Knowledge Compilation: power and limits

Florent Capelli

CRIL, Université d'Artois

Reasoning Web Summer School

28 August 2026

Part 0

Knowledge Compilation: an introduction

In a nutshell

Representing Boolean functions $f\colon \{0,1\}^X \to \{0,1\}$ in a *tractable way*.

Given a representation of f, can we:

- **Tractable tasks**:
  - Find τ st $f(\tau) = 1$.
  - Compute $\#f$, $Pr_{\tau \in \{0,1\}^X}(f(\tau) = 1)$.
  - Enumerate the models of f
  - …
- **Tractable transformations**:
  - Build a representation of $\neg f$
  - Build a representation of $f[\tau]$
  - Build a representation of $\exists Z.f$ for some $Z \subseteq X$
  - …

Representation examples

$f\colon \{0,1\}^{\{x,y,z\}} \to \{0,1\}$

f is represented as a CNF formula
$(x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee \neg z \vee \neg y)$

- Succinct
- Not tractable in general (SAT is NP-complete).

f as a list of models:

x	y	z	$f(x,y,z)$
0	1	0	1
1	0	0	1
1	0	1	1
1	1	0	1

- Not succint.
- Very tractable.

Why Knowledge?

Old roots in the study of *expert systems* (MYCIN, DENDRAL…):

- A Knowledge Base: ground facts and rules
- An Inference System: inferring new facts

Simplest abstraction: facts as propositional variables, knowledge as propositional formulas.

Propositional knowledge base can be seen as a Boolean function.

Why compilation?

Natural way of encoding this “knowledge” rarely optimal for computational purposes:

- NP-hard to check whether a list of constraints has a model.
- Each inference is costly.

Preprocess a better representation allowing more tractability.

The bike configuration example

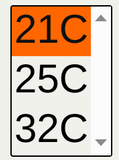
Knowledge base, “natural representation”:

- ExactlyOne(road, gravel, city)
- ExactlyOne(21C, 25C, 32C)
- road ⇒ (21C ∨ 25C)
- city ⇒ ¬21C
- electric ⇒ (city ∨ gravel)
- electric ⇒ 32C

Frame:

Electric:

Tires size:



Other Representations

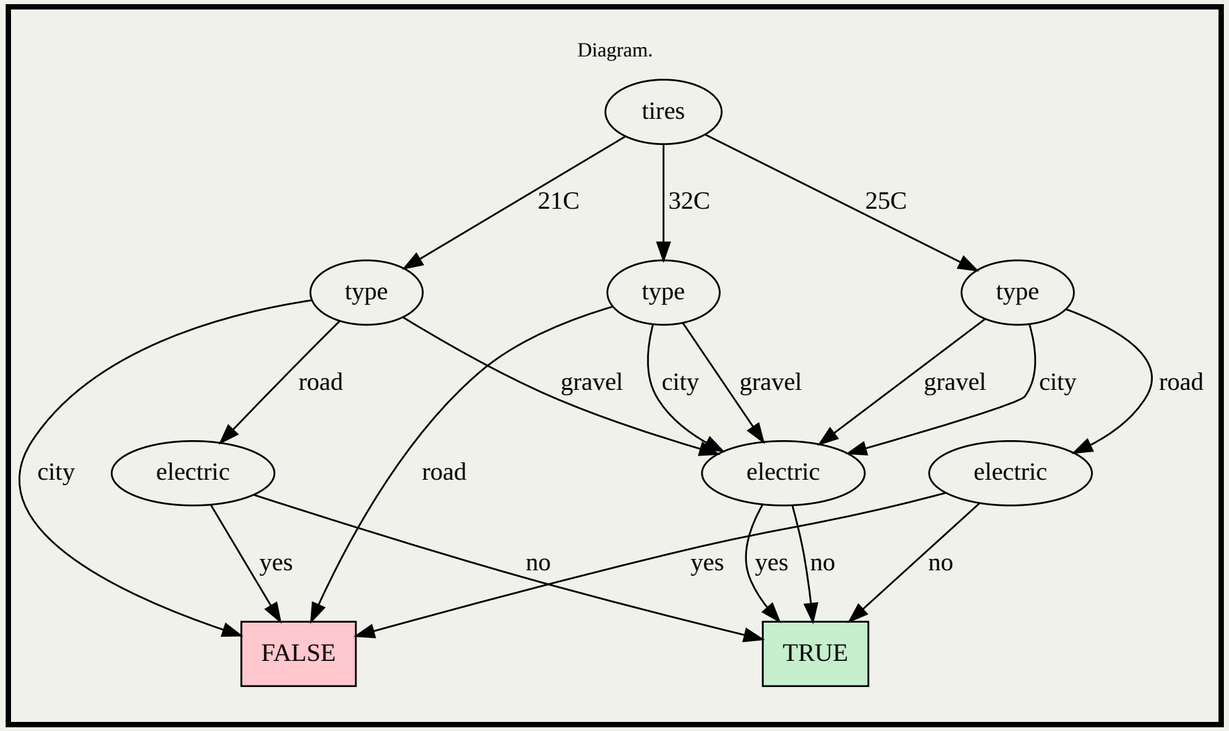
Set of constraints.

- ExactlyOne(road, gravel, city)
- ExactlyOne(21C, 25C, 32C)
- road ⇒ (21C ∨ 25C)
- city ⇒ ¬21C
- electric ⇒ (city ∨ gravel)
- electric ⇒ 32C

Set of models.

Tires	Type	Electric
21C	road	no
21C	gravel	yes
21C	gravel	no
25C	road	no
25C	city	yes
25C	city	no
25C	gravel	yes
25C	gravel	no
32C	city	yes
32C	city	no
32C	gravel	yes
32C	gravel	no

Diagram.



Knowledge Compilation today

Despite expert systems being outfashioned, KC is still relevant:

- *Practical applications*:
  - configuration systems
  - planning
- *Interesting abstraction*:
  - Insights into the **set of solutions**
  - Direct practical applications by (parcimonious) reduction to SAT.
- *Model counting* (#SAT):
  - leading knowledge compilers are also leading software for #SAT.
  - most agregation problems boil down to succinctly “representing the solutions” one way or another.

The Menu

I. An overview of interesting representation languages for Boolean functions.

II. How to build such representations.

III. How to prove limits of such representations.

Part I

Basics

Representation Language

A representation language $\mathcal{L} = (L, |\cdot|_{\mathcal{L}}, \| \cdot \|_{\mathcal{L}})$ for Boolean functions:

- L: set of objects
- $|\cdot|_{\mathcal{L}} : L \to \mathbb{N}$: size of objects (related to the number of bits needed to represent them)
- $\| \cdot \|_{\mathcal{L}}$ maps objects to a Boolean function.

Examples

	Object	Size	Interpretation
Truth Table	Variables X List of $\tau \in \{0,1\}^X$ with $b_\tau \in \{0,1\}$	$\lvert X\rvert + 2^{\lvert X\rvert}$	$f(\tau) = b_\tau$
List of models	Set $S \subseteq \{0,1\}^X$	$\lvert X\rvert \cdot \lvert S\rvert$	$f(\tau) = 1$ iff $\tau \in S$
CNF Formula	Set S of sets of literals	$\sum_{c \in S} \lvert c\rvert$	$f(\tau) = 1$ iff $\forall c \in S, \exists \ell \in c, \tau(\ell) = 1$
DNF Formula	Set S of sets of literals	$\sum_{c \in S} \lvert c\rvert$	$f(\tau) = 1$ iff $\exists c \in S, \forall \ell \in c, \tau(\ell) = 1$

- CNF formula $(x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee \neg y)$:
  - $x \mapsto 1, y \mapsto 0, z \mapsto 1$ is a model
  - $x \mapsto 1, y \mapsto 1, z \mapsto 1$ is not a model because clause $(\neg x \vee \neg y)$ is not satisfied.

- DNF formula $(x \wedge y) \vee (x \wedge \neg z) \vee (\neg x \wedge \neg y)$:
  - $x \mapsto 1, y \mapsto 1, z \mapsto 1$ is a model
  - $x \mapsto 1, y \mapsto 0, z \mapsto 1$ is not a model

Tractable Queries

Query $q(f, p)$ maps Boolean functions f and (optional) parameters p to a value.

- **Satisfiability**: $SAT(f) \in \{0, 1\}$: is f satisfiable?
- **Conditional Satisfiability**: $CO(f, \sigma) \in \{0, 1\}$: does f has a model τ extending σ
- **Ordered Satisfiability**: $OSAT(f, \sigma_1, \sigma_2) \in \{0, 1\}$: does f has a model τ st $\sigma_1 \leq \tau \leq \sigma_2$?
- **Model counting**: $MC(f) \in \mathbb{N}$: how many models does f have?
- **Conditional Model counting**: $CMC(f, \sigma) \in \mathbb{N}$: how many models extending σ does f have?

Tractable Queries

Query $q(f, p)$ maps Boolean functions f and (optional) parameters p to a value.

- **Satisfiability**: $SAT(f) \in \{0, 1\}$: is f satisfiable?
- **Conditional Satisfiability**: $CO(f, \sigma) \in \{0, 1\}$: does f has a model τ extending σ
- **Ordered Satisfiability**: $OSAT(f, \sigma_1, \sigma_2) \in \{0, 1\}$: does f has a model τ st $\sigma_1 \leq \tau \leq \sigma_2$?
- **Model counting**: $MC(f) \in \mathbb{N}$: how many models does f have?
- **Conditional Model counting**: $CMC(f, \sigma) \in \mathbb{N}$: how many models extending σ does f have?

$\mathcal{L}$ **supports** query $q(\cdot, \cdot)$ if given $o \in L$ and p, we can compute $q(\| o \|_{\mathcal{L}}, p)$ in ptime in $|o|_{\mathcal{L}}$.

We also say: q is **tractable** for $\mathcal{L}$.

Example of tractable queries

Details on board.

- CNF does not support satisfiability, nor model counting.
- DNF supports satisfiability but does not support model counting.
- Truth table supports satisfiability and model counting.

Example of tractable queries

Details on board.

- CNF does not support satisfiability, nor model counting.
- DNF supports satisfiability but does not support model counting.
- Truth table supports satisfiability and model counting.

“Not supporting” a query is often a conditional statement (e.g. “unless P $\neq$ NP).

Tractable Transformations

Transformation $t(f, p)$ maps Boolean functions f and (optional) parameters p to another Boolean function.

- $\neg(f)$ maps f to $1 - f$.
- Conditioning $COND(f, \sigma)$ maps f and partial assignment assignment σ to $f[\sigma]\colon \alpha \mapsto f(\alpha \times \sigma)$.

$\mathcal{L}$ **supports** transformation $t(\cdot, \cdot)$ if given $o \in L$ and p, we can compute $o^{'} \in L$ such that $\| o^{'} \|_{\mathcal{L}} = t(\| o \|_{\mathcal{L}}, p)$ in ptime in $|o|_{\mathcal{L}}$.
We also say: t is **tractable** for $\mathcal{L}$.

Example of tractable transformations

Details on board.

- CNF and DNF do not support negation but support conditioning: : $\bigwedge_{i=1}^{n}(x_i \vee y_i)$
- Truth table supports negation and conditioning.
- List of models supports conditioning but not negation.

Example of tractable transformations

Details on board.

- CNF and DNF do not support negation but support conditioning: : $\bigwedge_{i=1}^{n}(x_i \vee y_i)$
- Truth table supports negation and conditioning.
- List of models supports conditioning but not negation.

In many cases, we are able to show that a transformation is not supported unconditionnally.

Succinctness

$\mathcal{L}$ is more succinct that $\mathcal{L}^{'}$ written $\mathcal{L} < \mathcal{L}^{'}$ if there exists a polynomial p such that:

For every $L^{'} \in \mathcal{L}^{'}$, there exists $L \in \mathcal{L}$ such that:

- $\| L \|_{\mathcal{L}} = \| L^{'} \|_{\mathcal{L}^{'}}$
- $|L|_{\mathcal{L}} = p(|L^{'}|_{\mathcal{L}^{'}})$

> $\mathcal{L}$ can succinctly represent everything that $\mathcal{L}^{'}$ can (modulo a polynomial factor).

Succinctness examples

- List of models is *striclty* more succinct than truth tables.
- DNF is not comparable with CNF.
- Propositional formula is strictly more succinct than CNF.
- …

Compilability

Early theoretical notion (Cadoli, Liberatore, Schaerf [1]): *study the power of precomputation*.

Simplified version:

A query $q(f, p)$ is *compilable* if there exists a language $\mathcal{L}$ such that:

- For any propositional formula ϕ, there exists $o \in \mathcal{L}$ such that
  a. $\| o \|_{\mathcal{L}} = \| \phi \|_{Prop}$ and
  b. $|o|_{\mathcal{L}} = poly(|\phi|_{Prop})$.
- q is tractable in $\mathcal{L}$.

[1] Marco Cadoli, Francesco M. Donini, Paolo Liberatore, and Marco Schaerf. Preprocessing of intractable problems. Information and Computation, 176(2):89 – 120, 2002.

Compilability

Early theoretical notion (Cadoli, Liberatore, Schaerf [1]): *study the power of precomputation*.

Simplified version:

A query $q(f, p)$ is *compilable* if there exists a language $\mathcal{L}$ such that:

- For any propositional formula ϕ, there exists $o \in \mathcal{L}$ such that
  a. $\| o \|_{\mathcal{L}} = \| \phi \|_{Prop}$ and
  b. $|o|_{\mathcal{L}} = poly(|\phi|_{Prop})$.
- q is tractable in $\mathcal{L}$.

Observation / Example: if $q(f) \in S$ does not have parameter, then it is compilable.

1. $\mathcal{L} \subseteq Prop \times S$.
2. $(\phi, s) \in \mathcal{L}$ iff $q(\ \| \phi \|_{Prop}\ ) = s$.
3. q is tractable in $\mathcal{L}$ and any ϕ has a small representation in $\mathcal{L}$.

[1] Marco Cadoli, Francesco M. Donini, Paolo Liberatore, and Marco Schaerf. Preprocessing of intractable problems. Information and Computation, 176(2):89 – 120, 2002.

Compilability : bad news

Most interesting queries with parameters are not compilable (unless NP $\subseteq$ P/poly). [1]

Candidate: $q(f, \tau)$: does f has a model extending τ?

[1] Marco Cadoli, Francesco M Donini, and Marco Schaerf. Is intractability of non-monotonic reasoning a real drawback? Artificial intelligence, 88(1-2):215–251, 1996.

Compilability : bad news

Most interesting queries with parameters are not compilable (unless NP $\subseteq$ P/poly). [1]

Candidate: $q(f, \tau)$: does f has a model extending τ?

If q is compilable, then NP $\subseteq$ P/poly. More precisely, 3-SAT can be solved with non-uniform polynomial size circuits.

Proof (details on board): by compiling

$$\phi_n = \bigwedge_{c \in \text{3-clauses}(n)} (c \vee s_c)$$

[1] Marco Cadoli, Francesco M Donini, and Marco Schaerf. Is intractability of non-monotonic reasoning a real drawback? Artificial intelligence, 88(1-2):215–251, 1996.

Where to go from there

No representation language fits.

- Study the *most general representation language* for the set of queries the user wants to solve.
- Previous result is *worst case*. The formula the user is interested in may have a small representation anyway.
- Design *compilers* to find such small representations when they exist.

The Knowledge Compilation Maps

Term coined by Darwiche and Marquis in [1].

Help the practicionner “navigates” languages and their properties using tables.

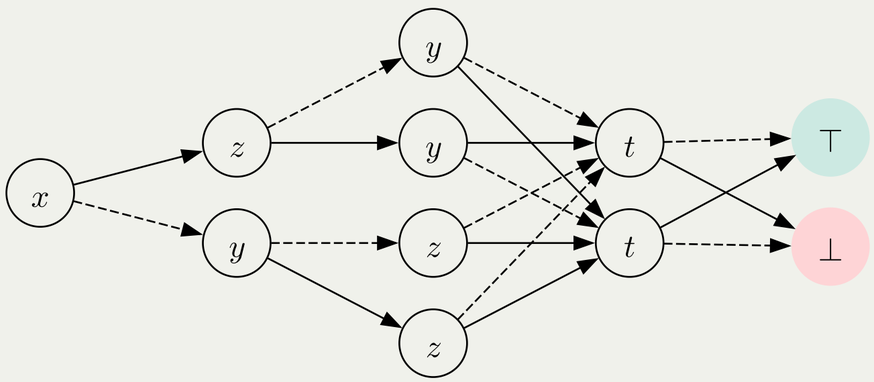
And now, a guided tour of the Circuit Zoo by David Kahdian, Oliver Broadrick, Renato Geh, and Guy Van den Broeck.

[1]: A. Darwiche and P. Marquis, “A Knowledge Compilation Map,” Journal of Artificial Intelligence Research, vol. 17, pp. 229–264, 2002.

A case study: OBDDs

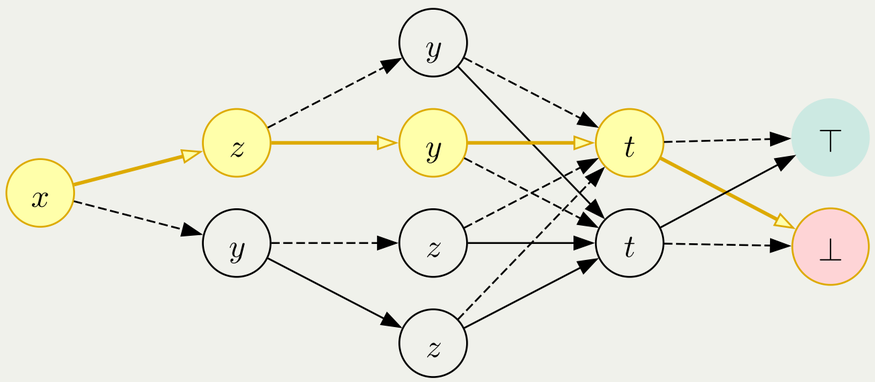
Free Binary Decision Diagrams (FBDD)

Represent functions by successively testing variables:



Free Binary Decision Diagrams (FBDD)

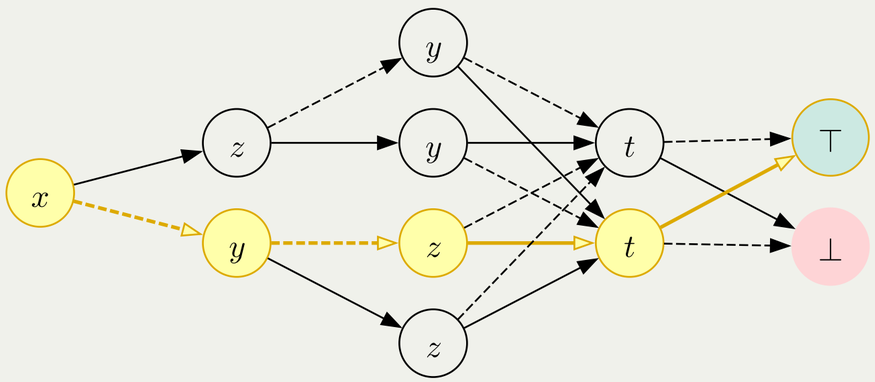
Represent functions by successively testing variables:



$x = 1, y = 1, z = 1, t = 1$ is a not model.

Free Binary Decision Diagrams (FBDD)

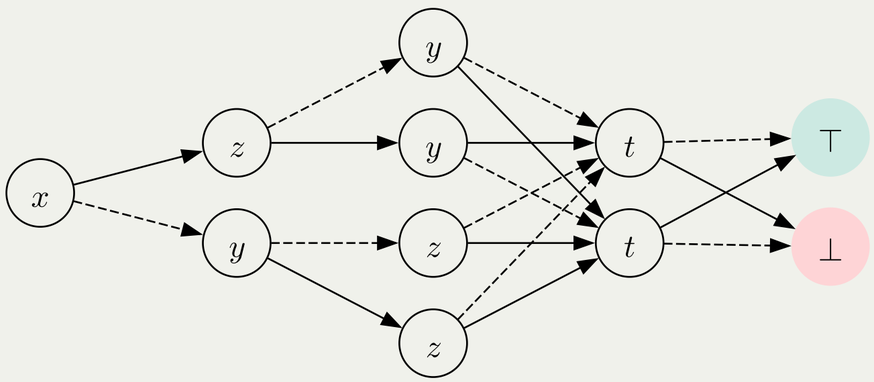
Represent functions by successively testing variables:



$x = 0, y = 0, z = 1, t = 1$ is a model.

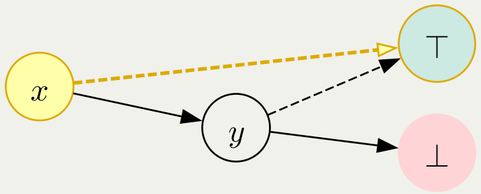
Free Binary Decision Diagrams (FBDD)

Represent functions by successively testing variables:



Read-once: on a path, a variable can be tested at most once.

Missing variables

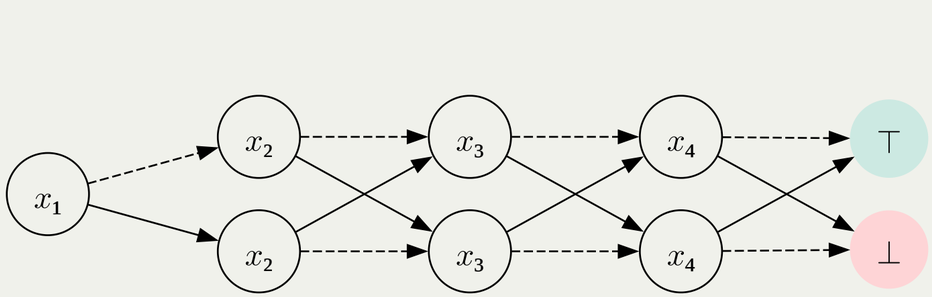


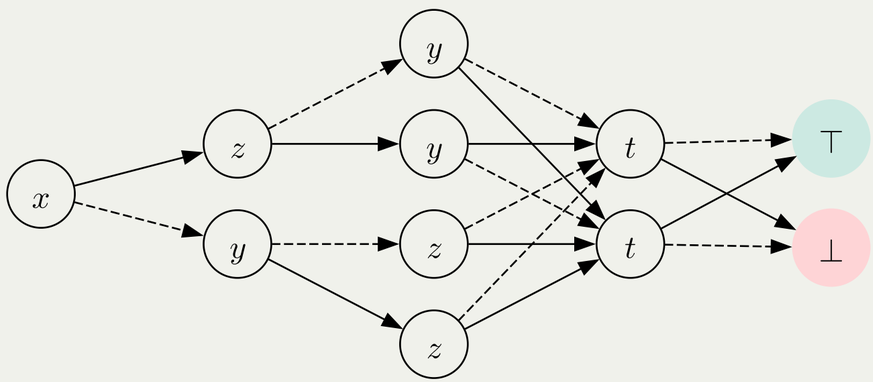
Both $x = 0, y = 0$ and $x = 0, y = 1$ are models.

Observation: this is a convention. Other popular convention: missing variables are default to 0 (in which case, we call the structure a ZDD).

Ordered BDD (OBDD)

Variables tested in a fixed order:

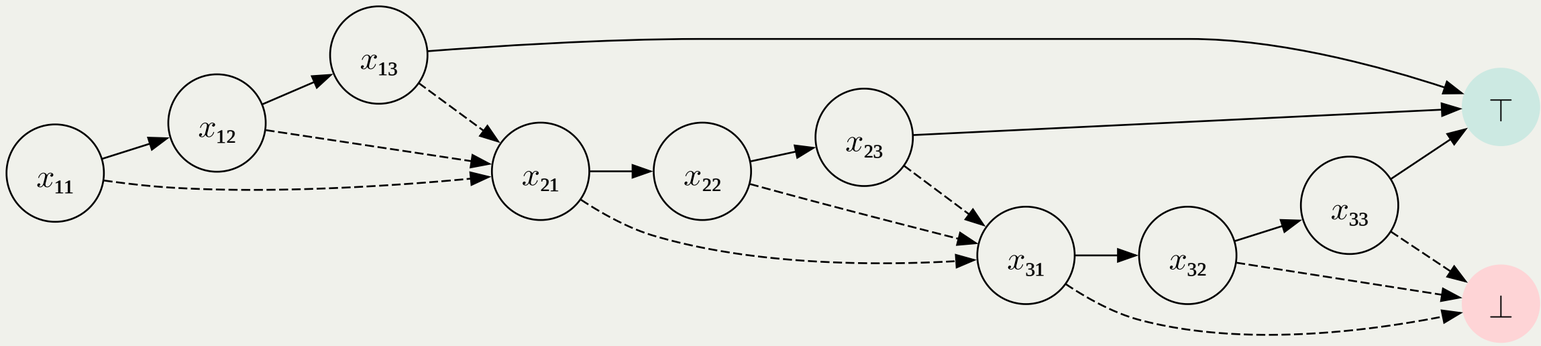




Order Matters

$ROW_n(x_{1,1}, ..., x_{n,n})$: the matrix contains a row of 1.

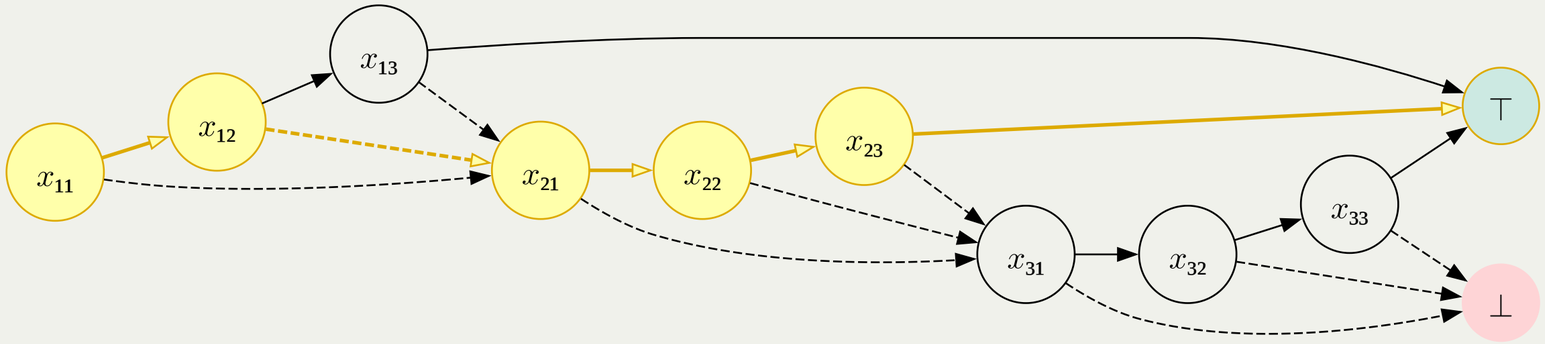
Easy when testing row by row ($x_{1,1}, x_{1,2}, ..., x_{1,n}, x_{2,1}, x_{2,2}, ..., x_{2,n}, ..., x_{n,1}, x_{n,2}, ..., x_{n,n}$).



Order Matters

$ROW_n(x_{1,1}, ..., x_{n,n})$: the matrix contains a row of 1.

Easy when testing row by row ($x_{1,1}, x_{1,2}, ..., x_{1,n}, x_{2,1}, x_{2,2}, ..., x_{2,n}, ..., x_{n,1}, x_{n,2}, ..., x_{n,n}$).

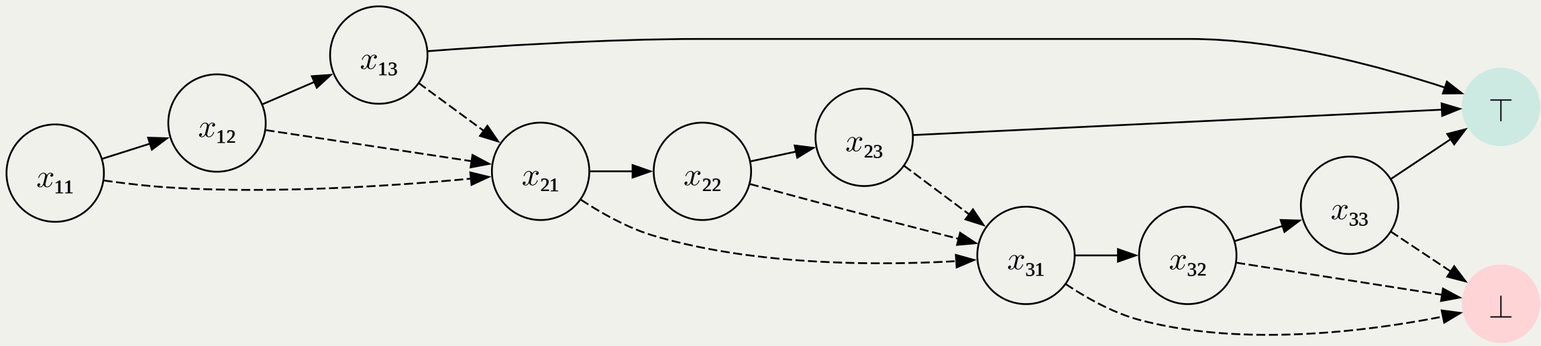


1	0	0
1	1	1
1	0	0

Order Matters

$ROW_n(x_{1,1}, ..., x_{n,n})$: the matrix contains a row of 1.

Easy when testing row by row ($x_{1,1}, x_{1,2}, ..., x_{1,n}, x_{2,1}, x_{2,2}, ..., x_{2,n}, ..., x_{n,1}, x_{n,2}, ..., x_{n,n}$).

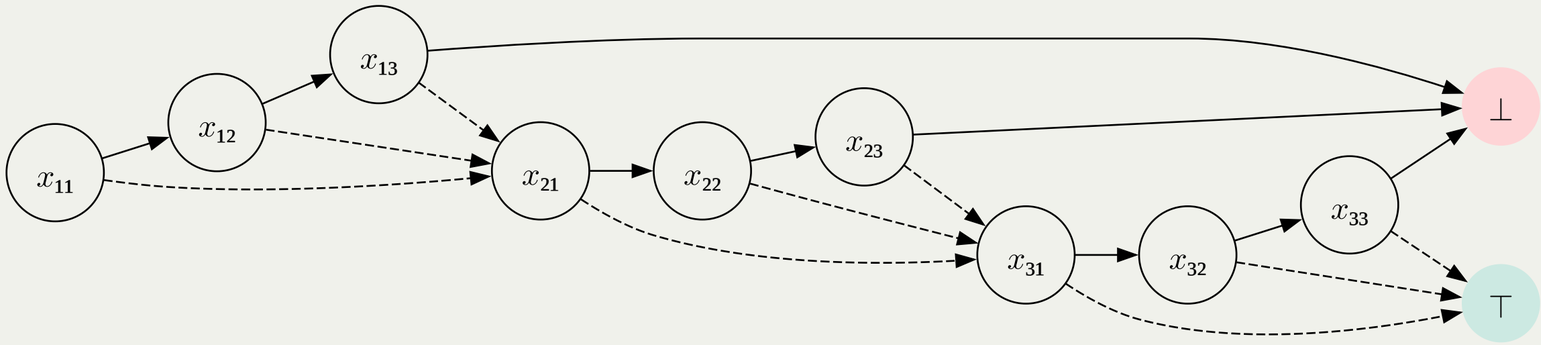


Hard when testing col by col ($x_{1,1}, x_{2,1}, ..., x_{n,1}, x_{1,2}, x_{2,2}, ..., x_{n,2}, ..., x_{1,n}, x_{2,n}, ..., x_{n,n}$).

Negation

FBDD and OBDD support **Negation**: swap $\top$ and $\bot$.

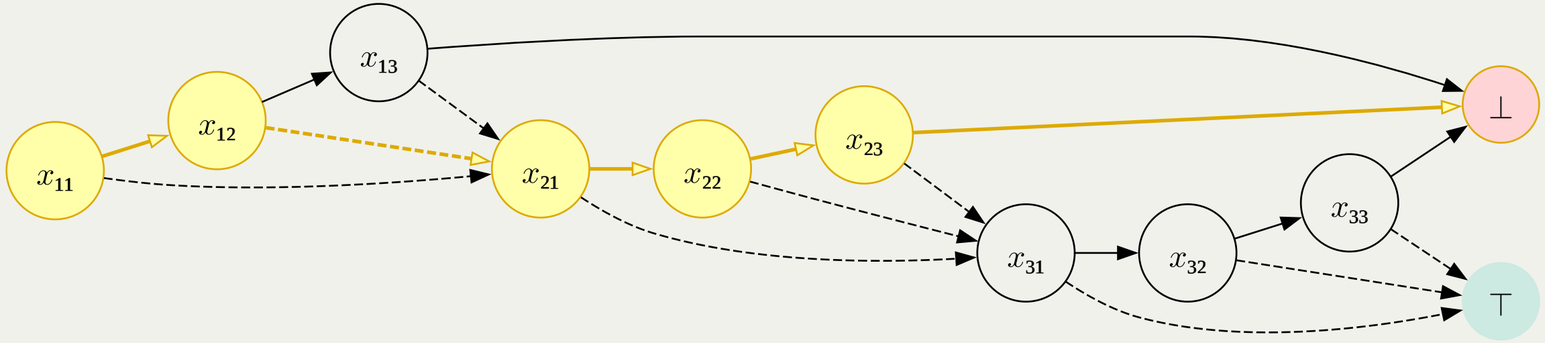
For example, $\neg ROW_3$:



Negation

FBDD and OBDD support **Negation**: swap $\top$ and $\bot$.

For example, $\neg ROW_3$:

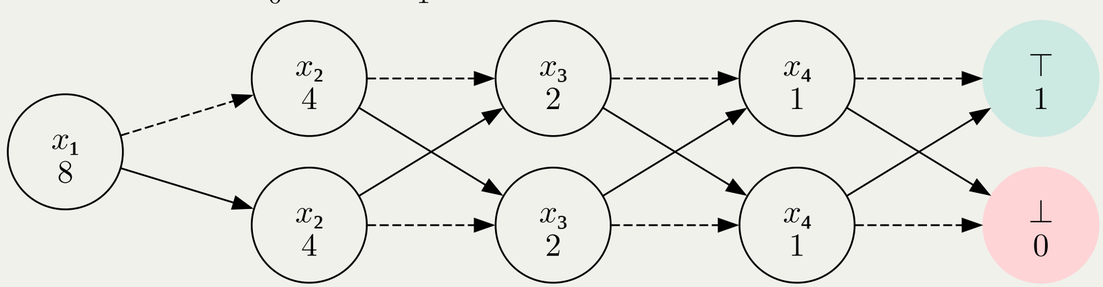


1	0	0
1	1	1
1	0	0

Counting with BDDs

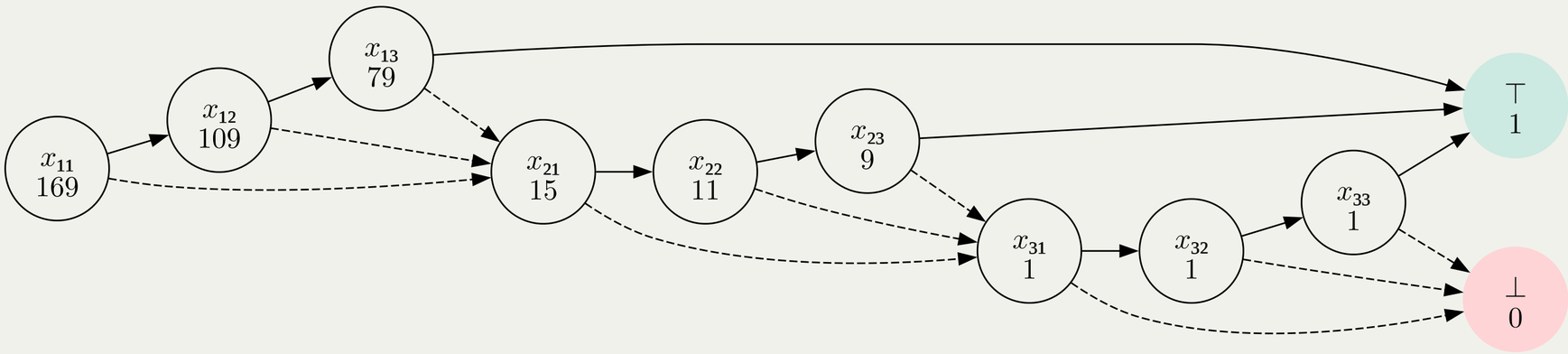
Dynamic programming algorithm:

- $N(g)$: number of models starting at node g
- $N(g) = N(g_0) + N(g_1)$



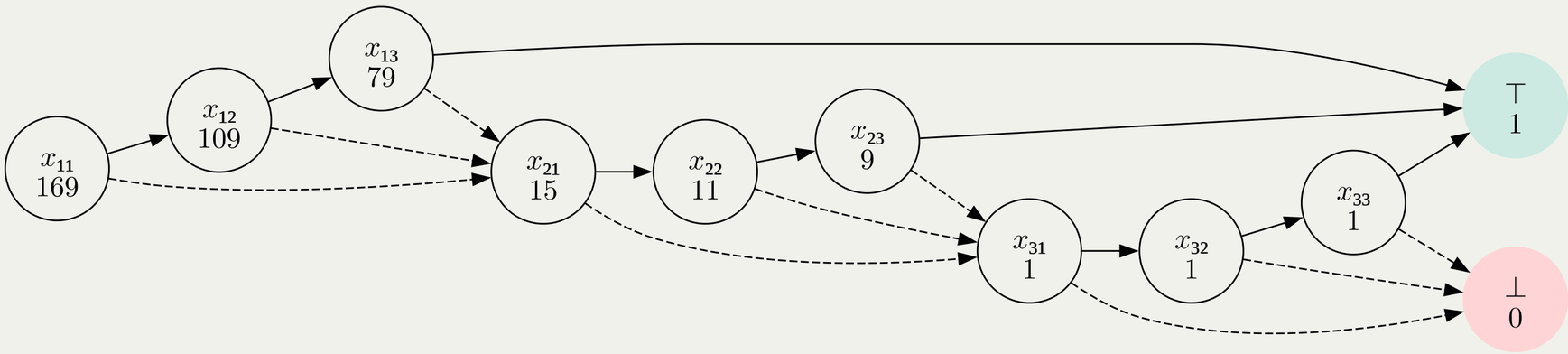
Counting: mind Δ

The previous algorithm is wrong: *missing variables are not accounted for*.



Counting: mind Δ

The previous algorithm is wrong: *missing variables are not accounted for*.



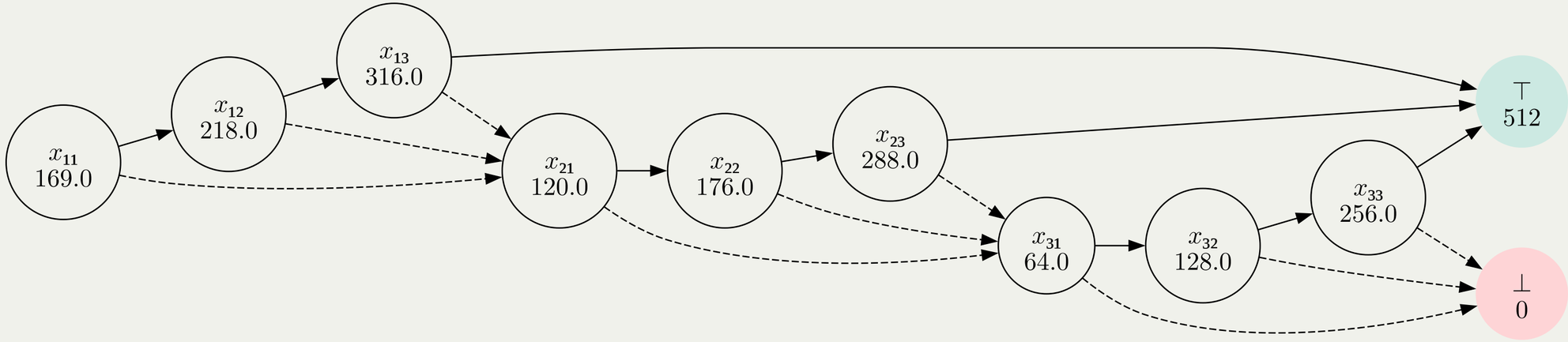
$$N(g) = 2^{|\Delta_0|}N(g_0) + 2^{|\Delta_1|}N(g_1)$$

where $\Delta_0 = var(g_1) \backslash var(g_0)$ and $\Delta_1 = var(g_0) \backslash var(g_1)$

Counting without Δ

Computing the variables after each gate is a bit costly (set operations). Other algorithm:

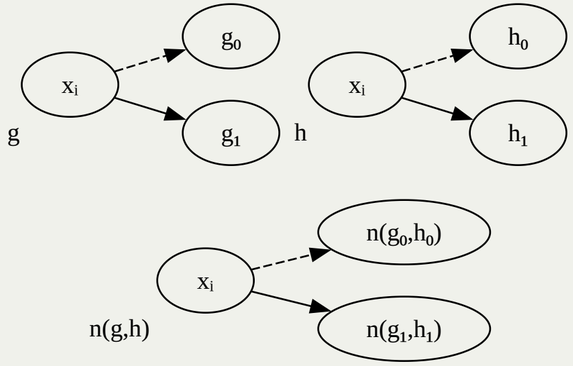
- $N_X(g)$: the number of assignments of X that are accepted by g,
- $N_X(g) = 0.5(N_X(g_0) + N_X(g_1))$,
- $N_X(\top) = 2^{|X|}$, $N_X(\bot) = 0$.



Conjunction

Theorem
Given two OBDD C_1, C_2 using the **same order** $\pi = x_1, ..., x_n$, we can construct an OBDD C' computing $C_1 \wedge C_2$ of size $|C_1| \cdot |C_2|$.

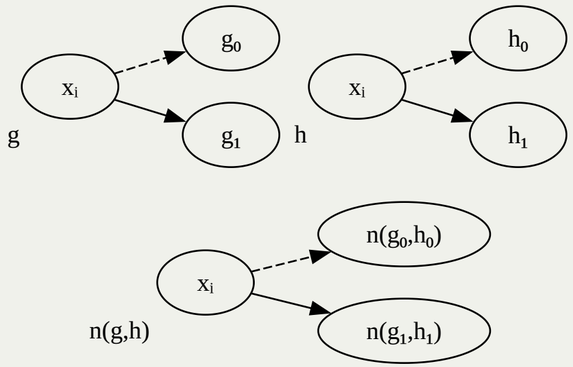
Product construction: for g in C_1 and h in C_2, build $n(g, h)$ accepting assignments that are accepted by both g and h.

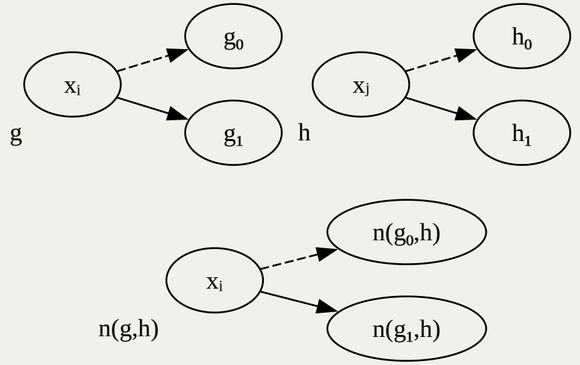


Conjunction

Theorem
Given two OBDD C_1, C_2 using the **same order** $\pi = x_1, ..., x_n$, we can construct an OBDD $C^{'}$ computing $C_1 \wedge C_2$ of size $|C_1| \cdot |C_2|$.

Product construction: for g in C_1 and h in C_2, build $n(g, h)$ accepting assignments that are accepted by both g and h.





if $i < j$.

APPLY

Since Negation and Conjunction are supported:

Theorem

Given two OBDD C_1, C_2 using the **same order** $\pi = x_1, ..., x_n$, $\diamond$ a Boolean operator, we can construct an OBDD C' computing $C_1 \diamond C_2$ of size $O(|C_1| \cdot |C_2|)$.

We write: $APPLY(C_1, C_2, \diamond)$.

Canonicity

Theorem
For a function f on X and an order π on X, there is a **unique** and **minimal** OBDD computing f and respecting π.

Iteratively apply **two simplifications**:

Wrap-up on OBDD

OBDD are **highly tractable**, and have nice properties (minimisation and canonicity).

Many supported **queries** and **transformations** not presented:

- weighted counting,
- model enumeration,
- direct access,
- sampling,
- …

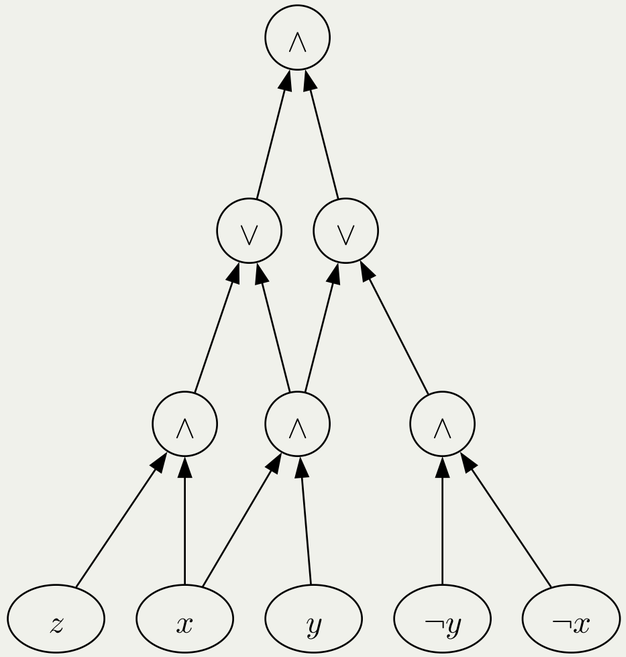
Not the most succinct data structure however, **order** is enforcing a lot of structure.

Circuit Based Representation

Negation Normal Form Circuits (NNF)

Boolean circuits with:

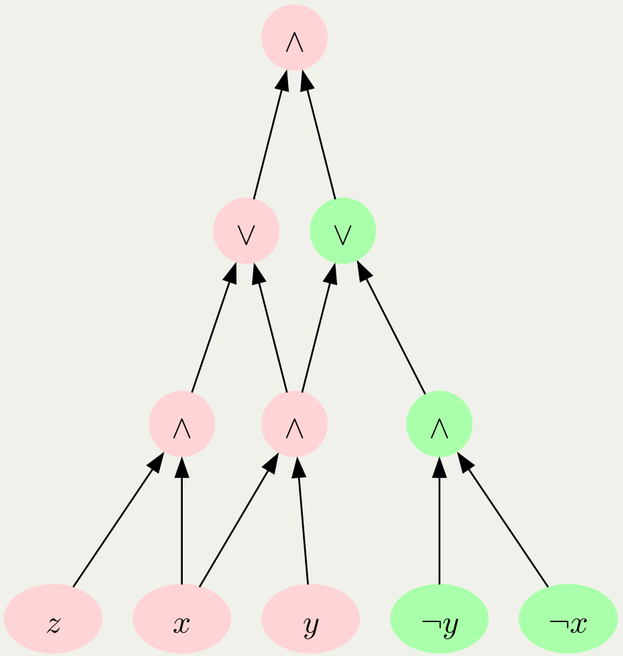
- Gates: $\wedge$ (AND) and $\vee$ (OR)
- Inputs: literals



Negation Normal Form Circuits (NNF)

Boolean circuits with:

- Gates: $\wedge$ (AND) and $\vee$ (OR)
- Inputs: literals

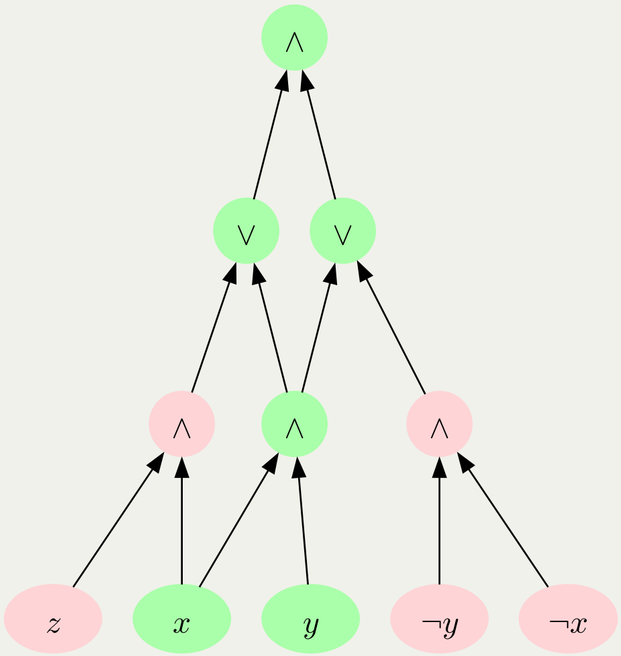


$x = 0, y = 0, z = 0$ is not a model.

Negation Normal Form Circuits (NNF)

Boolean circuits with:

- Gates: $\wedge$ (AND) and $\vee$ (OR)
- Inputs: literals

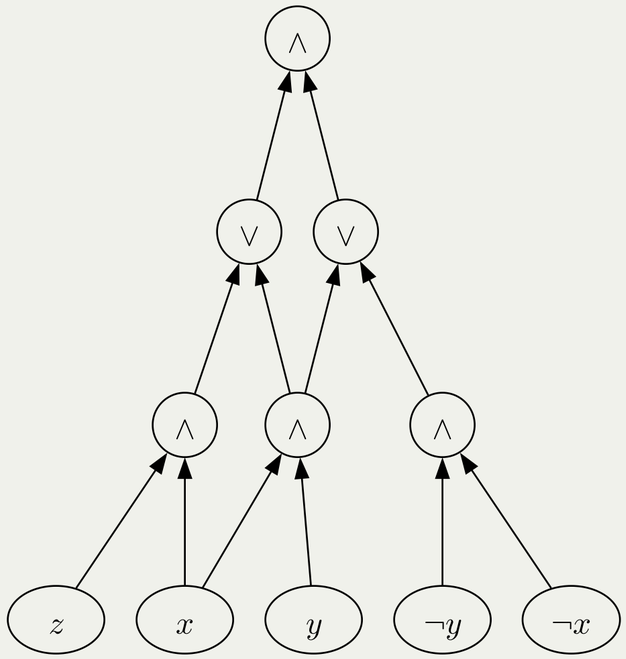


$x = 1, y = 1, z = 0$ is a model.

Negation Normal Form Circuits (NNF)

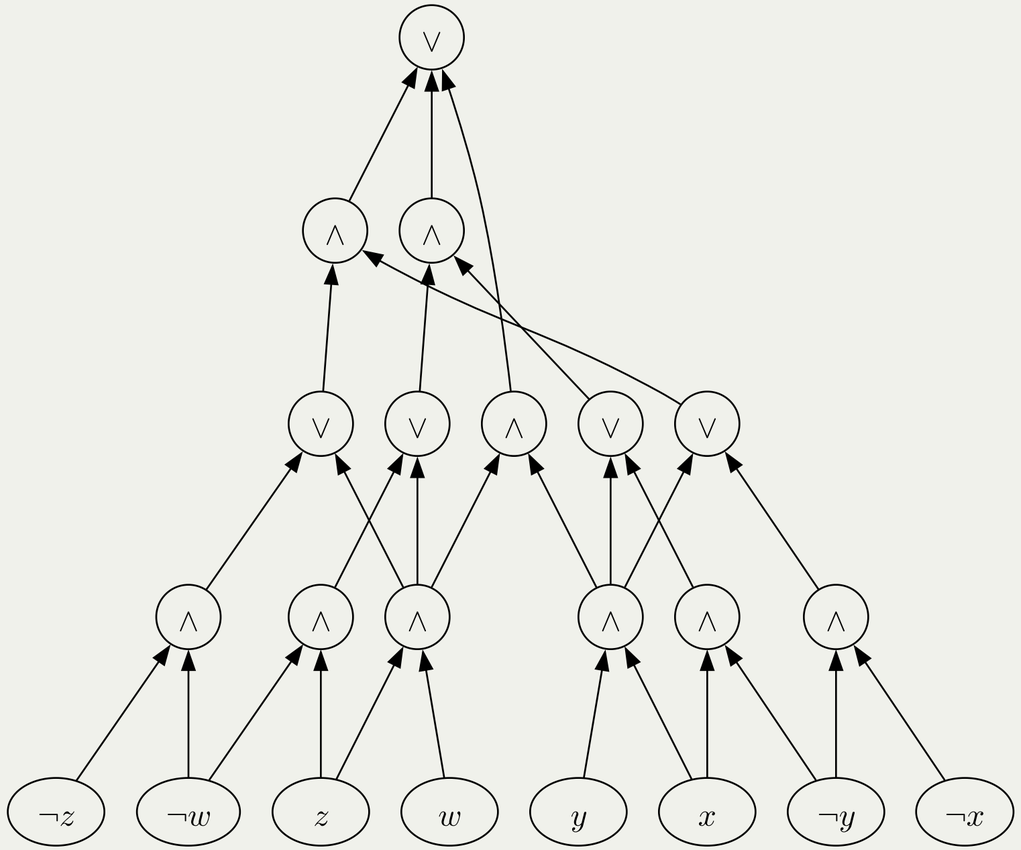
Boolean circuits with:

- Gates: $\wedge$ (AND) and $\vee$ (OR)
- Inputs: literals



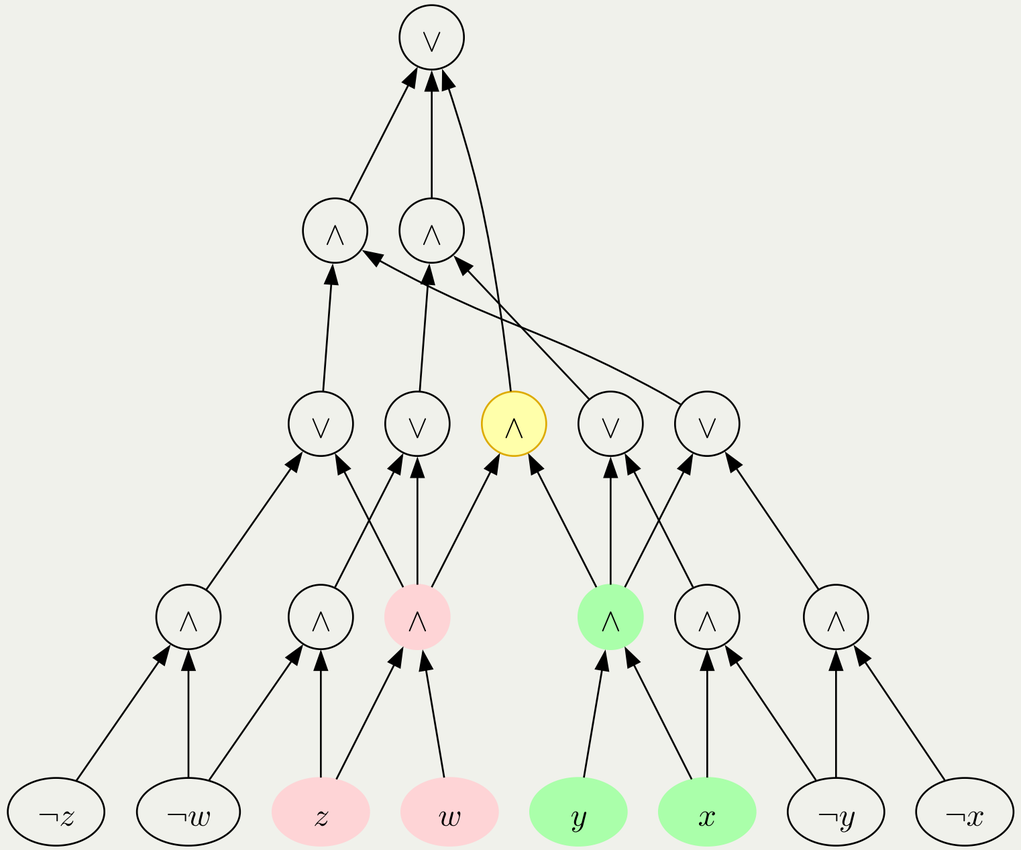
NNF circuits are *as succinct as Boolean circuits* but not **tractable**: deciding whether they have a model is NP-complete.

Decomposable NNF (DNNF)



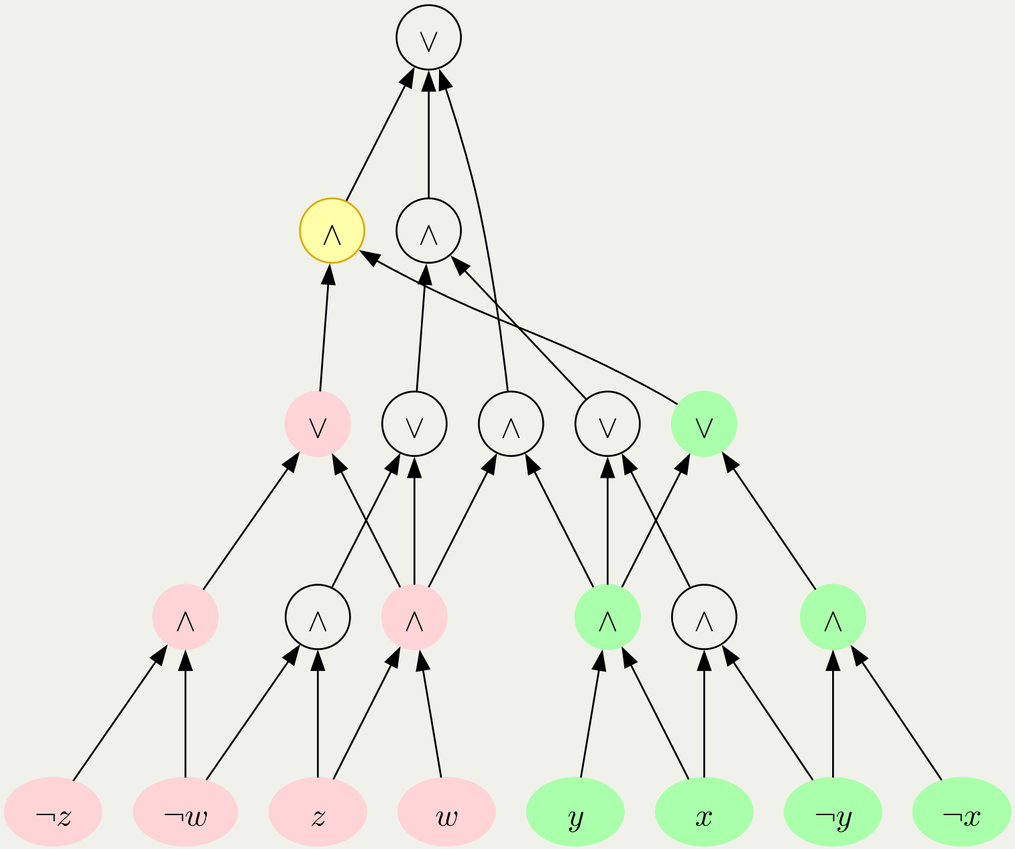
Every $\wedge$-node is *decomposable*: each input mentions distinct variables.

Decomposable NNF (DNNF)



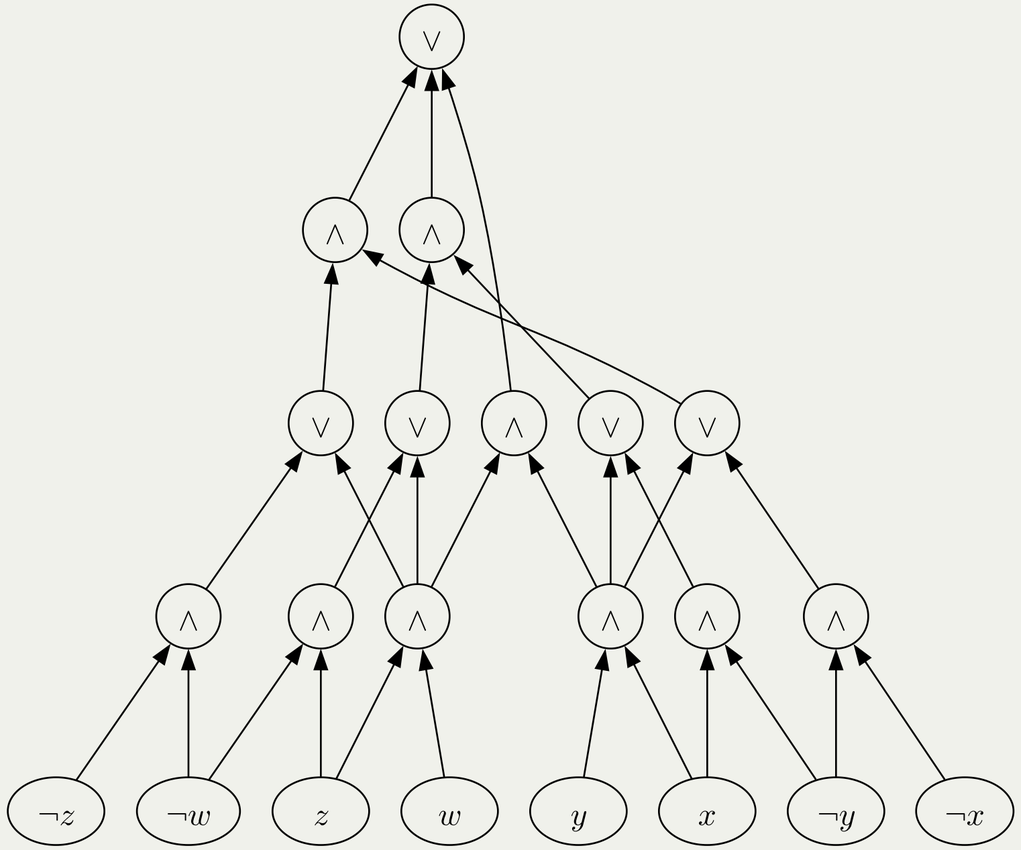
Every $\wedge$-node is *decomposable*: each input mentions distinct variables.

Decomposable NNF (DNNF)



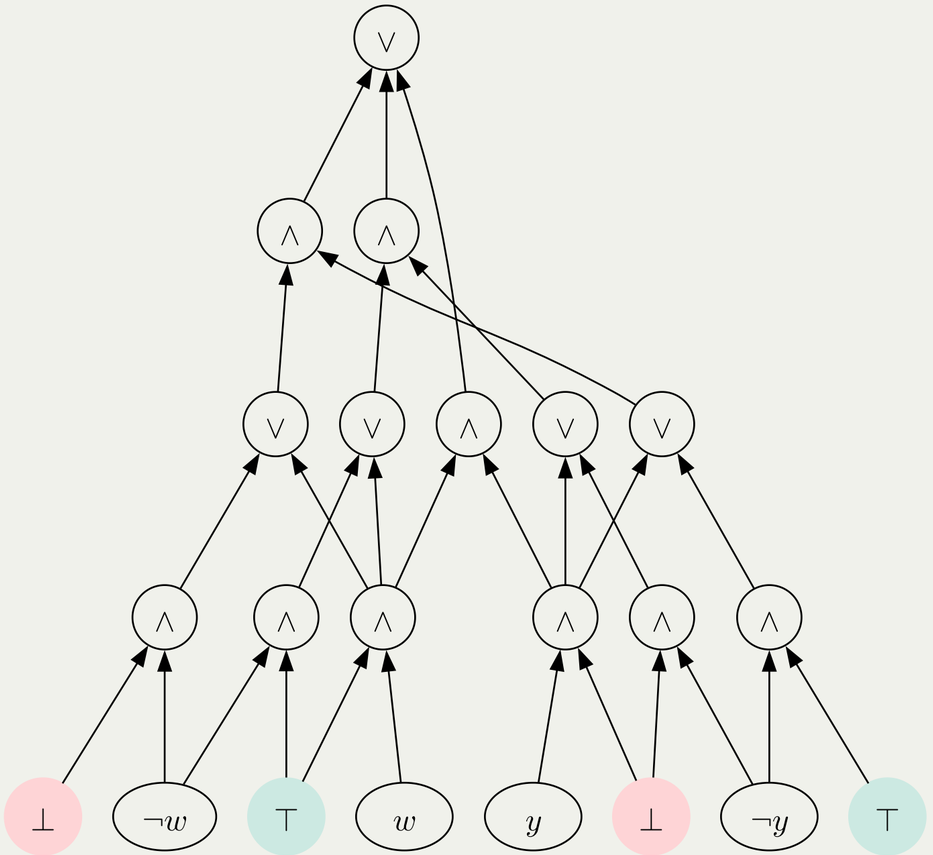
Every $\wedge$-node is *decomposable*: each input mentions distinct variables.

Finding models of DNNF



Given partial assignment τ, decide whether there is a model of C extending τ.

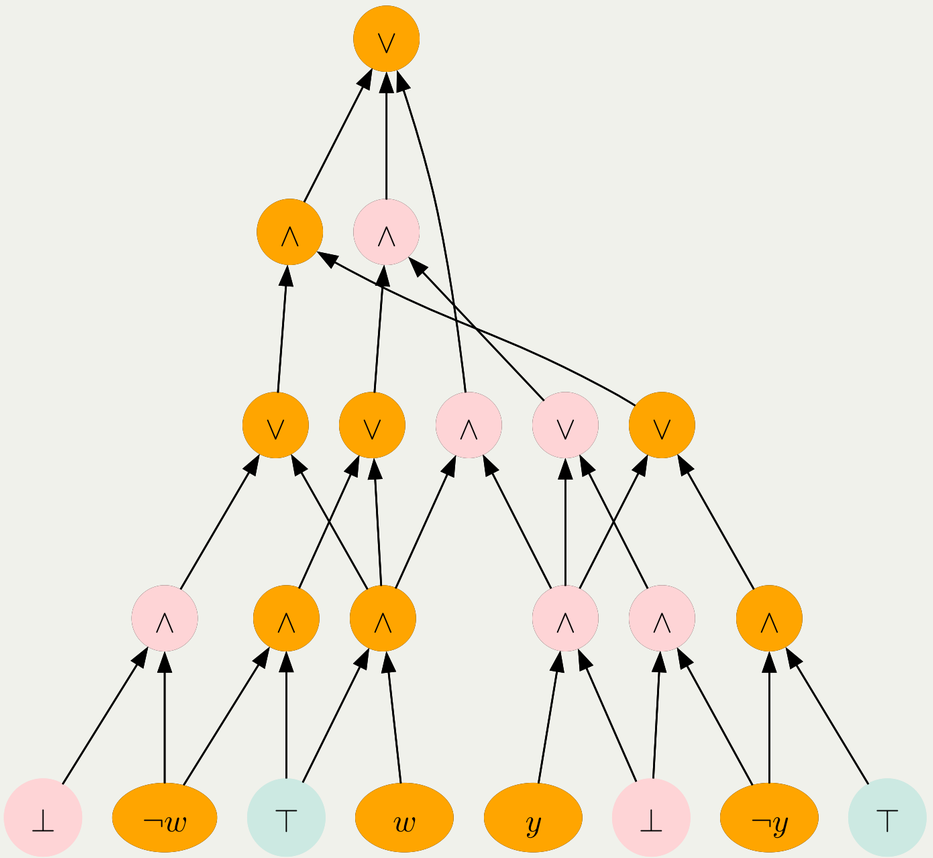
Finding models of DNNF



Given partial assignment τ, decide whether there is a model of C extending τ.

- Fix τ in the circuit (here $x = 0, z = 1$).

Finding models of DNNF

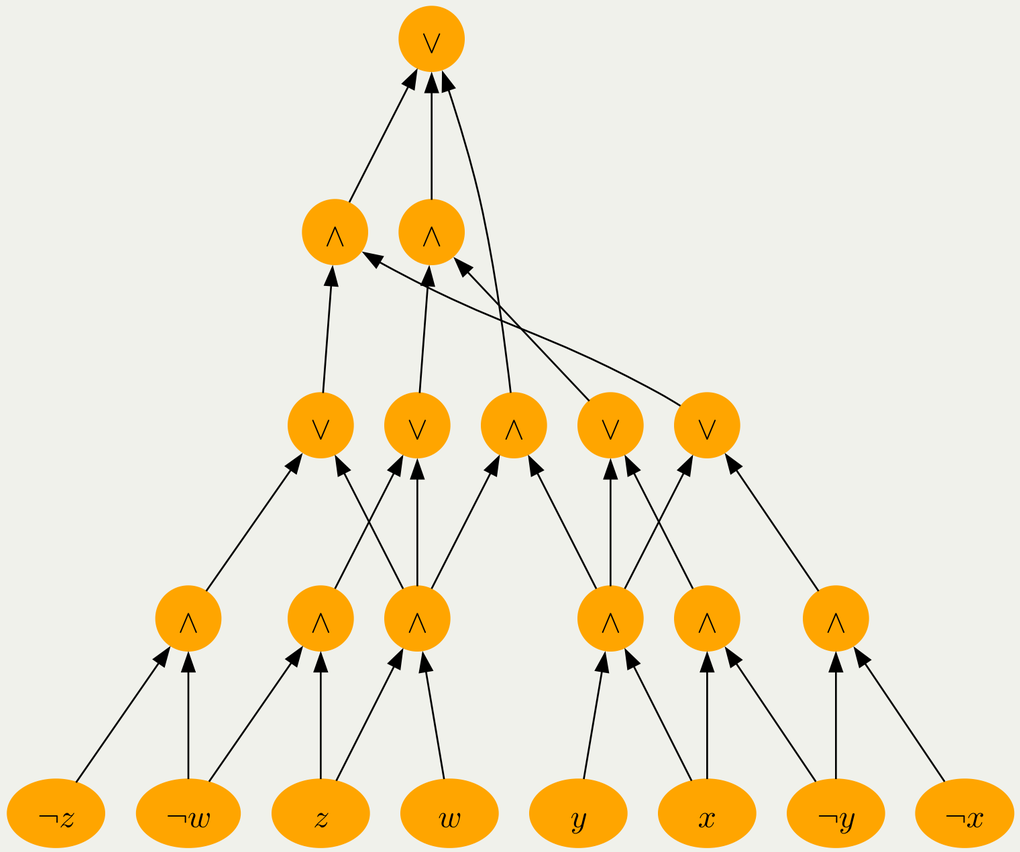


Given partial assignment τ, decide whether there is a model of C extending τ.

- Fix τ in the circuit (here $x = 0, z = 1$).
- Propagate upward the information: “the gate has a model”.
- A *decomposable* $\wedge$-gate g has a model **iff** every input g has a model.

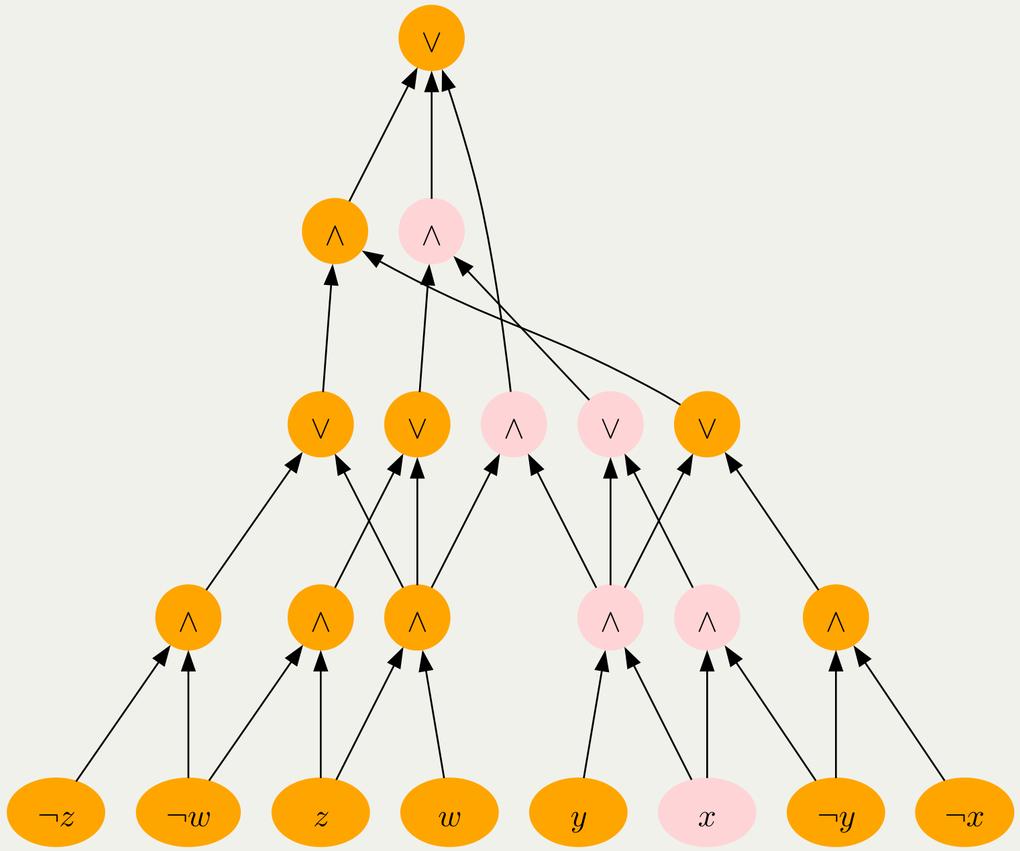
Finding all models of a DNNF

Finding all models of a DNNF



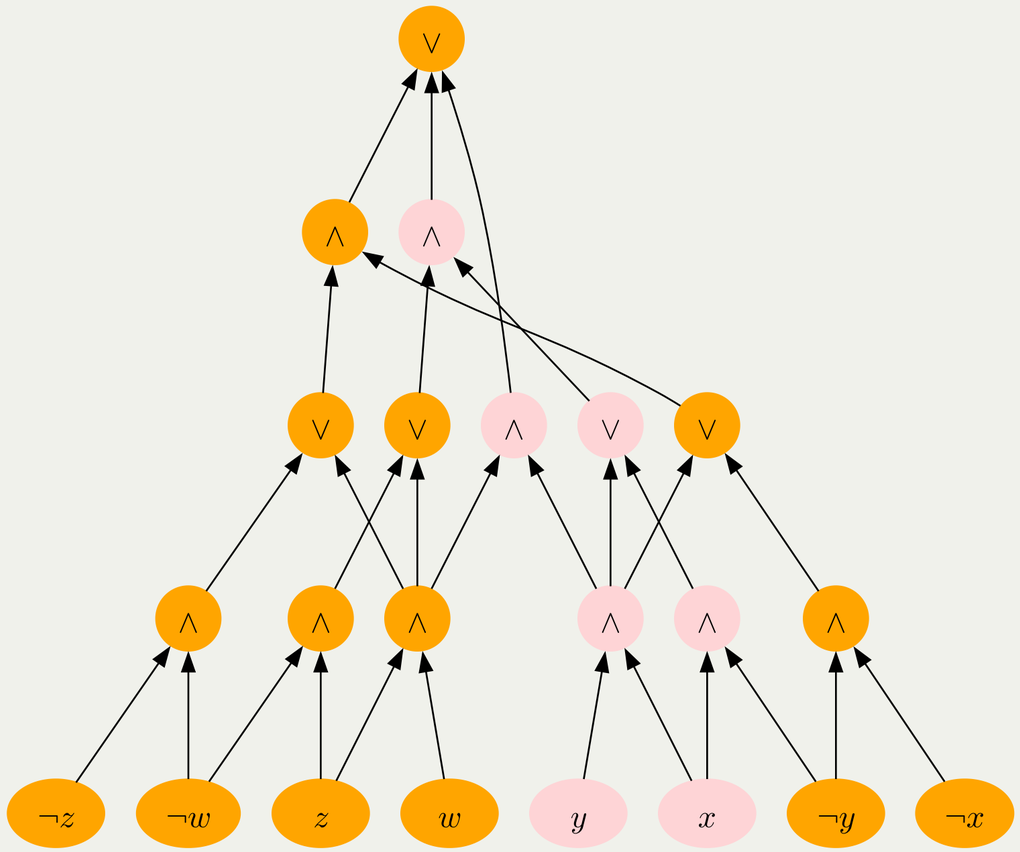
Conditioning on <>. Satisfiable. Keep going

Finding all models of a DNNF



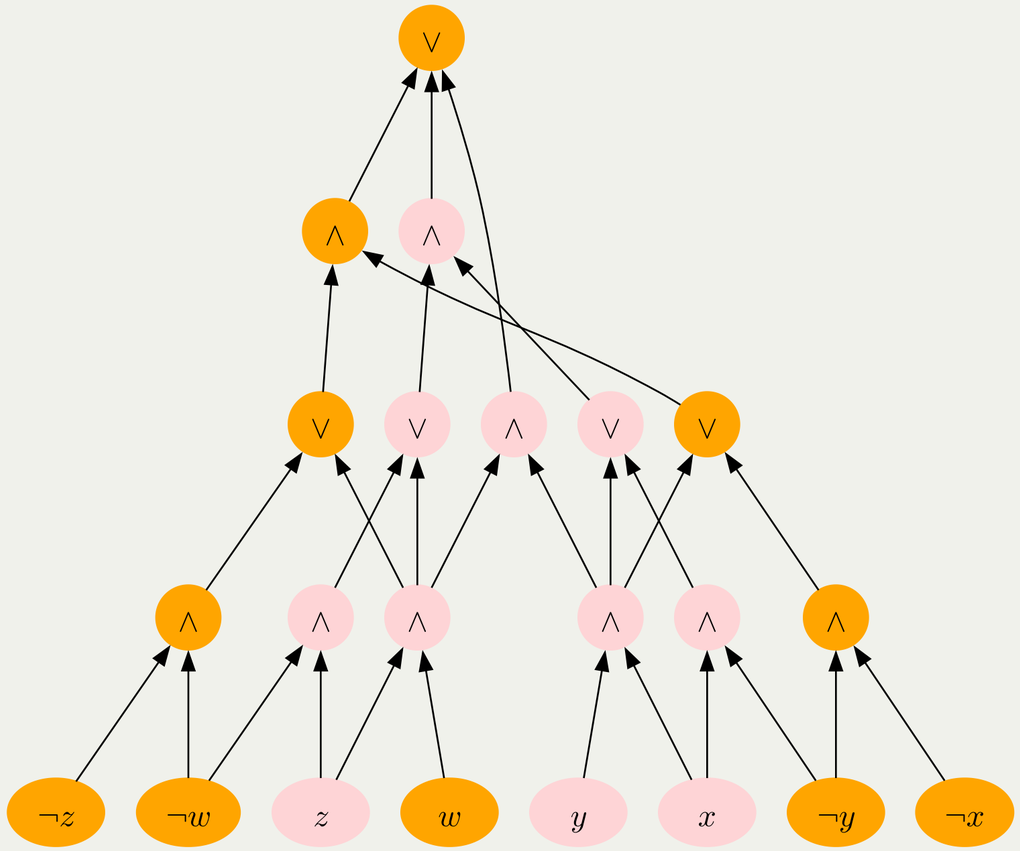
Conditioning on <x: 0>. Satisfiable. Keep going

Finding all models of a DNNF



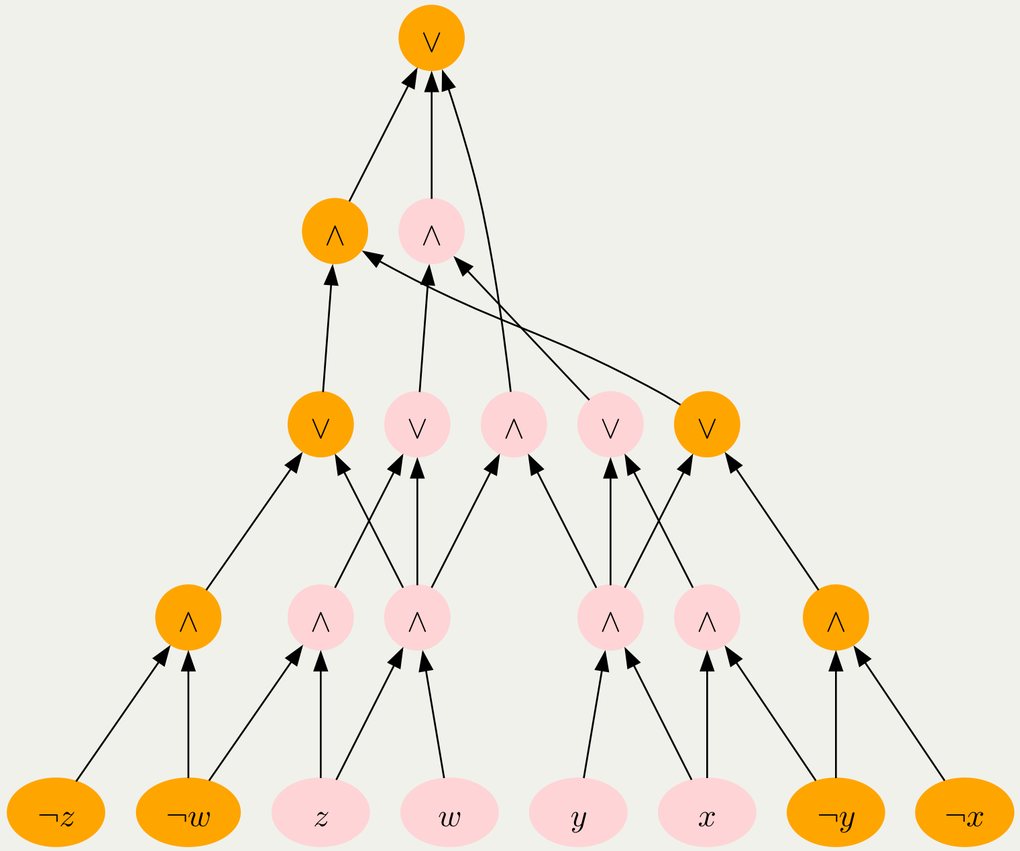
Conditioning on <x: 0,y: 0>. Satisfiable. Keep going

Finding all models of a DNNF



Conditioning on <x: 0,y: 0,z: 0>. Satisfiable. Keep going

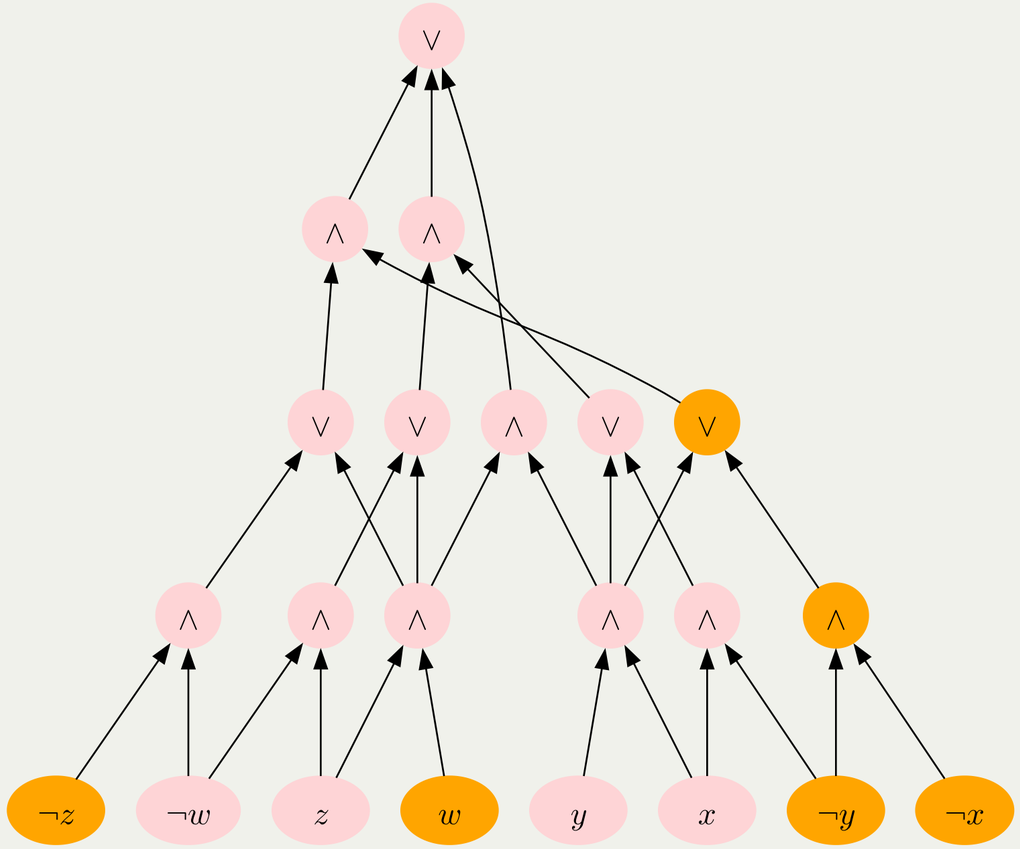
Finding all models of a DNNF



x: 0,y: 0,z: 0,w: 0

Conditioning on <x: 0,y: 0,z: 0,w: 0>. Model found.

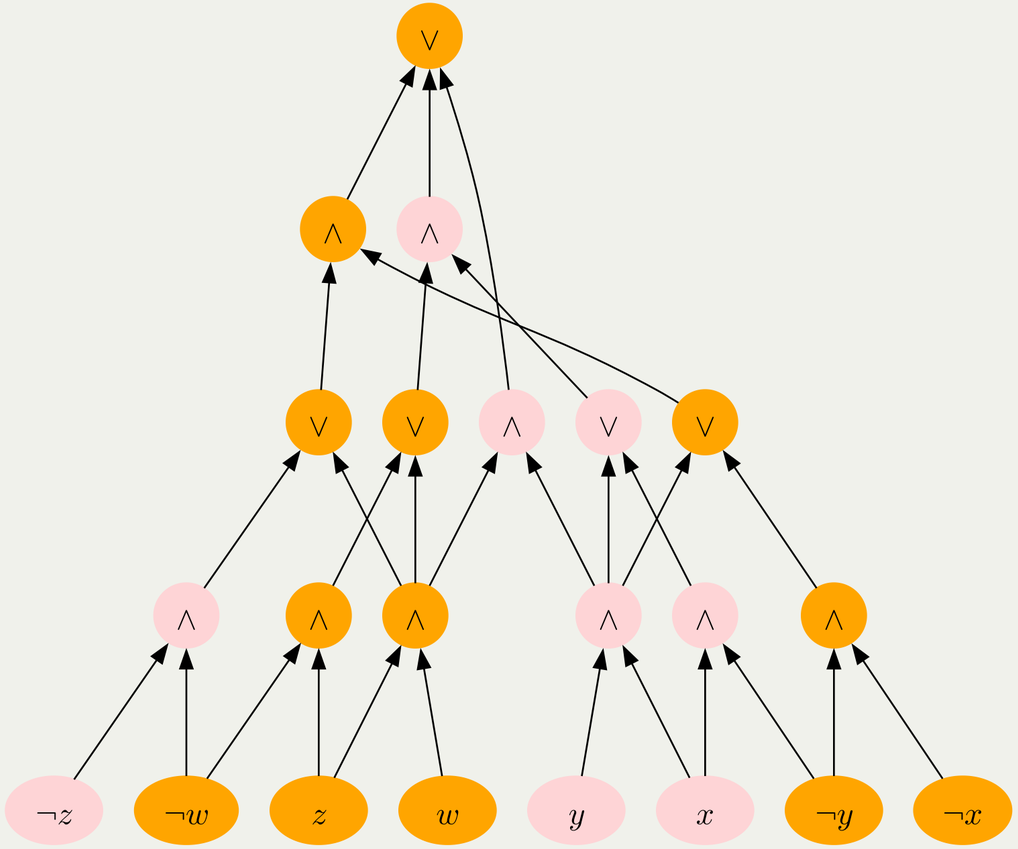
Finding all models of a DNNF



x: 0,y: 0,z: 0,w: 0

Conditioning on <x: 0,y: 0,z: 0,w: 1>. Not satisfiable: backtrack

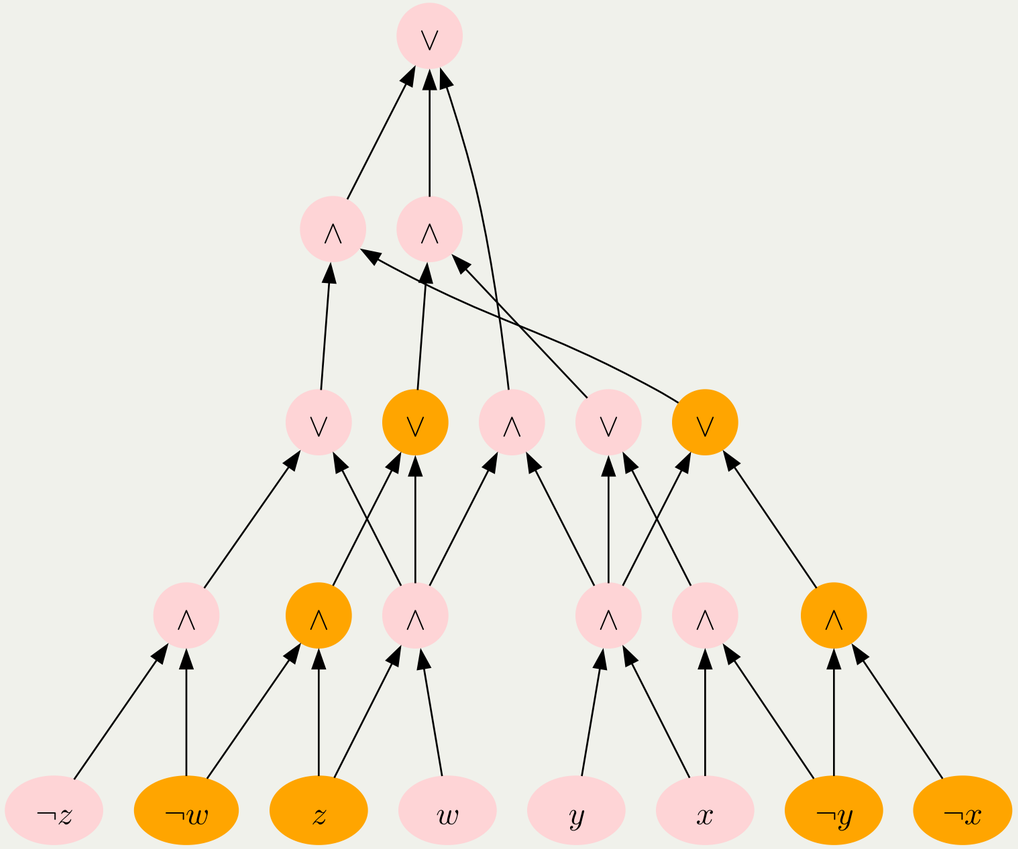
Finding all models of a DNNF



x: 0,y: 0,z: 0,w: 0

Conditioning on <x: 0,y: 0,z: 1>. Satisfiable. Keep going

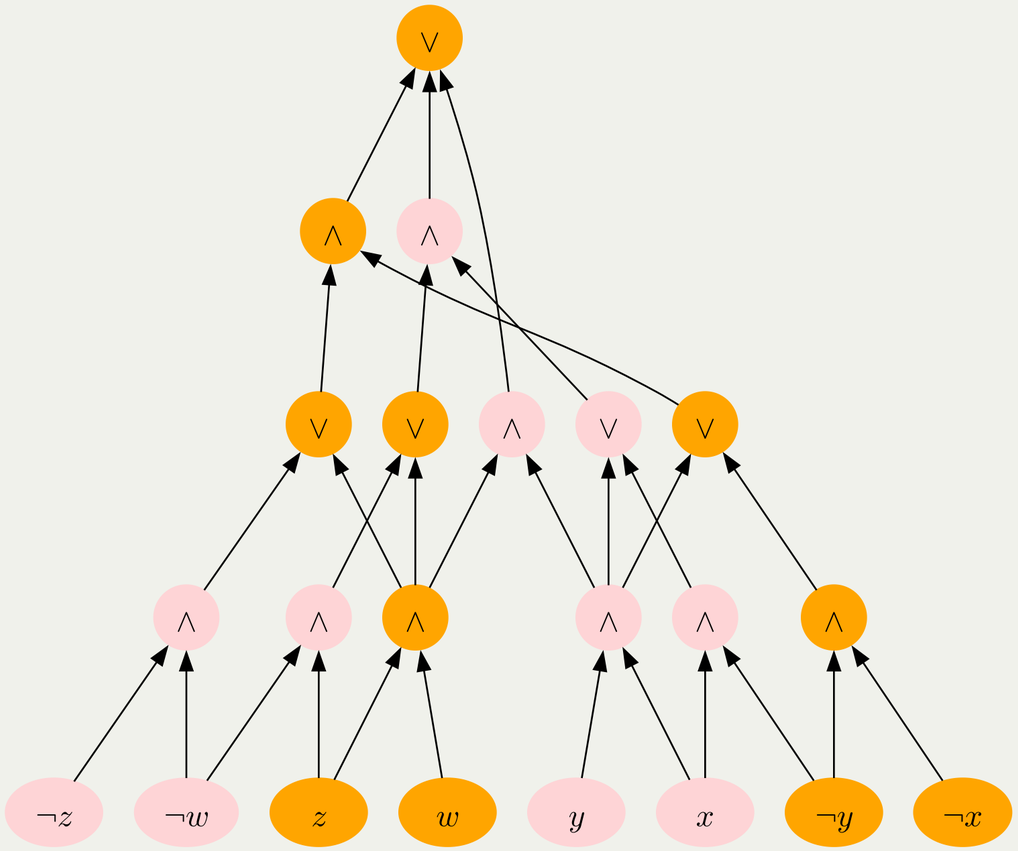
Finding all models of a DNNF



x: 0,y: 0,z: 0,w: 0

Conditioning on <x: 0,y: 0,z: 1,w: 0>. Not satisfiable: backtrack

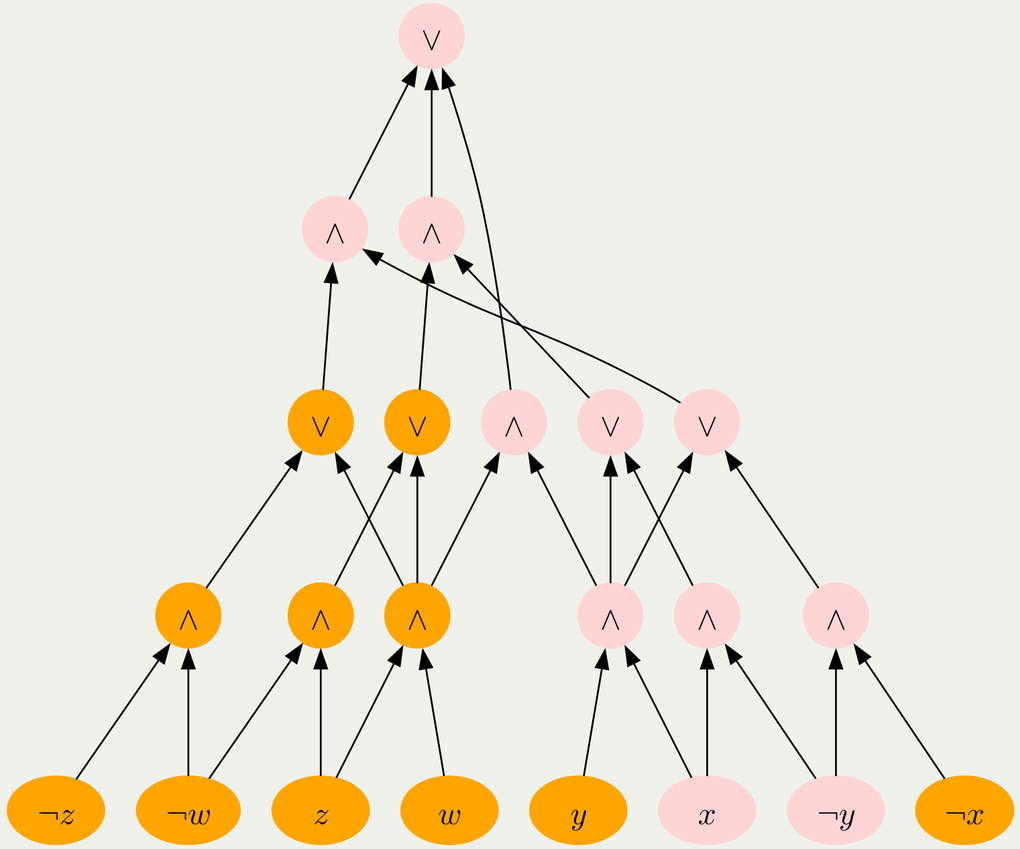
Finding all models of a DNNF



Conditioning on <*x*: 0,*y*: 0,*z*: 1,*w*: 1>. Model found.

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1

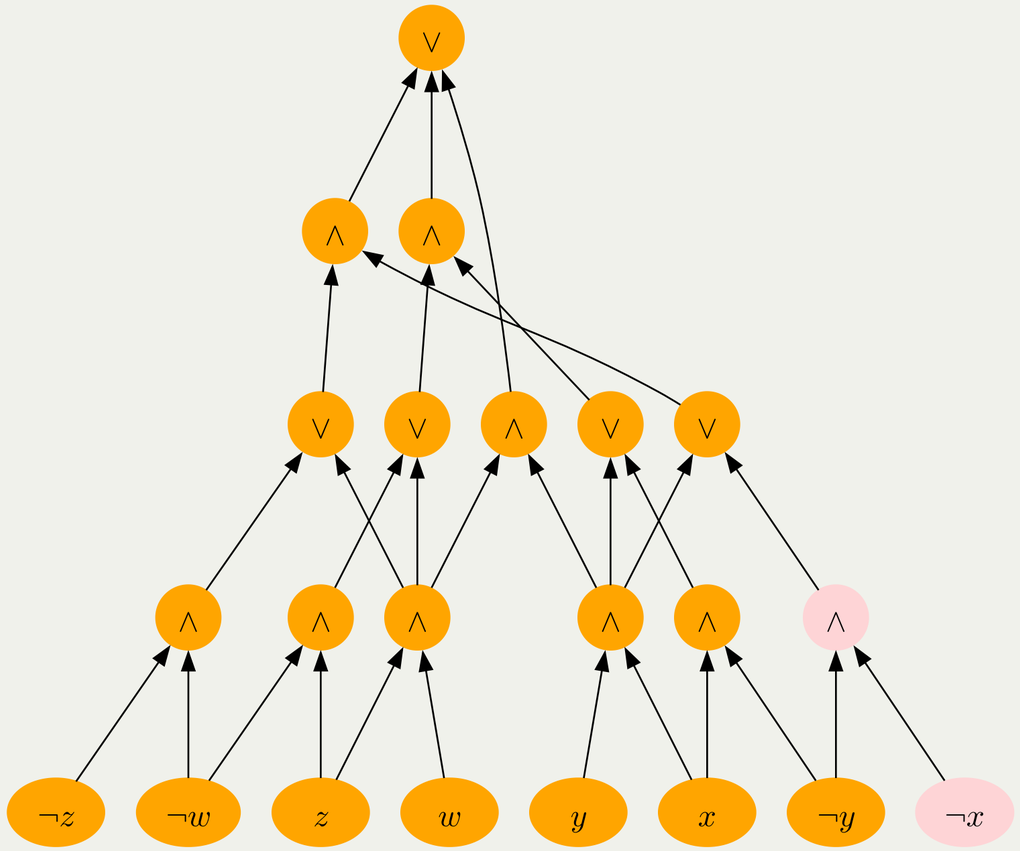
Finding all models of a DNNF



Conditioning on <*x*: 0,*y*: 1>. Not satisfiable: backtrack

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1

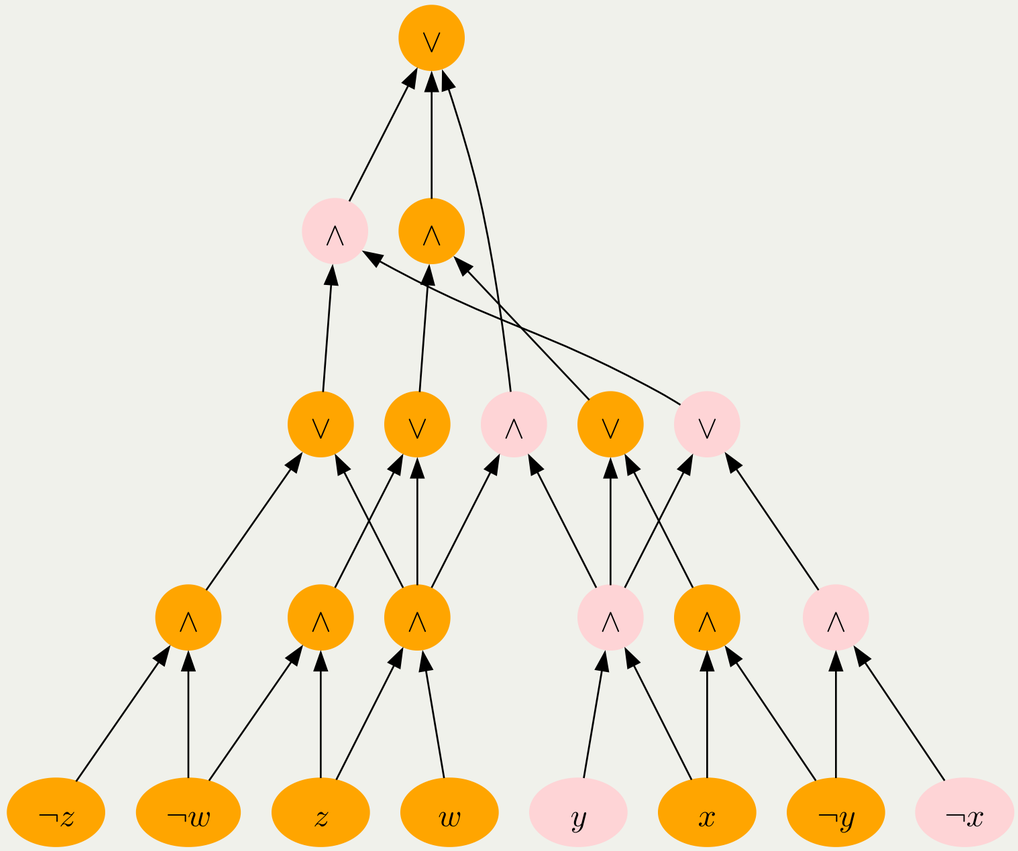
Finding all models of a DNNF



Conditioning on <x: 1>. Satisfiable. Keep going

x: 0,y: 0,z: 0,w: 0
x: 0,y: 0,z: 1,w: 1

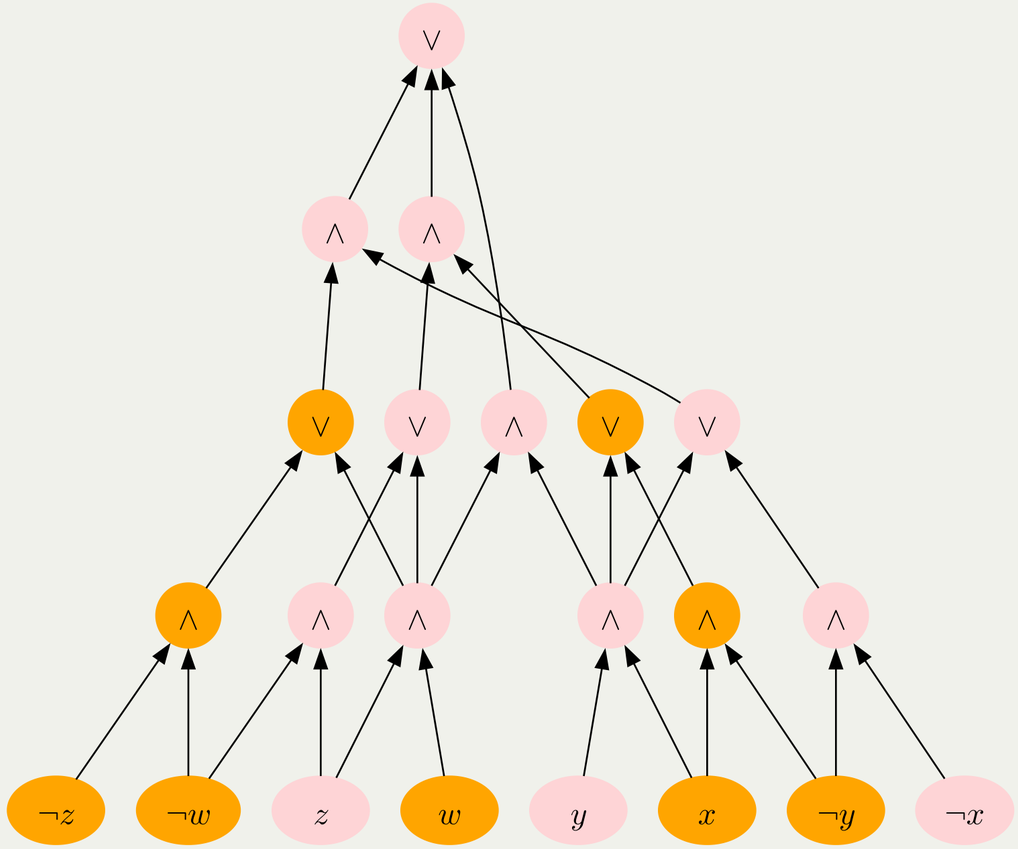
Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 0>. Satisfiable. Keep going

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1

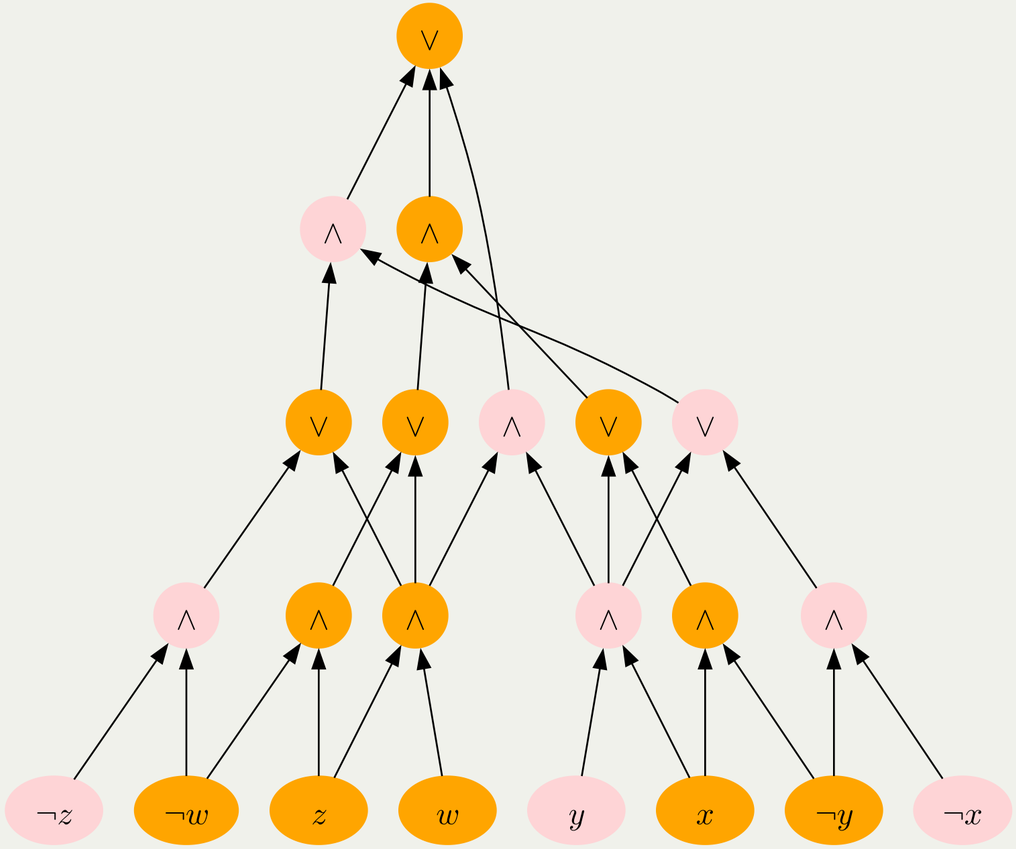
Finding all models of a DNNF



Conditioning on <x: 1,y: 0,z: 0>. Not satisfiable: backtrack

x: 0,y: 0,z: 0,w: 0
x: 0,y: 0,z: 1,w: 1

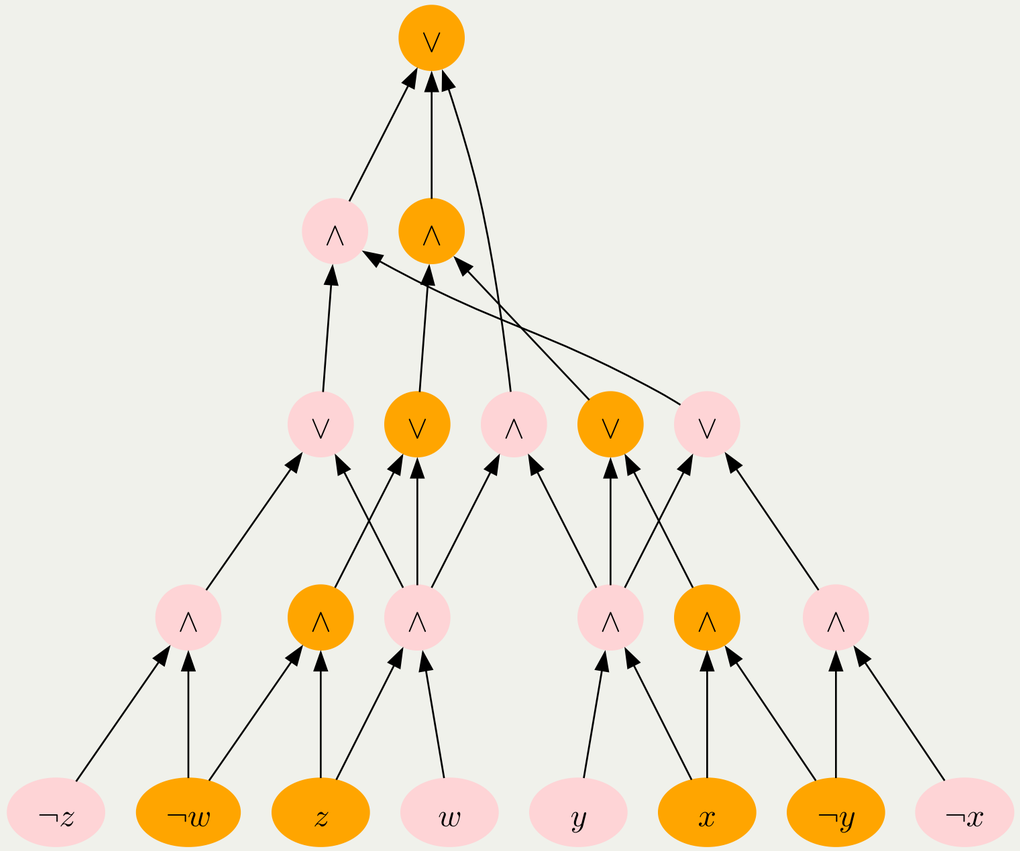
Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 0,*z*: 1>. Satisfiable. Keep going

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1

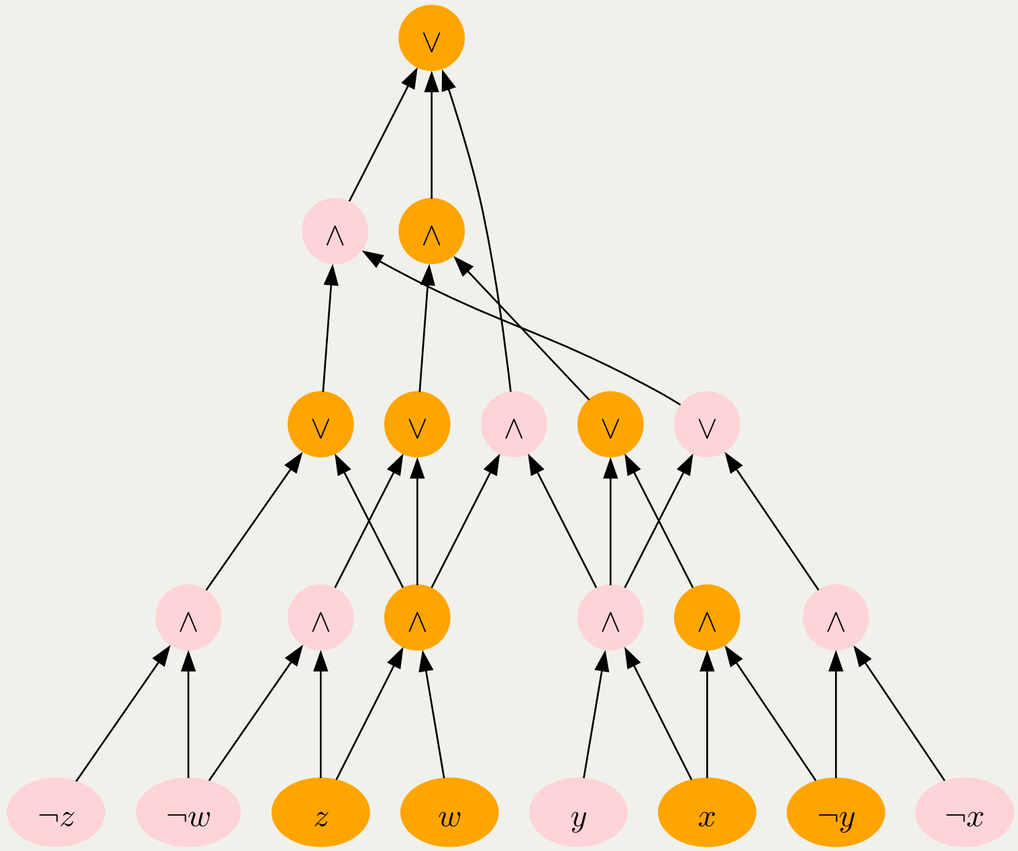
Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 0,*z*: 1,*w*: 0>. Model found.

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 0,*z*: 1,*w*: 0

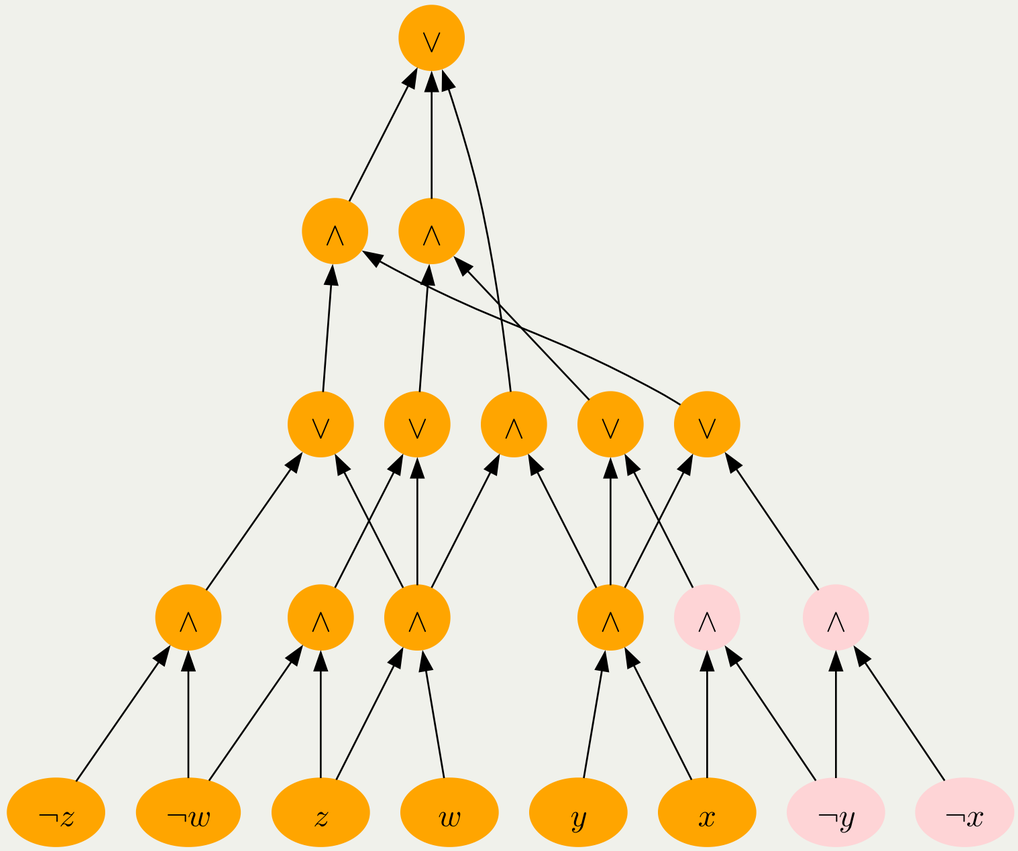
Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 0,*z*: 1,*w*: 1>. Model found.

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 0,*z*: 1,*w*: 0
x: 1,*y*: 0,*z*: 1,*w*: 1

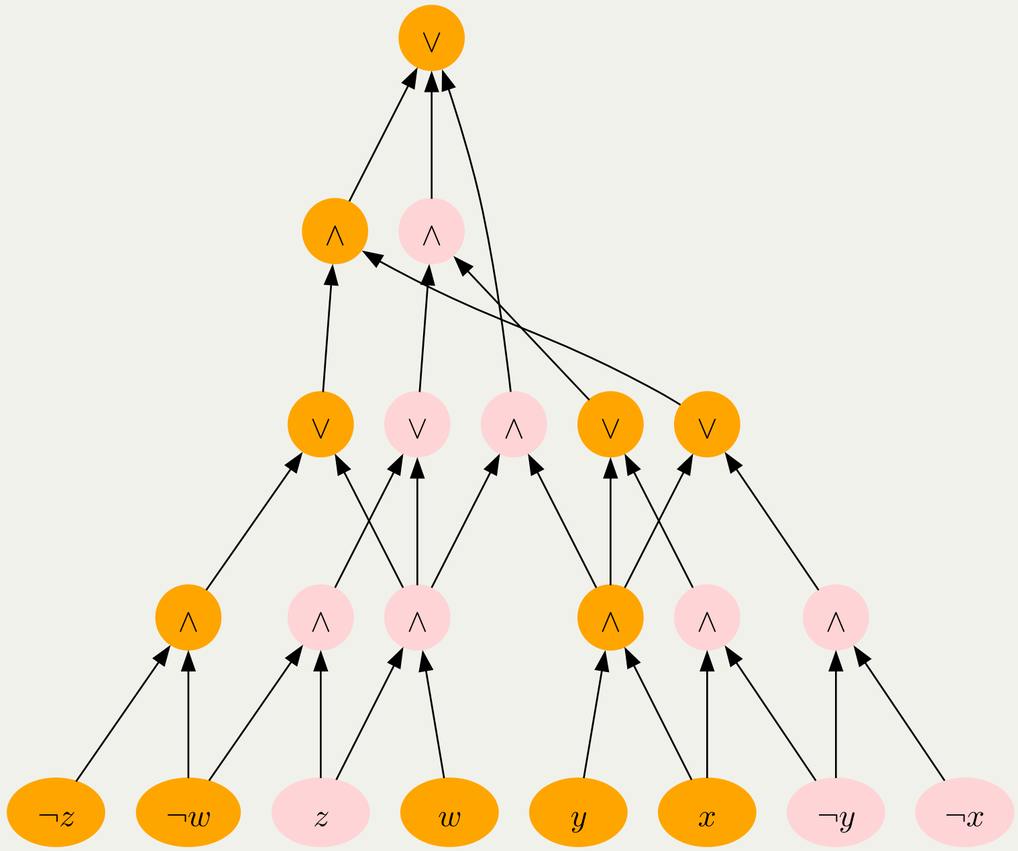
Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 1>. Satisfiable. Keep going

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 0,*z*: 1,*w*: 0
x: 1,*y*: 0,*z*: 1,*w*: 1

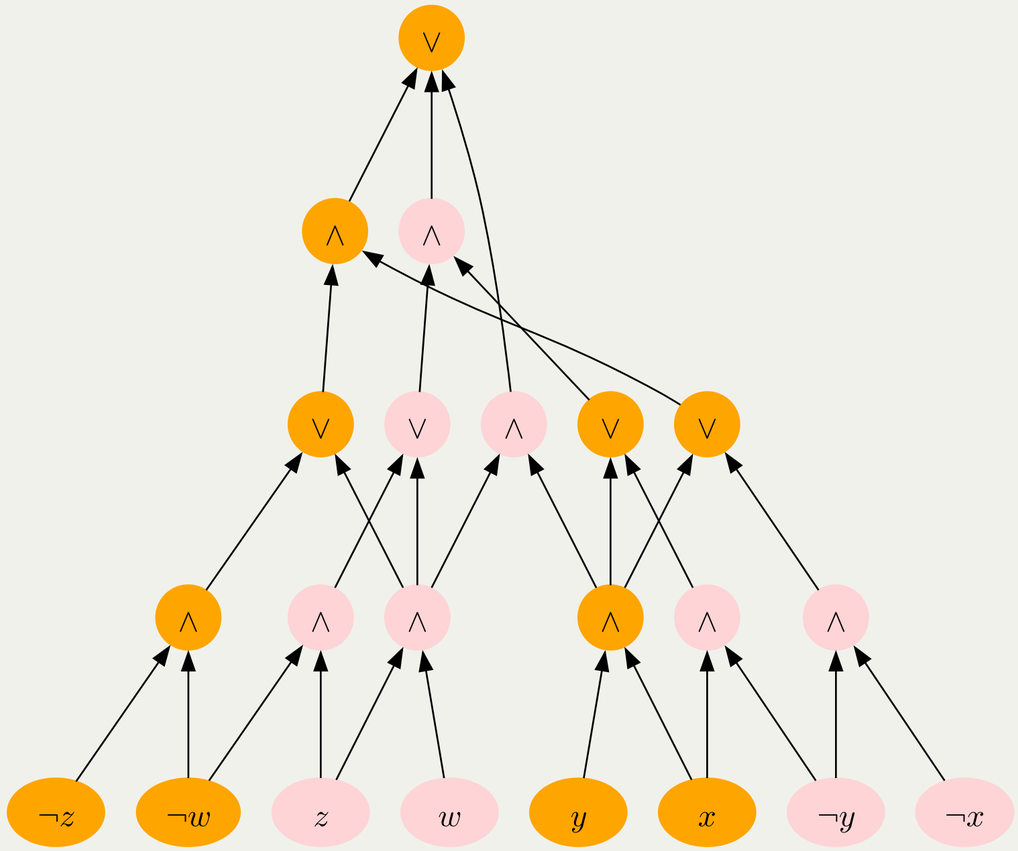
Finding all models of a DNNF



Conditioning on <x: 1,y: 1,z: 0>. Satisfiable. Keep going

x: 0,y: 0,z: 0,w: 0
x: 0,y: 0,z: 1,w: 1
x: 1,y: 0,z: 1,w: 0
x: 1,y: 0,z: 1,w: 1

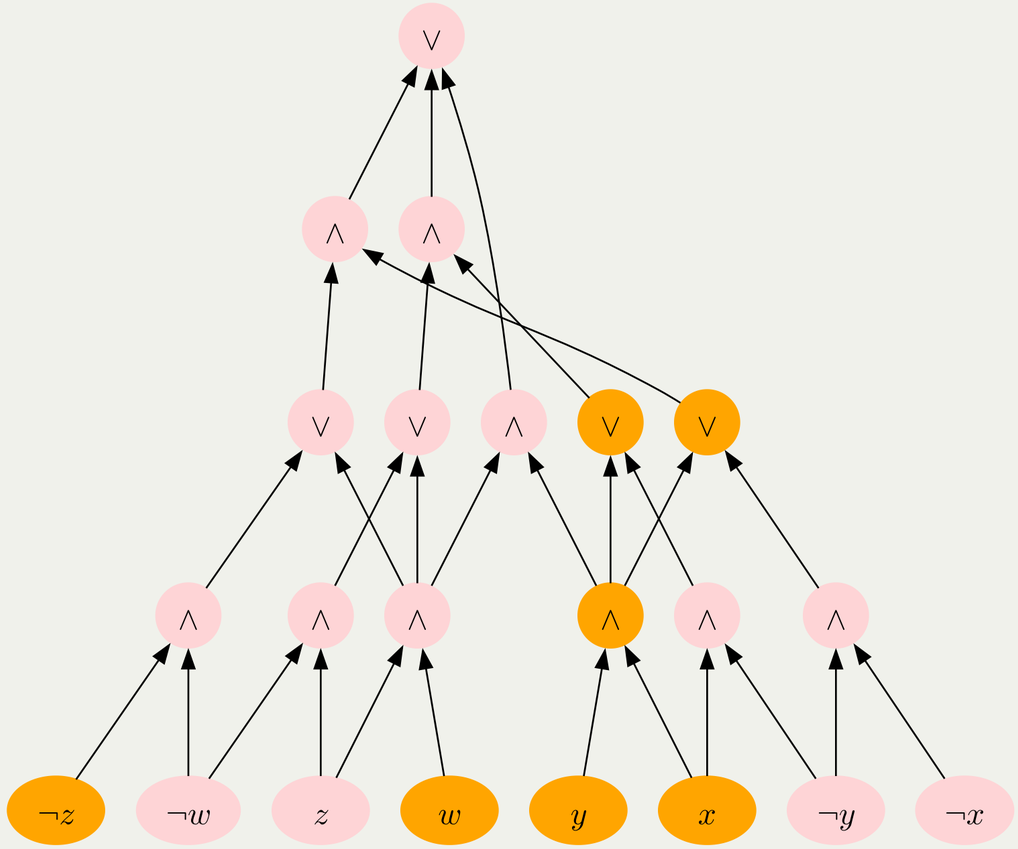
Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 1,*z*: 0,*w*: 0>. Model found.

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 0,*z*: 1,*w*: 0
x: 1,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 1,*z*: 0,*w*: 0

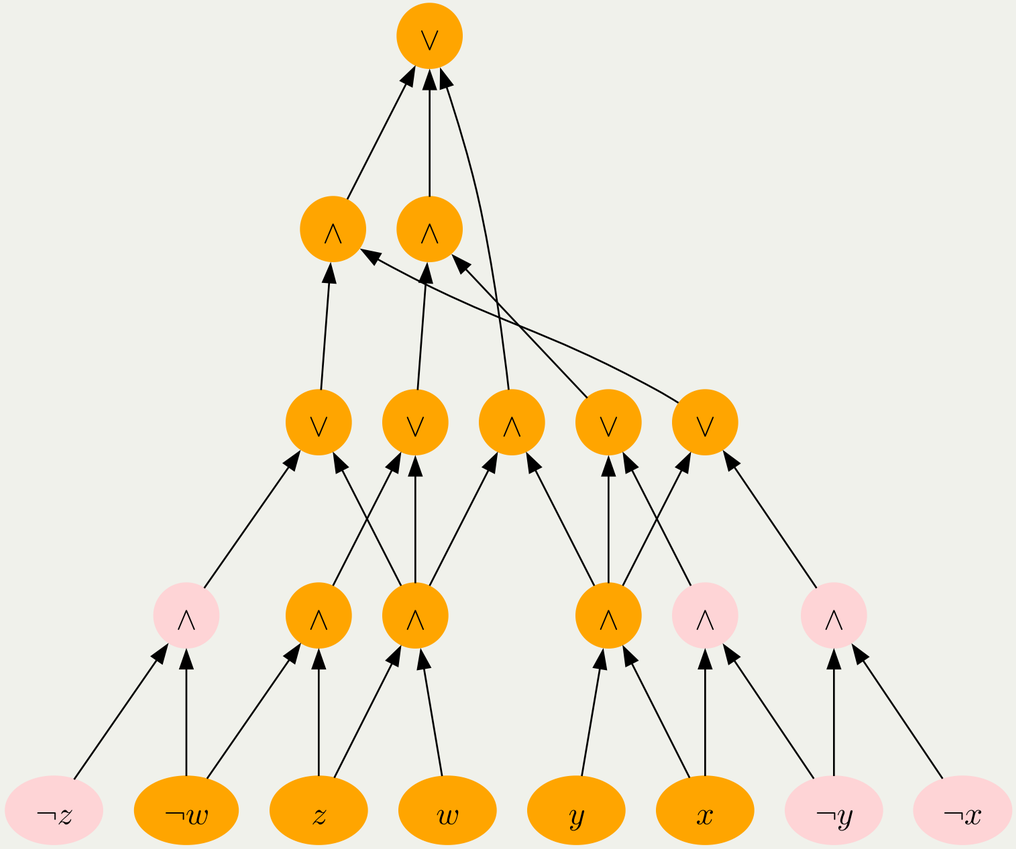
Finding all models of a DNNF



Conditioning on <x: 1,y: 1,z: 0,w: 1>. Not satisfiable: backtrack

x: 0,y: 0,z: 0,w: 0
x: 0,y: 0,z: 1,w: 1
x: 1,y: 0,z: 1,w: 0
x: 1,y: 0,z: 1,w: 1
x: 1,y: 1,z: 0,w: 0

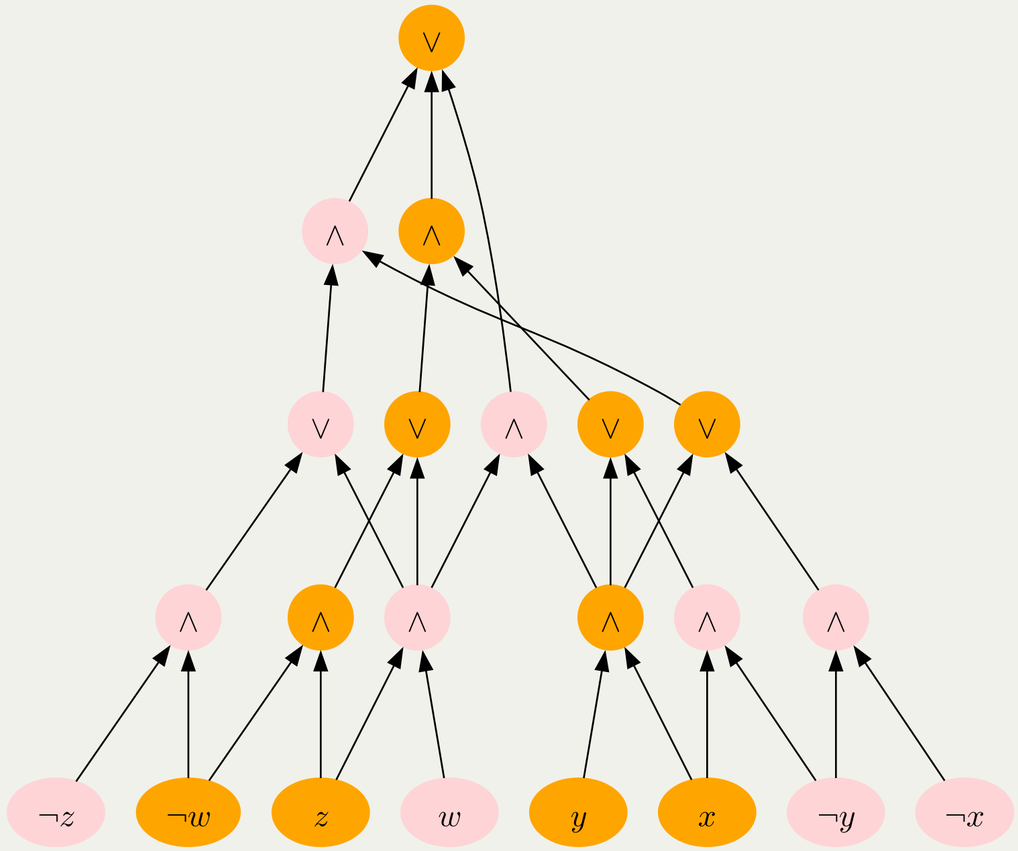
Finding all models of a DNNF



Conditioning on <x: 1,y: 1,z: 1>. Satisfiable. Keep going

x: 0,y: 0,z: 0,w: 0
x: 0,y: 0,z: 1,w: 1
x: 1,y: 0,z: 1,w: 0
x: 1,y: 0,z: 1,w: 1
x: 1,y: 1,z: 0,w: 0

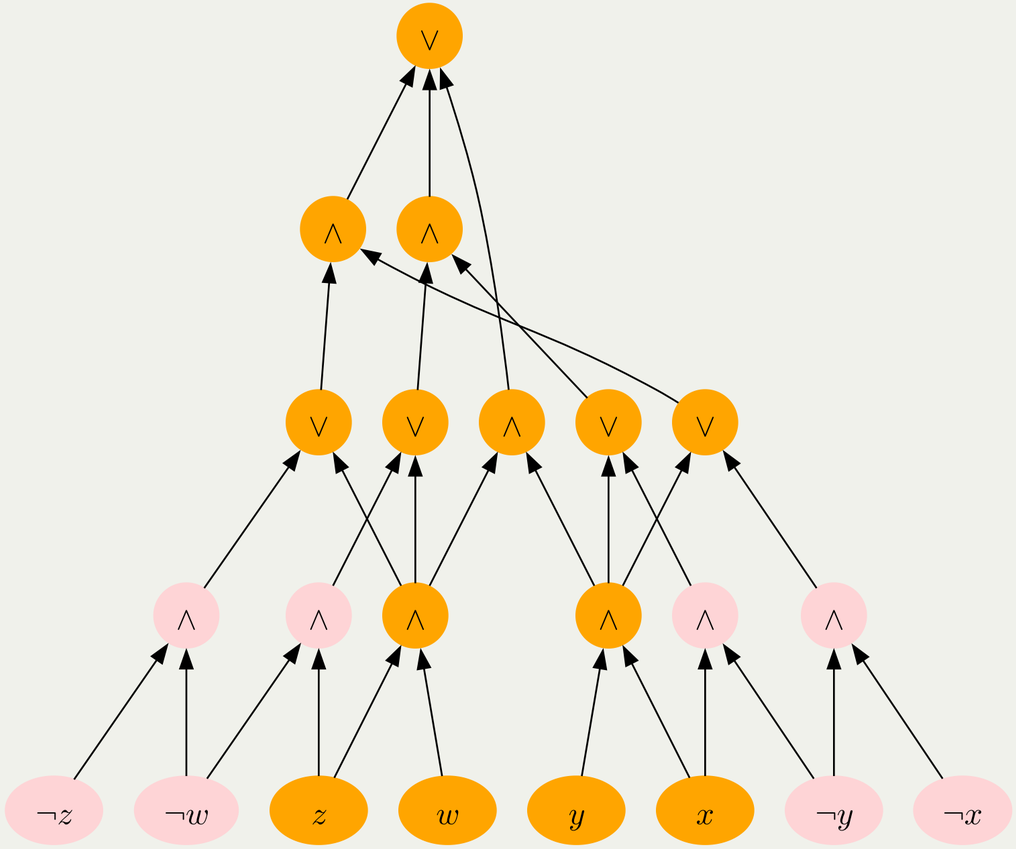
Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 1,*z*: 1,*w*: 0>. Model found.

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 0,*z*: 1,*w*: 0
x: 1,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 1,*z*: 0,*w*: 0
x: 1,*y*: 1,*z*: 1,*w*: 0

Finding all models of a DNNF



Conditioning on <*x*: 1,*y*: 1,*z*: 1,*w*: 1>. Model found.

x: 0,*y*: 0,*z*: 0,*w*: 0
x: 0,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 0,*z*: 1,*w*: 0
x: 1,*y*: 0,*z*: 1,*w*: 1
x: 1,*y*: 1,*z*: 0,*w*: 0
x: 1,*y*: 1,*z*: 1,*w*: 0
x: 1,*y*: 1,*z*: 1,*w*: 1

Model Counting on DNNF

Previous algorithm allows to count the models of C in time $O(\#var \cdot |C| \cdot \#C)$ but $\#C$ may be exponential in $|C|$.

Can we find the number of models of a given DNNF C in polynomial time in C?

- Not likely: #P-complete (as hard as #SAT): CNF ϕ has m models iff DNF $\neg\phi$ has $2^n - m$ models.
- Recently: there is an FPRAS (an approximation algorithm whp) [1]

[1] Kuldeep S. Meel, Alexis de Colnet, #CFG and #DNNF admit FPRAS. SODA 2026: 5978-6010.

Determinism (unambiguity)

- $\#(\alpha \wedge \beta) = \#\alpha \times \#\beta$ when α and β do not share variables (decomposable $\wedge$).
- $\#(\alpha \vee \beta) = \#\alpha + \#\beta - \#(\alpha \wedge \beta)$, when α and β are defined on the same variables.
  Hard to evaluate in a DNNF.

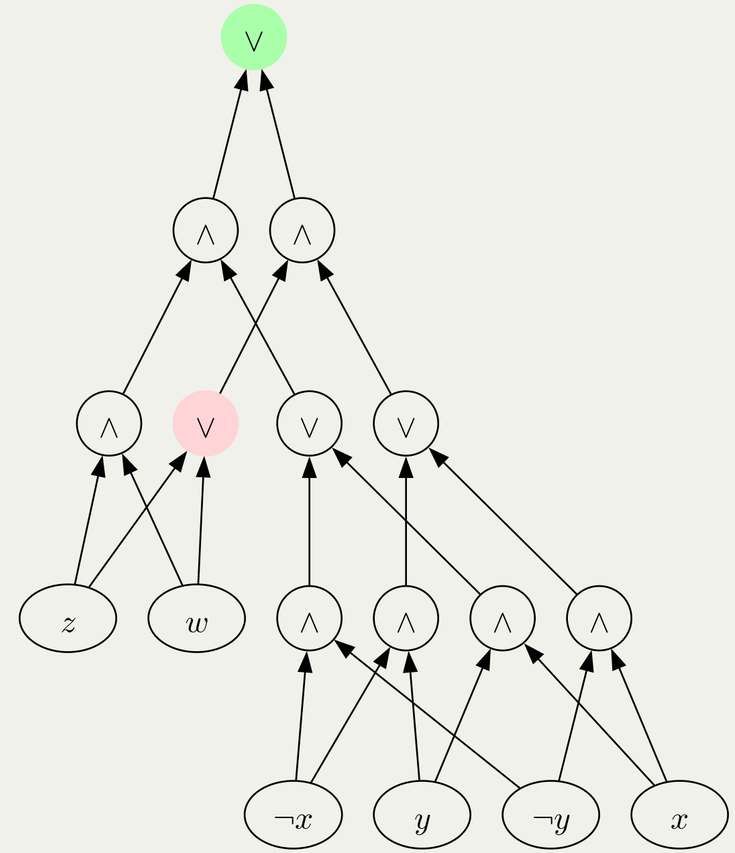
Determinism (unambiguity)

- $\#(\alpha \wedge \beta) = \#\alpha \times \#\beta$ when α and β do not share variables (decomposable $\wedge$).
- $\#(\alpha \vee \beta) = \#\alpha + \#\beta - \#(\alpha \wedge \beta)$, when α and β are defined on the same variables.
  Hard to evaluate in a DNNF.

A $\vee$-gate g is *deterministic* if $g_1 \wedge g_2 = \bot$ for inputs g_1, g_2 of g. In this case:

$$\#g = \#g_1 \cdot 2^{|\Delta_1|} + \#g_2 \cdot 2^{|\Delta_2|}$$

with $\Delta_1 = var(g_2) \backslash var(g_1)$ and $\Delta_2 = var(g_1) \backslash var(g_2)$.



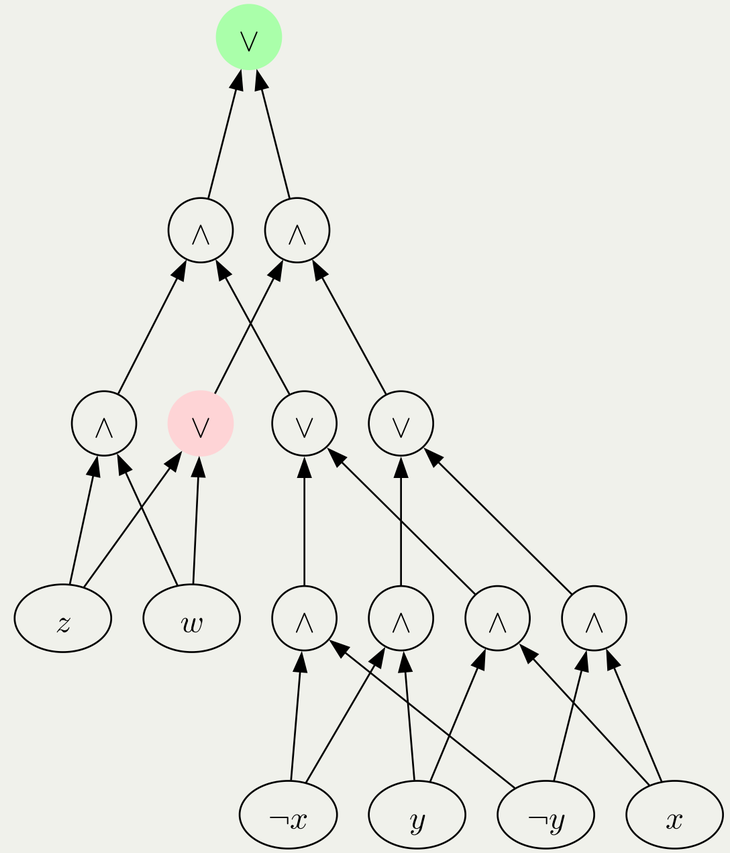
Determinism (unambiguity)

- $\#(\alpha \wedge \beta) = \#\alpha \times \#\beta$ when α and β do not share variables (decomposable $\wedge$).
- $\#(\alpha \vee \beta) = \#\alpha + \#\beta - \#(\alpha \wedge \beta)$, when α and β are defined on the same variables.
  Hard to evaluate in a DNNF.

A $\vee$-gate g is *deterministic* if $g_1 \wedge g_2 = \bot$ for inputs g_1, g_2 of g. In this case:

$$\#g = \#g_1 \cdot 2^{|\Delta_1|} + \#g_2 \cdot 2^{|\Delta_2|}$$

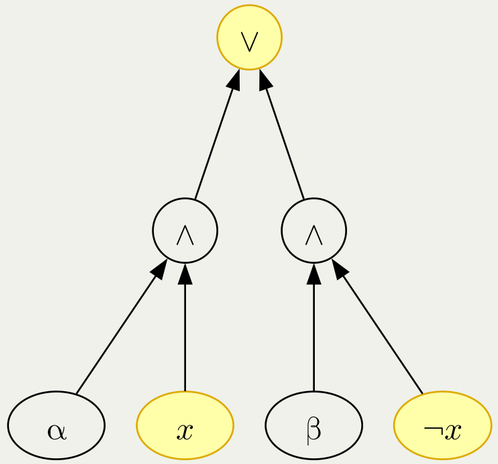
with $\Delta_1 = var(g_2) \backslash var(g_1)$ and $\Delta_2 = var(g_1) \backslash var(g_2)$.

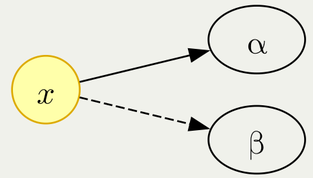


- **deterministic DNNF (d-DNNF)**: DNNF where every $\vee$-gate is deterministic.
- **Counting** is tractable on d-DNNF: intuitively, replace $\vee$ by $+$, $\wedge$ by $\times$ and inputs by 1 (more on this later).

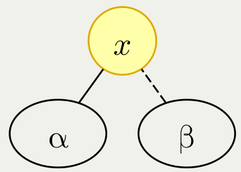
Decision-gates: a Syntactic Determinism

- **coNP-hard** to decide whether a given $\vee$-gate is deterministic.
- Determinism is often *guaranteed* by the algorithm producing the circuit.
- *Decision-gates* is a common way of ensuring it:



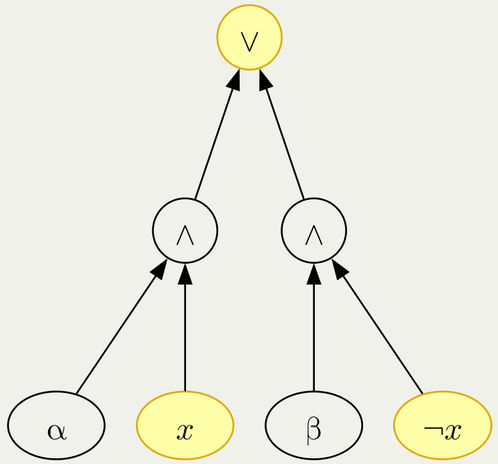


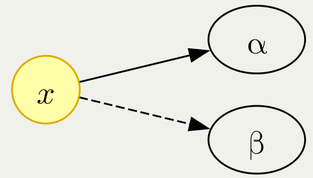
decDNNF: every $\vee$-gate is a *decision-gate*.



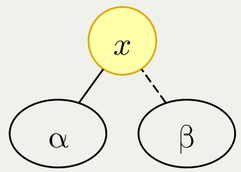
Decision-gates: a Syntactic Determinism

- **coNP-hard** to decide whether a given $\vee$-gate is deterministic.
- Determinism is often *guaranteed* by the algorithm producing the circuit.
- *Decision-gates* is a common way of ensuring it:





decDNNF: every $\vee$-gate is a *decision-gate*.



Observation: DNNF $<_s$ d-DNNF $<_s$ decDNNF (separation in the last part).

Weighted Model Counting

- f a Boolean function over X
- w a weight function over $X \times \{0, 1\}$

$$w(f) = \sum_{\tau \vDash f} \prod_{x \in X} w(x, \tau(x))$$

Example: $f = x \vee y$

- $w(x, 0) = 1$
- $w(x, 1) = 2$
- $w(y, 0) = 2$
- $w(y, 1) = 1$

x	y	w
1	0	$2 \times 2 = 4$
0	1	$1 \times 1 = 1$
1	1	$2 \times 1 = 2$
$w(f) =$		$4 + 1 + 2 = 7$

Application:

- if $w(x, b) = 1$ for every x, b, $w(f)$ is $\#f$.
- if $w(x, b)$ is the probability of x to have value b, $w(f)$ is the probability of f to be satisfied.

Optimising

Weighted model counting:

$$w(f) = \sum_{\tau \vDash f} \prod_{x \in X} w(x, \tau(x))$$

Related optimisation problems:

$$w_{max}(f) = \max\{ \prod_{x \in X} w(x, \tau(x)) \mid \tau \vDash f\}$$

$$w_m(f) = \min\{ \sum_{x \in X} w(x, \tau(x)) \mid \tau \vDash f\}$$

- If w_{max} is a probability: $w(f)$ is the most probable model.
- If w_m is a price to pay to set variables, $w_m(f)$ is the cheapest model.

What is the relation between w, w_{max}, w_m etc.?

Counting over semiring

Commutative semiring $\mathbb{K} = (K, \oplus, \otimes, 0_{\mathbb{K}}, 1_{\mathbb{K}})$:

- $\oplus, \otimes$ commutative associative operators on K with $0_{\mathbb{K}}$ neutral for $\oplus$ and $1_{\mathbb{K}}$ neutral for $\otimes$.
- $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$
- $0_{\mathbb{K}} \otimes a = 0_{\mathbb{K}}$.

Example: $(\mathbb{Q}, \min, +, +\infty, 0)$, $([0,1], \max, \times, 0, 1)$, $(2^X, \cup, \cap, \emptyset, X)$… (check Nina Pardal lecture on Sunday).

[1] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. Journal of Applied Logic, 22:46–62, 2017.

Counting over semiring

Commutative semiring $\mathbb{K} = (K, \oplus, \otimes, 0_{\mathbb{K}}, 1_{\mathbb{K}})$:

- $\oplus, \otimes$ commutative associative operators on K with $0_{\mathbb{K}}$ neutral for $\oplus$ and $1_{\mathbb{K}}$ neutral for $\otimes$.
- $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$
- $0_{\mathbb{K}} \otimes a = 0_{\mathbb{K}}$.

Example: $(\mathbb{Q}, \min, +, +\infty, 0)$, $([0,1], \max, \times, 0, 1)$, $(2^X, \cup, \cap, \emptyset, X)$… (check Nina Pardal lecture on Sunday).

Algebraic Model Counting [1] for w: $X \times \{0,1\} \to \mathbb{K}$ and f on variables X:

$$w(f) = \bigoplus_{\tau \vDash f} \bigoplus_{x \in X} w(x, \tau(x))$$

[1] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. Journal of Applied Logic, 22:46–62, 2017.

Counting over semiring

Commutative semiring $\mathbb{K} = (K, \oplus, \otimes, 0_{\mathbb{K}}, 1_{\mathbb{K}})$:

- $\oplus, \otimes$ commutative associative operators on K with $0_{\mathbb{K}}$ neutral for $\oplus$ and $1_{\mathbb{K}}$ neutral for $\otimes$.
- $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$
- $0_{\mathbb{K}} \otimes a = 0_{\mathbb{K}}$.

Example: $(\mathbb{Q}, \min, +, +\infty, 0)$, $([0,1], \max, \times, 0, 1)$, $(2^X, \cup, \cap, \emptyset, X)$… (check Nina Pardal lecture on Sunday).

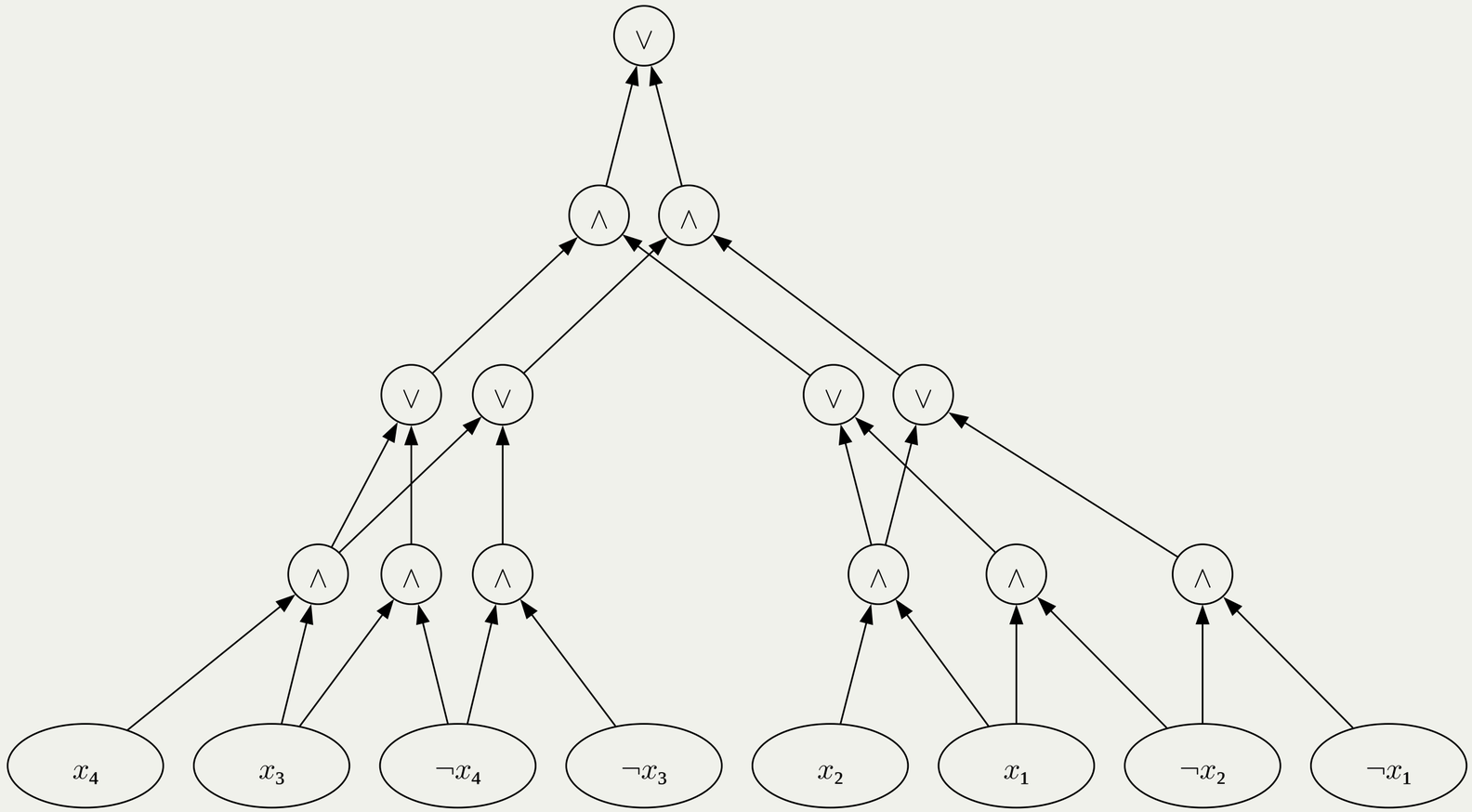
Algebraic Model Counting [1] for w: $X \times \{0,1\} \to \mathbb{K}$ and f on variables X:

$$w(f) = \bigoplus_{\tau \vDash f} \bigotimes_{x \in X} w(x, \tau(x))$$

By changing the semiring, we unify counting and optimisation problems.

[1] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. Journal of Applied Logic, 22:46–62, 2017.

Evaluation Algorithm

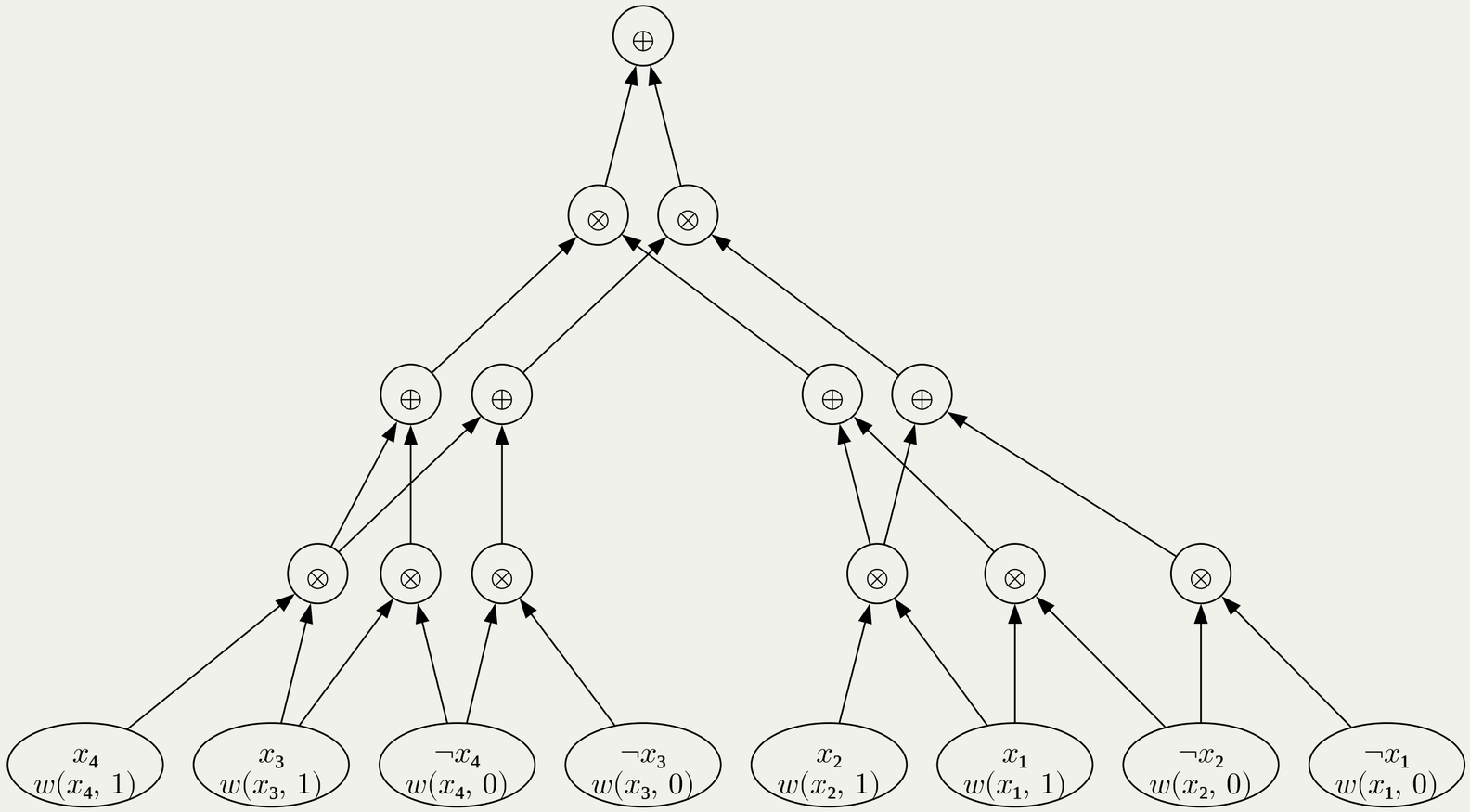


Theorem

If C is a *smooth d-DNNF*, it computes $w(f)$ by making $O(|C|)$ arithmetic operations.

Smooth: for $g_1 \vee g_2$, we must have $var(g_1) = var(g_2)$. Details on board.

Evaluation Algorithm



Theorem

If C is a *smooth d-DNNF*, it computes $w(f)$ by making $O(|C|)$ arithmetic operations.

Smooth: for $g_1 \vee g_2$, we must have $var(g_1) = var(g_2)$. Details on board.

Without determinism

Correctness depends on determinism but for some semiring, we can drop it if $\mathbb{K}$ is *idempotent*:

$$\forall a, a \oplus a = a.$$

Typically, $(min, +)$, $(max, \times)$ etc. are idempotent: many optimisation problems are tractable on DNNF directly.

Wrap up

- DNNF, d-DNNF, decDNNF: interesting factorisation properties.
- Many tractable queries even for DNNF.
- Strong link with counting.

Next part: how to build such circuits.

Part II

Building Circuits

Building circuits top-down

Original representation

We assume the input is always given as a CNF formula: $\phi = \bigwedge_{c\phi} \bigvee_{\ell \in c} \ell$.

1. Standard input for SAT solver.
2. Every Boolean circuit can be encoded into a CNF formula with isomorphic models (Tseitin transform).
3. Rich literature for encoding involved constraints.
4. Simple form.

Goal: transform a given CNF formula into a d-DNNF.

Convention

To lighten the notations, CNF formulas are denoted as follows:

- literal x_i denoted by i
- literal $\neg x_i$ denoted by $-i$
- clause: list of literals, formula: list of clauses

Example: $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_4)$ is denoted by

- $1\ -2$
- $1\ 3\ -4$
- $2\ -4$

Stupid Brute Force

-4 1 3
1 4
-1 2 4
-3 -2 1 4

Stupid Brute Force

-4 1 3
1 4
-1 2 4
-3 -2 1 4

Stupid Brute Force

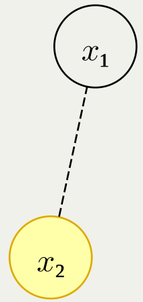
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



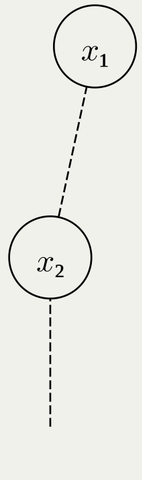
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



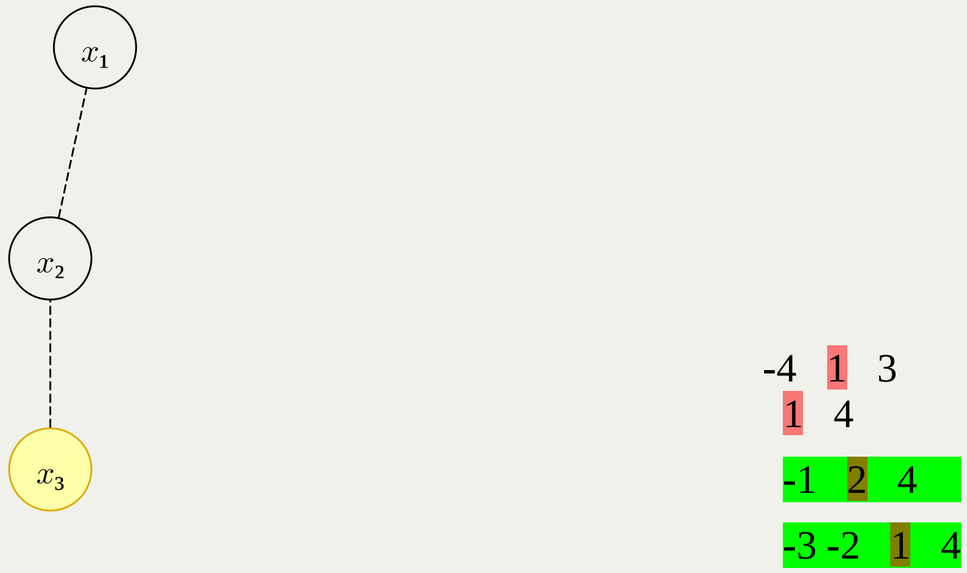
-4 1 3

1 4

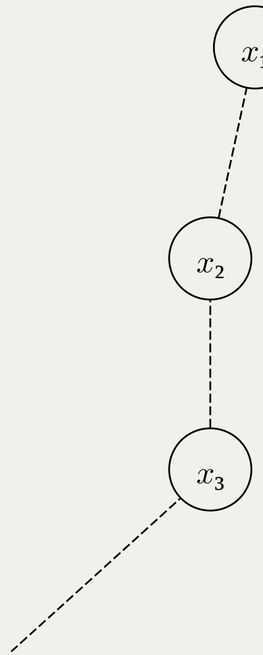
-1 2 4

-3 -2 1 4

Stupid Brute Force



Stupid Brute Force



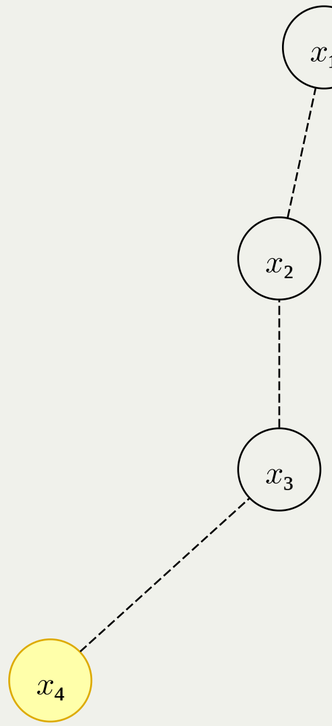
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



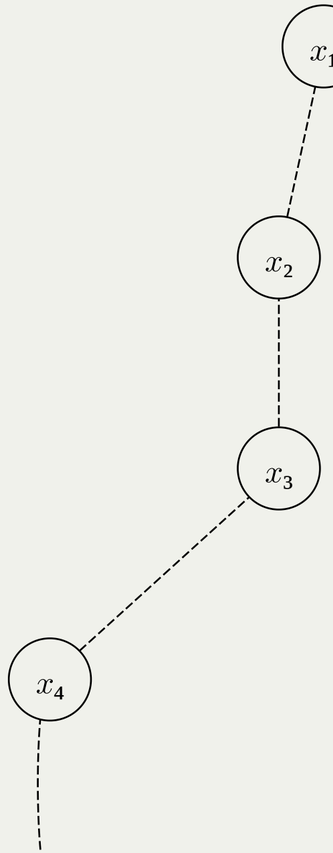
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



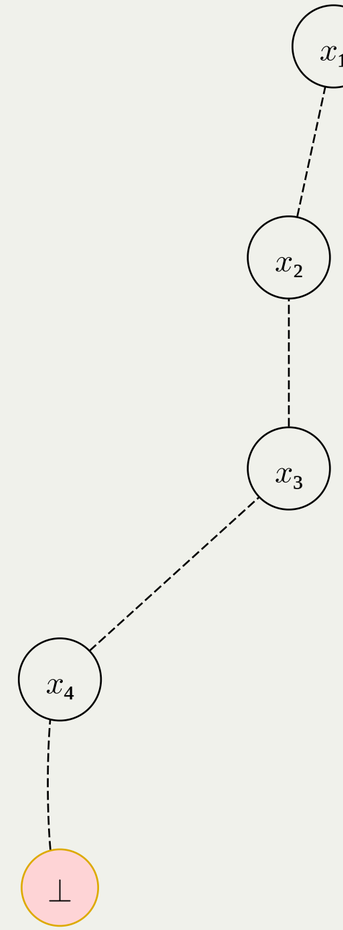
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



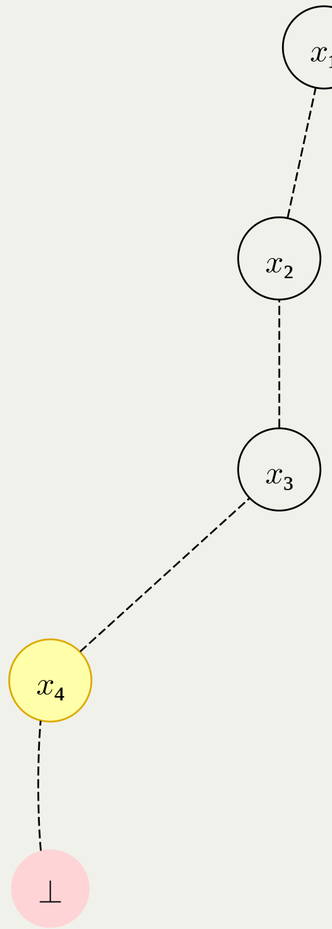
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



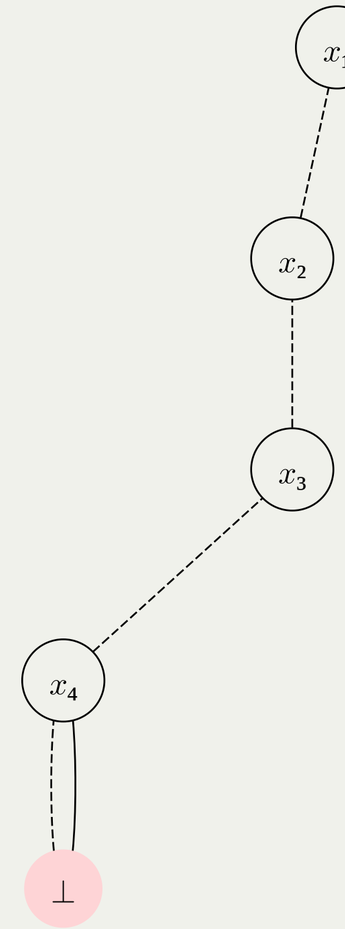
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



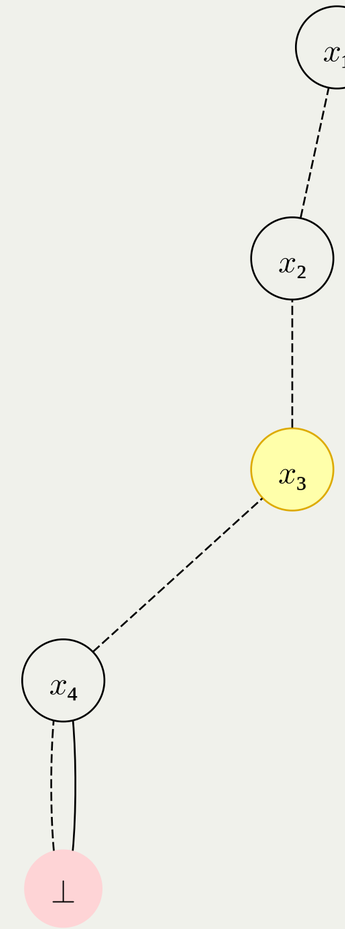
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



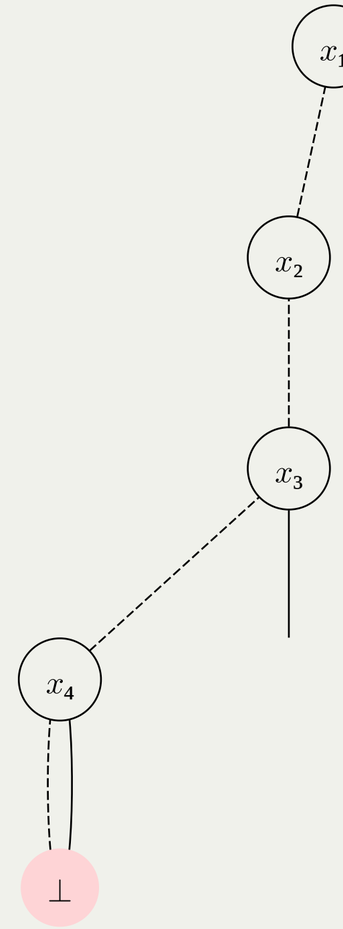
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



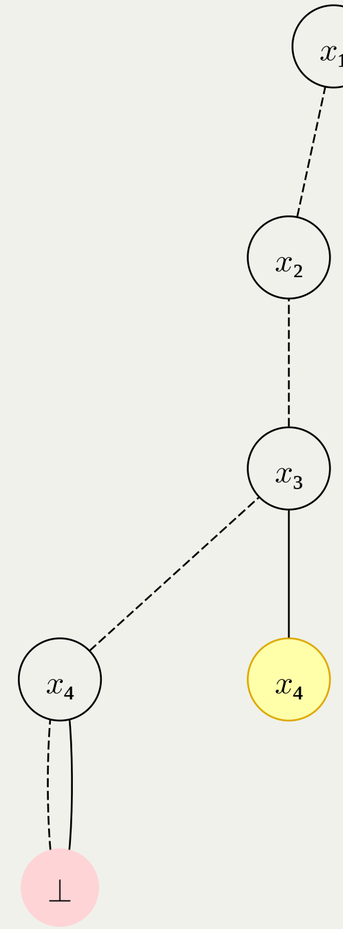
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



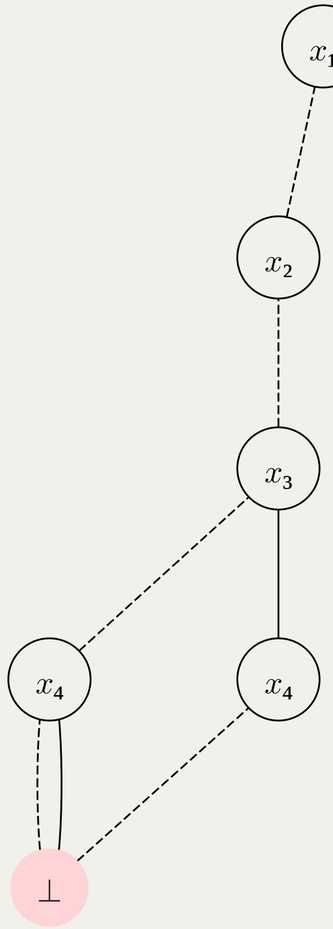
-4 1 3

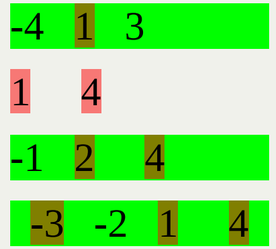
1 4

-1 2 4

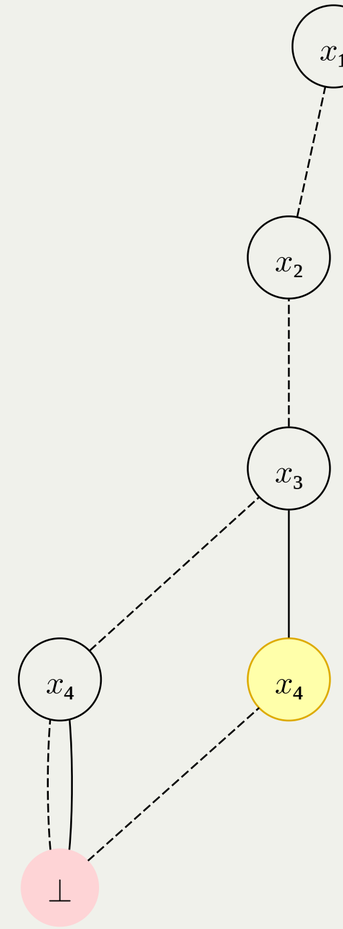
-3 -2 1 4

Stupid Brute Force





Stupid Brute Force



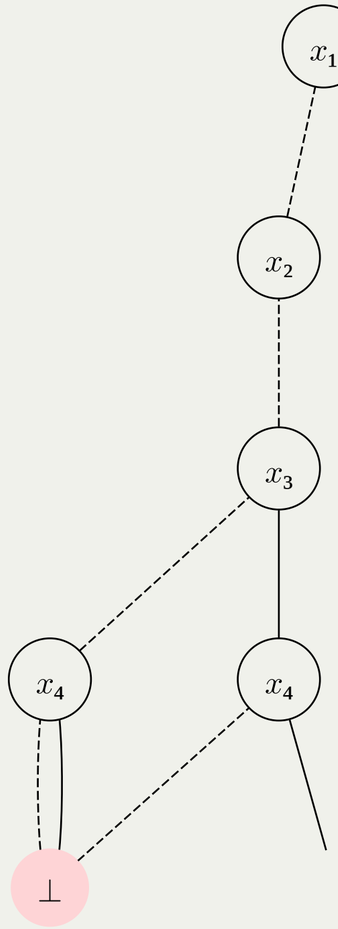
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



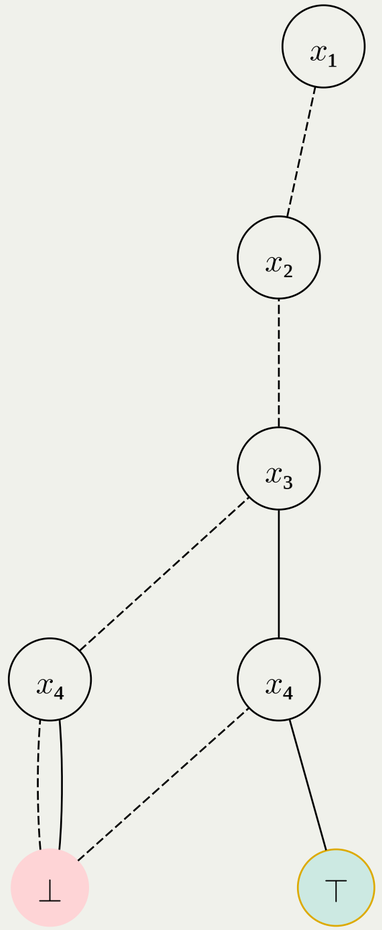
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



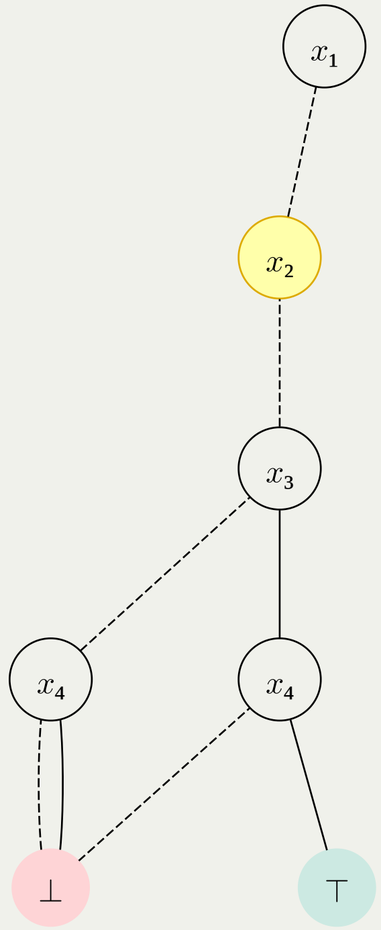
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



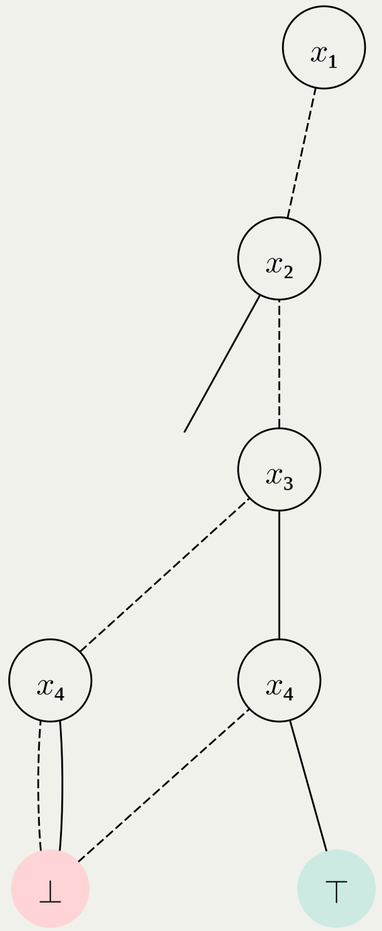
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



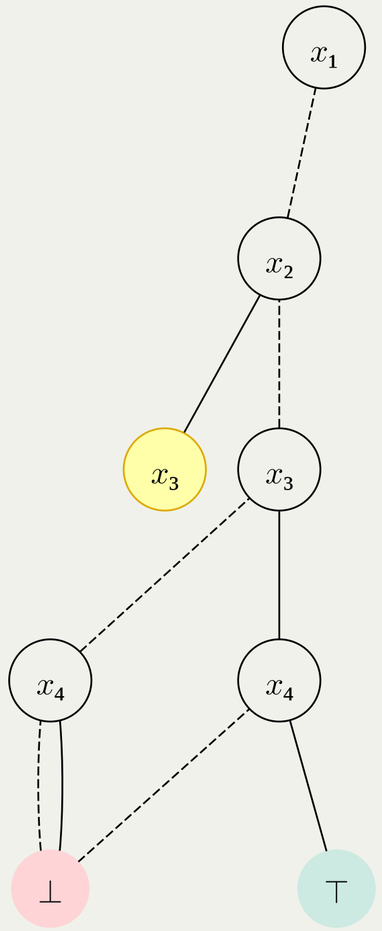
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



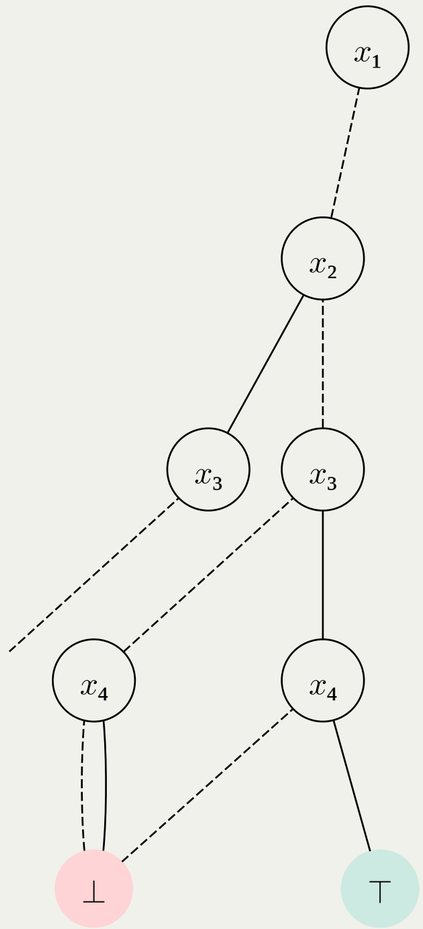
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



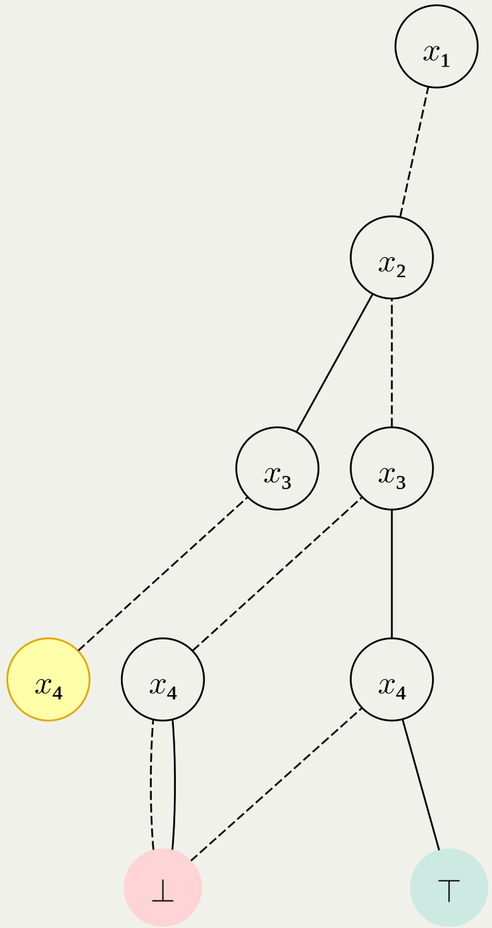
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



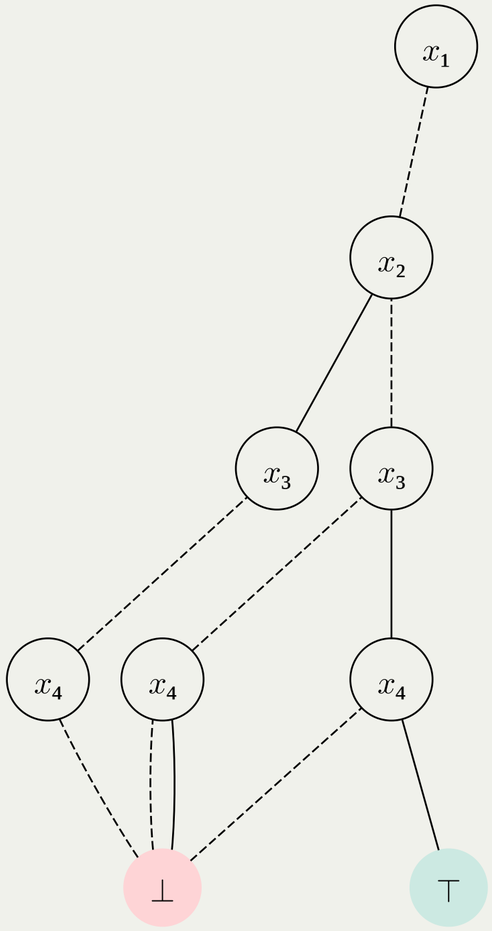
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



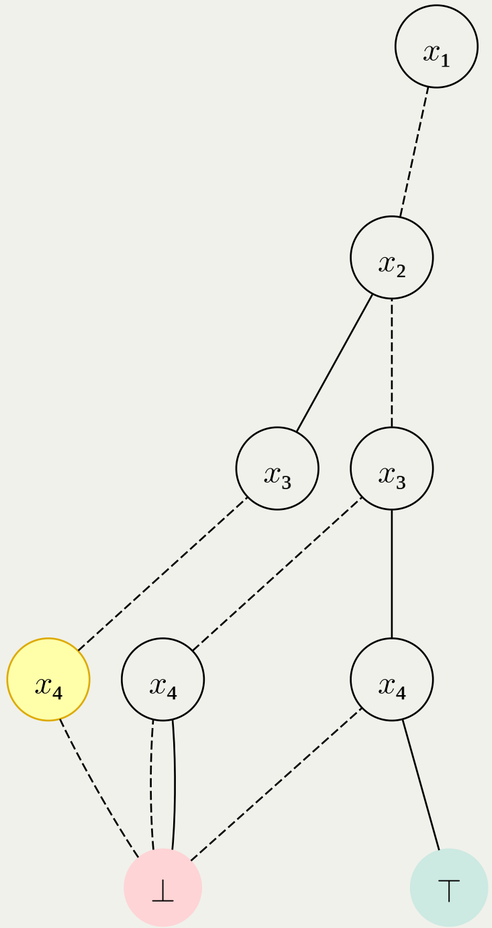
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



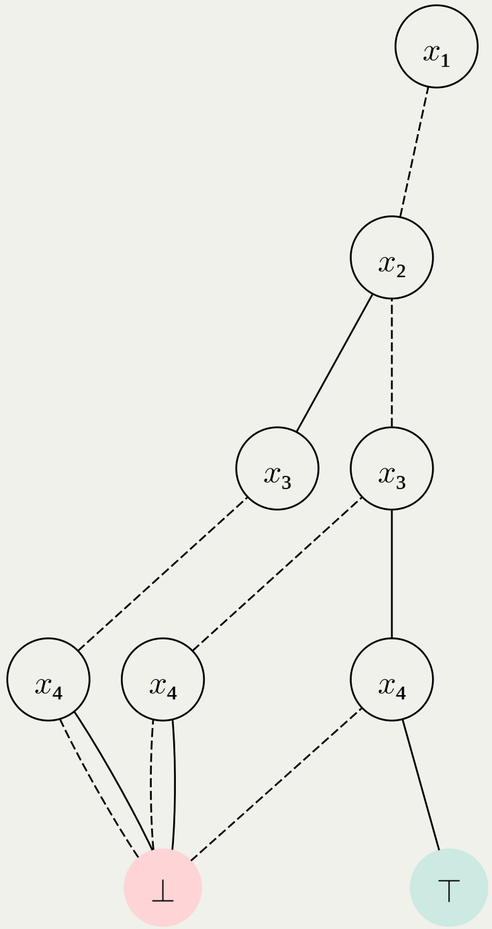
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



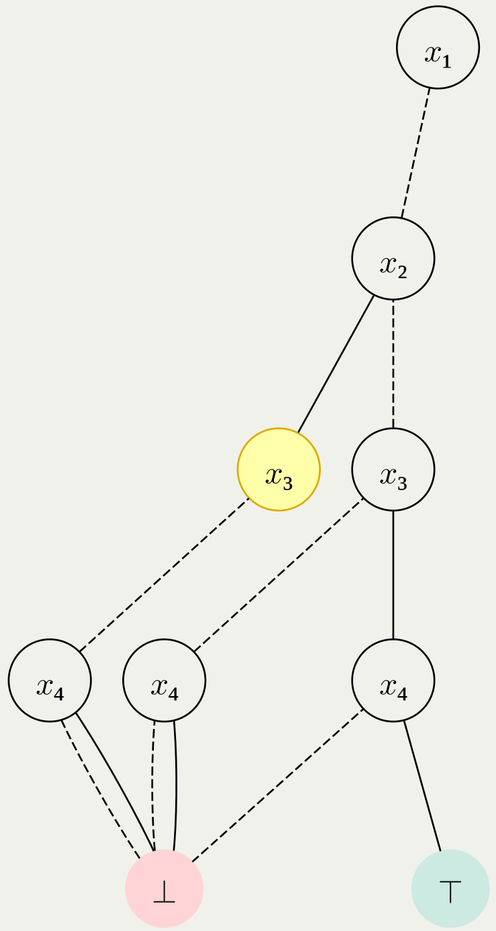
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



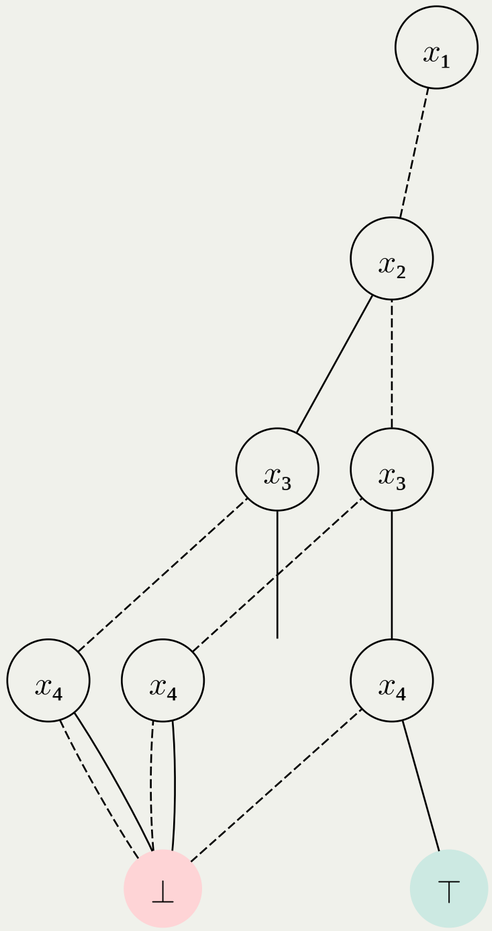
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



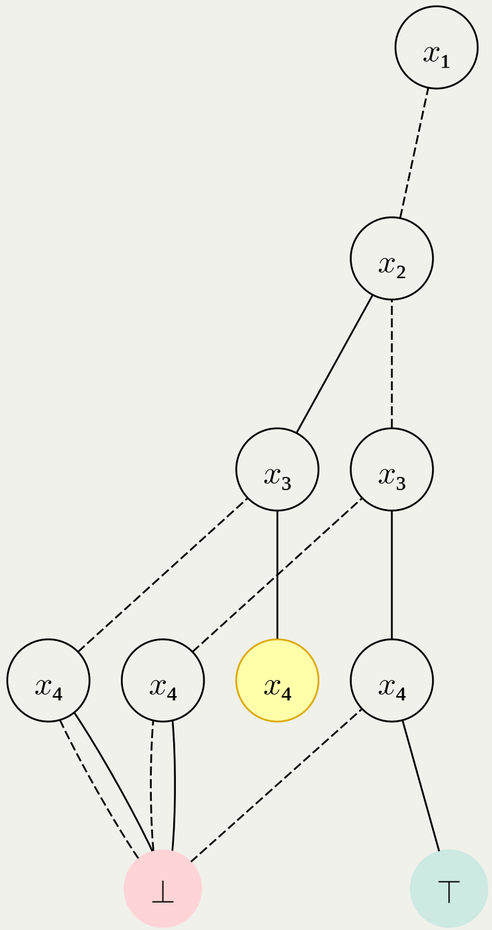
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



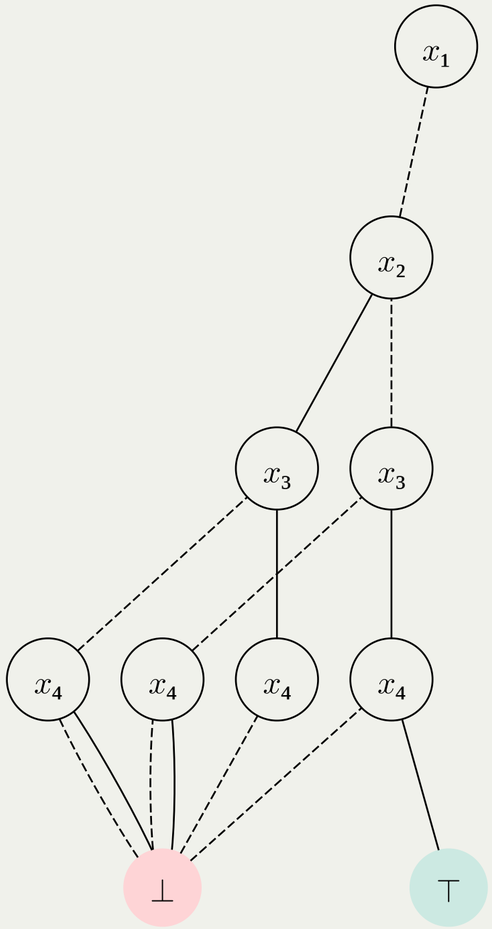
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



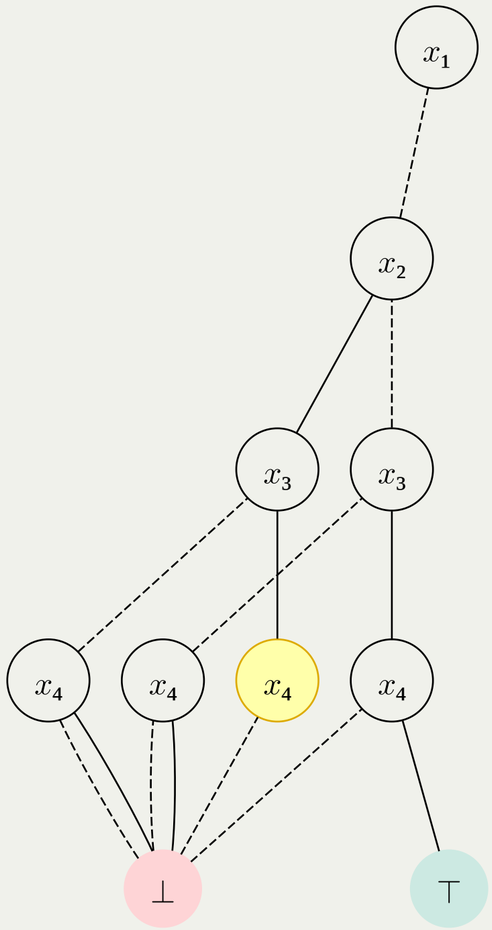
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



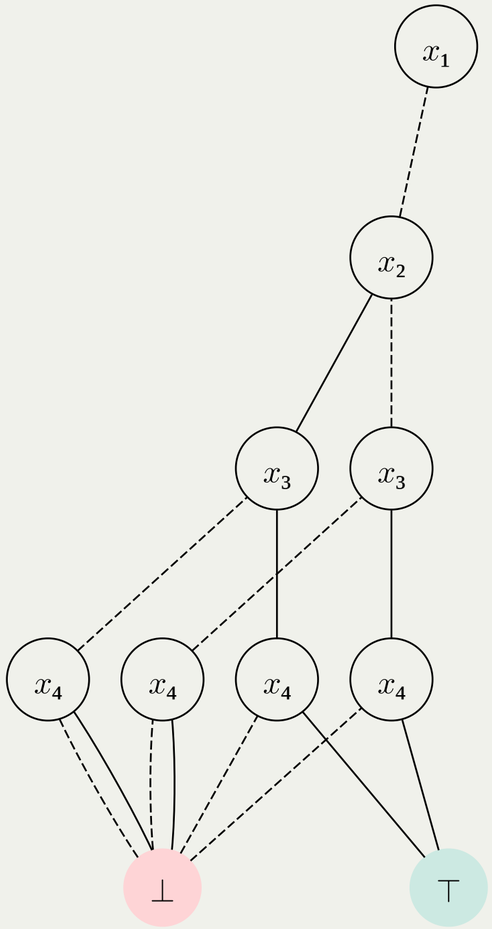
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



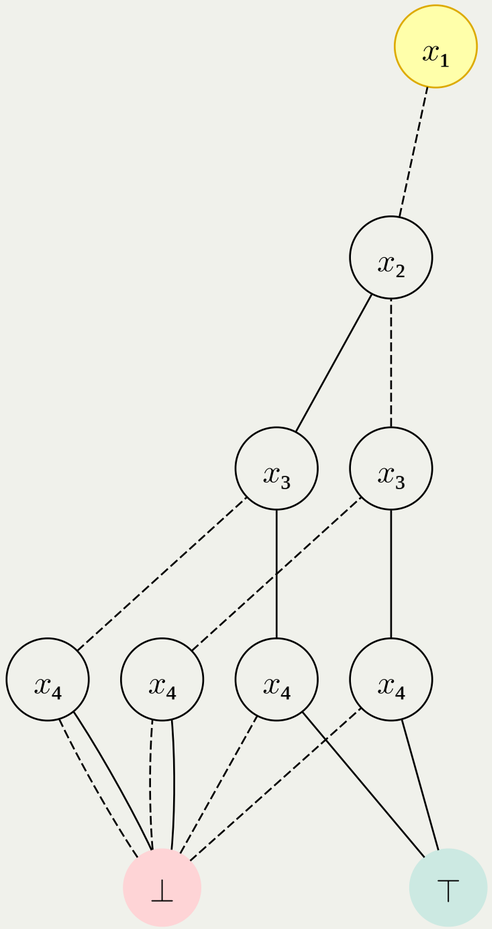
-4 1 3

1 4

-1 2 4

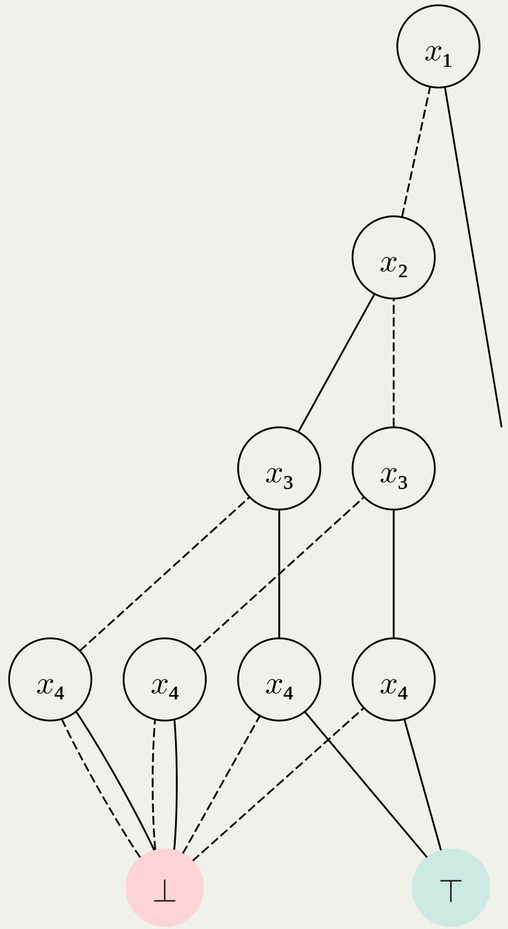
-3 -2 1 4

Stupid Brute Force



-4 1 3
1 4
-1 2 4
-3 -2 1 4

Stupid Brute Force



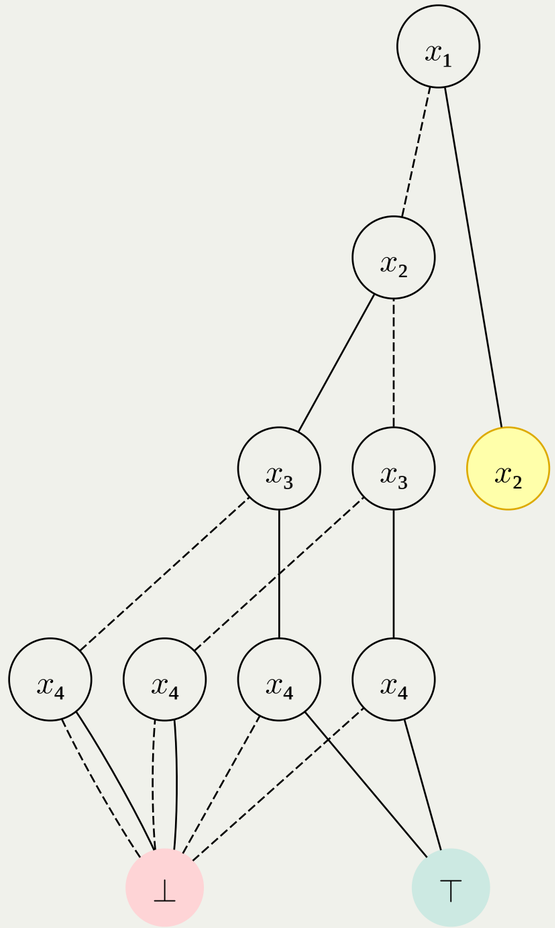
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



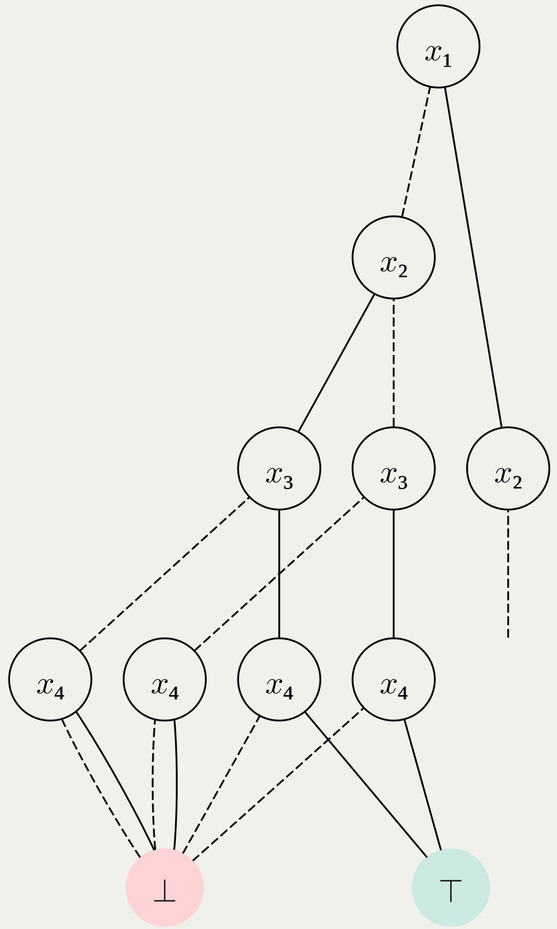
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



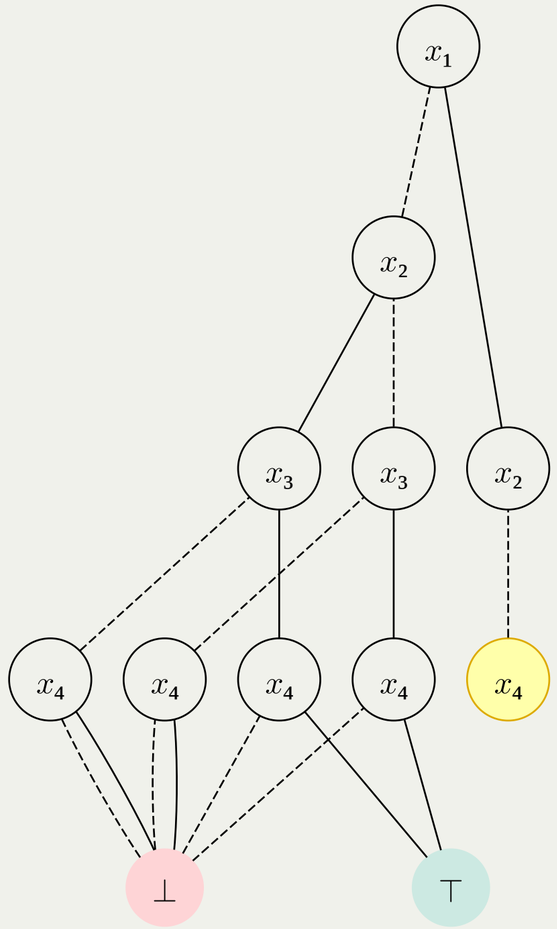
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



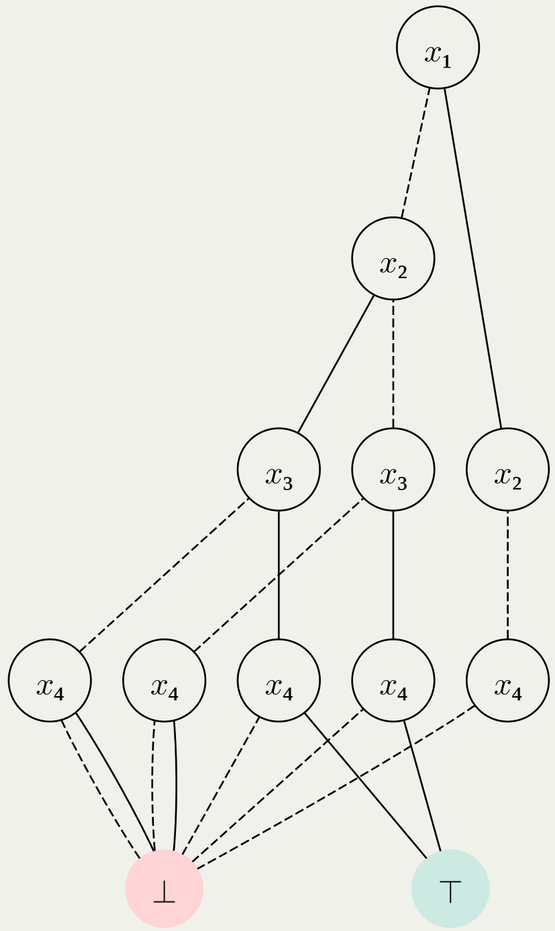
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



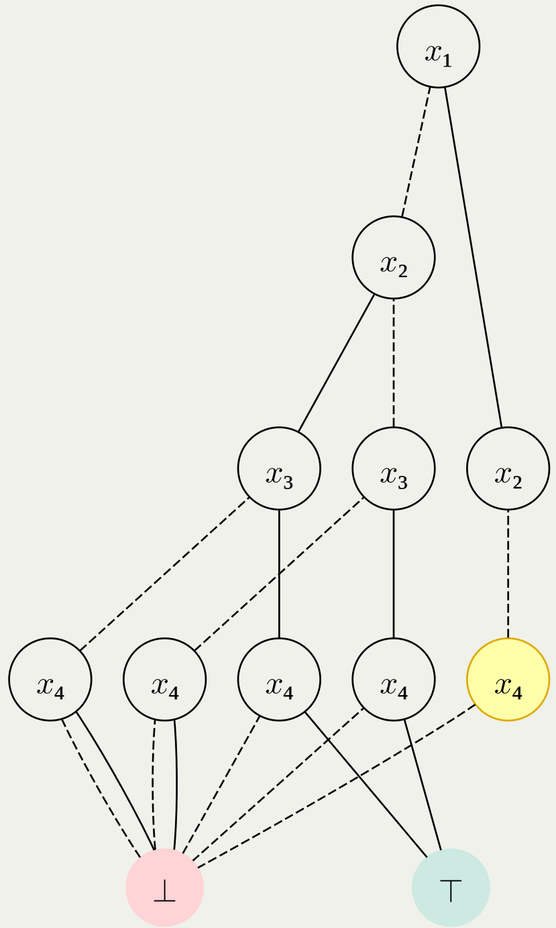
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



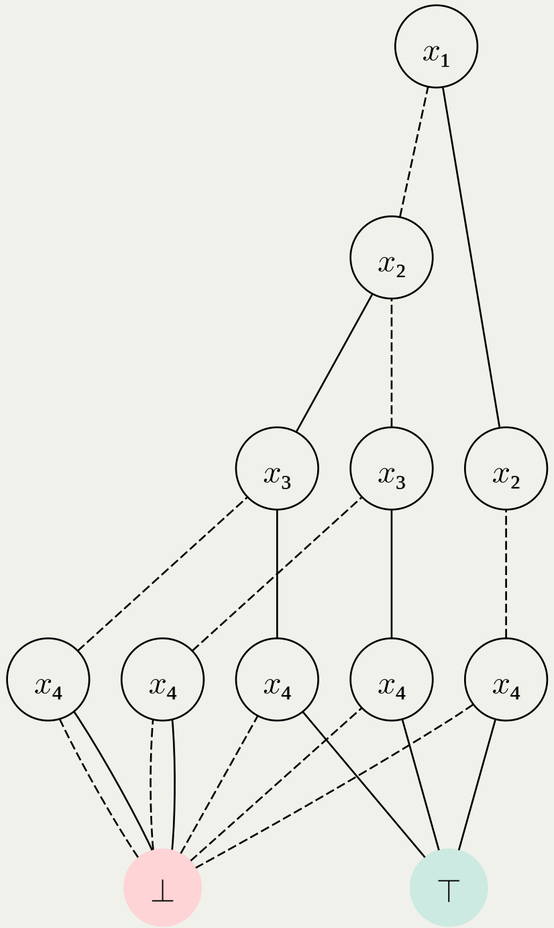
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



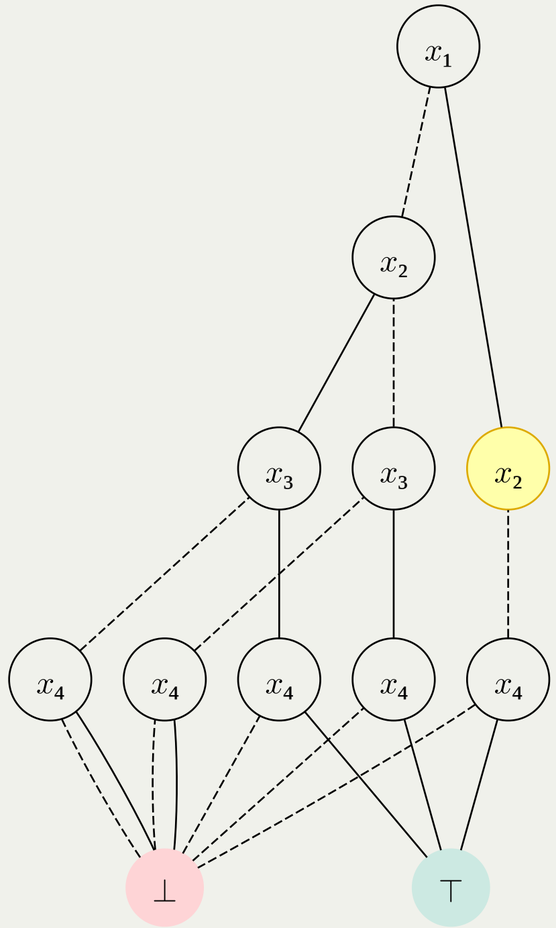
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



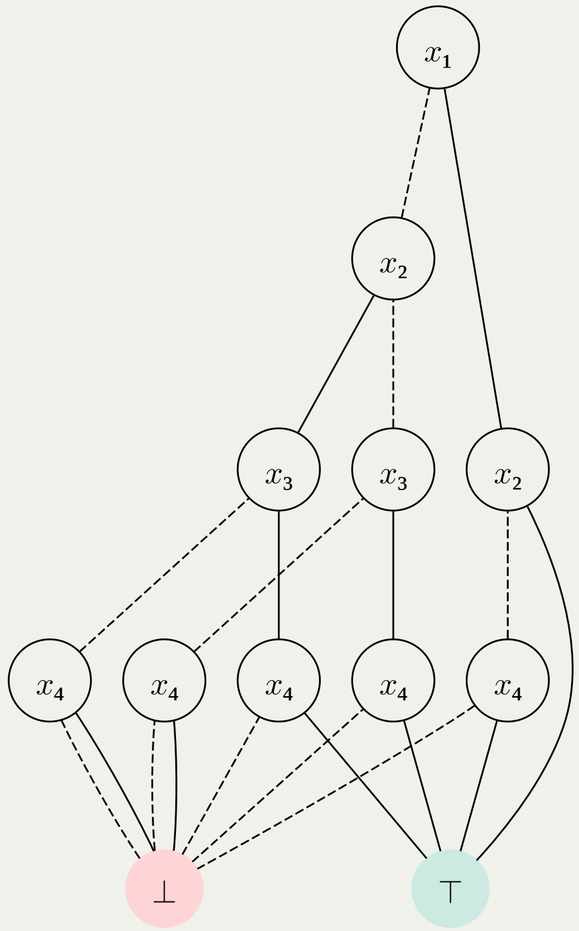
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Stupid Brute Force



-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation

-4 1 3
1 4
-1 2 4
-3 -2 1 4

Adding unit propagation

-4 1 3
1 4
-1 2 4
-3 -2 1 4

Adding unit propagation

-4 1 3
1 4
-1 2 4
-3 -2 1 4

Adding unit propagation

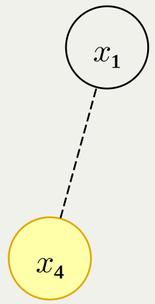
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation



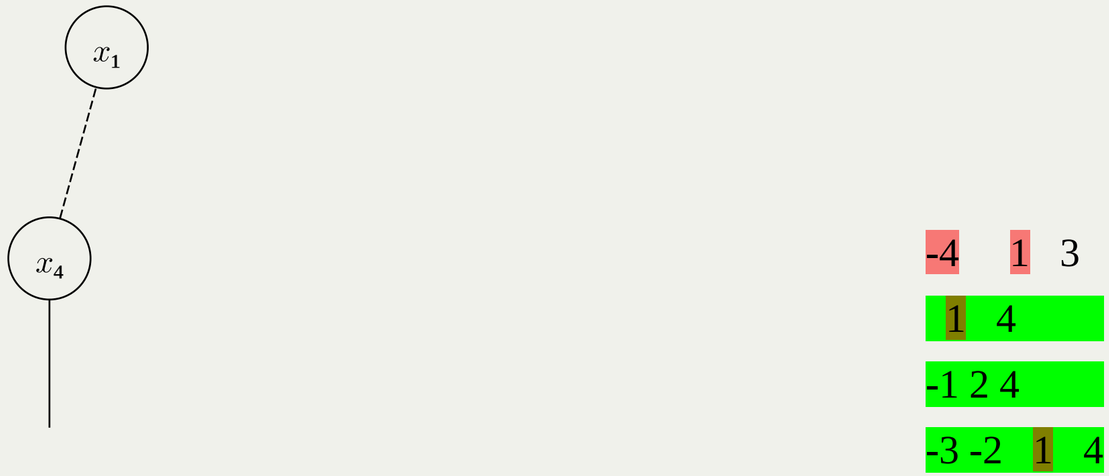
-4 1 3

1 4

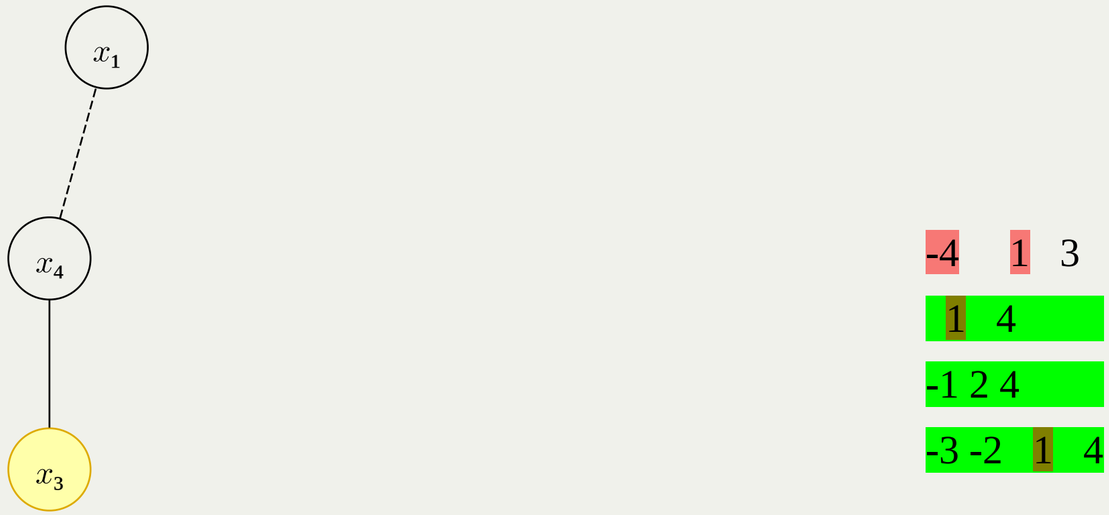
-1 2 4

-3 -2 1 4

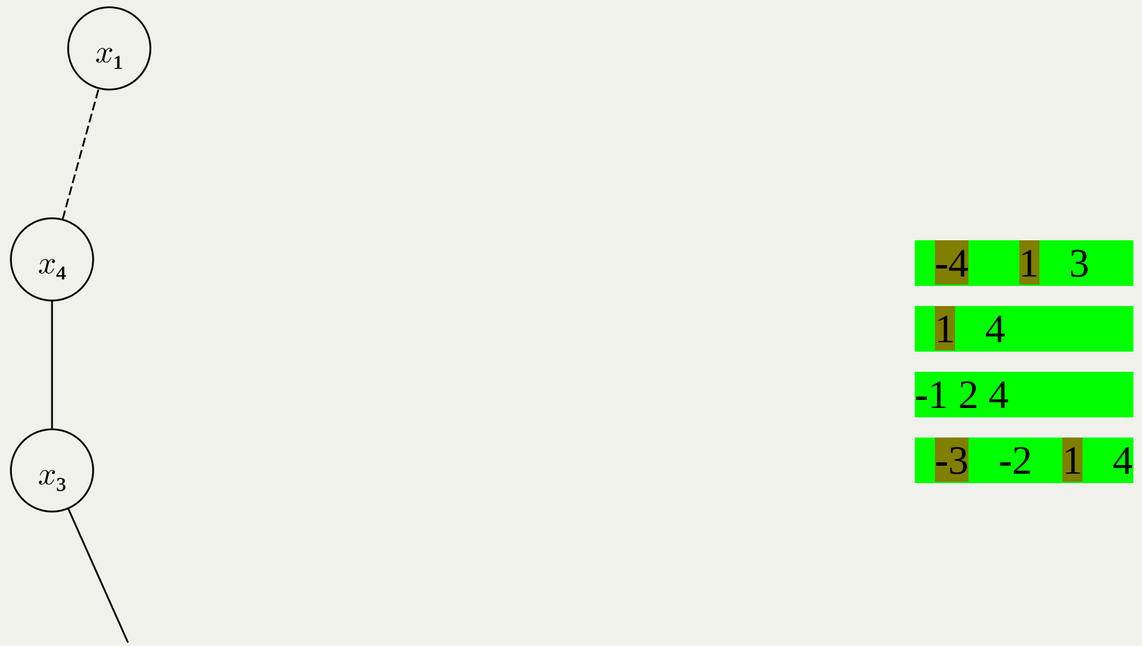
Adding unit propagation



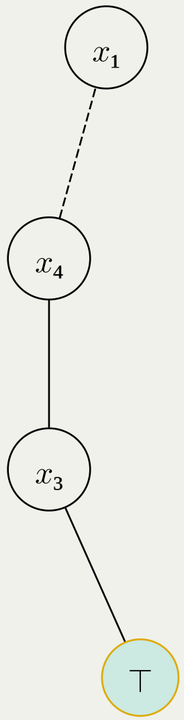
Adding unit propagation



Adding unit propagation



Adding unit propagation



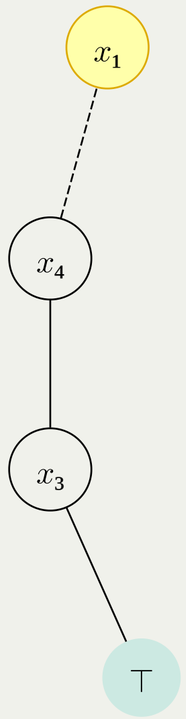
-4 1 3

1 4

-1 2 4

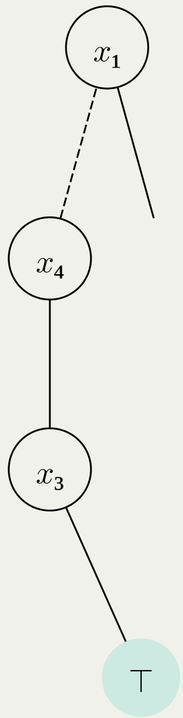
-3 -2 1 4

Adding unit propagation



-4 1 3
1 4
-1 2 4
-3 -2 1 4

Adding unit propagation



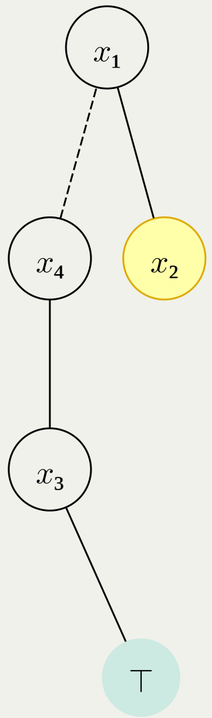
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation



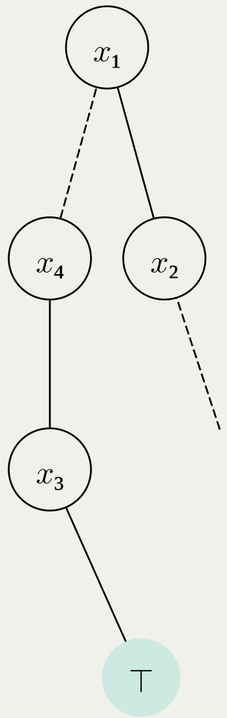
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation



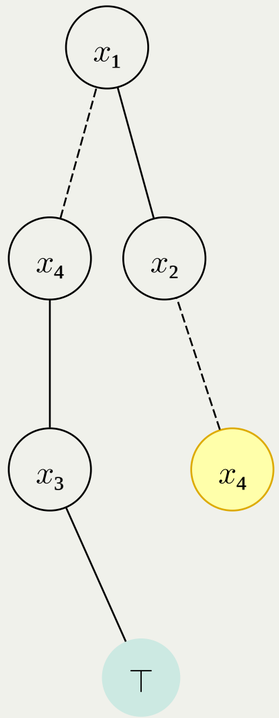
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation



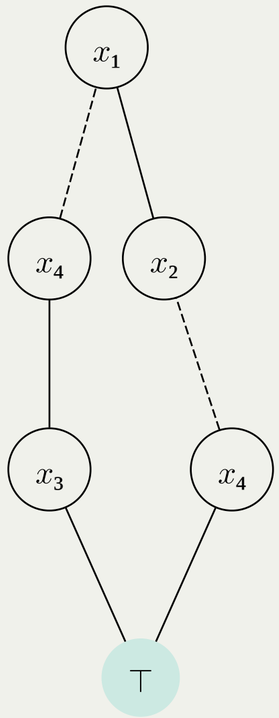
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation



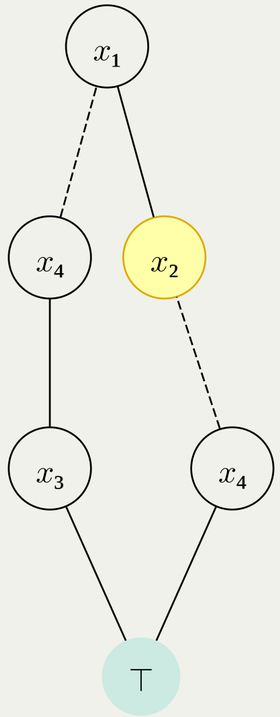
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation



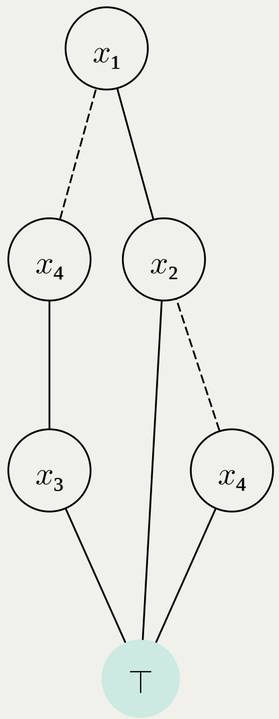
-4 1 3

1 4

-1 2 4

-3 -2 1 4

Adding unit propagation



-4 1 3

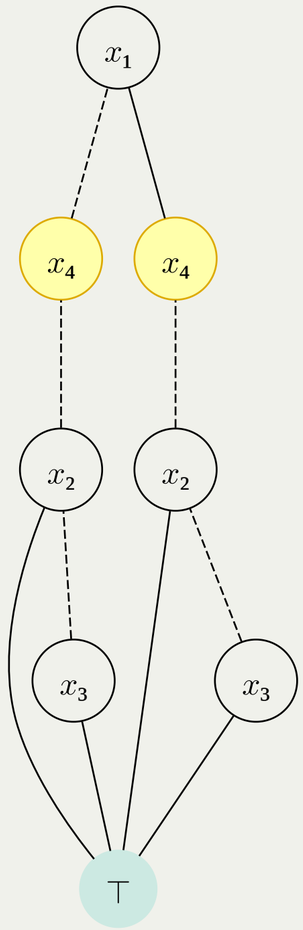
1 4

-1 2 4

-3 -2 1 4

No factorization

Current algorithm builds decision trees.



1 2 3
-1 2 3
-4 -1
-4 1

Caching

1 2 3
-1 2 3
-4 -1
-4 1

Caching

1 2 3
-1 2 3
-4 -1
-4 1

Caching

1 2 3
-1 2 3
-4 -1
-4 1

Caching

1 2 3

-1 2 3

-4 -1

-4 1

Caching

1 2 3

-1 2 3

-4 -1

-4 1

Caching

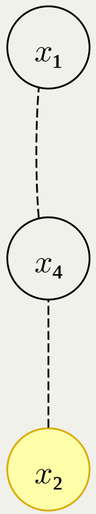
1 2 3

-1 2 3

-4 -1

-4 1

Caching



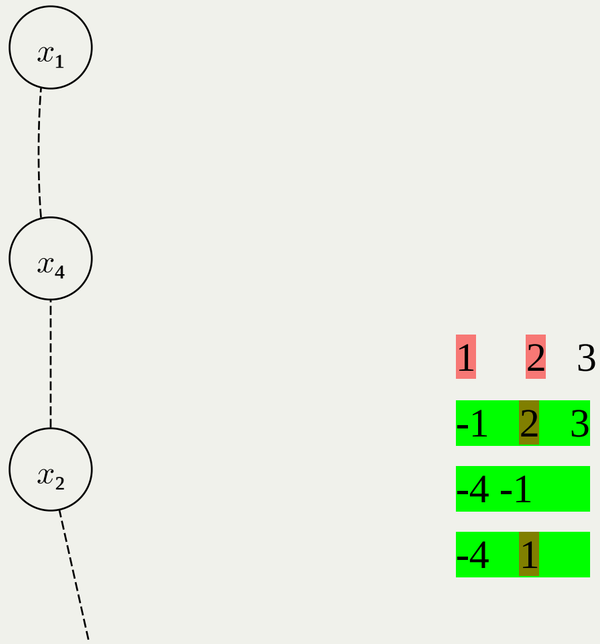
1 2 3

-1 2 3

-4 -1

-4 1

Caching



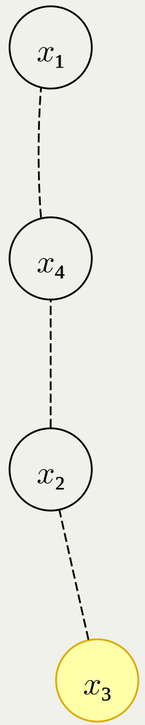
1 2 3

-1 2 3

-4 -1

-4 1

Caching



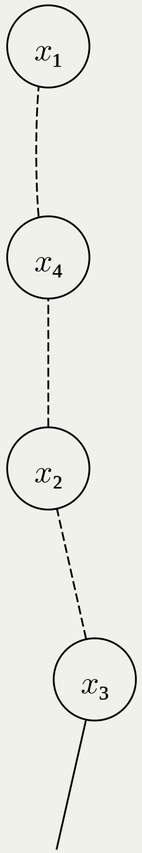
1 2 3

-1 2 3

-4 -1

-4 1

Caching



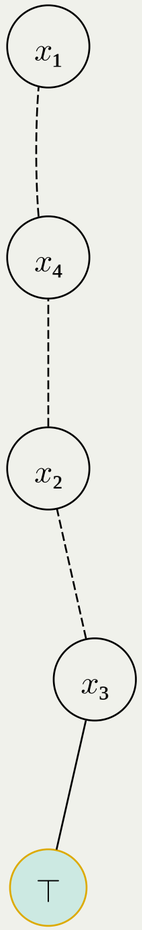
1 2 3

-1 2 3

-4 -1

-4 1

Caching



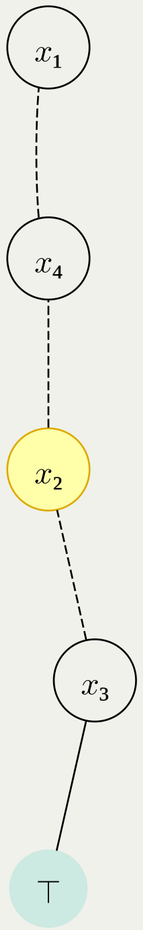
1 2 3

-1 2 3

-4 -1

-4 1

Caching



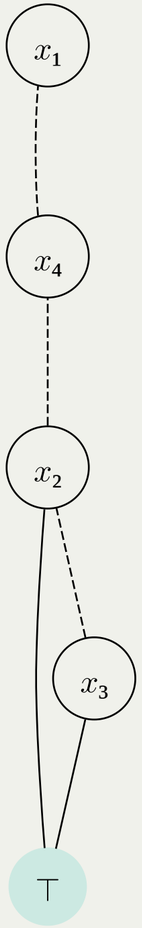
1 2 3

-1 2 3

-4 -1

-4 1

Caching



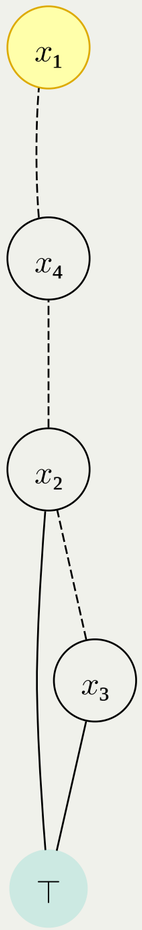
1 2 3

-1 2 3

-4 -1

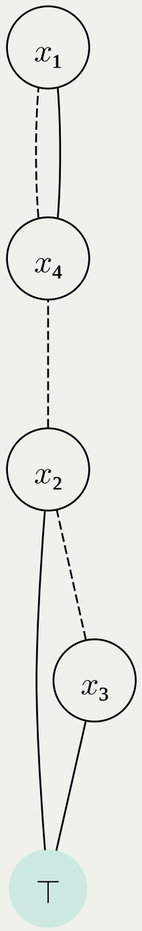
-4 1

Caching



1 2 3
-1 2 3
-4 -1
-4 1

Caching



1 2 3

-1 2 3

-4 -1

-4 1

Recall that $\phi[x_1 = 0]$:

1 2 3

-1 2 3

-4 -1

-4 1

Component Caching

- **Caching scheme**: r_c: CNF $\rightarrow$ Keys such that $r_c(\phi) = r_c(\psi)$ implies $\phi \equiv \psi$ (syntactic).
- **Cache**: Hashmap mapping keys to circuit nodes.
- **Naive scheme**: concatenate clauses written in increasing lexicographical ordering (`Cachet`).
  - big keys
  - costly to compute
- **Other scheme**:
  - `sharpSAT`: do not add binary clauses but keep set of variables (**UP** should be done before caching).
  - `d4`: only use modified clauses
  - `ganak`: only store the hash (probabilistic caching, not correct but works in practice).

Cache cleaning

In practice, the cache may grow too much.

- Add an maximum cache size.
- When cache is **too big**, clean the **less useful entries**.

Need for a *cleaning strategy*:

- `cachet` deletes half of the entries (oldest ones), when cache full.
- `sharpSAT` deletes the less active ones (the one that have not been used) when X% of the cache is full.
- `d4` scores usefulness by looking at the activity and the size of the component (smaller components are considered more useful because more likely to hit).

Decomposability

Exhaustive DPLL + Caching builds FBDD. Can we exploit decomposability?

If $\phi = \phi_1 \wedge \phi_2$ with $var(\phi_1) \cap var(\phi_2) = \emptyset$ then $compile(\phi_1) \wedge compile(\phi_2)$ is a decomposable $\wedge$-gate.

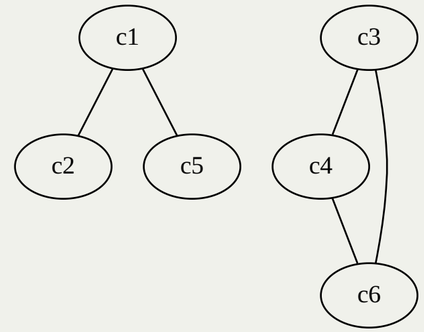
Decomposability

Exhaustive DPLL + Caching builds FBDD. Can we exploit decomposability?

If $\phi = \phi_1 \wedge \phi_2$ with $var(\phi_1) \cap var(\phi_2) = \emptyset$ then $compile(\phi_1) \wedge compile(\phi_2)$ is a decomposable $\wedge$-gate.

Connected components:

- c_1: 1 5
- c_2: -1 2
- c_3: 3 6
- c_4: 3 7
- c_5: 1 4
- c_6: 6 7 8



At each recursive call, if $\phi = \phi_1 \wedge ... \wedge \phi_k$ has more than 1 connected component, recursively compile each ϕ_i independently.

Decomposability example

-1 2 3
-6 -5
-4 -1
-1 5 6
1 2 3
-4 1

Decomposability example

-1 2 3
-6 -5
-4 -1
-1 5 6
1 2 3
-4 1

Decomposability example

-1 2 3
-6 -5
-4 -1
-1 5 6
1 2 3
-4 1

Decomposability example

-1 2 3

-6 -5

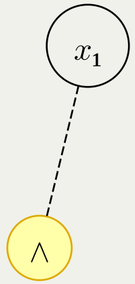
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

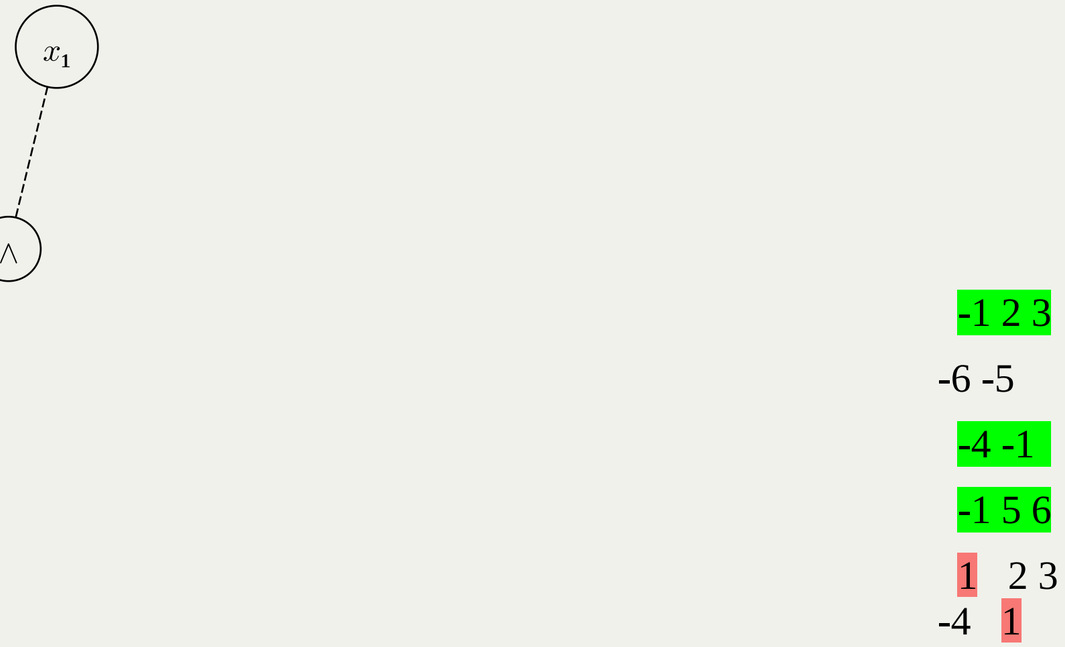
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

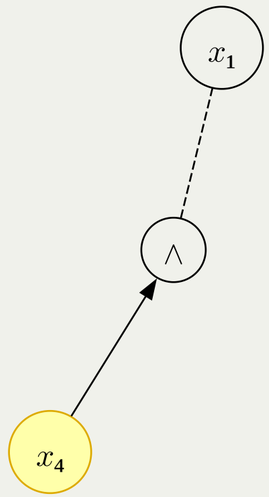
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

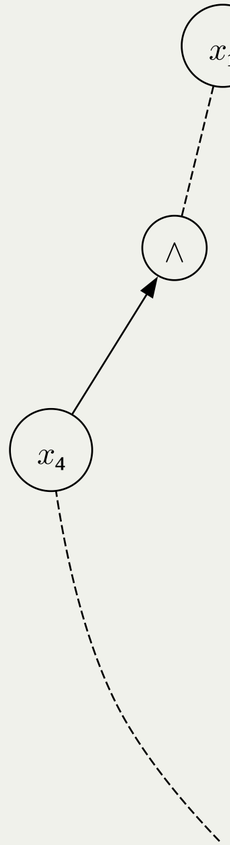
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

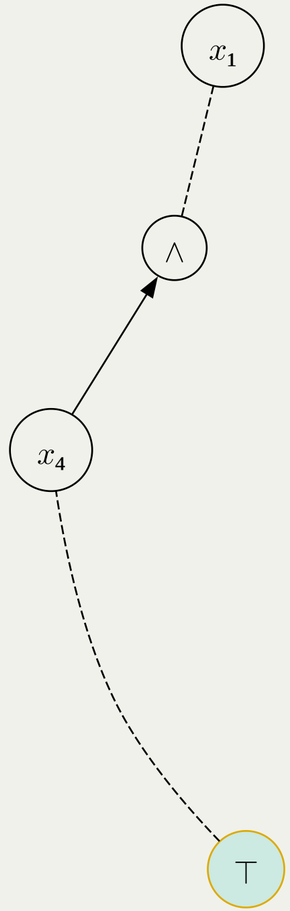
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

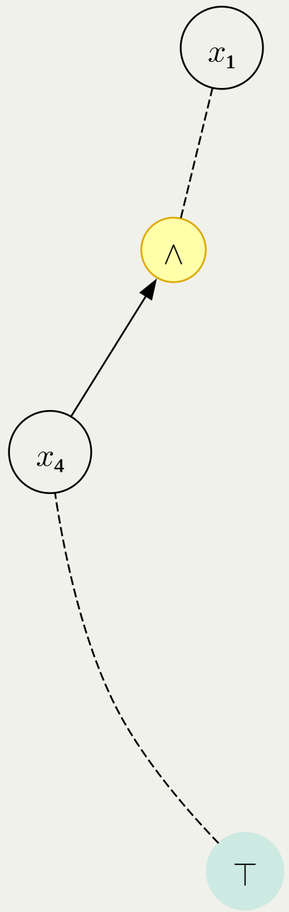
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

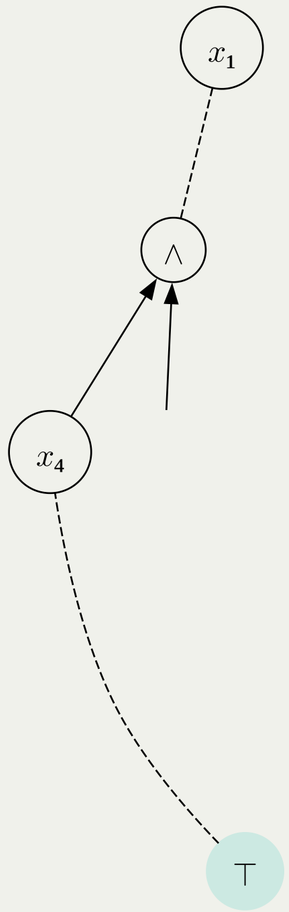
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

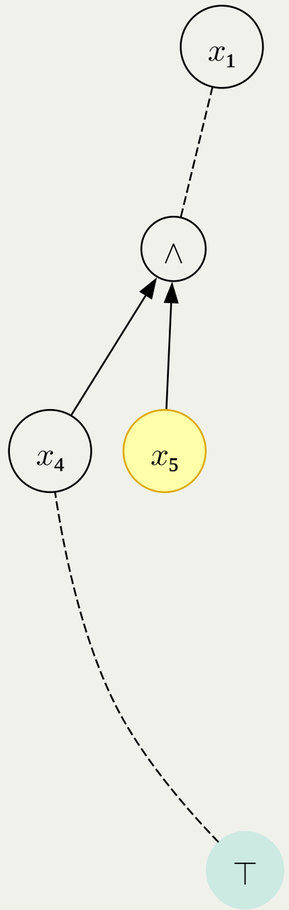
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

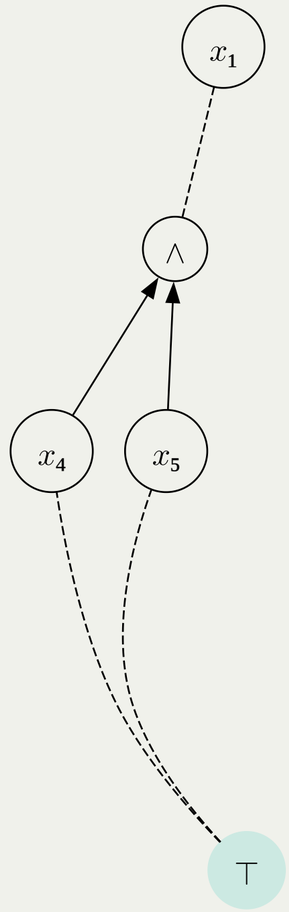
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

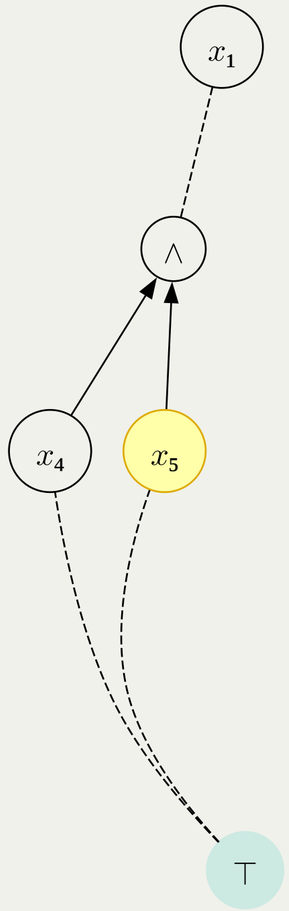
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

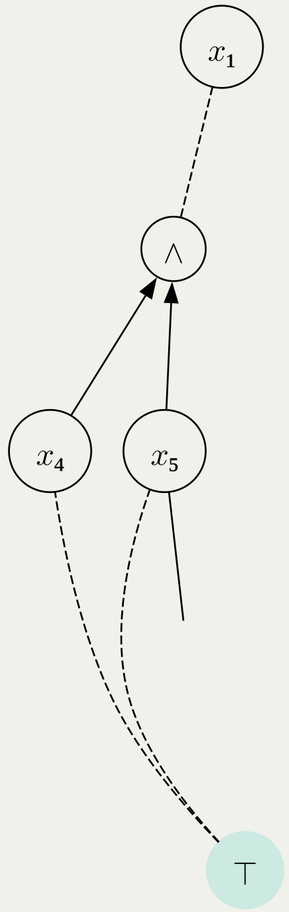
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

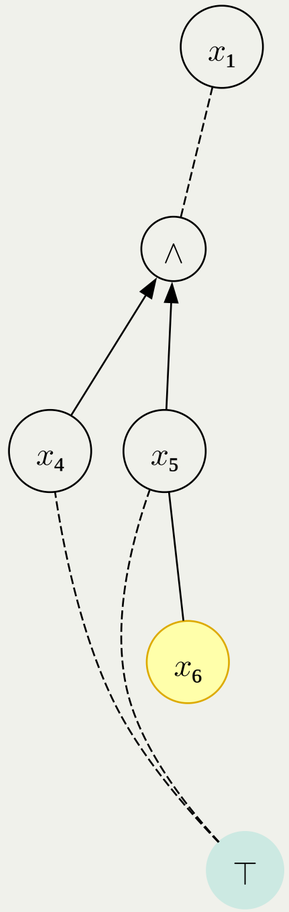
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

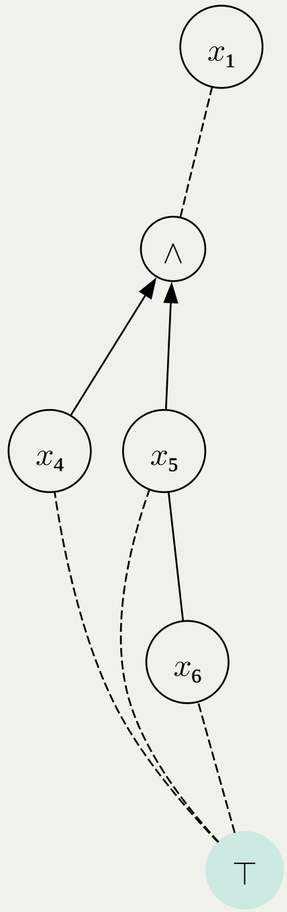
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

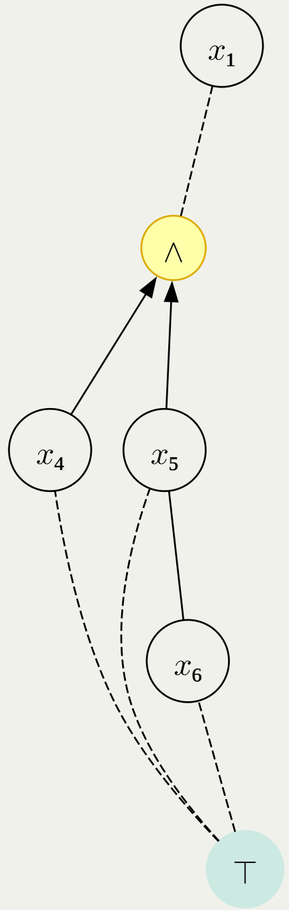
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

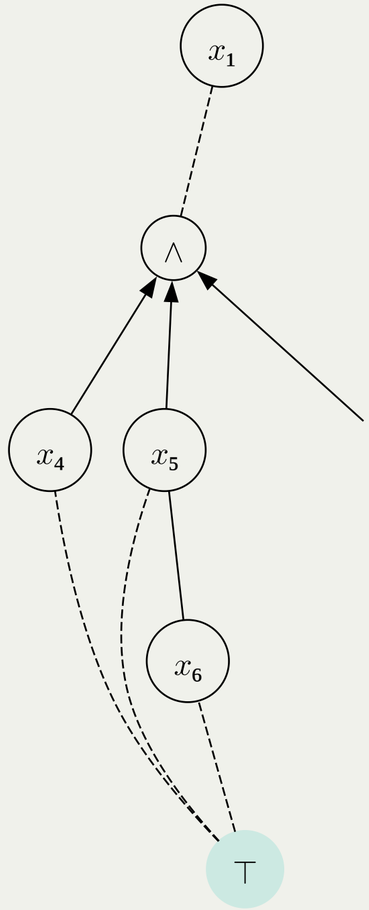
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

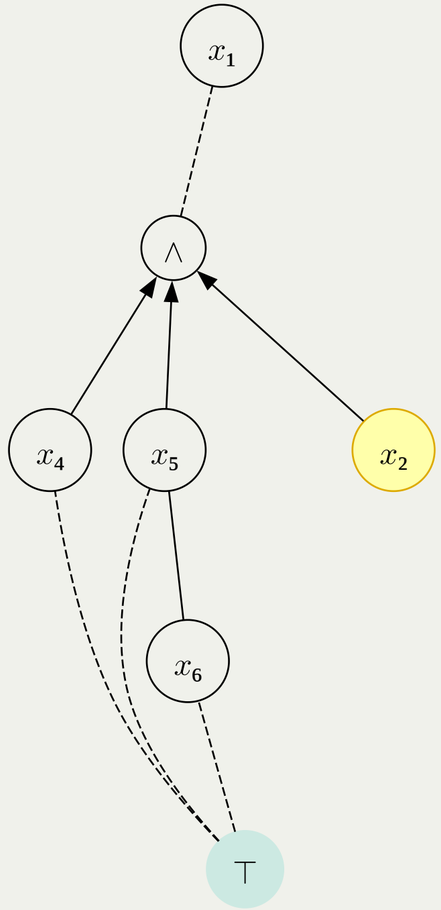
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

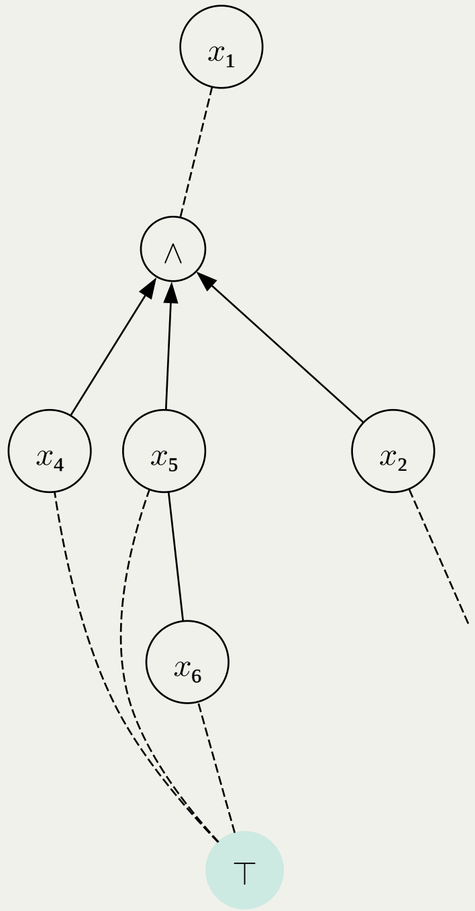
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

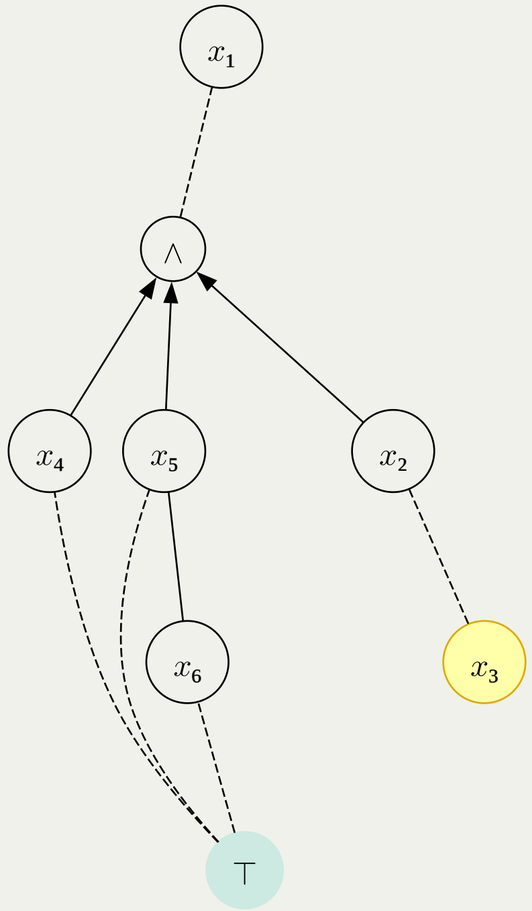
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

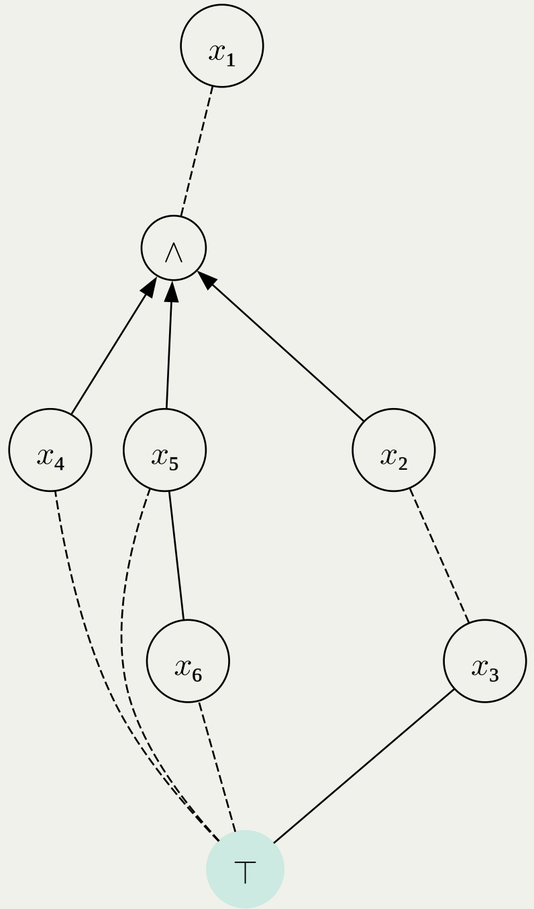
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

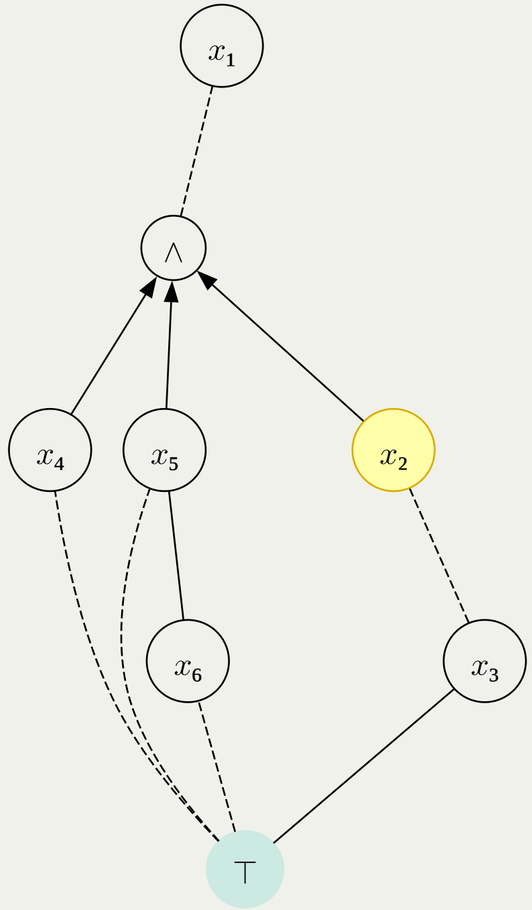
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

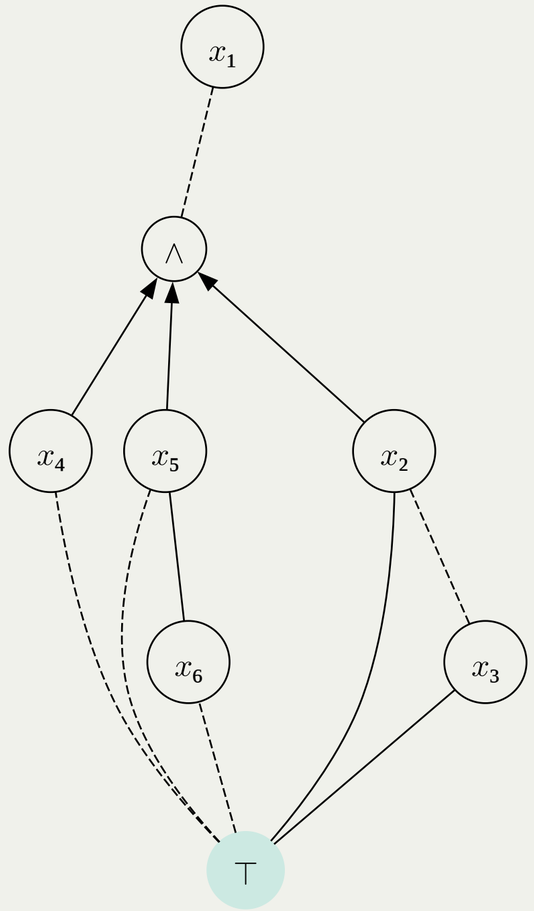
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

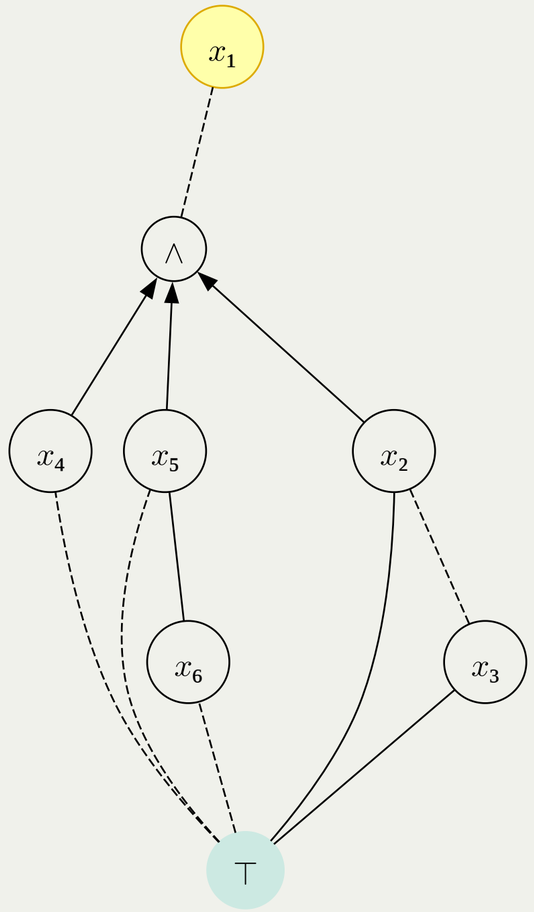
-4 -1

-1 5 6

1 2 3

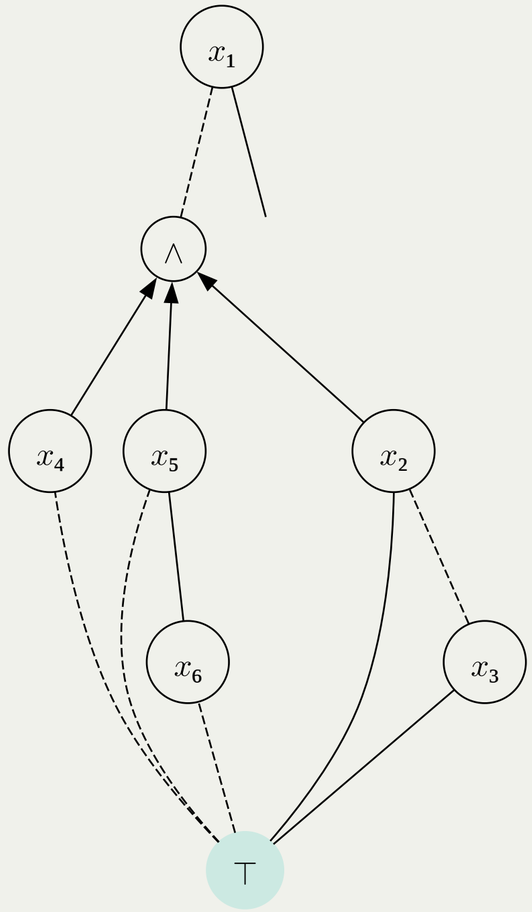
-4 1

Decomposability example



-1 2 3
-6 -5
-4 -1
-1 5 6
1 2 3
-4 1

Decomposability example

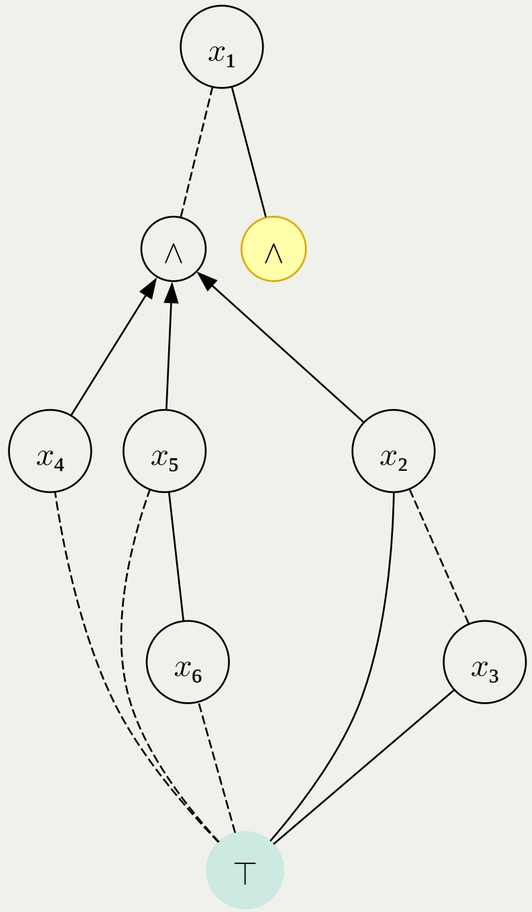


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example

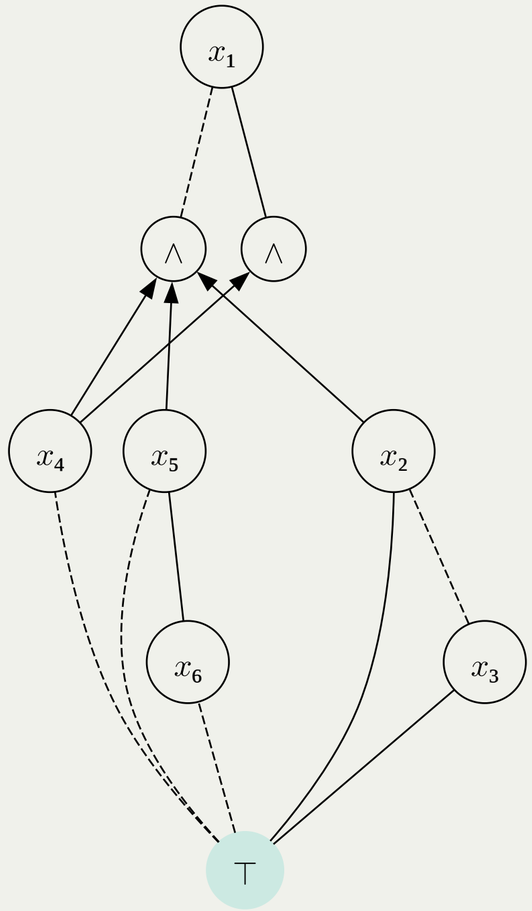


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example

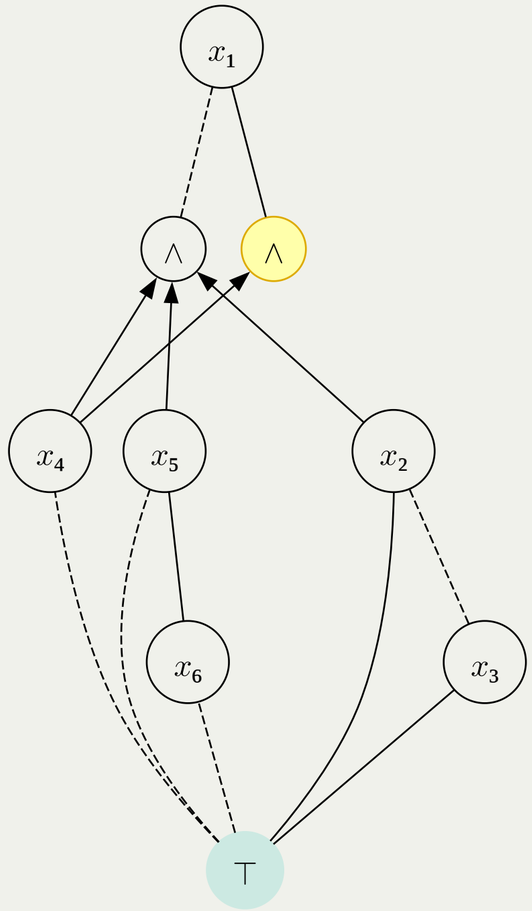


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example

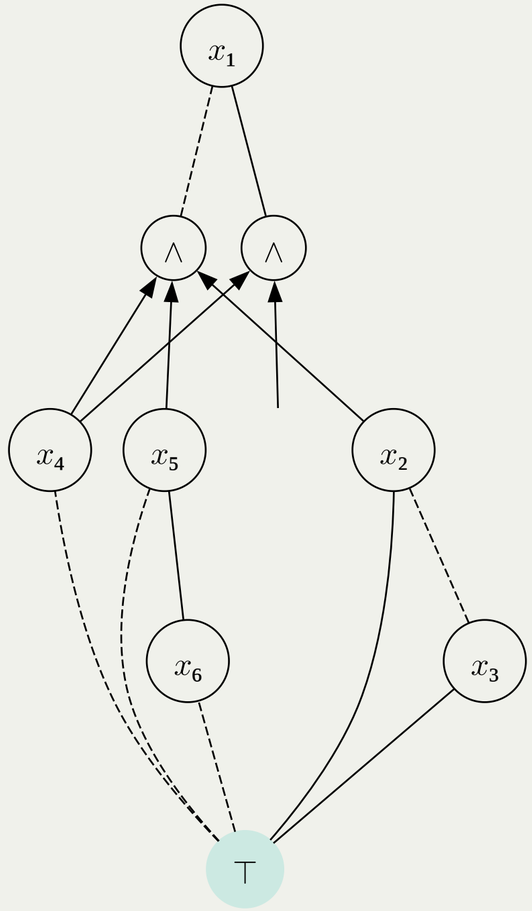


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example

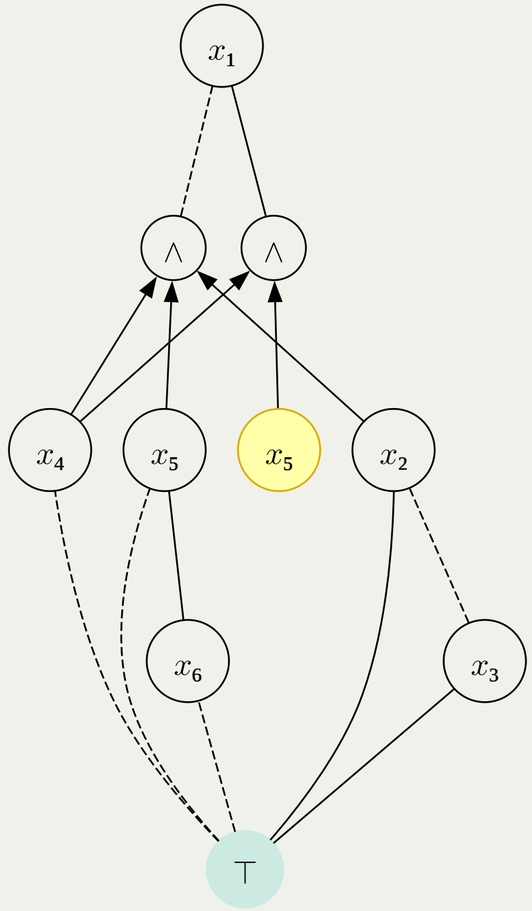


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example

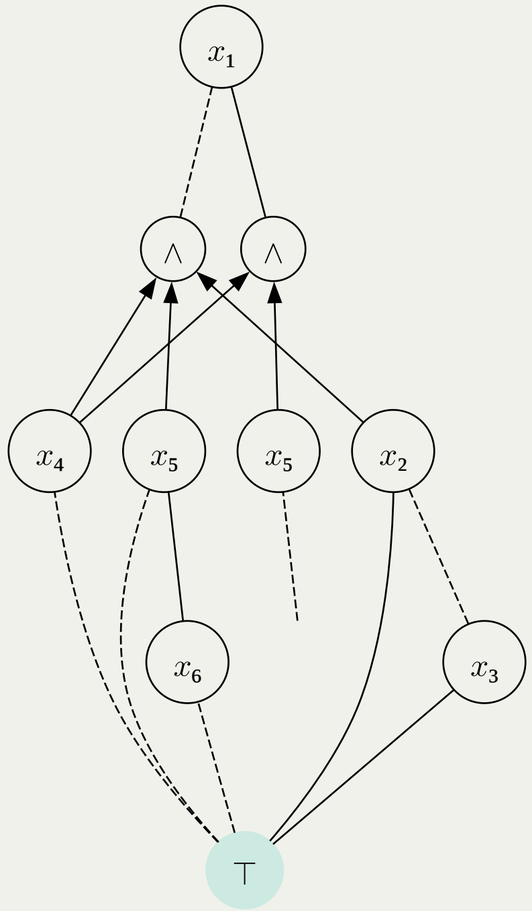


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

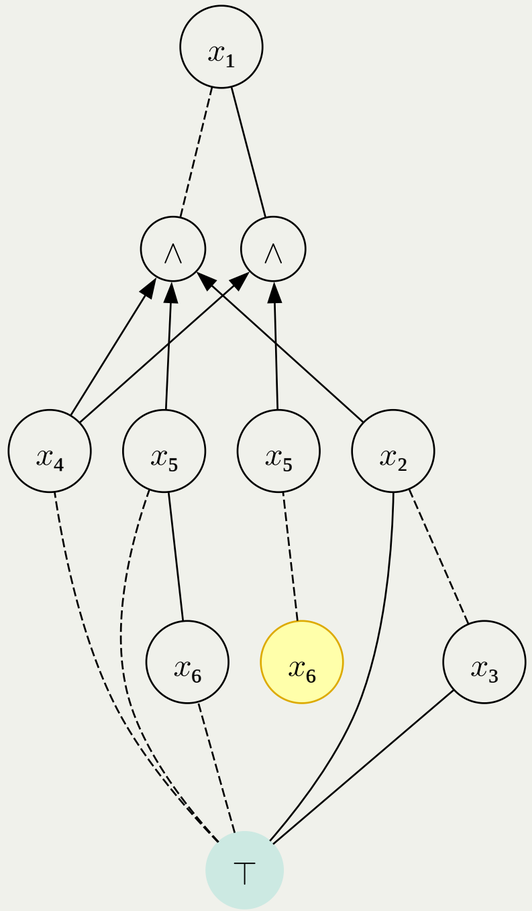
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

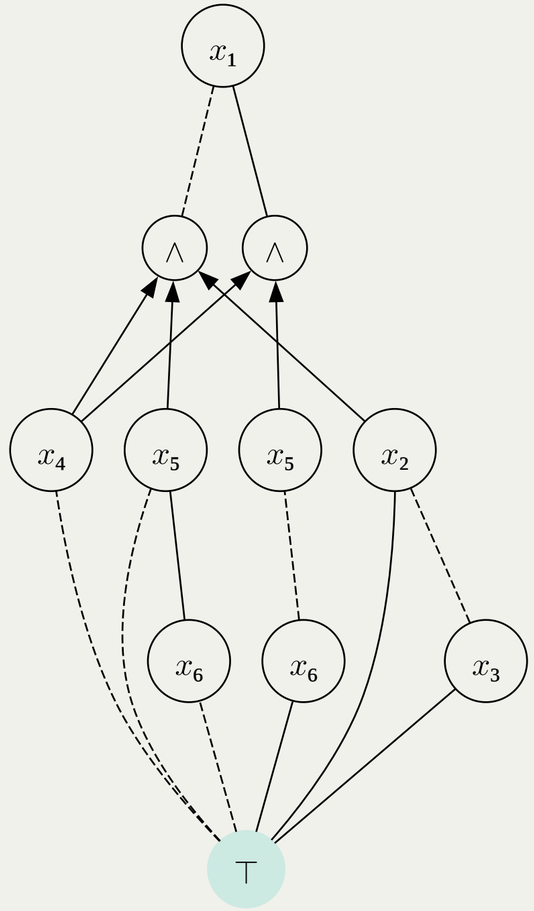
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3

-6 -5

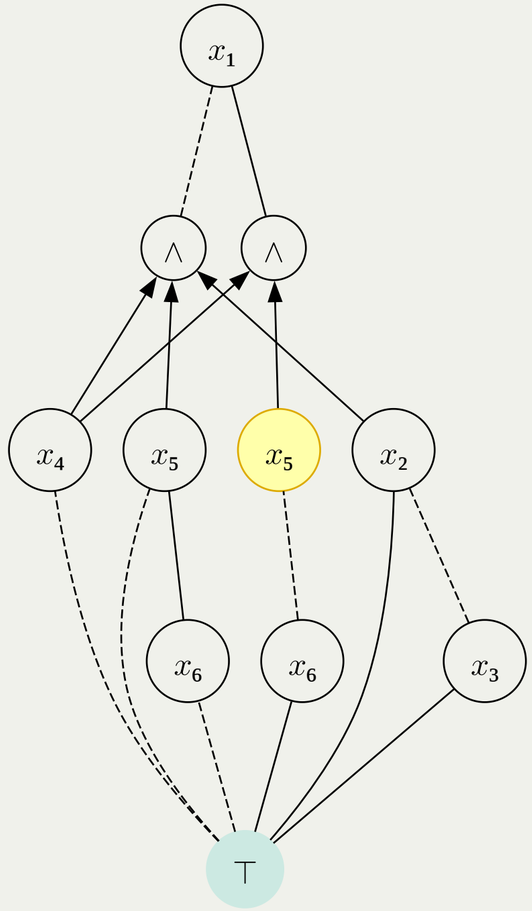
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example

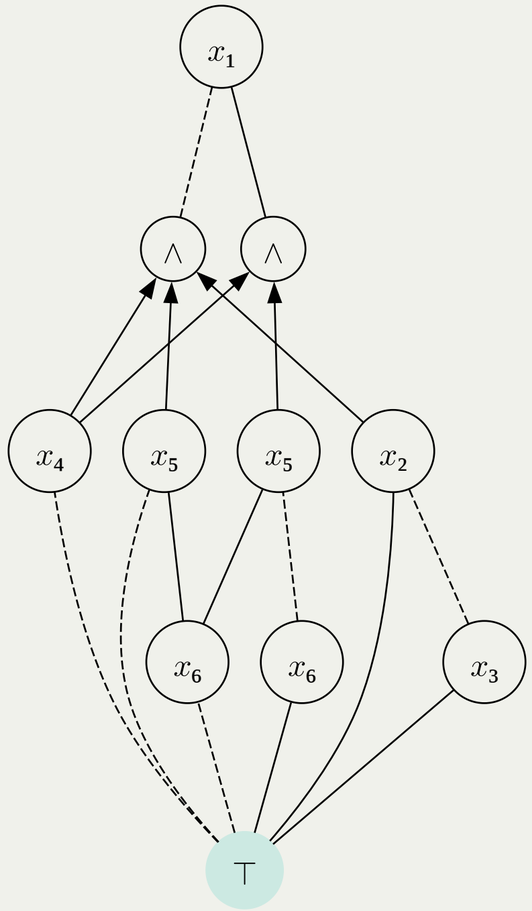


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example



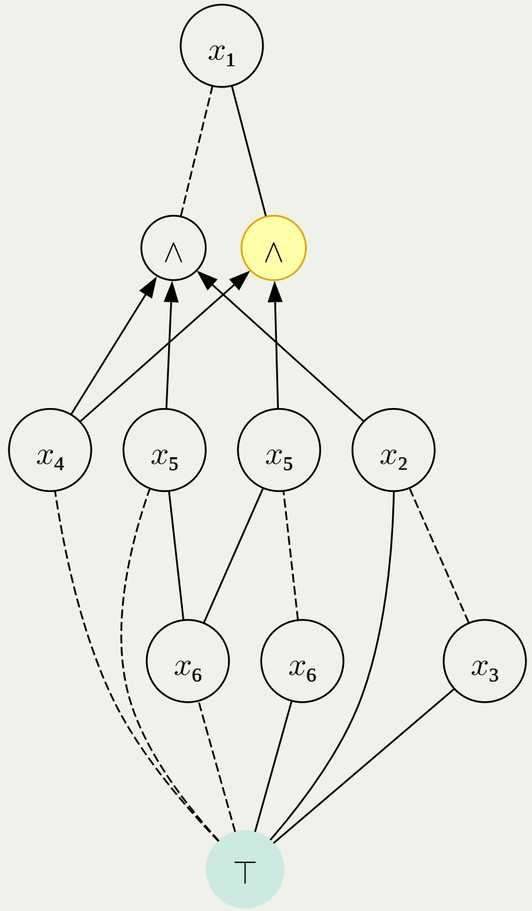
-1 2 3
-6 -5
-4 -1

-1 5 6

1 2 3

-4 1

Decomposability example

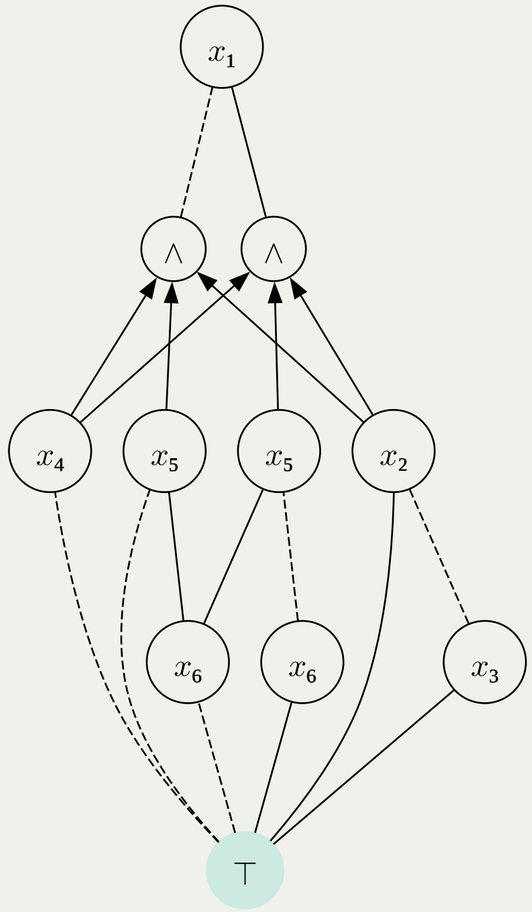


-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Decomposability example



-1 2 3
-6 -5
-4 -1
-1 5 6

1 2 3

-4 1

Variables heuristics

- Size of the outputs heavily depends on the order of variables.
- Decomposing the formula into balanced disjoint subformula is empirically the most significant improvement.

> Branch on variables likely to split the formula into balanced disjoint cc.

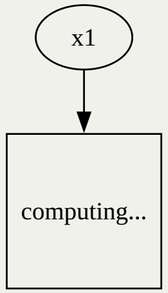
Heuristics rely on graph theory:

- `d4`: looks for a balanced (hyper)graph separator each time it branches (NP-hard to compute the optimal but existing heuristics).
- `sharpSAT-TD`: guide heuristics using a precomputed *tree decomposition*[*].

[*] I do love tree decompositions and treewidth, don't get me started.

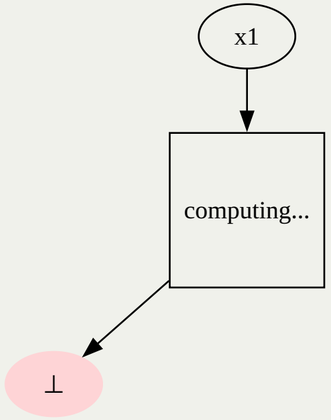
Improvement

Consider the following likely scenario:



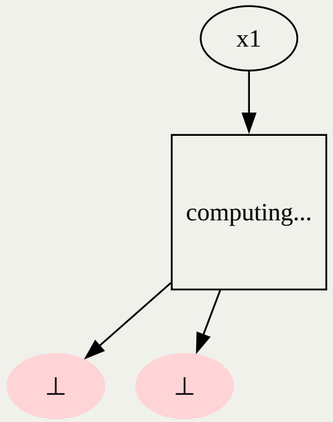
Improvement

Consider the following likely scenario:



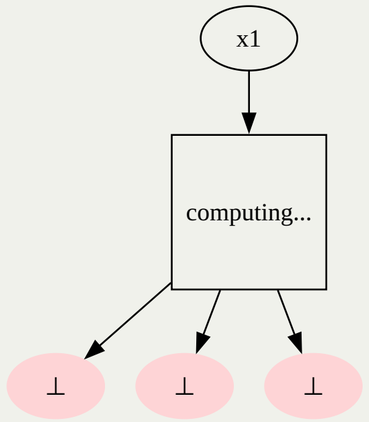
Improvement

Consider the following likely scenario:



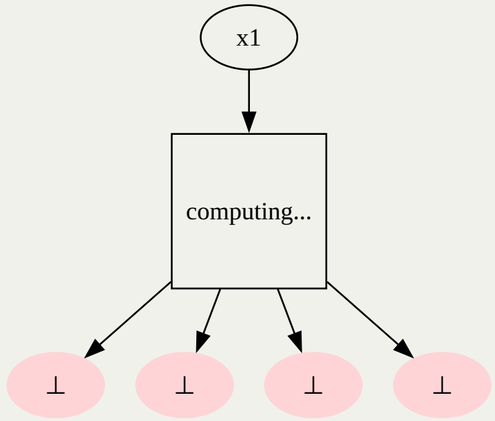
Improvement

Consider the following likely scenario:



Improvement

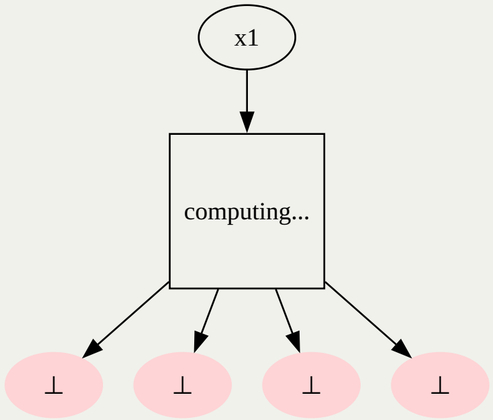
Consider the following likely scenario:



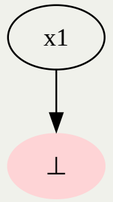
Lost in a conflict.

Improvement

Consider the following likely scenario:



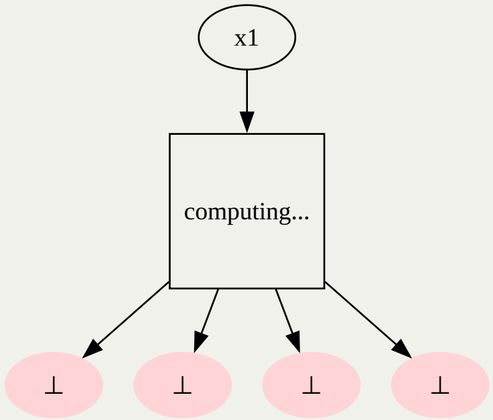
Lost in a conflict.



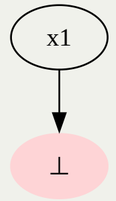
That’s better.

Improvement

Consider the following likely scenario:



Lost in a conflict.



That's better.

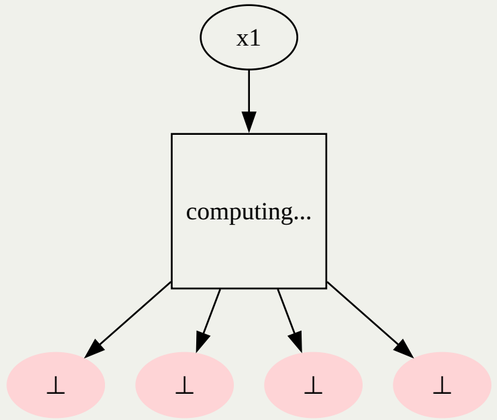
We are using DPLL to see that $\phi[x_1 = 1]$ is UNSAT…

SAT solvers use better algorithm CDCL.

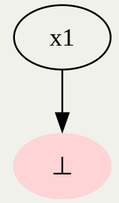
Always check a branch is satisfiable before exploring it.

Improvement

Consider the following likely scenario:



Lost in a conflict.



That's better.

We are using DPLL to see that $\phi[x_1 = 1]$ is UNSAT…
SAT solvers use better algorithm CDCL.
Always check a branch is satisfiable before exploring it.

Bonus: CDCL *returns implied clauses*, which can be used to later detect unit propagation.

Wrap up

- Exhaustive DPLL is a branching algorithm.
  - Factorisation ensured by syntactic equivalence.
  - Decomposition is also syntactic.
  - Builds decDNNF.
- Successful in practice:
  - Implemented in many good tools.
  - Approach which dominates model counting competition.
  - *side note*: preprocessing is really important too.

Building circuits bottom-up

A word on bottom-up

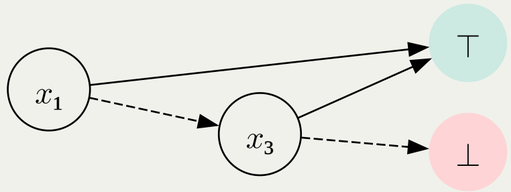
- Approach only works on representation language supporting *conjunction*.
- In this lecture: focus on OBDD.

In a nutshell: apply, minimise, apply, minimise, apply…

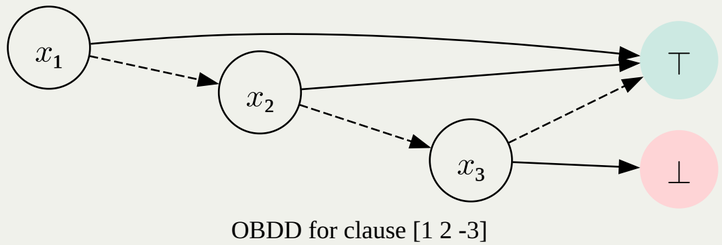
```
F = CNF(...)
p = choose a "good" order for F
D = OBDD(p)

for c in F:
    DC = OBDD(c,p)
    D.apply(DC)
    D.minimize()
```

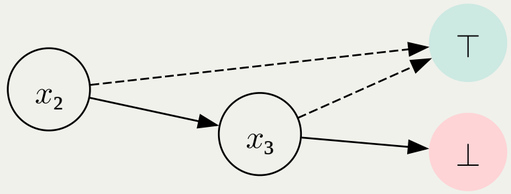
Apply, minimise, apply, minimise…



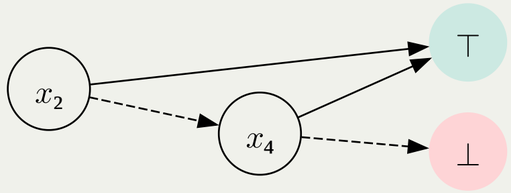
OBDD for clause [1 3]



OBDD for clause [1 2 -3]

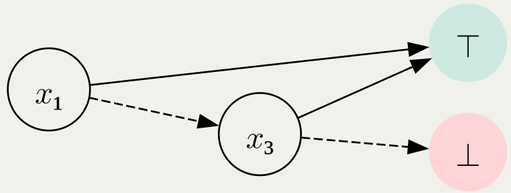


OBDD for clause [-2 -3]

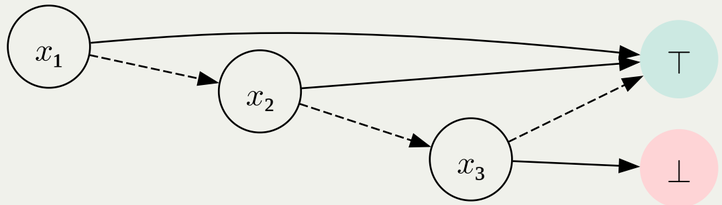


OBDD for clause [2 4]

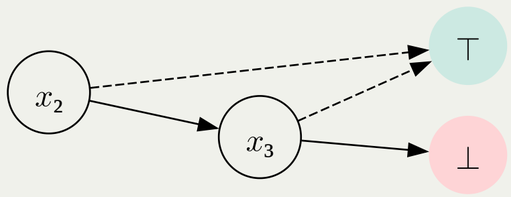
Apply, minimise, apply, minimise…



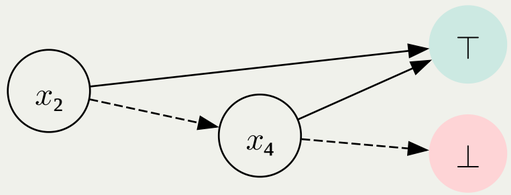
OBDD for clause [1 3]



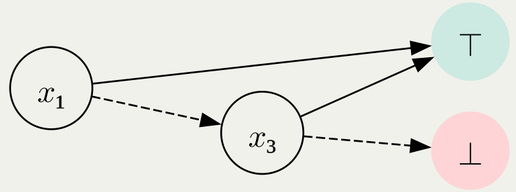
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]

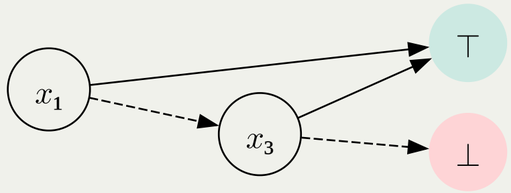


OBDD for clause [2 4]

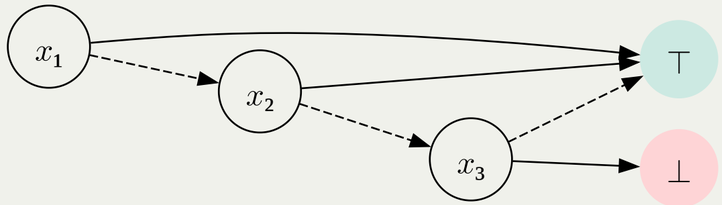


Apply clause 1 3.

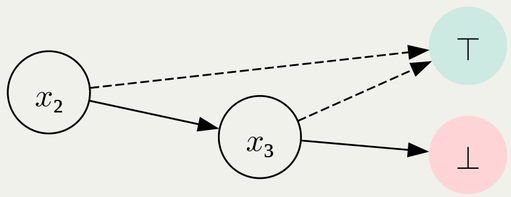
Apply, minimise, apply, minimise…



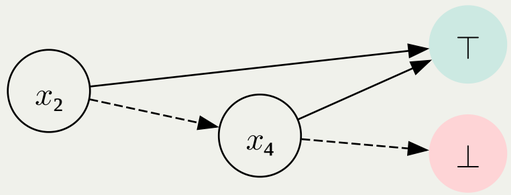
OBDD for clause [1 3]



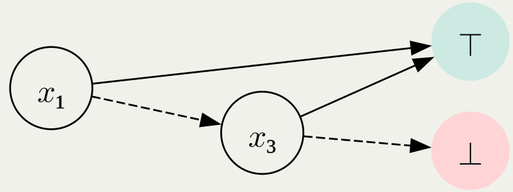
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]

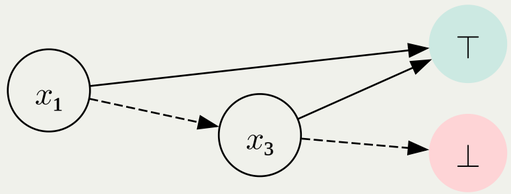


OBDD for clause [2 4]

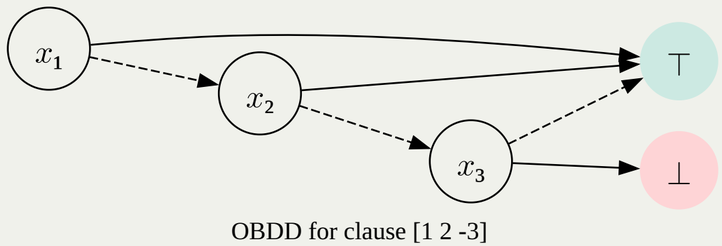


Minimise.

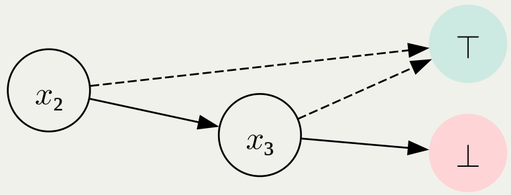
Apply, minimise, apply, minimise…



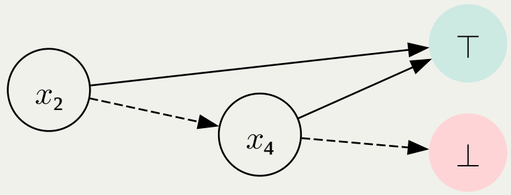
OBDD for clause [1 3]



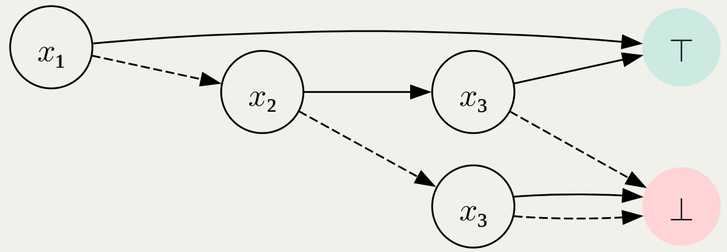
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]

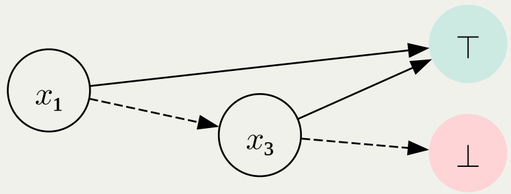


OBDD for clause [2 4]

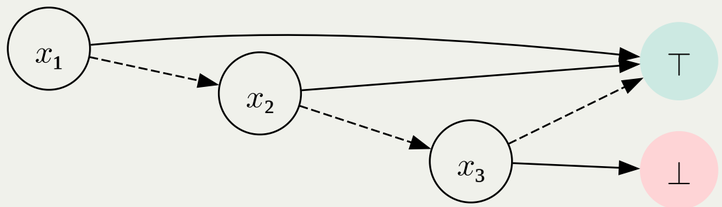


Apply clause 1 2 -3.

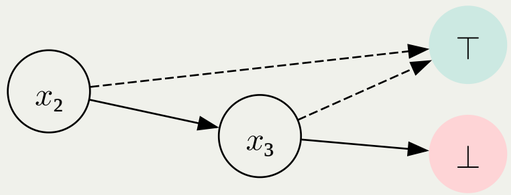
Apply, minimise, apply, minimise…



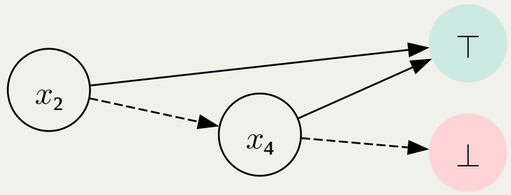
OBDD for clause [1 3]



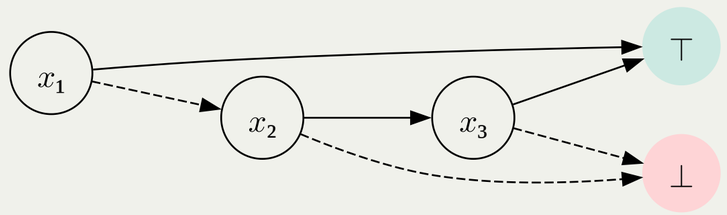
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]

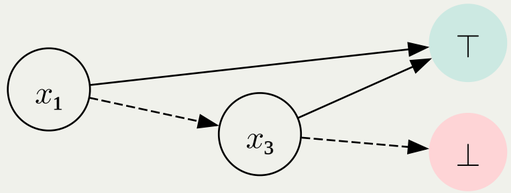


OBDD for clause [2 4]

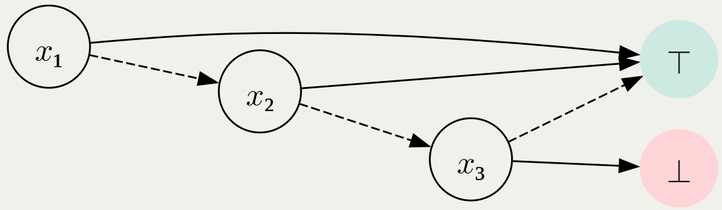


Minimise.

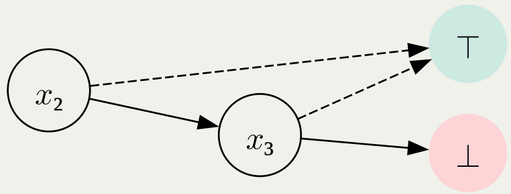
Apply, minimise, apply, minimise…



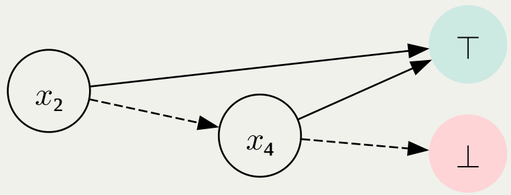
OBDD for clause [1 3]



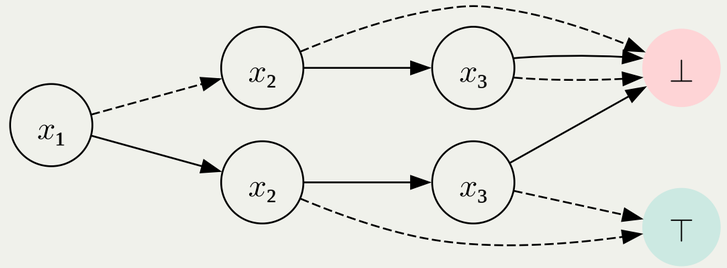
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]

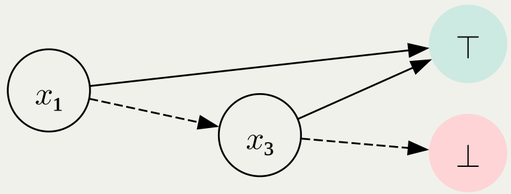


OBDD for clause [2 4]

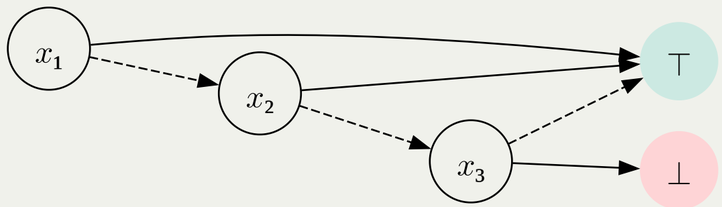


Apply clause -2 -3.

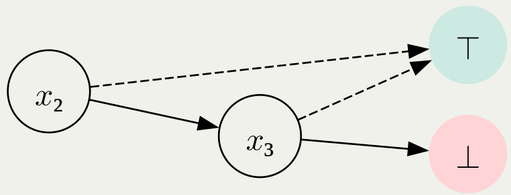
Apply, minimise, apply, minimise…



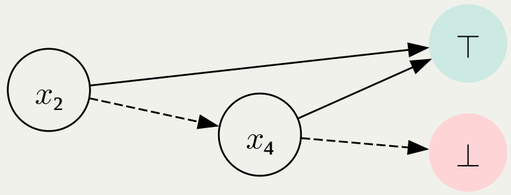
OBDD for clause [1 3]



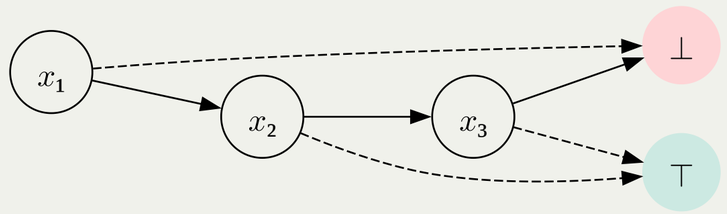
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]

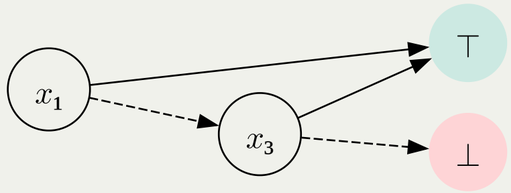


OBDD for clause [2 4]

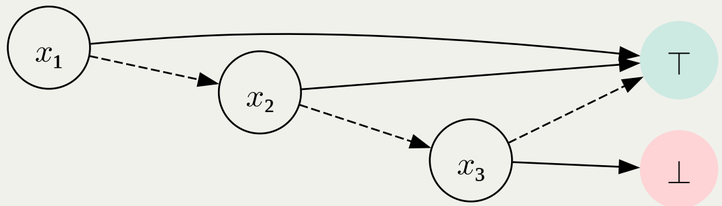


Minimise.

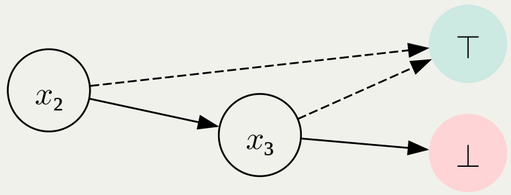
Apply, minimise, apply, minimise…



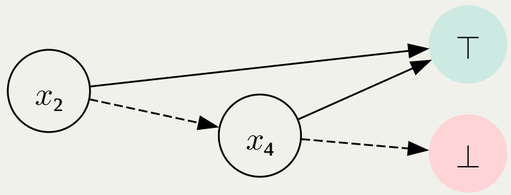
OBDD for clause [1 3]



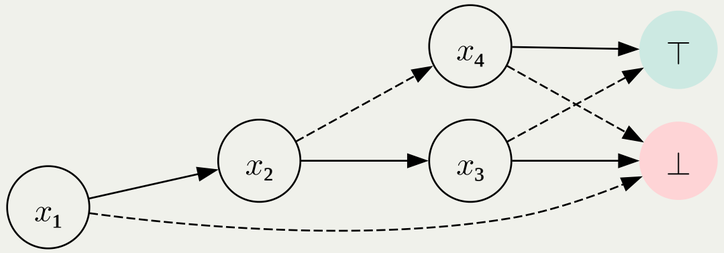
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]

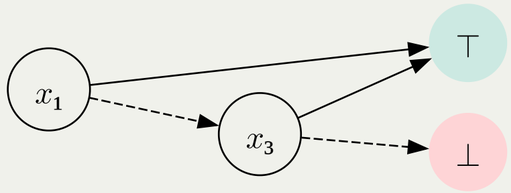


OBDD for clause [2 4]

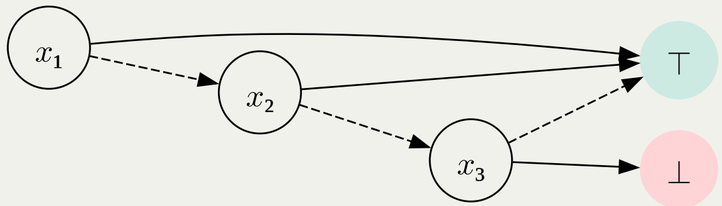


Apply clause 2 4.

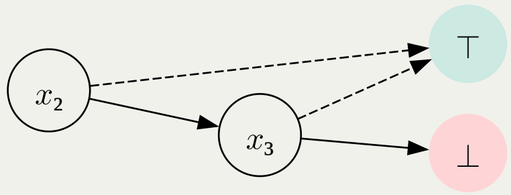
Apply, minimise, apply, minimise…



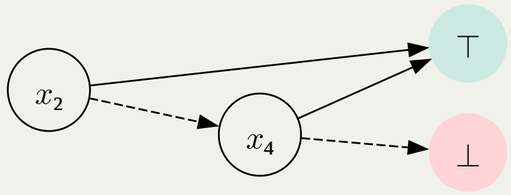
OBDD for clause [1 3]



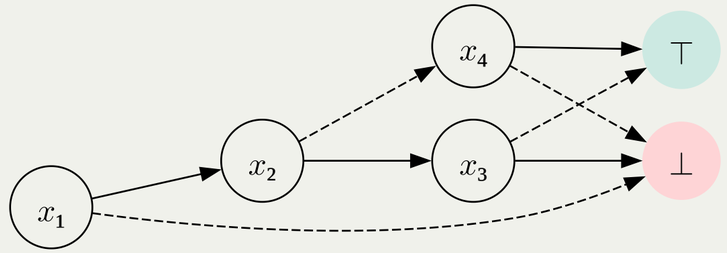
OBDD for clause [1 2 -3]



OBDD for clause [-2 -3]



OBDD for clause [2 4]



Minimise.

Advantages and disadvantages

Pros:

- Work on any constraints represented by OBDD (e.g., parity constraints)
- Canonicity/minimality ensures we get the “best” representation for the chosen order (top-down approaches may miss some equivalence if they ar e not syntactic).

Cons:

- Restricted target language
- Order of variables has a huge (and hard to predict) impact on performances.
- Order of clauses too:
  - $(x) \wedge (HARDCNF) \wedge (\neg x)$ vs
  - $(x) \wedge (\neg x) \wedge (HARDCNF)$

Min-fill heuristics

Computing optimal order is NP-hard [1]: use heuristics.

Example: Min-Fill.

- $fill(x)$: number of pairs of variables interacting with x but not together.
  - $x \wedge y | x \wedge z | y \wedge z | x \wedge w_1 \wedge w_2$
  - $fill(x)$ is 4 because $\{y, w_1\}, \{y, w_2\}, \{z, w_1\}, \{z, w_2\}$
- Remove variables greedily by picking the ones with the smallest fill.

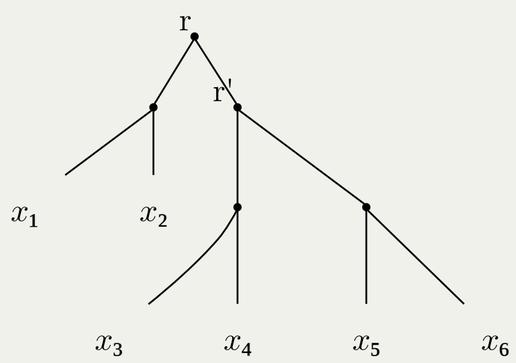
Rationale: branching on a variable will introduce “new” dependencies. We try to minimize it.

Beyond OBDD

- OBDD are more restricted than decision-DNNF.
- But decision-DNNF and d-DNNF do not support APPLY.

Idea: OBDD is a way of **structuring variables in FBDD**. Add structure variables in DNNF too?

Structured d-DNNF [1]



Structured along a variable tree (vtree).

Every $\wedge$-gate of the DNNF must split variables accoring to a node of the vtree:

- for example $g(x_1, x_2) \wedge g'(x_3, x_5)$ (split according to r)
- or $g(x_3, x_4) \wedge g'(x_5, x_6)$ (split according to r')
- **but not** $g(x_1, x_4) \wedge g'(x_2)$
- **nor** $g(x_3, x_5) \wedge g'(x_4, x_6)$

[1] Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In AAAI, volume 8, pages 517–522, 2008.

Structured d-DNNF and Bottom-up compilation

Theorem

If C_1, C_2 respect the same vtree T, we can build a circuit for $C_1 \wedge C_2$ respecting T of size at most $|C_1| \cdot |C_2|$.

Good news for bottom up compilation?

Structured d-DNNF and Bottom-up compilation

Theorem
If C_1, C_2 respect the same vtree T, we can build a circuit for $C_1 \wedge C_2$ respecting T of size at most $|C_1| \cdot |C_2|$.

Good news for bottom up compilation?

But not enough:

- no minimisation
- no canonical representation
- *the size of the circuit can only grow!*

SDD and TDD

- SDD (Sential Decision Diagrams)
- TDD (Tree Decision Diagrams)

Both are **canonical** restrictions of structured d-DNNF: allows for bottom-up compilation.

- `sdd` to compile into SDD.
- `TiDiDi` to compile into TDD (soon available): very good results at the MC Competition 2026.

Some observations:

- Both support negation and many other queries.
- Canonical reprentations for SDD are not *minimal* (may even be exponentially larger).
- SDD < TDD.
- Bottleneck for these tools: finding the right vtree! Use heuristics (min fill generalization, tree decompositions etc.)

Part III

Limits

SUBFUNCTIONS

Notion of Subfunctions

Let f: $\{0,1\}^X \to \{0,1\}$ be a Boolean function.

A Y-subfunction of f is a Boolean function on variables Y defined as: $f[\tau]$ for some $\tau \in \{0,1\}^{X \setminus Y}$.

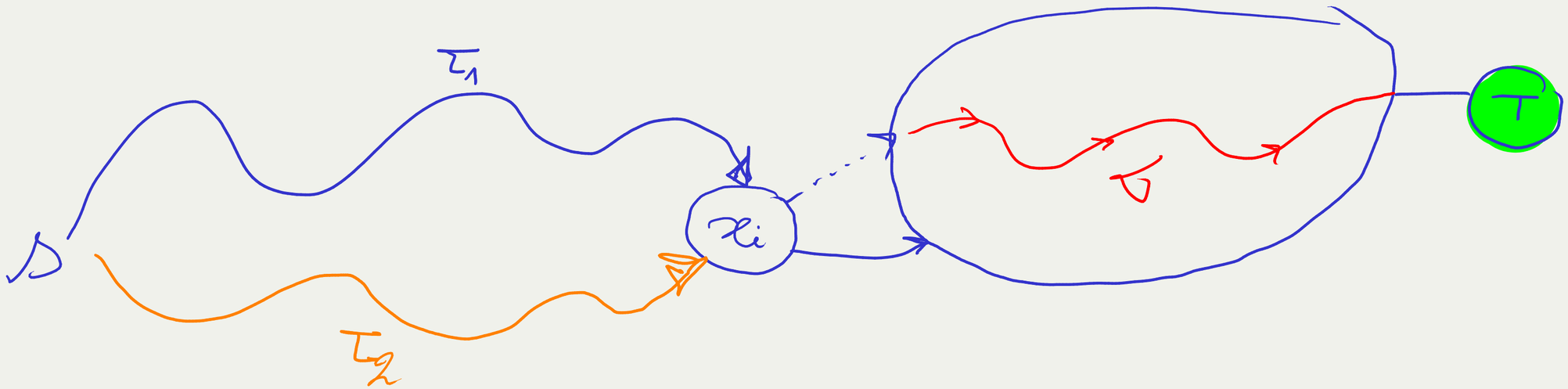
$$sub_Y(f) = \{f[\tau] \mid \tau \in \{0,1\}^{X \setminus Y}\}.$$

Of size at most $2^{|X|-|Y|}$ but often smaller.

$EVEN(x_1, ..., x_4)$ has two $\{x_3, x_4\}$-subfunctions:

- $EVEN_4[x_1 = 0, x_2 = 0] = EVEN_4[x_1 = 1, x_2 = 1] = EVEN_2(x_3, x_4)$
- $EVEN_4[x_1 = 1, x_2 = 0] = EVEN_4[x_1 = 0, x_2 = 1] = ODD_2(x_3, x_4)$

Subfunctions and OBDD



$f[\tau_1]$ and $f[\tau_2]$ define the same $\{x_i, ..., x_n\}$-subfunction.
At least $\#sub_{\{x_i, ..., x_n\}}(f)$ nodes testing x_i.

The Sieling-Wegener's bound

Theorem

Let C the smallest **complete** OBDD computing f. The decision-gates of C testing x_i are in one to one correspondence with $sub_{x_i,...,x_n}(f)$.

- Complete OBDD of size at most $n \cdot M$ where M is the size of the smallest OBDD for f.
- Lower bound on $\#sub_{x_i,...,x_n}(f)$ directly gives a lower bound on the size of the minimal complete OBDD for f using order π.

Orders and sizes

$MATCH_n(X,Y) = \bigwedge_{i=1}^{n} x_i \vee y_i$.

- $\pi_1 = \{x_1, y_1, ..., x_n, y_n\}$: OBDD of size $O(n)$.
- $\pi_2 = \{x_1, ..., x_n, y_1, ..., y_n\}$: smallest OBDD has size $\geq 2^n$.
  - $\tau_0, \tau_1 \in \{0,1\}^X$ s.t. $0 = \tau_0(x_i) \neq \tau_1(x_i) = 1$.
  - $\alpha \in \{0,1\}^Y$: $\alpha(y_i) = 0$, $\alpha(y_j) = 1$ for $j \neq i$.
  - $MATCH_n[\tau_0] \neq MATCH_n[\tau_1]$ because $0 = MATCH_n(\tau_0 \times \alpha) \neq MATCH_n(\tau_1 \times \alpha) = 1$.

Hard function for OBDD

$M = \{x_{1,1}, ..., x_{n,n}\}$ matrix of Boolean variables.

- $ROW_n(M)$ iff M has a row of ones.
- $COL_n(M)$ iff M has a column of ones.

$$f = ROW_n \vee COL_n$$

Hard function for OBDD

$M = \{x_{1,1}, ..., x_{n,n}\}$ matrix of Boolean variables.

- $ROW_n(M)$ iff M has a row of ones.
- $COL_n(M)$ iff M has a column of ones.

$$f = ROW_n \vee COL_n$$

No OBDD for f of size less than 2^n:

- $\pi = \{z_1, ..., z_{n^2}\}$, $Z = \{z_1, ..., z_n\}$.
- $\tau \in \{0,1\}^Z$: at least n partially tested rows or columns.
- gives 2^n distinct $(M \backslash Z)$-subfunctions.

Consequences: limits of OBDD

$f = ROW_n \vee COL_n$ has no OBDD for f of size less than 2^n:

- OBDD < FBDD because $(s \wedge ROW_n) \vee (\neg s \wedge COL_n)$.
- $APPLY$ does not work on OBDD having different orders.
- DNF is incomparable with OBDD.

Rectangles

Rectangles: A combinatorial object

Origin in communication complexity.

- $r\colon \{0,1\}^X \to \{0,1\}$ is a (Z,Y)-rectangle if $f = g \wedge h$ with:
  - $Z \cup Y = X$,
  - $Z \cap Y = \emptyset$.
  - $g\colon \{0,1\}^Z \to \{0,1\}$, $h\colon \{0,1\}^Y \to \{0,1\}$
- $\tau_1 \vDash g$ and $\tau_2 \vDash h$ iff $\tau_1 \times \tau_2 \vDash r$.
- If $\tau_1 \times \tau_2 \vDash r$ and $\alpha_1 \times \alpha_2 \vDash r$ then $\tau_1 \times \alpha_2 \vDash r$.

Rectangle covers I

Rectangle cover $\{r_1, ..., r_k\}$ of f: $\{0,1\}^X \to \{0,1\}$ is:

- r_i is a (Z_i, Y_i)-rectangle ($Z_i \cup Y_i = X$).
- $f = \bigvee_{i=1}^{k} r_i$
- k is the *size* of the cover

Rectangle covers I

Rectangle cover $\{r_1, ..., r_k\}$ of f: $\{0,1\}^X \to \{0,1\}$ is:

- r_i is a (Z_i, Y_i)-rectangle ($Z_i \cup Y_i = X$).
- $f = \bigvee_{i=1}^{k} r_i$
- k is the *size* of the cover

Example:

- $EVEN(x_1, ..., x_n) = r_1 \vee r_2$ where
  - $r_1 = EVEN(x_1, ..., x_i) \wedge EVEN(x_{i+1}, ..., x_{2n})$
  - $r_2 = ODD(x_1, ..., x_i) \wedge ODD(x_{i+1}, ..., x_{2n})$

- $f = r_0 \vee r_1$ using Shanon expansion
  - $r_0 = \neg x \wedge f[x \mapsto 0]$
  - $r_1 = x \wedge f[x \mapsto 1]$
  - both are $(\{x\}, X \backslash \{x\})$-rectangle.
  - **Not "balanced"**.

Rectangle covers II

Rectangle cover $\{r_1, ..., r_k\}$ of $f\colon \{0,1\}^X \to \{0,1\}$ is:

- *α-balanced* if $\min(|Y_i|, |Z_i|) \geq \alpha|X|$.
- *disjoint* if $r_i \wedge r_j = \bot$.

Rectangle and DNNF

Theorem

(Bova, C., Mengel, Slivovsky) If f is computed by a DNNF C of size s then f has a $(1/3)$-balanced rectangle cover of size at most s.

Observation: If C is deterministic, then we can assume the cover to be *disjoint*.

Covering DNNF with rectangles

Details on board.

DNNF C:

- Each gate g of C defines a $(var(g), \bar{var}(g))$-rectangle C/g.
- Greedily pick gates $g_1, ..., g_k$ where $n/3 \leq |var(g)| \leq 2n/3$.
- $C/g_1, ..., C/g_k$ is a rectangle cover of C.

A lower bound on rectangle covers

Details on board.

For a graph $G = (V, E)$, $VC(G) = \bigwedge_{\{x,y\} \in E} x \vee y$ (vertex cover of G).

Theorem

If G is the $n \times n$ grid, G cannot be covered by less than c^n $(1/3)$-balanced rectangles for some $c > 1$.

Corollaries:

1. $VC(G)$ cannot be represented by polynomial sized DNNFs.
2. CNF and DNNF are not comparable.
3. DNNF do not support negation.

Sauerhoff's function

- $a_n(M) = (\sum_{i=1}^{n} (\sum_{j=1}^{n} m_{i,j}) \mod 2)$: number of rows with even number of 1
- $b_n(M) = (\sum_{j=1}^{n} (\sum_{i=1}^{n} m_{i,j}) \mod 2)$: number of columns with even number of 1
- $R_n(M) = 1$ iff $(a_n(M) \mod 3)$ is 0, and $C_n(M) = 1$ iff $(b_n(M) \mod 3)$ is 0.

$$f_n = R_n \vee C_n$$

Theorem

(Sauerhoff03) Any *disjoint* rectangle cover for f_n has size at least c^n for $c > 1$.

Corollary: d-DNNF < DNNF

Challenging open questions

1. DNF vs d-DNNF.
2. Negation of d-DNNF.

Main hurdle: finding the right candidate function and prove the rectangle lower bounds. Work from communication complexity brings insights.

Conclusion

Wrap up

1. Many ways of representing Boolean functions.
   - Most used in practice: OBDD, decision-DNNF, SDD.
   - Others offer interesting theoretical framework.
2. Two classes of practical compilation algorithms.
   - Top-down (`d4`, `Ganak`…)
   - Bottom-up (`sdd`, `cudd`, `BuDDy`, `TiDiDi`).
   - Strong links with (algebraic) model counting.
3. Tools for lower bounds.
   - Subfunctions for OBDD.
   - Rectangle based techniques.

Uncovered aspects

- Many representation languages (go and check the Circuit Zoo)
  - DD (d-DNNF with “deterministic” negation)
  - Circuits exploiting symmetries when renaming variables/literals.d
- Applications to neighbouring fields:
  - Representing answers of **database queries**
  - **Probabilistic circuits** (representing and learning distributions)
  - **Proof complexity**: how to certify that ϕ has K models? What further syntactic restrictions applies to circuit produced by actual tools?
- Practical aspects and precise heuristics

Going further

- ESSAI26 Summer School by Jean-Marie Lagniez and Johannes Fichte
- Circuit Zoo)
- Check Knowledge Compiler pages
  - d4
  - ganak
  - sdd
  - SharpSAT-TD
  - c2d