Top-Down Knowledge Compilation

Jean-Marie Lagniez & Pierre Marquis*

CRIL, U. Artois & CNRS
Institut Universitaire de France*
France

Overview

SAT Solving

- Introduction
- DP
- DPLL
- Boolean Constraint Propagation (BCP)
- Heuristics
- CDCL

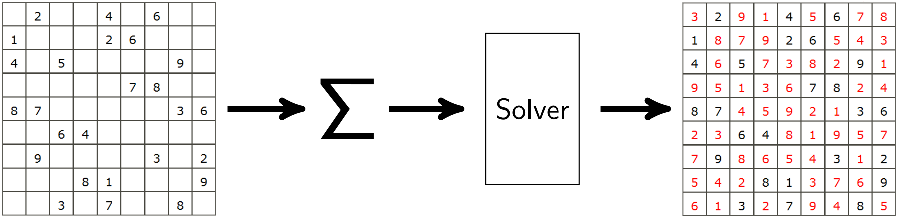
From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

Overview

- SAT Solving
  - Introduction
  - DP
  - DPLL
  - Boolean Constraint Propagation (BCP)
  - Heuristics
  - CDCL
- From SAT Solving to Top-Down Knowledge Compilation
- Heuristics for Decomposition

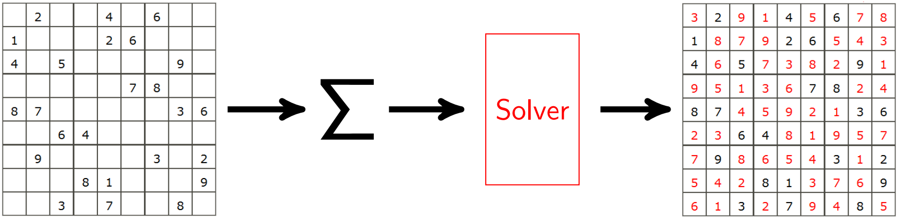
Contraint Programming



Constraint programming: two steps

- **modeling** the problem with a set of constraints Σ
  - $\Rightarrow$ constraints representation with a dedicated formalism: SAT, CSP, PSEUDO, ...
- **solving** the problem
  - $\Rightarrow$ using a constraint-based solver to find a solution

Contraint Programming



Constraint programming: two steps

- **modeling** the problem with a set of constraints Σ
  - $\Rightarrow$ constraints representation with a dedicated formalism: SAT, CSP, PSEUDO, ...
- **solving** the problem
  - $\Rightarrow$ using a constraint-based solver to find a solution

The SAT Problem

$$\Sigma = (\neg a \vee \neg b \vee \neg c) \\ \wedge (a \vee c) \\ \wedge (a \vee b) \\ \wedge (\neg b \vee \neg c)$$

- Propositional variables: a, b, c
- Literals: $a, \neg a$
- Clauses: $a \vee \neg b$ (the constraints)
- CNFformula: Σ
- SAT problem: does there exist an interpretation $\mathcal{I}$ of the variables that satisfies the formula Σ?

The SAT Problem

$$\Sigma = (\neg a \vee \neg b \vee \neg c) \wedge (a \vee c) \wedge (a \vee b) \wedge (\neg b \vee \neg c)$$

a	b	c
$\bot$	$\bot$	$\bot$

- Propositional variables: a, b, c
- Literals: $a, \neg a$
- Clauses: $a \vee \neg b$ (the constraints)
- CNFformula: Σ
- SAT problem: does there exist an interpretation $\mathcal{I}$ of the variables that satisfies the formula Σ?

The SAT Problem

$$\Sigma = (\neg a \vee \neg b \vee \neg c) \wedge (a \vee c) \wedge (a \vee b) \wedge (\neg b \vee \neg c)$$

a	b	c
$\top$	$\bot$	$\bot$

- Propositional variables: a, b, c
- Literals: $a, \neg a$
- Clauses: $a \vee \neg b$ (the constraints)
- CNFformula: Σ
- SAT problem: does there exist an interpretation $\mathcal{I}$ of the variables that satisfies the formula Σ?

The SAT Problem

$$\Sigma = (\neg a \vee \neg b \vee \neg c) \\ \wedge (a \vee c) \\ \wedge (a \vee b) \\ \wedge (\neg b \vee \neg c)$$

a	b	c
$\top$	$\bot$	$\bot$

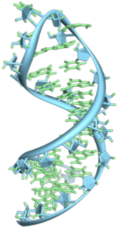
- Propositional variables: a, b, c
- Literals: $a, \neg a$
- Clauses: $a \vee \neg b$ (the constraints)
- CNFformula: Σ
- SAT problem: does there exist an interpretation $\mathcal{I}$ of the variables that satisfies the formula Σ?
- Try all the possibility: illusory!

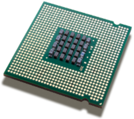
Number of instructions	Time needed
$2^3 = 8$	immediate
$2^{37} = 80 \times 10^9$	1 second
$2^{56} = 8 \times 10^{16}$	$\approx$ 277 hours
$2^{60} = 10^{18}$	166 days
$2^{128} = 340 \times 10^{38}$	$\geq$ 3 billion of years

The Power of SAT

The Power of SAT

- SAT is NP-complete
- Each problem in NP can be reduced in polynomial time to SAT





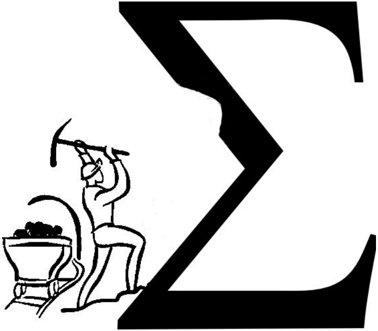
eclipse

```
#include <stdio.h>

int main()
{
    int a[8];
    a[9] = 1;
    printf("Hello, World!\n");
    printf("%d\n", a[9]);

    return 0;
}
```

Several Approaches to SAT Solving

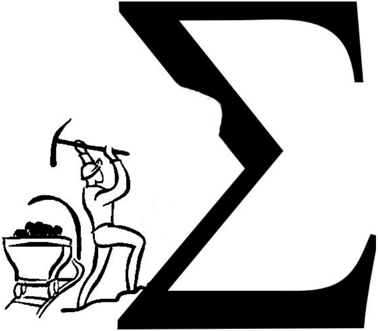
- Complete methods
  - DP algorithm
  - DPLL algorithm
  - CDCL SAT solver
  - ...

- Incomplete methods
  - genetic algorithms
  - ant colony algorithms
  - local search (RL)
  - ...



Several Approaches to SAT Solving

- Complete methods
  - DP algorithm
  - DPLL algorithm
  - CDCL SAT solver
  - ...
- Incomplete methods
  - genetic algorithms
  - ant colony algorithms
  - local search (RL)
  - ...



Overview

SAT Solving

- Introduction
- DP
- DPLL
- Boolean Constraint Propagation (BCP)
- Heuristics
- CDCL

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

Resolution

- Two **clauses** that contain a variable x in **opposite phases** (polarities) imply a new clause that contains **all literals except x and $\neg x$**

$$\frac{x \vee y \vee \neg z \quad \otimes \quad \neg x \vee t \vee u}{y \vee \neg z \vee t \vee u}$$

- Why is this true?

- Making all the resolutions on a variable x in Σ is a way to forget it:

$$\exists x.\Sigma \equiv (\Sigma|x) \vee (\Sigma|\neg x)$$

- Yields a complete proof system for unsatisfiability of CNFs

The Davis-Putnam Algorithm

- Iteratively select a variable x to perform resolution on
  - Consider the resolvents and the ones not containing x
  - Termination:
    - either the empty clause is derived (conclude UNSAT)
    - or all variables have been eliminated

The Davis-Putnam Algorithm

- Iteratively select a variable x to perform resolution on
  - Consider the resolvents and the ones not containing x
  - Termination:
    - either the empty clause is derived (conclude UNSAT)
    - or all variables have been eliminated
- Let Σ s.t. x does not occur in Ψ, α_i and β_i:

$$\begin{aligned}\Sigma =& \Psi \cup \{x \vee \alpha_1, x \vee \alpha_2, \ldots, x \vee \alpha_{n_x}, \neg x \vee \beta_1, \neg x \vee \beta_2, \ldots, \neg x \vee \beta_{n_{\neg x}}\} \\ =& \Psi \cup \{x \vee (\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{n_x}), \neg x \vee (\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_{n_{\neg x}})\}\end{aligned}$$

The Davis-Putnam Algorithm

- Iteratively select a variable x to perform resolution on
  - Consider the resolvents and the ones not containing x
  - Termination:
    - either the empty clause is derived (conclude UNSAT)
    - or all variables have been eliminated
- Let Σ s.t. x does not occur in Ψ, α_i and β_i:

$$\begin{aligned}\Sigma = & \Psi \cup \{x \vee \alpha_1, x \vee \alpha_2, \ldots, x \vee \alpha_{n_x}, \neg x \vee \beta_1, \neg x \vee \beta_2, \ldots, \neg x \vee \beta_{n_{\neg x}}\} \\ = & \Psi \cup \{x \vee (\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{n_x}), \neg x \vee (\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_{n_{\neg x}})\}\end{aligned}$$

- **The truth value of x** does not care, so satisying Σ is equivalent to satisfy:

$$\begin{aligned}\Sigma' = & \Psi \cup \{(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{n_x}) \vee (\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_{n_{\neg x}})\} \\ = & \Psi \cup \{\bigwedge_{i=1}^{n_x} \bigvee_{j=1}^{n_{\neg x}} \alpha_i \vee \beta_j\}\end{aligned}$$

The Davis-Putnam Algorithm

- Iteratively select a variable x to perform resolution on
  - Consider the resolvents and the ones not containing x
  - Termination:
    - either the empty clause is derived (conclude UNSAT)
    - or all variables have been eliminated
- Let Σ s.t. x does not occur in Ψ, α_i and β_i:

$$\begin{aligned}\Sigma =& \Psi \cup \{x \vee \alpha_1, x \vee \alpha_2, \ldots, x \vee \alpha_{n_x}, \neg x \vee \beta_1, \neg x \vee \beta_2, \ldots, \neg x \vee \beta_{n_{\neg x}}\} \\ =& \Psi \cup \{x \vee (\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{n_x}), \neg x \vee (\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_{n_{\neg x}})\}\end{aligned}$$

- **The truth value of** x does not care, so satisying Σ is equivalent to satisfy:

$$\begin{aligned}\Sigma' =& \Psi \cup \{(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{n_x}) \vee (\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_{n_{\neg x}})\} \\ =& \Psi \cup \{\bigwedge_{i=1}^{n_x} \bigvee_{j=1}^{n_{\neg x}} \alpha_i \vee \beta_j\}\end{aligned}$$

- Can generate an exponential number of clauses!

Overview

SAT Solving

- Introduction
- DP
- DPLL
- Boolean Constraint Propagation (BCP)
- Heuristics
- CDCL

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

$$\Sigma = \{\neg a, \neg b \vee c\}$$

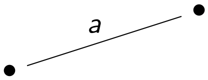
DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

$$\Sigma = \{\neg a, \neg b \vee c\}$$

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

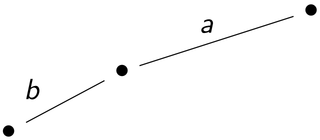
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

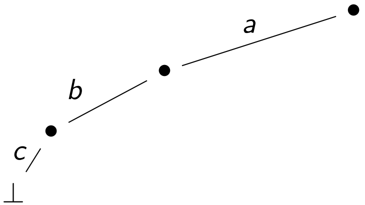
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

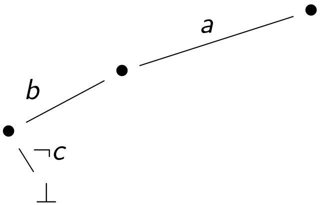
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

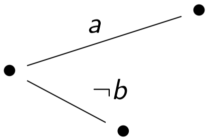
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

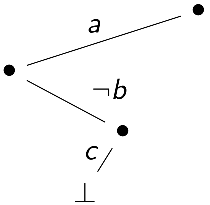
- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

$$\Sigma = \{\neg a, \neg b \vee c\}$$



- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

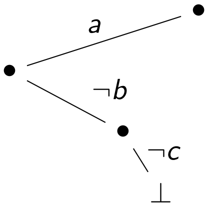
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

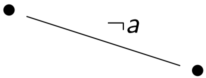
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

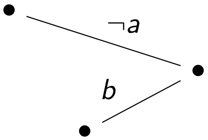
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

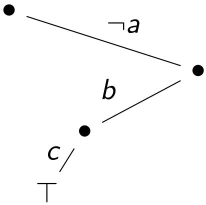
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

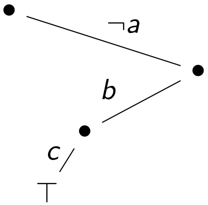
$$\Sigma = \{\neg a, \neg b \vee c\}$$



DPLL-based SAT Solvers

- Perform a **depth-first search** through the space of possible variable assignments
- Stop when a satisfying assignment is found or all possibilities have been tried

$$\Sigma = \{\neg a, \neg b \vee c\}$$



Possible optimizations:

- Skip branches where no satisfying assignments can occur
- Organize the search to maximize the amount of the search space that can be skipped

DPLL algorithm

Algorithm 1: DPLL

Input: Σ a set of clauses

Output: $\top$ if Σ is satisfiable, $\bot$ otherwise

$\Sigma \longleftarrow$ `simplification`(Σ);
if $(\Sigma = \emptyset)$ **then return** $\top$;
if $(\bot \in \Sigma)$ **then return** $\bot$;
$\ell \longleftarrow$ `pickLiteral`(Σ);
return `DPLL`$(\Sigma \wedge \ell)$ or `DPLL`$(\Sigma \wedge \neg\ell)$

- `pickLiteral`: select some variable and assign it a value
- `simplification`: simplify the formula using syntactic rules (unit propagation a.k.a. boolean constraint propagation (BCP))

Overview

SAT Solving
- Introduction
- DP
- DPLL
- Boolean Constraint Propagation (BCP)
- Heuristics
- CDCL

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

Boolean Constraint Propagation (BCP)

- A clause of **size 1** is called **unit clause**
- The literal belonging to a unit clause is called **unit literal**

- The unit propagation process is the **simplification** rule which is used in every DPLL-based SAT solver
- Applying the rule consists in **recursively assigning the unit literals and then simplifying the formula** until a **fixed point** is reached

$$\alpha_1 : a \qquad \alpha_2 : \neg a \vee \neg c \vee \neg b \qquad \alpha_3 : \neg a \vee c \vee b$$
$$\alpha_4 : \neg a \vee b \qquad \alpha_5 : \neg c \vee \neg e \qquad \alpha_6 : b \vee \neg d \vee \neg a$$

- In practice, most of the affectations result from the unit propagation process (more than 90%)
- This explains why a lot of efforts has been devoted to improve this process (watched literals)

Boolean Constraint Propagation (BCP)

- A clause of **size 1** is called **unit clause**
- The literal belonging to a unit clause is called **unit literal**

- The unit propagation process is the **simplification** rule which is used in every DPLL-based SAT solver
- Applying the rule consists in **recursively assigning the unit literals and then simplifying the formula** until a **fixed point** is reached

$\alpha_1 : a$ $\quad \alpha_2 : \neg a \vee \neg c \vee \neg b$ $\quad \alpha_3 : \neg a \vee c \vee b$

$\alpha_4 : \neg a \vee b$ $\quad \alpha_5 : \neg c \vee \neg e$ $\quad \alpha_6 : b \vee \neg d \vee \neg a$

- In practice, most of the affectations result from the unit propagation process (more than 90%)
- This explains why a lot of efforts has been devoted to improve this process (watched literals)

Boolean Constraint Propagation (BCP)

- A clause of **size 1** is called **unit clause**
- The literal belonging to a unit clause is called **unit literal**

- The unit propagation process is the **simplification** rule which is used in every DPLL-based SAT solver
- Applying the rule consists in **recursively assigning the unit literals and then simplifying the formula** until a **fixed point** is reached

$\alpha_1 : a$	$\alpha_2 : \neg a \vee \neg c \vee \neg b$	$\alpha_3 : \neg a \vee c \vee b$
$\alpha_4 : \neg a \vee b$	$\alpha_5 : \neg c \vee \neg e$	$\alpha_6 : b \vee \neg d \vee \neg a$

- In practice, most of the affectations result from the unit propagation process (more than 90%)
- This explains why a lot of efforts has been devoted to improve this process (watched literals)

Boolean Constraint Propagation (BCP)

- A clause of **size 1** is called **unit clause**
- The literal belonging to a unit clause is called **unit literal**

- The unit propagation process is the **simplification** rule which is used in every DPLL-based SAT solver
- Applying the rule consists in **recursively assigning the unit literals and then simplifying the formula** until a **fixed point** is reached

$\alpha_1 : a \qquad \alpha_2 : \neg a \vee \neg c \vee \neg b \qquad \alpha_3 : \neg a \vee c \vee b$

$\alpha_4 : \neg a \vee b \qquad \alpha_5 : \neg c \vee \neg e \qquad \alpha_6 : b \vee \neg d \vee \neg a$

- In practice, most of the affectations result from the unit propagation process (more than 90%)
- This explains why a lot of efforts has been devoted to improve this process (watched literals)

Boolean Constraint Propagation (BCP)

- A clause of **size 1** is called **unit clause**
- The literal belonging to a unit clause is called **unit literal**

- The unit propagation process is the **simplification** rule which is used in every DPLL-based SAT solver
- Applying the rule consists in **recursively assigning the unit literals and then simplifying the formula** until a **fixed point** is reached

$\alpha_1 : a \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee b$

$\alpha_4 : \neg a \vee b \quad \alpha_5 : \neg c \vee \neg e \quad \alpha_6 : b \vee \neg d \vee \neg a$

- In practice, most of the affectations result from the unit propagation process (more than 90%)
- This explains why a lot of efforts has been devoted to improve this process (watched literals)

Boolean Constraint Propagation (BCP)

- A clause of **size 1** is called **unit clause**
- The literal belonging to a unit clause is called **unit literal**

- The unit propagation process is the **simplification** rule which is used in every DPLL-based SAT solver
- Applying the rule consists in **recursively assigning the unit literals and then simplifying the formula** until a **fixed point** is reached

$\alpha_1 : a$	$\alpha_2 : \neg a \vee \neg c \vee \neg b$	$\alpha_3 : \neg a \vee c \vee b$
$\alpha_4 : \neg a \vee b$	$\alpha_5 : \neg c \vee \neg e$	$\alpha_6 : b \vee \neg d \vee \neg a$

- In practice, most of the affectations result from the unit propagation process (more than 90%)
- This explains why a lot of efforts has been devoted to improve this process (watched literals)

Boolean Constraint Propagation (BCP)

- A clause of **size 1** is called **unit clause**
- The literal belonging to a unit clause is called **unit literal**

- The unit propagation process is the **simplification** rule which is used in every DPLL-based SAT solver
- Applying the rule consists in **recursively assigning the unit literals and then simplifying the formula** until a **fixed point** is reached

$\alpha_1 : a \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee b$

$\alpha_4 : \neg a \vee b \quad \alpha_5 : \neg c \vee \neg e \quad \alpha_6 : b \vee \neg d \vee \neg a$

- In practice, most of the affectations result from the unit propagation process (more than 90%)
- This explains why a lot of efforts has been devoted to improve this process (watched literals)

Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

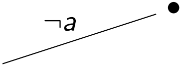
•

Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

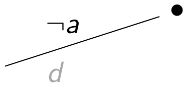


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

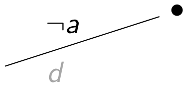


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

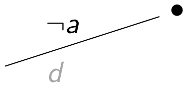


Example

$\alpha_1 : a \vee d \qquad \alpha_2 : a \vee \neg c \vee \neg f \qquad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h \qquad \alpha_5 : \neg c \vee \neg e \vee i \qquad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k \qquad \alpha_8 : e \vee \neg h \vee k \qquad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

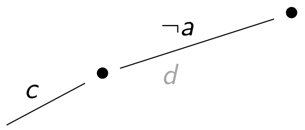


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

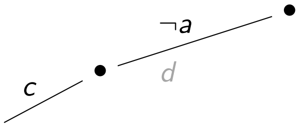


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

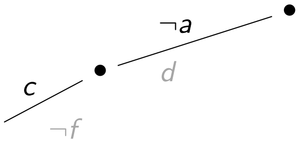


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

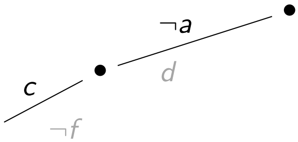


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

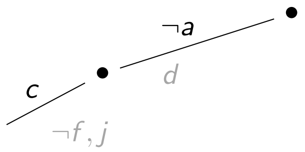


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

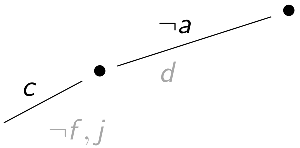


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

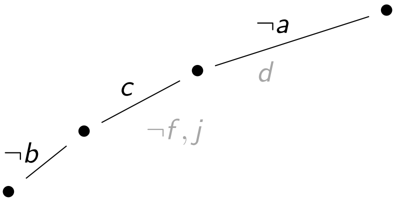


Example

$\alpha_1 : a \lor d$ $\alpha_2 : a \lor \neg c \lor \neg f$ $\alpha_3 : \neg d \lor j \lor f$

$\alpha_4 : b \lor h$ $\alpha_5 : \neg c \lor \neg e \lor i$ $\alpha_6 : \neg i \lor \neg j \lor \neg g$

$\alpha_7 : e \lor \neg k$ $\alpha_8 : e \lor \neg h \lor k$ $\alpha_9 : \neg c \lor \neg e \lor \neg i \lor g$

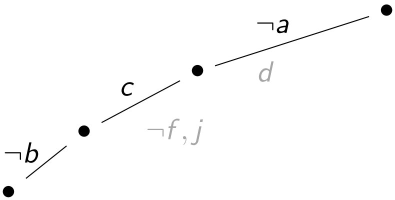


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

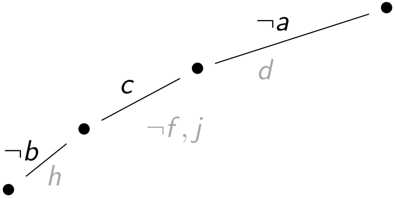


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

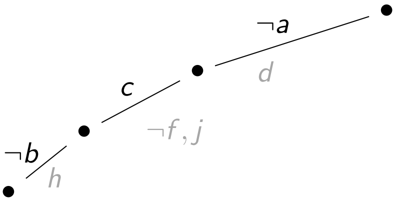


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

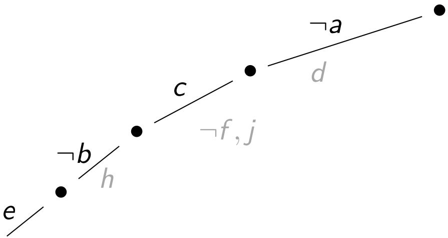


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

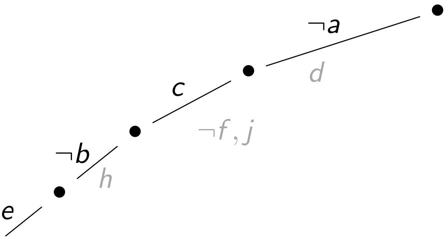


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

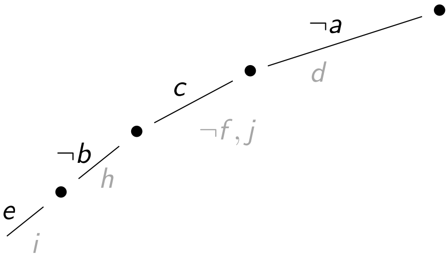


Example

$\alpha_1 : a \vee d \quad \alpha_2 : a \vee \neg c \vee \neg f \quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h \quad \alpha_5 : \neg c \vee \neg e \vee i \quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k \quad \alpha_8 : e \vee \neg h \vee k \quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

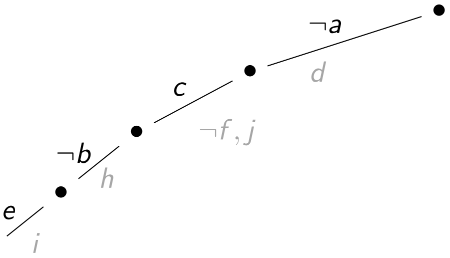


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

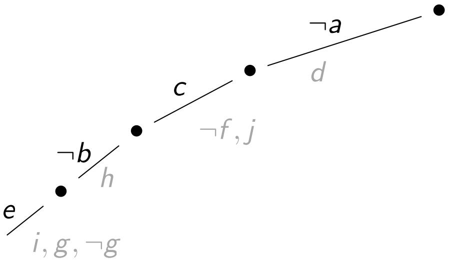


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

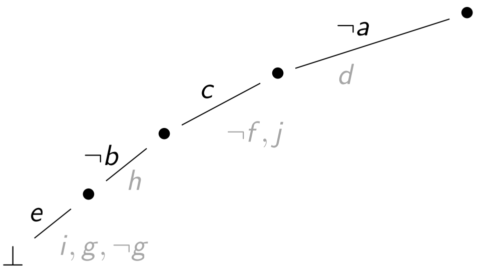


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

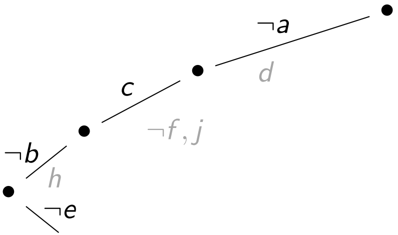


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

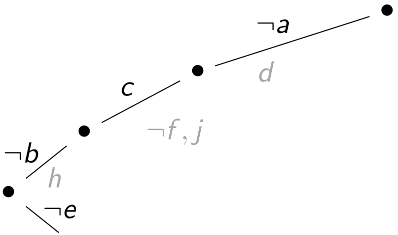


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

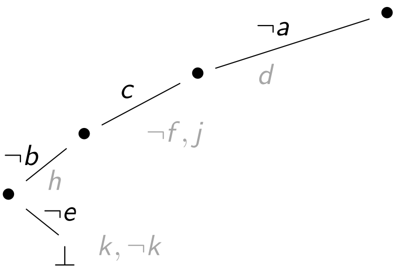


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

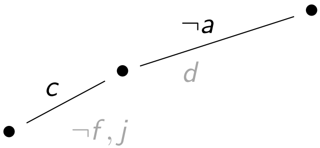


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

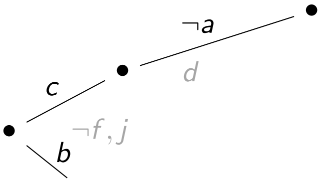


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

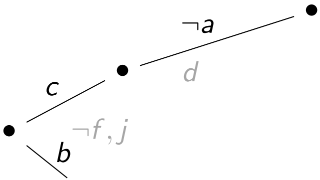


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

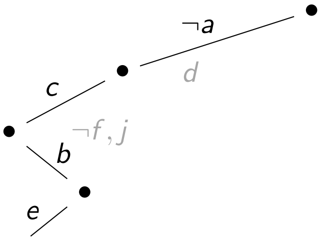


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

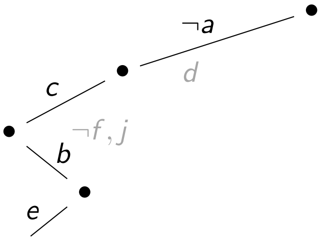


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

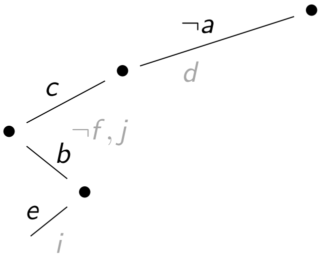


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

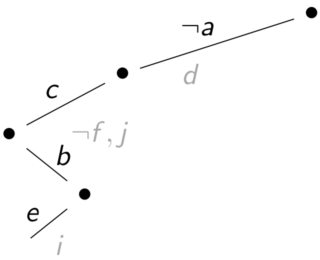


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

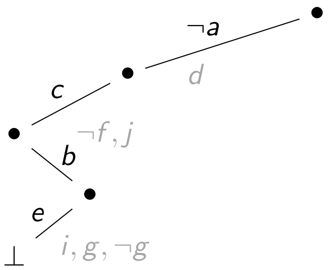


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

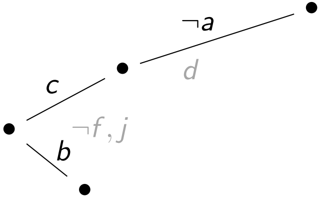


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

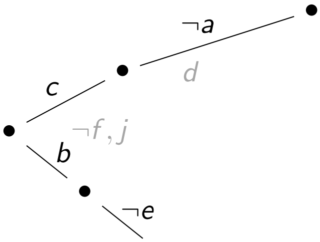


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

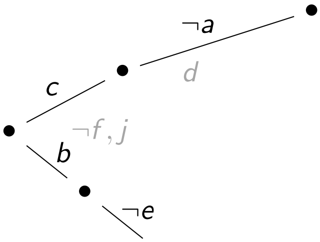


Example

$\alpha_1 : a \lor d \qquad \alpha_2 : a \lor \neg c \lor \neg f \qquad \alpha_3 : \neg d \lor j \lor f$

$\alpha_4 : b \lor h \qquad \alpha_5 : \neg c \lor \neg e \lor i \qquad \alpha_6 : \neg i \lor \neg j \lor \neg g$

$\alpha_7 : e \lor \neg k \qquad \alpha_8 : e \lor \neg h \lor k \qquad \alpha_9 : \neg c \lor \neg e \lor \neg i \lor g$

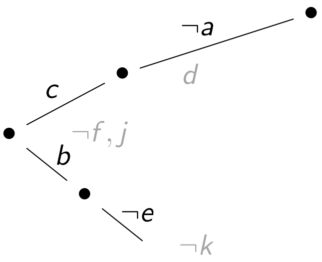


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

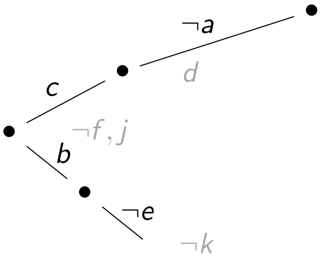


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

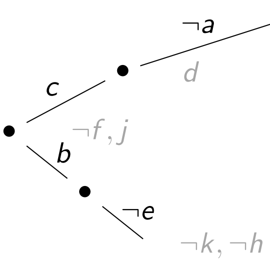


Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

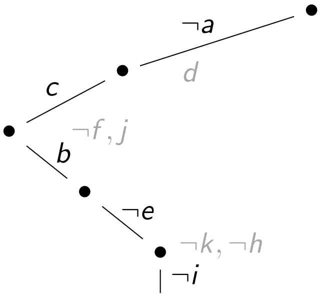


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

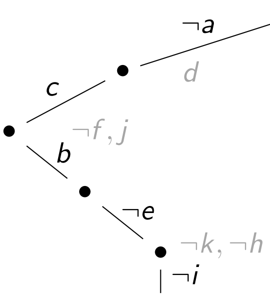


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

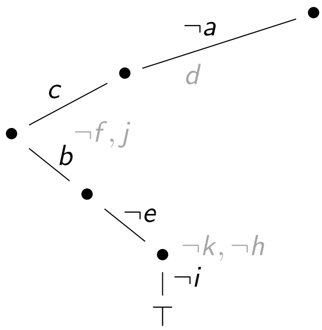


Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



Overview

SAT Solving

- Introduction
- DP
- DPLL
- Boolean Constraint Propagation (BCP)
- Heuristics
- CDCL

From SAT Solving to Top-Down Knowledge Compilation

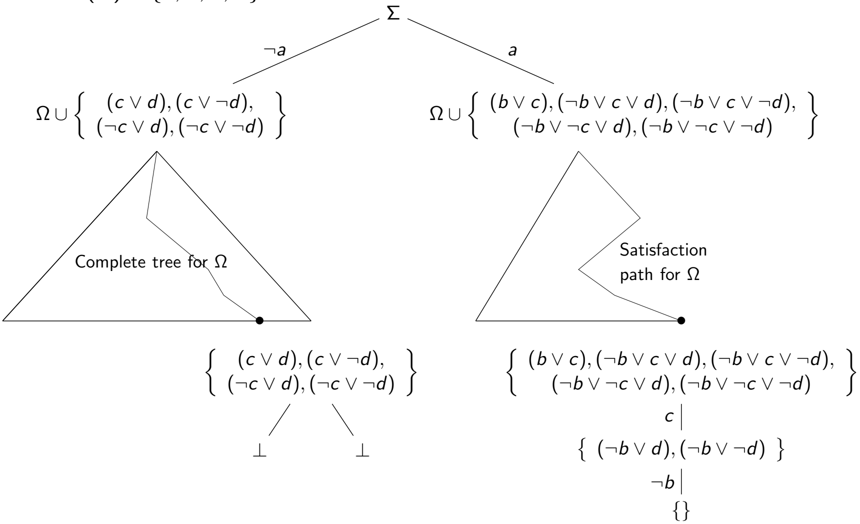
Heuristics for Decomposition

Trashing

$\Sigma = \{a \vee b, \neg a \vee b \vee c, \neg b \vee c \vee d, \neg b \vee c \vee \neg d, \neg b \vee \neg c \vee d, \neg b \vee \neg c \vee \neg d\} \cup \Omega$
with $\texttt{Var}(\Omega) \cap \{a, b, c, d\} = \emptyset$

Trashing

$\Sigma = \{a \vee b, \neg a \vee b \vee c, \neg b \vee c \vee d, \neg b \vee c \vee \neg d, \neg b \vee \neg c \vee d, \neg b \vee \neg c \vee \neg d\} \cup \Omega$
with $\mathtt{Var}(\Omega) \cap \{a, b, c, d\} = \emptyset$



Branching Heuristics

- **Choosing the next variable** to assign and its first polarity is a **decisive** step
- Its **impact** on the size of the search tree explored (so on the CPU time to explore it) is **huge**
- However, choosing the variables that minimize the size of the search tree is hard (NP-**hard**)

- Several branching heuristics have been pointed out
- **Three families**:
  - syntactic approaches
  - look-ahead approaches
  - look-back approaches

Syntactic Branching Heuristics (I)

Aim: choosing a variable that produces a **maximum of unit propagation** or that **satisfies a maximum number of clauses**

- BOHM selects a variable that maximizes, w.r.t. the lexicographic order, the vector $(H_1(x), H_2(x), \ldots, H_n(x))$ with:

  $$H_i(x) = 1 \times max(h_i(x), h_i(\neg x)) + 2 \times min(h_i(x), h_i(\neg x))$$

  where $h_i(x)$ is the number of clauses of size i containing x

Syntactic branching heuristics (II)

- `MOMS` selects a variable with a *Maximum number of Occurrences in Minimum Size Clauses*

  $\mathrm{MOMS}(x, k) = max_k((f^k(x) + f^k(\neg x)) \times 2^k + f^k(x) \times f^k(\neg x))$

  with $f^k(x)$ is the number of unsatisfied clauses of size $\leq k$ containing x

- `JW` is based on a similar idea as `MOMS`

  $$J(\ell) = \sum_{\alpha \in \Sigma | \ell \in \alpha} 2^{-|\alpha|}$$

  `JW-OS` maximizes $J(\ell)$ and `JW-TS` maximizes $J(x) + J(\neg x)$

Look-Ahead Branching Heuristics

<u>Aim</u>: **anticipate** the effect of affecting a variable. Such approaches leads to a "local" breadth-first exploration of the search tree

- BCP uses the **unit propagation process** to decide the next variable to assign. The variable that maximizes the number of unit literals is selected first
- BSH is a *Backbone Search Heuristic*. A variable x that maximizes $\texttt{score}(k, x) = \texttt{bsh}(k, x) \times \texttt{bsh}(k, \neg x)$ is selected first

Algorithm 2: $\texttt{bsh}(i : \text{int}, \ell : \text{literal})$

$\mathcal{B}(\ell) \leftarrow \{\alpha_1, \ldots, \alpha_n\} \subseteq \Sigma$ s.t. $\forall\alpha$, $|\alpha| \leq 3$ and $\ell \in \alpha$;
if $i = 1$ **then**
  return $\sum_{(u \vee v) \in \mathcal{B}(\ell)} (2 \times bin(\neg u) + ter(\neg u)) \times (2 \times bin(\neg v) + ter(\neg v))$
else
  return $\sum_{(u \vee v) \in \mathcal{B}(\ell)} \texttt{bsh(i - 1, } \neg u\texttt{)} \times \texttt{bsh(i - 1, } \neg v\texttt{)}$;
end

Look-Back Branching Heuristics

Aim: **keeping information** from a long phase of search and deduction to **avoid the repetition of the same mistakes** in the future (*nogoods* or variable activity)

- **The weighting of the conflict clauses** is based on the following observation: when a **clause has been proved unsatisfiable** it is important to exploit this piece of **information** for the rest of the search. To do so, it is enough to increase the weight of the variables that conducted to unsatisfiability
- VSIDS associates a counter, called activity, with each variable. When a conflict occurs, the **activity of variables** that are responsible of this failure are **bumped**

Polarity Heuristics

- When a **variable is selected** to be assigned **a truth value** must be chosen. This choice is at least as important as the choice of the variable itself
- Deciding the best way to assign a variable is NP-hard, so **heuristics must be used**:
  - `false` **always assigns to false** (used in MINISAT)
  - `JW` selects the phase of the variable that **maximizes the `jw` function**
  - `occurrence` tries to **maximize the number of satisfied clauses**. The weight of ℓ is given by the number of its occurrences
  - `progress saving` tries to **avoid solving several times the same part of the instance**. To do so, when a variable is assigned during the search, its phase is saved. Then, when a variable has to be assigned again, its phase is chosen as previously

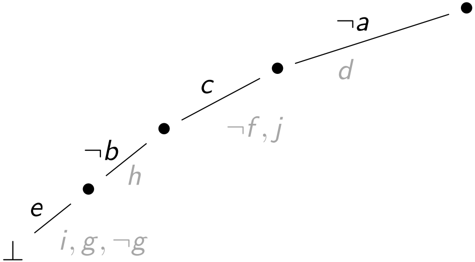
Overview

What is a CDCL SAT Solver?

- **Extend DPLL SAT** solver with:
  - Clause learning and non-chronological backtracking
    - Exploit UIPs
    - Minimize learned clauses
    - Opportunistically delete clauses
  - Can **restart** the current search
  - **Lazy data structures**
    - Watched literals
  - **Conflict-guiding** branching
    - Lightweight branching heuristics
    - Phase saving

A Motivating Example

$\alpha_1 : a \vee d$	$\alpha_2 : a \vee \neg c \vee \neg f$	$\alpha_3 : \neg d \vee j \vee f$
$\alpha_4 : b \vee h$	$\alpha_5 : \neg c \vee \neg e \vee i$	$\alpha_6 : \neg i \vee \neg j \vee \neg g$
$\alpha_7 : e \vee \neg k$	$\alpha_8 : e \vee \neg h \vee k$	$\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

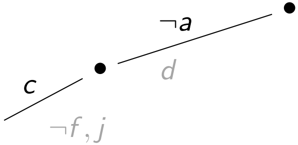


A Motivating Example

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

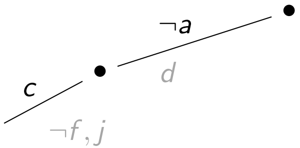


A Motivating Example

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

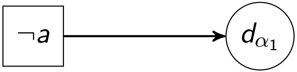


$$\neg e \vee \neg i \vee g \otimes \neg i \vee \neg g = \neg e \vee \neg i$$

$$\neg e \vee \neg i \otimes \neg e \vee i = \neg e$$

- Assignment, BCP
  - heuristic to choose the next variable to assign
  - heuristic to choose its polarity
  - BCP

$\Sigma = \{\alpha_1 : a \vee d\}$



- Conflict analysis and learning
  - **implication graph**
  - **learning**
  - **back-jumping**

Constructing and analyzing an implication graph

Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$
$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$
$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation

Conflict Graph Generation

$\alpha_1 : a \vee d$	$\alpha_2 : a \vee \neg c \vee \neg f$	$\alpha_3 : \neg d \vee j \vee f$
$\alpha_4 : b \vee h$	$\alpha_5 : \neg c \vee \neg e \vee i$	$\alpha_6 : \neg i \vee \neg j \vee \neg g$
$\alpha_7 : e \vee \neg k$	$\alpha_8 : e \vee \neg h \vee k$	$\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation

$\neg a^1$

Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation

$\neg a^1$

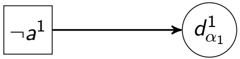
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



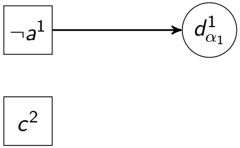
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



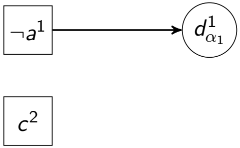
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

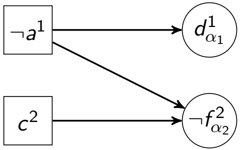
Assignment, Propagation



Conflict Graph Generation

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$
$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$
$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



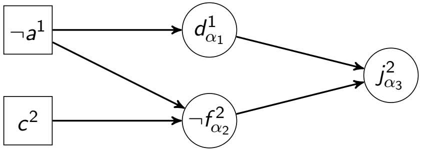
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



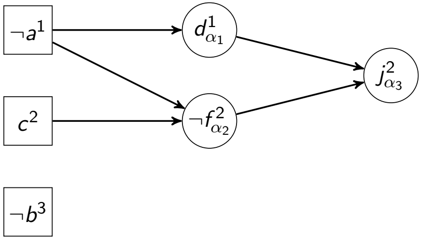
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



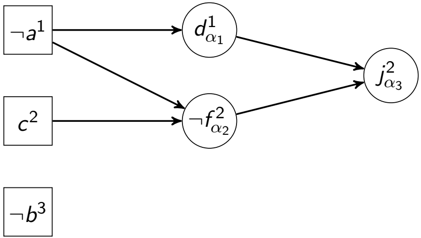
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



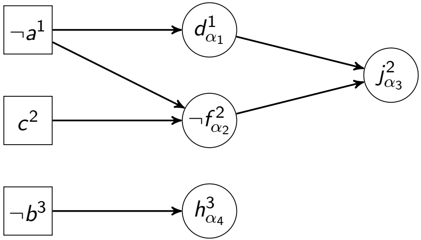
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



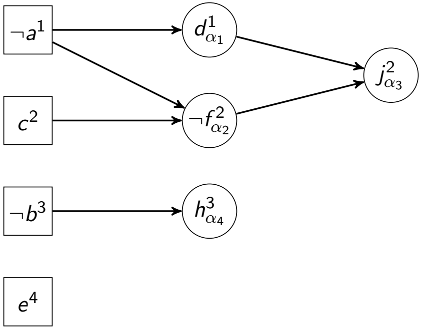
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



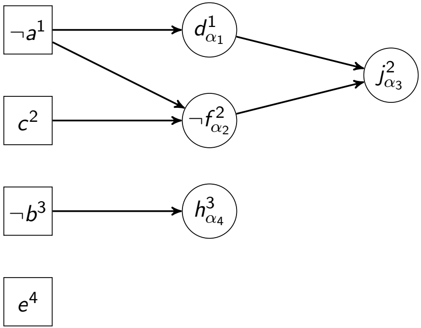
Conflict Graph Generation

$\alpha_1 : a \vee d \qquad \alpha_2 : a \vee \neg c \vee \neg f \qquad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h \qquad \alpha_5 : \neg c \vee \neg e \vee i \qquad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k \qquad \alpha_8 : e \vee \neg h \vee k \qquad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



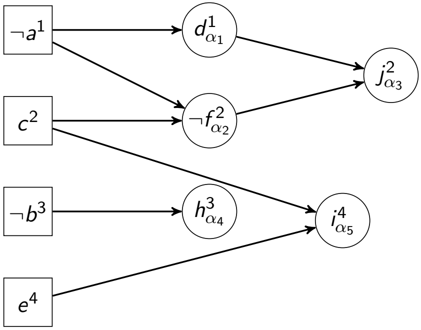
Conflict Graph Generation

$\alpha_1 : a \lor d$ $\quad \alpha_2 : a \lor \neg c \lor \neg f$ $\quad \alpha_3 : \neg d \lor j \lor f$

$\alpha_4 : b \lor h$ $\quad \alpha_5 : \neg c \lor \neg e \lor i$ $\quad \alpha_6 : \neg i \lor \neg j \lor \neg g$

$\alpha_7 : e \lor \neg k$ $\quad \alpha_8 : e \lor \neg h \lor k$ $\quad \alpha_9 : \neg c \lor \neg e \lor \neg i \lor g$

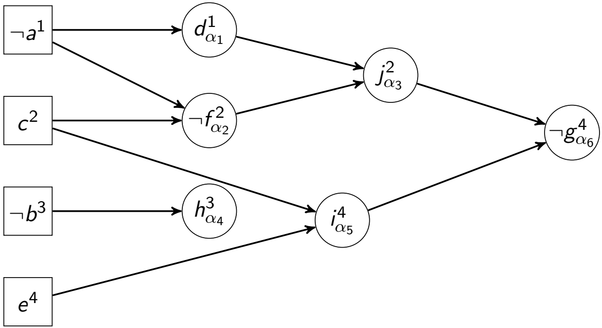
Assignment, Propagation



Conflict Graph Generation

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$
$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$
$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



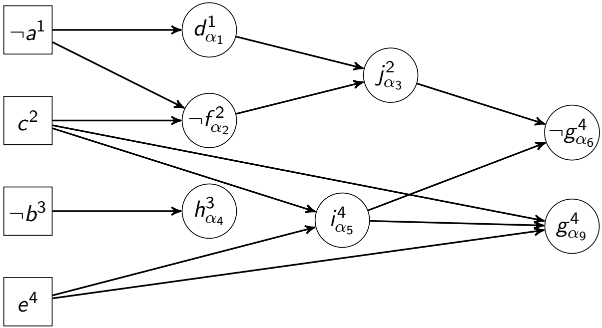
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



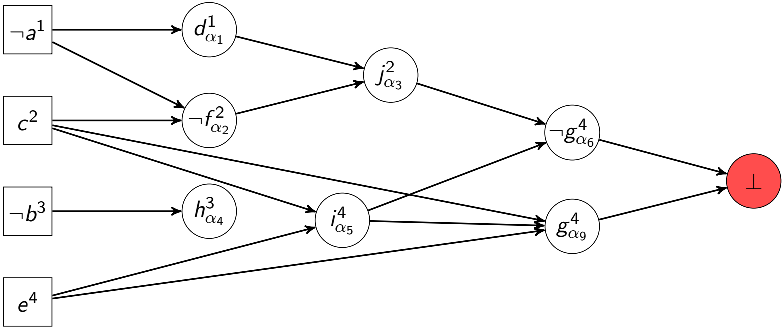
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



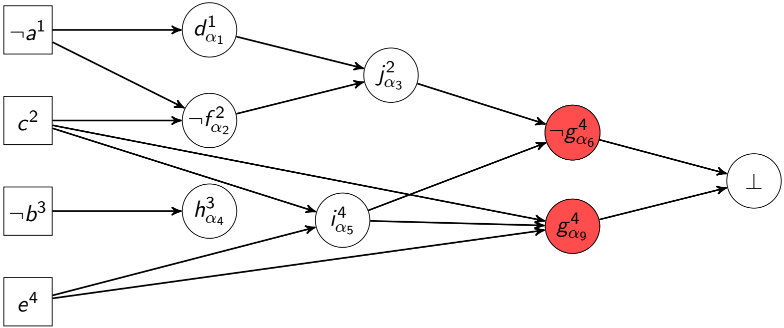
Conflict Graph Generation

$\alpha_1 : a \vee d \quad \alpha_2 : a \vee \neg c \vee \neg f \quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h \quad \alpha_5 : \neg c \vee \neg e \vee i \quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k \quad \alpha_8 : e \vee \neg h \vee k \quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



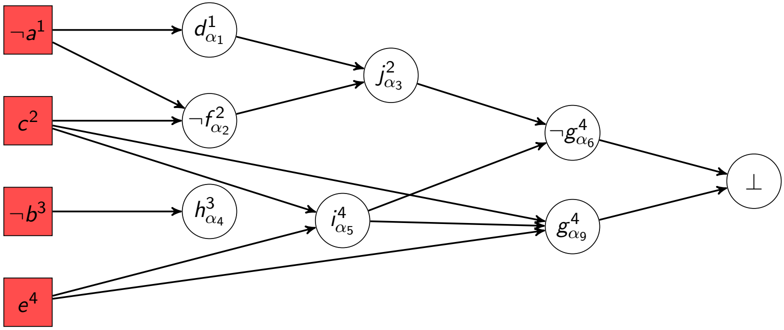
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad\alpha_2 : a \vee \neg c \vee \neg f$ $\quad\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad\alpha_5 : \neg c \vee \neg e \vee i$ $\quad\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad\alpha_8 : e \vee \neg h \vee k$ $\quad\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



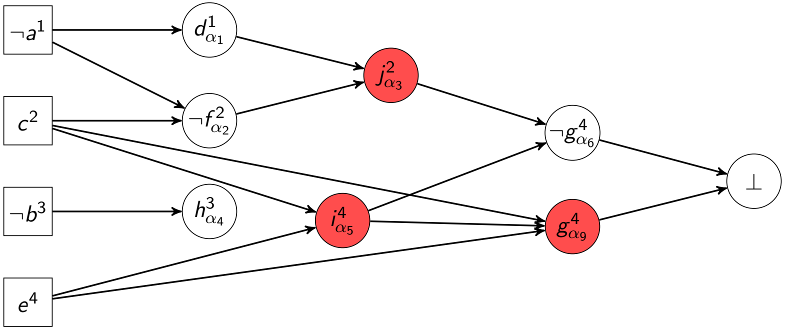
Conflict Graph Generation

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

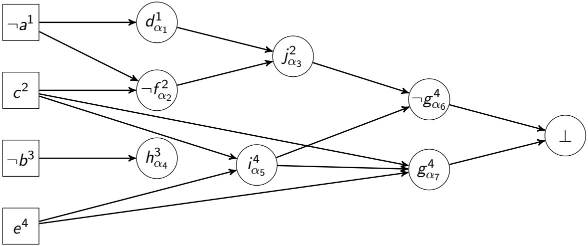
$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

Assignment, Propagation



Conflict Graph Analysis

$\alpha_1 : a \vee d$	$\alpha_2 : a \vee \neg c \vee \neg f$	$\alpha_3 : \neg d \vee j \vee f$
$\alpha_4 : b \vee h$	$\alpha_5 : \neg c \vee \neg e \vee i$	$\alpha_6 : \neg i \vee \neg j \vee \neg g$
$\alpha_7 : e \vee \neg k$	$\alpha_8 : e \vee \neg h \vee k$	$\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

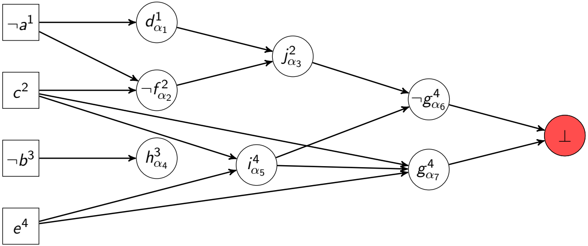


Conflict Graph Analysis

$\alpha_1 : a \vee d$ $\quad$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\quad$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\quad$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad$ $\alpha_8 : e \vee \neg h \vee k$ $\quad$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

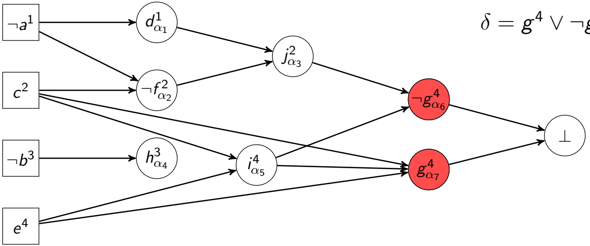


Conflict Graph Analysis

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



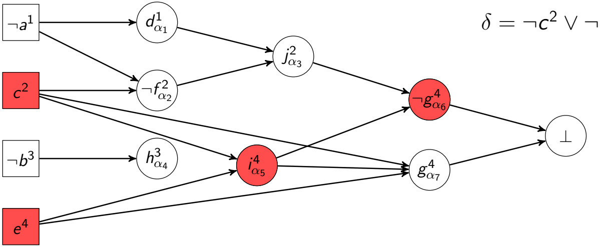
$\delta = g^4 \vee \neg g^4$

Conflict Graph Analysis

$\alpha_1 : a \lor d$ $\quad \alpha_2 : a \lor \neg c \lor \neg f$ $\quad \alpha_3 : \neg d \lor j \lor f$

$\alpha_4 : b \lor h$ $\quad \alpha_5 : \neg c \lor \neg e \lor i$ $\quad \alpha_6 : \neg i \lor \neg j \lor \neg g$

$\alpha_7 : e \lor \neg k$ $\quad \alpha_8 : e \lor \neg h \lor k$ $\quad \alpha_9 : \neg c \lor \neg e \lor \neg i \lor g$



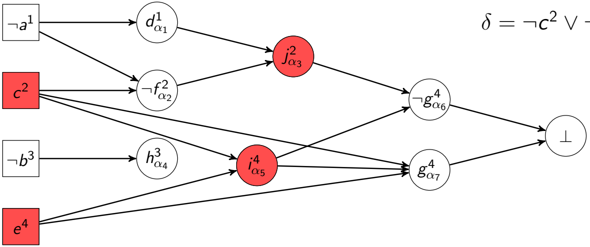
$\delta = \neg c^2 \lor \neg e^4 \lor \neg i^4 \lor g^4$

Conflict Graph Analysis

$\alpha_1 : a \vee d \qquad \alpha_2 : a \vee \neg c \vee \neg f \qquad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h \qquad \alpha_5 : \neg c \vee \neg e \vee i \qquad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k \qquad \alpha_8 : e \vee \neg h \vee k \qquad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



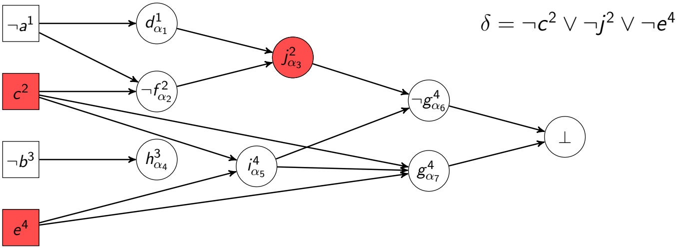
$\delta = \neg c^2 \vee \neg j^2 \vee \neg e^4 \vee \neg i^4$

Conflict Graph Analysis

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

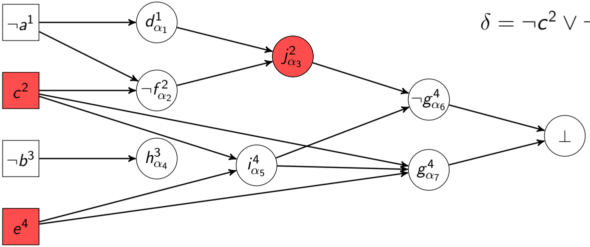


Conflict Graph Analysis

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



$\delta = \neg c^2 \vee \neg j^2 \vee \neg e^4$

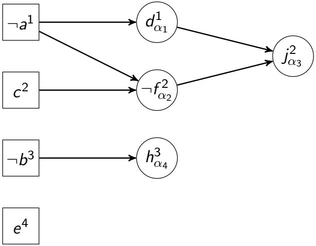
- Stops as soon as the resolvent has a **unique literal from the last decision level** (FUIP)
- δ is added to the CNF(this ensures the completeness of the search)

Back-Jumping

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



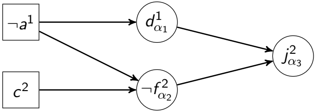
$$\delta_1 = \neg c^2 \vee \neg j^2_{\alpha_3} \vee \neg e^4$$

Back-Jumping

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



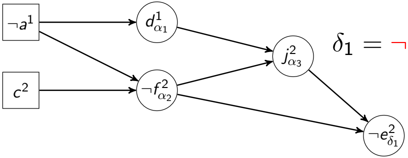
$$\delta_1 = \neg c^2 \vee \neg j^2_{\alpha_3} \vee \neg e$$

Back-Jumping

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



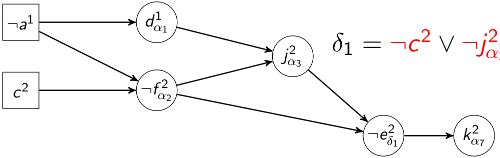
$\delta_1 = \neg c^2 \vee \neg j^2_{\alpha_3} \vee \neg e^2$

Back-Jumping

$\alpha_1 : a \vee d$ $\alpha_2 : a \vee \neg c \vee \neg f$ $\alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\alpha_5 : \neg c \vee \neg e \vee i$ $\alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\alpha_8 : e \vee \neg h \vee k$ $\alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



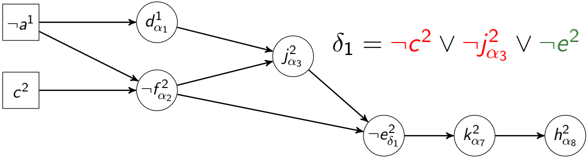
$\delta_1 = \neg c^2 \vee \neg j^2_{\alpha_3} \vee \neg e^2$

Back-Jumping

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



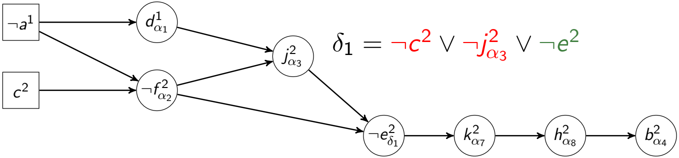
$\delta_1 = \neg c^2 \vee \neg j^2_{\alpha_3} \vee \neg e^2$

Back-Jumping

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



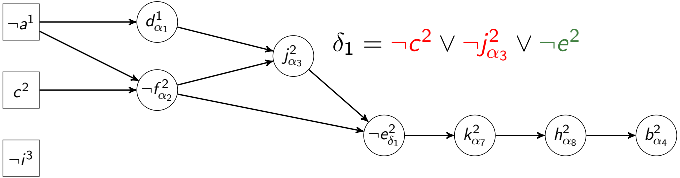
$\delta_1 = \neg c^2 \vee \neg j^2_{\alpha_3} \vee \neg e^2$

Back-Jumping

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$

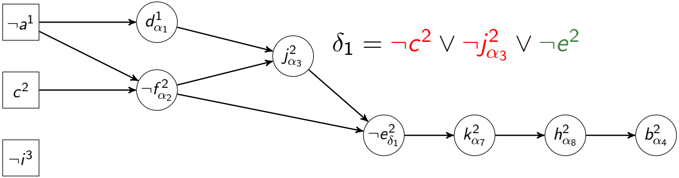


Back-Jumping

$\alpha_1 : a \vee d$ $\quad \alpha_2 : a \vee \neg c \vee \neg f$ $\quad \alpha_3 : \neg d \vee j \vee f$

$\alpha_4 : b \vee h$ $\quad \alpha_5 : \neg c \vee \neg e \vee i$ $\quad \alpha_6 : \neg i \vee \neg j \vee \neg g$

$\alpha_7 : e \vee \neg k$ $\quad \alpha_8 : e \vee \neg h \vee k$ $\quad \alpha_9 : \neg c \vee \neg e \vee \neg i \vee g$



$\delta_1 = \neg c^2 \vee \neg j^2_{\alpha_3} \vee \neg e^2$

SATISFIABILITY PROVED

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg \ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$ $\quad b : \{\alpha_1\}$

$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$	$b : \{\alpha_1\}$	$c : \{\alpha_3\}$
$\neg a : \{\alpha_1, \alpha_3\}$	$\neg b : \{\alpha_2\}$	$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$$\alpha_1 : \neg a \lor b \lor c \quad \alpha_2 : \neg a \lor \neg c \lor \neg b \quad \alpha_3 : \neg a \lor c \lor \neg b$$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$$a : \{\} \qquad b : \{\alpha_1\} \qquad c : \{\alpha_3\}$$
$$\neg a : \{\alpha_1, \alpha_3\} \qquad \neg b : \{\alpha_2\} \qquad \neg c : \{\alpha_2\}$$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$	$b : \{\alpha_1\}$	$c : \{\alpha_3\}$
$\neg a : \{\alpha_1, \alpha_3\}$	$\neg b : \{\alpha_2\}$	$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$	$b : \{\alpha_1\}$	$c : \{\alpha_3\}$
$\neg a : \{\alpha_1, \alpha_3\}$	$\neg b : \{\alpha_2\}$	$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$	$b : \{\alpha_1\}$	$c : \{\alpha_3\}$
$\neg a : \{\alpha_1, \alpha_3\}$	$\neg b : \{\alpha_2\}$	$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$ $\quad b : \{\alpha_1\}$ $\quad c : \{\alpha_3, \alpha_1\}$

$\neg a : \{\alpha_1, \alpha_3\}$ $\quad \neg b : \{\alpha_2\}$ $\quad \neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$	$b : \{\alpha_1\}$	$c : \{\alpha_3, \alpha_1\}$
$\neg a : \{\alpha_1, \alpha_3\}$	$\neg b : \{\alpha_2\}$	$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$	$b : \{\alpha_1\}$	$c : \{\alpha_3, \alpha_1\}$
$\neg a : \{\alpha_1, \alpha_3\}$	$\neg b : \{\alpha_2\}$	$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

$\alpha_1 : \neg a \lor b \lor c \quad \alpha_2 : \neg a \lor \neg c \lor \neg b \quad \alpha_3 : \neg a \lor c \lor \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$	$b : \{\alpha_1\}$	$c : \{\alpha_3, \alpha_1\}$
$\neg a : \{\alpha_1, \alpha_3\}$	$\neg b : \{\alpha_2, \alpha_3\}$	$\neg c : \{\alpha_2\}$

Watched Literals

- BCP is **triggered** when **all but one literal** in a clause is assigned to **false**
- Idea: when two variables are either unassigned or one is assigned to true, no need to do anything
- Checking whether this condition is satisfied is enough

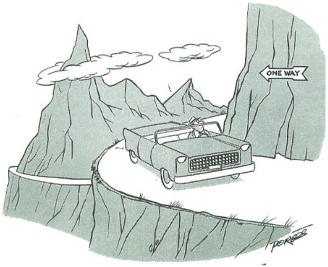
$\alpha_1 : \neg a \vee b \vee c \quad \alpha_2 : \neg a \vee \neg c \vee \neg b \quad \alpha_3 : \neg a \vee c \vee \neg b$

- **Mapping** between watched **literals** and the **clauses** containing them
- When ℓ **is propagated** to true it is enough to consider the **clauses mapped to** $\neg\ell$ and to search for another watched literal
- Let us suppose that a is assigned to true

$a : \{\}$ $\quad b : \{\alpha_1\}$ $\quad c : \{\alpha_3, \alpha_1\}$

$\neg a : \{\}$ $\quad \neg b : \{\alpha_2, \alpha_3\}$ $\quad \neg c : \{\alpha_2\}$

Heavy-Tailed Phenomenon



- Depth-first search procedures often exhibit a remarkable **variability in the time** required to solve the instance
- Heavy-tailed behavior arises from the fact that **wrong branching decisions** may lead to explore an **exponentially large subtree** that contains no solutions
- Restarts is a good mechanism for avoiding such an issue

Restarts

- Often it a good strategy to abandon what you do and restart
  - for satisfiable instances the solver may get stuck in a part of the search space with no solutions
  - for unsatisfiable instances focusing on one part might miss short proofs
  - ⇒ restart the solver once the number of conflicts has reached a given limit
- Avoid to run into the same dead end
  - by randomization (either on the decision variable or its phase)
  - and/or just keep all the learned clauses
- For completeness the limit must be increased dynamically
  - arithmetically, geometrically, Luby, Inner/Outer, Glucose restart

Reducing Learnt Clauses

- CDCL SAT solvers learn clauses at each conflict
- Keeping all these clauses can slow down the BCP process

Reducing Learnt Clauses

- CDCL SAT solvers learn clauses at each conflict
- Keeping all these clauses can slow down the BCP process
- "Useless" learnt clauses are periodically deleted ($t_0, t_1 \dots t_k, \dots$)

α_1	α_2	α_3	α_4	α_5	...	...	α_k	α_n

Reducing Learnt Clauses

- CDCL SAT solvers learn clauses at each conflict
- Keeping all these clauses can slow down the BCP process
- "Useless" learnt clauses are periodically deleted ($t_0, t_1 \ldots t_k, \ldots$)

α_k	α_5	α_2	α_1	α_n	$\ldots$ $\ldots$	α_3	α_4

Reducing Learnt Clauses

- CDCL SAT solvers learn clauses at each conflict
- Keeping all these clauses can slow down the BCP process

- "Useless" learnt clauses are periodically deleted
  $(t_0, t_1 \ldots t_k, \ldots)$

α_k	α_5	α_2	α_1	α_n	$\ldots \qquad \ldots$	α_3	α_4

Reducing Learnt Clauses

- CDCL SAT solvers learn clauses at each conflict
- Keeping all these clauses can slow down the BCP process

- "Useless" learnt clauses are periodically deleted
  $(t_0, t_1 \dots t_k, \dots)$

α_k	α_5	α_2	α_1	α_n

- Deleting too many clauses makes the learning process useless

Reducing Learnt Clauses

- CDCL SAT solvers learn clauses at each conflict
- Keeping all these clauses can slow down the BCP process
- "Useless" learnt clauses are periodically deleted ($t_0, t_1 \dots t_k, \dots$)

α_k | α_5 | α_2 | α_1 | α_n

- Deleting too many clauses makes the learning process useless
- However, identifying whether a clause will be useful in the future is a hard task!

Estimating the Clauses Utility

- <u>The VSIDS measure</u>
  - Keeping clauses that are often – and recently – used in the conflict analysis process
  - Dynamic measure
  - A clause useful in the past will be useful again in the future!

Estimating the Clauses Utility

- The VSIDS measure
  - Keeping clauses that are often – and recently – used in the conflict analysis process
  - Dynamic measure
  - A clause useful in the past will be useful again in the future!
- The LBD measure
  - Gives the number of decision-levels in the learnt clause
  - Static measure
  - Keeping clauses with a small LBD

Estimating the Clauses Utility

- The VSIDS measure
  - Keeping clauses that are often – and recently – used in the conflict analysis process
  - Dynamic measure
  - A clause useful in the past will be useful again in the future!
- The LBD measure
  - Gives the number of decision-levels in the learnt clause
  - Static measure
  - Keeping clauses with a small LBD
- The PSM measure
  - Gives the number of literals assigned to false in the interpretation handled by *Progress Saving*
  - Static measure
  - Keeping clauses with a small PSM

CDCL algorithm

Input: a CNFformula Σ
Output: SAT or UNSAT

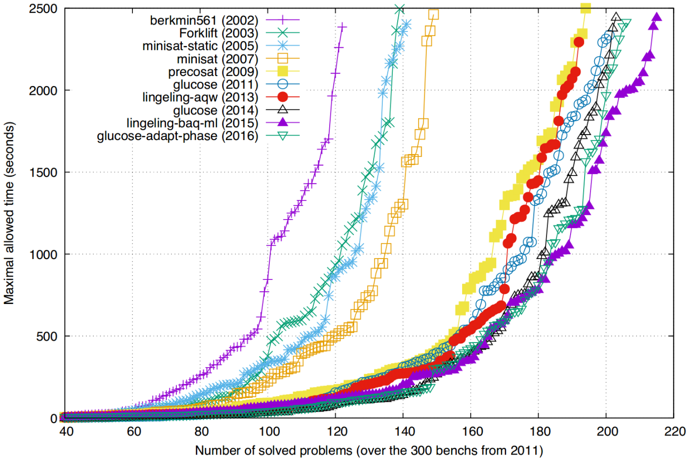
$\Delta = \emptyset$ // learnt clauses database
while (true) **do**
  if (!propagate()) **then**
    if (($c = \mathit{analyzeConflict}()) == \emptyset$) **then** **return** UNSAT ;
    $\Delta = \Delta \cup \{c\}$;
    if (timeToRestart() **then** backtrack to level 0;
    else
      backtrack to the assertion level of c;
  else
    ℓ = decide();
    if (ℓ == null) **then** return SAT ;
    assert ℓ in a new decision level;
    if (timeToReduce()) **then** clean(Δ);

Input: a CNFformula Σ
Output: SAT or UNSAT

```
Δ = ∅ // learnt clauses database
while (true) do
    if (!propagate()) then
        if ((c = analyzeConflict()) == ∅) then return UNSAT ;
        Δ = Δ ∪ {c};
        if (timeToRestart() then backtrack to level 0;
        else
            backtrack to the assertion level of c;
    else
        ℓ = decide();
        if (ℓ == null) then return SAT ;
        assert ℓ in a new decision level;
        if (timeToReduce()) then clean(Δ);
```

About the Performance of SAT Solvers

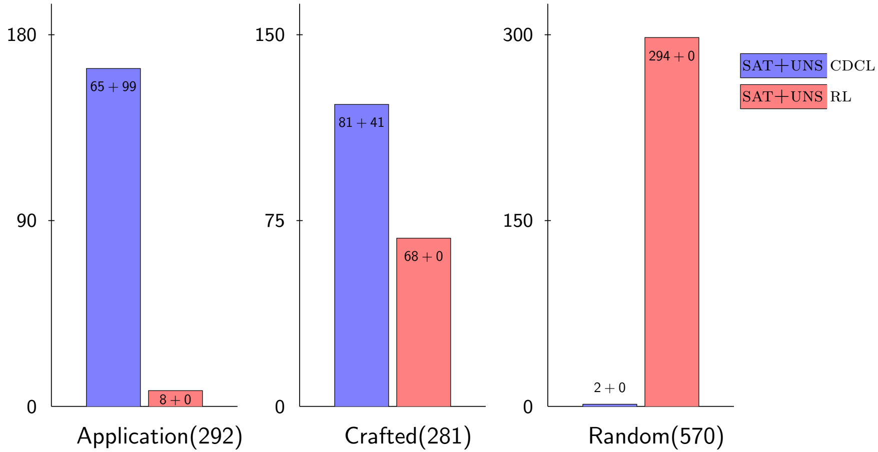
- Since 2001



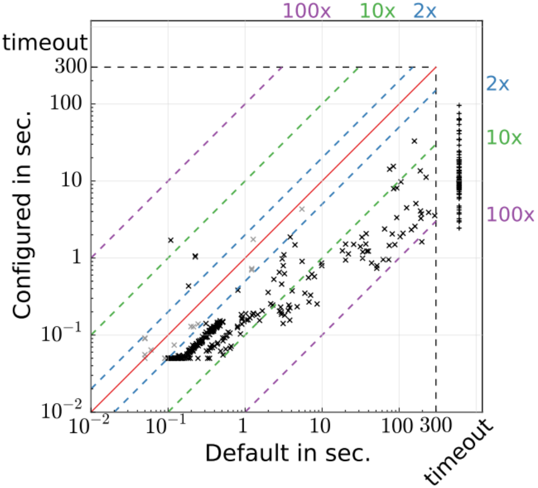
the winners

About the Performance of SAT Solvers

- CDCL SAT solvers are not efficient on all families



- CDCL SAT solvers use several constants impacting their efficiency



Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

- Introduction
- MODS
- DT
- FBDD
- decision-DNNF

Heuristics for Decomposition

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

- Introduction
- MODS
- DT
- FBDD
- decision-DNNF

Heuristics for Decomposition

Motivations

- SAT is NP-complete ⇒ in practice no guarantee to solve the instance within a short delay
- **Compile** the instance into a **representation** from a language $\mathcal{L}$ for which satisfiabily and more difficult issues (e.g. model counting) are **easy**
- Useful when the compilation effort can be balanced by considering sufficiently many queries sharing the same fixed part (pieces of information that are compiled)
- Which $\mathcal{L}$ to choose?
  - **Use the knowledge compilation map!**

KC for Boolean Functions: Queries

Decision or functions problems / properties of languages

- **CO** (consistency)
- **CE** (clause entailment: implicates)
- **VA** (validity)
- **EQ** (equivalence)
- **SE** (sentential entailment)
- **IM** (implicants)
- **CT** (model counting)
- **ME** (model enumeration)

KC for Boolean Functions: Queries

Decision or functions problems / properties of languages

- **CO** (consistency)
- **CE** (clause entailment: implicates)
- **VA** (validity)
- **EQ** (equivalence)
- **SE** (sentential entailment)
- **IM** (implicants)
- **CT** (model counting)
- **ME** (model enumeration)

KC for Boolean Functions: Transformations

Function problems / properties of languages

- **CD** (conditioning)
- $\wedge$ **C** ($\wedge$**BC**) (closure under $\wedge$)
- $\vee$**C** ($\vee$**BC**) (closure under $\vee$)
- $\neg$**C** (closure under $\neg$)
- **FO** (**SFO**) (forgetting)

KC for Boolean Functions: Transformations

Function problems / properties of languages

- **CD** (conditioning)
- $\wedge$**C** ($\wedge$**BC**) (closure under $\wedge$)
- $\vee$**C** ($\vee$**BC**) (closure under $\vee$)
- $\neg$**C** (closure under $\neg$)
- **FO** (**SFO**) (forgetting)

The KC Map for Circ

- $\surd$ means that a polynomial-time algorithm exists for answering this query/making this transformation
- $\circ$ means that a polynomial-time algorithm does not exist for answering this query/making this transformation, unless $\mathrm{P} \neq \mathrm{NP}$

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
Circ	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$

TABLE : Queries

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
Circ	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

TABLE : Transformations

Fragment of the KC Map: Queries

$\mathcal{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
Circ	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
d-DNNF	√	√	√	√	?	○	√	√

TABLE : Queries

Fragment of the KC Map: Transformations

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
Circ	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
CNF	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\circ$	$\surd$	$\circ$
DNF	$\surd$	$\surd$	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\circ$
d-DNNF	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	?

TABLE : Transformations

Succinctness

Succinctness captures **the ability of a language to represent information using little space**

- $\leq_s$ is **polynomial-space translatability**
- $\mathcal{L}_1$ is **at least as succinct as** $\mathcal{L}_2$, denoted $\mathcal{L}_1 \leq_s \mathcal{L}_2$, iff there exists a polynomial p such that for every formula $\alpha \in \mathcal{L}_2$, there exists an equivalent formula $\beta \in \mathcal{L}_1$ where $|\beta| \leq p(|\alpha|)$
- $\leq_s$ is a **pre-order** over the subsets of `Circ`

Succinctness Picture for some Languages

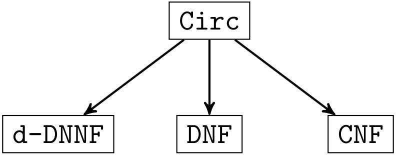


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Overview

SAT Solving

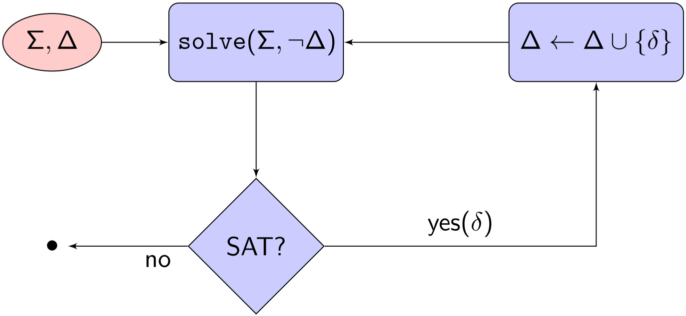
From SAT Solving to Top-Down Knowledge Compilation

- Introduction
- `MODS`
- `DT`
- `FBDD`
- `decision-DNNF`

Heuristics for Decomposition

Enumerate all solutions using a SAT solver (MODS)

- A very simple way to compute the number of models of a propositional formula is to incrementally compute each of them
- To do so, we can easily use a SAT solver



- With Δ initially set to $\emptyset$

KC for DT: queries

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
Circ	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$
CNF	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$
DNF	$\surd$	$\circ$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\surd$
d-DNNF	$\surd$	$\surd$	$\surd$	$\surd$	?	$\circ$	$\surd$	$\surd$
MODS	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

TABLE : Queries

KC for DT: transformations

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
Circ	√	◦	√	√	√	√	√	√
CNF	√	◦	√	√	√	◦	√	◦
DNF	√	√	√	◦	√	√	√	◦
d-DNNF	√	◦	◦	◦	◦	◦	◦	?
MODS	√	◦	√	◦	√	◦	√	◦

TABLE : Transformations

Succinctness

- The size of the representation is given by the number of models of the formula

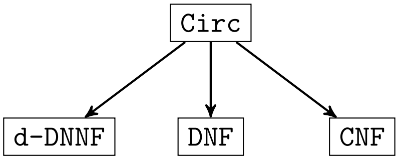


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Succinctness

- The size of the representation is given by the number of models of the formula

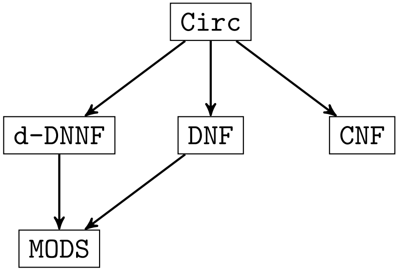


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Is MODS a Good KC Language?

- Can I compile efficiently the following formula into MODS?

$$\Sigma = \bigvee_{i=1}^{n} x_i$$

Is MODS a Good KC Language?

- Can I compile efficiently the following formula into MODS?

$$\Sigma = \bigvee_{i=1}^{n} x_i$$

- No!
- Σ has $2^n - 1$ models

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

- Introduction
- MODS
- DT
- FBDD
- decision-DNNF

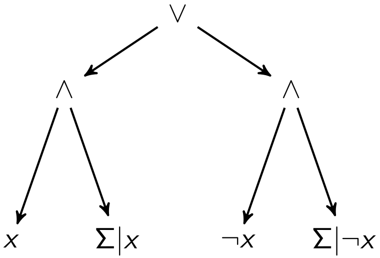
Heuristics for Decomposition

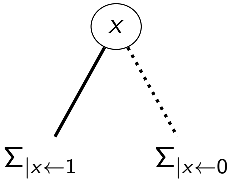
Taking Advantage of the Trace of the Solver

- When a SAT solver is used to solve a CNF instance Σ, it explores the search space of all interpretations until a model is found, if any
- The **same search space** needs to be considered for compiling Σ, except that the process should not stop when a model is found
- Consequently, we can take advantage of the trace of the solver for generating a compiled form

Decision Tree (DT)

- **Shannon Expansion**: $\Sigma \equiv (x \land \Sigma|x) \lor (\neg x \land \Sigma|\neg x)$





- DT is **complete** but **is not succinct**
- **A decision tree** for Σ can be seen as the joined representation of a deterministic DNF of Σ and a deterministic DNF of $\neg\Sigma$

Decision Tree (DT): an Example

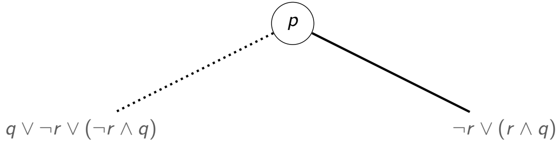
- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$

Decision Tree (DT): an Example

- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$

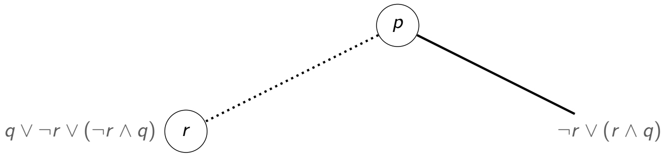
Decision Tree (DT): an Example

- $\Sigma = (q \land \neg p) \lor \neg r \lor (((\neg p \land \neg r) \lor (p \land r)) \land q)$



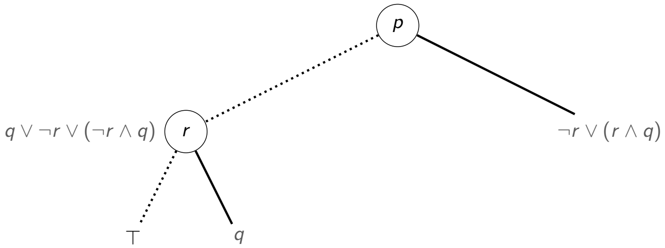
Decision Tree (DT): an Example

- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$



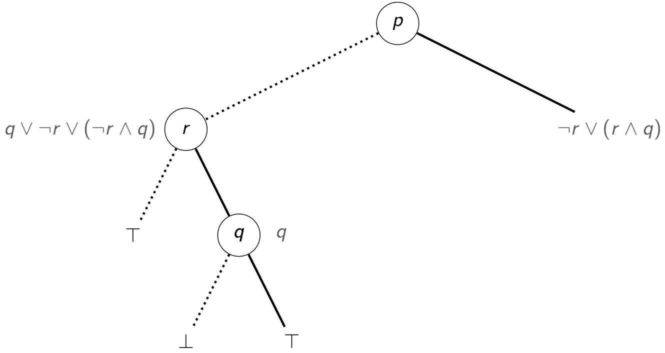
Decision Tree (DT): an Example

- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$



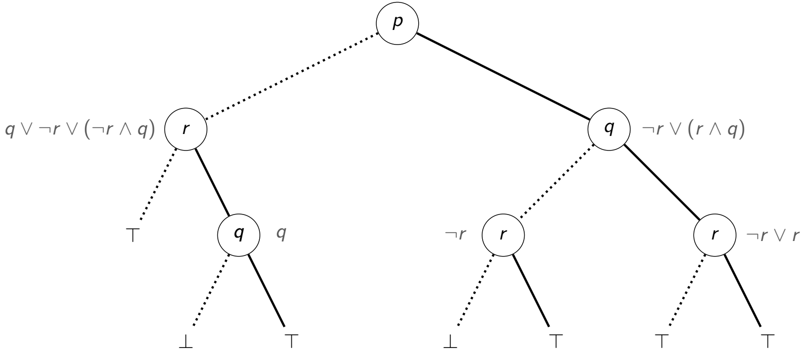
Decision Tree (DT): an Example

- $\Sigma = (q \land \neg p) \lor \neg r \lor (((\neg p \land \neg r) \lor (p \land r)) \land q)$



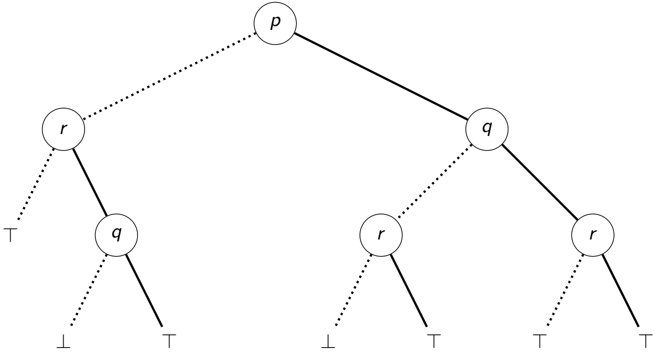
Decision Tree (DT): an Example

- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$



Decision Tree (DT): an Example

- $\Sigma = (q \land \neg p) \lor \neg r \lor (((\neg p \land \neg r) \lor (p \land r)) \land q)$



- The size of the representation is the number of edges of the graph: $|\Sigma| = 25$

KC for DT: Queries

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
Circ	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
d-DNNF	√	√	√	√	?	○	√	√
MODS	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√

TABLE : Queries

KC for DT: Transformations

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
Circ	√	◦	√	√	√	√	√	√
CNF	√	◦	√	√	√	◦	√	◦
DNF	√	√	√	◦	√	√	√	◦
d-DNNF	√	◦	◦	◦	◦	◦	◦	?
MODS	√	◦	√	◦	√	◦	√	◦
DT	√	◦	√	◦	√	◦	√	√

TABLE : Transformations

Succinctness

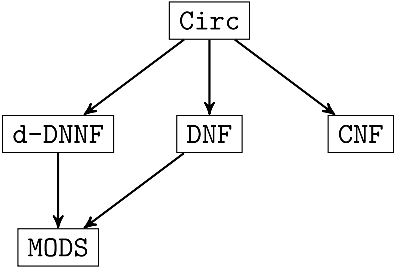


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Succinctness

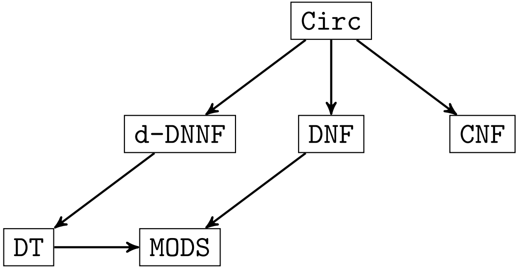


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Is DT a Good KC Language?

- How to represent the following Boolean function into DT?

$$\sum_{i=1}^{n} x_i \equiv 0(mod\ 2)$$

Is DT a Good KC Language?

- How to represent the following Boolean function into DT?

$$\sum_{i=1}^{n} x_i \equiv 0 (mod\ 2)$$

- All the variables must be assigned to be able to decide whether the function evaluates to true
- So all the interpretations must be considered

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

- Introduction
- MODS
- DT
- FBDD
- decision-DNNF

Heuristics for Decomposition

Caching

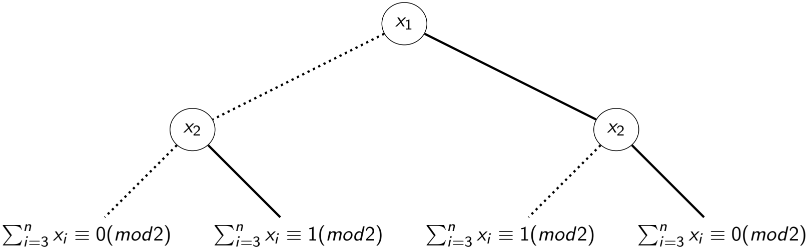
- Caching = sub-circuit sharing
- Let us consider again the previous example:

$$\sum_{i=1}^{n} x_i \equiv 0 (mod 2)$$

Caching

- Caching = sub-circuit sharing
- Let us consider again the previous example:

$$\sum_{i=1}^{n} x_i \equiv 0 (mod 2)$$

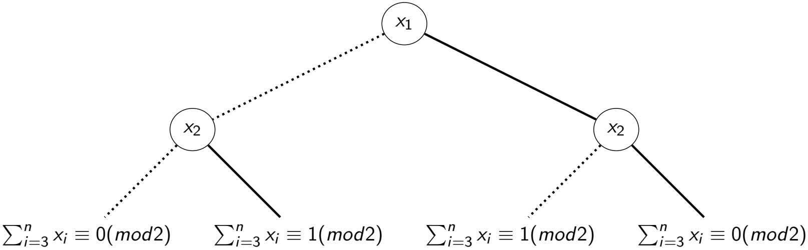


- May the parity function be efficiently compiled using caching?

Caching

- Caching = sub-circuit sharing
- Let us consider again the previous example:

$$\sum_{i=1}^{n} x_i \equiv 0(mod2)$$



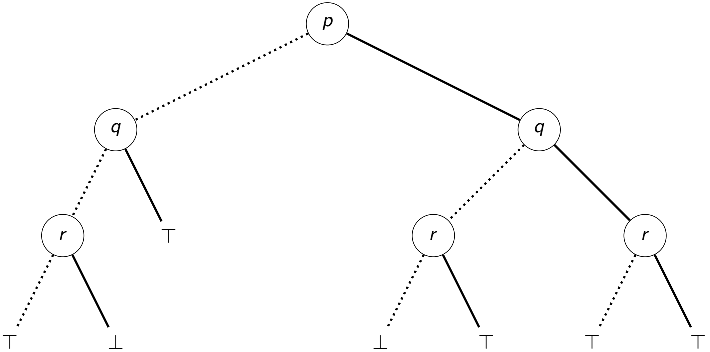
- May the parity function be efficiently compiled using caching? Yes!

Free Binary Decision Diagram (FBDD)

- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$

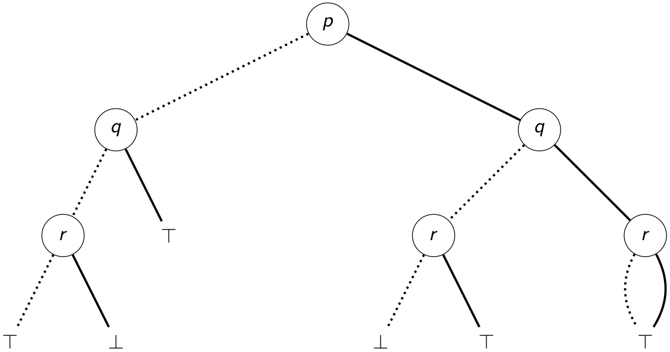
Free Binary Decision Diagram (FBDD)

- $\Sigma = (q \land \neg p) \lor \neg r \lor (((\neg p \land \neg r) \lor (p \land r)) \land q)$



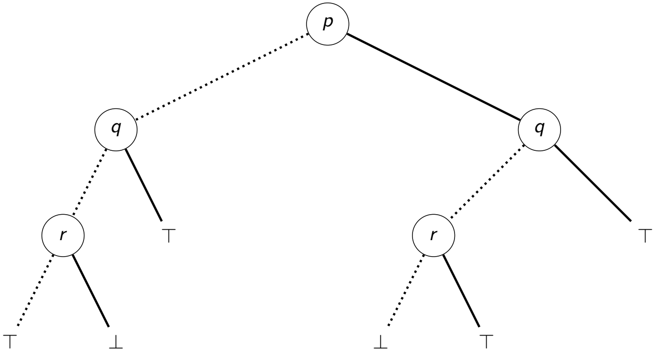
Free Binary Decision Diagram (FBDD)

- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$



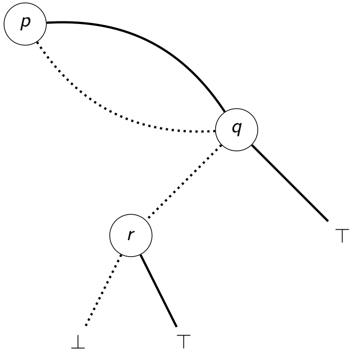
Free Binary Decision Diagram (FBDD)

- $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$



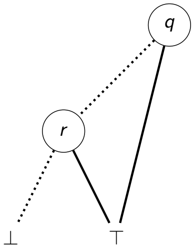
Free Binary Decision Diagram (FBDD)

- $\Sigma = (q \land \neg p) \lor \neg r \lor (((\neg p \land \neg r) \lor (p \land r)) \land q)$



Free Binary Decision Diagram (FBDD)

- $\Sigma = (q \land \neg p) \lor \neg r \lor (((\neg p \land \neg r) \lor (p \land r)) \land q)$



KC for DT: Queries

$\mathcal{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
Circ	∘	∘	∘	∘	∘	∘	∘	∘
CNF	∘	√	∘	√	∘	∘	∘	∘
DNF	√	∘	√	∘	∘	∘	∘	√
d-DNNF	√	√	√	√	?	∘	√	√
MODS	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√
FBDD	√	√	√	√	?	∘	√	√

TABLE : Queries

KC for DT: Transformations

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
Circ	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
CNF	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\circ$	$\surd$	$\circ$
DNF	$\surd$	$\surd$	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\circ$
d-DNNF	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	?
MODS	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$
DT	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
FBDD	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\surd$	$\surd$

TABLE : Transformations

Succinctness

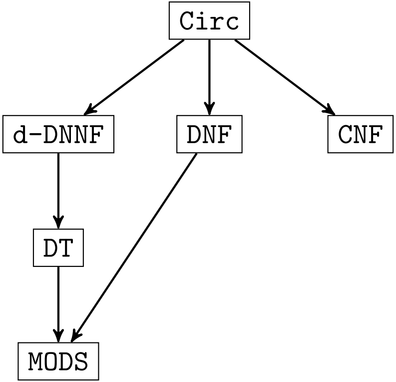


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Succinctness

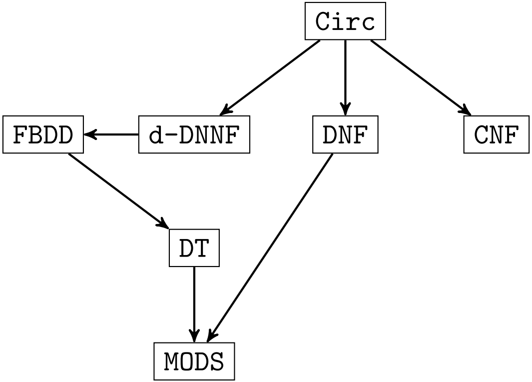


FIGURE : Succinctness : $\mathcal{L}_1 \to \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Can we Turn a DT Compiler into an FBDD Compiler?

- Compiling into DT and then searching for identical sub-circuits to reduce it is impractical!

- Instead one stores in a map pairs $\langle \texttt{CNF}, \texttt{FBDD} \rangle$ consisting of all the CNF considered so far in the search, associated with their corresponding FBDD representation
- At each new decision node, the map is looked up to determine whether the current CNF has already been considered
- If so, one does not need to compile it again!
  - Is it practical to test the equivalence with the CNF formulas present in this map?

Can we Turn a DT Compiler into an FBDD Compiler?

- Compiling into DT and then searching for identical sub-circuits to reduce it is impractical!
- Instead one stores in a map pairs $\langle \mathtt{CNF}, \mathtt{FBDD} \rangle$ consisting of all the CNF considered so far in the search, associated with their corresponding FBDD representation
- At each new decision node, the map is looked up to determine whether the current CNF has already been considered
- If so, one does not need to compile it again!
  - Is it practical to test the equivalence with the CNF formulas present in this map?
  - No! coNP-complete

Can we Turn a DT Compiler into an FBDD Compiler?

- Compiling into DT and then searching for identical sub-circuits to reduce it is impractical!
- Instead one stores in a map pairs $\langle \texttt{CNF}, \texttt{FBDD} \rangle$ consisting of all the CNF considered so far in the search, associated with their corresponding FBDD representation
- At each new decision node, the map is looked up to determine whether the current CNF has already been considered
- If so, one does not need to compile it again!
  - Is it practical to test the equivalence with the CNF formulas present in this map?
  - No! coNP-complete
  - In practice, we replace equivalence by a stronger, yet more easy to decide, relation (identity up to the ordering of the clauses)

Is FBDD a Good KC Language?

- How to compile efficiently the following formula into an FBDD representation?

$$\bigwedge_{i=1}^{n} x_1^i \vee x_2^i \vee \ldots \vee x_n^i$$

Is FBDD a Good KC Language?

- How to compile efficiently the following formula into an FBDD representation?

$$\bigwedge_{i=1}^{n} x_1^i \vee x_2^i \vee \ldots \vee x_n^i$$

- Each clause must be compiled separately
- Branching heuristics for SAT are not suited to this objective!

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

- Introduction
- MODS
- DT
- FBDD
- decision-DNNF

Heuristics for Decomposition

Decomposition

- Let consider again the previous formula:

$$\bigwedge_{i=1}^{n} x_1^i \vee x_2^i \vee \ldots \vee x_n^i$$

- We can observe that the clauses do not share variables
- Can we separately compile the clauses and then aggregate them using an `and` node while offering **model counting**?

Decomposition

- Let consider again the previous formula:

$$\bigwedge_{i=1}^{n} x_1^i \vee x_2^i \vee \ldots \vee x_n^i$$

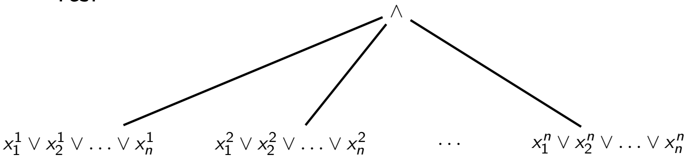
- We can observe that the clauses do not share variables
- Can we separately compile the clauses and then aggregate them using an `and` node while offering **model counting**?
  - Yes!

Decomposition

- Let consider again the previous formula:

$$\bigwedge_{i=1}^{n} x_1^i \vee x_2^i \vee \ldots \vee x_n^i$$

- We can observe that the clauses do not share variables
- Can we separately compile the clauses and then aggregate them using an `and` node while offering **model counting**?
  - Yes!



Decision-d-NNF(decision-DNNF)

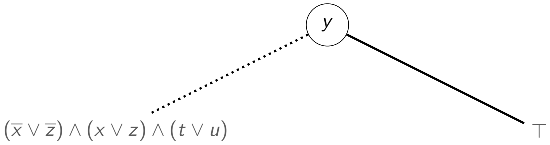
- $\Sigma = (\overline{x} \vee y \vee \overline{z}) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$

Decision-d-NNF(decision-DNNF)

- $\Sigma = (\overline{x} \vee y \vee \overline{z}) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$

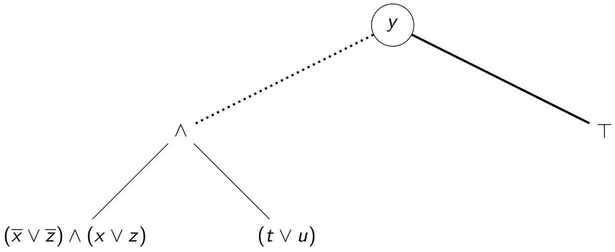
Decision-d-NNF(decision-DNNF)

- $\Sigma = (\overline{x} \vee y \vee \overline{z}) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



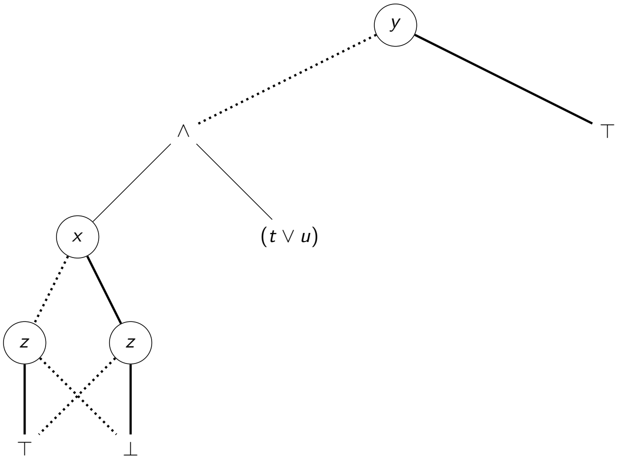
Decision-d-NNF(decision-DNNF)

- $\Sigma = (\overline{x} \vee y \vee \overline{z}) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



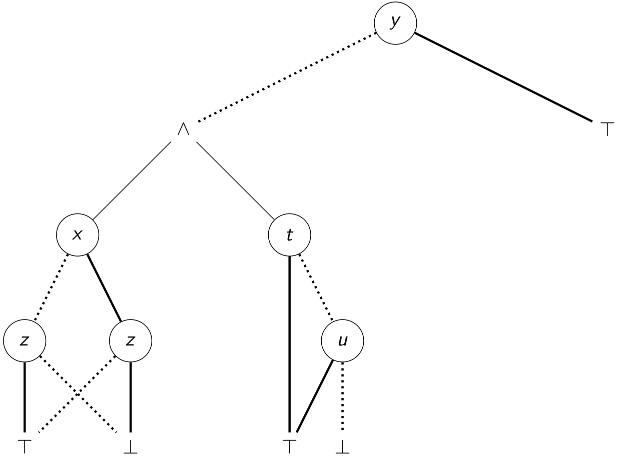
Decision-d-NNF(decision-DNNF)

- $\Sigma = (\overline{x} \vee y \vee \overline{z}) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



Decision-d-NNF(decision-DNNF)

- $\Sigma = (\overline{x} \vee y \vee \overline{z}) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$



KC for DT: queries

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

TABLE : Queries

KC for DT: transformations

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
Circ	√	∘	√	√	√	√	√	√
CNF	√	∘	√	√	√	∘	√	∘
DNF	√	√	√	∘	√	√	√	∘
d-DNNF	√	∘	∘	∘	∘	∘	∘	?
MODS	√	∘	√	∘	√	∘	√	∘
DT	√	∘	√	∘	√	∘	√	√
FBDD	√	∘	∘	∘	∘	∘	√	√
decision-DNNF	√	∘	∘	∘	∘	∘	∘	?

TABLE : Transformations

Succinctness

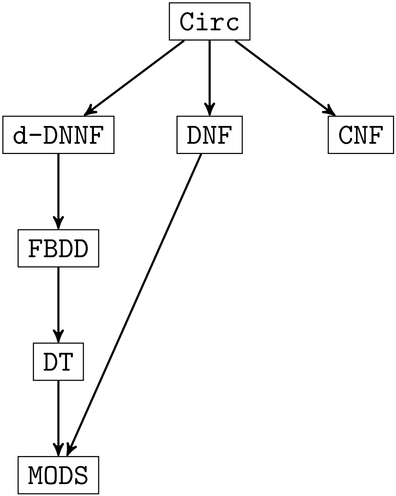


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Succinctness

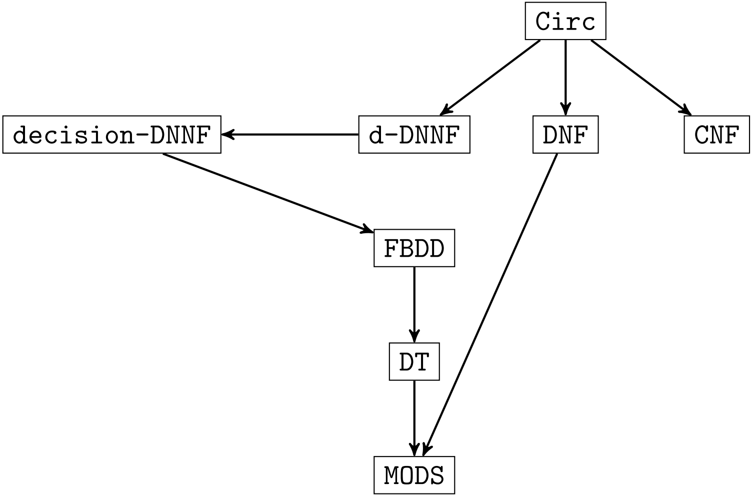


FIGURE : Succinctness : $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that $\mathcal{L}_1 <_s \mathcal{L}_2$

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

- Semantical vs. Syntactic Decompositions
- The Power of Decomposition
- Strategies for Finding Decompositions and Related Compilers

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

- Semantical vs. Syntactic Decompositions
- The Power of Decomposition
- Strategies for Finding Decompositions and Related Compilers

A Key Issue: Decomposition

- The Cartesian approach to problem solving: decomposing a problem into independent subproblems
- Need to design branching heuristics favoring the decomposition of the current CNF formula Σ (i.e., at the current decision node of the search tree) into (at least two) independent CNF formulae Σ_1, Σ_2
- Independence means that no variable is shared between Σ_1 and Σ_2
- If a decomposition of Σ into $\Sigma_1 \wedge \Sigma_2$ is found, a decomposable $\wedge$-node can be generated in the decision-DNNF representation of Σ one wants to build up

Decompositions

Several types of decomposition can be envisioned

- semantical decomposition: Σ_1 and Σ_2 are any CNF such that $\Sigma \equiv (\Sigma_1 \land \Sigma_2)$
- syntactic decomposition: Σ_1 and Σ_2 are subformulae of Σ such that $\Sigma \equiv (\Sigma_1 \land \Sigma_2)$

Every syntactic decomposition of Σ into Σ_1 and Σ_2 also is a semantical one, but not vice-versa

Decompositions: Example

$$\Sigma = (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (c \vee d)$$

- semantical decomposition: Σ is equivalent to

$$\underbrace{(a \vee b)}_{\Sigma_1} \wedge \underbrace{(c \vee d)}_{\Sigma_2}$$

- syntactic decomposition: there is no syntactic decomposition of Σ, but the semantical decomposition above is a syntactic decomposition of the `CNF` $\Sigma' = (a \vee b) \wedge (c \vee d)$ which is equivalent to Σ

Semantical Decomposition

- Guessing Σ_1, Σ_2 and checking that $\Sigma \equiv (\Sigma_1 \wedge \Sigma_2)$ would be prohibitive!
- Fortunately, guessing subsets of variables of Σ is enough
- $\Sigma_1 \wedge \Sigma_2$ is a (nontrivial) semantical decomposition of Σ if and only if there exists an (ordered) bipartition (X_1, X_2) of $Var(\Sigma)$ such that $Var(\Sigma_1) \subseteq X_1$, $Var(\Sigma_2) \subseteq X_2$,

$$\Sigma_1 \equiv \exists X_2.\Sigma, \Sigma_2 \equiv \exists X_1.\Sigma, \text{ and } \Sigma_1 \wedge \Sigma_2 \models \Sigma$$

- Such a bipartition (X_1, X_2) induces a semantical decomposition of Σ

Back to the Example

$$\Sigma = (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (c \vee d)$$

- (X_1, X_2) with $X_1 = \{a, b\}$ and $X_2 = \{c, d\}$ induces a semantical decomposition of Σ
- $\exists X_1.\Sigma \equiv c \vee d$
- $\exists X_2.\Sigma \equiv a \vee b$
- $(a \vee b) \wedge (c \vee d) \models \Sigma$

Semantical Decomposition is Too Expensive

- In order to generate a bipartition (X_1, X_2) inducing a semantical decomposition of Σ, one must be able to decide for each $x \in Var(\Sigma)$ whether x should be put in X_1 or in X_2
- x and y must be put in the same set whenever there exists a prime implicate of Σ which contains them both (as variables)
- Determining whether Σ has a prime implicate containing both x and y is Σ_2^p-complete
- Calling a Σ_2^p oracle at every decision node of the search tree is too much demanding in practice

Semantical Decomposition is Too Expensive

- Once a semantical decomposition (X_1, X_2) has been found, we are not done: variable elimination must be applied to turn each of $\exists X_1.\Sigma$ and $\exists X_2.\Sigma$ into equivalent CNF formulae
- Variable elimination is expensive as well in general

$\Rightarrow$ Look for syntactic decompositions, only

Syntactic Decomposition is Easy to Find

- Use BFS of the primal graph of the current `CNF` Σ to determine whether it has several (disjoint) connected components (feasible in linear time in the size of Σ)
- Σ has a syntactic decomposition if and only if the number of connected components is at least 2
- Back to the example: $\Sigma' = (a \vee b) \wedge (c \vee d)$

Generating a Syntactic Decomposition

- What if Σ has no syntactic decomposition?
- Assigning some variables X_1 of Σ to create such a decomposition
- Let Σ be a CNF. A syntactic decomposition scheme of Σ is a 3-splitting (X_1, X_2, X_3) of $Var(\Sigma)$ such that for every canonical term γ_1 over X_1, the CNF formula $\Sigma \mid \gamma_1$ has a syntactic decomposition $\Sigma_2^{\gamma_1} \wedge \Sigma_3^{\gamma_1}$, where $Var(\Sigma_2^{\gamma_1}) \subseteq X_2$ and $Var(\Sigma_3^{\gamma_1}) \subseteq X_3$
- N.B. 3-splitting = 3-partition except that the sets can be empty

From a Syntactic Decomposition Scheme to a decision-DNNF Representation

If (X_1, X_2, X_3) is a syntactic decomposition scheme of Σ, then

$$\bigvee_{\gamma_1 \text{ canonical term over} X_1} (\gamma_1 \wedge \textit{decision}\text{-}\mathtt{DNNF}(\Sigma_2^{\gamma_1}) \wedge \textit{decision} - \mathtt{DNNF}(\Sigma_3^{\gamma_1}))$$

is a d−DNNF of Σ which corresponds to a decision-DNNF of it (viewing each γ as a path of a decision tree), noted

$\textit{ite}(\gamma_1 \text{ canonical term over } X_1, \textit{decision}\text{-}\mathtt{DNNF}(\Sigma_2^{\gamma_1}) \wedge \textit{decision}\text{-}\mathtt{DNNF}(\Sigma_3^{\gamma_1}))$

Back to the Example

$$\Sigma = (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (c \vee d)$$

- $(X_1 = \{b, c\}, X_2 = \{a\}, X_3 = \{d\})$ is a syntactic decomposition scheme of Σ
- $\Sigma \mid (\bar{b} \wedge \bar{c}) = \underbrace{a}_{\Sigma_2^{(\bar{b}\wedge\bar{c})}} \wedge \underbrace{d}_{\Sigma_3^{(\bar{b}\wedge\bar{c})})}$
- $\Sigma \mid (\bar{b} \wedge c) = \underbrace{a}_{\Sigma_2^{(\bar{b}\wedge c)}} \wedge \underbrace{\top}_{\Sigma_3^{(\bar{b}\wedge c)}}$
- $\Sigma \mid (b \wedge \bar{c}) = \underbrace{\top}_{\Sigma_2^{(b\wedge\bar{c})}} \wedge \underbrace{d}_{\Sigma_3^{(b\wedge\bar{c})}}$
- $\Sigma \mid (b \wedge c) = \underbrace{\top}_{\Sigma_2^{(b\wedge c)}} \wedge \underbrace{\top}_{\Sigma_3^{(b\wedge c)}}$

A Decision-DNNF Representation of Σ

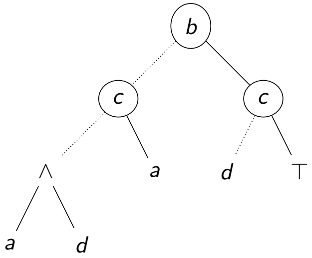
A decision-DNNF

$ite(\gamma_1$ canonical term over X_1, $decision$-DNNF$(\Sigma_2^{\gamma_1}) \wedge decision$-DNNF$(\Sigma_3^{\gamma_1}))$

associated with the syntactic decomposition scheme of Σ given by

$$(X_1 = \{b, c\}, X_2 = \{a\}, X_3 = \{d\})$$

is



Another Syntactic Decomposition Scheme

$$\Sigma = (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (c \vee d)$$

- $(X_1 = \{c\}, X_2 = \{a, b\}, X_3 = \{d\})$ is a syntactic decomposition scheme of Σ
- $\Sigma \mid \bar{c} = \underbrace{(a \vee b)}_{\Sigma_2^{\bar{c}}} \wedge \underbrace{d}_{\Sigma_3^{\bar{c}}}$
- $\Sigma \mid c = \underbrace{(a \vee b)}_{\Sigma_2^{c}} \wedge \underbrace{\top}_{\Sigma_3^{c}}$

Another Decision-DNNF Representation of Σ

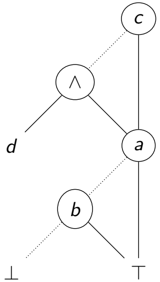
A decision-DNNF

$ite(\gamma_1$ canonical term over X_1, *decision*-DNNF$(\Sigma_2^{\gamma_1}) \wedge$ *decision*-DNNF$(\Sigma_3^{\gamma_1}))$

associated with the syntactic decomposition scheme of Σ given by

$$(X_1 = \{c\}, X_2 = \{a, b\}, X_3 = \{d\})$$

is



Targeting "Small-sized" Decision-DNNF Representations

- Every CNF Σ has a syntactic decomposition scheme: $(Var(\Sigma), \emptyset, \emptyset)$
- This one leads to a compiled representation of Σ as a decision-DNNF which boils down to a decision tree or to an FBDD representation if caching is exploited!
- Better syntactic decomposition schemes (i.e., with decomposable $\wedge$-nodes, leading to "smaller" decision-DNNF compiled forms) are sought for

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition
- Semantical vs. Syntactic Decompositions
- The Power of Decomposition
- Strategies for Finding Decompositions and Related Compilers

The Power of Decomposition

Consider a syntactic decomposition scheme of Σ: (X_1, X_2, X_3) such that $\#(X_i) = x_i$ ($i \in \{1, \ldots, 3\}$)

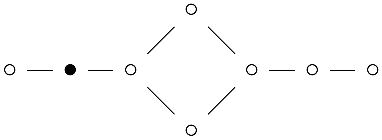
- Suppose that every decision-DNNF representation of Σ has a size which is a fraction k ($0 < k \leq 1$) of the search space of all interpretations explored for generating it (which implies that the corresponding compilation time will be at least as high)
- The size of a decision-DNNF representation of Σ will be $2^{x_1} \times (k \times 2^{x_2} + k \times 2^{x_3})$
- $2^{x_1} \times (k \times 2^{x_2} + k \times 2^{x_3}) < k \times 2^{x_1+x_2+x_3}$ unless $x_2 \leq 2$ and $x_3 \leq 2$

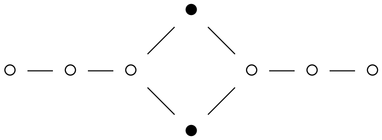
$\Rightarrow$ This explains why introducing decomposable $\wedge$-nodes (and not only decision nodes) in the compiled form is useful

Efficient Syntactic Decomposition Schemes

- The syntactic decomposition scheme (X_1, X_2, X_3) of Σ leads to a decision-DNNF of Σ which is as small as
  - x_1 is small
  - x_2 is close to x_3: $x_2^* = \lfloor \frac{x_2+x_3}{2} \rfloor$ and $x_3^* = \lceil \frac{x_2+x_3}{2} \rceil$ minimize the value of $2^{x_2} + 2^{x_3}$ when the sum $x_2 + x_3$ is fixed
- An efficient syntactic decomposition scheme (X_1, X_2, X_3) of Σ is one minimizing the two criteria (size of the cut set, balance of the decomposition) when possible

The Two Criteria are Antagonistic!





⇒ Trade-offs must be looked for! One typically relaxes the second optimality criterion by asking only that the two disjoint components forming the decomposition have approximately the same cardinal

Complexity of Finding out "Good" Syntactic Decomposition Schemes

- Finding a minimal cut X_1 of the primal graph of Σ can be achieved in polynomial time (e.g. using Stoer-Wagner algorithm which is in time $\mathcal{O}(|V||E| + |V|^2 log_2 |V|))$
- Adding a balance constraint

$$|\#(X_2) - \#(X_3)| \leq \alpha$$

where α is a constant, renders the problem NP-hard

$\Rightarrow$ How to maintain small enough in practice the complexity of finding out "good" syntactic decomposition schemes?

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

- Semantical vs. Syntactic Decompositions
- The Power of Decomposition
- Strategies for Finding Decompositions and Related Compilers
  - Lazy Decomposition: `Dsharp`
  - Global Decomposition: `C2D`
  - Local Decomposition: `D4`

Several Strategies can be Considered

1. Using state-of-the-art branching heuristics for SAT and detecting decompositions in a lazy fashion
2. Relaxing the optimality criteria for syntactic decompositions scheme (use local search techniques for graph partitioning)
3. Avoiding to compute a syntactic decomposition scheme at each decision node
   a. Prior to the compilation of Σ, compute a decomposition tree (dtree) for guiding the decompositions
   b. Use a graph partitioner sparingly during the compilation process on a simplified graph, taking advantage of in-processing techniques (especially literal equivalence) on Σ

- Compilers:
  - The `Dsharp` compiler is based on 1.
  - The `C2D` compiler is based on 2., and 3.a.
  - The `D4` compiler is based on 2., and 3.b.

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

- Semantical vs. Syntactic Decompositions
- The Power of Decomposition
- Strategies for Finding Decompositions and Related Compilers
  - Lazy Decomposition: `Dsharp`
  - Global Decomposition: `C2D`
  - Local Decomposition: `D4`

The VSADS Branching Heuristics

- In the SAT case and in the compilation case, the smaller the search tree the better
- To detect conflicts as soon as possible, SAT solvers take advantage of look-back branching heuristics
- Hence it makes sense to use such heuristics in the compilation case
- VSADS is a look-back branching heuristics that is based on VSIDS and the number of occurrences of the variables in the clauses

A Pseudo-Code of Dsharp

Algorithm 3: Dsharp(Σ)

input : a CNF formula Σ

output: the root node N of a decision-DNNF representation of Σ

$S \leftarrow \mathsf{solve}(\Sigma)$;
if $S = \{\emptyset\}$ then return $\mathsf{leaf}(\bot)$;
if $Var(\Sigma) = \emptyset$ then return $\mathsf{aNode}(S, [\mathsf{leaf}(\top)])$;
if $\mathsf{cache}(\Sigma) \neq \mathsf{nil}$ then return $\mathsf{aNode}(S, [\mathsf{cache}(\Sigma)])$;
$comps \leftarrow \mathsf{connectedComponents}(\Sigma)$;
$LN_d \leftarrow []$;
foreach $c \in comps$ do
  $v \leftarrow \mathsf{VSADS}(Var(c))$;
  $N_d \leftarrow ite(v, \mathtt{Dsharp}(c|\neg v), \mathtt{Dsharp}(c|v))$;
  $LN_d \leftarrow \mathsf{add}(N_d, LN_d)$;
$N_\wedge \leftarrow \mathsf{aNode}(S, LN_d)$;
$\mathsf{cache}(\Sigma) \leftarrow N_\wedge$;
return $N_\wedge$

Overview

SAT Solving

From SAT Solving to Top-Down Knowledge Compilation

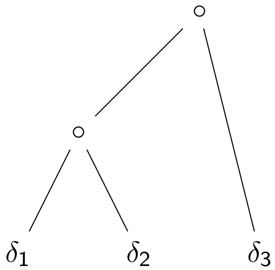
Heuristics for Decomposition

- Semantical vs. Syntactic Decompositions
- The Power of Decomposition
- Strategies for Finding Decompositions and Related Compilers
  - Lazy Decomposition: `Dsharp`
  - Global Decomposition: `C2D`
  - Local Decomposition: `D4`

Decomposition Trees

A decomposition tree (dtree) for a CNF Σ is a full binary tree, with leaves in one-to-one correspondance with the clauses of Σ

$$\Sigma = \underbrace{(a \lor b \lor c)}_{\delta_1} \land \underbrace{(a \lor b \lor \bar{c})}_{\delta_2} \land \underbrace{(c \lor d)}_{\delta_3}$$



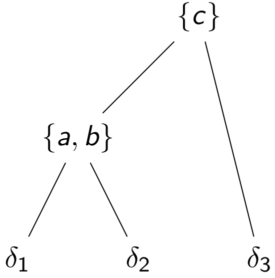
Cutsets

Each internal node of a dtree is associated with a cutset

$$\Sigma = \underbrace{(a \vee b \vee c)}_{\delta_1} \wedge \underbrace{(a \vee b \vee \bar{c})}_{\delta_2} \wedge \underbrace{(c \vee d)}_{\delta_3}$$

For every internal node N, let N^l and N^r its two children

- $Var(N) = Var(N^l) \cup Var(N^r)$
- $Cutset(N) = (Var(N^l) \cap Var(N^r)) \setminus AncCutset(N)$
- $AncCutset(N) = \bigcup_{N' \text{ ancestor of } N} Cutset(N')$



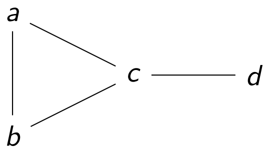
Decomposition Trees

Dtrees can be computed in various ways:

- in a bottom-up way, starting with an elimination ordering (i.e., a strict, total ordering $<$ over $Var(\Sigma)$)
- several heuristics exist for determining an elimination ordering leading to "good" decompositions
  - min-degree: order the variables of Σ in an ascending way w.r.t. their incidence degree in the primal graph of Σ
  - min-fill: order the variables of Σ in an ascending way w.r.t. their number of neighbors which are not pairwise connected in the primal graph of Σ
- in a top-down way, using a graph partitioner

Back to the Example: Heuristics

$$\Sigma = \underbrace{(a \vee b \vee c)}_{\delta_1} \wedge \underbrace{(a \vee b \vee \bar{c})}_{\delta_2} \wedge \underbrace{(c \vee d)}_{\delta_3}$$



Min-degree and min-fill leads to the same ordering for this example:

$$d < a < b < c$$

Algorithm 4: dtree-bu(Σ, $<$)

input : a `CNF` formula Σ and an elimination ordering $<$ over $Var(\Sigma)$

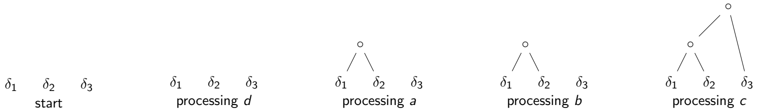
output: a dtree dt for Σ

F $\leftarrow \{\delta_i \in \Sigma\}$;
Var $\leftarrow Var(\Sigma)$;
while Var $\neq \emptyset$ do
    v $\leftarrow$ *head*(Var, $<$);
    gather every dtree of F with a leaf containing v into a single dtree;
    remove v from Var
Gather every dtree of F into a single dtree dt;
return dt

Back to the Example

$$\Sigma = \underbrace{(a \vee b \vee c)}_{\delta_1} \wedge \underbrace{(a \vee b \vee \bar{c})}_{\delta_2} \wedge \underbrace{(c \vee d)}_{\delta_3}$$

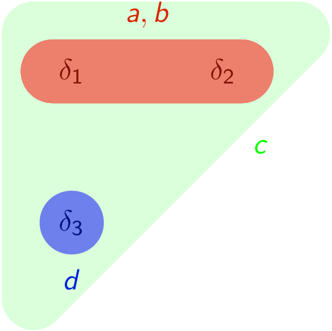
$$d < a < b < c$$



In a Top-Down Way

One exploits a graph partitioner for finding a cutset in the dual hypergraph of Σ (if possible, a cutset of "small size" leading to a balanced decomposition)

$$\Sigma = \underbrace{(a \vee b \vee c)}_{\delta_1} \wedge \underbrace{(a \vee b \vee \bar{c})}_{\delta_2} \wedge \underbrace{(c \vee d)}_{\delta_3}$$



In a Top-Down Way: A Pseudo-Code of `dtree-td`

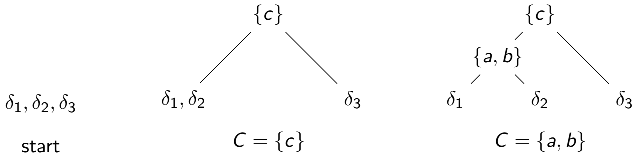
Algorithm 5: dtree-td(Σ)

input : a CNF formula Σ
output: the root N of a dtree for Σ

```
1 if Σ has a decomposition Σ1 ∧ Σ2 then
      N ← node(∅, dtree-td(Σ1), dtree-td(Σ2))
2 else
      while there exist two distinct clauses connected by a hyperedge
      in the dual hypergraph of Σ do
          C ← HGP(Σ);
          Σ1 ← one connected component of Σ simplified by
          removing from its clauses all the variables from C;
          Σ2 ← the union of the other connected components of Σ
          simplified by removing from its clauses all the variables
          from C;
          N ← node(C, dtree-td(Σ1), dtree-td(Σ2));
3 return N
```

Back to the Example

$$\Sigma = \underbrace{(a \vee b \vee c)}_{\delta_1} \wedge \underbrace{(a \vee b \vee \bar{c})}_{\delta_2} \wedge \underbrace{(c \vee d)}_{\delta_3}$$



A Pseudo-Code of C2D

Algorithm 6: C2D(Σ, N)

input : a CNF formula Σ and the root N of a dtree dt for Σ
output: the root M of a decision-DNNF representation of Σ

S $\leftarrow$ solve(Σ);
if S $= \{\emptyset\}$ then return leaf($\bot$);
if $Var(\Sigma) = \emptyset$ then return aNode(S, [leaf($\top$)]);
if cache(Σ) $\neq$ nil then return aNode(S, [cache(Σ)]);
if N reduces to a leaf node labelled by δ then
    return a decision-DNNF representation of δ
else
    $C \leftarrow label(N)$;
    $M \leftarrow ite(\gamma_1$ canonical term over C, C2D($\Sigma \mid \gamma_1, N^2$), C2D($\Sigma \mid \gamma_1, N^3$));
    /* $(C, Var(N^2), Var(N^3))$ is by construction a syntactic decomposition scheme of Σ */
    cache(Σ) $\leftarrow M$;
return M

Overview

SAT Solving

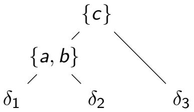
From SAT Solving to Top-Down Knowledge Compilation

Heuristics for Decomposition

- Semantical vs. Syntactic Decompositions
- The Power of Decomposition
- Strategies for Finding Decompositions and Related Compilers
  - Lazy Decomposition: `Dsharp`
  - Global Decomposition: `C2D`
  - Local Decomposition: `D4`

Static vs. Dynamic Decomposition

- When a dtree is computed first for finding out the cutsets leading to decompositions, the same cutsets are considered whatever their ancestor cutsets (hence whatever the assignments γ of their variables)
- The CNF formula conditioned by γ which results at the current decision node of the search tree is not considered



$\{a, b\}$ is considered as a cutset whatever c has been assigned to true or to false

Static vs. Dynamic Decomposition

- Pros: No need to call a hypergraph partitioner for every assignment γ of the variables from the ancestor cutset (this is an expensive operation)
- Cons: $\Sigma \mid \gamma$ may heavily vary depending on γ, so that better syntactic decomposition schemes could be obtained if the assignments themselves were taken into account

The Decision-DNNF Compiler D4

- D4: a **D**ecision-**D**NNF compiler based on **D**ynamic **D**ecomposition
  - Input: a CNF formula Σ
  - Output: a decision-DNNF representation equivalent to the input
- D4 is a top-down compiler which generates a Decision-DNNF representation by following the trace of a SAT solver
- D4 is based on the same ingredients as the previous compilers C2D and Dsharp: disjoint component analysis, conflict analysis and non-chronological backtracking, component caching

D4: What's Up?

- The variable selection heuristics is dynamic like `Dsharp` (and unlike `C2D`)
- It is based on a partitioning of the dual hypergraph of the input `CNF` like `C2D` (and unlike `Dsharp`)
- Two new features:
  - hypergraph partitioning (based on the `PaToH` partitioner) is used sparingly and during the search for finding decompositions
  - a set of simplification rules are also used to minimize the time spent in the partitioning steps and to promote the quality of the decompositions

A Pseudo-Code of D4

Algorithm 7: D4(Σ, *LV*)

input : a CNF formula Σ and a list of variables *LV* (empty at start)
output: the root node *N* of a decision-DNNF representation of Σ

```
S ← solve(Σ);
if S = {∅} then return leaf(⊥);
if Var(Σ) = ∅ then return aNode(S, [leaf(⊤)]);
if cache(Σ) ≠ nil then return aNode(S, [cache(Σ)]);
comps ← connectedComponents(Σ);
LN_d ← [];
foreach c ∈ comps do
    LV_c ← restrict(LV, Var(c));
    if LV_c = ∅ or #(Var(S) ∩ Var(c)) > 1/10 #(Var(c)) then
        LV_c ← sort(HGP(c));
    v ← head(LV_c);
    LV_c ← tail(LV_c);
    N_d ← ite(v, D4(c|¬v, LV_c), D4(c|v, LV_c));
    LN_d ← add(N_d, LN_d);
N_∧ ← aNode(S, LN_d);
cache(Σ) ← N_∧;
return N_∧
```

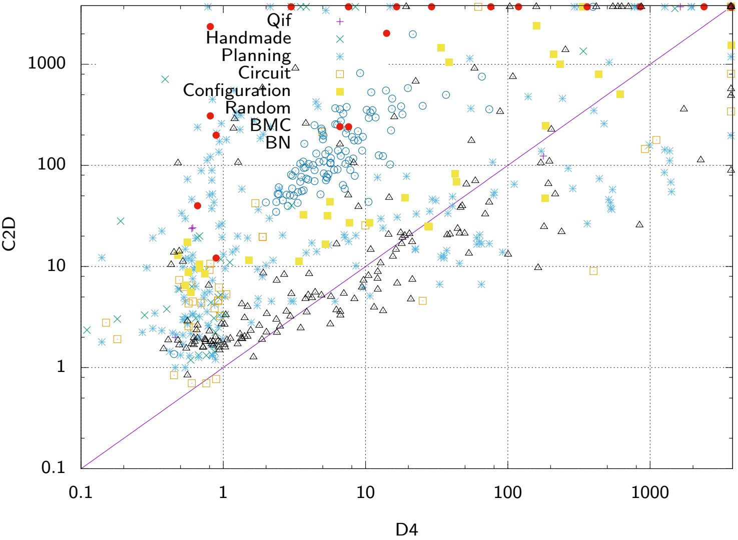
Improving the Hypergraph Partitioning Steps

- We avoid calling HGP at each recursion step or each time a decision node must be generated
- We designed some specific rules which are used inside HGP and aim at simplifying the hypergraph associated with the current formula before calling `PaToH` on it
- The simplification achieved can also lead `PaToH` to find better decompositions
  - we exploit an algorithm for the detection of literal equivalences based on BCP (more details on Wednesday!)
  - we simplify the dual hypergraph of the resulting formula, removing some useless nodes and hyperedges

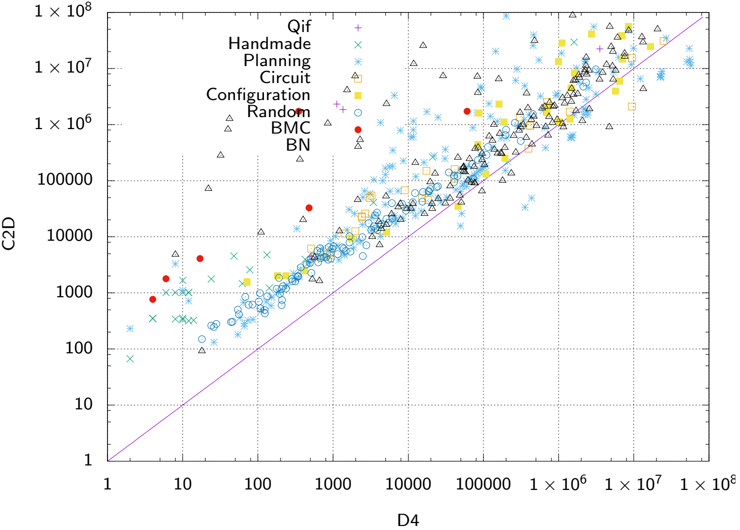
Empirical Evaluation

- 703 CNF instances from the SAT LIBrary
- 8 data sets: BN (Bayesian networks) (192), BMC (Bounded Model Checking) (18), Circuit (41), Configuration (35), Handmade (58), Planning (248), Random (104), Qif (7) (Quantitative Information Flow analysis - security)
- Experiments conducted on Intel Xeon E5-2643 (3.30 GHz) processors with 32 GiB RAM on Linux CentOS
- A time-out of 1h and a memory-out of 7.6 GiB has been considered for each instance

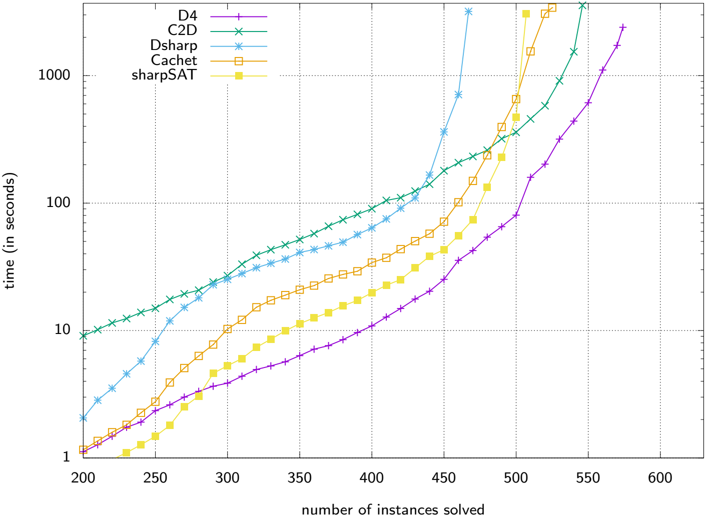
Comparison with C2D (compilation times)



Comparison with C2D (sizes of the compiled forms)



D4 as a Model Counter



References (for further reading)

A. Darwiche. Decomposable negation normal form. Journal of the ACM, 48(4):608–647, 2001.

A. Darwiche. New advances in compiling cnf into decomposable negation normal form. ECAI'04, pages 328–332, 2004.

J.-M. Lagniez, and P. Marquis. An Improved Decision-DNNF Compiler. IJCAI'17, pages 667-673, 2017.

Top-Down Knowledge Compilation

Jean-Marie Lagniez & Pierre Marquis*

CRIL, U. Artois & CNRS
Institut Universitaire de France*
France