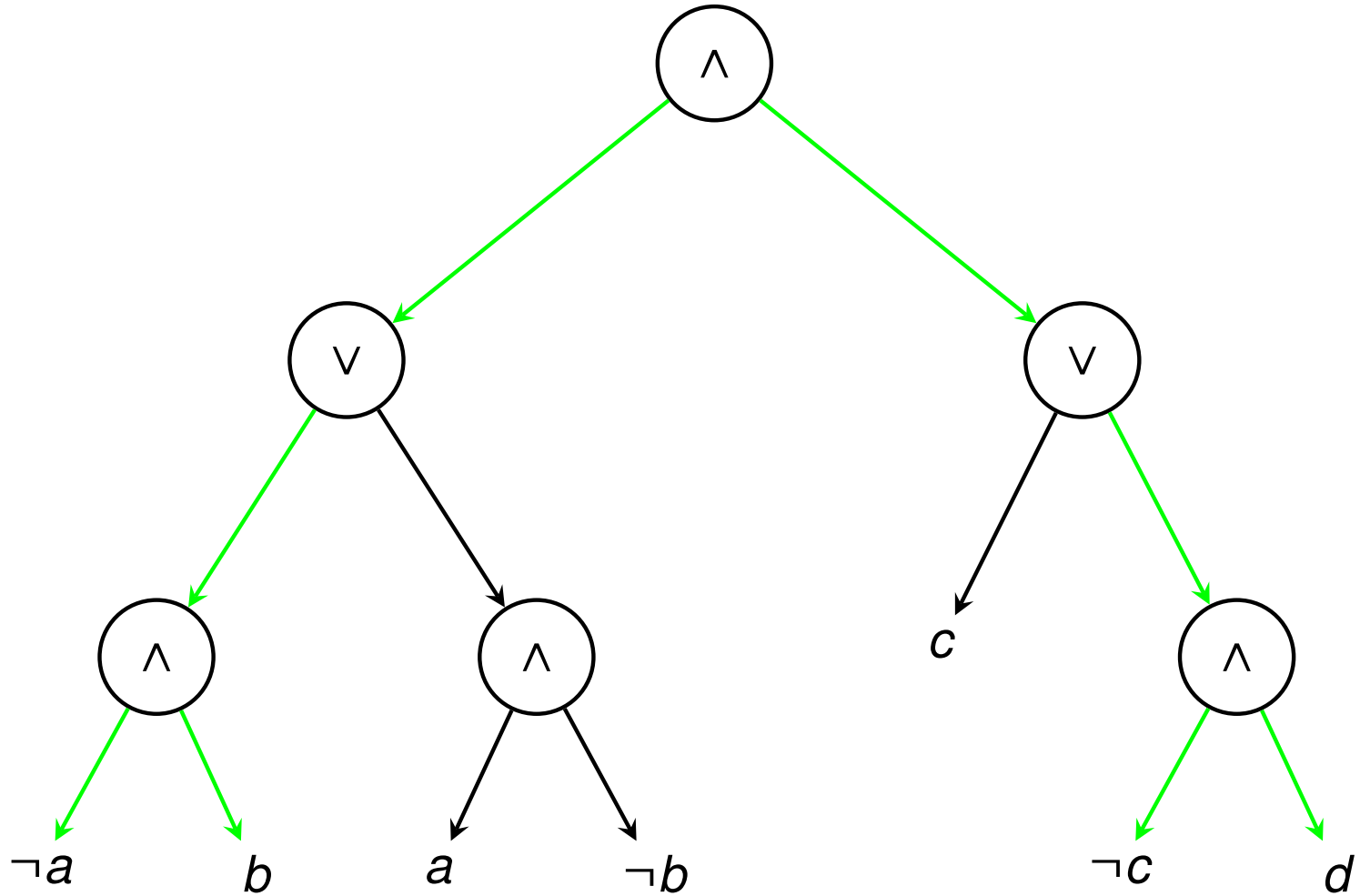

$\Gamma = \{(\neg a, b, c)$ $\}$

# Example: Model Graph Enumeration



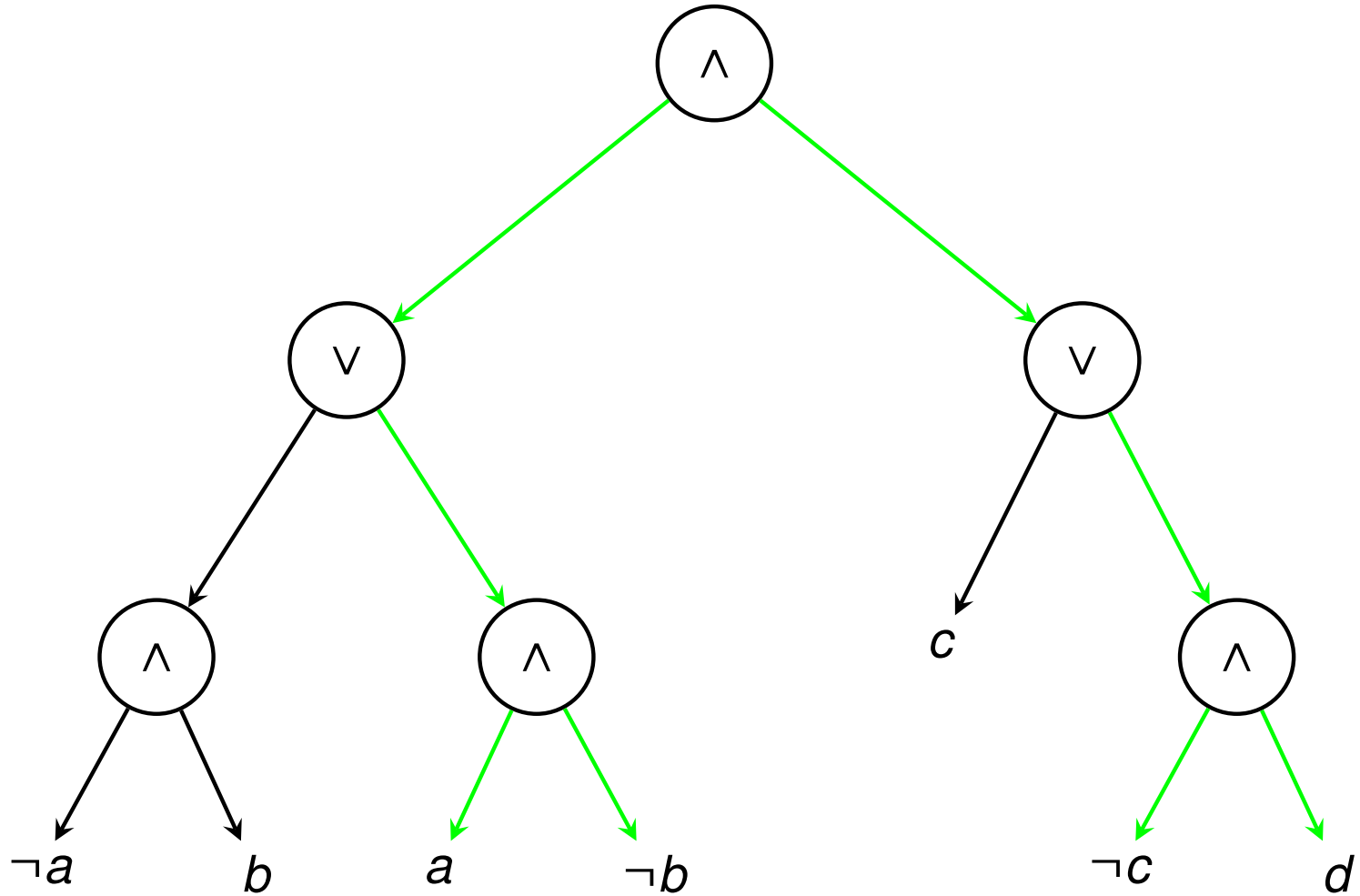

$\Gamma = \{(\neg a, b, c)(a, \neg b, c)$ $\}$

# Example: Model Graph Enumeration




$\Gamma = \{(\neg a, b, c)(a, \neg b, c)(\neg a, b, \neg c, d) \quad \}$

# Example: Model Graph Enumeration




$\Gamma = \{(\neg a, b, c)(a, \neg b, c)(\neg a, b, \neg c, d)(a, \neg b, \neg c, d)\}$

# Outline

# Model Counting (#SAT)

## Definitions

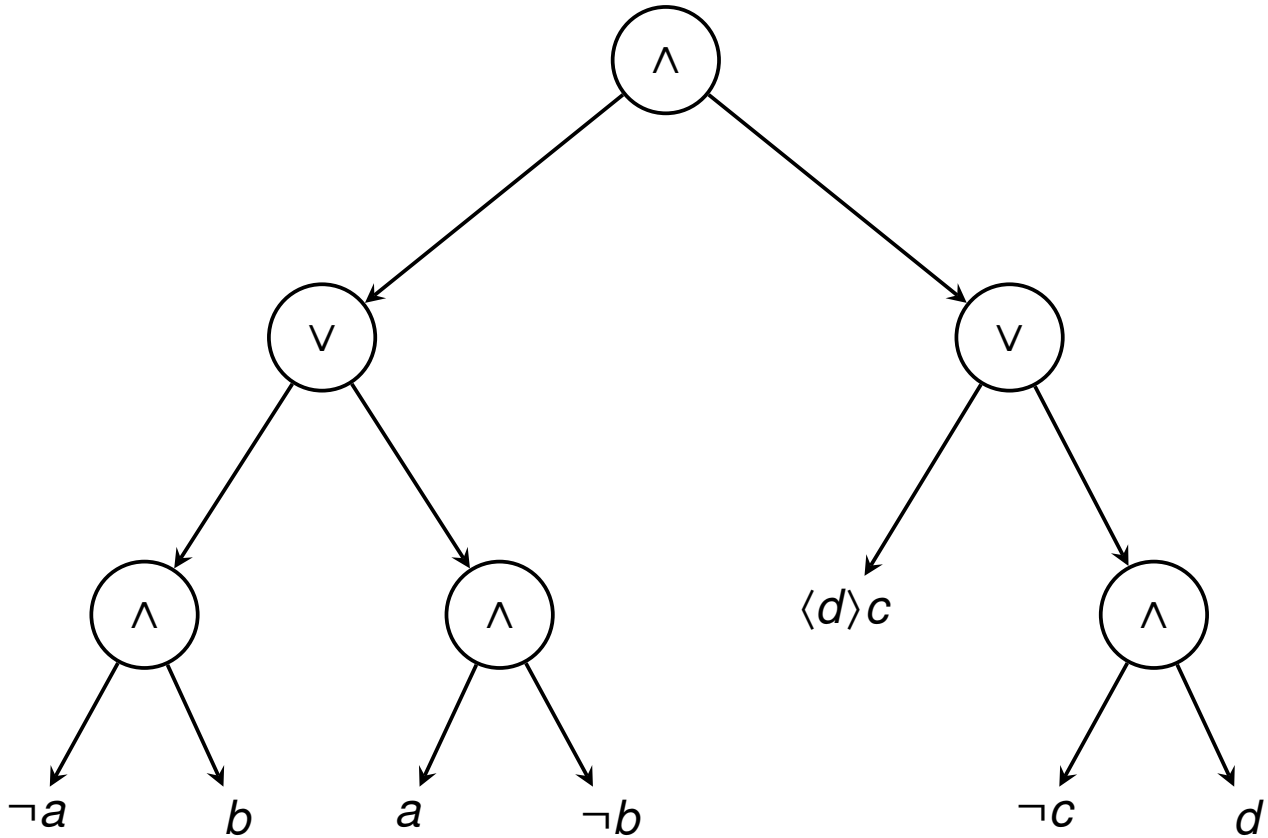
- **Goal:** Determine the exact number of satisfying assignments, $\|\Phi[E]\|$.
- **Mechanism:** Linear-time algebraic evaluation via bottom-up traversal.
- Decomposable $\wedge$ gates **multiply** children counts; deterministic $\vee$ gates **sum** mutually exclusive children counts. Injects powers of 2 for free variables.
- **Dynamic Smoothness:** For each node, we first compute the free variables associated with the sub-circuit $\Phi$ and store them in $\Phi.\mathit{free}$.

## Algorithm: #SAT $(\Phi, E)$

```
1   if Φ == ⊥:
2     return 0
3   if Φ is a literal ℓ:
4     if ¬ℓ ∈ E:
5       return 0
6     return 1
7   if Φ == Φ1 ∨ Φ2:
8     return 2^|Φ.free| × ((#SAT(Φ1, E) + #SAT(Φ2, E)))
9   if Φ == Φ1 ∧ Φ2:
10    return 2^|Φ.free| × ((#SAT(Φ1, E) × #SAT(Φ2, E)))
11  return 1   // Φ == ⊤
```

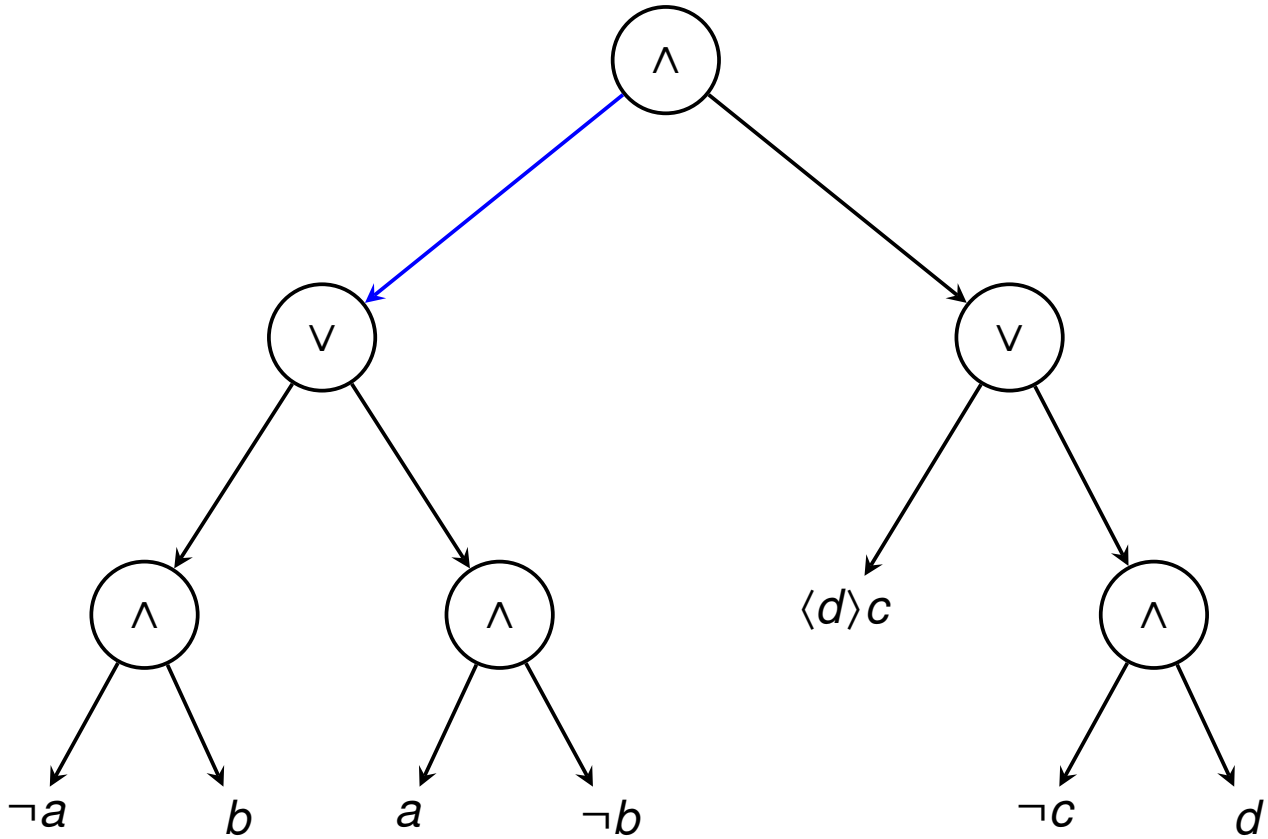
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



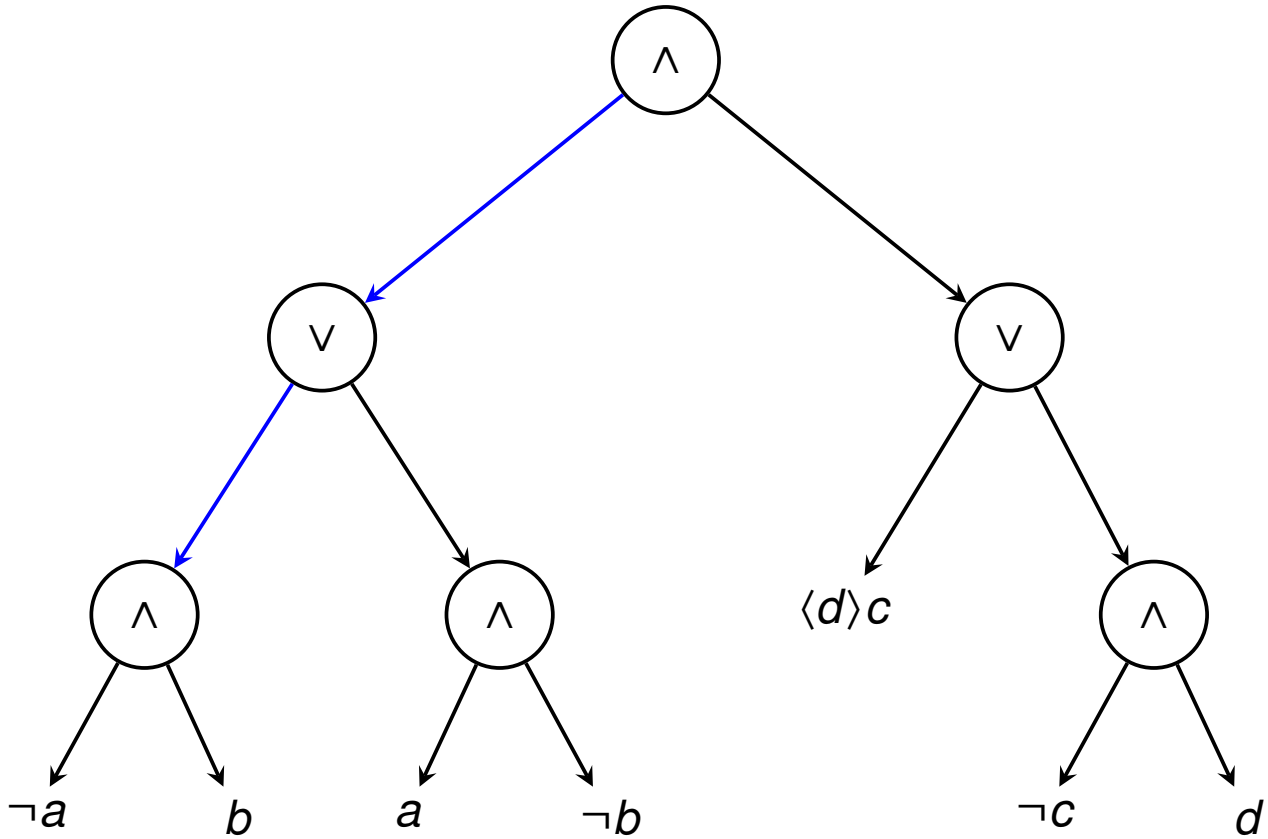
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



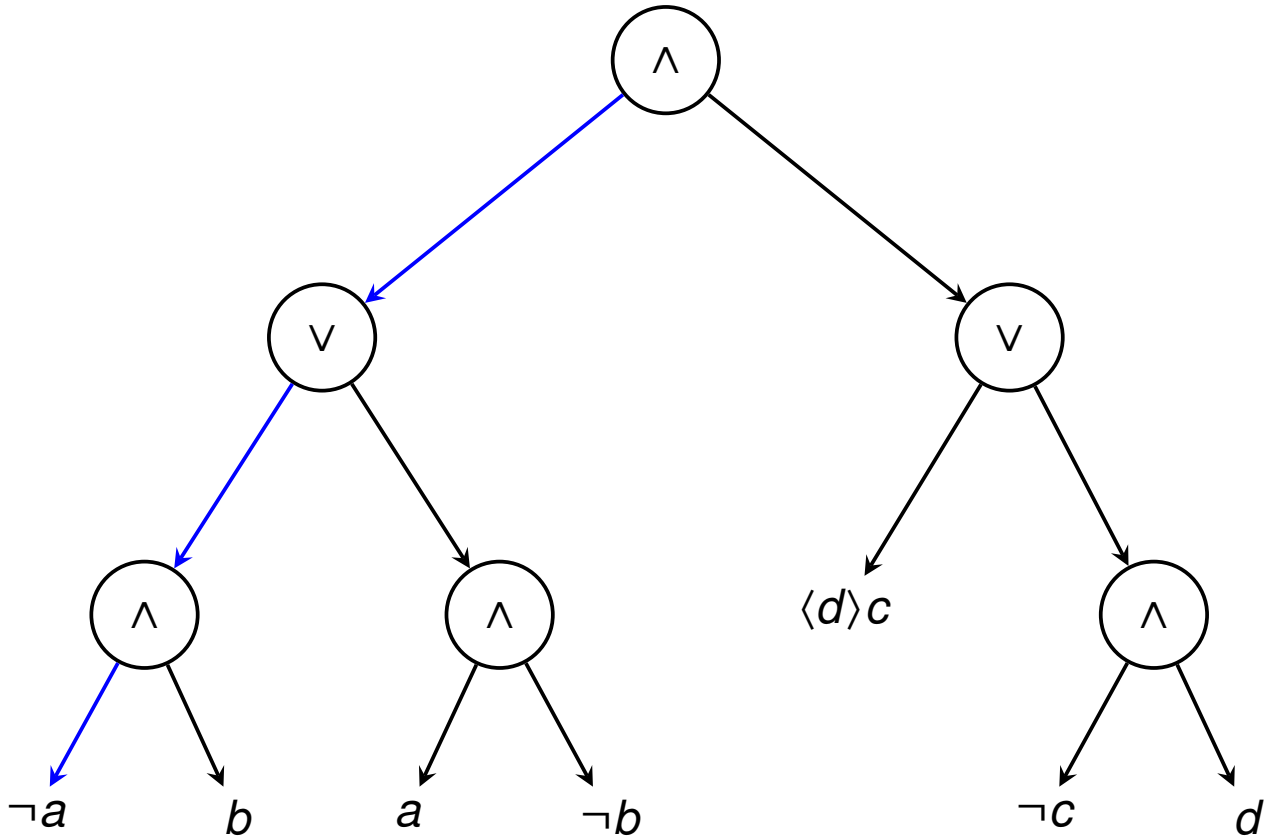
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



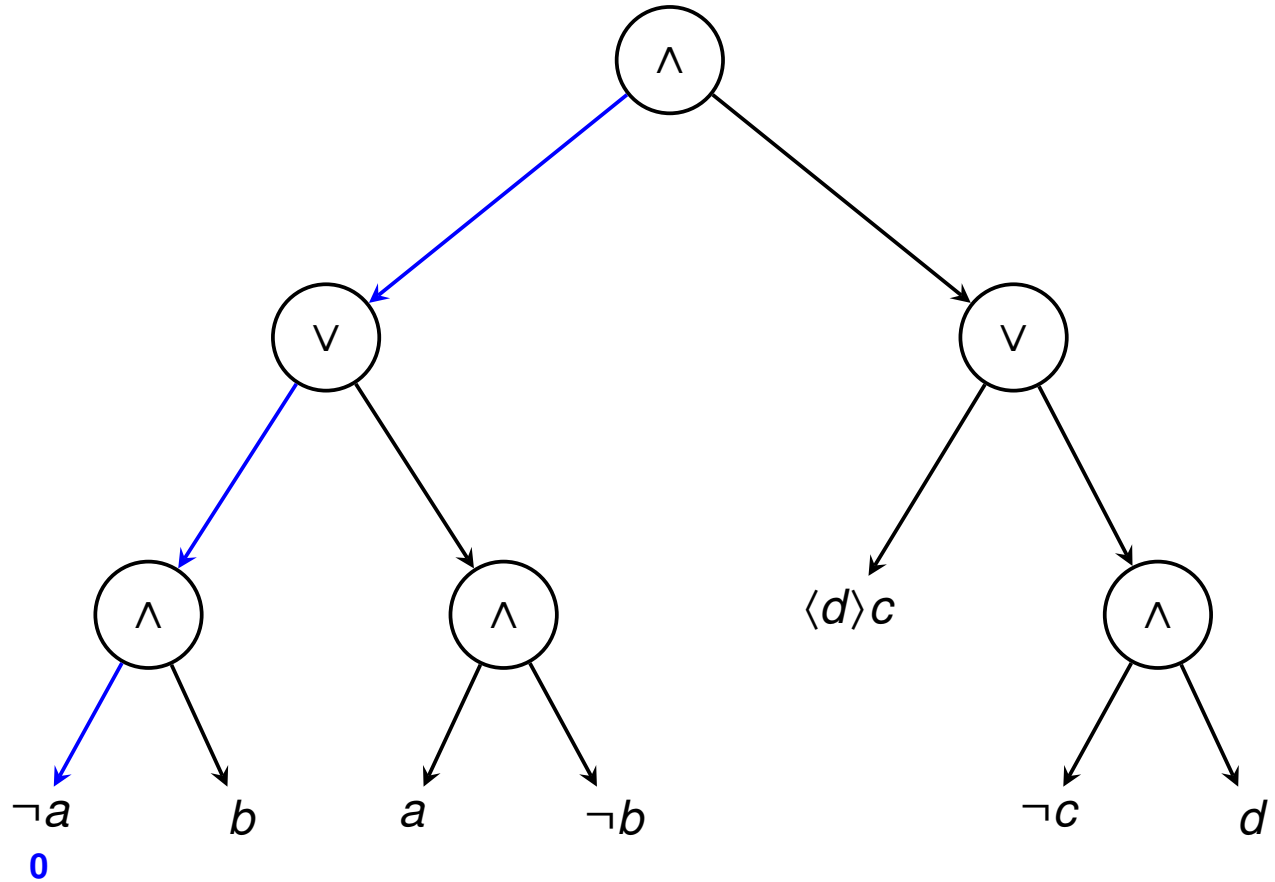
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



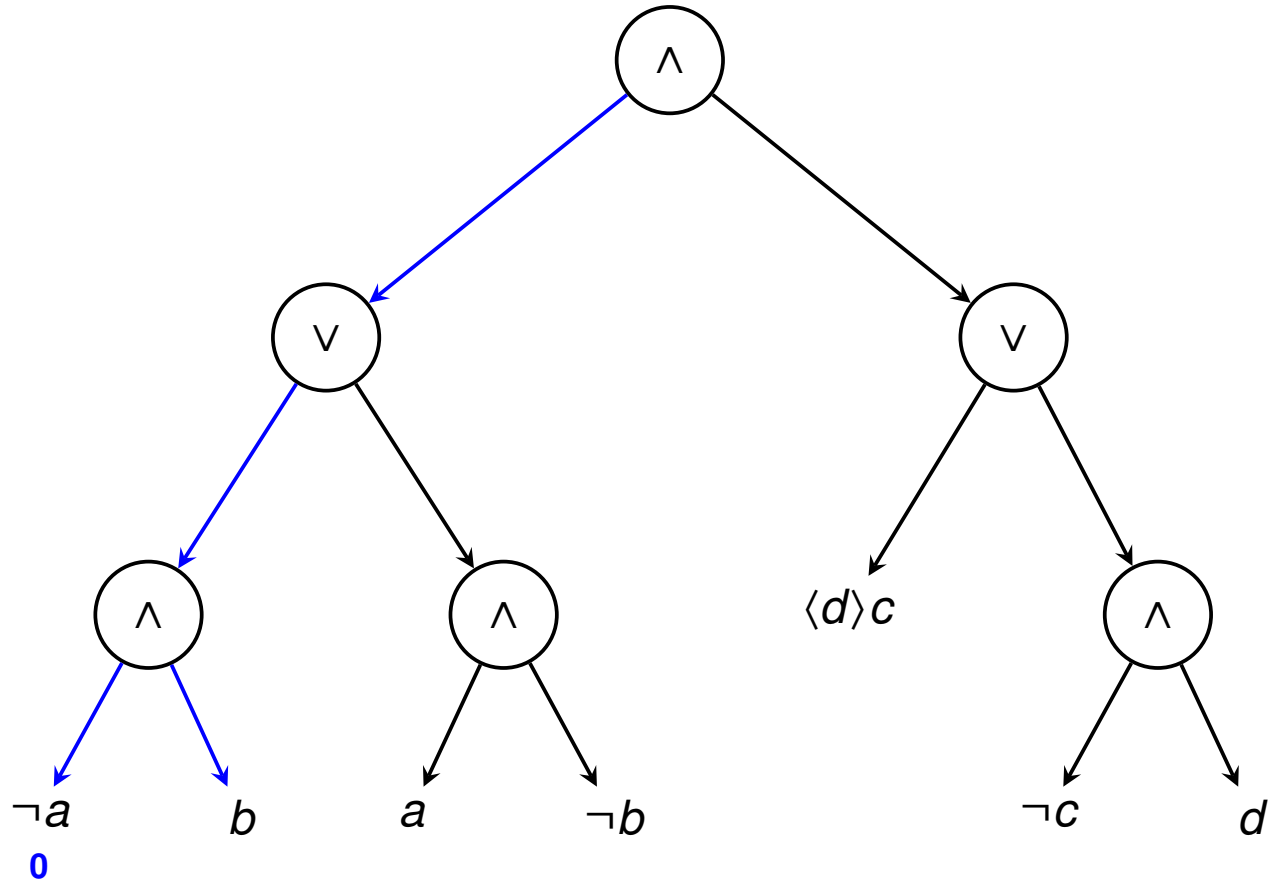
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



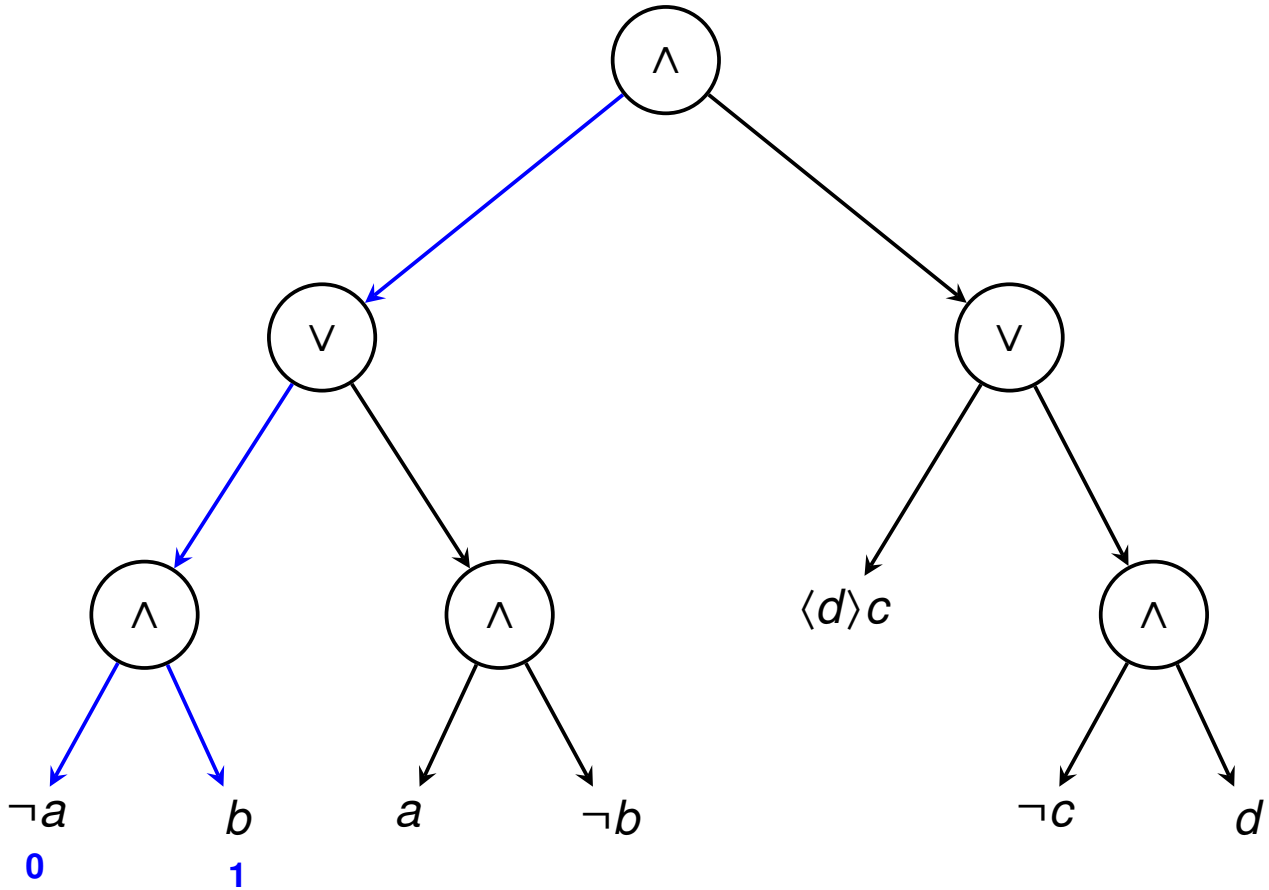
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



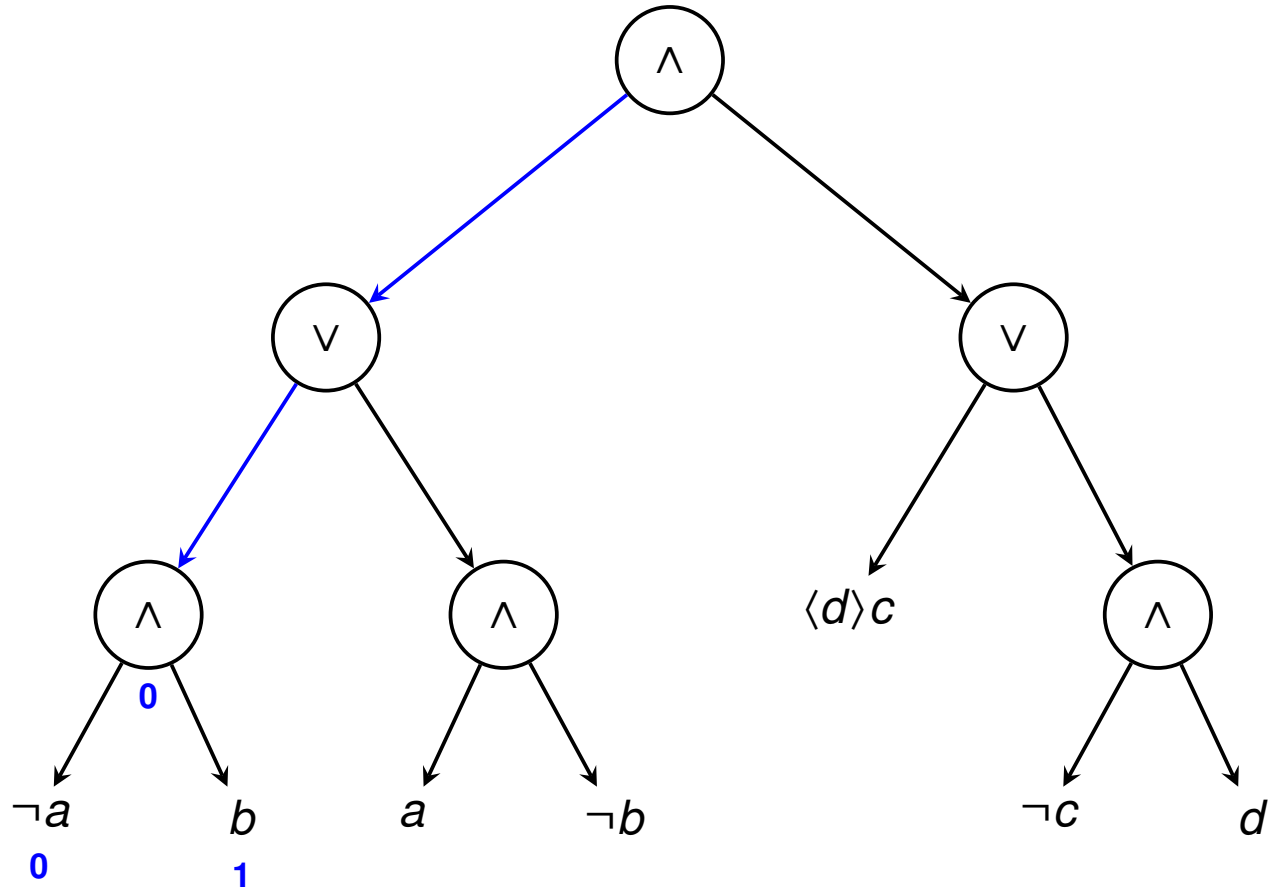
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



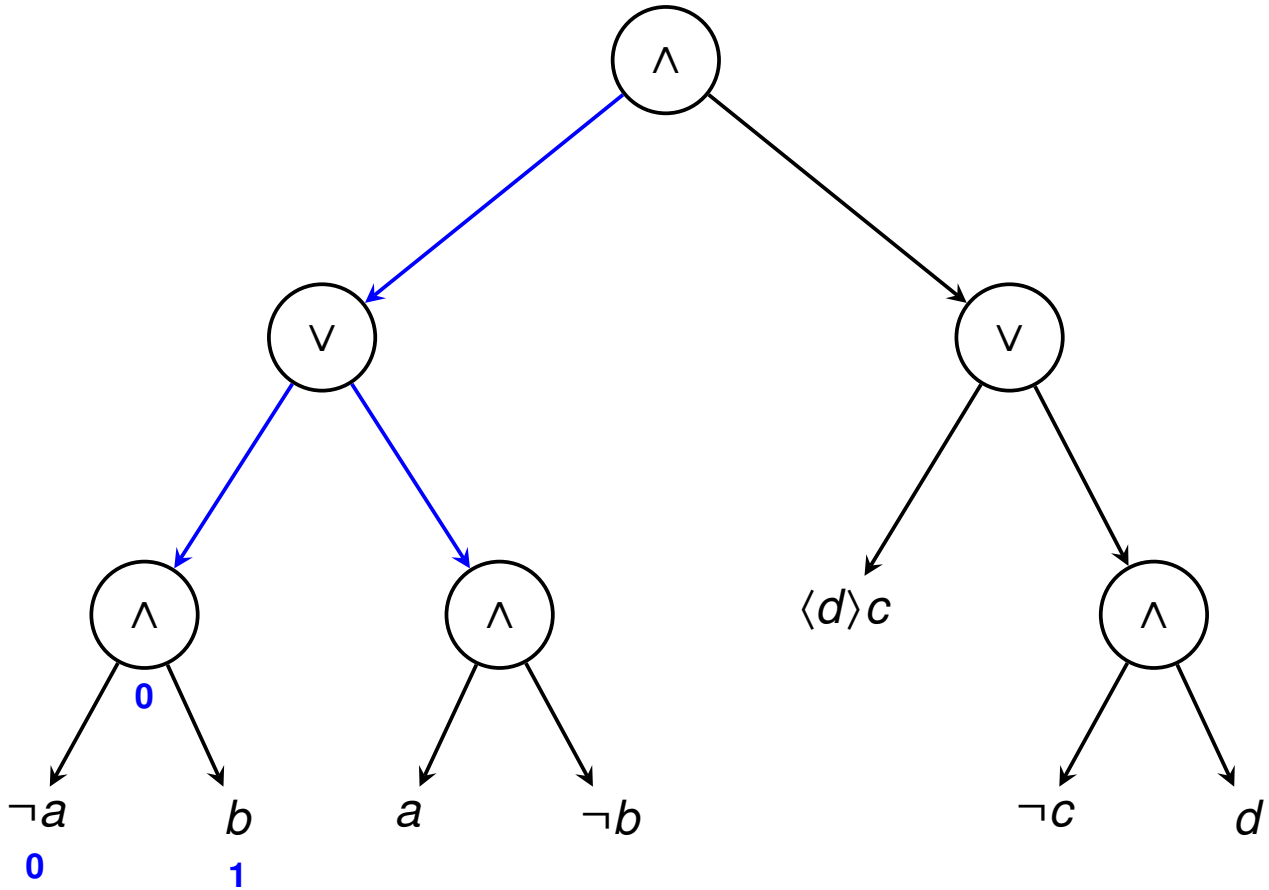
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



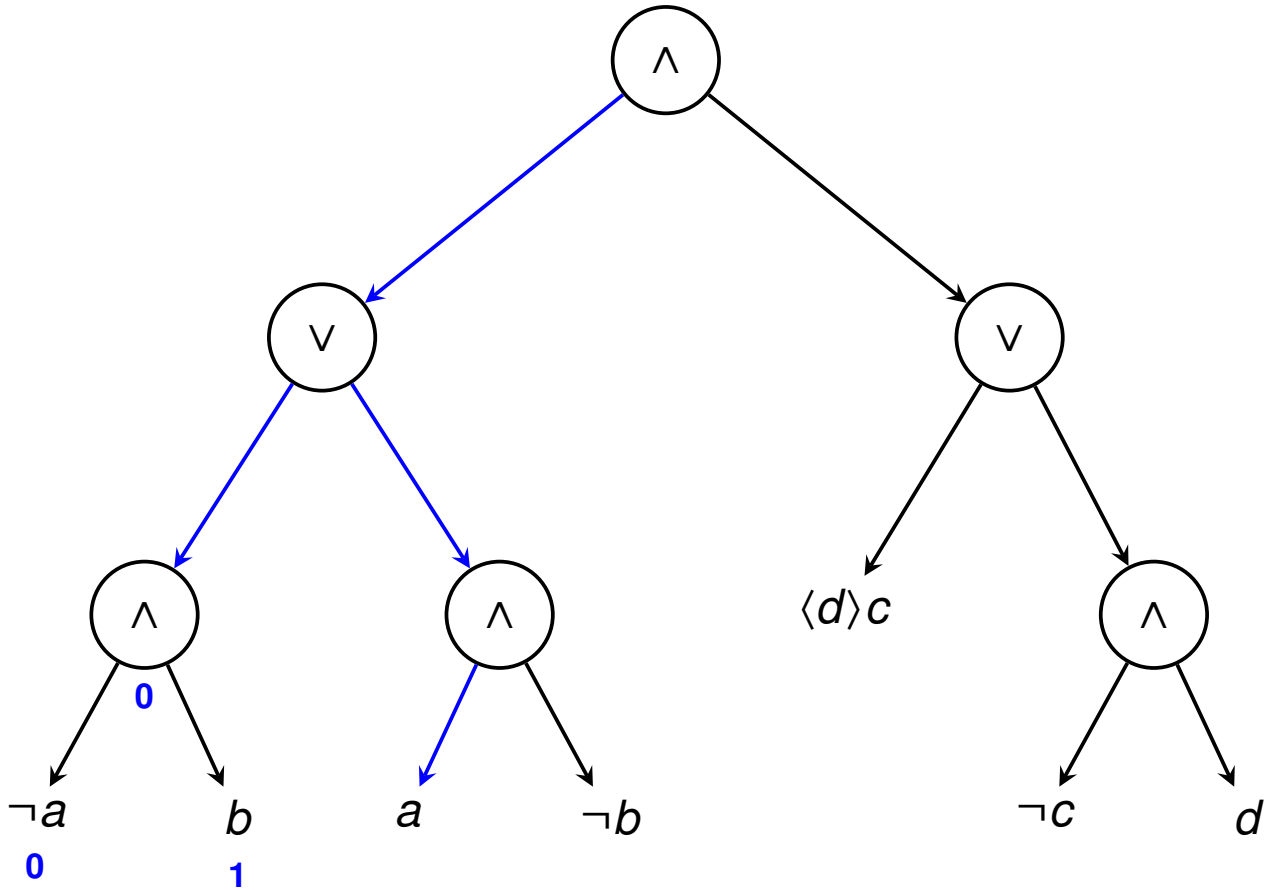
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



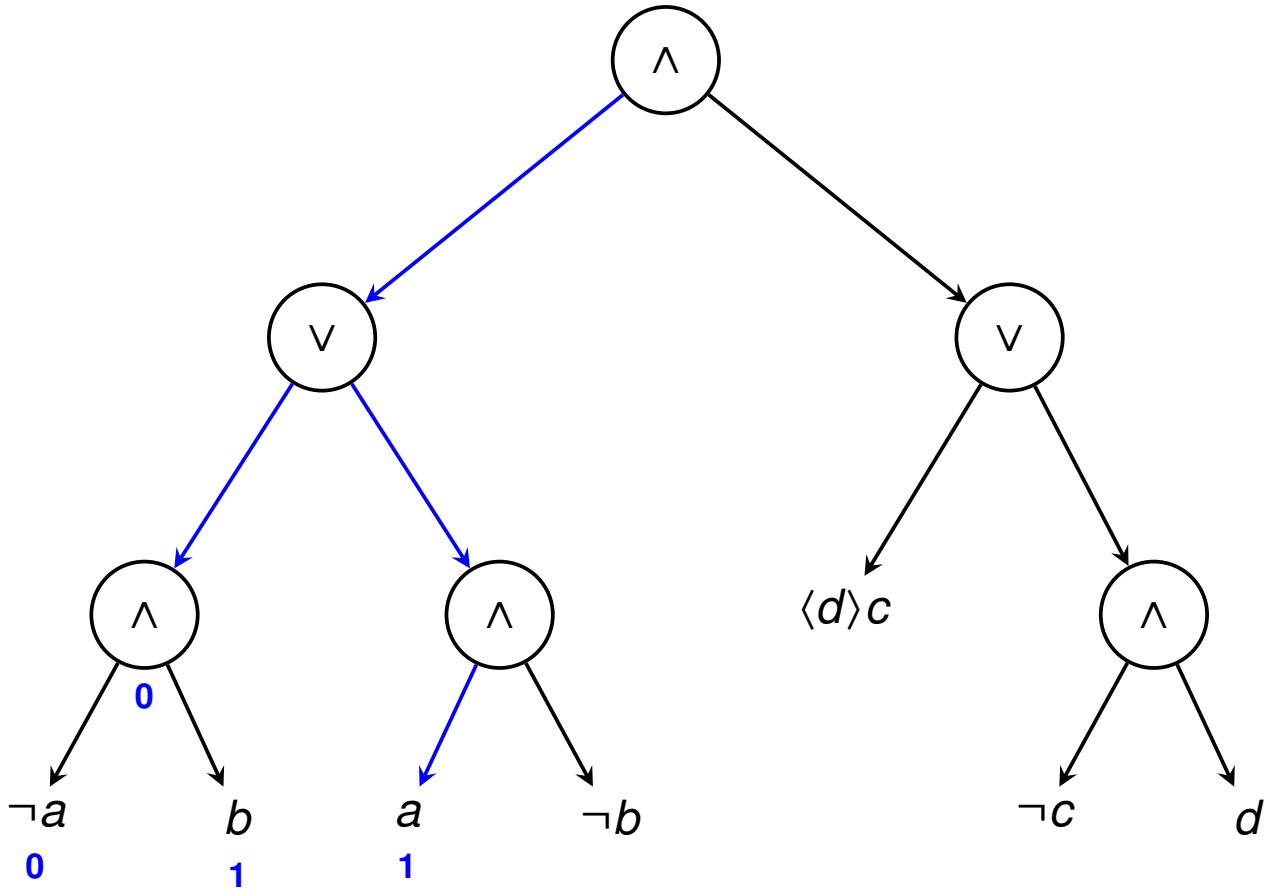
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



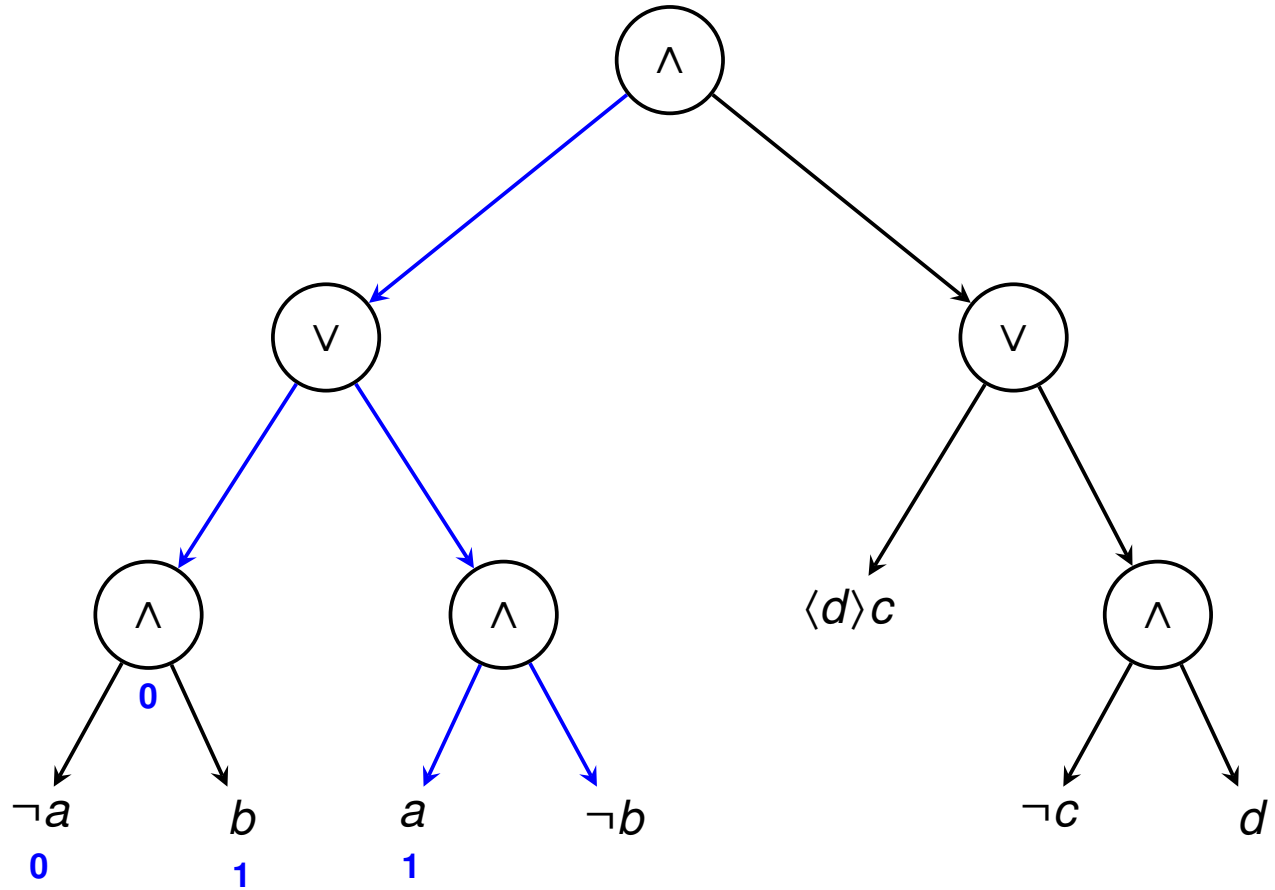
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



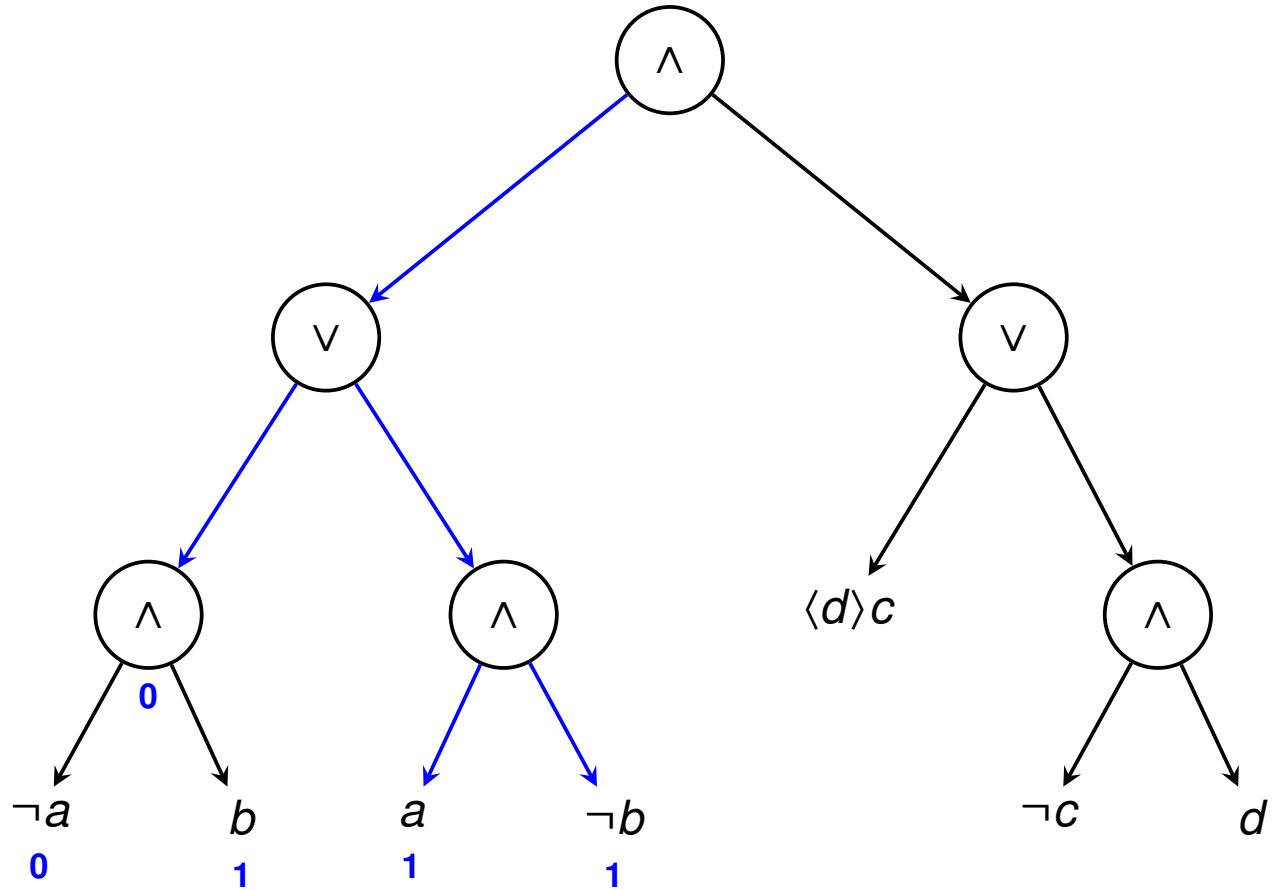
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



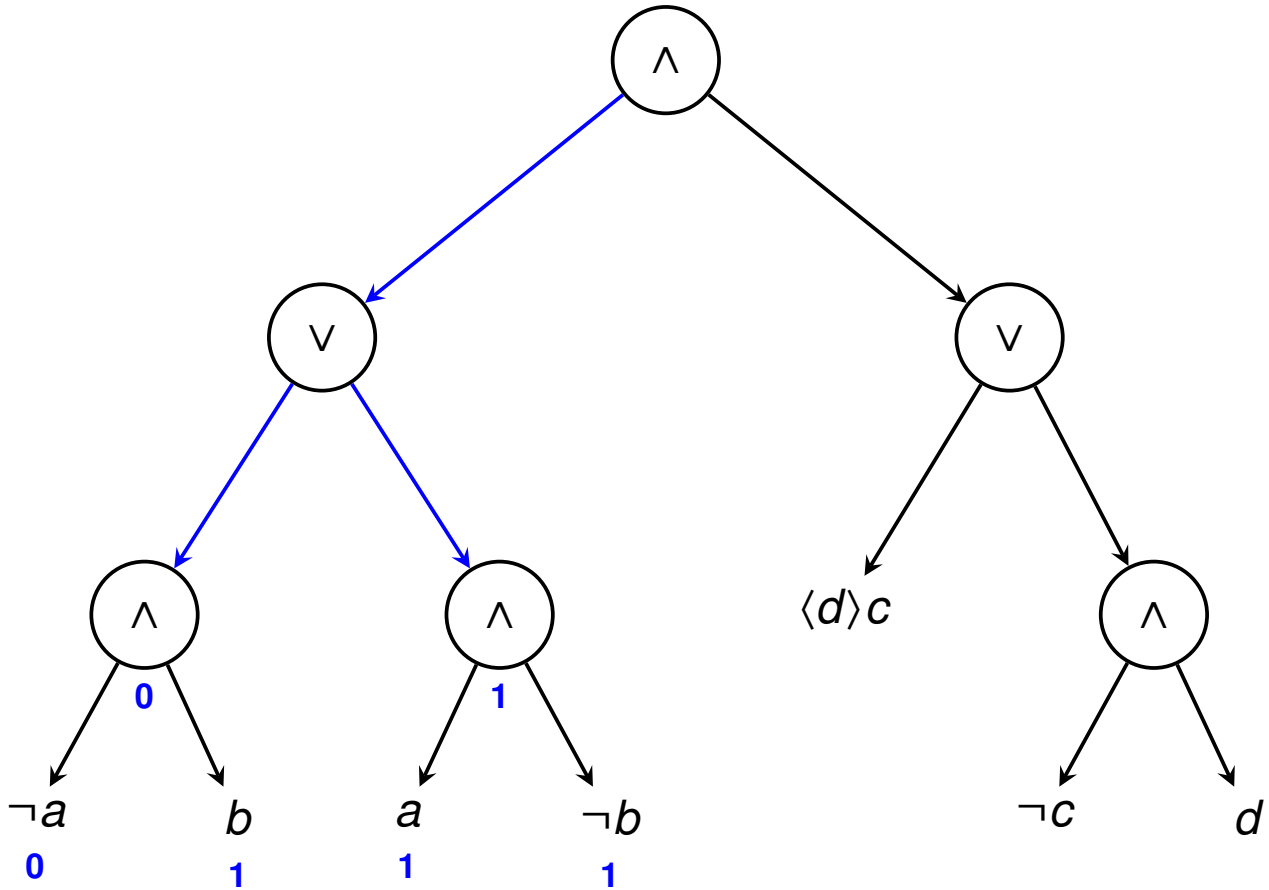
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



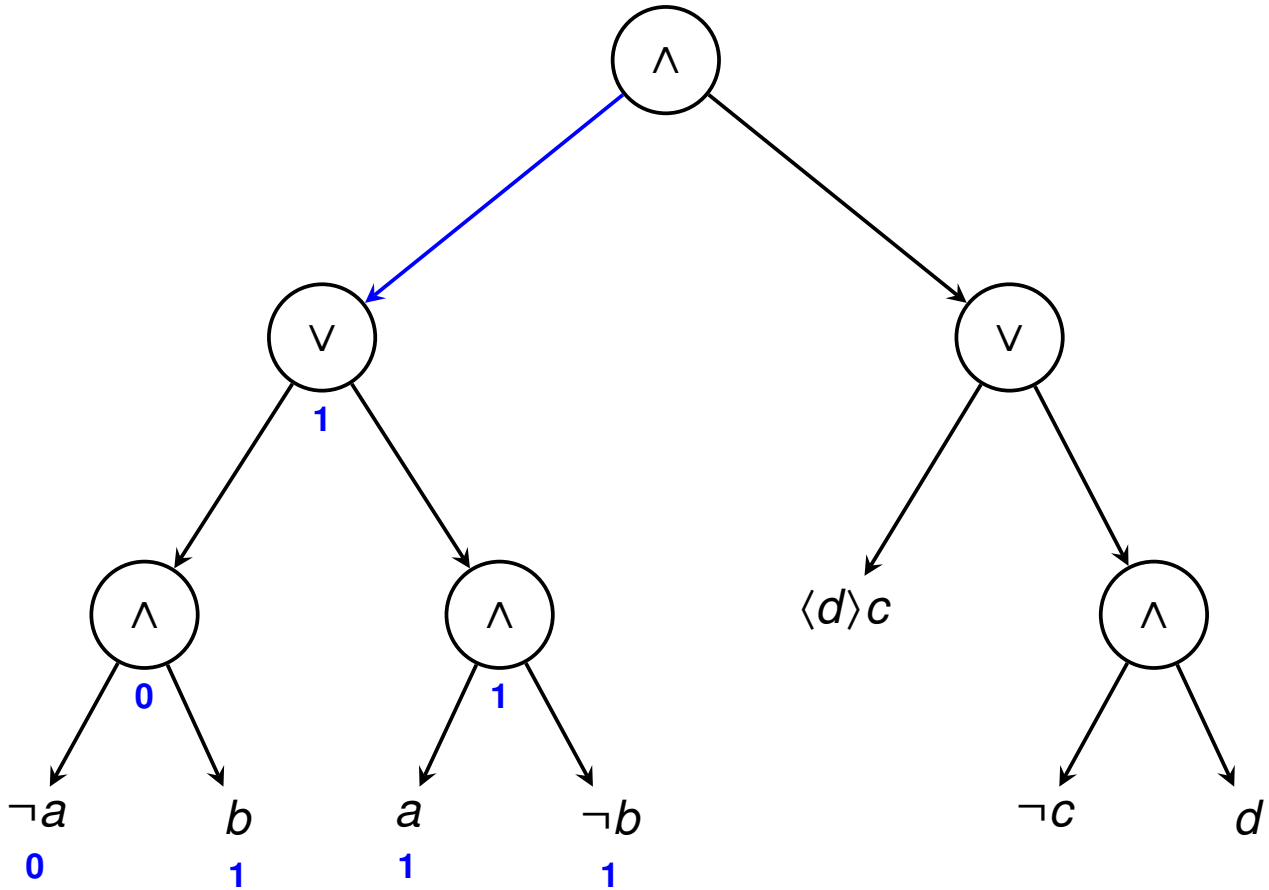
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



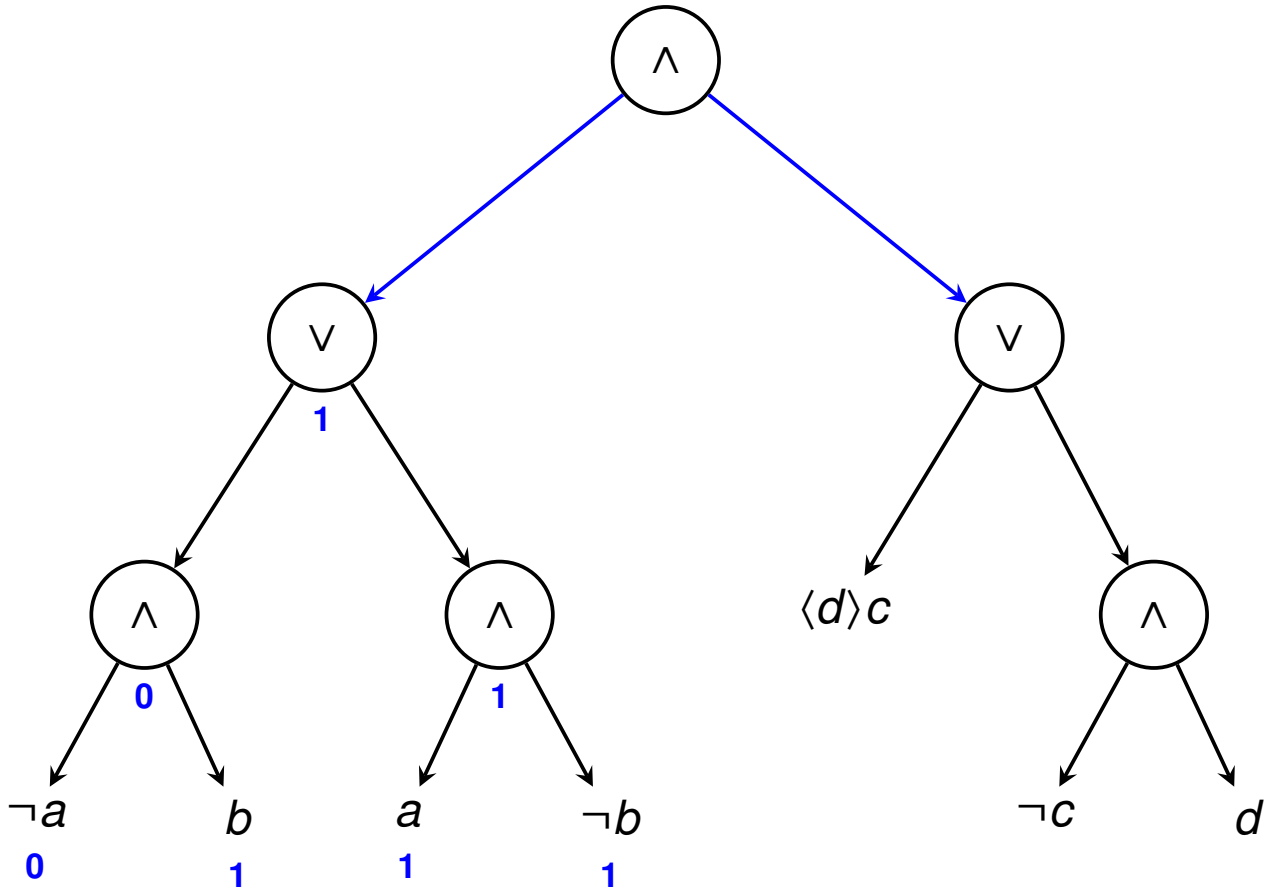
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



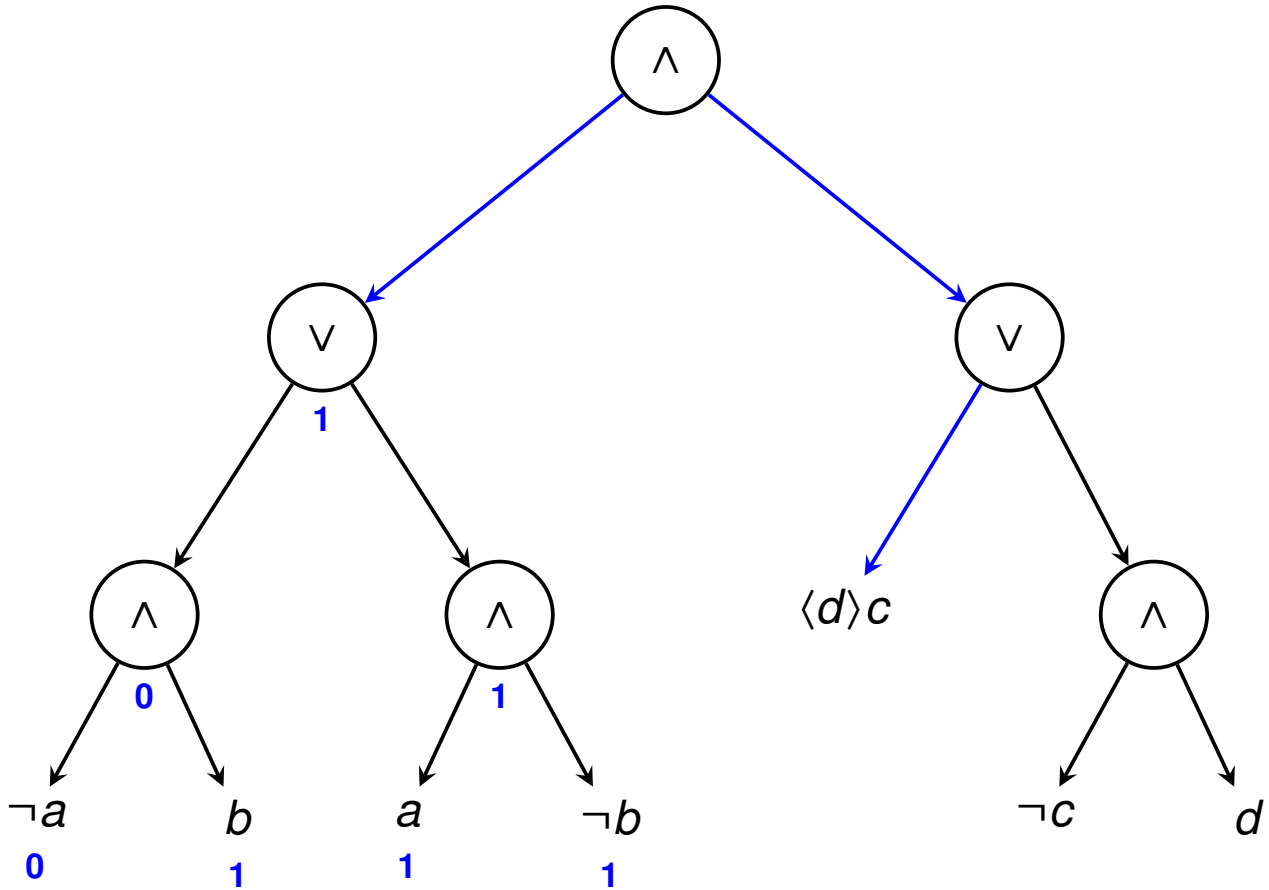
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



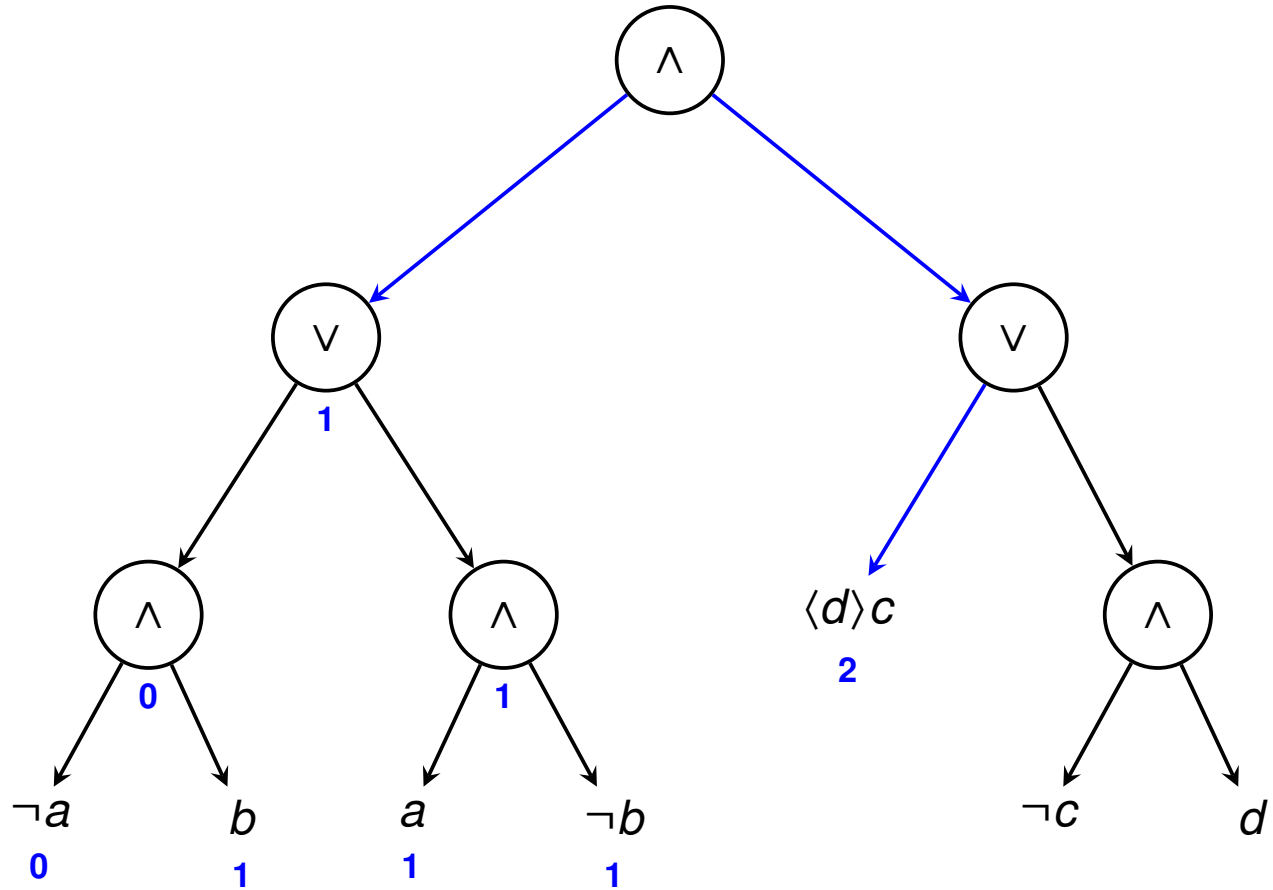
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



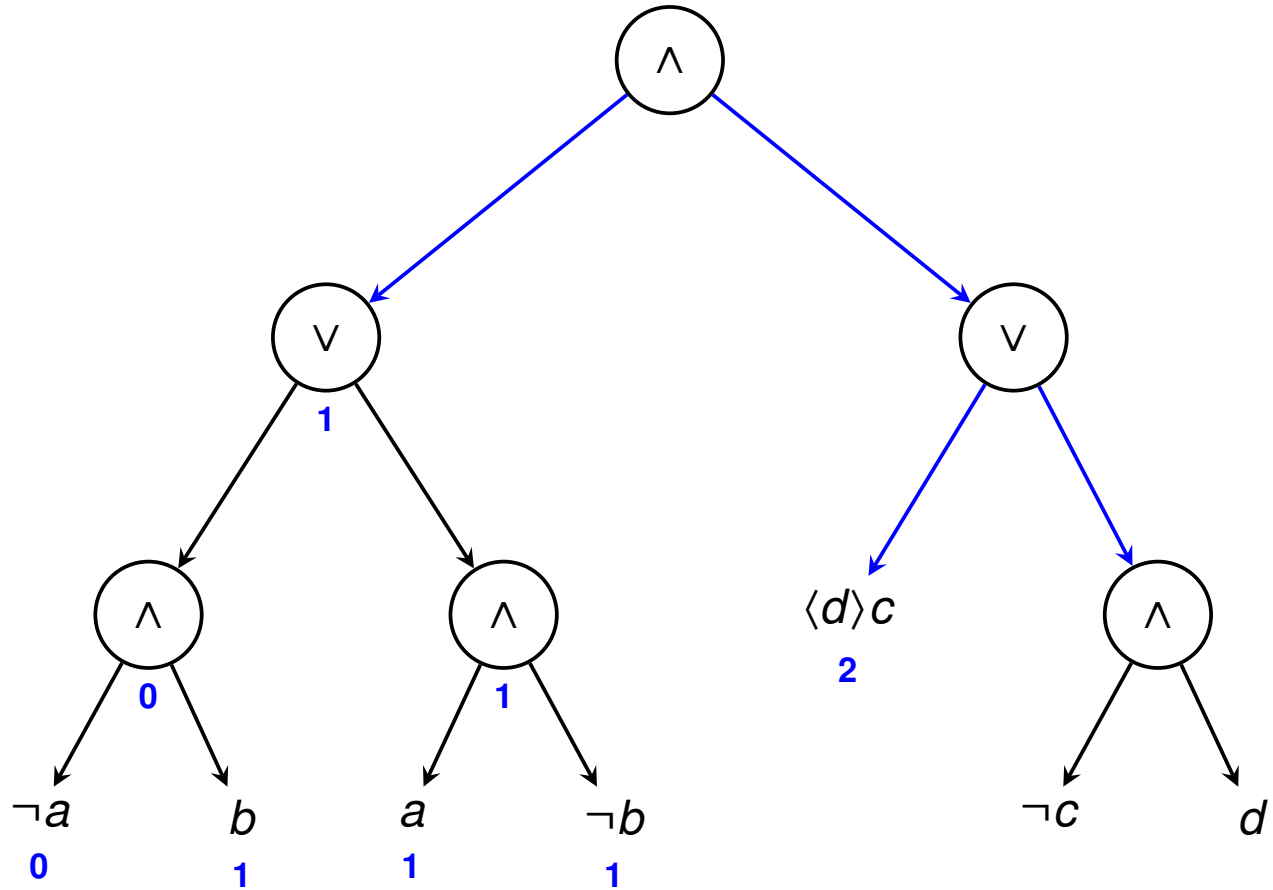
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



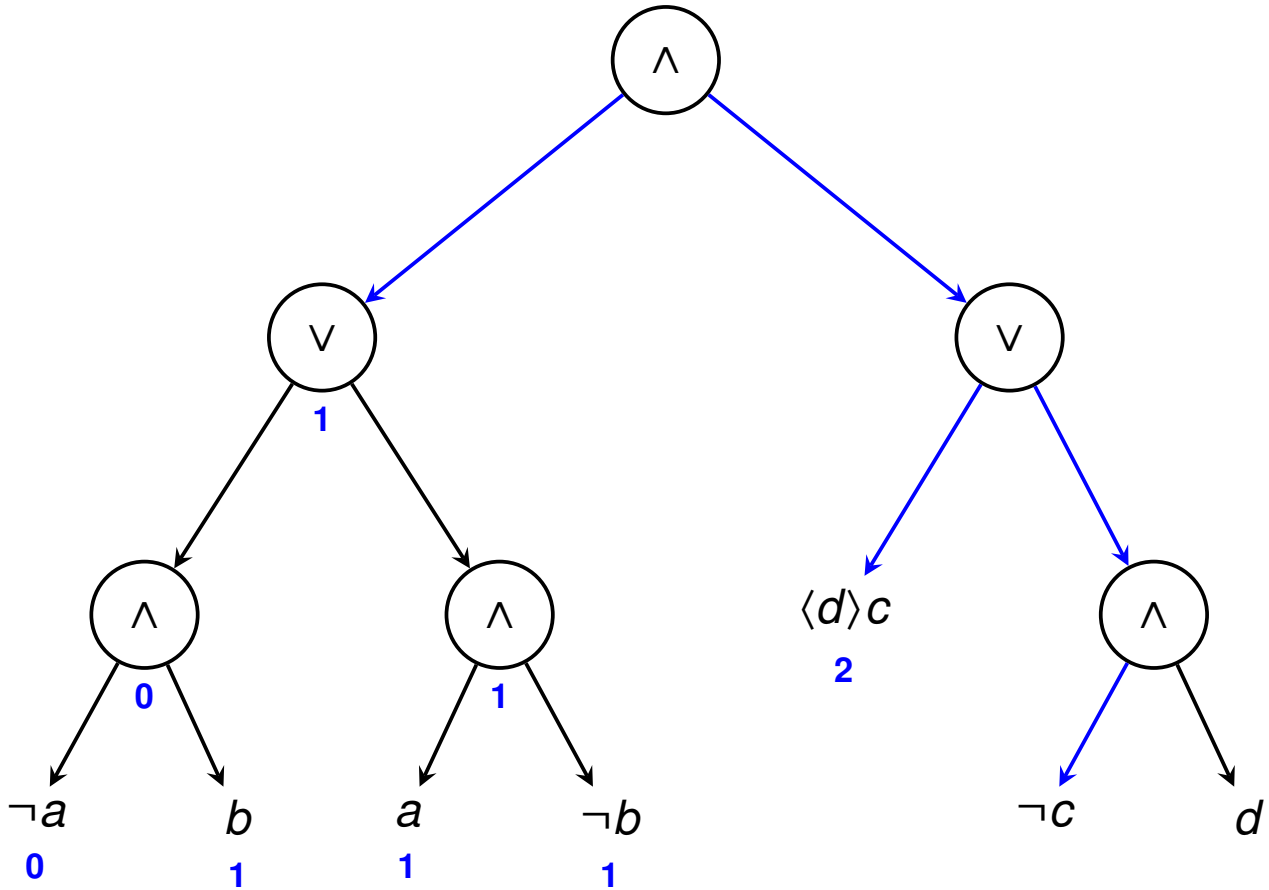
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



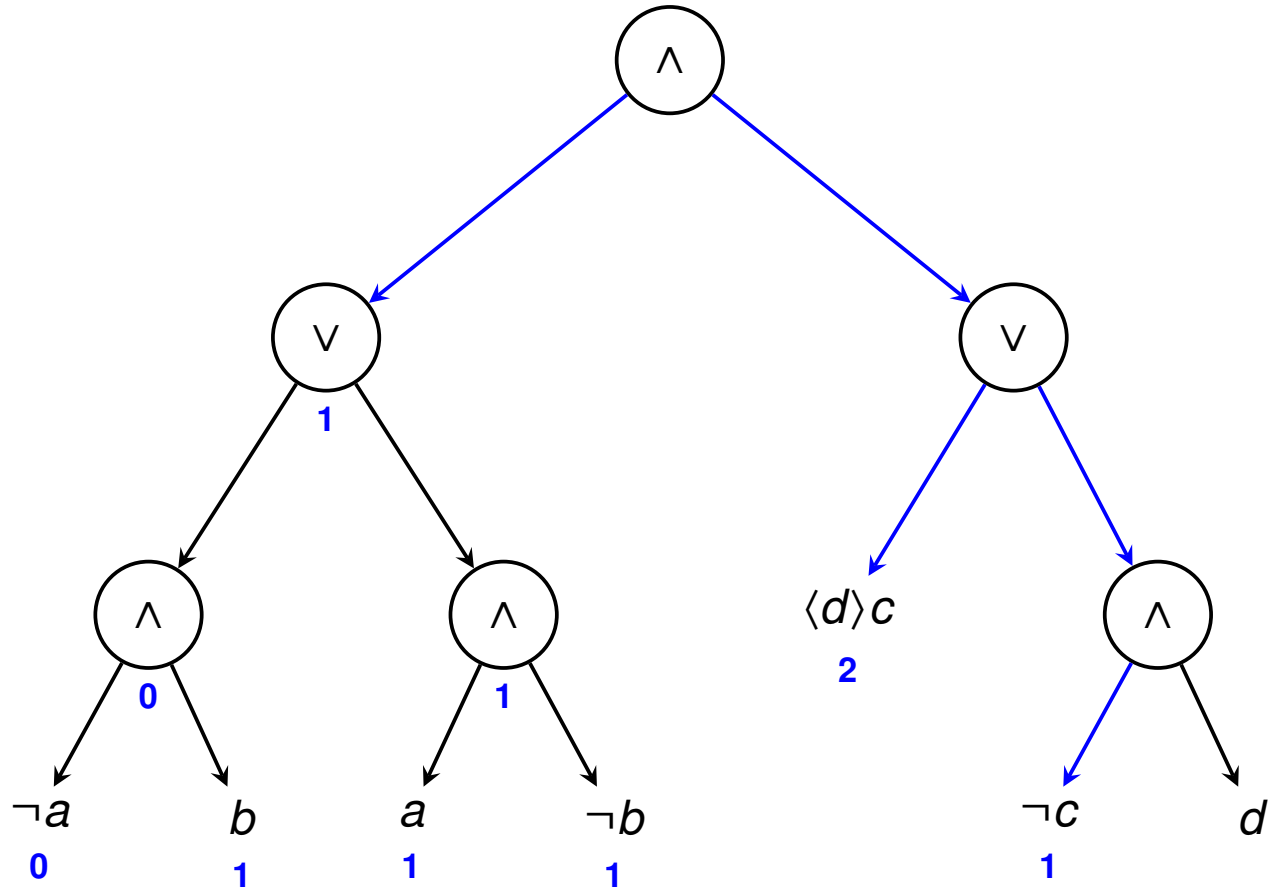
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



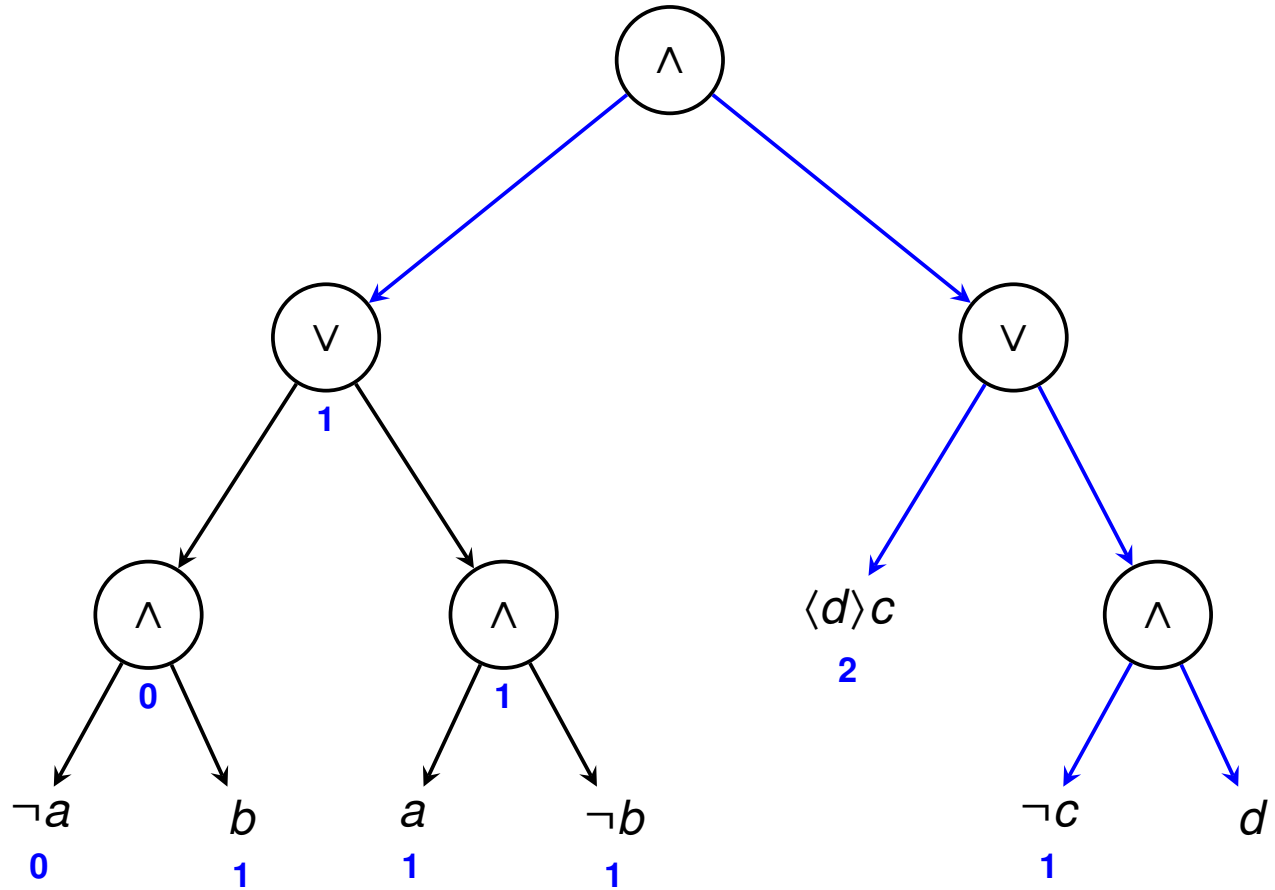
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



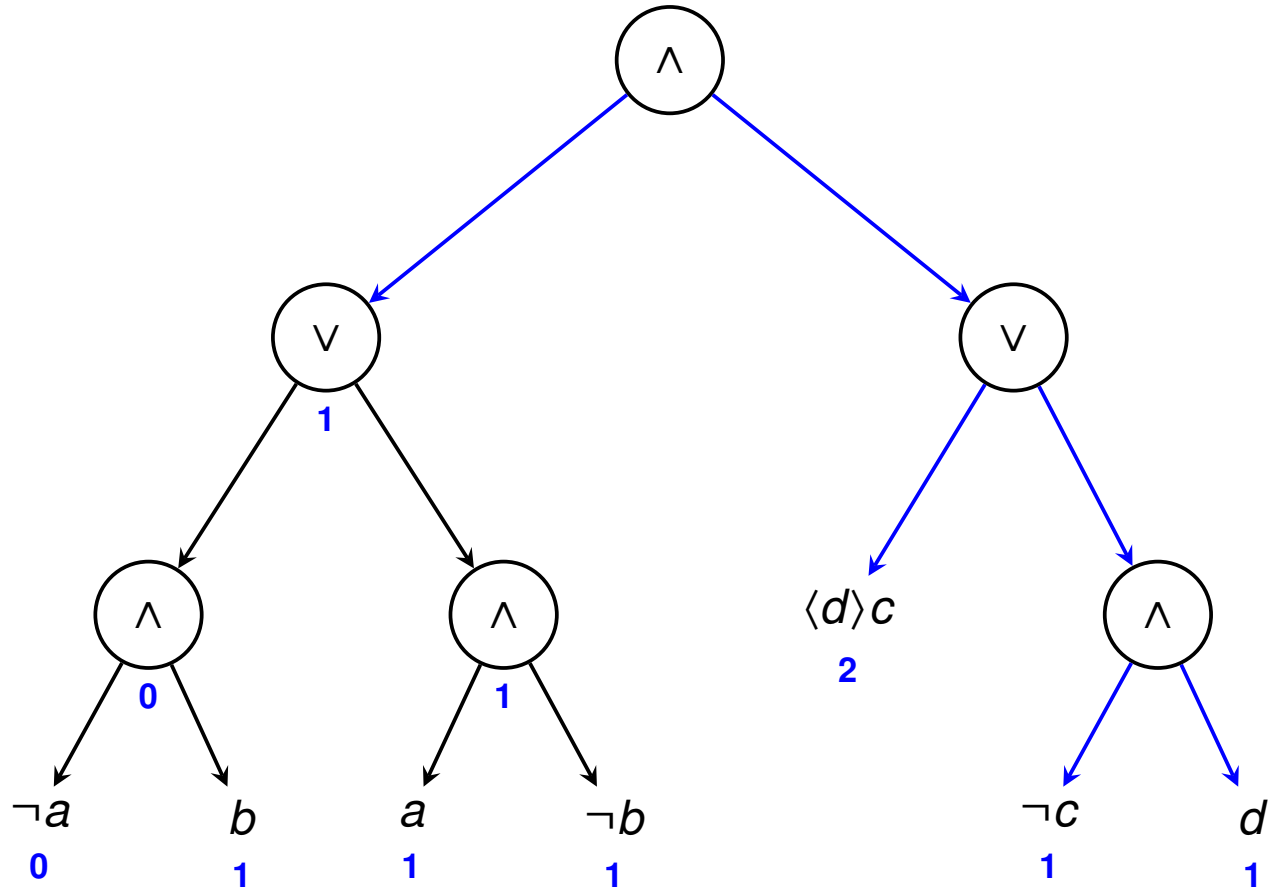
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



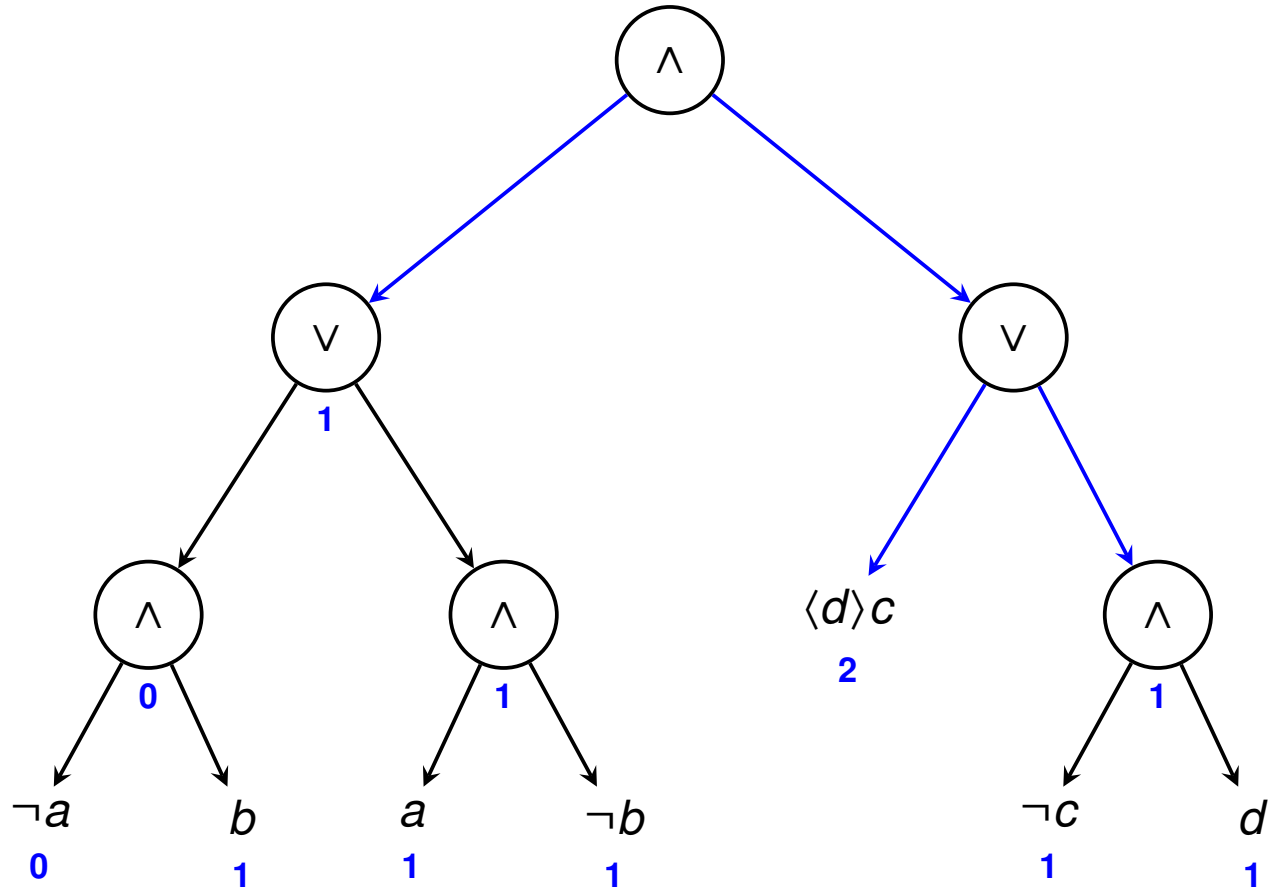
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



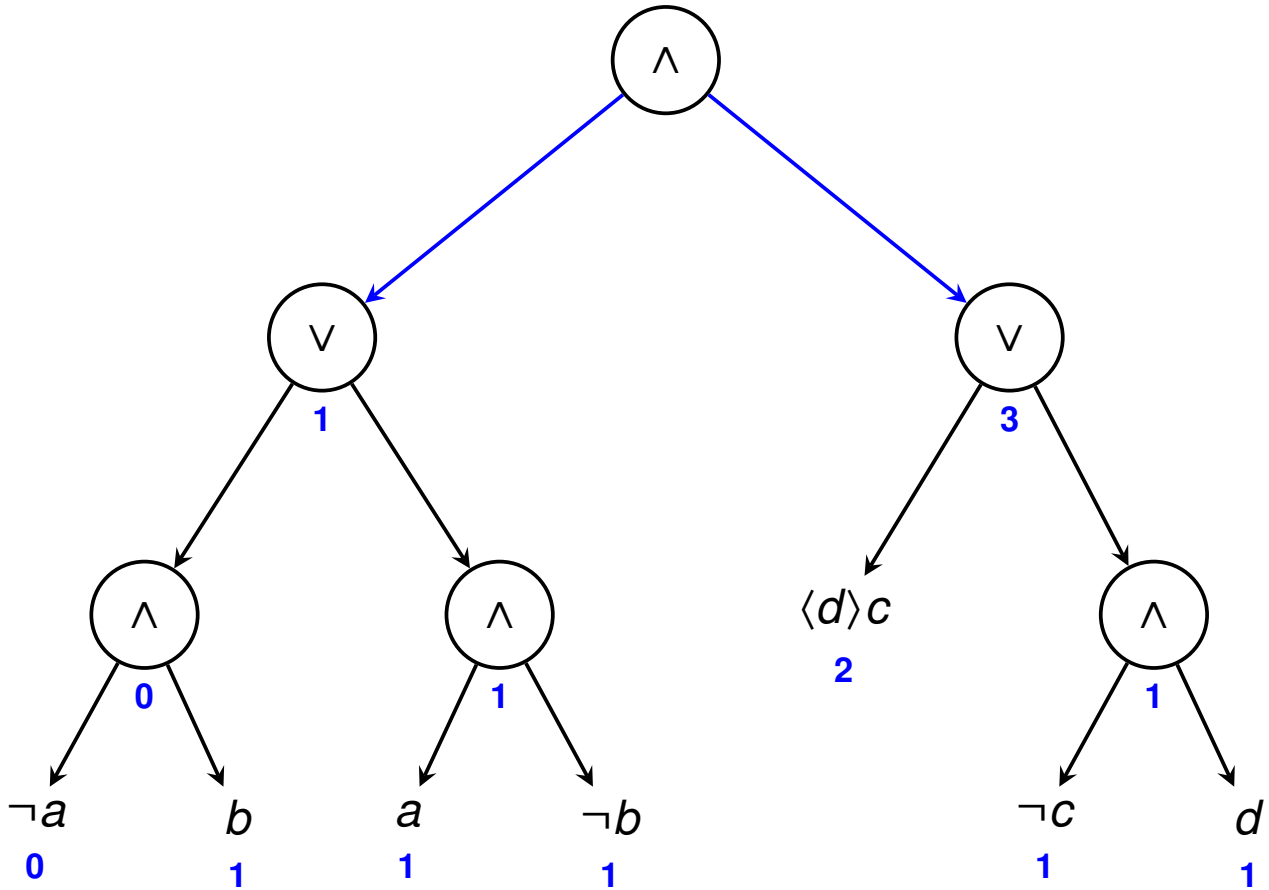
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



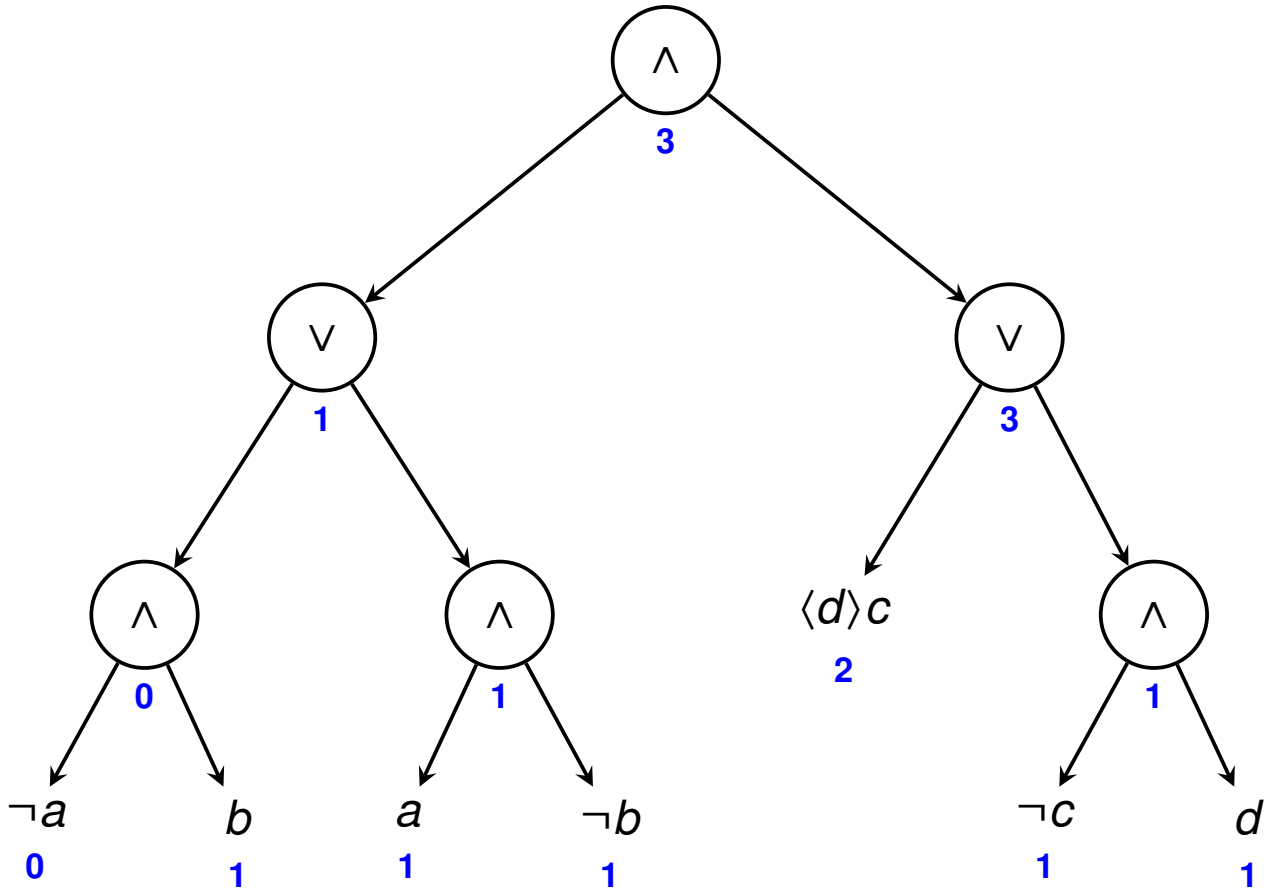
# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



# Running Example: #SAT Evaluation

Consider the CNF formula $\Psi$ with evidence $E = \{a\}$.



# Outline

# Direct Access

## Definitions

- **Goal:** Retrieve the $k$-th satisfying assignment based on a strict lexicographical order ($\tau$).
- **Mechanism:** Polynomial-time deterministic descent. Checks the model count of the negative branch ($c_{\neg x}$). If target $k$ falls within it, descend negatively; otherwise, descend positively and shift the search interval.

## Running Example

Let $\Psi = \{a \vee b, \neg a \vee \neg b, c \vee d \vee a, c \vee d \vee b\}$ a CNF formula and $Var(\Psi) = \{a, b, c, d\}$.
The complete models of $\mathscr{M}(\Psi)$ are:

$$\{\neg a, b, \neg c, d\},\ \{\neg a, b, c, \neg d\},\ \{\neg a, b, c, d\},\ \{a, \neg b, \neg c, d\},\ \{a, \neg b, c, \neg d\},\ \{a, \neg b, c, d\}$$

Using the ordering $\tau = (a, b, c, d)$:

- The **1st** model is $\{\neg a, b, \neg c, d\}$ (word 0101).
- The **3rd** model is $\{\neg a, b, c, d\}$ (word 0111).

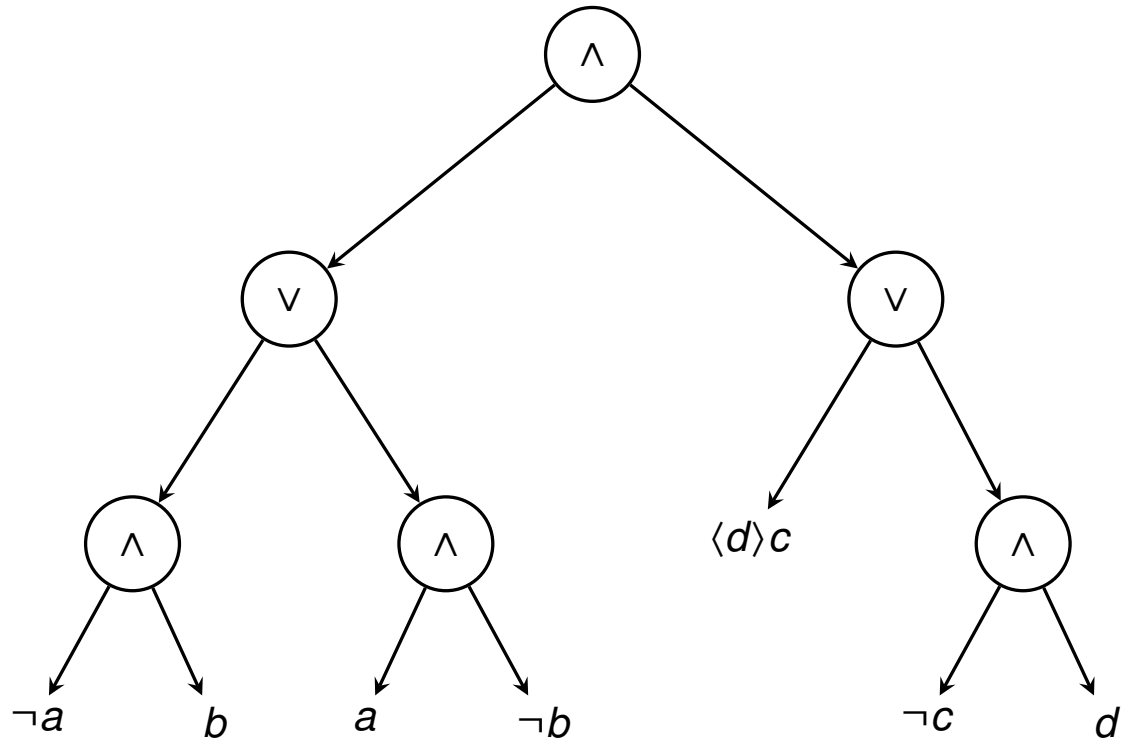
# Direct Access Algorithm

## Algorithm: DIRECT-ACCESS$(\Phi, E, k, \tau)$

```
1   N = #SAT(Φ, E)
2   if N < k:
3     return ERROR
4
5   σ = ∅
6   I = [0, N]  // Interval [I.min, I.max]
7
8   for each x ∈ τ:
9     c_¬x = #SAT(Φ, E ∪ σ ∪ {¬x})
10
11    if I.min + c_¬x > k:
12      σ = σ ∪ {¬x}
13      I = [I.min, I.min + c_¬x]
14    else:
15      σ = σ ∪ {x}
16      I = [I.min + c_¬x, I.max]
17
18  return σ
```

# Running Example: Direct Access Evaluation

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).

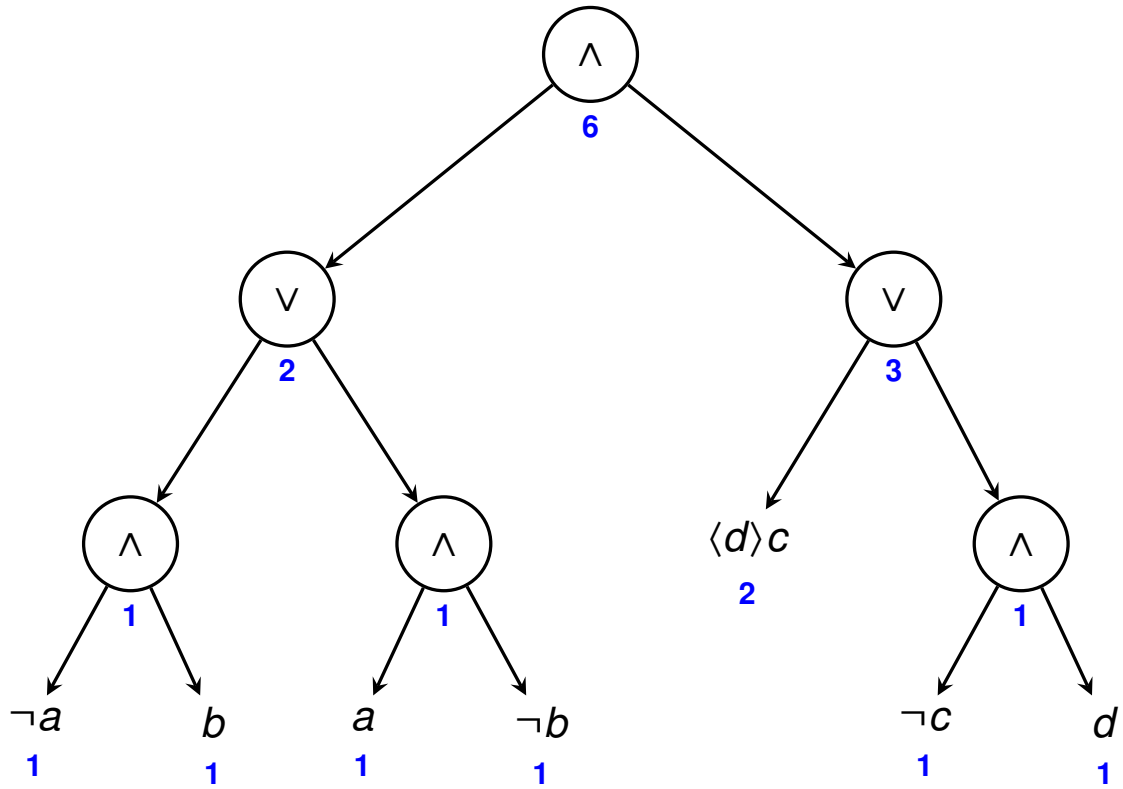


# Running Example: Direct Access Evaluation

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).

- $c = \#SAT(\Psi) = 6$

  $I = [0, 6]$ and $\sigma = \{\}$



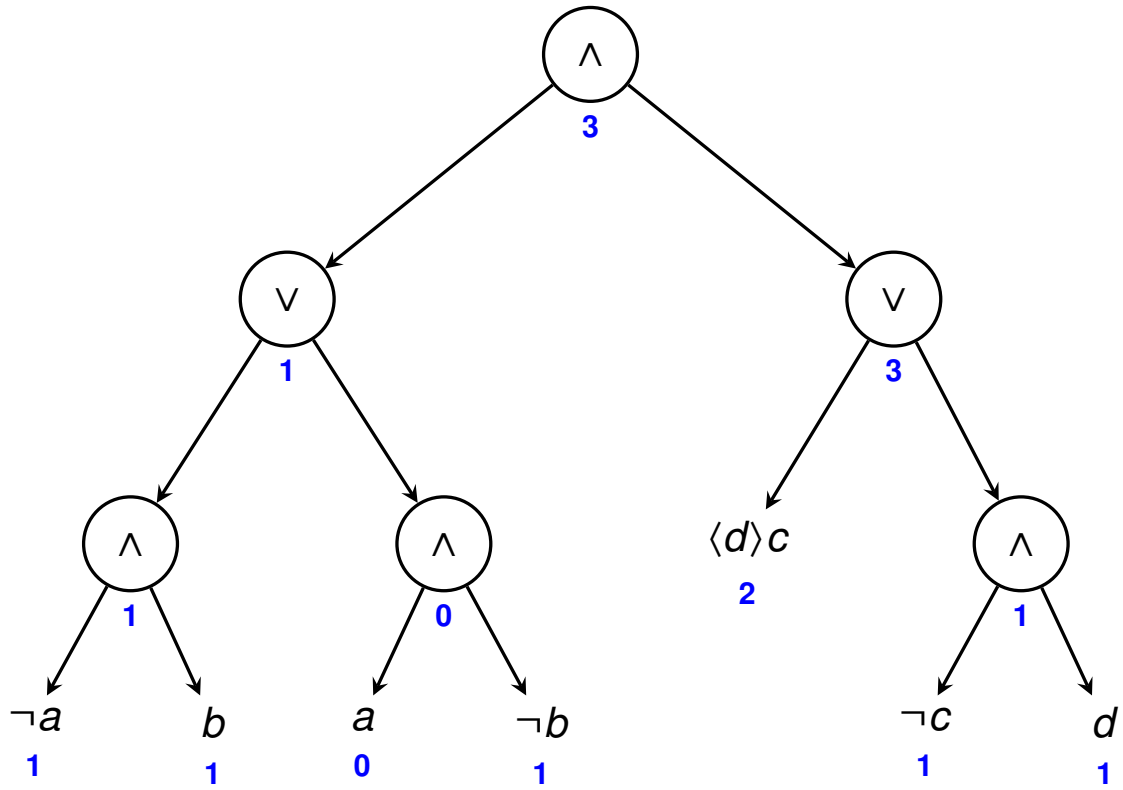
# Running Example: Direct Access Evaluation

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).

- $c = \#SAT(\Psi) = 6$

  $I = [0, 6]$ and $\sigma = \{\}$
- $x = a$**:** $c_{\neg a} = 3 \implies$ select $a$

  $I = [3, 6]$ and $\sigma = \{a\}$



# Running Example: Direct Access Evaluation

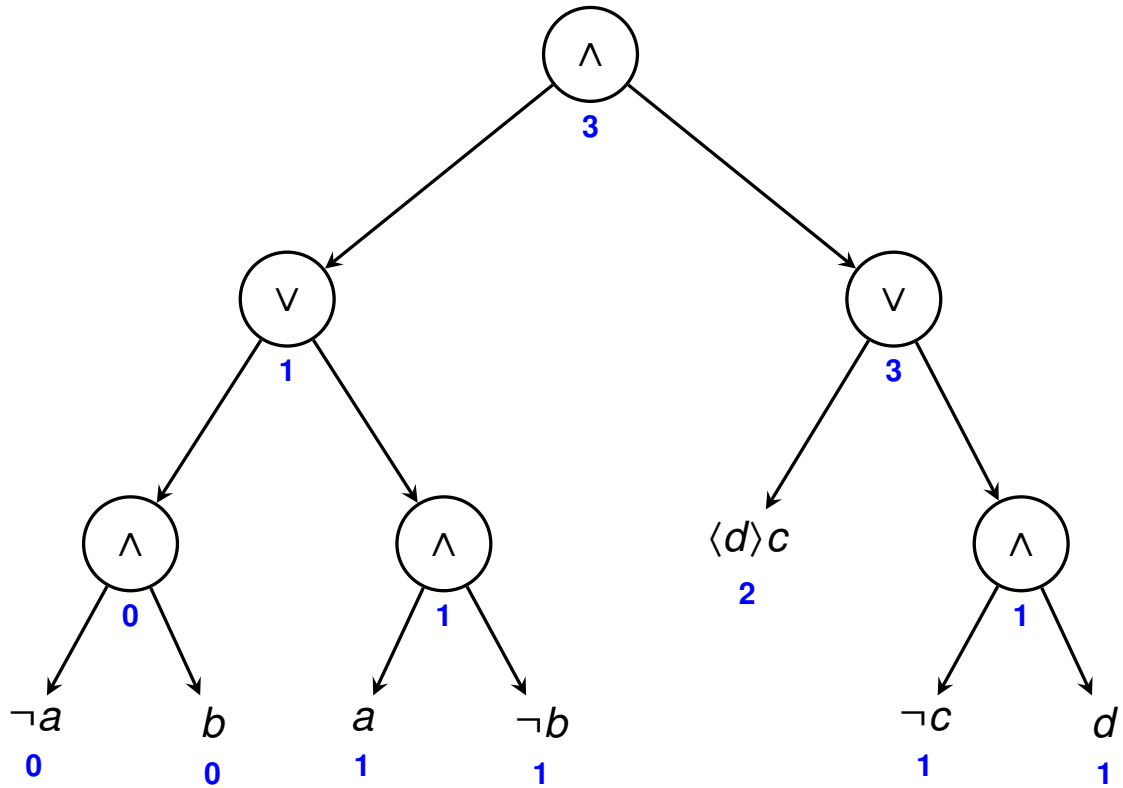
Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).

- $c = \#SAT(\Psi) = 6$

  $I = [0, 6]$ and $\sigma = \{\}$
- $x = a$**:** $c_{\neg a} = 3 \implies$ select $a$

  $I = [3, 6]$ and $\sigma = \{a\}$
- $x = b$**:** $c_{\neg b} = 3 \implies$ select $\neg b$

  $I = [3, 6]$ and $\sigma = \{a, \neg b\}$



# Running Example: Direct Access Evaluation

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).

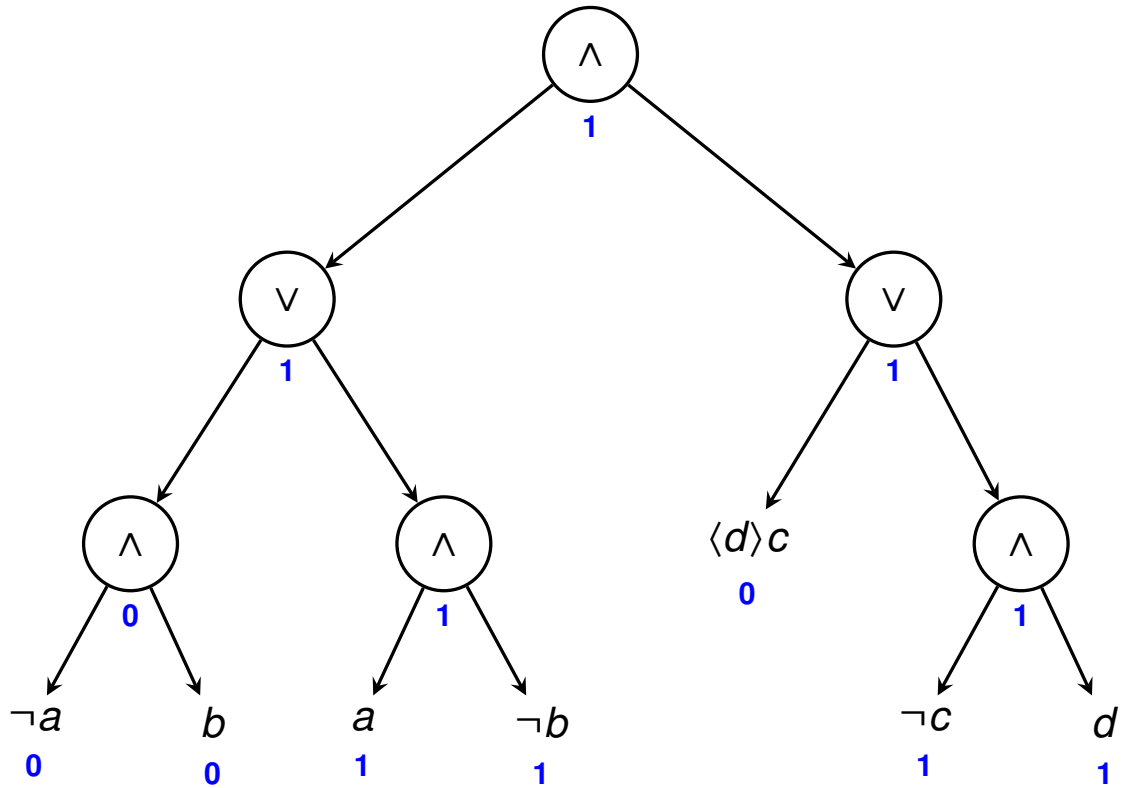
- $c = \#SAT(\Psi) = 6$

  $I = [0, 6]$ and $\sigma = \{\}$
- $x = a$: $c_{\neg a} = 3 \implies$ select $a$

  $I = [3, 6]$ and $\sigma = \{a\}$
- $x = b$: $c_{\neg b} = 3 \implies$ select $\neg b$

  $I = [3, 6]$ and $\sigma = \{a, \neg b\}$
- $x = c$: $c_{\neg c} = 1 \implies$ select $\neg c$

  $I = [3, 4]$ and $\sigma = \{a, \neg b, \neg c\}$



# Running Example: Direct Access Evaluation

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).

- $c = \#SAT(\Psi) = 6$

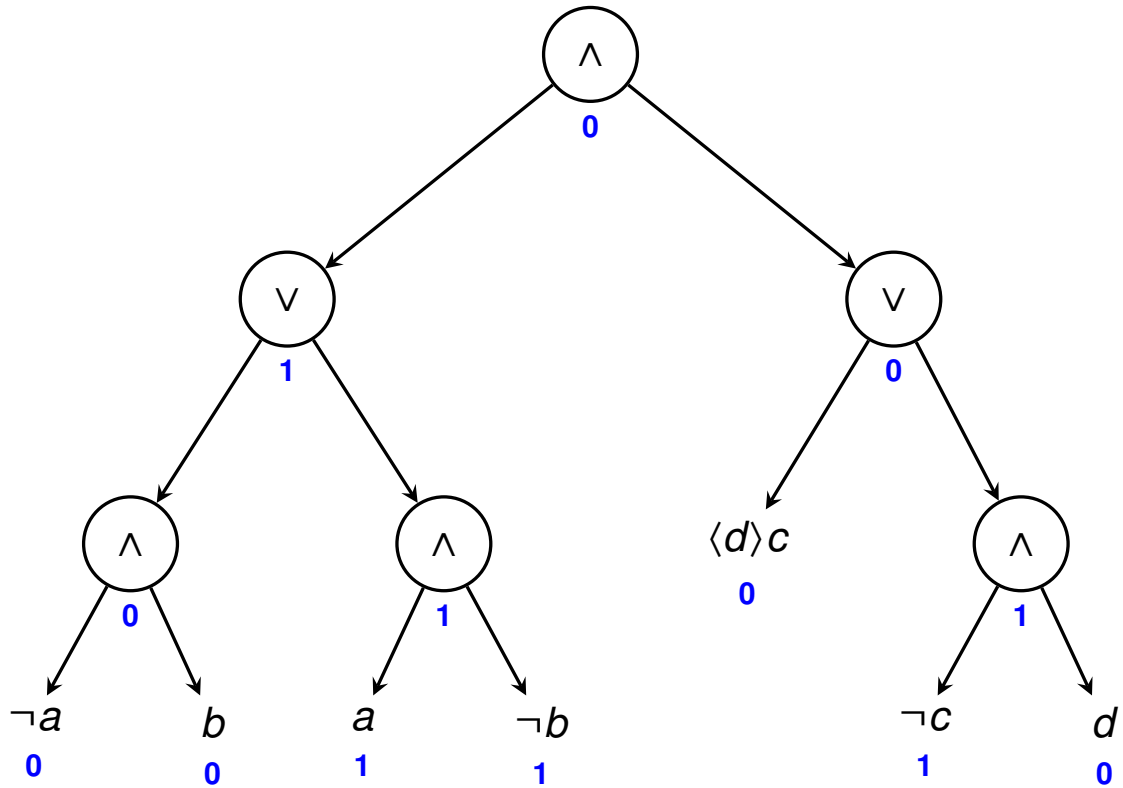
  $I = [0, 6]$ and $\sigma = \{\}$
- $x = a$: $c_{\neg a} = 3 \implies$ select $a$

  $I = [3, 6]$ and $\sigma = \{a\}$
- $x = b$: $c_{\neg b} = 3 \implies$ select $\neg b$

  $I = [3, 6]$ and $\sigma = \{a, \neg b\}$
- $x = c$: $c_{\neg c} = 1 \implies$ select $\neg c$

  $I = [3, 4]$ and $\sigma = \{a, \neg b, \neg c\}$
- $x = d$: $c_{\neg d} = 0 \implies$ select $d$

  $I = [4, 4]$ and $\sigma = \{a, \neg b, \neg c, d\}$

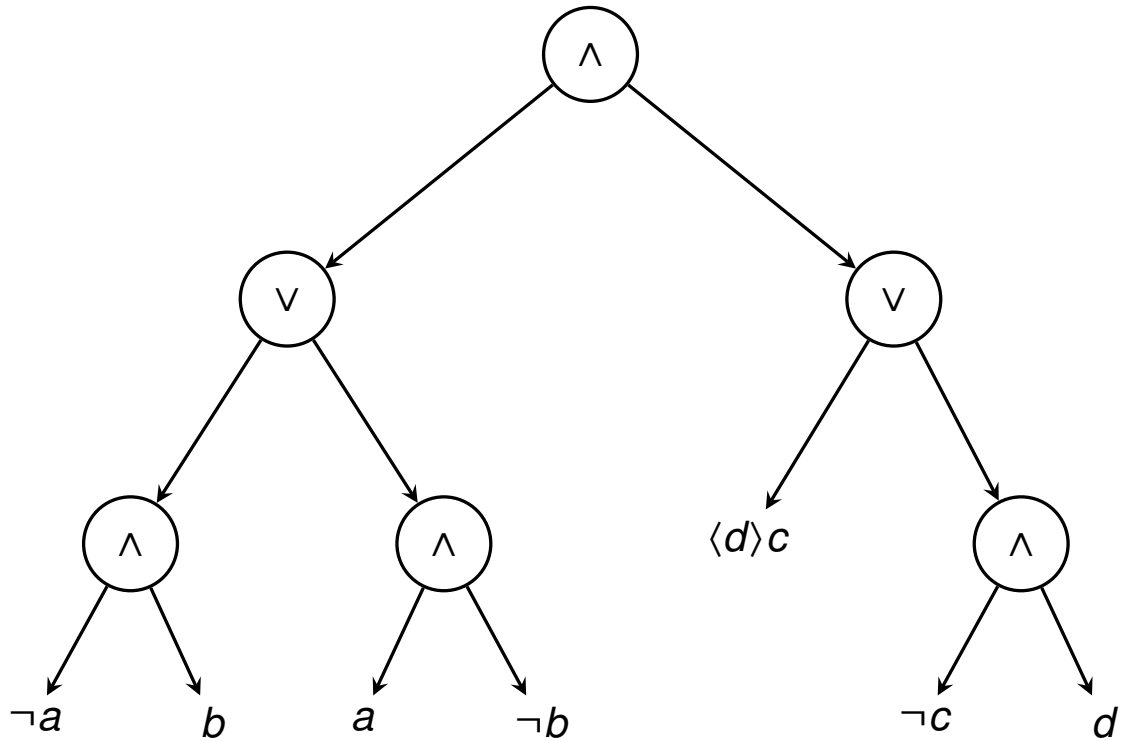


# Running Example: Direct Access Evaluation

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).

- $c = \#SAT(\Psi) = 6$
  $I = [0, 6]$ and $\sigma = \{\}$
- $x = a$**:** $c_{\neg a} = 3 \implies$ select $a$
  $I = [3, 6]$ and $\sigma = \{a\}$
- $x = b$**:** $c_{\neg b} = 3 \implies$ select $\neg b$
  $I = [3, 6]$ and $\sigma = \{a, \neg b\}$
- $x = c$**:** $c_{\neg c} = 1 \implies$ select $\neg c$
  $I = [3, 4]$ and $\sigma = \{a, \neg b, \neg c\}$
- $x = d$**:** $c_{\neg d} = 0 \implies$ select $d$
  $I = [4, 4]$ and $\sigma = \{a, \neg b, \neg c, d\}$

- **Result:**
  $\sigma = \{a, \neg b, \neg c, d\}$



# Outline

# Uniform Sampling: The Main Loop

## The Two Retrieval Approaches

- **Index-based:** Uses Direct Access. Ensures strict, seed-based reproducibility regardless of the underlying DAG topology (Lagniez and Lonca, 2025).
- **Structural Top-Down:** Annotates nodes with model counts $c(N)$ and recursively splits a target index $k$ through the gates (Sharma et al., 2018).

## Algorithm: `UniformSample`$(\Psi, E, m)$

```
1    N = #SAT(Ψ, E)
2    R = GenerateRandomIndices(m, N)
3
4    Δ = ∅
5    for each k ∈ R:
6      Δ = Δ ∪ {Retrieve(Ψ, E, k)}
7
8    return Δ
```

## Running Example: Sampling $m = 2$ models

Assume our random sequence generator produces $R = \{2, 4\}$.

- **Index-based:** Retrieves $\{\neg a, b, c, \neg d\}$ and $\{a, \neg b, \neg c, d\}$.
- **Structural:** The output depends on the DAG structure.

# Structural Retrieve Model

After annotating the DAG with the model count for each node.

## Algorithm: `Retrieve`$(N, k)$

$\sigma = \varnothing$

if $N$ has free variables $V$:
  $k_N = 1 + \lfloor (k-1)/2^{|V|} \rfloor$
  $k_V = 1 + ((k-1) \bmod 2^{|V|})$
  $\sigma = \sigma \cup \{x_i$ if $i$-th bit of $(k_V - 1)$ is 1 else $\neg x_i \mid \forall x_i \in V\}$
  $k = k_N$ // Update target $k$ to process the core node $N$

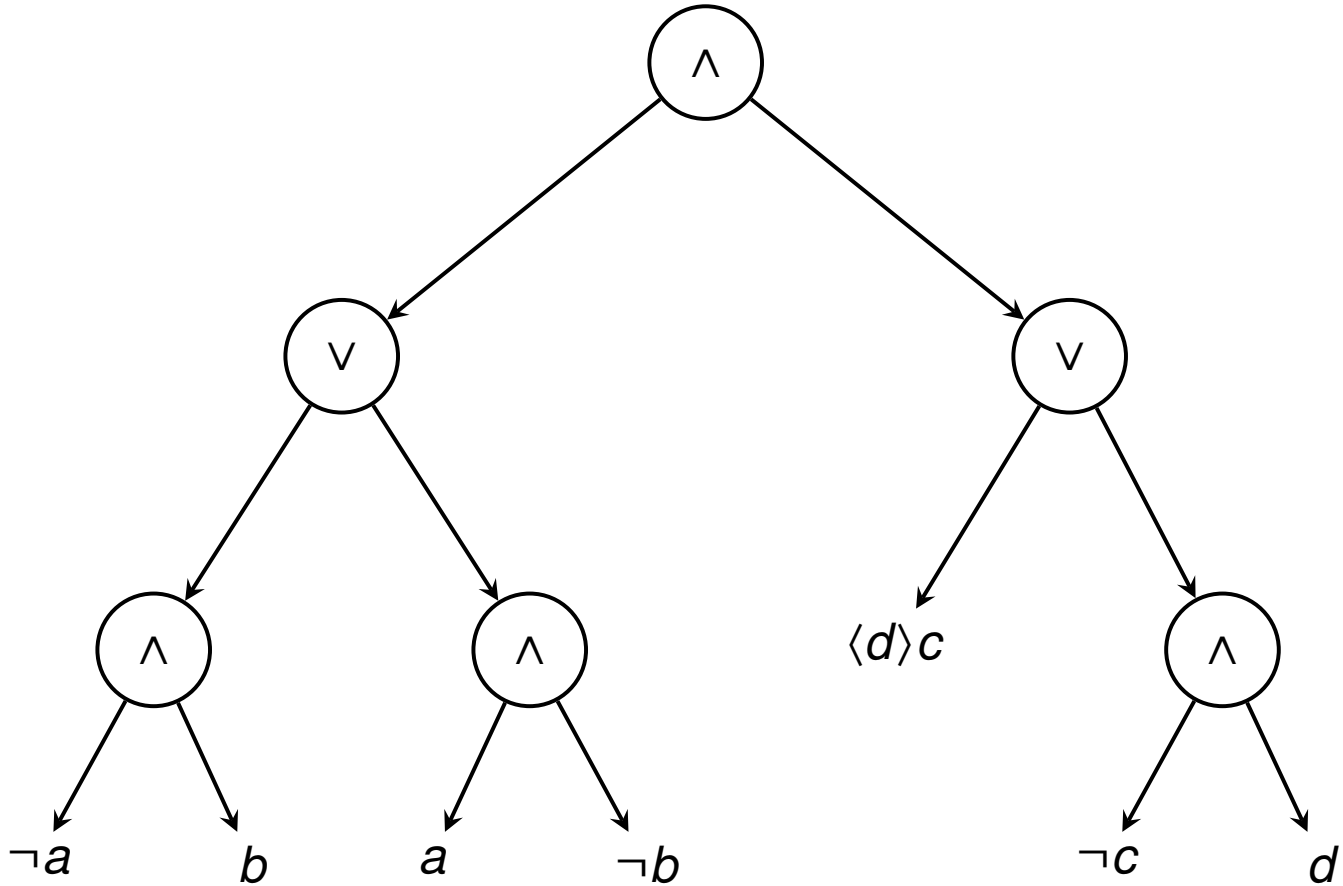
if $N$ is a literal leaf $\ell$:
  return $\sigma \cup \{\ell\}$

if $N == N_\ell \vee N_r$:
  if $k \leq c(N_\ell)$:
    return $\sigma \cup$ `Retrieve`$(N_\ell, k)$
  else:
    return $\sigma \cup$ `Retrieve`$(N_r, k - c(N_\ell))$

if $N == N_\ell \wedge N_r$:
  return $\sigma \cup$ `Retrieve`$(N_\ell, 1 + \lfloor (k-1)/c(N_r) \rfloor) \cup$ `Retrieve`$(N_r, 1 + ((k-1) \bmod c(N_r)))$

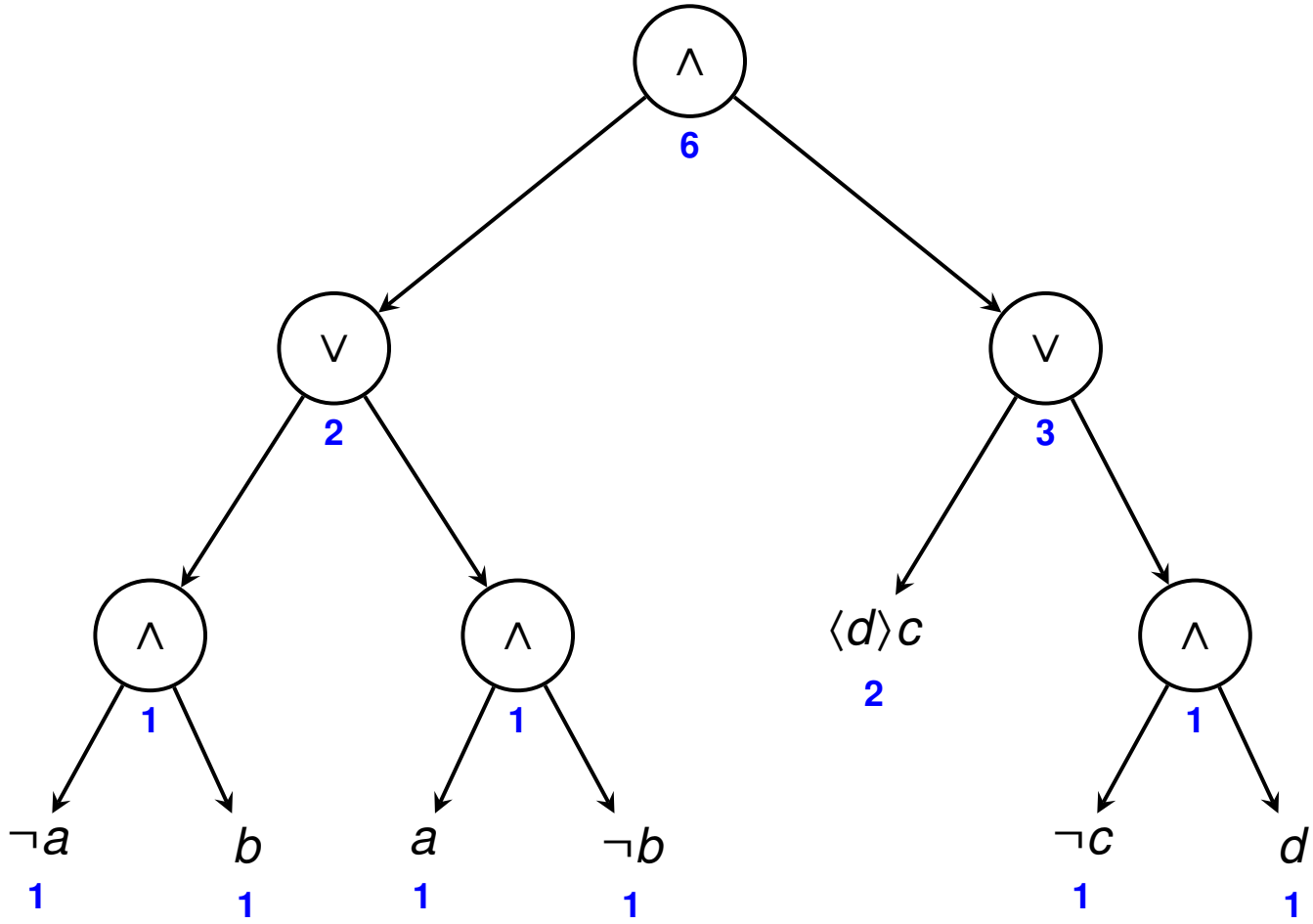
# Running Example: Structural Retrieval

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).



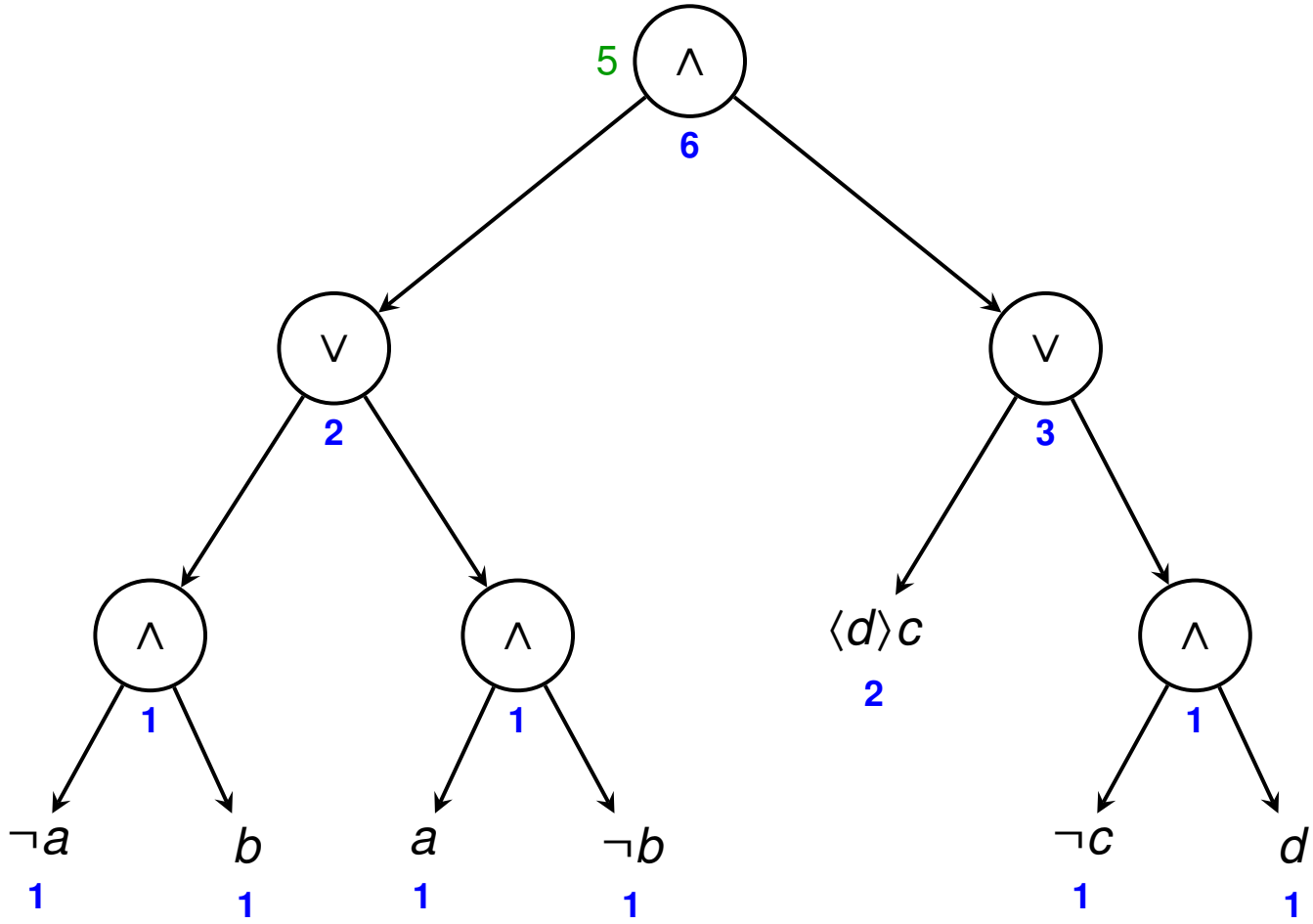
# Running Example: Structural Retrieval

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).



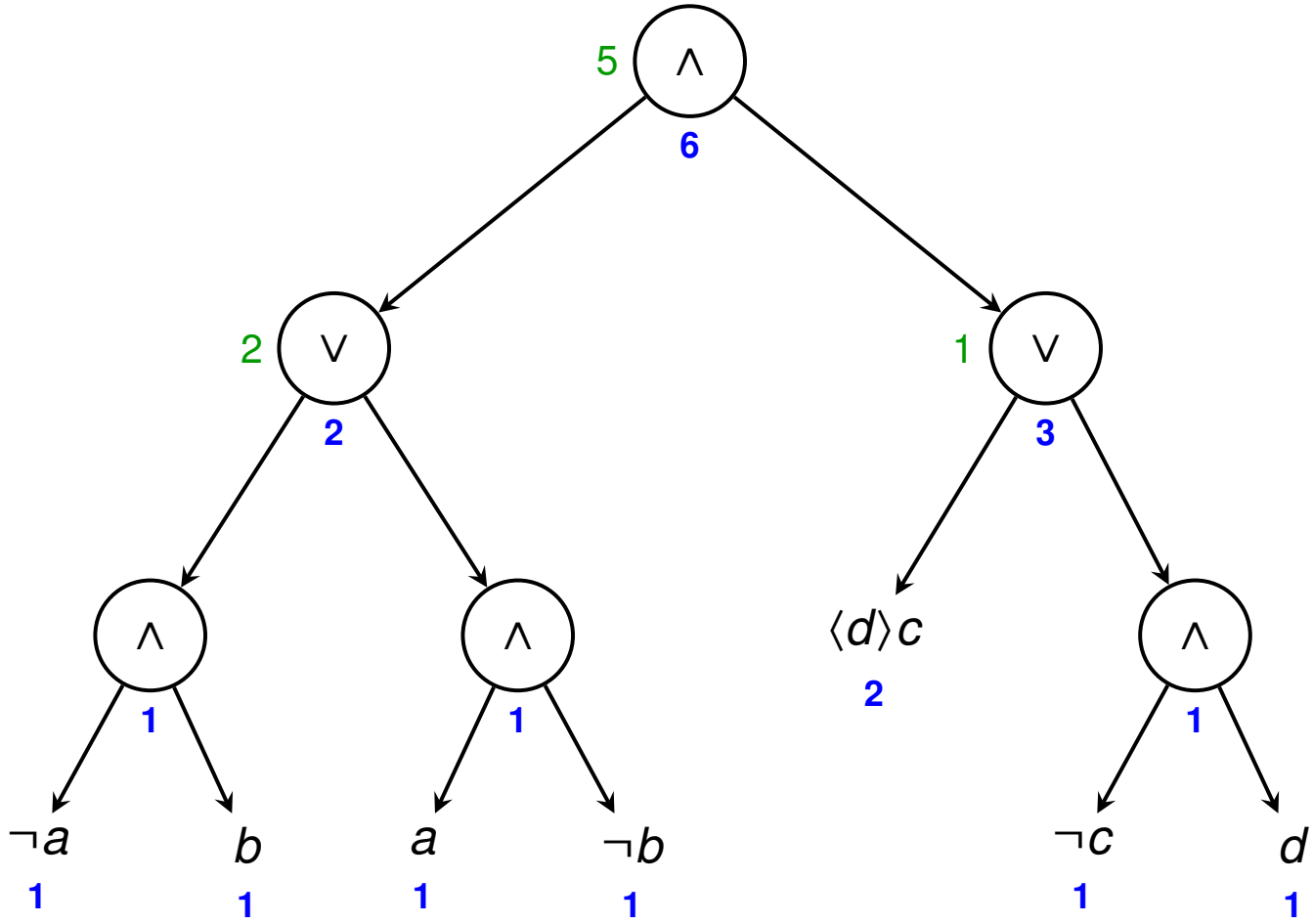
# Running Example: Structural Retrieval

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).



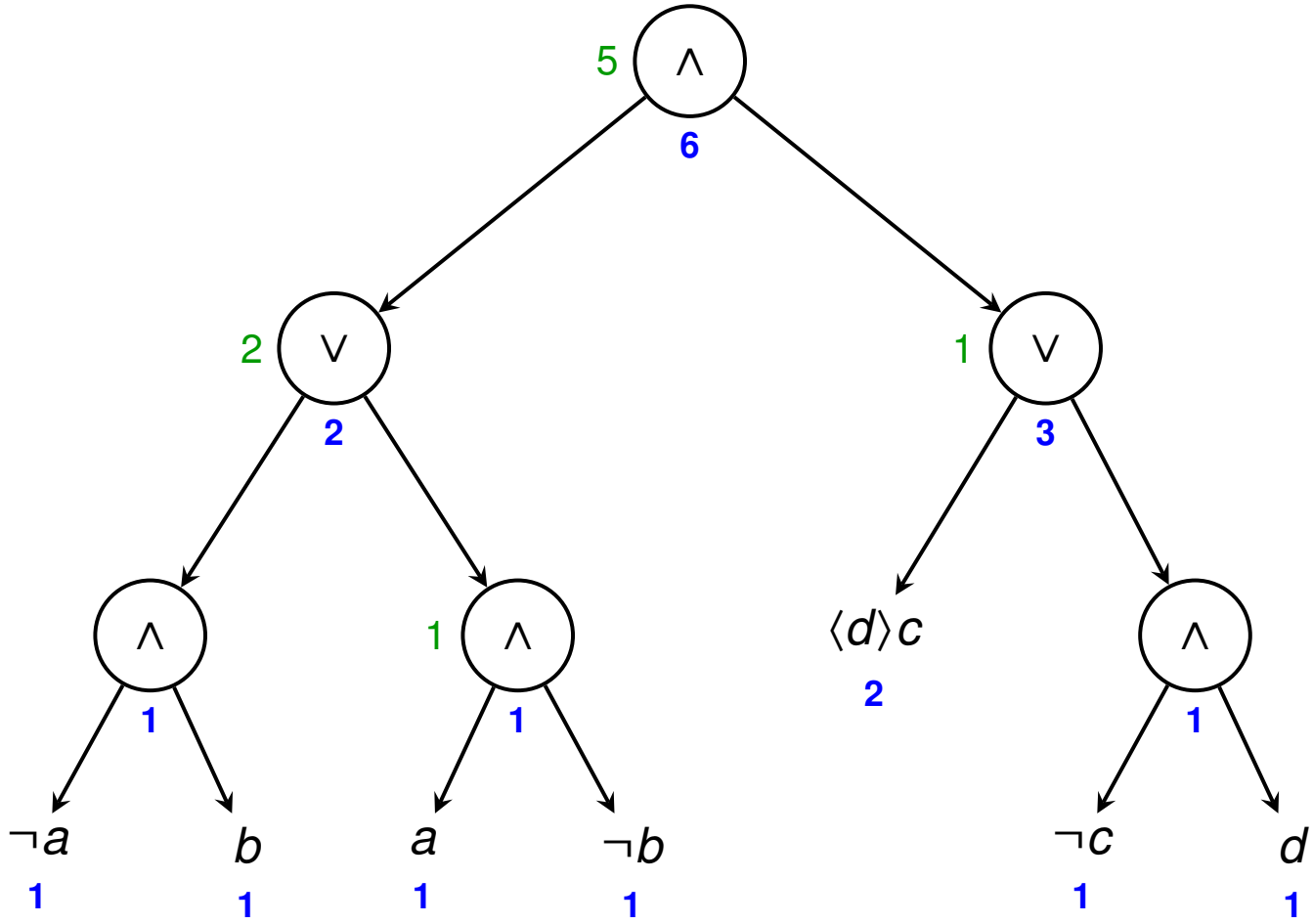
# Running Example: Structural Retrieval

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).



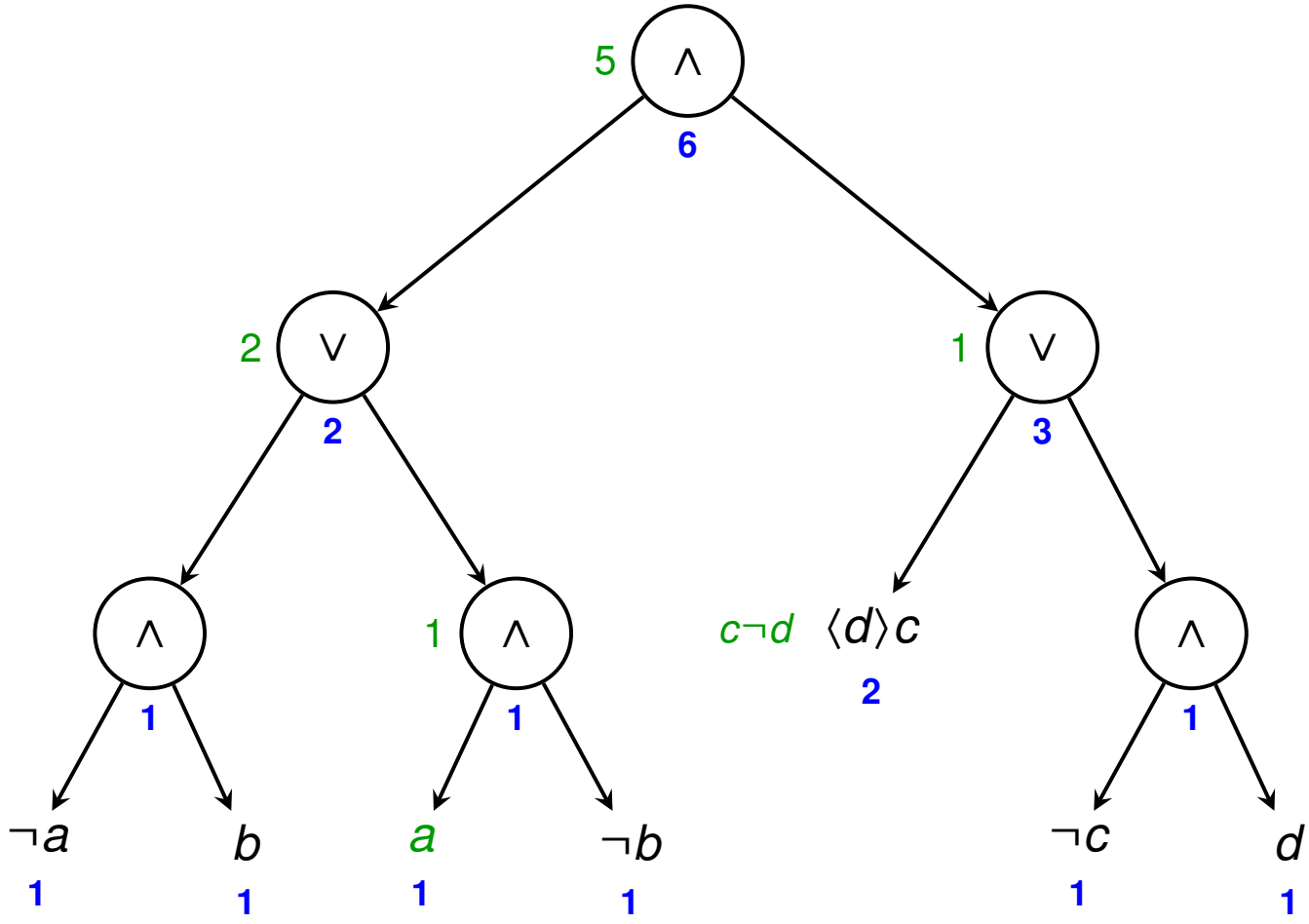
# Running Example: Structural Retrieval

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).



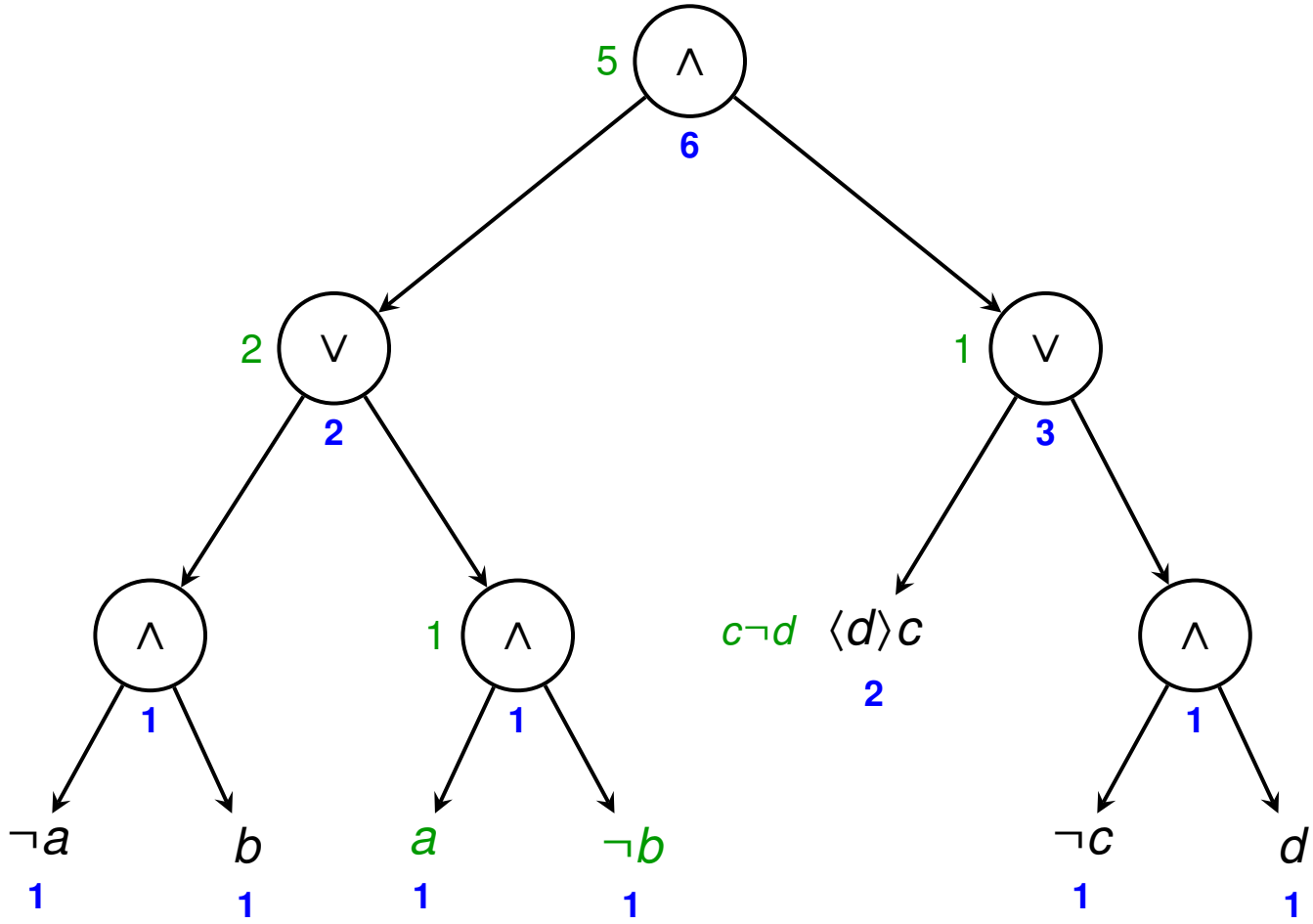
# Running Example: Structural Retrieval

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).



# Running Example: Structural Retrieval

Consider $\Psi$ with $\tau = (a, b, c, d)$. We want to retrieve the **4th** model ($k = 4$).



# Outline

# Conclusion and References

# Concluding Remarks and Advanced Topics

## What We Left Out

While we covered the core algorithms for compiling and querying `d-DNNF` and `SDD`, state-of-the-art compilers rely on several advanced techniques for practical scalability:

- **Formula Preprocessing:** Applying techniques like variable elimination, subsumption, and equivalence reasoning *before* compilation to drastically shrink the initial search space (Lagniez and Marquis, 2017b).
- **Beyond CNF:** Compiling richer formalisms directly, such as Constraint Satisfaction Problems (CSP) or Pseudo-Boolean constraints, to avoid the structural bloat of forcing everything into CNF encodings (Koriche et al., 2015).
- **Parallel Compilation:** Leveraging multi-core architectures to distribute independent compilation tasks (e.g., evaluating disjoint sub-trees or parallelizing the APPLY operator).

**Thank you for your attention!**

*Any questions?*

# Knowledge Compilation: Theory, Practice, and Applications

Jean-Marie Lagniez[1] Johannes Fichte[2]

[1] CRIL, Univ. Artois & CNRS, France
[2] Linköping University, Sweden

# References


Antoine Amarilli, Florent Capelli, Mikaël Monet, and Pierre Senellart. Connecting knowledge compilation classes width parameters. *Theory of Computing Systems*, 64(5):861–914, 2020. doi: 10.1007/s00224-019-09930-2. URL `https://doi.org/10.1007/s00224-019-09930-2`.

Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Just-in-time compilation of knowledge bases. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pages 447–453, 2013.

Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. Symmetry-driven decision diagrams for knowledge compilation. In *ECAI 2014 - 21st European Conference on Artificial Intelligence*, pages 51–56, 2014.

Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. An improved CNF encoding scheme for probabilistic inference. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016*, pages 613–621, 2016.

Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973. ISSN 0097-3165. doi: https://doi.org/10.1016/0097-3165(73)90016-2. URL `https://www.sciencedirect.com/science/article/pii/0097316573900162`.

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. 2021.