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

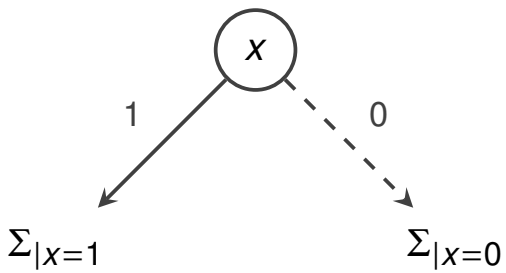
Outline

Outline

The Decision Tree (DT)

- **Shannon Expansion:** A formula Σ can be split on any variable x:

$$\Sigma \equiv (x \wedge \Sigma_{|x=1}) \vee (\neg x \wedge \Sigma_{|x=0})$$



- A Decision Tree is built by recursively applying Shannon Expansion.
- **Logically:** It is the joined representation of a deterministic DNF for Σ and a deterministic DNF for $\neg\Sigma$.

Decision Tree Compiler

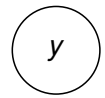
Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$

Decision Tree Compiler

Formula

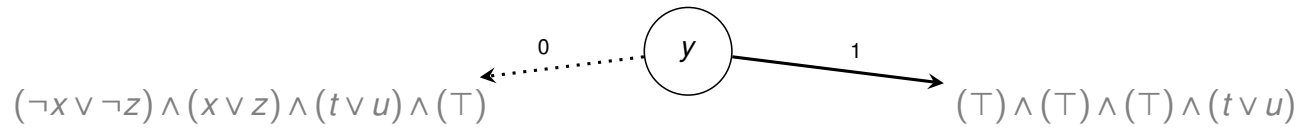
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

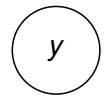
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

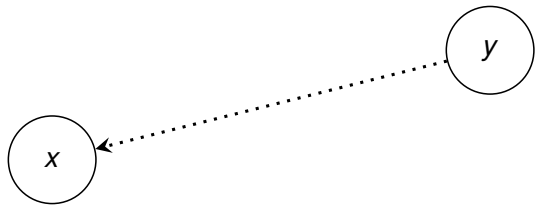
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

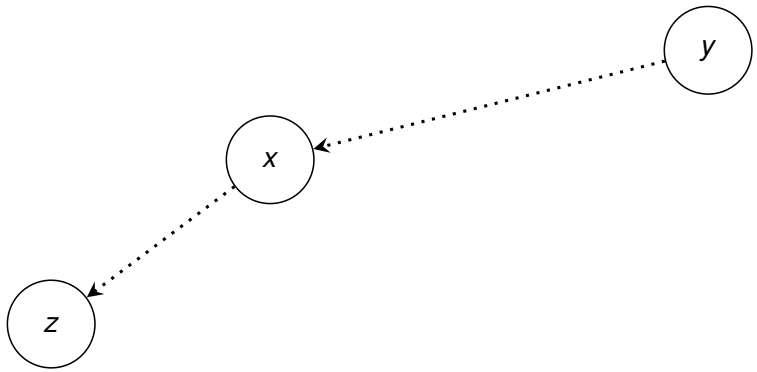
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

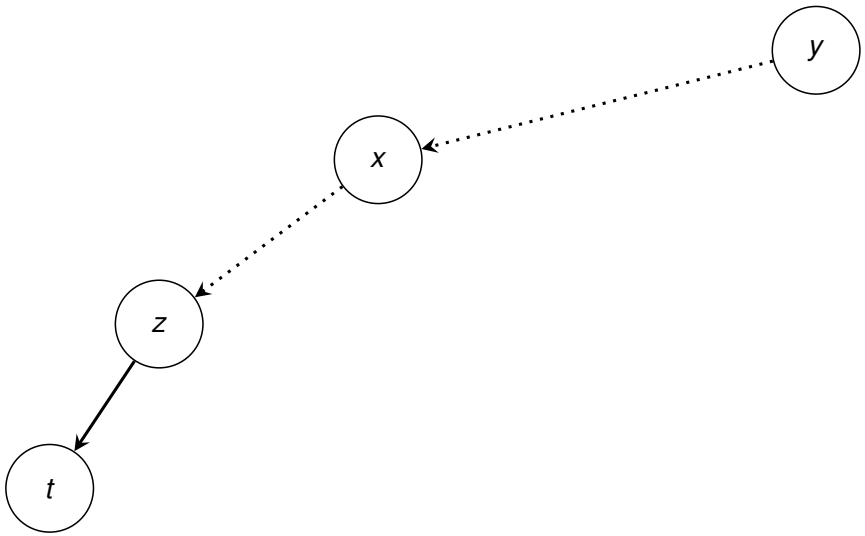
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

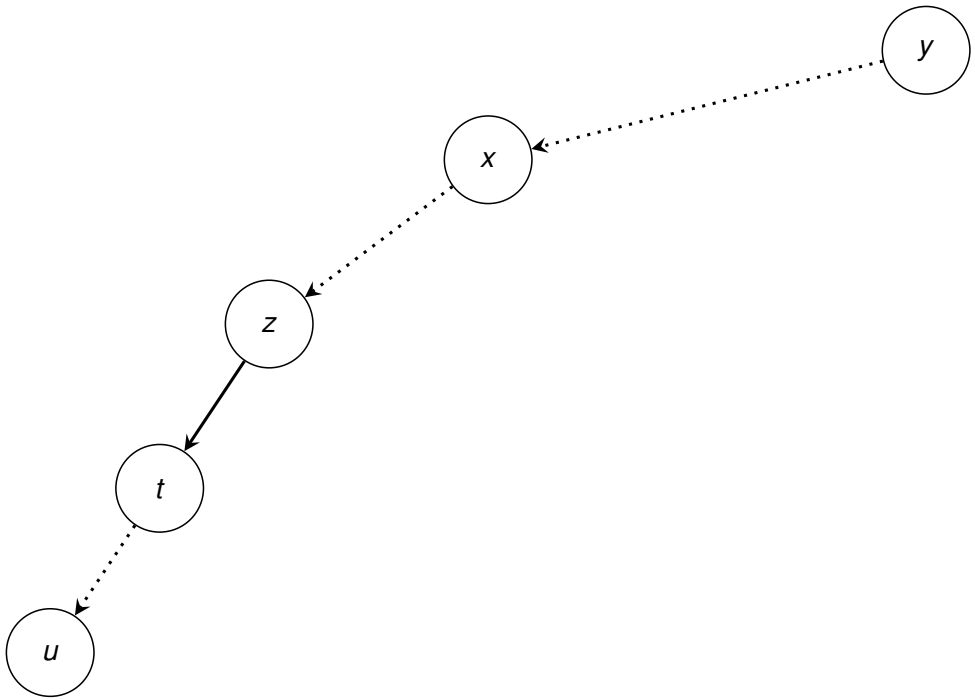
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

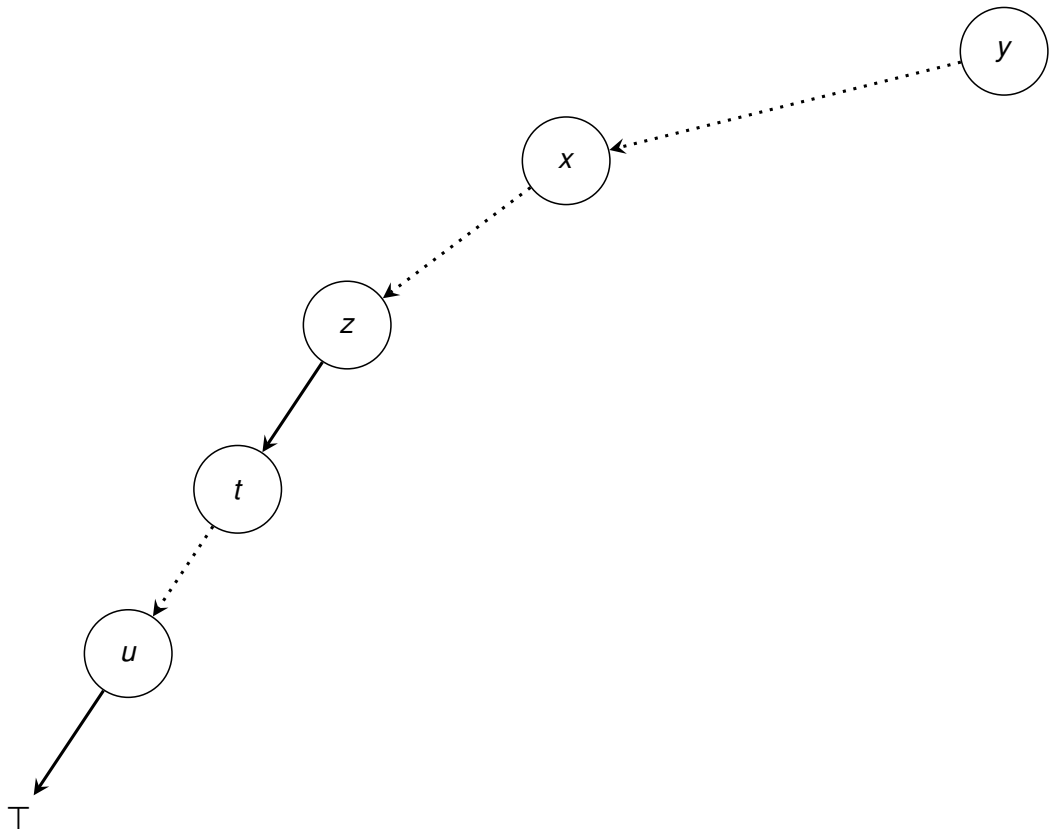
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

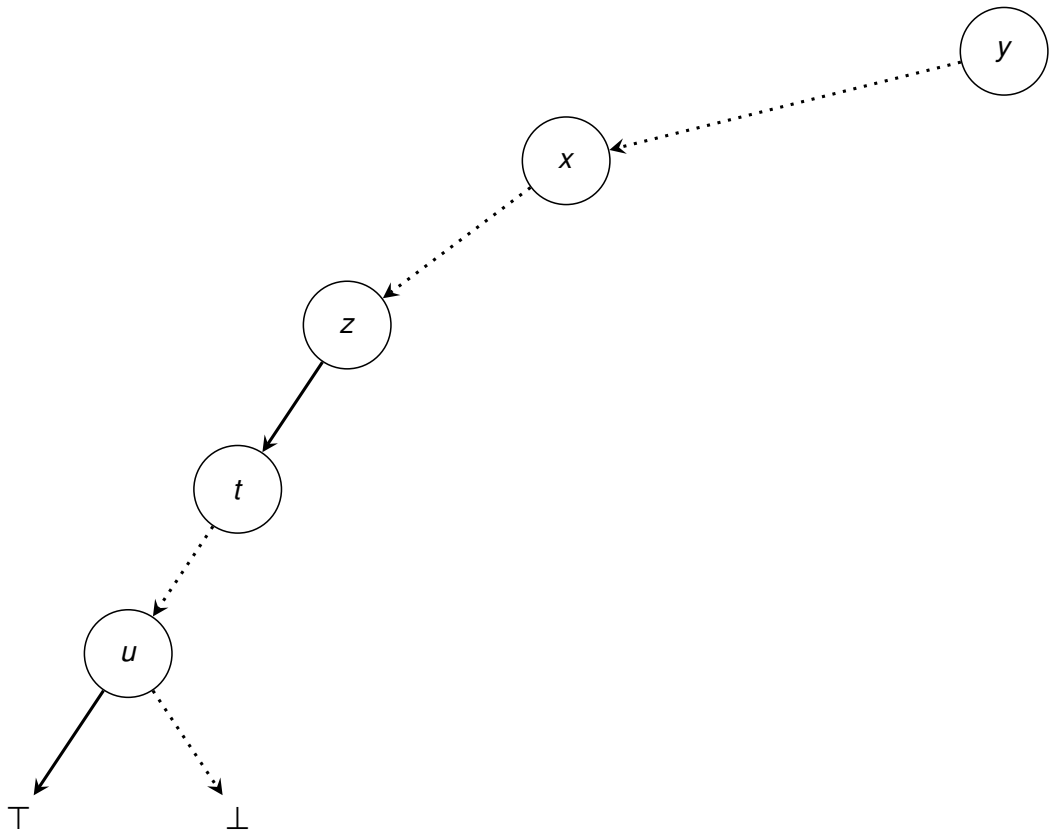
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

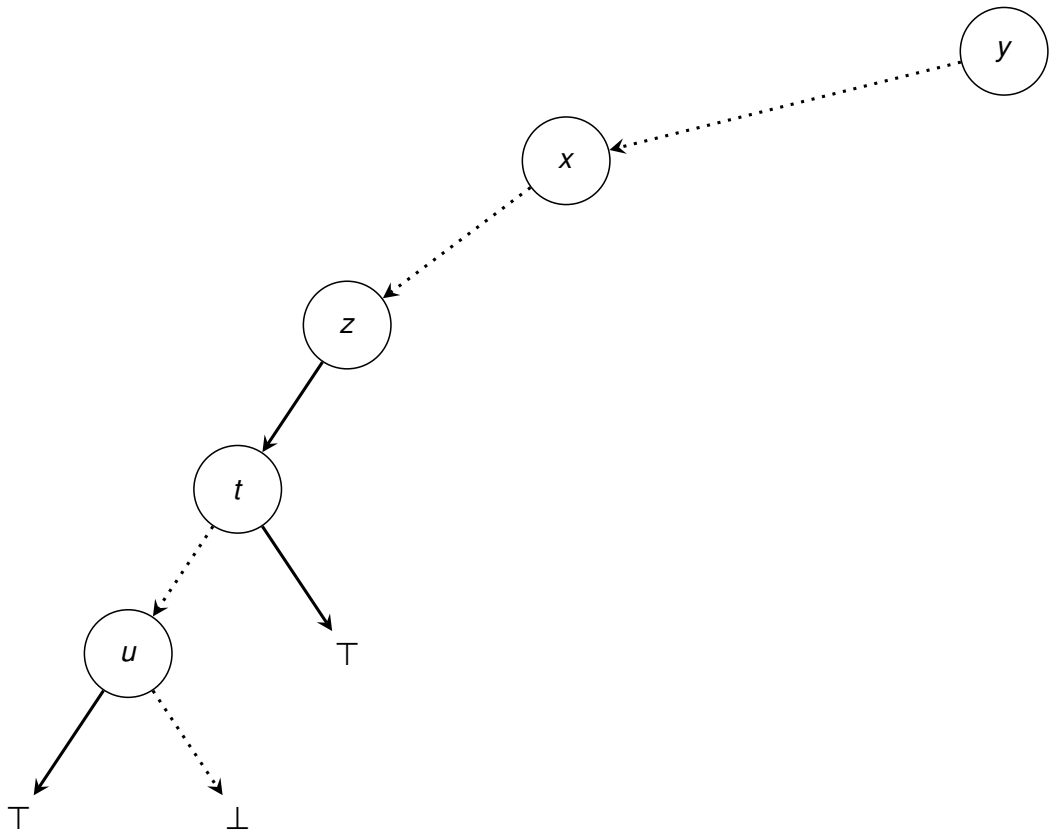
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

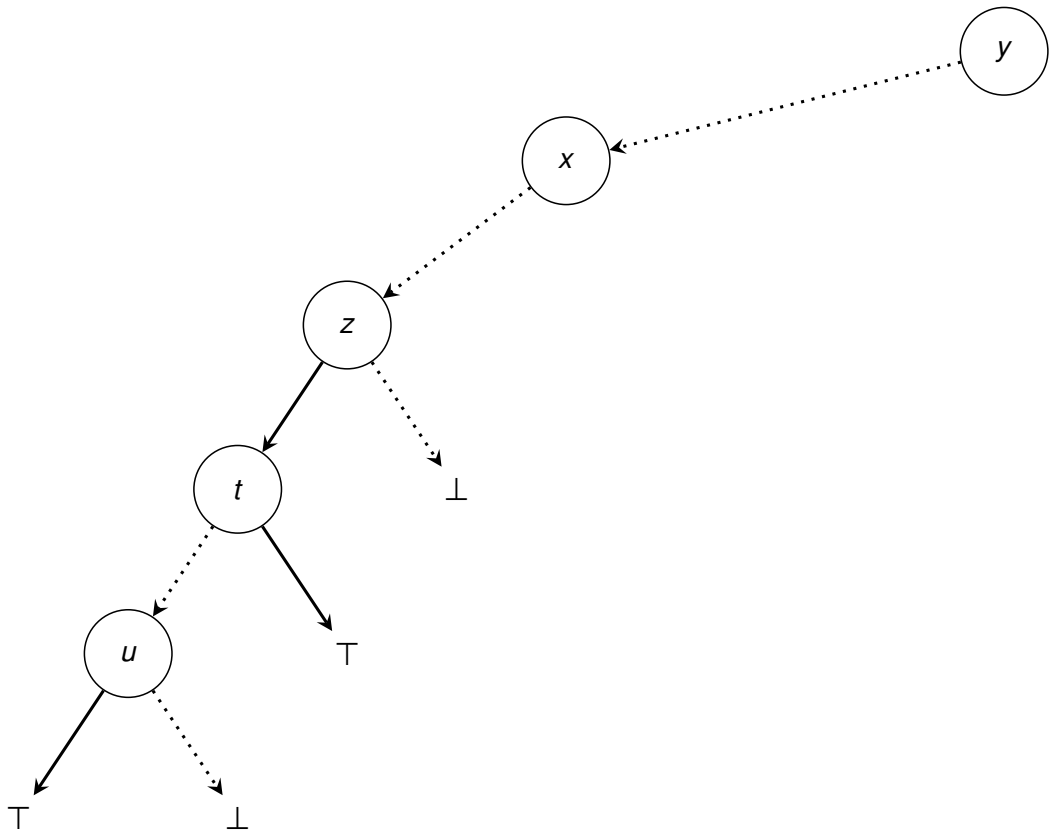
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

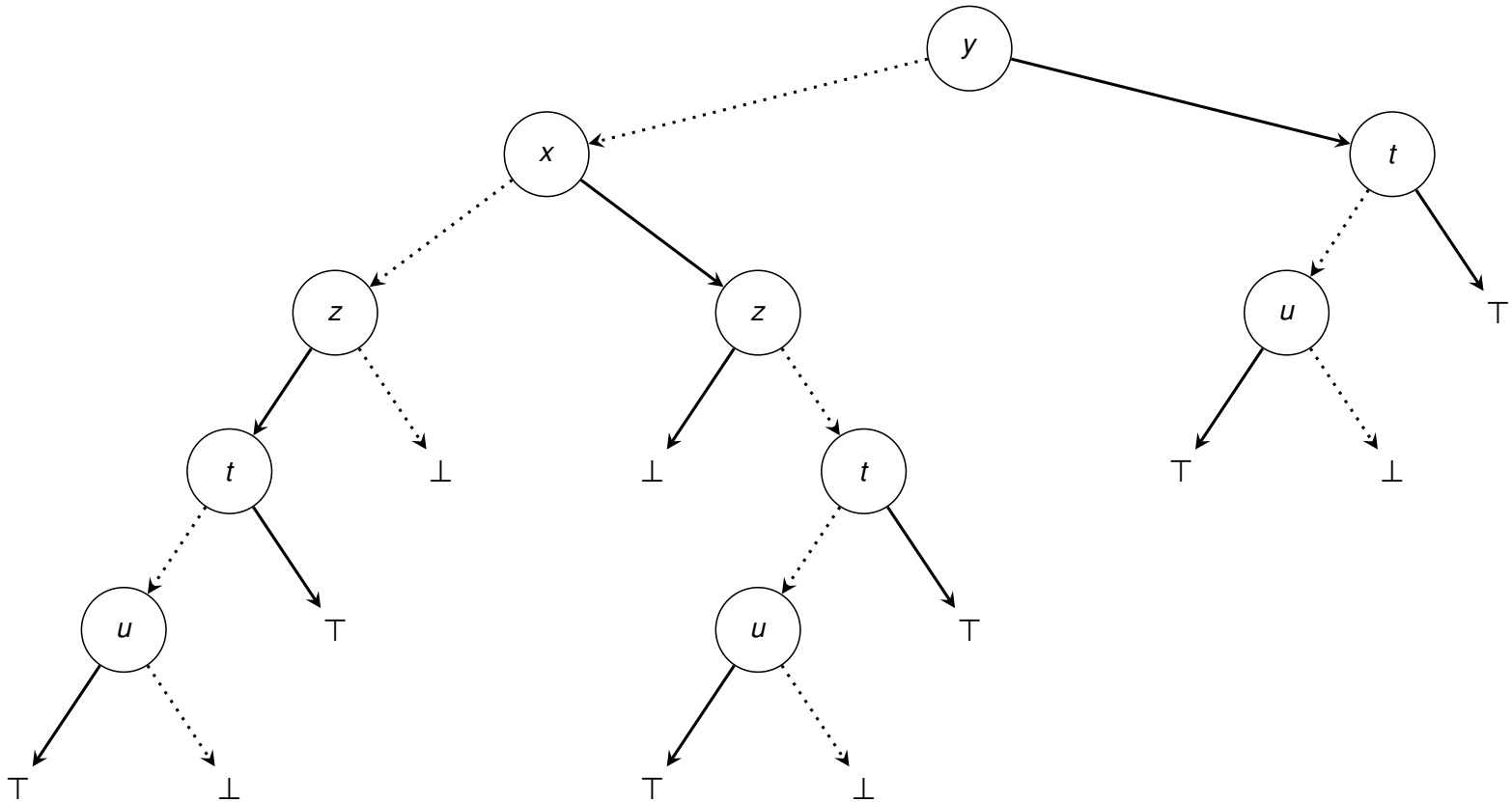
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

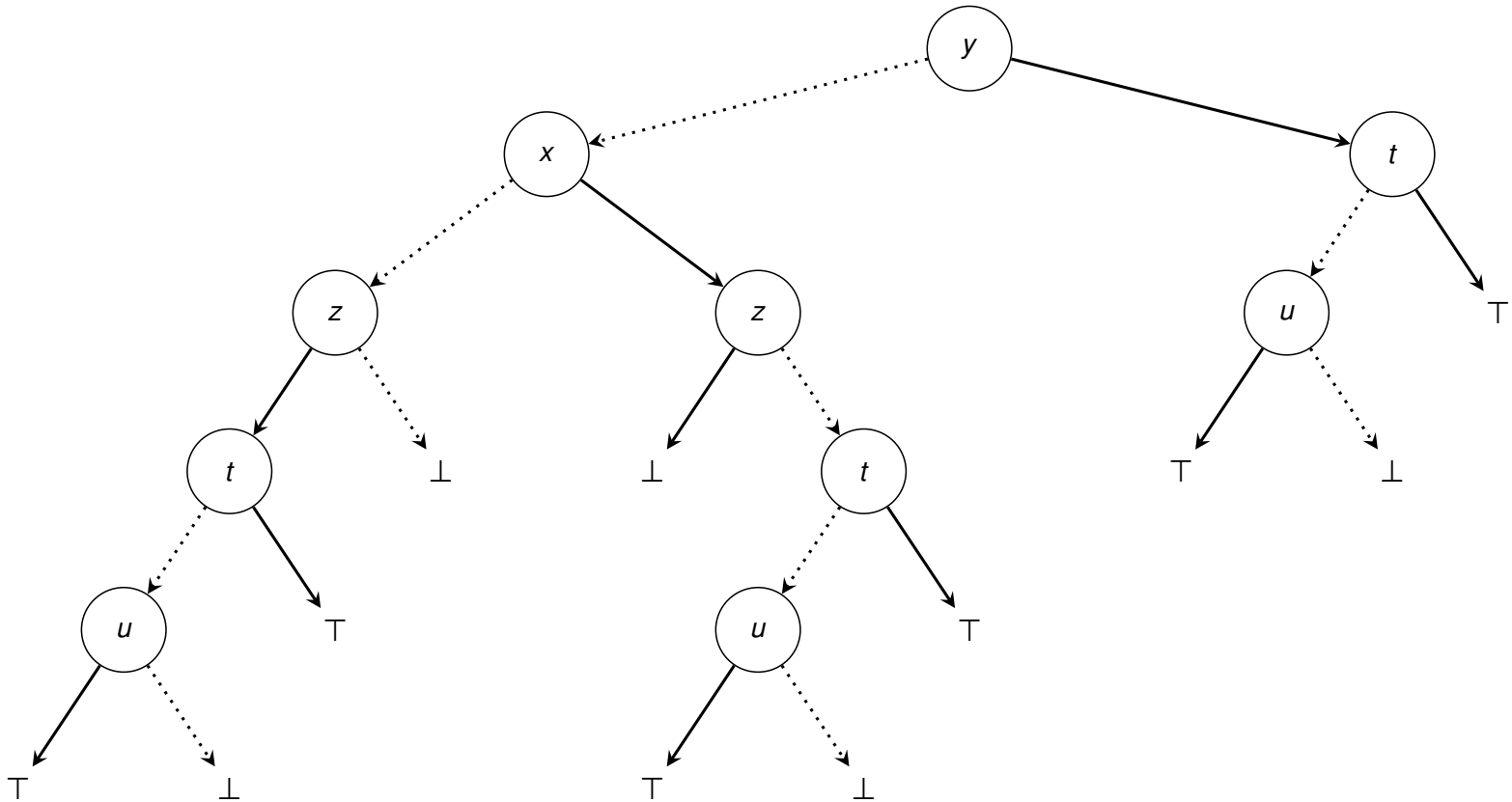
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision Tree Compiler

Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



- **Problem:** The tree size is exponential in the number of variables (2^n). We re-solve similar sub-problems (like $t \vee u$) multiple times.

The KC Map for DT: Queries

$\mathscr{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
Circ	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
ODNF	√	√	√	√	√	√	√	√
MODS	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√

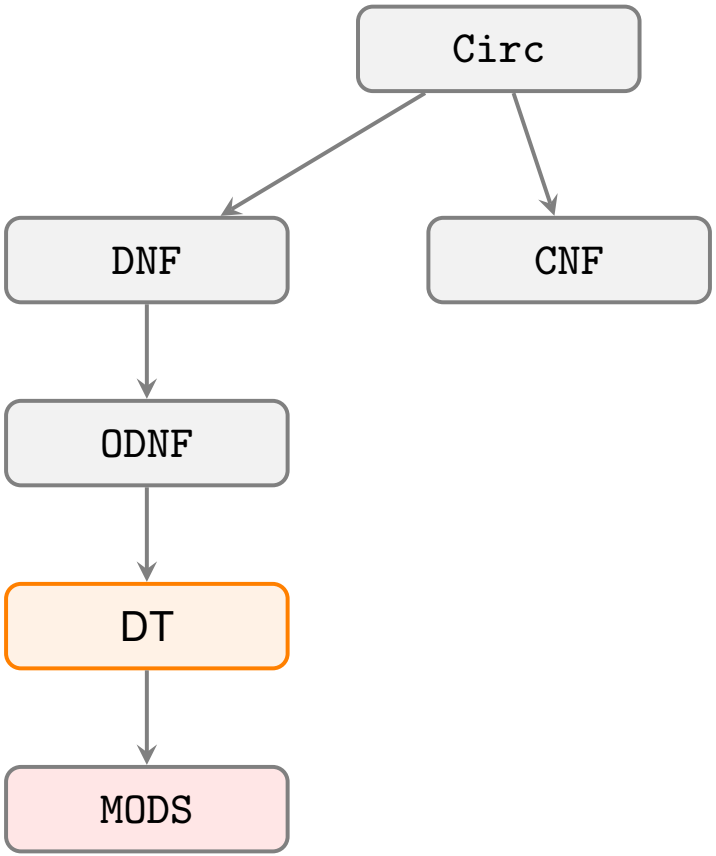
- **A Powerful Result:** Decision Trees (DT) satisfy **all** the queries offered by Ordered Binary Decision Diagrams (OBDD$_<$).
- **Why?** Because DT is structurally a tree, computing the intersection of two trees T_1, T_2 yields a size bounded by $\mathcal{O}(|T_1| \times |T_2|)$. This makes complex binary queries like Equivalence (**EQ**) and Sentential Entailment (**SE**) strictly polynomial!

The KC Map for DT: 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	$\surd$	$\surd$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$
DT	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$

- **Inheriting OBDD$_<$ Traits:**
  - **Negation ($\neg$C):** Trivial $\surd$. Just swap the $\top$ and $\bot$ leaves.
  - **Bounded Conjunction/Disjunction ($\wedge$BC, $\vee$BC):** $\surd$, because the product of a *constant number* of trees remains polynomial.
  - **Singleton Forgetting (SFO):** $\surd$. Existential quantification reduces to applying an OR operation on the branches, which is poly-time due to $\vee$**BC**.
- **Weakness:** Unbounded operations (**FO**, $\wedge$**C**, $\vee$**C**) remain $\circ$.

Succinctness: The Problem with Trees



Why is ODNF $<_s$ DT?

An ODNF allows terms of different lengths to coexist natively. A DT forces rigid, nested branching on variables, which can require exponentially more space to represent the exact same orthogonal logic.

The Tree Bottleneck

While DT has incredible query power (matching $\text{OBDD}_<$), it lacks a DAG structure. **Isomorphic subgraphs cannot be shared**; they must be duplicated in memory. This lack of compression is why we must transition to DAG-based languages!

Outline

The Achilles Heel of DT: The Parity Function

- Consider the Boolean Parity function (which evaluates to True if an even number of variables are True):

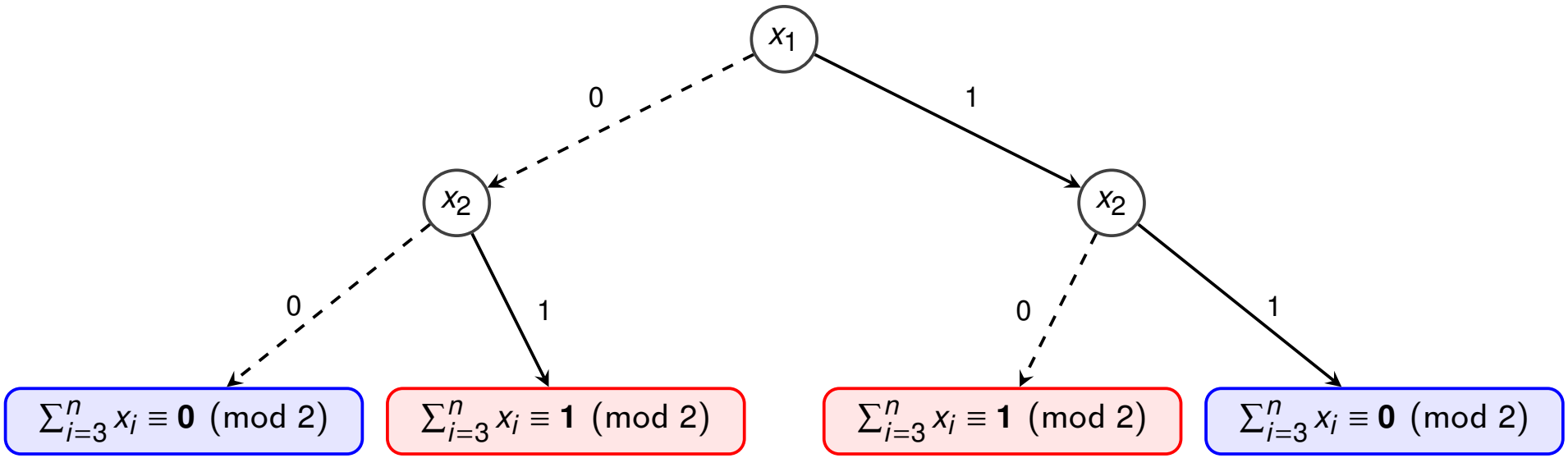
$$\sum_{i=1}^{n} x_i \equiv 0 \pmod 2$$

Why does DT fail here?

- To determine parity, we **must** read the value of every single variable x_i. We can never short-circuit the evaluation.
- In a strict Decision Tree, this forces us to build a complete binary tree of depth n.
- **The Consequence:** The DT will always have $\mathcal{O}(2^n)$ nodes. It effectively degenerates into explicitly listing all interpretations (like `MODS`)!

The Solution: Caching (Sub-circuit Sharing)

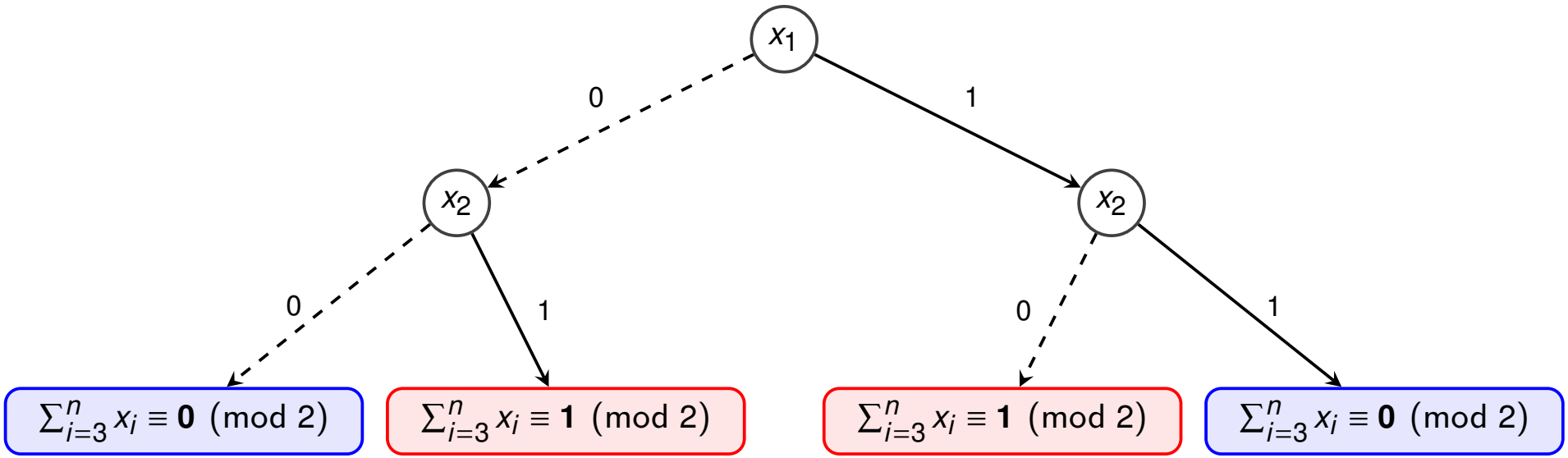
- Let's trace the Shannon expansion of the parity function after two variables.



- **The Observation:** Despite having 4 branches, there are only **two** unique sub-problems remaining!

The Solution: Caching (Sub-circuit Sharing)

- Let's trace the Shannon expansion of the parity function after two variables.



- **The Observation:** Despite having 4 branches, there are only **two** unique sub-problems remaining!
- **The DAG Trick:** By caching and merging isomorphic nodes, the size of this representation drops from $\mathcal{O}(2^n)$ down to just $\mathcal{O}(n)$!

Free Binary Decision Diagram (FBDD)

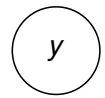
Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$

Free Binary Decision Diagram (FBDD)

Formula

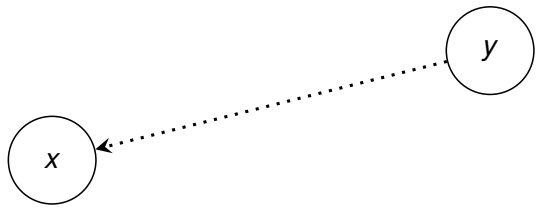
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

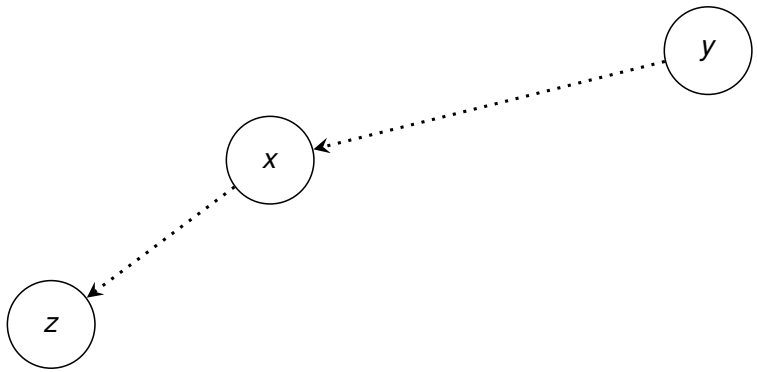
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

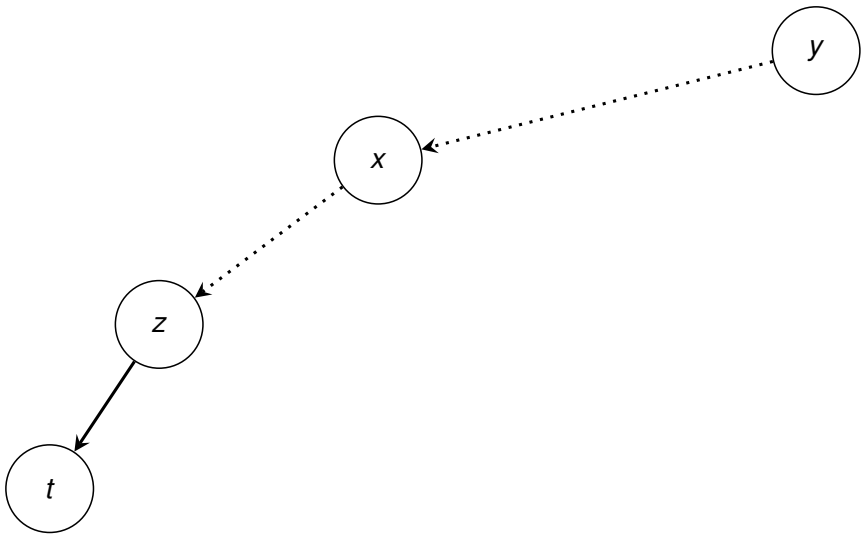
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

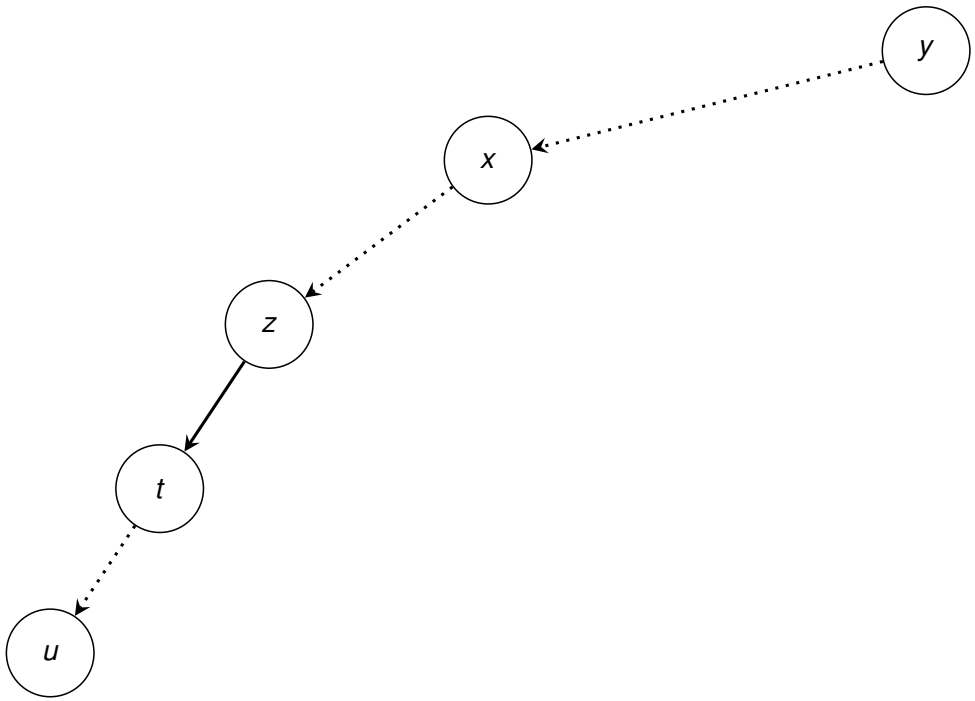
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

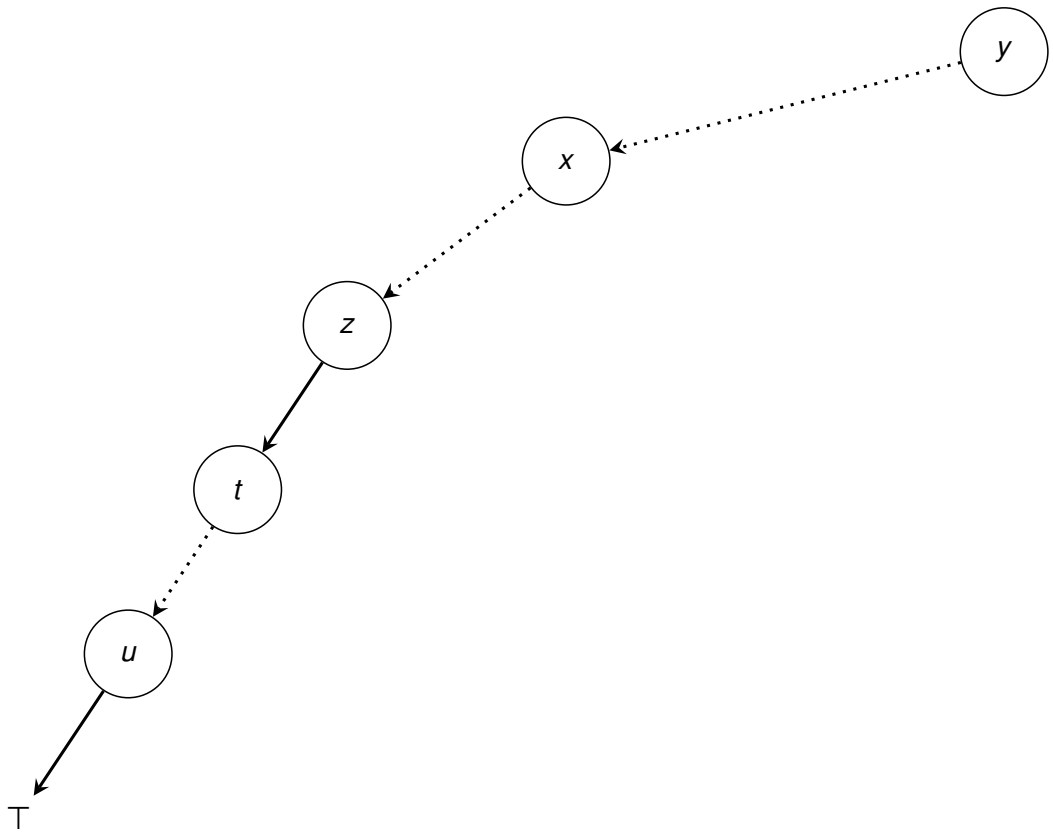
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

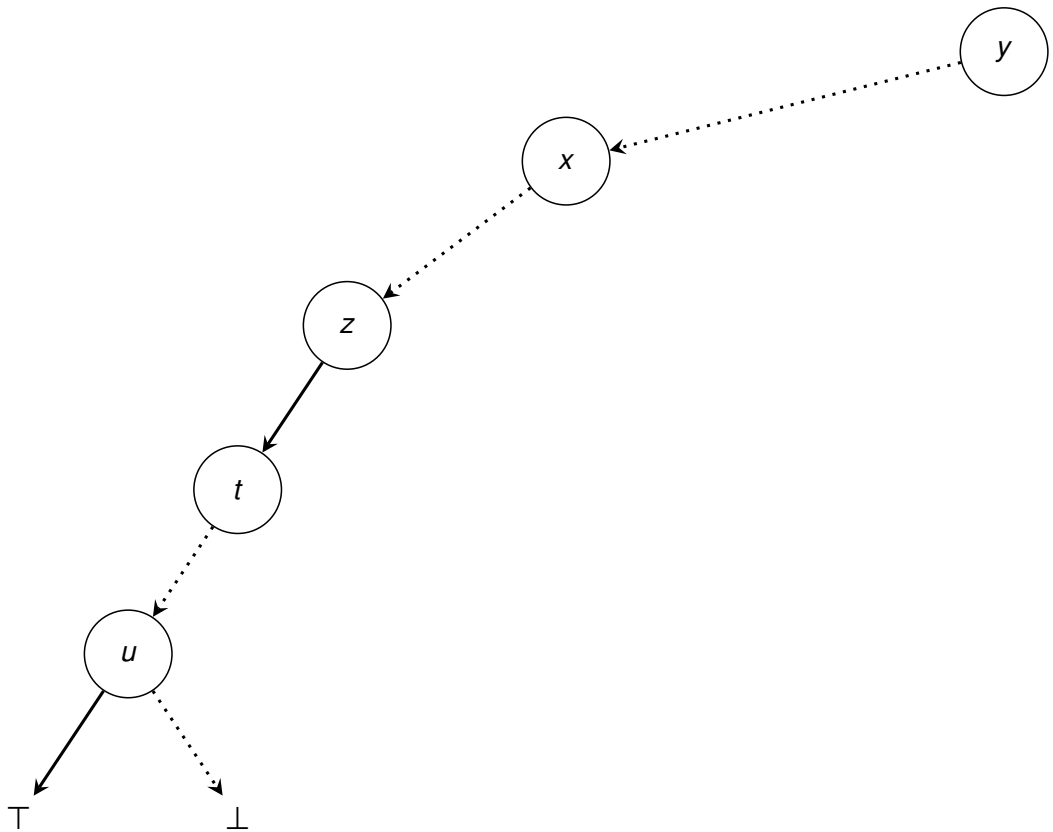
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

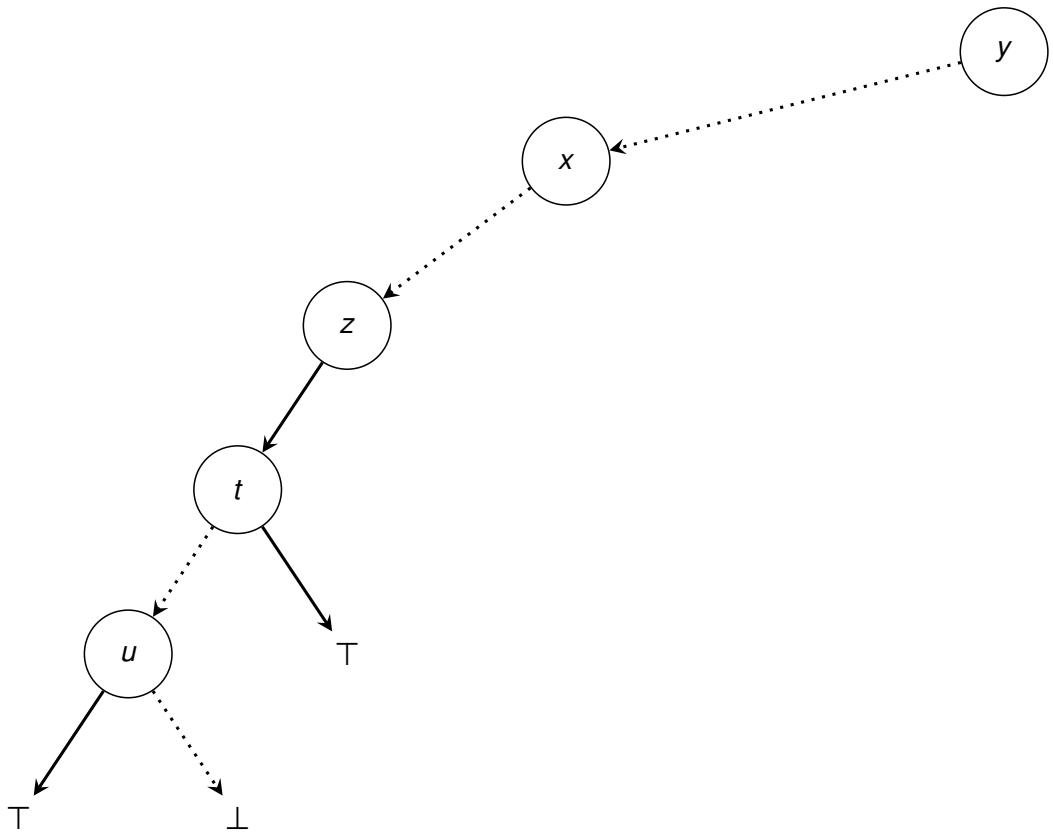
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

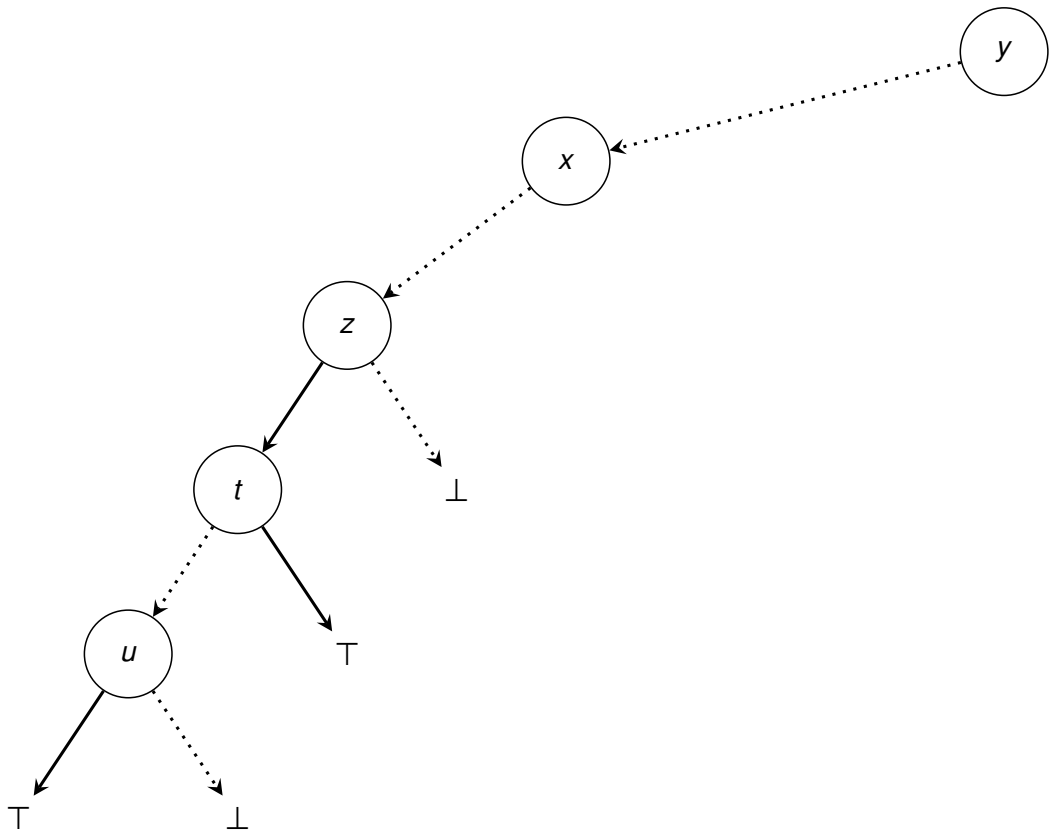
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

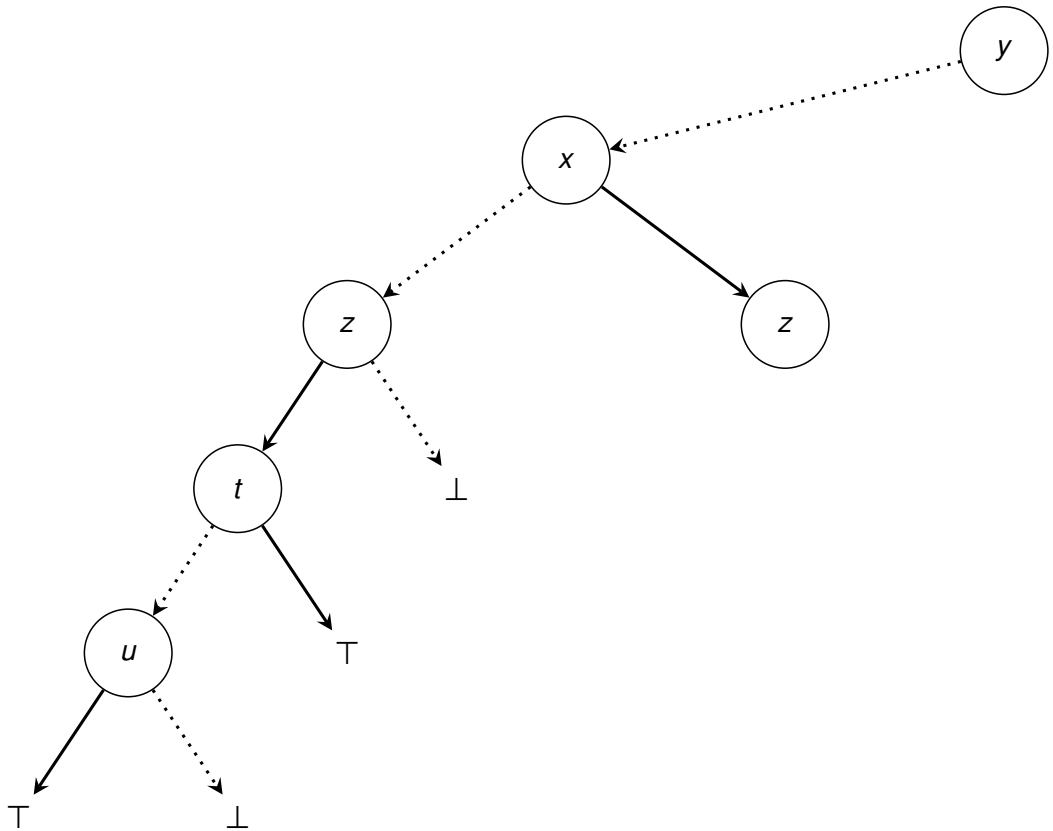
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

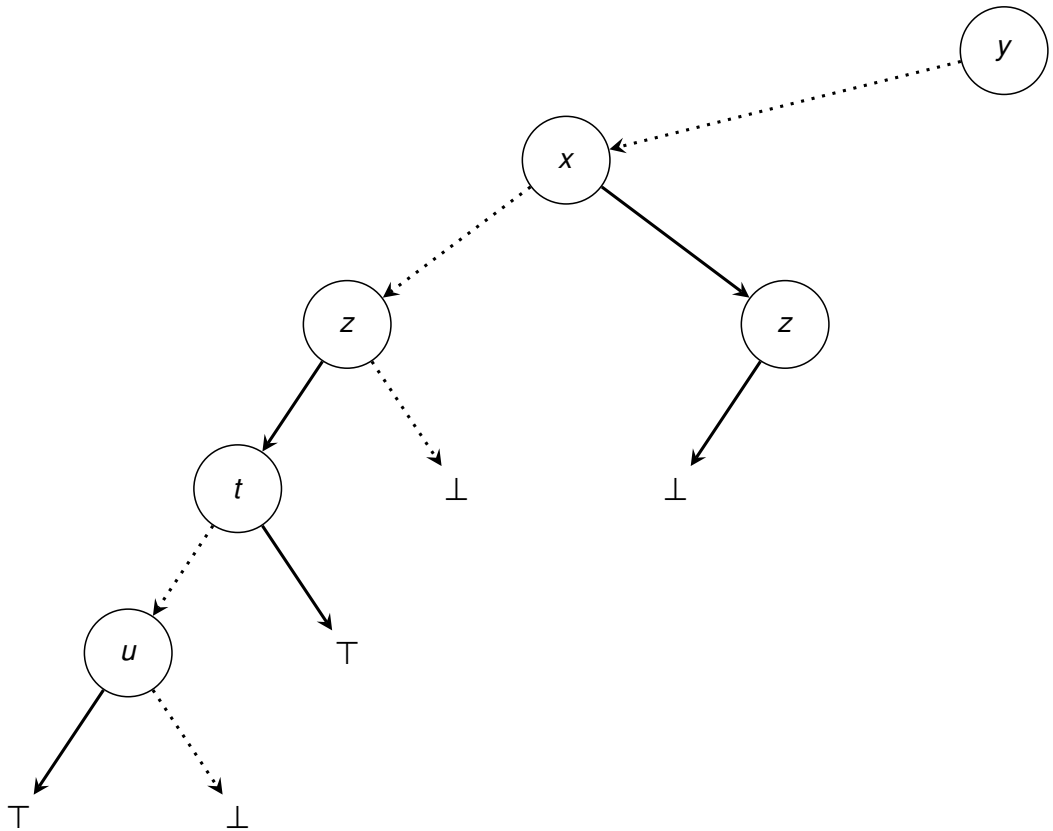
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

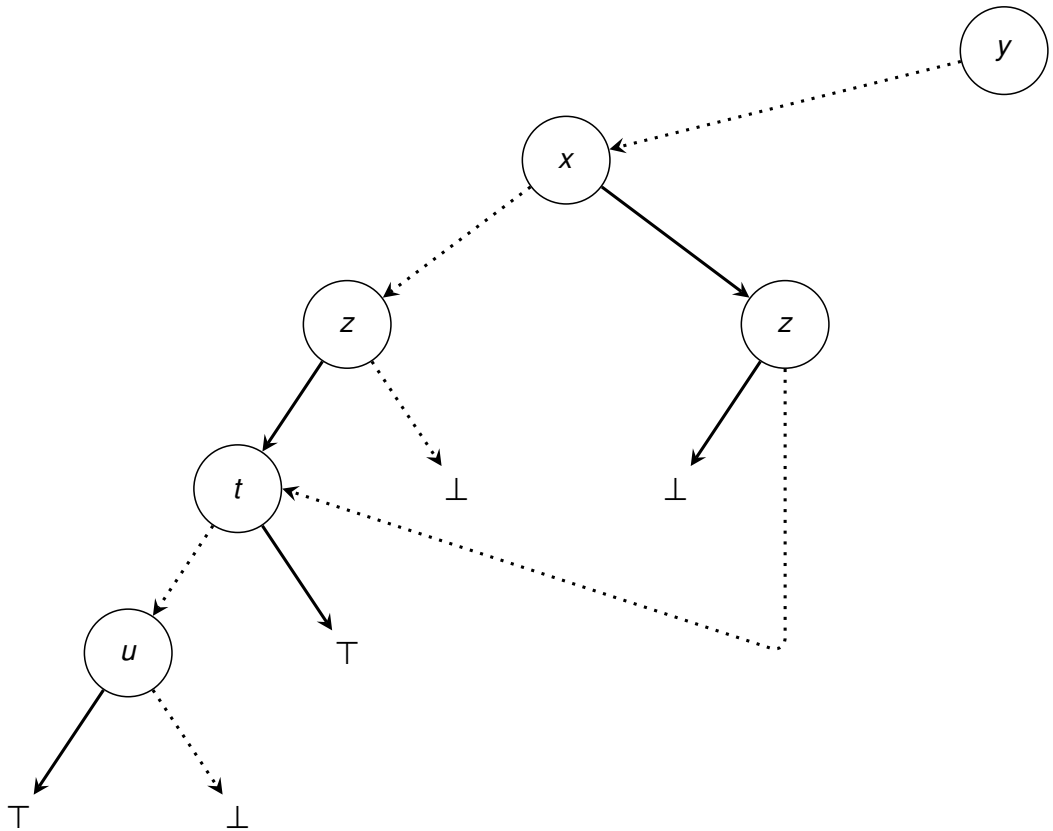
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

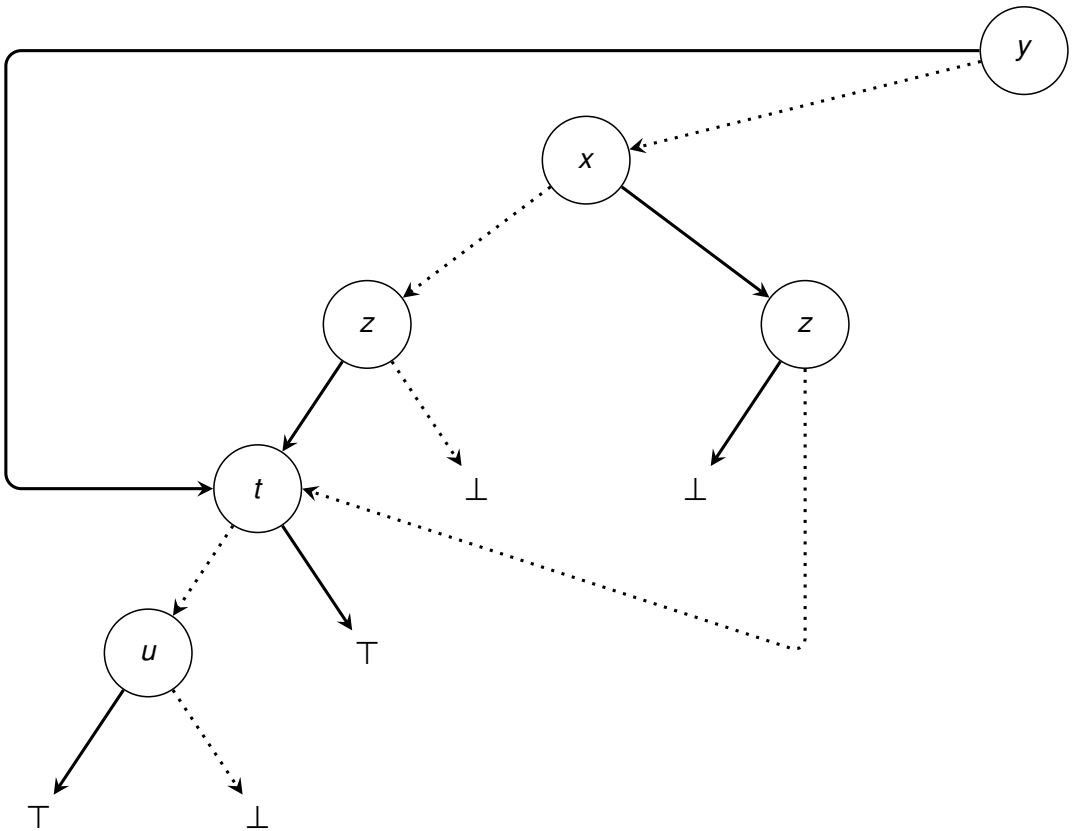
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Free Binary Decision Diagram (FBDD)

Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



The KC Map for FBDD: Queries

$\mathscr{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
Circ	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
ODNF	√	√	√	√	√	√	√	√
MODS	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√
FBDD	√	√	√	√	○	○	√	√

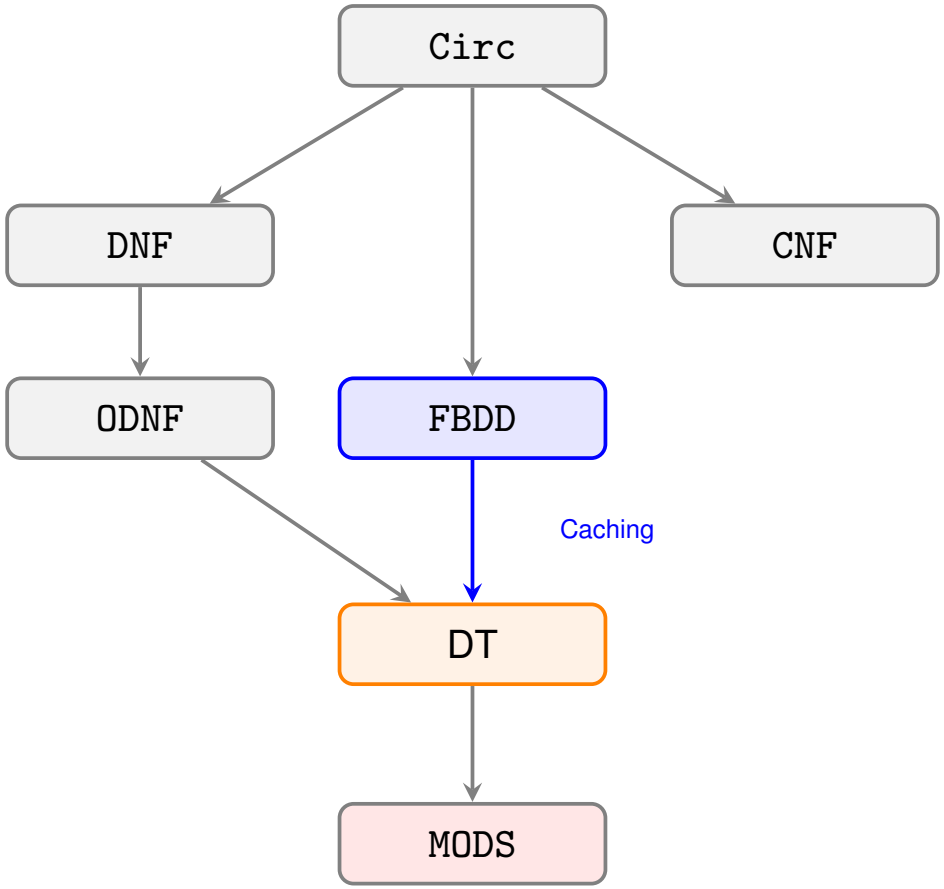
- **The Trade-off for Compression:** By allowing the tree to merge into a DAG (FBDD), we successfully shrink the size while preserving poly-time Model Counting (**CT**) and Consistency (**CO**).
- **What we lose:** Because the paths can now merge and variables can be tested in different orders on different branches, Equivalence (**EQ**) and Sentential Entailment (**SE**) become **NP-hard**.

The KC Map for DT: Transformations

$\mathscr{L}$	**CD**	**FO**	**SFO**	**∧C**	**∧BC**	**∨C**	**∨BC**	**¬C**
Circ	√	○	√	√	√	√	√	√
CNF	√	○	√	√	√	○	√	○
DNF	√	√	√	○	√	√	√	○
ODNF	√	○	○	○	√	○	○	○
MODS	√	√	√	○	√	○	√	○
DT	√	○	√	○	√	○	√	√
FBDD	√	○	○	○	○	○	○	√

- **Negation remains easy:** **¬C** is still √ because we can simply swap the $\top$ and $\bot$ leaves of the DAG.
- **Binary operations fail:** Merging two DAGs that have completely different, unrestricted variable orderings causes an exponential blow-up. Consequently, we lose bounded Transformations.

Succinctness: DAGs vs Trees



- $\texttt{FBDD} <_s \text{DT}$: By allowing isomorphic subgraphs to be shared, FBDD is **strictly more succinct** than DT.
- *Recall the Parity function:* It requires $\mathcal{O}(2^n)$ size in DT, but only $\mathcal{O}(n)$ size in FBDD!

Building the FBDD Compiler: Memoization

The Naive Approach

Compiling a full DT first and then searching for identical sub-trees to merge them is impractical (the DT would not fit in memory!).

The Solution: On-the-fly Caching

- We maintain a **Cache** (a hash map) storing pairs of $\langle \texttt{CNF}, \texttt{FBDD} \rangle$.
- At every new decision node in the solver, we check if the current remaining CNF sub-problem is already in the map.
- **If yes:** We link to the existing FBDD node (sharing).
- **If no:** We compile it, and add the result to the cache.

The Implementation Catch

- Testing if two formulas are *logically equivalent* is coNP-complete. We cannot do this at every step!
- **In practice:** We check for **Syntactic Identity** (are the clauses exactly the same, up to ordering?). This is fast via hashing.

Component Caching

Component Caching: The Concept

What is Component Caching?

In model counting, we often encounter the same sub-problems (components) in different branches of the search tree. Caching stores these results to avoid re-solving them (Sang et al., 2004; Lagniez and Marquis, 2021).

- **The Caching Scheme (r_c):** A mapping that converts a sub-formula φ into a **Cache Key** (signature).

$$\varphi \xrightarrow{r_c} \text{Key}$$

- **The Cache:** A hash map storing: Key $\mapsto$ `Circ`.
- **Correctness Condition:** If two sub-formulas generate the same key, they must be logically equivalent.

$$r_c(\varphi_1) = r_c(\varphi_2) \implies \varphi_1 \equiv \varphi_2$$

Step 1: Representing a Single Clause

Since a component is a set of clauses, we first need a way to represent individual clauses in the cache key.

1. Literal Representation

The clause is explicitly stored as a sorted list of literals.

- **Format:** Integers (Variable IDs).
- **Sign:** Positive ID for x_i, negative for $\neg x_i$.
- **Terminator:** Ends with 0.

Ex: $(x_1 \vee \neg x_3) \rightarrow [1, -3, 0]$

2. Index Representation

The clause is represented by its unique ID from the original CNF.

- **Format:** A single integer (Index) and a list of the variables indices.
- **Pros:** Very compact (low memory).
- **Cons:** Less precise.

Ex: #42:$(x_1 \vee x_3) \rightarrow [42][1, 3]$

Step 2: Representing the Formula (The Scheme)

Which clauses should be included in the Cache Key?

- **Scheme A: "Standard" All Clauses (cachet) (Sang et al., 2004)**
  - Include **all** clauses belonging to the current component.
  - *Safe but verbose.*
  - **Requirement:**
    - *Literal Rep:* Variables are implicit (contained in the literals).
    - *Index Rep:* Need to explicitly store the variable set.

- **Scheme B: "Non-Binary" (sharpSAT) (Muise et al., 2010)**
  - Include only clauses of **size** > 2.
  - *Rationale:* Binary clauses are numerous but imply little structure. Excluding them saves memory and increases cache hits (abstraction).
  - **Requirement:** Need to explicitly store which variables are involved.

- **Scheme C: "Not Touched" (D4) (Lagniez and Marquis, 2017a)**
  - Include only clauses that have been **modified** (shortened) by the current partial assignment.
  - *Rationale:* If a clause is "untouched" (original state), it is implicitly defined by the set of variables involved in the component.
  - **Requirement:** Need to explicitly store which variables are involved.

Running Example: Setup

Let us consider a specific CNF Σ and a partial assignment.

Original CNF Σ

Indices:

1. $(x_1 \vee x_2 \vee x_4)$
2. $(x_2 \vee x_3 \vee x_4)$
3. $(x_3 \vee x_4 \vee x_5)$
4. $(x_1 \vee \neg x_6)$

Current State (φ)

Decision: Set $x_2 = \top$. **BCP (Unit Propagation):**

- Clause 1 $(x_1 \vee \top \vee x_4)$ is Satisfied (Removed).
- Clause 2 $(\top \vee x_3 \vee x_4)$ is Satisfied (Removed).

Remaining Component φ:

$$(x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$$

Active Variables: $\{1, 3, 4, 5, 6\}$

Example 1: “All Clauses” Scheme

Stores the variable set plus **every** active clause.

$$\text{Remaining Clauses: } \underbrace{(x_3 \vee x_4 \vee x_5)}_{\text{Clause 3}} \wedge \underbrace{(x_1 \vee \neg x_6)}_{\text{Clause 4}}$$

Representations

- **Literal Representation:** Explicitly list literals for both clauses.

$$r(\varphi) = ([\underbrace{3,4,5,0}_{C_3}, \underbrace{1,-6,0}_{C_4}])$$

- **Index Representation:** List the IDs of the active clauses plus variables.

$$r(\varphi) = ([1,3,4,5,6], \quad [\mathbf{3},\mathbf{4}])$$

Example 2: “Non-Binary” Scheme (sharpSAT)

Stores variables plus only **large** clauses (Size > 2).

$$\text{Active Clauses: } \underbrace{(x_3 \vee x_4 \vee x_5)}_{\text{Size 3 (Keep)}} \wedge \underbrace{(x_1 \vee \neg x_6)}_{\text{Size 2 (Drop)}}$$

Representations

- **Literal Representation:** Only the ternary clause is stored explicitly.

$$r(\varphi) = \big([1,3,4,5,6], \quad [\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{0}]\big)$$

- **Index Representation:** Only the index of the ternary clause.

$$r(\varphi) = \big([1,3,4,5,6], \quad [\mathbf{3}]\big)$$

Example 3: “Not Touched” Scheme (D4)

Stores variables plus only **modified** clauses.

- $C_3 : (x_3 \vee x_4 \vee x_5)$. Original: Same. **Status: Untouched.**
- $C_4 : (x_1 \vee \neg x_6)$. Original: Same. **Status: Untouched.**

Since neither clause was modified by the assignment $x_2 = \top$ (they didn't contain x_2), they are implicit.

Representations

- **Literal & Index Representation:** The clause list is empty! The component is identified solely by its variables.

$$r(\varphi) = \big([1,3,4,5,6], \quad [\varnothing]\big)$$

Note: If a clause had been shortened (e.g., $(x_3 \vee x_4)$ from an original $(x_3 \vee x_4 \vee \neg x_2)$), it would be included here.

Cache Management: The Cleaning Problem

The Problem

The cache grows indefinitely.

- Storing millions of components consumes gigabytes of RAM.
- Hash collisions increase, slowing down lookups.

The Strategy

We need a **Cleaning Policy** to periodically remove “low-value” entries.

$$\text{Value}(\textit{Entry}) = f(\text{Age}, \text{Activity}, \text{Size})$$

Cache Cleaning Strategies (I)

1. Cachet (Age-Based)

Old entries are useless.

- **Metric:** Age (time since creation).
- **Trigger:** When cache is full (e.g., 2^{21} entries).
- **Action:** Delete the oldest entries (keep 25% newest).

2. SharpSAT (Activity-Based)

Used entries are useful.

- **Metric:** Score (Activity).
  - Init: Age.
  - Hit: Reset score (boost).
  - Periodic: Decay all scores.
- **Trigger:** Cache exceeds max size fraction.
- **Action:** Delete lowest scores.

Cache Cleaning Strategies (II)

3. D4 (Size-Aware)

Small components (few variables) are hit more often than large ones. A “one-size-fits-all” score is inefficient.

1. **Track Performance by Size:** Compute the hit ratio for each component size s:

$$r(s) = \frac{\text{Hits on components of size } s}{\text{Total components of size } s}$$

2. **Identify “Dead” Entries:** An entry e is flushed if:
   - Its size class is performing poorly ($r(|e|) < \text{Threshold}$).
   - AND it hasn’t been used recently ($\text{flag}(e) = \text{false}$).
   - AND its activity score is zero.
3. **Aging:** For kept entries, divide score by 2. If score drops to 0, uncheck flag.

The Limit of FBDD: Independent Components

- How efficiently can we compile the following formula into an FBDD?

$$\Sigma = \bigwedge_{i=1}^{n} \left(x_1^i \vee x_2^i \vee \ldots \vee x_n^i \right)$$

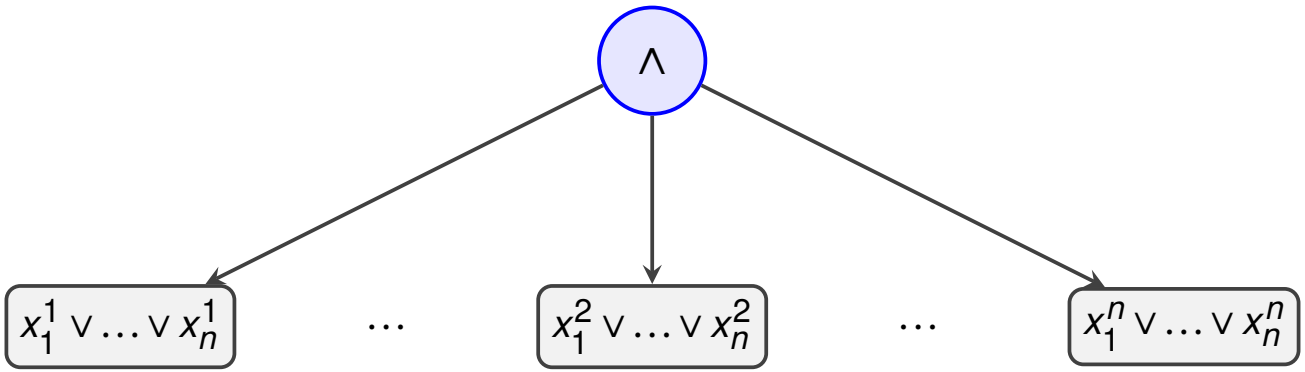
The Problem

- Each clause is completely independent (they share no variables).
- An FBDD **forces** us to make sequential decision branches, weaving these independent problems together into a single path.
- This inability to cleanly separate independent sub-problems can cause an exponential blow-up in the DAG size!

Outline

The Solution: Decomposition

- We observe that the clauses share no variables. They are disjoint.
- Can we compile the clauses separately and aggregate them using an explicit **AND (∧)** node while preserving key queries like counting?
- **Yes!** Because the variables are disjoint, the number of models of the AND node is exactly the **product** of the models of its children.



KC Map for `decision-DNNF`: Queries

$\mathscr{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
`Circ`	○	○	○	○	○	○	○	○
`CNF`	○	√	○	√	○	○	○	○
`DNF`	√	○	√	○	○	○	○	√
`ODNF`	√	√	√	√	√	√	√	√
`MODS`	√	√	√	√	√	√	√	√
`DT`	√	√	√	√	√	√	√	√
`FBDD`	√	√	√	√	○	○	√	√
`decision-DNNF`	√	√	√	√	**?**	○	√	√

- **No Loss in Query Power!** Adding Decomposable AND nodes preserves all the poly-time queries we had in `FBDD`, including Model Counting (**CT**).
- *Note:* Whether Equivalence (**EQ**) is poly-time for `decision-DNNF` remains an open question (?) in the strict Darwiche & Marquis framework.

KC Map for `decision-DNNF`: Transformations

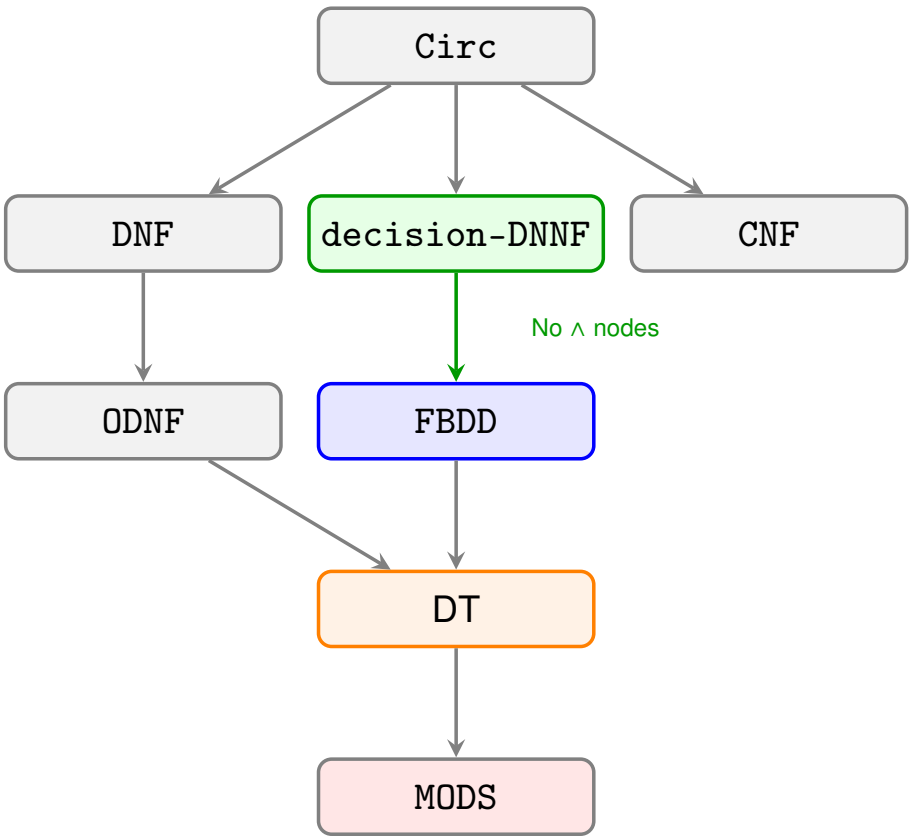
$\mathscr{L}$	**CD**	**FO**	**SFO**	**∧C**	**∧BC**	**∨C**	**∨BC**	**¬C**
`Circ`	√	○	√	√	√	√	√	√
`CNF`	√	○	√	√	√	○	√	○
`DNF`	√	√	√	○	√	√	√	○
`ODNF`	√	○	○	○	√	○	○	○
`MODS`	√	√	√	○	√	○	√	○
DT	√	○	√	○	√	○	√	√
`FBDD`	√	○	○	○	○	○	○	√
`decision-DNNF`	√	○	○	○	○	○	○	**?**

The Loss of Negation (¬**C**)

For DT and `FBDD`, negation was simply swapping $\top$ and $\bot$ leaves. But `decision-DNNF` introduces explicit AND ($\wedge$) nodes!

- By De Morgan's laws, negating an AND node creates an OR node ($\vee$).
- We have **no guarantee** that the resulting OR node is deterministic (orthogonal). If determinism breaks, we can no longer count models!

Succinctness: The Power of AND Nodes



- `decision-DNNF` $<_s$ `FBDD`: By allowing independent sub-formulas to be explicitly split via AND nodes, we prevent them from being needlessly interleaved.
- **The Result:** `decision-DNNF` is **strictly more succinct** than `FBDD`.

Decision-DNNF (decision-DNNF)

Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$

Decision-DNNF (decision-DNNF)

Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$

Decision-DNNF (decision-DNNF)

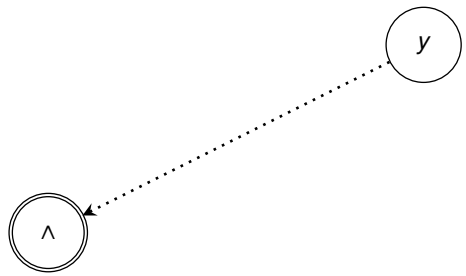
Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$

Decision-DNNF (decision-DNNF)

Formula

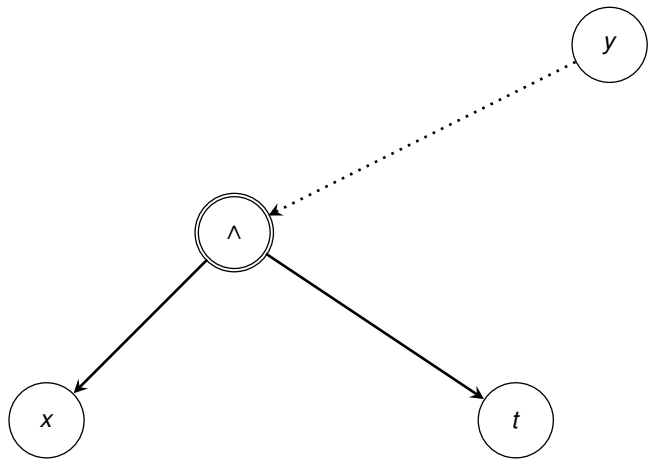
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision-DNNF (decision-DNNF)

Formula

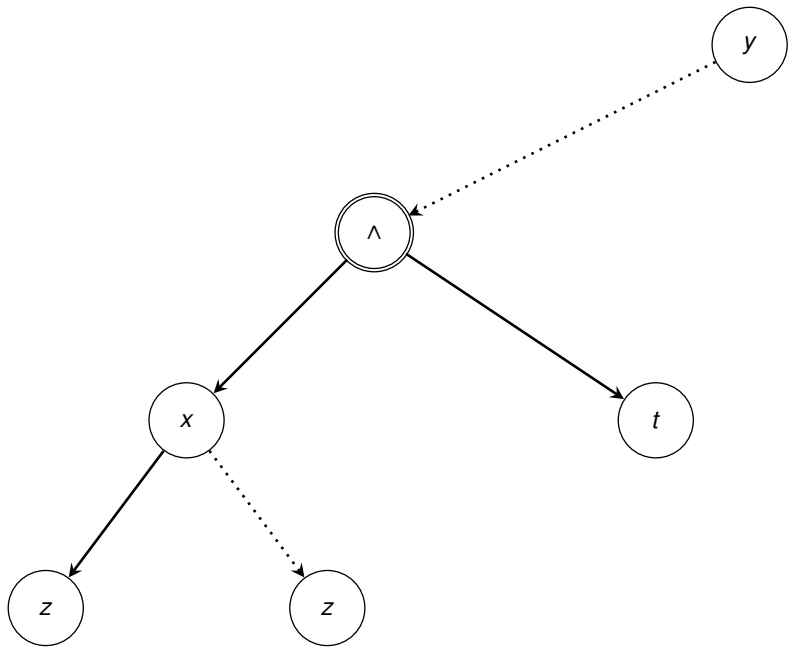
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision-DNNF (decision-DNNF)

Formula

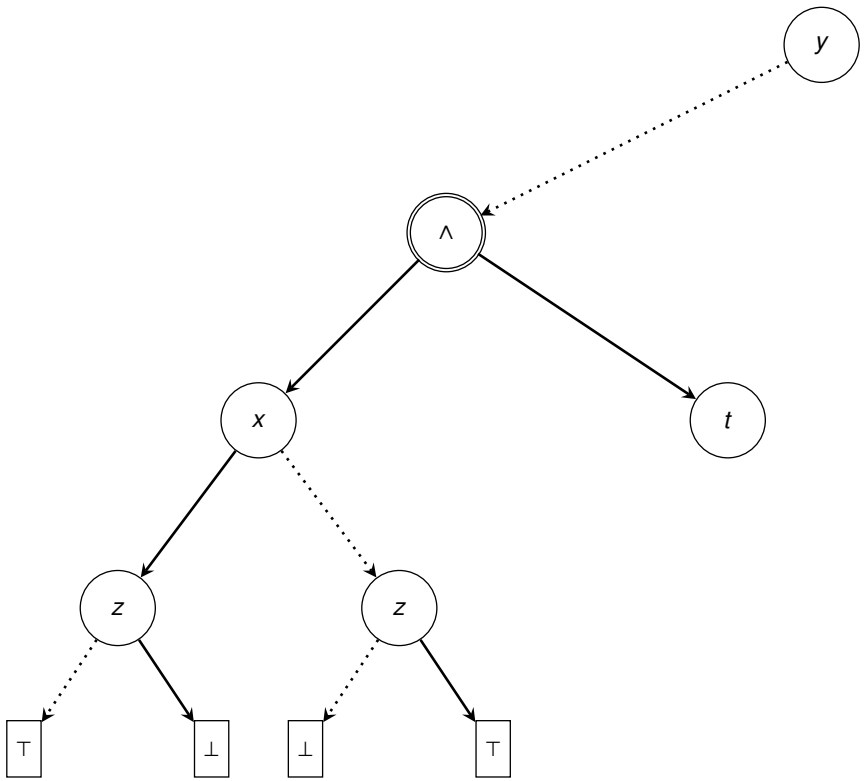
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision-DNNF (decision-DNNF)

Formula

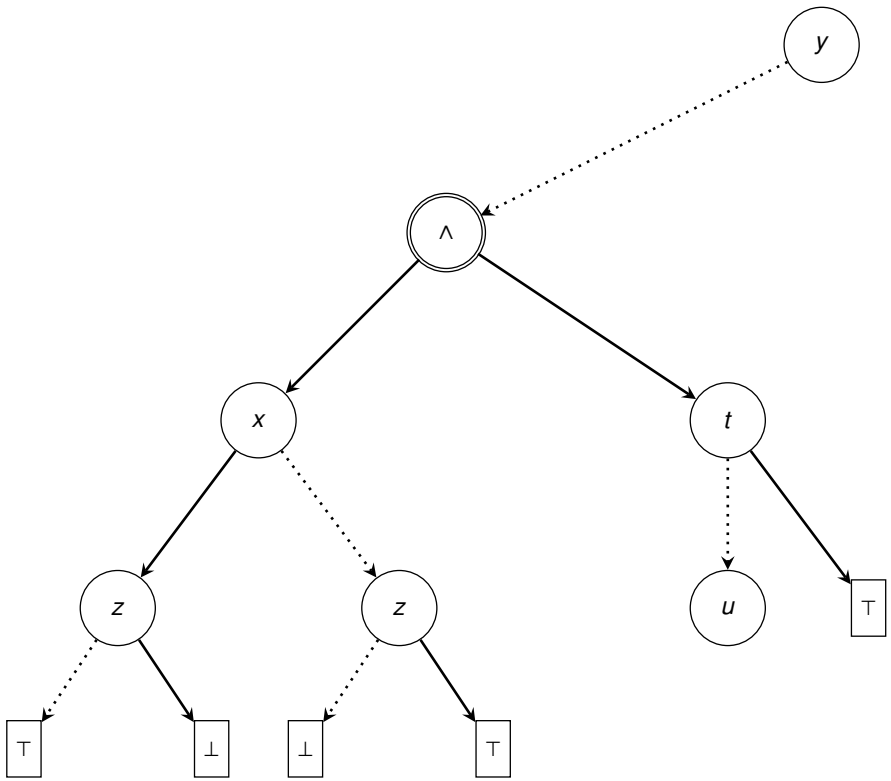
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision-DNNF (decision-DNNF)

Formula

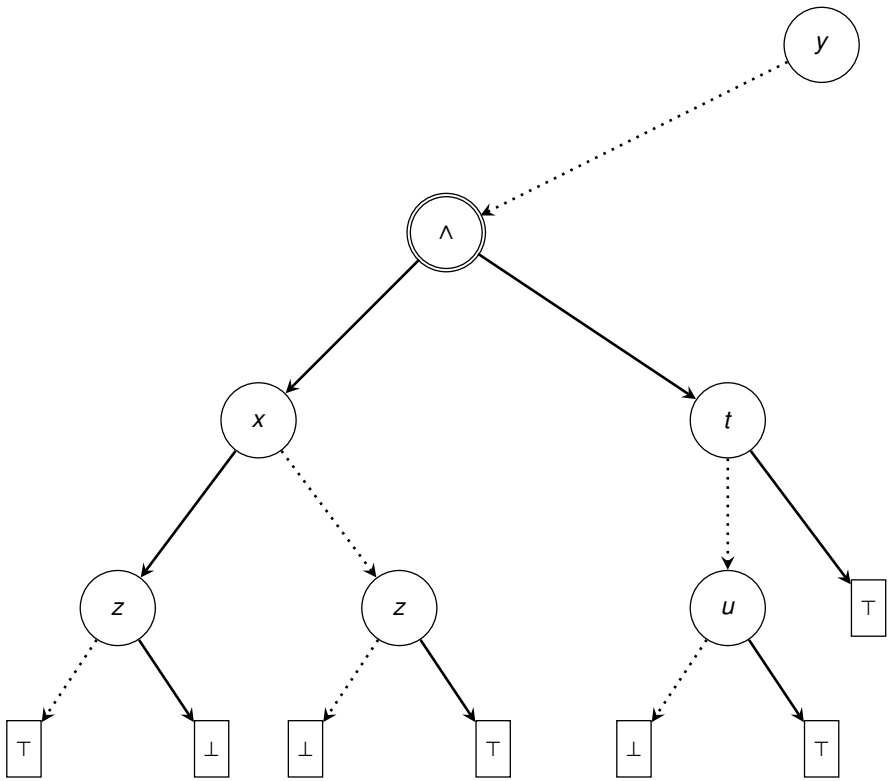
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision-DNNF (decision-DNNF)

Formula

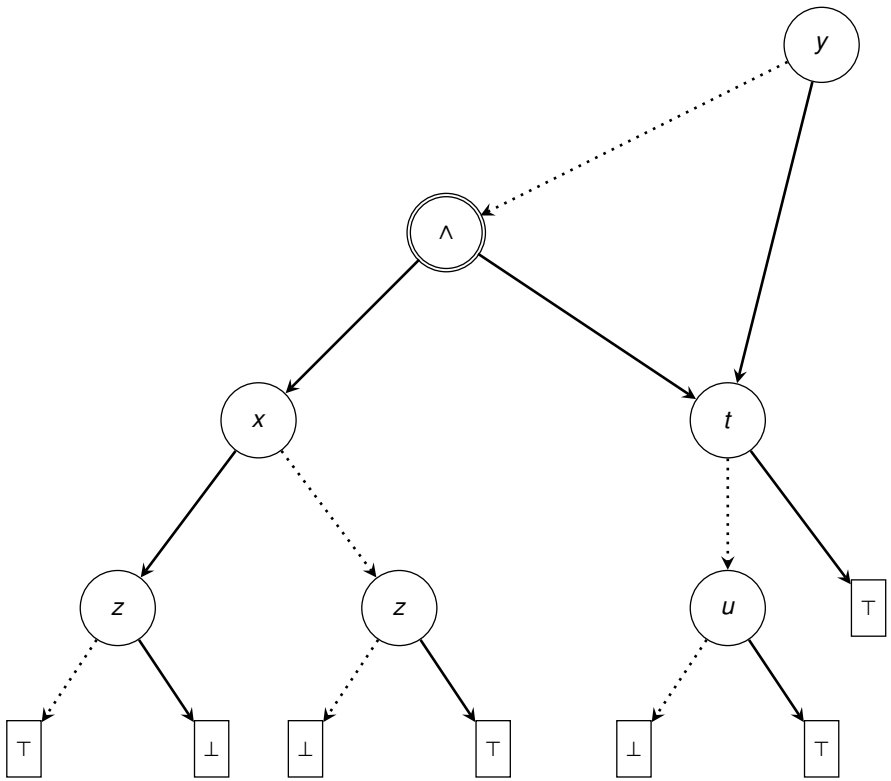
$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Decision-DNNF (decision-DNNF)

Formula

$\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u) \wedge (\neg y \vee t \vee u)$



Branching Heuristics

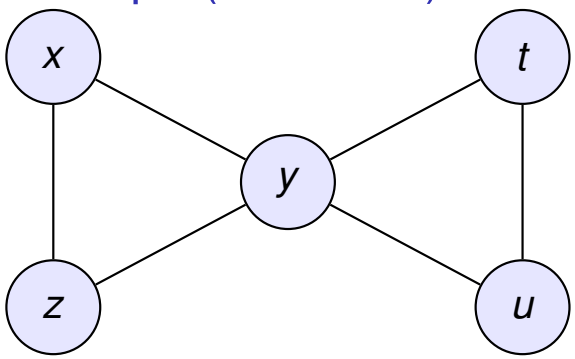
Graph Representations of CNF

We represent the CNF Σ as a graph to find structure.

Formula with 4 Clauses

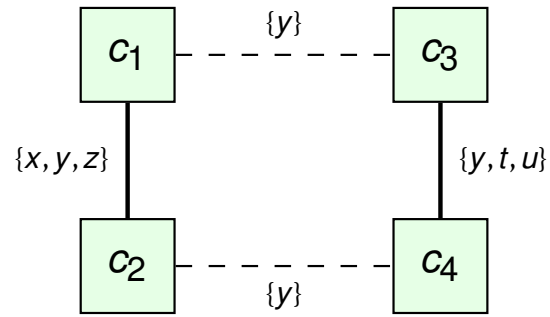
$$\Sigma = \underbrace{(\neg x \vee y \vee \neg z)}_{c_1} \wedge \underbrace{(x \vee y \vee z)}_{c_2} \wedge \underbrace{(y \vee t \vee u)}_{c_3} \wedge \underbrace{(\neg y \vee t \vee u)}_{c_4}$$

1. Primal Graph (Variables)



Shows variables interacting. Note that c_3 and c_4 cover the same clique $\{y, t, u\}$.

2. Dual Graph (Clauses)

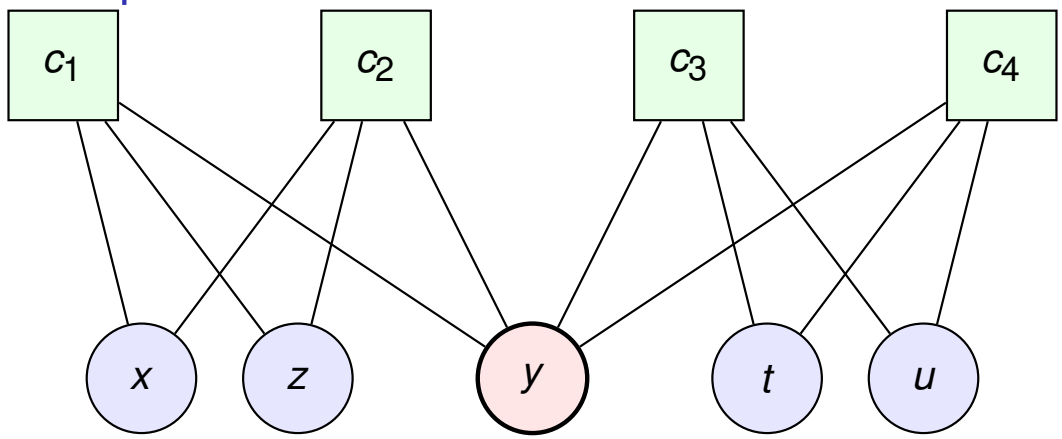


Shows clusters of clauses. Cutting y clearly separates $\{c_1, c_2\}$ from $\{c_3, c_4\}$.

Graph Representations: The Incidence Graph

- The *Incidence Graph* is a **Bipartite Graph** that preserves the exact structure of the formula without adding "clique edges" (unlike the Primal Graph).
- **Nodes:** $V \cup C$ (Variables and Clauses).
- **Edges:** Edge (v, c) exists if variable v appears in clause c.

Example: Incidence Graph of Σ



- **Visual Insight:** Notice how variable y acts as the sole *cut vertex* (bridge).
- If y is assigned (removed), the graph splits into two disjoint connected components: $\{c_1, c_2\}$ and $\{c_3, c_4\}$.

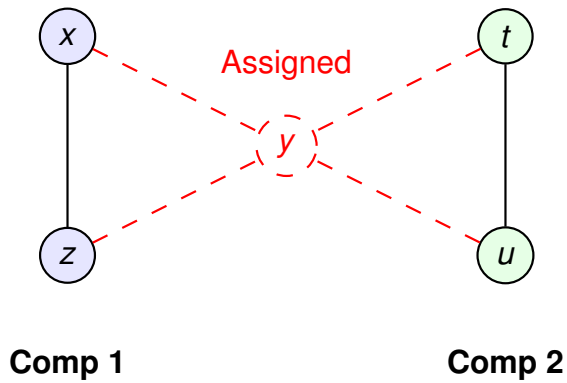
Computing Decomposition: Algorithms

How do we identify independent components?

Dynamic Detection: Component Retrieval

During search (at each decision node), the graph changes.

- **DFS** / **BFS:** Standard traversal.
- **Union-Find:** Maintains disjoint sets.



The Bottleneck: Cost vs. Gain

Computing connected components has polynomial complexity $O(V+E)$, but it must be done at **every decision node**.

- In a search tree with millions of nodes, this overhead is significant.
- Standard graph libraries are too slow; solvers use specialized, lightweight adjacency lists.

Why do we pay this price?

Despite the cost, component detection is the **single most important feature** for tractable counting.

The Complexity Gap:

- **Without Decomposition:** Complexity is $O(2^n)$.
- **With Decomposition:** If the graph splits into two roughly equal parts, complexity drops to $O(2^{n/2}+2^{n/2}) \approx O(2^{n/2})$.

The exponential reduction in search space drastically outweighs the polynomial cost of the BFS.

Classical SAT Heuristics

- **Goal:** Pick the variable most likely to lead to a conflict (to learn clauses fast) or to satisfy the formula.

1. Structure Aware

- **MOMS:** Maximum Occurrence in clauses of Minimum Size (Crawford and Auton, 1996).
- **Jeroslow-Wang:** Weighted occurrence (favors short clauses exponentially) (Jeroslow and Wang, 1990).

2. Activity-Based

VSIDS (Variable State Independent Decaying Sum) (Moskewicz et al., 2001)

- **Mechanism:** Increment a counter for a variable every time it participates in conflict analysis. Decay all counters periodically.
- **Effect:** Focuses search on the “hot” part of the formula (recent conflicts).

Adapting to Counting: VSADS

The Counting Dilemma

- **SAT solvers** want conflicts (to prove UNSAT).
- **Counters** want **decomposition** (to multiply sub-counts).
- Pure VSIDS might stick to one giant component for too long.

VSADS (Sang et al., 2004)

Variable State Aware Decaying Sum
A hybrid score combining activity and structural relevance:

$$\text{Score}(v) = \alpha \cdot \textbf{Activity}(v) + \beta \cdot \textbf{Count}(v)$$

- **Activity:** Standard VSIDS score (conflicts).
- **Count:** How many times v appears in the *current* formula.
- **Dynamic:** If many conflicts occur, α dominates (act like CDCL). If formula simplifies, β dominates (act like MOMS/Decomposition).

Cache-Aware Branching: CSVSADS

- **Observation:** In model counting, **Component Caching** is the primary speedup mechanism.
- We should branch on variables that define *cache keys*, increasing the probability of a **Cache Hit**.

CSVSADS (Sharma et al., 2019)

Cache Score based VSADS

$$\text{Score}(v) = \text{VSIDS}(v) + \textbf{CacheScore}(v)$$

How CacheScore works:

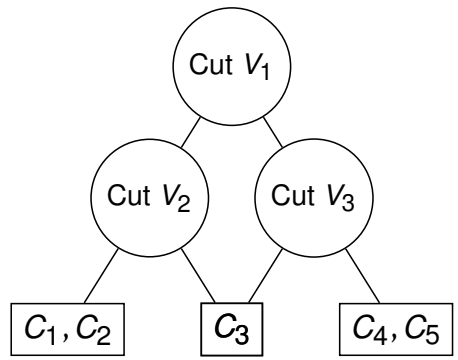
- When storing a component in the cache, increase score of variables appearing in that component.
- **Intuition:** If a variable appears in many cached components, assigning it might lead to a state we have seen before.
- Reduces the number of unique components generated during search.

Structure-Guided Branching: The Static Approach (C2D)

The Idea (Darwiche, 2004)

Instead of letting the solver wander (VSIDS), we force it to follow a **pre-computed roadmap** that guarantees decomposition.

1. **Offline Phase:** Construct a **dtree** (Decomposition Tree) for the CNF.
   - A binary tree where leaves are clauses.
   - Internal nodes represent the union of clauses.
2. **Online Phase (Compilation):**
   - Visit the dtree nodes.
   - Branch on the **cutset** variables (variables shared between the left and right child).



Variables in "Cut" are instantiated first to separate the clauses below.

- **Pros:** Guarantees low treewidth behavior.
- **Cons: Static.** Does not adapt to the simplifications (unit propagations) occurring during the search.

Dynamic Structural Branching (D4)

The idea (Lagniez and Marquis, 2017a)

D4 integrates graph partitioning directly into the search loop:

1. **Compute Cutset:** Call a partitioner (e.g., KaHyPar) to find a separator S for the current component.
2. **Filter:** Restrict branching candidates to S (ignoring others).
3. **Select:** Choose the best variable $v \in S$ using VSADS.
4. **Repeat:** When S is exhausted, re-partition the remaining graph.

Advantage

Computes cuts on the **simplified** graph (post-propagation). Often yields much smaller separators than static methods (C2D).

Drawbacks

- **Overhead:** Partitioning at runtime is expensive.
- **"Tunnel Vision":** Ignores high-activity variables outside S, potentially delaying conflict resolution.

Structure-Guided Branching (sharpSAT-TD)

Concept: Hybrid Guidance (Korhonen and Järvisalo, 2021)

Theory suggests counting is efficient for small treewidth k ($O(2^k n)$), but practical k is often too large. **sharpSAT-TD** uses Tree Decomposition (TD) as a **soft guide** rather than a rigid order.

Modified Heuristic

$$\mathrm{sc}(x) = \underbrace{\mathrm{act}(x) + \mathrm{fr}(x)}_{\text{VSADS}} - \underbrace{C \cdot \min_{t \in T_x} d_T(t)}_{\text{Struct. Penalty}}$$

Key Components:

- **Penalty:** $d_T(t) \in [0,1]$ is the distance of bag t from the root.
- **Effect:** Variables at the "top" of the TD (separators) have lower penalty → higher priority.

- **Tuning:** C scales with width w.
  - **Large C:** Strictly follows decomposition order.
  - **Small C:** Behaves like standard conflict-driven search.

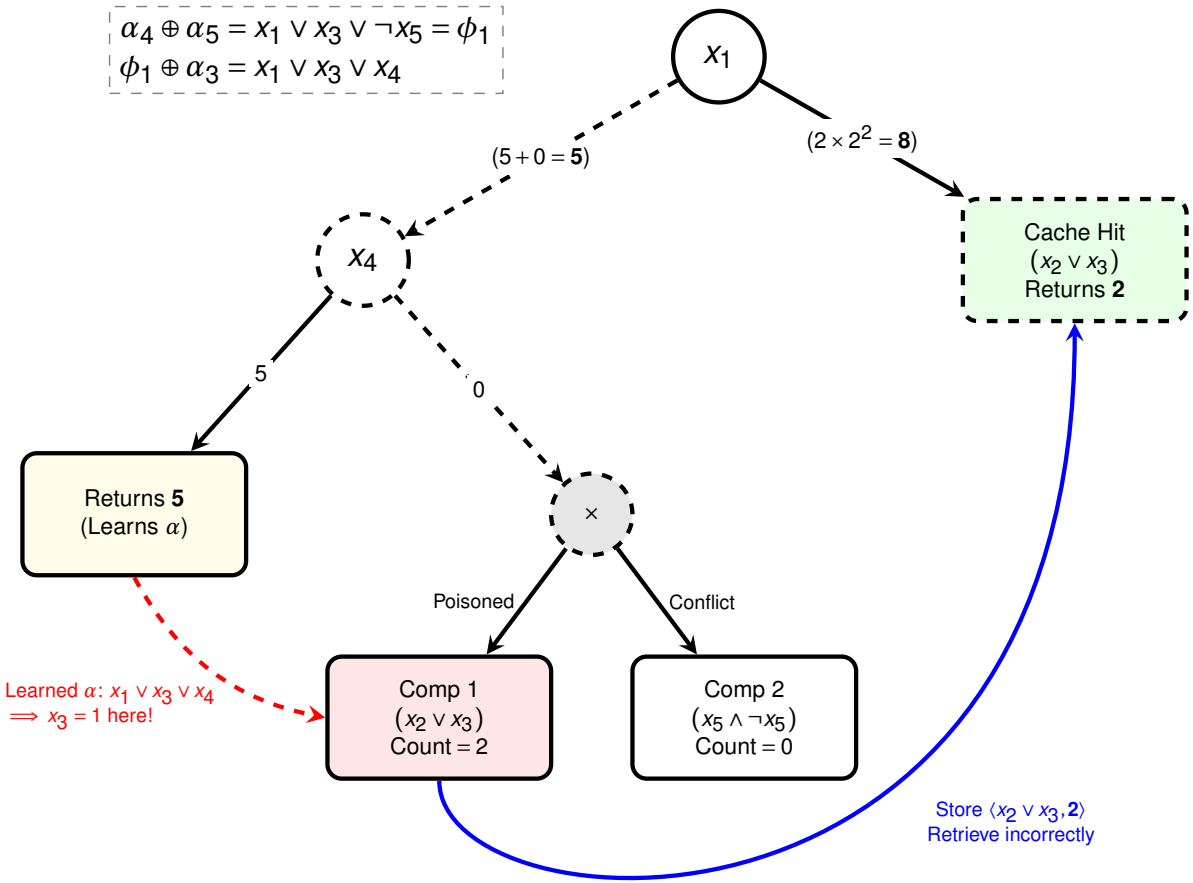
Cache Inconsistency

Cache Inconsistency: Learned Clauses

Input Formula & Scenario

$$\Sigma = \{\underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{\alpha_1}, \underbrace{(x_1 \vee x_2 \vee x_3)}_{\alpha_2}, \underbrace{(x_1 \vee x_4 \vee x_5)}_{\alpha_3}, \underbrace{(x_1 \vee x_4 \vee \neg x_5)}_{\alpha_4}, \underbrace{(x_1 \vee x_3 \vee \neg x_4 \vee \neg x_5)}_{\alpha_5}\}$$

Total: $5 + 8 = 13 \neq 17$ **(Error)**



Cache Inconsistency: Handling Conflicts

The Problem: Context Dependence

- CDCL learned clauses (α) are globally valid but may not be implied by the local component.
- If α triggers propagations inside a component, the computed count is **smaller** than the true count (over-constrained).

The Consequence

Cache Poisoning

- In the example, we stored $\langle A, 2 \rangle$.
- This value was only true because the branch eventually led to a conflict (UNSAT).
- If kept, it corrupts future valid branches.

The Solution

Reactive Cache Cleaning

- **Track:** Keep track of entries added during the exploration of the current sub-tree.
- **Trigger:** If the sub-tree returns **0 (UNSAT)**.
- **Action: Remove** all entries added during this exploration.
- **Result:** The "poisoned" entry $\langle A, 2 \rangle$ is deleted because the branch 'x4=0' failed.

Outline

Affine Decision Trees (Koriche et al., 2013)

- **Objective:**
  - Define new propositional languages (ADT, EADT) that offer polynomial-time Model Counting (**CT**).
  - Place them accurately on the Knowledge Compilation map.
- ADT and EADT are built by combining **Affine Formulae (**AFF**)** and **Decision Trees (**DT**)**:
  - AFF satisfies **CT** natively but is logically *incomplete* (it cannot express all Boolean functions).
  - `decision-DNNF` and its subsets (`FBDD`, DT) are *complete* and satisfy **CT**.
- **The Core Idea:** Can we merge the algebraic counting power of AFF with the structural completeness of DT?

Affine Formulae (AFF)

- An **affine clause** is a finite XOR-disjunction of literals (including $\bot$ and $\top$).
- A **simplified affine clause** is a finite XOR-disjunction of pure atoms, plus at most one occurrence of $\top$.
- **Property:** Every affine clause can be rewritten in *linear time* into an equivalent simplified affine clause:

$$x \oplus \neg y \oplus \neg z \mapsto x \oplus y \oplus z$$

- An **affine formula (Φ)** is a finite conjunction of affine clauses:

$$\Phi = (x \oplus \neg y \oplus \neg z) \wedge (x \oplus u \oplus \neg w) \wedge (w \oplus y \oplus \neg z)$$

The **CT** Query

Model counting can be achieved in **polynomial time** when the input is an Affine Formula.

- We do this by turning Φ into reduced row echelon form using the **Gauss Elimination** algorithm (in the field of characteristic 2).

$$\Phi = \begin{pmatrix} x & y & z & & & | & 1 \\ x & & & u & w & | & 0 \\ & y & z & & w & | & 0 \end{pmatrix} \quad \begin{matrix} (E1) \\ (E2) \\ (E3) \end{matrix}$$

The **CT** Query

Model counting can be achieved in **polynomial time** when the input is an Affine Formula.

- We do this by turning Φ into reduced row echelon form using the **Gauss Elimination** algorithm (in the field of characteristic 2).

$$\Phi = \begin{pmatrix} x & y & z & & & | & 1 \\ & y & z & u & w & | & 1 \\ & y & z & & w & | & 0 \end{pmatrix} \quad \begin{matrix} (E1) \\ (E2' = E1 \oplus E2) \\ (E3) \end{matrix}$$

The **CT** Query

Model counting can be achieved in **polynomial time** when the input is an Affine Formula.

- We do this by turning Φ into reduced row echelon form using the **Gauss Elimination** algorithm (in the field of characteristic 2).

$$\Phi = \begin{pmatrix} x & y & z & & & | & 1 \\ & y & z & u & w & | & 1 \\ & & & u & & | & 1 \end{pmatrix} \quad \begin{array}{l} (E1) \\ (E2' = E1 \oplus E2) \\ (E3' = E2' \oplus E3) \end{array}$$

The **CT** Query

Model counting can be achieved in **polynomial time** when the input is an Affine Formula.

- We do this by turning Φ into reduced row echelon form using the **Gauss Elimination** algorithm (in the field of characteristic 2).

$$\Phi = \begin{pmatrix} x & y & z & & & | & 1 \\ & y & z & u & w & | & 1 \\ & & & u & & | & 1 \end{pmatrix} \quad \begin{array}{l} (E1) \\ (E2' = E1 \oplus E2) \\ (E3' = E2' \oplus E3) \end{array}$$

- The resulting system of linear equations has $\#f = 2$ "free" variables (w and z). Therefore, the model count is exactly $\|\Phi\| = 2^{\#f} = 4$.

KC Map for AFF: Queries

$\mathscr{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
Circ	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
ODNF	√	√	√	√	√	√	√	√
MODS	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√
FBDD	√	√	√	√	○	○	√	√
decision-DNNF	√	√	√	√	**?**	○	√	√
AFF	√	√	√	√	√	√	√	√

- **A Perfect Score:** AFF satisfies every single query in polynomial time, matching DT and MODS!

KC Map for AFF: Transformations

$\mathscr{L}$	**CD**	**FO**	**SFO**	**∧C**	**∧BC**	**∨C**	**∨BC**	**¬C**
Circ	√	○	√	√	√	√	√	√
CNF	√	○	√	√	√	○	√	○
DNF	√	√	√	○	√	√	√	○
ODNF	√	○	○	○	√	○	○	○
MODS	√	√	√	○	√	○	√	○
DT	√	○	√	○	√	○	√	√
FBDD	√	○	○	○	○	○	○	√
decision-DNNF	√	○	○	○	○	○	○	**?**
AFF	√	√	√	√	√	**!**	**!**	**!**

- **The Incompleteness Penalty:** AFF cannot express OR (∨) natively. Therefore, operations like Disjunction (∨**C**, ∨**BC**) and Negation (¬**C**) are not guaranteed to produce a valid Affine formula.

Outline

Affine Decision Tree (ADT)

- **The Idea:** Generalize the standard Shannon expansion by replacing the literal (x) with an **Affine Clause (δ)**.

Generalized Shannon Expansion

$$\Phi \equiv \left((x \Leftrightarrow \neg\delta) \wedge \Phi_{|x \leftarrow \neg\delta}\right) \vee \left((x \Leftrightarrow \delta) \wedge \Phi_{|x \leftarrow \delta}\right)$$

- Since $(x \Leftrightarrow \neg\delta) \equiv x \oplus \delta$ and $(x \Leftrightarrow \delta) \equiv \neg x \oplus \delta$, we rewrite it as:

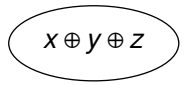
$$\Phi \equiv \left((x \oplus \delta) \wedge \Phi_{|x \leftarrow \delta \oplus \top}\right) \vee \left((\neg x \oplus \delta) \wedge \Phi_{|x \leftarrow \delta}\right)$$

Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$

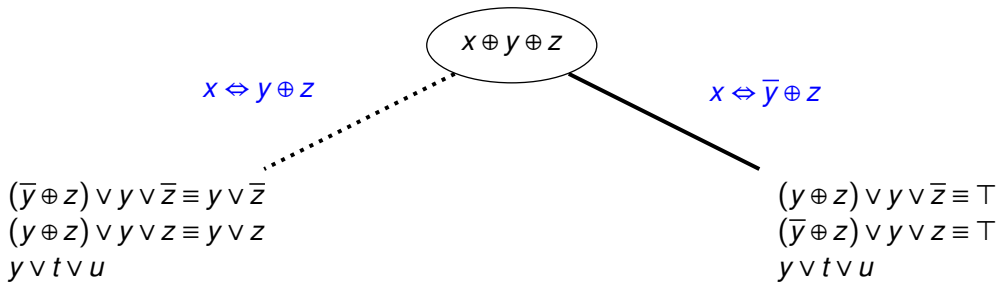
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



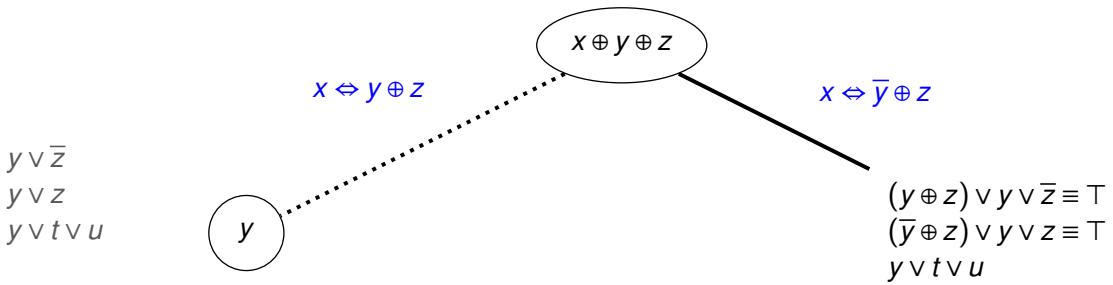
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



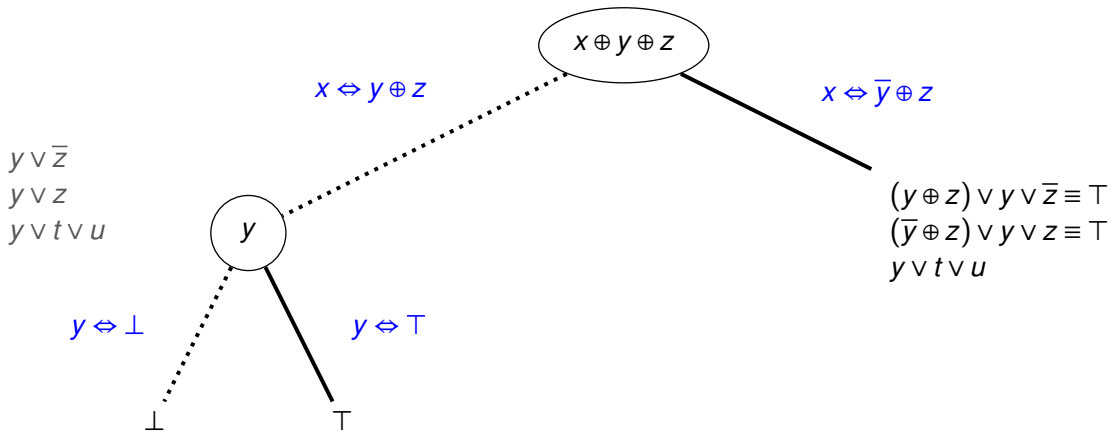
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



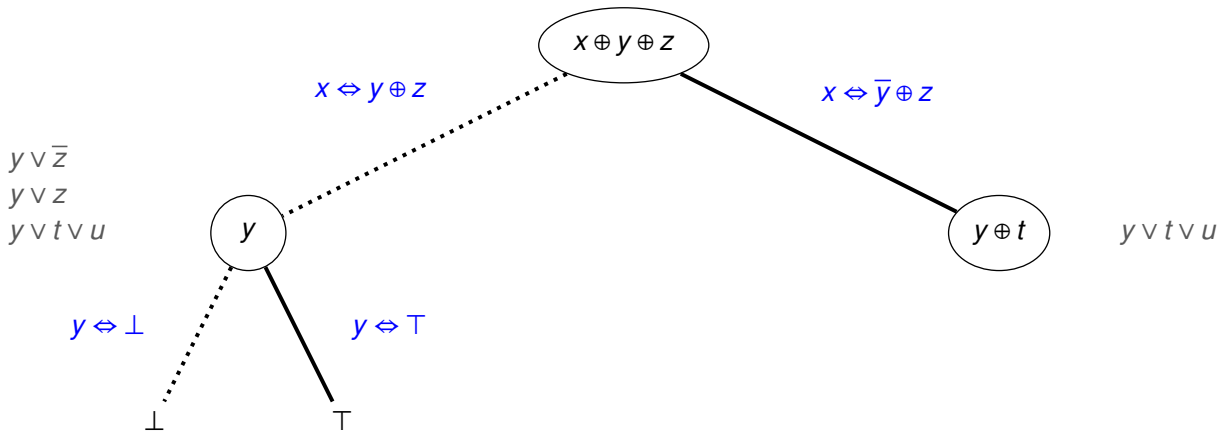
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



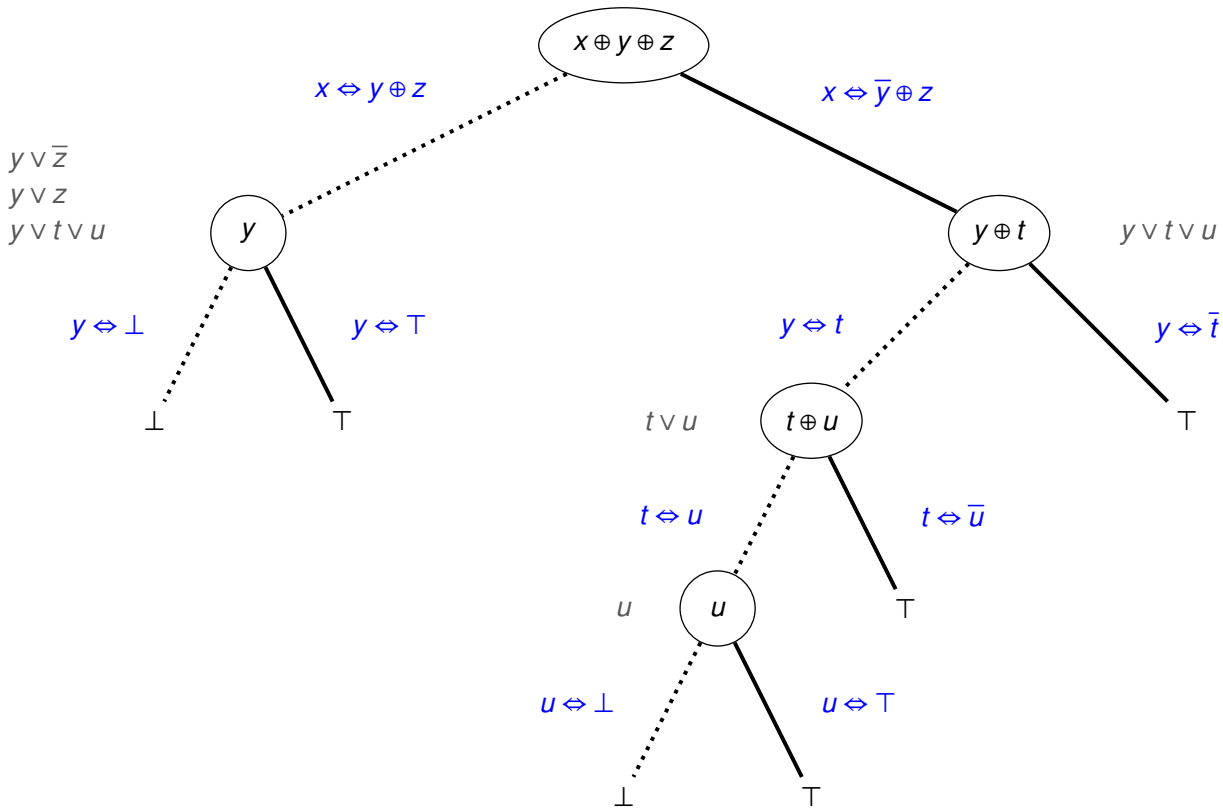
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



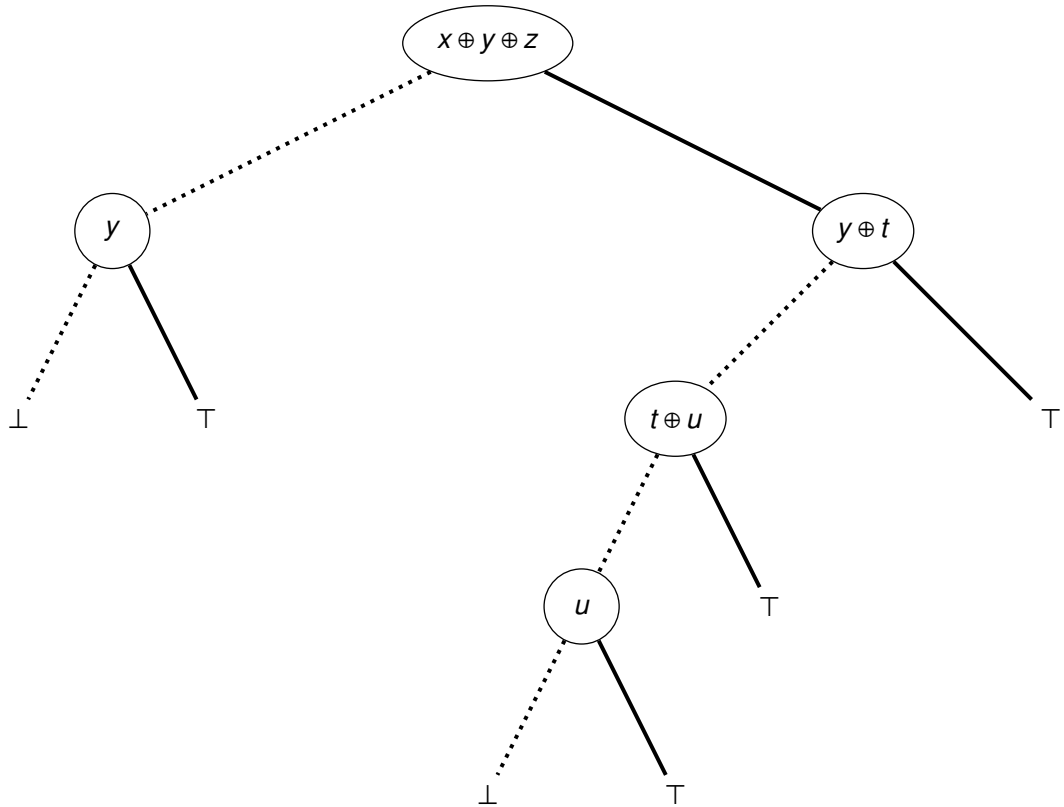
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



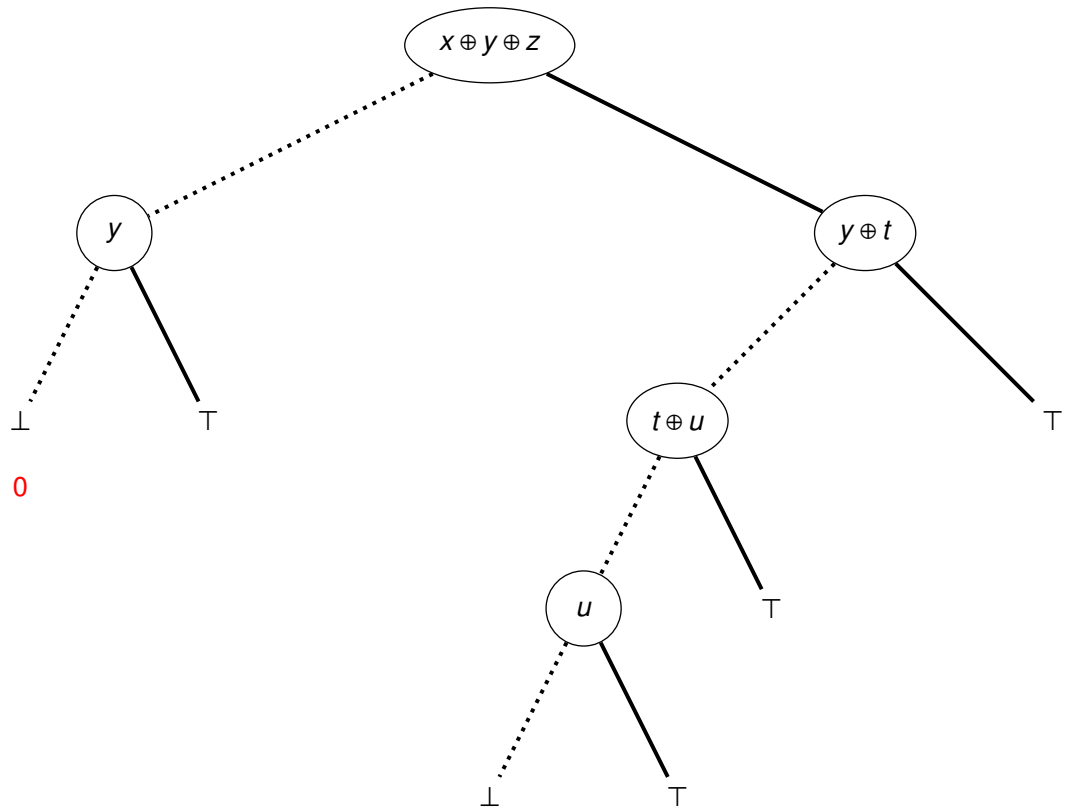
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



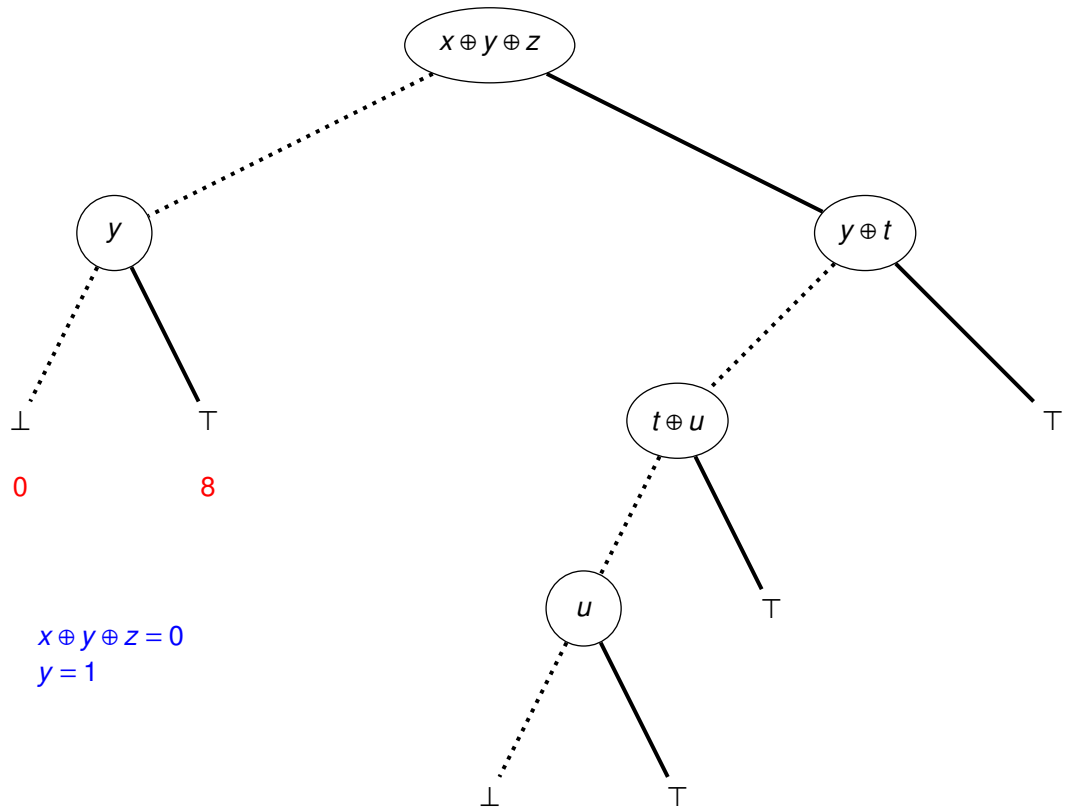
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



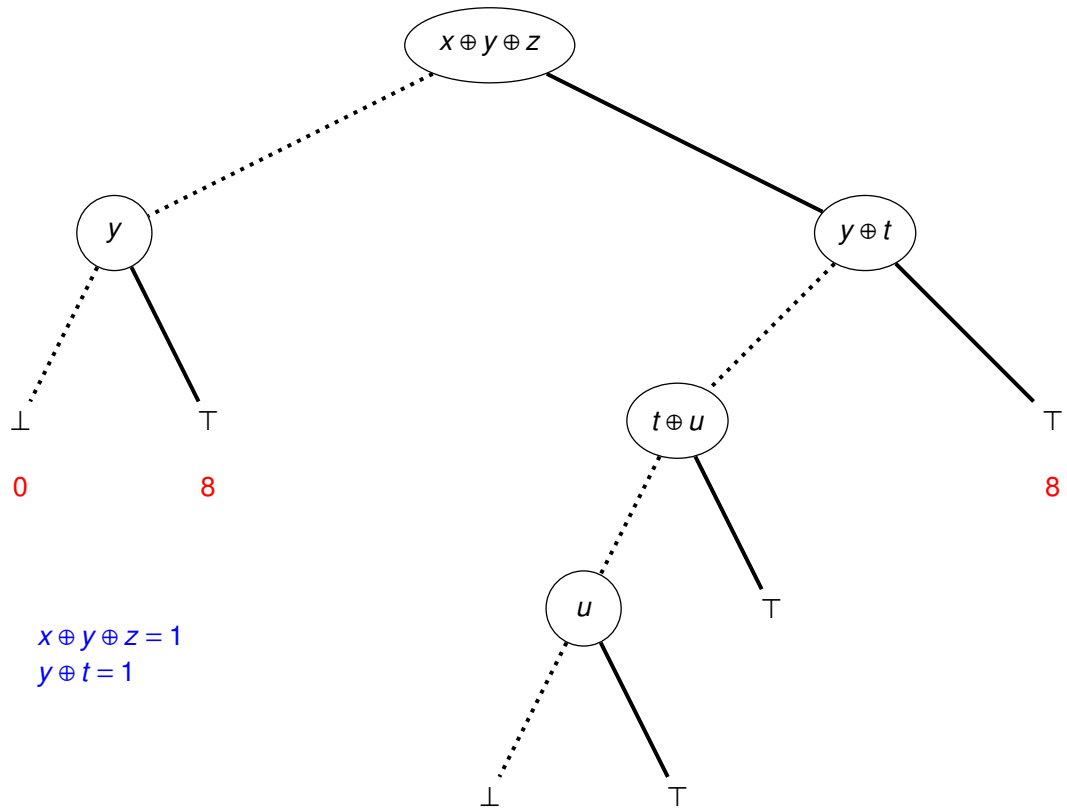
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



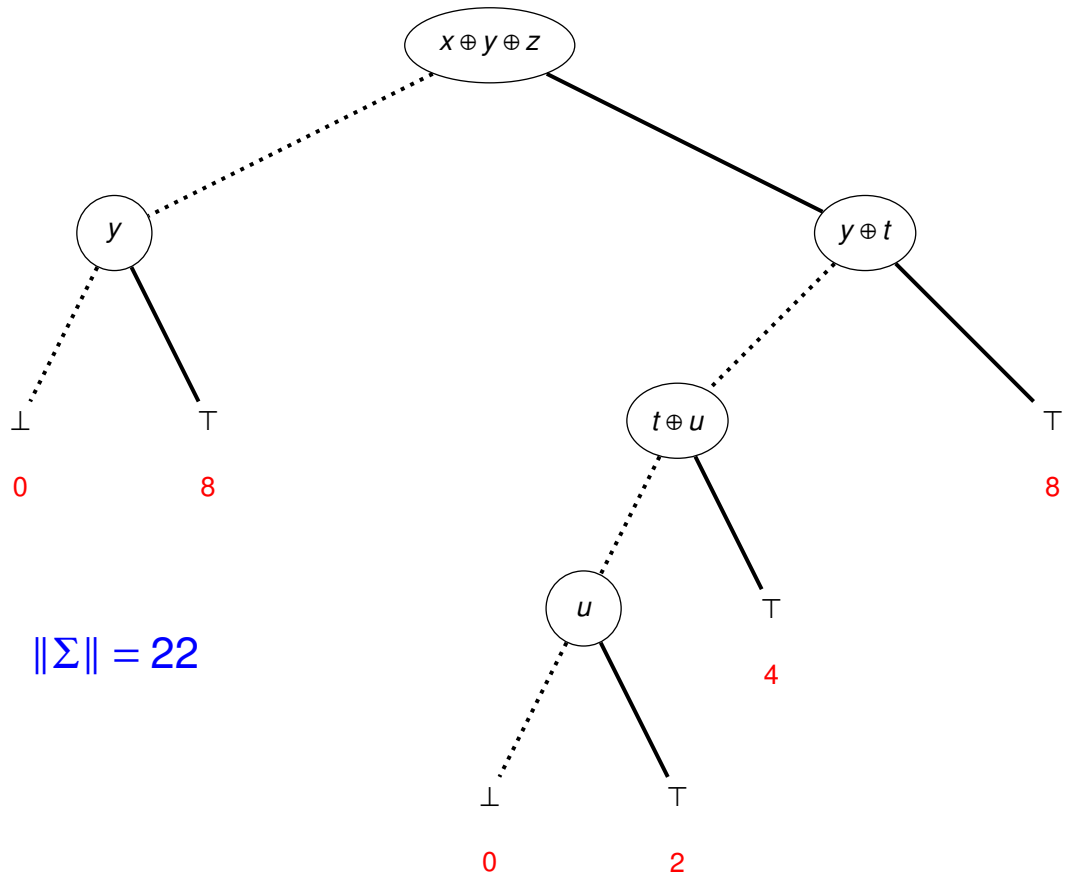
Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



Generation and Model Counting

- Let's compile $\Phi = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



$\|\Sigma\| = 22$

KC Map for ADT: Queries

$\mathscr{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
Circ	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
ODNF	√	√	√	√	√	√	√	√
MODS	√	√	√	√	√	√	√	√
decision − DNNF	√	√	√	√	**?**	○	√	√
FBDD	√	√	√	√	○	○	√	√
DT	√	√	√	√	√	√	√	√
AFF	√	√	√	√	√	√	√	√
ADT	√	√	√	√	√	√	√	√

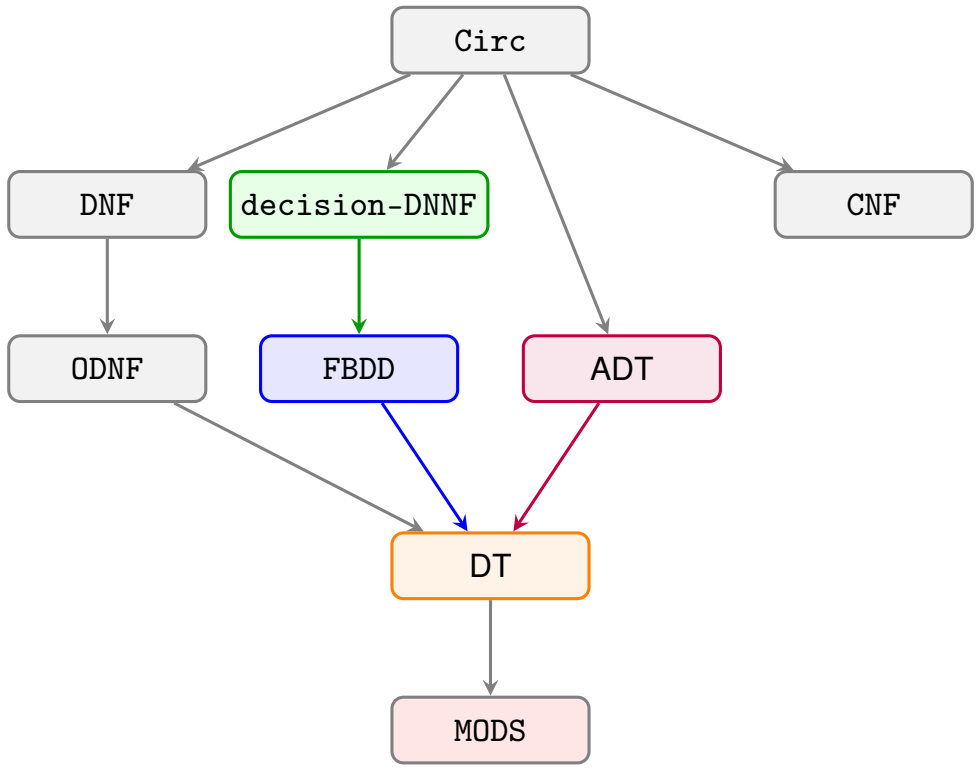
- **A Perfect Score:** Because ADT is fundamentally a tree structure that evaluates affine clauses, it completely inherits the polynomial-time query power of both DT and AFF.

KC Map for ADT: 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	√	∘	√	∘	√	∘	√	√
AFF	√	√	√	√	√	∘	∘	∘
ADT	√	∘	√	∘	√	∘	√	√

- **The Baseline:** ADT mirrors the exact transformation profile of DT.
- **The Advantage:** Even though the profile is identical, ADT is **strictly more succinct** because the decision nodes can encode XOR relationships natively, avoiding exponential depth blow-ups.

Succinctness Hierarchy

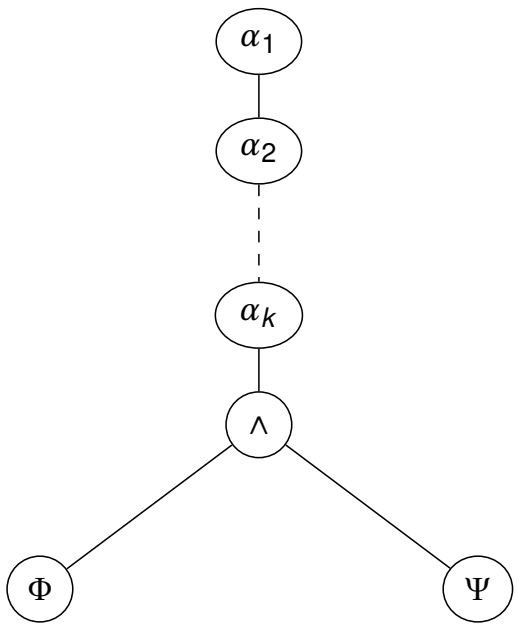


- `decision-DNNF` and ADT are incomparable ($\not\leq_s$) to each other, representing two entirely different paradigms for pushing the limits of Knowledge Compilation!

Outline

Extended Affine Decision Tree (EADT)

- Just like we upgraded `FBDD` to `decision-DNNF`, we can take advantage of **Decomposability** to reduce the size of ADT.
- We introduce explicit $\wedge$-nodes (and $\vee$-nodes) which are **affine decomposable**.



Affine Decomposability

- $\forall i, \alpha_i$ is an affine clause.
- The $\wedge$-node is standard decomposable: $\mathit{Vars}(\Phi) \cap \mathit{Vars}(\Psi) = \varnothing$.
- **Crucially:** For every α_i, it cannot share variables with *both* children simultaneously.

KC Map for EADT: Queries

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

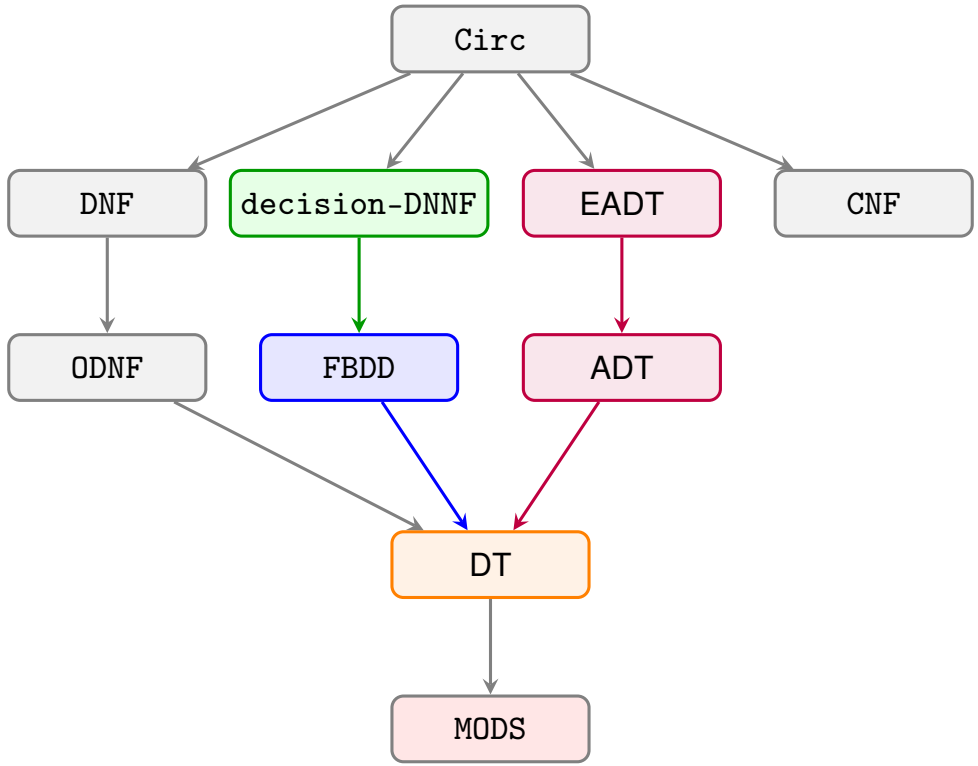
- **The Cost of Decomposability:** Just as moving from FBDD to decision-DNNF cost us Equivalence (**EQ**) and Sentential Entailment (**SE**), adding decomposable $\wedge$-nodes to ADT forces EADT to lose those same properties!

KC Map for EADT: 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	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
AFF	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$	$\circ$	$\circ$
ADT	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
EADT	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\surd$

- While standard decision-DNNF loses the guarantee of Negation (?), EADT **retains it ($\surd$)**!

Succinctness Hierarchy



- `decision-DNNF` and EADT are incomparable ($\not\leq_s$) to each other, EADT is more succinct than ADT!