

# Foundations of Artificial Intelligence

## 8. Satisfiability and Model Construction

DPLL Procedure, Phase Transitions, Local Search, State of the Art

Joschka Boedecker and Wolfram Burgard and  
Frank Hutter and Bernhard Nebel and Michael Tangermann



Albert-Ludwigs-Universität Freiburg

May 29, 2019

SAT solving is the best available technology for practical solutions to many NP-hard problems

Formal verification

- Verification of software

Ruling out unintended states (null-pointer exceptions, etc.)

Proving that the program computes the right solution

- Verification of hardware (Pentium bug, etc)

Practical approach:

encode into SAT & exploit the rapid progress in SAT solving

Solving CSP instances in practice

Solving graph coloring problems in practice

...

- 1 The SAT Problem
- 2 Davis-Putnam-Logemann-Loveland (DPLL) Procedure
- 3 “Average” Complexity of the Satisfiability Problem
- 4 Local Search Procedures
- 5 State of the Art

# Lecture Overview

- 1 The SAT Problem
- 2 Davis-Putnam-Logemann-Loveland (DPLL) Procedure
- 3 “Average” Complexity of the Satisfiability Problem
- 4 Local Search Procedures
- 5 State of the Art

# Logical deduction vs. satisfiability

Propositional Logic — typical algorithmic questions:

Logical deduction  $\rightarrow$  *KO*

**Given:** A logical theory (set of propositions)

**Question:** Does a proposition **logically follow** from this theory?

Reduction to **unsatisfiability**, which is **coNP-complete** (complementary to NP problems)

Satisfiability of a formula (SAT)

**Given:** A logical theory

**Wanted:** **Model of the theory** *Solution*

**Example:** Configurations that fulfill the constraints given in the theory  
Can be “easier” because it is enough to find one model

# The Satisfiability Problem (SAT)

Given:

Propositional formula  $\varphi$  in CNF

Wanted:

Model of  $\varphi$ .

or proof, that no such model exists.

$$(a) \wedge (b \vee c) \wedge (\neg b \vee c \vee d)$$

SAT can be formulated as a Constraint-Satisfaction-Problem ( $\rightarrow$  search):

Var :  $a, b, c, d$

Dom :  $T/F$

Constraints :  $a = \text{true}$

$b \vee c = \text{true}$

$\neg b \vee c \vee d = \text{true}$

SAT can be formulated as a Constraint-Satisfaction-Problem ( $\rightarrow$  search):

CSP-Variables = Symbols of the alphabet

Domain of values =  $\{T, F\}$

Constraints given by clauses



# Lecture Overview

- 1 The SAT Problem
- 2 Davis-Putnam-Logemann-Loveland (DPLL) Procedure
- 3 “Average” Complexity of the Satisfiability Problem
- 4 Local Search Procedures
- 5 State of the Art

^

# The DPLL algorithm

The DPLL algorithm (Davis, Putnam, Logemann, Loveland, 1962) corresponds to backtracking with inference in CSPs:

Recursive call  $\text{DPLL}(\Delta, l)$  with

$\Delta$ : set of clauses

$l$ : partial variable assignment

Result: satisfying assignment that extends  $l$   
or “unsatisfiable” if no such assignment exists.

First call by  $\text{DPLL}(\Delta, \emptyset)$

*empty partial assign.*

# The DPLL algorithm

The DPLL algorithm (Davis, Putnam, Logemann, Loveland, 1962) corresponds to backtracking with inference in CSPs:

Recursive call  $\text{DPLL}(\Delta, l)$  with

$\Delta$ : set of clauses

$l$ : partial variable assignment

Result: satisfying assignment that extends  $l$   
or “unsatisfiable” if no such assignment exists.

First call by  $\text{DPLL}(\Delta, \emptyset)$

Inference in DPLL:

Simplify: if variable  $v$  is assigned a value  $d$ , then all clauses containing  $v$  are simplified immediately (corresponds to forward checking)

# The DPLL algorithm

The DPLL algorithm (Davis, Putnam, Logemann, Loveland, 1962) corresponds to backtracking with inference in CSPs:

Recursive call  $\text{DPLL}(\Delta, l)$  with

$\Delta$ : set of clauses

$l$ : partial variable assignment

Result: satisfying assignment that extends  $l$   
or “unsatisfiable” if no such assignment exists.

First call by  $\text{DPLL}(\Delta, \emptyset)$

Inference in DPLL:

Simplify: if variable  $v$  is assigned a value  $d$ , then all clauses containing  $v$  are simplified immediately (corresponds to forward checking)

Variables in unit clauses (= clauses with only one variable) are immediately assigned (corresponds to minimum remaining values ordering in CSPs)

# The DPLL Procedure

## DPLL Function

Given a set of clauses  $\Delta$  defined over a set of variables  $\Sigma$ , return “satisfiable” if  $\Delta$  is satisfiable. Otherwise return “unsatisfiable”.

1. If  $\Delta = \emptyset$  return “satisfiable”
2. If  $\square \in \Delta$  return “unsatisfiable”
3. **Unit-propagation Rule:** If  $\Delta$  contains a **unit-clause**  $C$ , assign a truth-value to the variable in  $C$  that satisfies  $C$ , simplify  $\Delta$  to  $\Delta'$  and return **DPLL**( $\Delta'$ ).

3 cases.

# The DPLL Procedure

## DPLL Function

Given a set of clauses  $\Delta$  defined over a set of variables  $\Sigma$ , return “satisfiable” if  $\Delta$  is satisfiable. Otherwise return “unsatisfiable”.

1. If  $\Delta = \emptyset$  return “satisfiable”
2. If  $\square \in \Delta$  return “unsatisfiable”
3. **Unit-propagation Rule**: If  $\Delta$  contains a **unit-clause**  $C$ , assign a truth-value to the variable in  $C$  that satisfies  $C$ , simplify  $\Delta$  to  $\Delta'$  and return **DPLL**( $\Delta'$ ).
4. **Splitting Rule**: Select from  $\Sigma$  a variable  $v$  which has not been assigned a truth-value. Assign one truth value  $t$  to it, simplify  $\Delta$  to  $\Delta'$  and call **DPLL**( $\Delta'$ )
  - a. If the call returns “satisfiable”, then return “satisfiable”.
  - b. Otherwise assign *the other* truth-value to  $v$  in  $\Delta$ , simplify to  $\Delta''$  and return **DPLL**( $\Delta''$ ).

## Example (1)

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

# Example (1)

$$\Delta = \{\{a, b, \overset{F}{\cancel{\neg c}}\}, \{\neg a, \neg b\}, \{\overset{T}{\cancel{c}}\}, \{a, \neg b\}\}$$

1. Unit-propagation rule:  $c \mapsto T$  *perforce*



# Example (1)

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

1. Unit-propagation rule:  $c \mapsto T$   
 $\{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}$

# Example (1)

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

1. Unit-propagation rule:  $c \mapsto T$   
 $\{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}$
2. Splitting rule:

# Example (1)


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

1. Unit-propagation rule:  $c \mapsto T$

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

2. Splitting rule:

2a.  $a \mapsto F$   
 $\{\{b\}, \{\neg b\}\}$



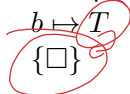
# Example (1)

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

1. Unit-propagation rule:  $c \mapsto T$   
 $\{\{a, \cancel{b}\}, \{\neg a, \neg b\}, \{a, \cancel{\neg b}\}\}$
2. Splitting rule:

2a.  $a \mapsto F$   
 $\{\{b\}, \{\neg b\}\}$

- 3a. Unit-propagation rule:


$$\begin{array}{l} b \mapsto T \\ \{\square\} \end{array}$$

# Example (1)

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

1. Unit-propagation rule:  $c \mapsto T$   
 $\{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}$

2. Splitting rule:

2a.  $a \mapsto F$   
 $\{\{b\}, \{\neg b\}\}$

2b.  $a \mapsto T$   
 $\{\{\neg b\}\}$

3a. Unit-propagation rule:  
 $b \mapsto T$   
 $\{\square\}$

# Example (1)

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

1. Unit-propagation rule:  $c \mapsto T$

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

2. Splitting rule:

2a.  $a \mapsto F$

$$\{\{b\}, \{\neg b\}\}$$

3a. Unit-propagation rule:

$$b \mapsto T$$

$$\{\square\}$$

2b.  $a \mapsto T$

$$\{\{\neg b\}\}$$

3b. Unit-propagation rule:  $b \mapsto F$

$$\{\}$$

# Example (1)

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

1. Unit-propagation rule:  $c \mapsto T$

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

2. Splitting rule:

2a.  $a \mapsto F$

$$\{\{b\}, \{\neg b\}\}$$

2b.  $a \mapsto T$

$$\{\{\neg b\}\}$$

3a. Unit-propagation rule:

$$b \mapsto T$$

$$\{\Box\}$$

3b. Unit-propagation rule:  $b \mapsto F$

$$\{\}$$

## Example (2)

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



## Example (2)

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

1. Unit-propagation rule:  $d \mapsto T$

## Example (2)

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

1. Unit-propagation rule:  $d \mapsto T$   
 $\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$

## Example (2)

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

1. Unit-propagation rule:  $d \mapsto T$   
 $\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$
2. Unit-propagation rule:  $b \mapsto T$   
 $\{\{a, \neg c\}, \{c\}\}$

## Example (2)

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

1. Unit-propagation rule:  $d \mapsto T$   
 $\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$
2. Unit-propagation rule:  $b \mapsto T$   
 $\{\{a, \neg c\}, \{c\}\}$
3. Unit-propagation rule:  $c \mapsto T$   
 $\{\{a\}\}$

## Example (2)

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

1. Unit-propagation rule:  $d \mapsto T$   
 $\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$
2. Unit-propagation rule:  $b \mapsto T$   
 $\{\{a, \neg c\}, \{c\}\}$
3. Unit-propagation rule:  $c \mapsto T$   
 $\{\{a\}\}$
4. Unit-propagation rule:  $a \mapsto T$   
 $\{\}$

## Example (2)

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

1. Unit-propagation rule:  $d \mapsto T$   
 $\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$
2. Unit-propagation rule:  $b \mapsto T$   
 $\{\{a, \neg c\}, \{c\}\}$
3. Unit-propagation rule:  $c \mapsto T$   
 $\{\{a\}\}$
4. Unit-propagation rule:  $a \mapsto T$   
 $\{\}$

# Properties of DPLL

DPLL is complete, correct, and guaranteed to terminate.

DPLL constructs a model, if one exists.

In general, DPLL requires **exponential time** (splitting rule!).

*NP-Hard.*

→ Heuristics are needed to determine which variable should be instantiated next and which value should be used.

DPLL is **polynomial** on **Horn clauses** (see next slides).

In current **SAT** competitions, **DPLL-based** procedures have shown the best performance.

# DPLL on Horn Clauses (0)

Horn Clauses constitute an important special case, since they require **only polynomial runtime of DPLL**.

**Definition:** A Horn clause is a clause with **maximally one positive literal**

E.g.,  $\neg A_1 \vee \dots \vee \neg A_n \vee B$  or  $\neg A_1 \vee \dots \vee \neg A_n$   
( $n = 0$  is permitted).

Equivalent representation:  $\neg A_1 \vee \dots \vee \neg A_n \vee B \Leftrightarrow \bigwedge_i A_i \Rightarrow B$

→ Basis of logic programming (e.g., PROLOG)



# DPLL on Horn Clauses (1)

Note:

1. The simplifications in DPLL on Horn clauses **always generate Horn clauses**
2. If the **first sequence of applications of the unit propagation rule in DPLL does not lead to termination**, **a set of Horn clauses without unit clauses is generated**
3. A set of Horn clauses **without unit clauses** and **without the empty clause** is satisfiable, since

*All clauses have at least one negative literal (since all non-unit clauses have at least two literals, where at most one can be positive (Def. Horn))*

*Assigning false to all variables satisfies formula*

## DPLL on Horn Clauses (2)

4. It follows from 3.:
  - a. every time the splitting rule is applied, the current formula is satisfiable
  - b. every time, when the wrong decision (= assignment in the splitting rule) is made, this will be immediately detected (e.g., only through unit propagation steps and the derivation of the empty clause).
5. Therefore, the search trees for  $n$  variables can only contain a maximum of  $n$  nodes, in which the splitting rule is applied (and the tree branches).
6. Therefore, the size of the search tree is only polynomial in  $n$  and therefore the running time is also polynomial.

# Lecture Overview

- 1 The SAT Problem
- 2 Davis-Putnam-Logemann-Loveland (DPLL) Procedure
- 3 “Average” Complexity of the Satisfiability Problem
- 4 Local Search Procedures
- 5 State of the Art

# How Good is DPLL in the Average Case?

We know that SAT is NP-complete, i.e., in the worst case, it takes exponential time.

This is clearly also true for the DPLL-procedure.

→ Couldn't we do better in the **average case**?

For CNF-formulae, in which the probability for a positive appearance, negative appearance and non-appearance in a clause is  $1/3$ , DPLL needs on average **quadratic time** (Goldberg 79)!

→ The probability that these formulae are satisfiable is, however, very high.

Conversely, we can, of course, try to identify **hard to solve** problem instances.

Cheeseman et al. (IJCAI-91) came up with the following plausible conjecture:

All NP-complete problems have at least *one order* parameter and the hard to solve problems are around a critical value of this order parameter. This critical value (a **phase transition**) separates one region from another, such as over-constrained and under-constrained regions of the problem space.

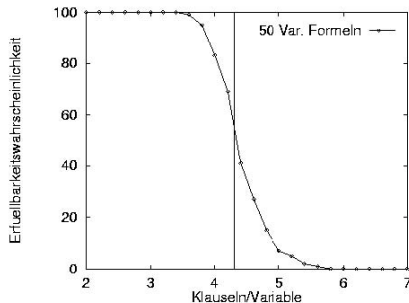
Confirmation for graph coloring and Hamiltonian path ..., later also for other NP-complete problems.

# Phase Transitions with 3-SAT

Constant clause length model (Mitchell et al., AAAI-92):

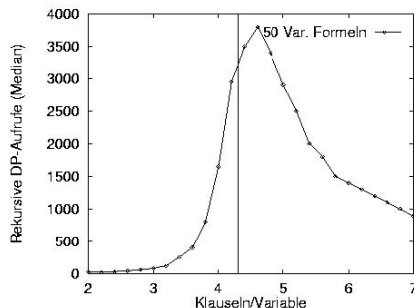
Clause length  $k$  is given. Choose variables for every clause  $k$  and use the complement with probability 0.5 for each variable.

Phase transition for 3-SAT with a clause/variable ratio of approx. 4.3:



# Empirical Difficulty

The Davis-Putnam (DPLL) Procedure shows extreme runtime peaks at the phase transition:



**Note:** Hard instances can exist even in the regions of the more easily satisfiable/unsatisfiable instances!

When the probability of a solution is close to 1 (**under-constrained**), there are many solutions, and the first search path of a backtracking search is usually successful.

If the probability of a solution is close to 0 (**over-constrained**), this fact can usually be determined early in the search.

In the phase transition stage, there are many near successes

→ (limited) possibility of predicting the difficulty of finding a solution based on the parameters

→ (search intensive) benchmark problems are located in the phase region (but they have a special structure)



# Lecture Overview

- 1 The SAT Problem
- 2 Davis-Putnam-Logemann-Loveland (DPLL) Procedure
- 3 “Average” Complexity of the Satisfiability Problem
- 4 Local Search Procedures**
- 5 State of the Art

# Local Search Methods for Solving Logical Problems

In many cases, we are interested in finding a satisfying assignment of variables (example CSP), and we can sacrifice completeness if we can “solve” much larger instances this way.

Standard process for optimization problems: **Local Search**

Based on a (random) configuration

Through local modifications, we hope to produce better configurations

→ Main problem: **local maxima** / *minima*

As a measure of the value of a configuration in a logical problem, we could use the number of satisfied constraints/clauses.

At first glance, local search seems inappropriate, considering that we want to find a global maximum (all constraints/clauses satisfied).

However:

By **restarting** and/or **injecting** noise, we can often escape local maxima.

Local search can perform very well for SAT solving

# A pioneering local search method for SAT: GSAT (1993)

## Procedure GSAT

**INPUT:** a set of clauses  $\alpha$ , MAX-FLIPS, and MAX-TRIES

**OUTPUT:** a satisfying truth assignment of  $\alpha$ , if found

**begin**

**for**  $i := 1$  to MAX-TRIES

$T :=$  a randomly-generated truth assignment

**for**  $j := 1$  to MAX-FLIPS

**if**  $T$  satisfies  $\alpha$  **then return**  $T$

$v :=$  a propositional variable such that a change in its  
        truth assignment gives the largest increase in  
        the number of clauses of  $\alpha$  that are satisfied by  $T$

$T := T$  with the truth assignment of  $v$  reversed

**end for**

**end for**

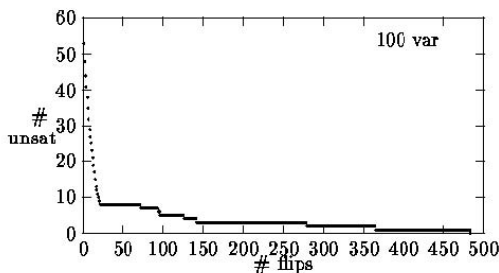
**return** “no satisfying assignment found”

**end**

# The Search Behavior of GSAT

In contrast to many other local search methods, we must also allow sideways movements!

Most time is spent searching on **plateaus**.



# Lecture Overview

- 1 The SAT Problem
- 2 Davis-Putnam-Logemann-Loveland (DPLL) Procedure
- 3 “Average” Complexity of the Satisfiability Problem
- 4 Local Search Procedures
- 5 State of the Art

# Practical Improvements of DPLL Algorithms

## Clause Learning

- Consider an exemplary SAT problem

26 variables  $A, \dots, Z$

Amongst many other clauses, we have

$\{(\neg A, \underline{Y}, \underline{Z})\}, \{(\neg A, \neg \underline{Y}, \underline{Z})\}, \{(\neg A, \underline{Y}, \neg \underline{Z})\}, \{(\neg A, \neg \underline{Y}, \neg \underline{Z})\}$

We'll branch on variables in lexicographic order and try true first

- What will happen?

# Practical Improvements of DPLL Algorithms

## Clause Learning

- Consider an exemplary SAT problem

26 variables  $A, \dots, Z$

Amongst many other clauses, we have

$\{(\neg A, Y, Z)\}, \{(\neg A, \neg Y, Z)\}, \{(\neg A, Y, \neg Z)\}, \{(\neg A, \neg Y, \neg Z)\}$

We'll branch on variables in lexicographic order and try true first

- What will happen?

There is no satisfying assignment to the clauses above when  
 $A=T$



# Practical Improvements of DPLL Algorithms

## Clause Learning

- Consider an exemplary SAT problem

26 variables  $A, \dots, Z$

Amongst many other clauses, we have

$\{(\neg A, Y, Z)\}, \{(\neg A, \neg Y, Z)\}, \{(\neg A, Y, \neg Z)\}, \{(\neg A, \neg Y, \neg Z)\}$

We'll branch on variables in lexicographic order and try true first

- What will happen?

There is no satisfying assignment to the clauses above when  $A=T$

For each assignment to variables  $B, \dots, X$ , we'll have to rediscover this fact

Rather: reason about the variables that led to a conflict:  $A, Y$  and  $Z$

We can 'learn' (here: logically infer) a new clause:  $\neg A$

Leads to **conflict-directed clause learning (CDCL)**

# Practical Improvements of DPLL Algorithms

## Clause Learning

- Consider an exemplary SAT problem

26 variables  $A, \dots, Z$

Amongst many other clauses, we have

$\{(\neg A, Y, Z)\}, \{(\neg A, \neg Y, Z)\}, \{(\neg A, Y, \neg Z)\}, \{(\neg A, \neg Y, \neg Z)\}$

We'll branch on variables in lexicographic order and try true first

- What will happen?

There is no satisfying assignment to the clauses above when  $A=T$

For each assignment to variables  $B, \dots, X$ , we'll have to rediscover this fact

Rather: reason about the variables that led to a conflict:  $A, Y$  and  $Z$

We can 'learn' (here: logically infer) a new clause:  $\neg A$

Leads to **conflict-directed clause learning (CDCL)**

## Intelligent Backjumping

Close related to clause learning

# Practical Improvements of SAT Algorithms

Both for DPLL/CDCL algorithms and local search algorithms

- Randomization and restarts

- Efficient data structures, indexing, etc

- Engineering ingenious heuristics

Both for DPLL/CDCL algorithms and local search algorithms

Randomization and restarts

Efficient data structures, indexing, etc

Engineering ingenious heuristics

## Meta-algorithmic advances

- Automated parameter tuning and algorithm configuration
- Selection of the best-fitting algorithm based on instance characteristics
- Selection of the best-fitting parameters based on instance characteristics
- Use of machine learning to pinpoint what factors most affects performance

# The Current State of the Art

SAT competitions since beginning of the 90s

Current SAT competitions (<http://www.satcompetition.org/>):

Largest “industrial” instances:  $> 10,000,000$  variables

Complete solvers dominate handcrafted and industrial tracks

Incomplete local search solvers best on random satisfiable instances

# The Current State of the Art

SAT competitions since beginning of the 90s

Current SAT competitions (<http://www.satcompetition.org/>):

Largest “industrial” instances:  $> 10,000,000$  variables

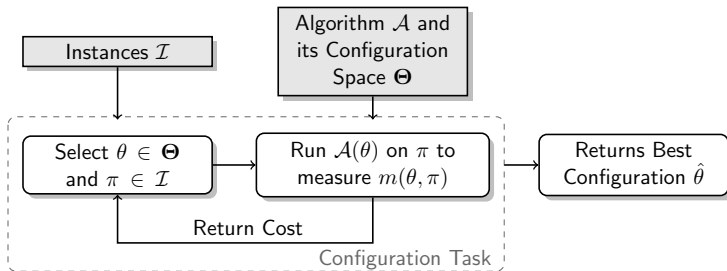
Complete solvers dominate handcrafted and industrial tracks

Incomplete local search solvers best on random satisfiable instances

Best solvers use meta-algorithmic methods, such as algorithm configuration, selection, etc.

We thus discuss these briefly next

# Algorithm Configuration

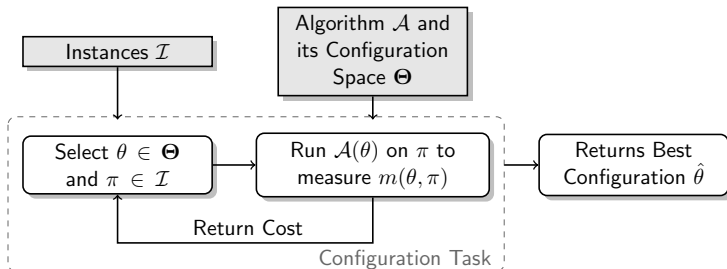


## Definition: algorithm configuration

Given:

- a parameterized algorithm  $\mathcal{A}$  with possible parameter settings  $\Theta$ ;
- a distribution  $\mathcal{D}$  over problem instances with domain  $\mathcal{I}$ ; and

# Algorithm Configuration



## Definition: algorithm configuration

Given:

- a parameterized algorithm  $\mathcal{A}$  with possible parameter settings  $\Theta$ ;
- a distribution  $\mathcal{D}$  over problem instances with domain  $\mathcal{I}$ ; and
- a cost metric  $m : \Theta \times \mathcal{I} \rightarrow \mathbb{R}$ ,

Find:  $\theta^* \in \arg \min_{\theta \in \Theta} \mathbb{E}_{\pi \sim \mathcal{D}}(m(\theta, \pi))$ .



## Formal verification

Software verification [Babić & Hu; CAV '07]

Hardware verification (Bounded model checking) [Zarpas; SAT '05]

Recent progress based on SAT solvers

## Formal verification

Software verification [Babić & Hu; CAV '07]

Hardware verification (Bounded model checking) [Zarpas; SAT '05]

Recent progress based on SAT solvers

## CDCL solver for SAT-based verification

SPEAR, developed by Domagoj Babić at UBC

26 parameters,  $8.34 \times 10^{17}$  configurations

- Ran algorithm configuration method ParamILS: 2 days on 10 machines
- On a training set from each benchmark

Ran algorithm configuration method ParamILS: 2 days on 10 machines

- On a training set from each benchmark

Compared to manually-engineered default

- 1 week of performance tuning
- Competitive with the state of the art
- Comparison on unseen test instances

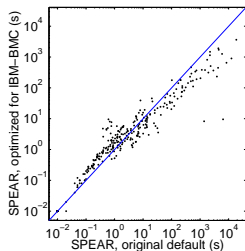
# Configuration of a SAT Solver for Verification [Hutter et al, 2007]

Ran algorithm configuration method ParamILS: 2 days on 10 machines

- On a training set from each benchmark

Compared to manually-engineered default

- 1 week of performance tuning
- Competitive with the state of the art
- Comparison on unseen test instances



4.5-fold speedup

on hardware verification

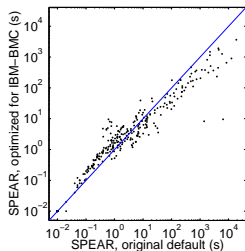
# Configuration of a SAT Solver for Verification [Hutter et al, 2007]

Ran algorithm configuration method ParamILS: 2 days on 10 machines

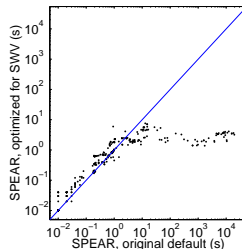
- On a training set from each benchmark

Compared to manually-engineered default

- 1 week of performance tuning
- Competitive with the state of the art
- Comparison on unseen test instances



4.5-fold speedup  
on hardware verification



500-fold speedup  $\rightsquigarrow$  won category  
QF\_BV in 2007 SMT competition

## Definition: algorithm selection

Given

- a set  $\mathcal{I}$  of problem instances,
- a portfolio of algorithms  $\mathcal{P}$ ,
- and a cost metric  $m : \mathcal{P} \times \mathcal{I} \rightarrow \mathbb{R}$ ,

the per-instance algorithm selection problem is to find a mapping  $s : \mathcal{I} \rightarrow \mathcal{P}$  that optimizes  $\sum_{\pi \in \mathcal{I}} m(s(\pi), \pi)$ , the sum of cost measures achieved by running the selected algorithm  $s(\pi)$  for instance  $\pi$ .

# Algorithm Selection

## Definition: algorithm selection

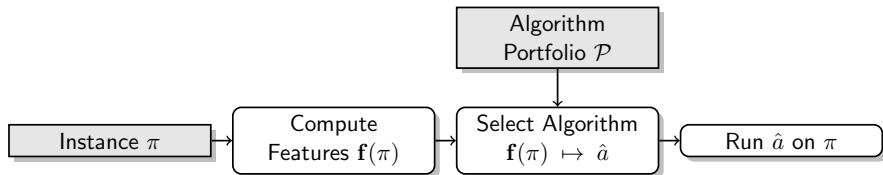
Given

a set  $\mathcal{I}$  of problem instances,

a portfolio of algorithms  $\mathcal{P}$ ,

and a cost metric  $m : \mathcal{P} \times \mathcal{I} \rightarrow \mathbb{R}$ ,

the per-instance algorithm selection problem is to find a mapping  $s : \mathcal{I} \rightarrow \mathcal{P}$  that optimizes  $\sum_{\pi \in \mathcal{I}} m(s(\pi), \pi)$ , the sum of cost measures achieved by running the selected algorithm  $s(\pi)$  for instance  $\pi$ .





# Example SAT Challenge 2012

Rank	RiG	Solver	#solved
–	–	Virtual Best Solver (VBS)	568
1	1	SATzilla2012 APP	531
2	2	SATzilla2012 ALL	515
3	1	Industrial SAT Solver	499
–	–	lingeling (SAT Competition 2011 Bronze)	488
4	2	interactSAT	480
5	1	glucose	475
6	2	SINN	472
7	3	ZENN	468
8	4	Lingeling	467
9	5	linge_dyphase	458
10	6	simpsat	453

The VBS is the best possible performance  
of an algorithm selection system.

(pink: algorithm selectors, blue: portfolios, green: single-engine solvers)

# Automated construction of portfolios from a single algorithm

## Algorithm Configuration

Strength: find a single configuration with strong performance for a given cost metric

Weakness: for heterogeneous instance sets, there is often no configuration that performs great for all instances

# Automated construction of portfolios from a single algorithm

## Algorithm Configuration

Strength: find a single configuration with strong performance for a given cost metric

Weakness: for heterogeneous instance sets, there is often no configuration that performs great for all instances

## Algorithm Selection

Strength: for heterogeneous instance sets, pick the right algorithm from a set

Weakness: the set to choose from typically only contains a few algorithms

# Automated construction of portfolios from a single algorithm

## Algorithm Configuration

Strength: find a single configuration with strong performance for a given cost metric

Weakness: for heterogeneous instance sets, there is often no configuration that performs great for all instances

## Algorithm Selection

Strength: for heterogeneous instance sets, pick the right algorithm from a set

Weakness: the set to choose from typically only contains a few algorithms

## Putting the two together

Use algorithm configuration to determine useful configurations

Use algorithm selection to select from them based on instance characteristics

# Automated construction of portfolios from a single algorithm: Hydra [Xu et al. 2010, 2011]

## Idea

Iteratively add configurations to a portfolio  $\mathcal{P}$ , starting with  $\mathcal{P} = \emptyset$

In each iteration, determine configuration that is complementary to  $\mathcal{P}$

# Automated construction of portfolios from a single algorithm: Hydra [Xu et al. 2010, 2011]

## Idea

Iteratively add configurations to a portfolio  $\mathcal{P}$ , starting with  $\mathcal{P} = \emptyset$

In each iteration, determine configuration that is complementary to  $\mathcal{P}$

Maximize marginal contribution of configuration  $\theta$  to current portfolio  $\mathcal{P}$ :

$$m(\mathcal{P}) - m(\mathcal{P} \cup \{\theta\})$$

# Automated construction of portfolios from a single algorithm: Hydra [Xu et al. 2010, 2011]

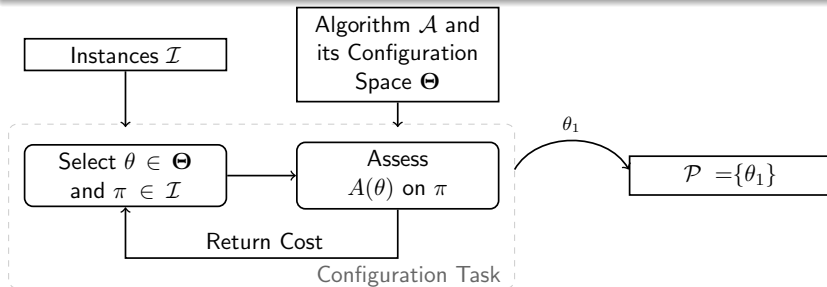
## Idea

Iteratively add configurations to a portfolio  $\mathcal{P}$ , starting with  $\mathcal{P} = \emptyset$

In each iteration, determine configuration that is complementary to  $\mathcal{P}$

Maximize marginal contribution of configuration  $\theta$  to current portfolio  $\mathcal{P}$ :

$$m(\mathcal{P}) - m(\mathcal{P} \cup \{\theta\})$$



# Automated construction of portfolios from a single algorithm: Hydra [Xu et al. 2010, 2011]

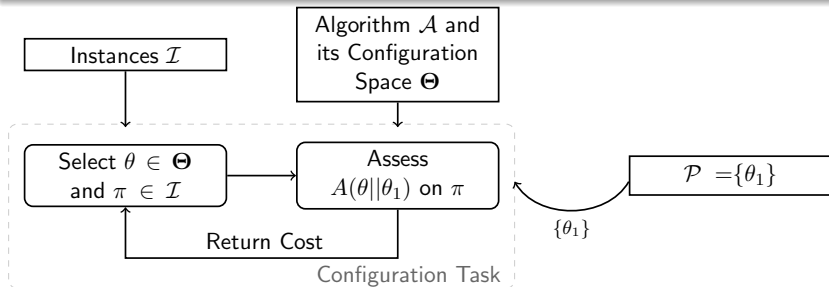
## Idea

Iteratively add configurations to a portfolio  $\mathcal{P}$ , starting with  $\mathcal{P} = \emptyset$

In each iteration, determine configuration that is complementary to  $\mathcal{P}$

Maximize marginal contribution of configuration  $\theta$  to current portfolio  $\mathcal{P}$ :

$$m(\mathcal{P}) - m(\mathcal{P} \cup \{\theta\})$$





# Automated construction of portfolios from a single algorithm: Hydra [Xu et al. 2010, 2011]

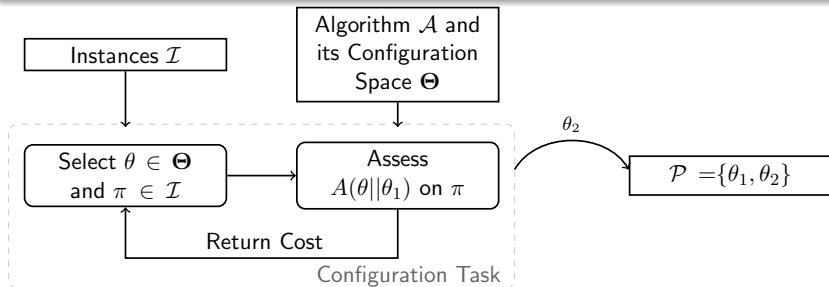
## Idea

Iteratively add configurations to a portfolio  $\mathcal{P}$ , starting with  $\mathcal{P} = \emptyset$

In each iteration, determine configuration that is complementary to  $\mathcal{P}$

Maximize marginal contribution of configuration  $\theta$  to current portfolio  $\mathcal{P}$ :

$$m(\mathcal{P}) - m(\mathcal{P} \cup \{\theta\})$$



# Automated construction of portfolios from a single algorithm: Hydra [Xu et al. 2010, 2011]

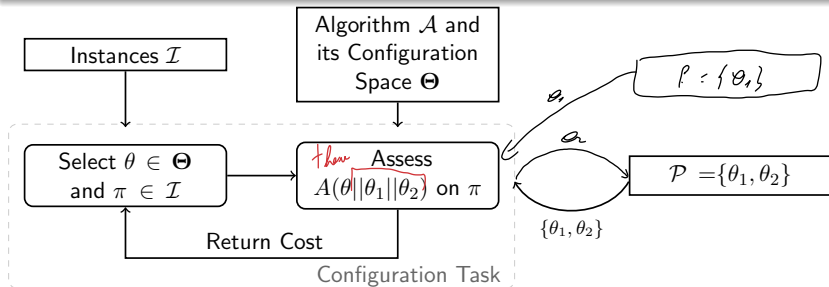
## Idea

Iteratively add configurations to a portfolio  $\mathcal{P}$ , starting with  $\mathcal{P} = \emptyset$

In each iteration, determine configuration that is complementary to  $\mathcal{P}$

Maximize marginal contribution of configuration  $\theta$  to current portfolio  $\mathcal{P}$ :

$$m(\mathcal{P}) - m(\mathcal{P} \cup \{\theta\})$$



# A Large-Scale Application of SAT Technology

## FCC Spectrum Auction

Wireless frequency spectra: demand increases

US Federal Communications Commission (FCC) held 13-month auction

**Key Computational Problem:** feasibility testing based on interference constraints

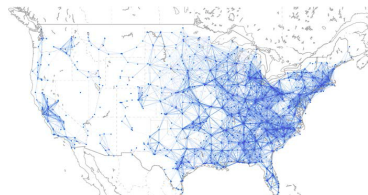
a hard **graph colouring problem**

2991 stations (nodes) &

2.7 million interference constraints

Need to solve many different instances

More instances solved: higher revenue



# A Large-Scale Application of SAT Technology

## FCC Spectrum Auction

Wireless frequency spectra: demand increases

US Federal Communications Commission (FCC) held 13-month auction

**Key Computational Problem:** feasibility testing based on interference constraints

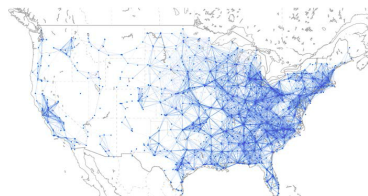
a hard **graph colouring problem**

2991 stations (nodes) &

2.7 million interference constraints

Need to solve many different instances

More instances solved: higher revenue



**Best solution:** based on SAT solving & meta-algorithmic improvements

CDCL Solver Clasp, optimized with algo. configuration method SMAC

Instance-specific configuration with Hydra (using SATzilla for algo. selection)

# Concluding Remarks

DPLL: combines simplification, unit-propagation and backtracking

- Very efficient implementation techniques

- Good branching heuristics

- Clause learning

Incomplete randomized SAT-solvers

- Perform best on random satisfiable problem instances

State of the art

- Typically obtained by automatic algorithm configuration & selection