Algorithms for Propositional Satisfiability Solving

Anbulagan

Anbulagan@nicta.com.au

http://users.rsise.anu.edu.au/~anbu/

Logic and Computation Program, NICTA

CSL, RSISE, Australian National University

Presented at DSL, SITACS, University of Wollongong, 16-17 January 2007

Roadmap

- SAT Resources
- Introduction to SAT
  - Real-world problems
  - Basic notation and definitions
  - Problems in CNF formula
  - Phase transition
- Algorithms for SAT solving
  - DP and DPLL procedure
  - Complete methods: Look-ahead based DPLL and CDCL-based DPLL
  - Stochastic methods: GSAT, Random Walk, Clause Weighting SLS

SAT Resources - 1

- Conferences
  - Main: SAT
  - Others: IJCAI, AAAI, ECAI, PRICAI, CP, IJCAR, CP-AI-OR
  - Applications: DAC, ICCAD, ICAPS
- Journals
  - Main: JSAT
  - Others: AIJ, JAIR, JAR, DAM, Constraint
- Competitions
  - Since 2002
  - Website: www.satcompetition.org

International SAT Competition

- Annual competition since 2002
- In 2005, there are 9 gold, 9 silver and 9 bronze medals
- More than 50 SAT solvers entered the contest
- 1657 problems used: random, crafted and industrial
- 2 stages competition: (20 mins in 1st stage, 100 or 200 mins in 2nd stage)
- Announcement at 2005 International Conference SAT Conference, in St. Andrews, UK, June 21st, 2005.
- No competition in 2006 but there was a SAT-Race for solving only the industrial problems.

- Next competition starts soon.
  - Solver submission deadline is 31 January 2007.

SAT Resources - 2

- Benchmark Data
  - www.satlib.org
- More information about SAT
  - www.satlive.org
- SAT Solvers
  - Most of the solvers can be downloaded from the Internet
  - www.satcompetition.org
  - Authors website

SAT Resources - 3

- Challenges
  - Selman et. al. : IJCAI 1997
    - "Ten Challenges in Propositional Reasoning and Search"
  - Kautz and Selman : CP 2003
    - "Ten Challenges Redux: Recent Progress in Propositional Reasoning and Search"

Problems - Traffic Management

Problems - Nurses Rostering

Many real world problems can be expressed as a list of constraints.

Answer is assignment to variables that satisfy all the constraints.

Example:

- Scheduling people to work in shifts at a hospital.
  - Some people do not work at night.
  - No one can work more than H hours a week.
  - Some pairs of people cannot be on the same shift.
  - Is there assignment of people to shifts that satisfy all constraints?

Problems - Games

- Sudoku

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

Constraint:
There is no same number in the same row, column, or region.

- N-Queens

	Q_1		
			Q_2
Q_3			
		Q_4	

Constraint:
In chess, a queen can move horizontally, vertically, or diagonally.

Problems - Games

- Sudoku

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

Constraint:
There is no same number in the same row, column, or region.

- N-Queens

		Q_1	
Q_2			
			Q_3
	Q_4		

Constraint:
In chess, a queen can move horizontally, vertically, or diagonally.

A Simple Problem: Student-Courses

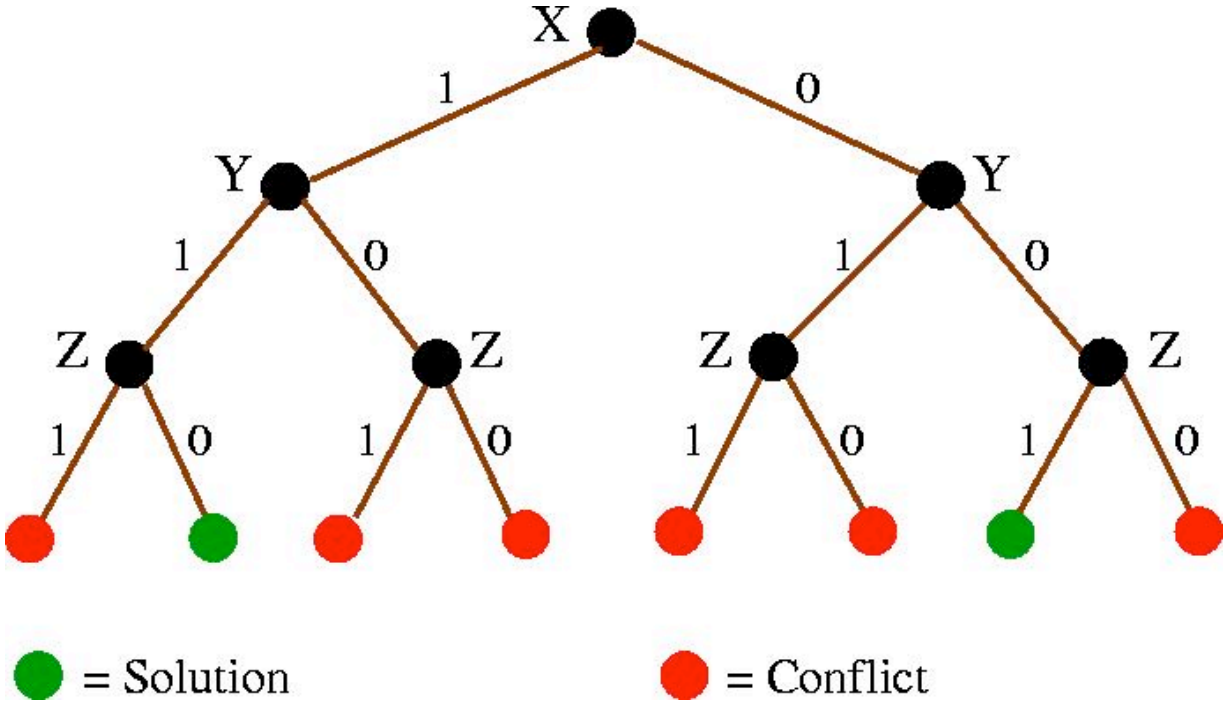
A student would like to decide on which subjects he should take for the next session. He has the following requirements:

- He would like to take Math or drop Biology.
- He would like to take Biology or Algorithms.
- He does not want to take Math and Algorithms together.

Which subjects this student can take?

$$F = (X \vee \neg Y) \wedge (Y \vee Z) \wedge (\neg X \vee \neg Z)$$

Binary Tree of the Student-Courses Problem

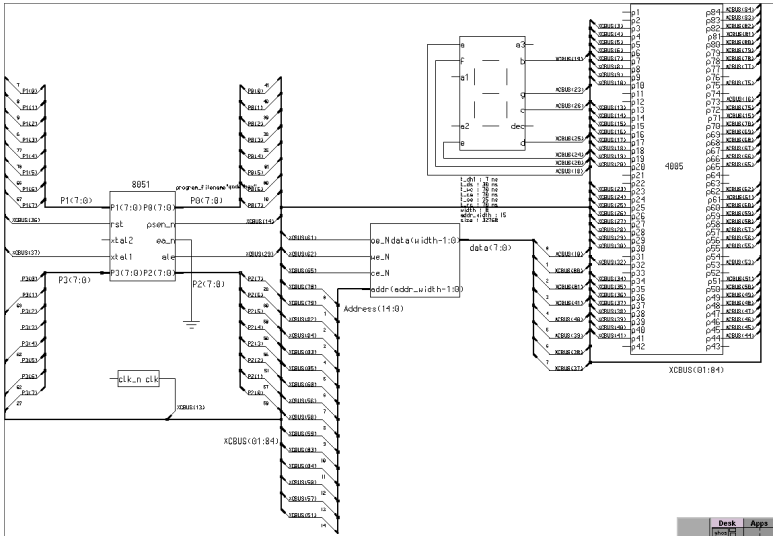


There are 2 possible solutions:

- He could take Math and Biology together. (X=T, Y=T, Z= ⊥)
- He could only take Algorithms. (X= ⊥, Y= ⊥, Z=T)

Practical Applications of SAT

- AI Planning and Scheduling
- Bioinformatics
- Bounded Model Checking
- Data Cleaning
- Diagnosis
- Electronic Design Automation and Verification
- FPGA routing
- Knowledge Discovery
- Security: cryptographic key search
- Software Verification
- Theorem Proving



Propositional Logic

English	Standard	Boolean	Other
false	$\bot$	0	F
true	T	1	T
not x	$\neg x$	$\bar{x}$	$-x$, $\sim x$
x and y	$x \wedge y$	xy	$x \,\&\, y$, $x \,.\, y$
x or y	$x \vee y$	$x + y$	$x \mid y$, x or y
x implies y	$x \Rightarrow y$	$x \leq y$	$x \rightarrow y$, $x \supset y$
x iff y	$x \Leftrightarrow y$	$x = y$	$x \equiv y$, $x \sim y$

Semantics of Boolean Operators

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \Rightarrow y$	$x \Leftrightarrow y$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$

N.B.:

$$x \vee y = \neg(\neg x \wedge \neg y)$$

$$x \Rightarrow y = (\neg x \vee y)$$

$$x \Leftrightarrow y = (x \Rightarrow y) \wedge (y \Rightarrow x)$$

Basic Notation & Definitions

A SAT problem consists of

- Formula F: a conjunction of clauses using AND ($\wedge$) operator
- A set of Boolean variables $\{x_1, x_2, ..., x_n\}$
- Literal: x_i is a positive literal of variable x_i , and $\neg x_i$ is its negation
- Clause: a disjunction of literals using OR ($\vee$) operator
- Unit Clause: a clause containing a single literal
- Binary Clause: a clause that contains two literals
- Empty Clause: a clause without any literal
- Pure Literal: a variable appearing only negatively or positively

$$F = (x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (\neg x_3)$$

Basic Notation & Definitions

Solution for a given SAT problem

- Satisfiable (SAT):
  - There is at least an assignment of values {true, false} to the variables of the formula where all its clauses are satisfiable.
  - Only one model
  - Many models: find one quickly or find all
- Unsatisfiable (UNSAT):
  - There is no model found
  - MAX-SAT: satisfy maximum clauses

Solution of a SAT formula is when a solver can prove whether the formula is satisfiable (SAT) or unsatisfiable (UNSAT).

SAT Problems - Definition

Input: A formula F in Conjunctive Normal Form (CNF)

Output: F is satisfiable by a consistent assignment of truth value to variables or F is unsatisfiable.

Example of a CNF formula:

$$F = (x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee \neg x_5) \wedge (\neg x_2 \vee x_6 \vee x_7)$$

The first NP-Complete problem [Cook, 1971]

NATIONAL ICT AUSTRALIA

SAT Problems - Random Versus Structured

Random Problems: generated using a random problem generator.

The random SAT formula $\boldsymbol{F_k}(n, m)$

- Given a set V of n Boolean variables $\{x_1, x_2, x_3, \ldots, x_n\}$, m clauses of length k are generated randomly. Each clause is produced by randomly choosing k variables from V and negating each with probability 0.5.
- Fixed clause lengths formula
- Mitchell et al., 1992

Structured Problems:

- Structures: symmetries, variable dependencies, clustering
- Generated from real-world problems
- Crafted problems

Random+Structured Problems:

- QWH = quasigroup with holes
- bQWH = balanced quasigroup with holes
- Problem generator is available

Simple Random Problem

```
c  1
p cnf 5 20
 -1  -2  -3 0
  1  -3  -4 0
 -1  -3   5 0
  3   4  -5 0
 -1   2  -5 0
 -2  -3  -5 0
  1  -2  -3 0
  1  -3   4 0
  1  -3  -5 0
 -2   3  -5 0
  2  -4  -5 0
 -1  -2  -4 0
  2  -3   4 0
 -3  -4  -5 0
 -2   4   5 0
 -3  -4   5 0
 -1   3  -5 0
  2  -3   5 0
  1   2  -3 0
 -2   3   5 0
```

- Random 3-SAT problem with 5 variables and 20 clauses
- generated using random SAT problem generator.

Structured Problem: par8-1-c.cnf

c parXX-Y-c denotes a parity problem on XX bits.

c Y is simply the instance number. c means that the instance has been simplified

p cnf 64 254

-2 1 0

-3 -2 0

-3 -2 -1 0

3 2 -1 0

-3 2 1 0

3 -2 1 0

-4 2 0

-5 -4 0

-5 -4 -2 0

5 4 -2 0

-5 4 2 0

5 -4 2 0

…..

…..

Structured Problem: Bounded Model Checking

```
c  The instance bmc-ibm-6.cnf, IBM, 1997
c  6.6 MB data
p cnf 51639 368352
-1 7 0
-1 6 0
-1 5 0
-1 -4 0
-1 3 0
-1 2 0
-1 -8 0
……
10224 -10043 0
10224 -10044 0
10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022
      10023 10024 10025 10026 10027 10028 10029 10030 10031 10032 10033 10034 10035 10036
      10037 10086 10087 10088 10089 10090 10091 10092 10093 10094 10095 10096 10097 10098
      10099 10100 10101 10102 10103 10104 10105 10106 10107 10108 10109 10189 -55 -54 -53 52 51
      50 10043 10044 -10224 0
10083 -10157 0
10083 -10227 0
10083 -10228 0
10157 10227 10228 -10083 0
```

i.e. **((not x_1) or x_7)**
and ((not x_1) or x_6)
and … etc

// a constraint with 64 literals at line 72054

Structured Problem: Bounded Model Checking

At the end of the file

7 -260 0
1072 1070 0
-15 -14 -13 -12 -11 -10 0
-15 -14 -13 -12 -11 10 0
-15 -14 -13 -12 11 -10 0
-15 -14 -13 -12 11 10 0
-7 -6 -5 -4 -3 -2 0
-7 -6 -5 -4 -3 2 0
-7 -6 -5 -4 3 -2 0
-7 -6 -5 -4 3 2 0
185 0

Note that: 2^{51639} is a very big number !!!

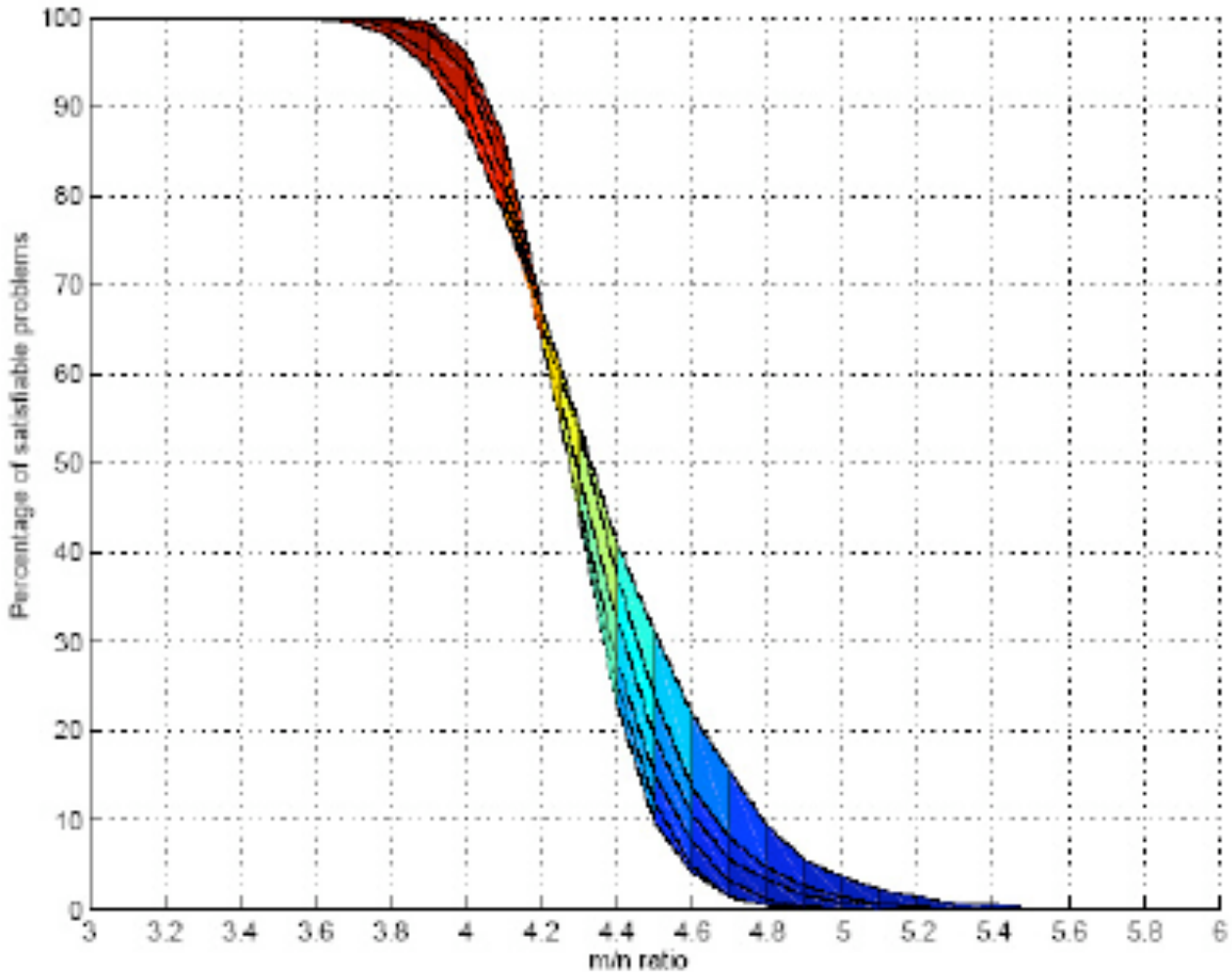
$$2^{100} = 1{,}267{,}650{,}000{,}000{,}000{,}000{,}000{,}000{,}000$$

Dew_Satz SAT solver (Anbulagan, 2005) solves this instance in 36 seconds.

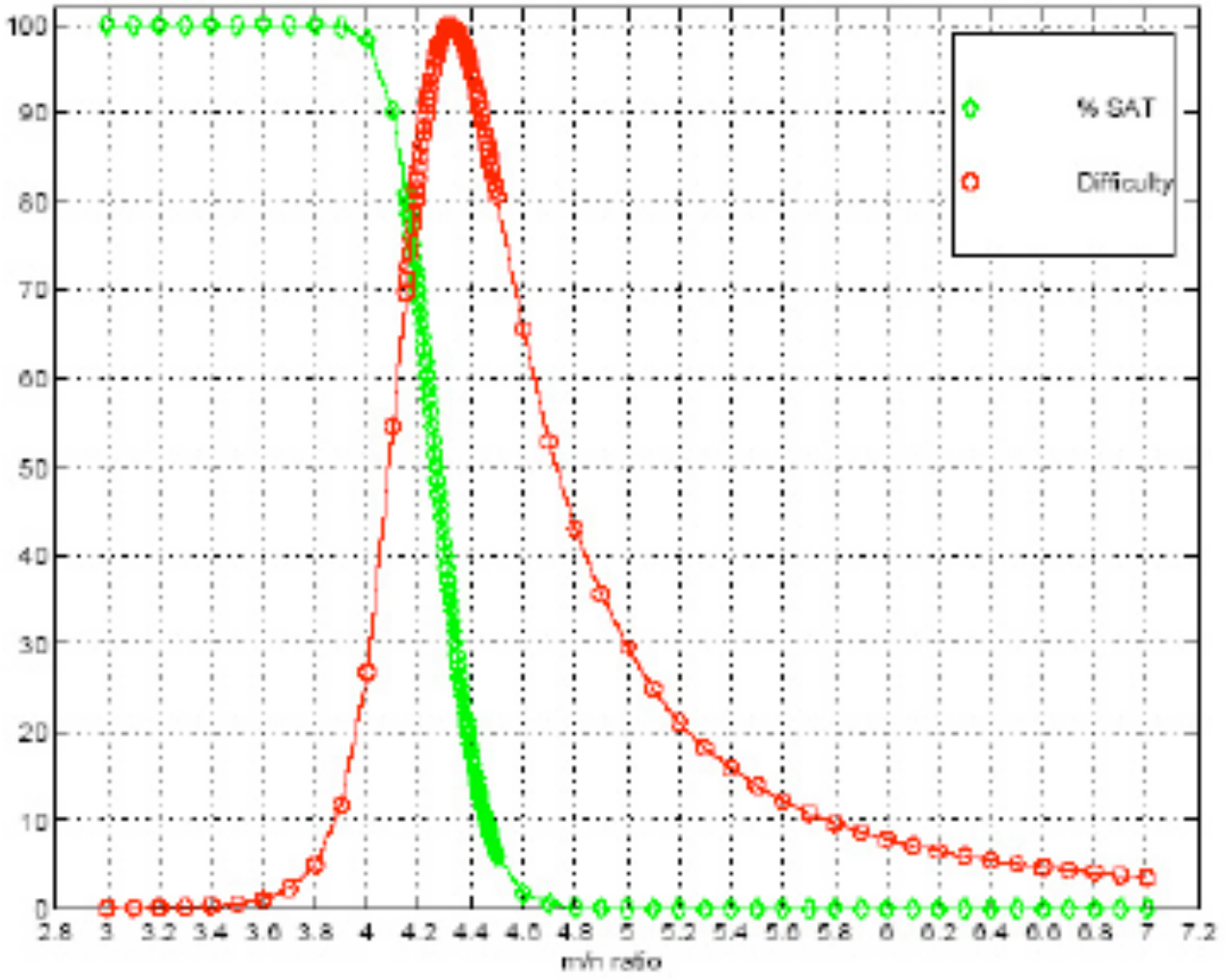
Phase Transition in Random Problems

- Also called threshold phenomenon
- A phenomenon in which satisfiability problems suddenly change from easy (100% SAT) to hard (100% UNSAT) as abruptly as water freezing into ice.
- r1=0; r2=1; r3≅4.258;
- r = ratio number of clauses to number of variables (clause density).
- $r\alpha$ = critical value for formula with fixed clause lengths α.
- The critical value divides the space of SAT formulas into 3 regions, such as:
  - Under-constraint region: almost all the formulas are satisfiable and easy to solve.
  - Hard-constraint region: the region of 50% satisfiable formulas and hard to solve.
  - Over-constraint region: almost all the formulas are unsatisfiable and easy to solve.

Phase Transition: n=60:20:160



Phase Transition and Difficulty Level: n=200



How to Solve the Problems

- Complete method: guarantee to obtain a solution
  - Based on DPLL procedure [Davis et. al., 1962]
    - Enhanced by look-ahead: Satz, Dew_Satz, kcnfs, march_dl, …
    - Enhanched by CDCL: GRASP, RELSAT, Chaff, zChaff, MiniSat, Siege, Berkmin, Jerusat, Tinisat, …
- Stochastic method: no guarantee to obtain a solution
  - Stochastic Local Search:
    - Random Walk: WalkSAT, AdaptNovelty$^+$, g2wsat, R+AdaptNovelty$^+$, …
    - Clause Weighting: SAPS, PAWS, DDFW, R+DDFW$^+$
  - Evolutionary algorithms
  - Neural networks
  - etc…
- Hybrid approach

Complete Method: Davis Putnam Procedure

- The original procedure (DP) used a resolution rule, leading to potentially exponential use of space. [Davis & Putnam, 1960]
- Davis, Logemann and Loveland replaced the resolution rule with a splitting rule. The new procedure is know as the DPLL or DPL procedure. [Davis et al., 1962]
- Despite its age, still one of the most popular and successful complete methods. Basic framework for many modern SAT solvers.
- Exponential time is still a problem.

DP Procedure

Procedure DP($\boldsymbol{F}$**)**

for l = 1 to NumberofVariableIn($\boldsymbol{F}$)

choose variable x ∈ VAR($\boldsymbol{F}$)

Resolvants = Ø

forall (C1,C2) such that

C1 ∈ $\boldsymbol{F}$, C2 ∈ $\boldsymbol{F}$, x ∈ C1, ¬x ∈ C2

Resolvants = Resolvants ∪ resolve(C1,C2)

// don't generate tautological resolvants.

C_x = {C: C ∈ $\boldsymbol{F}$ and x ∈ C}

$\boldsymbol{F}$ = $\boldsymbol{F}$ - C_x

$\boldsymbol{F}$ = $\boldsymbol{F}$ ∪ Resolvants

// x is not in any clause in Resolvants. So now x is not in $\boldsymbol{F}$.

if $\boldsymbol{F}$ = Ø **return** UNSATISFIABLE

else return SATISFIABLE

Resolution for SAT example

$$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_6) \wedge (\neg x_2 \vee x_5)$$

$$\Downarrow$$

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

$$\Downarrow$$

$$(x_1 \vee x_5 \vee \neg x_6)$$

$\Rightarrow$ SAT

Resolution for UNSAT example

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

$$\Downarrow$$

$$(x_2) \wedge (x_2 \vee \neg x_2) \wedge (\neg x_2 \vee x_2) \wedge (\neg x_2)$$

$$\Downarrow$$

$$\varnothing$$

$\Rightarrow$ UNSAT

Resolution for UNSAT example

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

$$\Downarrow$$

$$(x_1) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

$$\Downarrow$$

$$(x_3) \wedge (\neg x_3)$$

$$\Downarrow$$

$$\emptyset$$

$\Rightarrow$ UNSAT

NATIONAL ICT AUSTRALIA

DPLL Procedure

Procedure DPLL(F)

(Sat) if $F = \emptyset$ then "SAT"

(Empty) if F contains the empty clause then"UNSAT"

(Unit Pr) if F has unit clause $\{u\}$, then DPLL($F\{u$/true$\}$)

(Pure) if F has pure literal p, then DPLL($F\{p$/true$\}$)

(Split) if DPLL($F\{l$/true$\}$) is satisfiable then "SAT"

else DPLL($F\{l$/false$\}$)

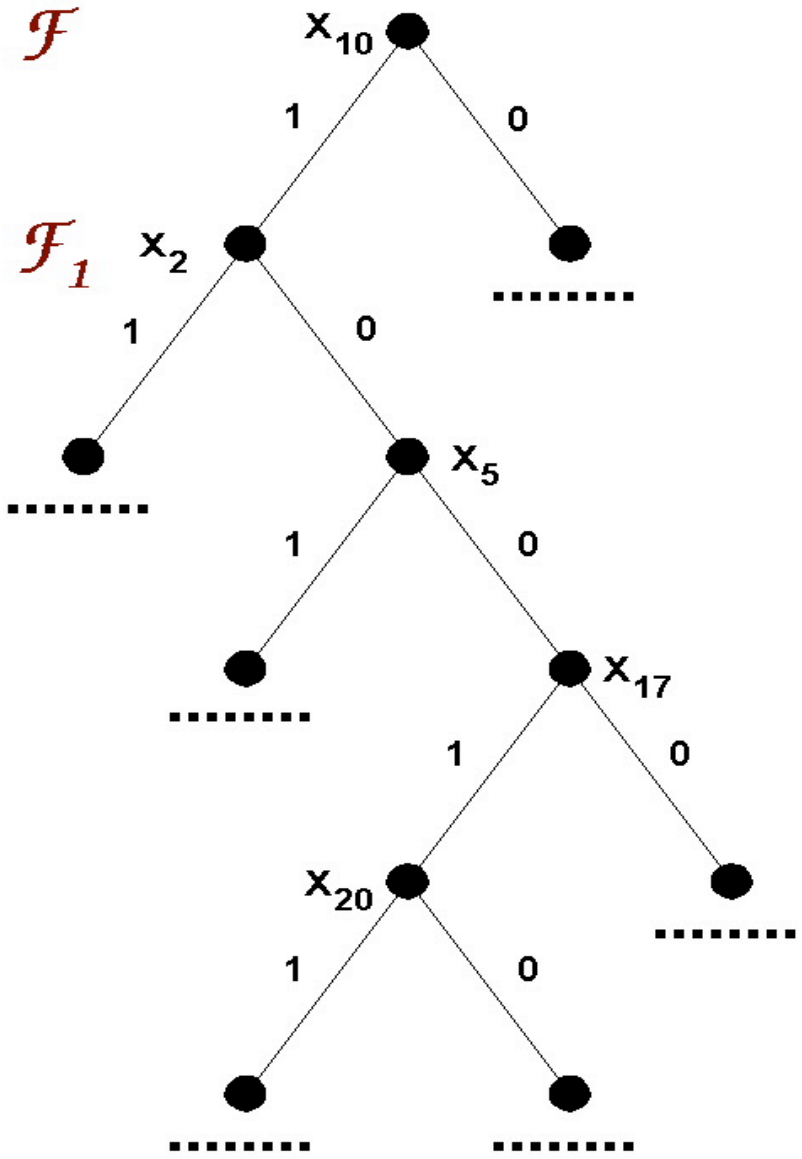
Unit Propagation (UP) is also called Boolean Constraint Propagation (BCP). It is used to detect unit clauses and conflict clauses.

DPLL: Basic Notation & Definitions

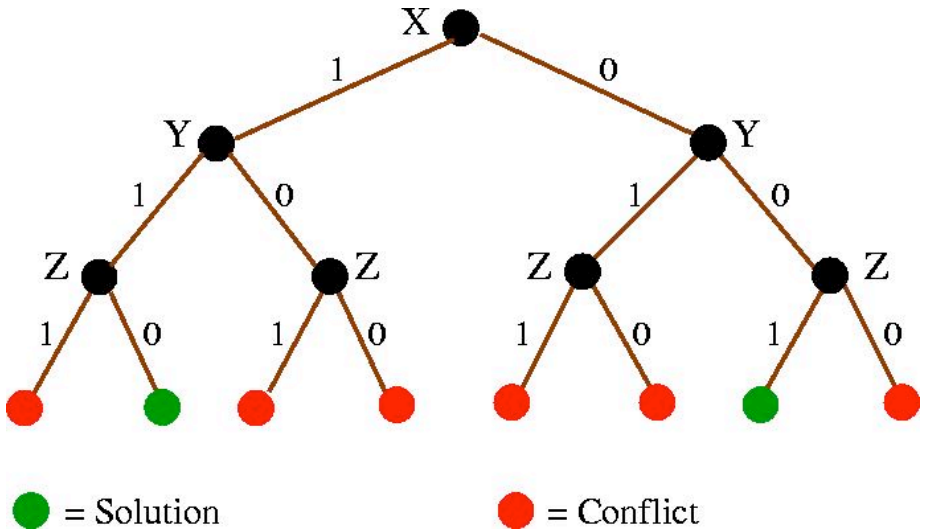
- Branching variable: a variable that can have 2 sub-branches
- Free variable: a variable without any value
- Contradiction / dead-end / conflict: an empty clause is found
- Backtracking: An algorithmic technique to find solutions by trying one of several choices. If the choice proves incorrect, computation *backtracks* or restarts at the point of choice and tries another choice.

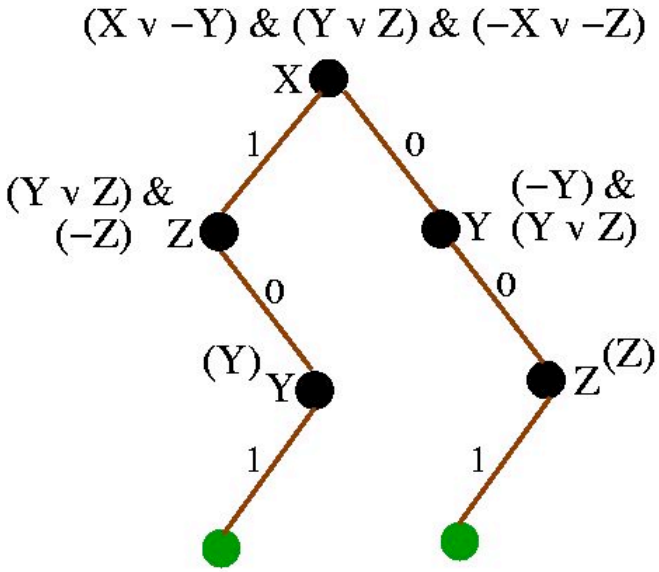
Search Tree of DPLL Procedure

- Binary Search Tree
- Large search tree size ⇔ Hard problem
- The hardness of a random SAT problem is independent from the SAT Solver used.
- Depth first search with backtracking



DPLL Tree of Student-Courses Problem





Using DPLL Procedure

DPLL Performance: Original vs. Variants

- The worst case complexity of the algorithm in our experience is $O(2^{n/21.83-1.70})$, based on UNSAT problems.
- This is an improvement!… Notice that for example,

2^{100} = 1,267,650,000,000,000,000,000,000,000,000

	for n=100	#nodes
DPLL (1962):	$O(2^{n/5})$	= 1,048,576
Satz (1997):	$O(2^{n/20.63+0.44})$	= 39
Satz215 (1999):	$O(2^{n/21.04-1.01})$	= 13
Kcnfs (2003):	$O(2^{n/21.10-1.35})$	= 10
Opt_Satz (2003):	$O(2^{n/21.83-1.70})$	= 7

- look-ahead-enhanced DPLL based SAT solver can reliably solve problems with up to 700 variables.

Heuristics for DPLL Procedure

Objective: to reduce search tree size by choosing a best branching variable at each node of the search tree.

Central issue:

How to select the next best branching variable?

Branching Heuristics in DPLL Procedure

- Simple
  - use simple heuristics for branching variable selection
  - Based on a literal or a variable occurrences counting

- Sophisticated
  - use sophisticated heuristics for branching variable selection.
  - Need more resources and efforts.

Simple Branching Heuristics

- MOMS (Maximum Occurrences in Minimum Sized clauses) heuristics: pick the literal that occurs most often in the minimum size clauses.
  - Maximum binary occurrences
  - too simplistic
  - CSAT [Dubois et. al, 1993]

- Jeroslow-Wang's heuristics [Jeroslow & Wang, 1990; Hooker & Vinay, 1995]: estimate the contribution of each literal l to satisfying the clause set and pick the best

$$\text{score}(l) = \sum_{c \in F \;\&\; l \in c} 2^{-|c|}$$

for each clause c the literal l appears in $2^{-|c|}$ is added where $|c|$ is the number of literals in c.

Sophisticated Branching Heuristics

- Look-ahead-based DPLL
- Unit Propagation Look-Ahead(UPLA) heuristics
  - Satz [Li & Anbulagan, 1997]
- Backbone Search heuristics
  - kcnfs [Dequen, 2003]
- Dynamic Variable Filtering(DVF) heuristics
  - ssc34 and ssc355 [Anbulagan, 2004]
  - LAS+NVO
- LAS+NVO+DEW heuristics
  - Dew_Satz [Anbulagan & Slaney, 2005]

Satz - Complete SAT Solver

- No physical modification on variables and clauses
- An efficient backtracking management
- Count the number of clauses at each node
- Using UPLA Heuristic for detecting contradictions earlier.
- Resolvents resolution as pre-processing (3-Resolution)
- Open for the integration of new ideas

UPLA Heuristics

- UPLA = Unit Propagation Look-Ahead
- Goal: to find the best branching variable by performing UPLA.
- UPLA was one of the main improvements to the DPLL procedure, after more than 3 decades.
- Paper: "Heuristics based on Unit Propagation for Satisfiability Problems" [Li and Anbulagan, IJCAI-1997].

Predicate PROP in UPLA of Satz

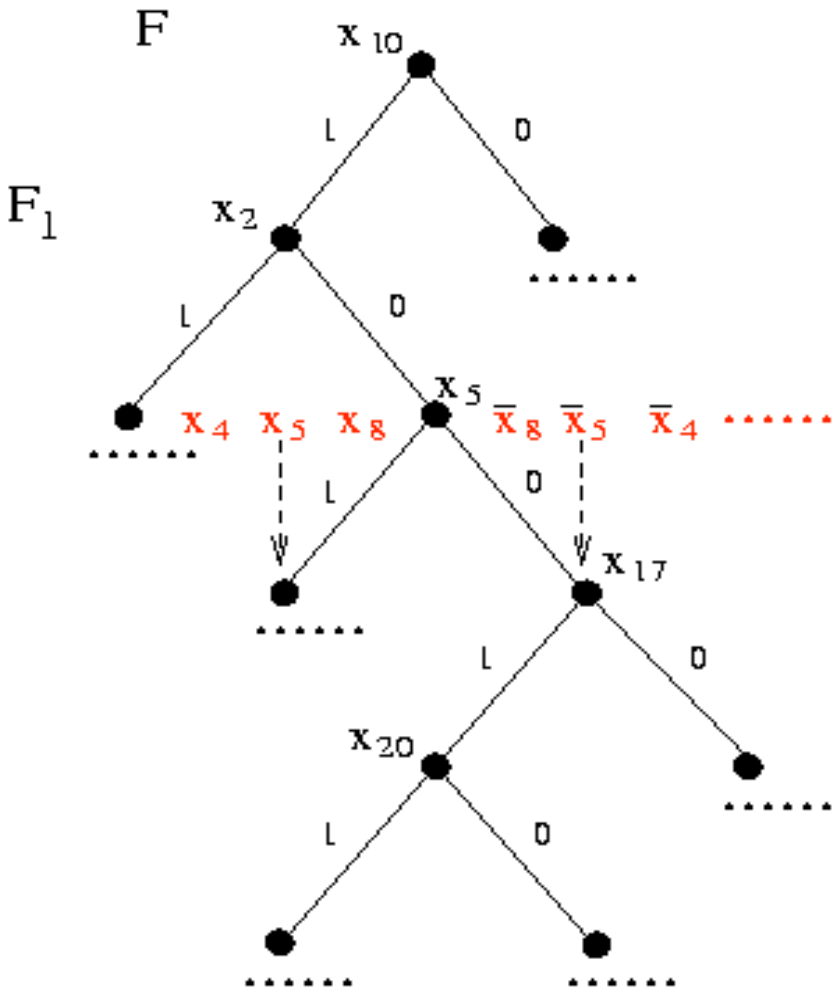
Let *PROP* be a binary predicate such that *PROP(x,i)* is true iff *x* is a variable that occurs both positively and negatively in binary clauses and occurs in at least *i* binary clauses in *F*, and let *T* be an integer, then $PROP_z(x)$ is defined to be the first of the three predicates *PROP(x,4)*, *PROP(x,3)*, *true* (in this order) whose denotational semantics contains more than T variables.

T is fixed to 10 in Satz.

UPLA-based Branching Variable Selection

Algorithm 1 LA-BranchingRule($\mathcal{F}$)

1: **for** each variable $x_i \in \mathcal{V}$ **do**
2: Let $\mathcal{F}_i'$ and $\mathcal{F}_i''$ be two copies of $\mathcal{F}$;
3: $\mathcal{F}_i' := \mathrm{UP}(\mathcal{F}_i' \cup \{x_i\})$;
4: $\mathcal{F}_i'' := \mathrm{UP}(\mathcal{F}_i'' \cup \{\bar{x}_i\})$;
5: **if** empty clause $\in \mathcal{F}_i'$ **and** empty clause $\in \mathcal{F}_i''$ **then**
6: **return** "unsatisfiable";
7: **else if** empty clause $\in \mathcal{F}_i'$ **then**
8: $\mathcal{F} := \mathcal{F}_i''$;
9: **else if** empty clause $\in \mathcal{F}_i''$ **then**
10: $\mathcal{F} := \mathcal{F}_i'$;
11: **else**
12: $w(x_i) := diff(\mathcal{F}_i', \mathcal{F})$;
13: $w(\bar{x}_i) := diff(\mathcal{F}_i'', \mathcal{F})$;
14: $\mathcal{W}(x_i) := w(x_i) * w(\bar{x}_i) + w(x_i) + w(\bar{x}_i)$;
15: **end if**
16: **end for**
17: **return** x_i with highest $\mathcal{W}(x_i)$ to branch on;



LAS+NVO Heuristics

Algorithm 3 NVO-LAS-BranchingRule($\mathcal{F}$)

```
1: Push each variable x_i ∈ V to NVO_STACK;
2: repeat
3:    F_init := F;
4:    for each variable x_i ∈ NVO_STACK do
5:       Let F'_i and F''_i be two copies of F;
6:       F'_i := UP(F'_i ∪ {x_i});
7:       F''_i := UP(F''_i ∪ {x̄_i});
8:       if empty clause ∈ F'_i and empty clause ∈ F''_i then
9:          return "unsatisfiable";
10:      else if empty clause ∈ F'_i then
11:         F := F''_i;
12:         NVO(x_i);
13:      else if empty clause ∈ F''_i then
14:         F := F'_i;
15:         NVO(x_i);
16:      else
17:         w(x_i) := diff(F'_i, F);
18:         w(x̄_i) := diff(F''_i, F);
19:         W(x_i) := w(x_i) * w(x̄_i) + w(x_i) + w(x̄_i);
20:      end if
21:   end for
22: until F = F_init
23: NVO(x_i);
24: return x_i with highest W(x_i) to branch on;
```

- LAS = look-ahead Saturation
- NVO = Neighbourhood Variables Ordering
- The idea of integrating LAS: do UPLA process until the sub-formulae at each node becomes non-reducible. Then execute MOMS heuristic to choose a best branching variable.
- Result: Success in finding a best branching variable at each node and reduce significantly the number of branching nodes.
- The idea of integrating NVO: attempt to limit the number of free variables examined by exploring next only the neighbourhood variables of the current assigned variable.

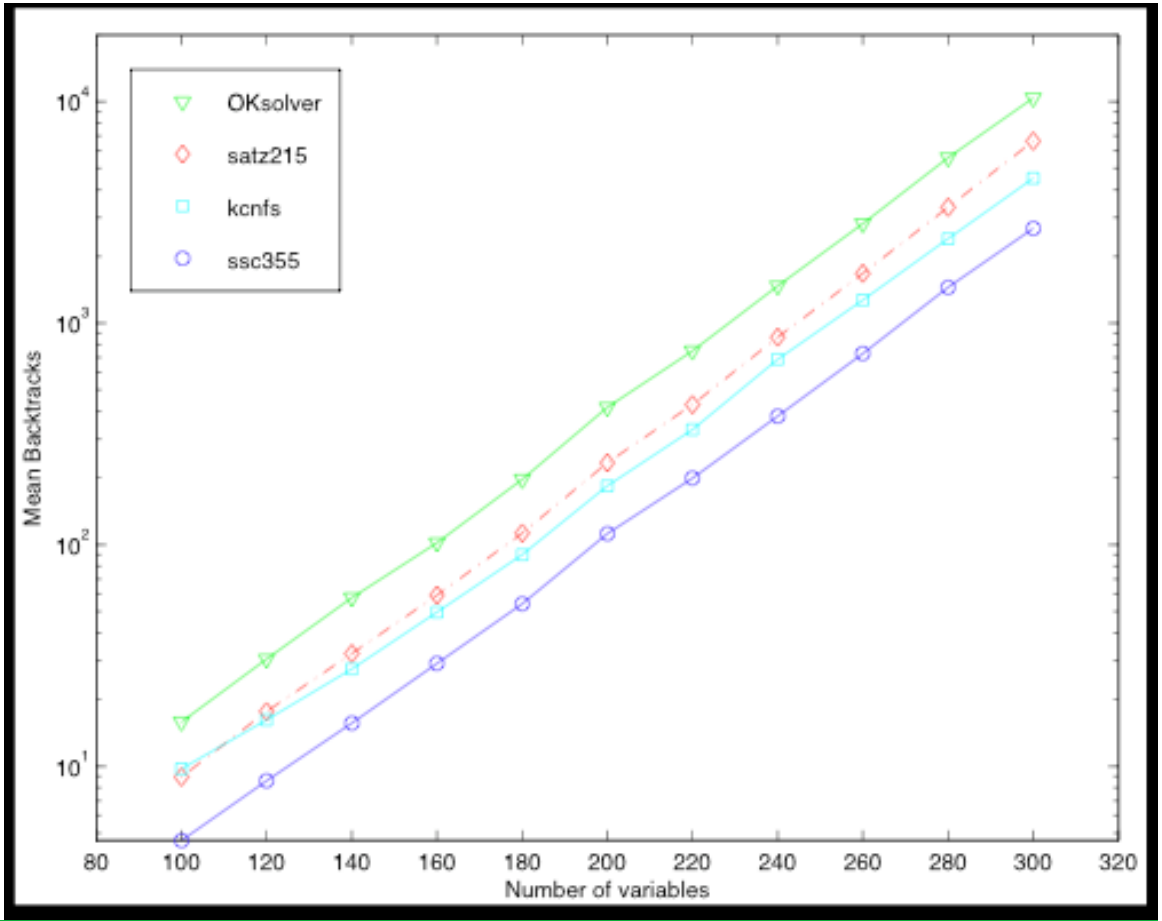
Empirical Results on Security problem (cnf-r3*)

			Search tree size (nodes)			Runtime (seconds)		
Probs.	# Vars	#Cls	satz215	ssc34	ssc355	satz215	ssc34	ssc355
b1-k1.1	21536	8966	2008485	1265	3551	2966	71	124
b1-k1.2	152608	8891	N/A	3002	1500	>3600	174	53
b2-k1.1	152608	17857	128061	0	0	792	1.05	0.88
b2-k1.2	414752	17960	181576	0	0	1254	1.19	1.09
b3-k1.1	283680	26778	31647	0	0	448	1.89	1.51
b3-k1.2	676896	27503	38279	0	0	600	2.25	1.64
b4-k1.1	414752	35817	11790	0	0	348	3.00	2.41
b4-k1.2	939040	35963	20954	0	0	624	3.37	2.71

ssc34 uses LAS, while ssc355 uses LAS+NVO

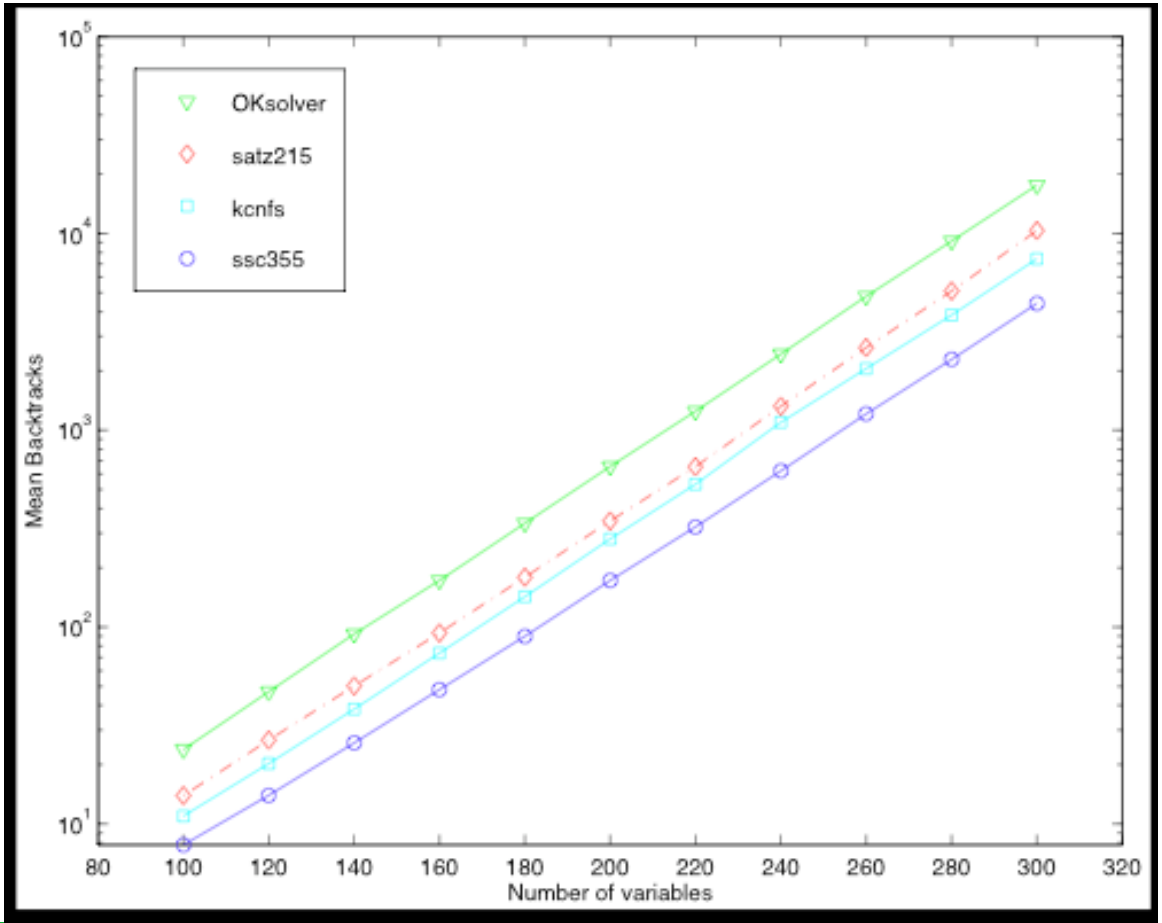
Empirical Results on Random 3-SAT Problem

Mean search tree size of each DPLL procedure as a function of nb. of variables for hard random 3-SAT problems at ratio 4.25 (1000 problems are solved at each point)



Empirical Results on Random 3-SAT Problem

Mean search tree size of each DPLL procedure as a function of number of variables for hard random unsatisfiable 3-SAT problems at ratio 4.25



Empirical Results on Random 3-SAT Problem

- On hard random 3-SAT problems with 350 variables (300 problems are solved), mean search tree size of:
  - Satz215: 36156 branching nodes
  - kcnfs: 24669 branching nodes
  - ssc34: 15507 branching nodes
  - ssc355: 13675 branching nodes
- Search tree size of ssc355 is 164% and 80% smaller, respectively than those for Satz215 and kcnfs.

More Reasoning & Less Searching

- Problem : v350c1488g255 (unsatisfiable)

	# Branch. Nodes	Runtime (s)
ssc355	65,784	189
kcnfs	93,655	40
Satz215	123,735	61
OKsolver	275,159	438
MiniSat	25,456,254	1660
Siege	n/a	> 9000
zChaff	n/a	> 9000
Tinisat	n/a	> 9000

- Number of Branching Nodes versus Runtime
- More reasoning at each node increases the runtime cost.

LA+Backbone Variable Detection Heuristics

- Backbone variable is a variable which is assigned the same value for all solutions to the SAT/CSP problems.
- Such variables are also called frozen variables.
- Detection of backbone variables during LA process.
- The cnfs and kcnfs solvers implemented a pseudo-backbone variables detection heuristic.

Backbone Variable: an example

$$(x_1 \vee \neg x_2) \wedge$$
$$(x_1 \vee \neg x_3) \wedge$$
$$(\neg x_1 \vee x_7) \wedge$$
$$(\neg x_1 \vee x_8) \wedge$$
$$(x_4 \vee \neg x_7 \vee \neg x_8) \wedge$$
$$(\neg x_4 \vee x_5 \vee x_6) \wedge$$
$$(x_4 \vee x_2 \vee x_3)$$

Find the backbone variable(s)!

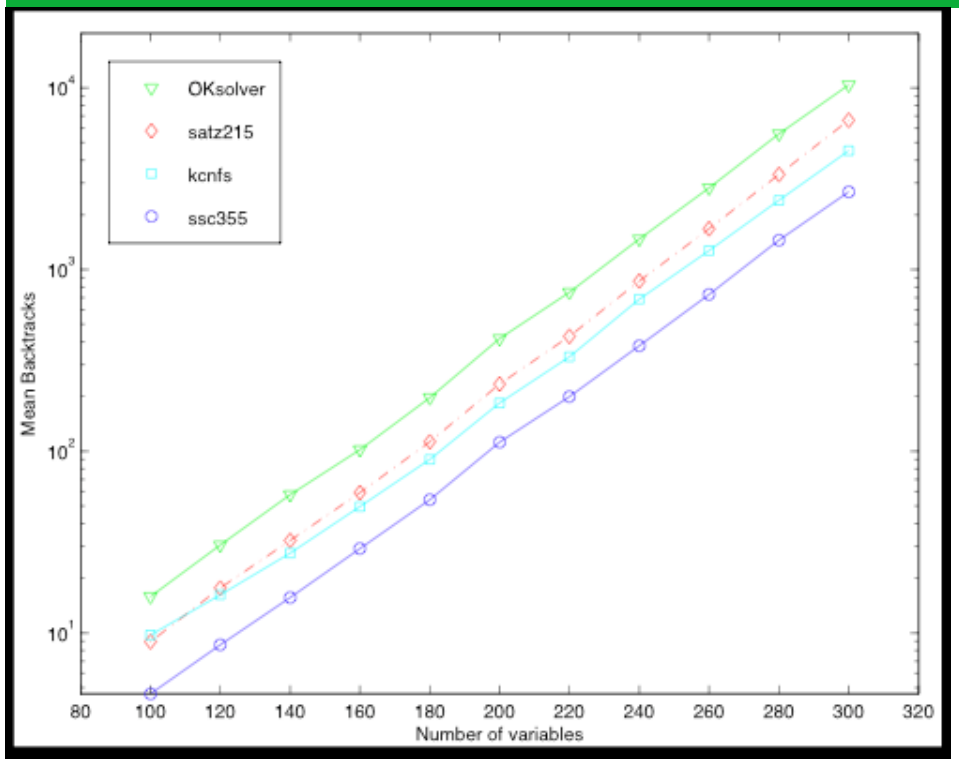
Backbone Variable: an example

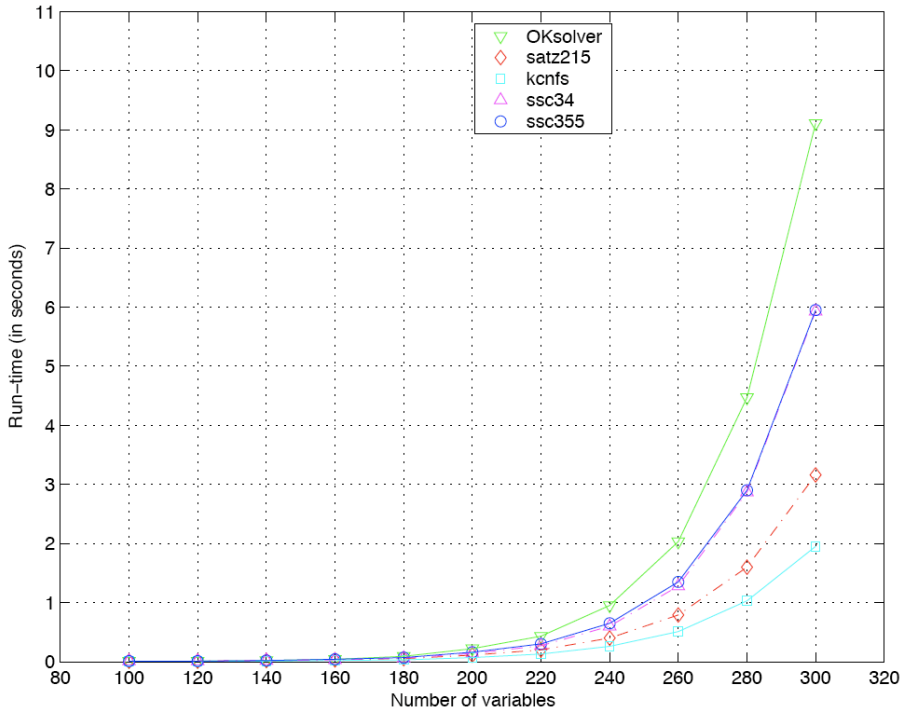
$(x_1 \vee \neg x_2) \wedge$
$(x_1 \vee \neg x_3) \wedge$
$(\neg x_1 \vee x_7) \wedge$
$(\neg x_1 \vee x_8) \wedge$
$(x_4 \vee \neg x_7 \vee \neg x_8) \wedge$
$(\neg x_4 \vee x_5 \vee x_6) \wedge$
$(x_4 \vee x_2 \vee x_3)$

Find the backbone variable(s)!

The answer is x_4

Comparison Results





LAS+NVO+DEW Heuristics

Algorithm 2 DewSatz-BranchingRule($\mathcal{F}$)

```
1: Push each variable x_i ∈ V to NVO_STACK at the root node;
2: repeat
3:    B := ∅;
4:    F_init := F;
5:    for each variable x_i ∈ NVO_STACK do
6:       Let F'_i and F''_i be two copies of F;
7:       if w(x_i) = 0 then
8:          F'_i := UP(F'_i ∪ {x_i});
9:       if w(x̄_i) = 0 then
10:         F''_i := UP(F''_i ∪ {x̄_i});
11:      if empty clause ∈ F'_i and empty clause ∈ F''_i then
12:         return UNSATISFIABLE;
13:      else if empty clause ∈ F'_i then
14:         F := F''_i;
15:         NVO(x_i);
16:      else if empty clause ∈ F''_i then
17:         F := F'_i;
18:         NVO(x_i);
19:      else
20:         B := B ∪ {x_i};
21:         w(x_i) := diff(F'_i, F);
22:         w(x̄_i) := diff(F''_i, F);
23:         Compute_DEW(x_i);
24: until F = F_init
25: for each variable x_i ∈ B do
26:    W(x_i) := w(x_i) * w(x̄_i) + w(x_i) + w(x̄_i);
27: NVO(x_i);
28: return x_i with highest W(x_i) to branch on;
```

The basic idea of integrating DEW (dynamic equivalency weighting):

Whenever the binary equivalency clause ($x_i \Leftrightarrow x_j$), which is equivalent to 2 CNF clauses ($\neg x_i \vee x_j$) and ($x_i \vee \neg x_j$), occurs in the formula at a node, Satz needs to perform look-ahead on x_i, $\neg x_i$, x_j, and $\neg x_j$.

As result, variables x_i and x_j will be associated the same weight.

Clearly, the look-aheads on x_j and $\neg x_j$ are redundant, so we avoid them by assigning the implied literal x_j ($\neg x_j$'s) the weight of its parent literal x_i ($\neg x_i$'s), and then by avoiding look-ahead on literals with weight zero.

By doing so, we save two look-aheads.

EqSatz

- Based on Satz
- Enhanced with equivalency reasoning during search process.
  - Substitute the equivalent literals during the search in order to reduce the number of active variables in the current formula.
  - Example: given the clause ($x_i \Leftrightarrow x_j$), we can substitute x_j by x_i.

Equivalency: Reasoning vs. Weighting

On 32-bit Parity Learning problem

A challenging problem [Selman et al., 1997]

Instance (#Vars/#Cls)	Satz	Dew_Satz	EqSatz	Lsat	March_eq	zChaff
par32-1 (3176/10227)	>24h	12,918	242	126	0.22	>24h
par32-2 (3176/10253)	>24h	5,804	69	60	0.27	>24h
par32-3 (3176/10297)	>24h	7,198	2,863	183	2.89	>24h
par32-4 (3176/10313)	>24h	11,005	209	86	1.64	>24h
par32-5 (3176/10325)	>24h	17,564	2,639	418	8.07	>24h
par32-1-c (1315/5254)	>24h	10,990	335	270	2.63	>24h
par32-2-c (1303/5206)	>24h	411	13	16	2.19	>24h
par32-3-c (1325/5294)	>24h	4,474	1,220	374	6.65	>24h
par32-4-c (1333/5326)	>24h	7,090	202	115	0.45	>24h
par32-5-c (1339/5350)	>24h	11,899	2,896	97	6.44	>24h

The first solver which solved all the instances.

Equivalency: Reasoning vs. Weighting

On BMC and circuit-related problems

Problem	Dew_Satz	EqSatz	March_eq	zChaff
barrel6	4.13	0.17	0.13	2.95
barrel7	8.62	0.23	0.25	11
barrel8	72	0.36	0.38	44
barrel9	158	0.80	0.87	66
longmult10	64	385	213	872
longmult11	79	480	232	1,625
longmult12	97	542	167	1,643
longmult13	127	617	53	2,225
longmult14	154	706	30	1,456
longmult15	256	743	23	392
philips-org	697	1974	>5,000	>5,000
philips	295	2401	726	>5,000

NATIONAL ICT AUSTRALIA

Results on Hard Random k-SAT Problems

- Benchmark: from 2005 International SAT Competition.
- Experiment: Each solver was timed out after 200 minutes in the second stage of 2005 International SAT Competition.
- Nb. of hard random k-SAT problems solved by a given DPLL solver.

Solver	Problem		
	SAT (285)	UNSAT (105)	ALL (390)
Kcnfs	92	75	167
Dew_Satz	68	50	118
March_dl	56	43	99

- Dew_Satz won 2 bronze medals for UNSAT and ALL categories.

PSatz

- Parallelization for reduce the problem solving time
- Using dynamic load balancing mechanism based on the work stealing techniques.
- Significant improvement.
- Paper by [Jurkowiak, Li and Utard, 2005]

MaxSatz

- MAX-SAT is an optimisation variant of SAT
- For over-constraint problems
- MAX-SAT is a special case of weighted MAX-SAT where all clauses have weight one.
- The goal: to find a variable assignment that satisfies a maximal number of clauses of a given CNF formula.
- Paper by [Li, Manya and Planes, 2006]

CDCL-based DPLL

Backjumping

- Idea: when a branch fails,
  - Reveal the sub-assignment causing the contradiction (conflict set)
  - Backtrack to the most recent branching point in the conflict set
- A conflict set is constructed from the conflict clause by tracking backwards the unit-implications causing it and by keeping the branching literals.
- When a branching point fails, a conflict set is obtained by resolving the two conflict sets of the two branches.
- May avoid a lot of redundant search.

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$
$(\neg x_1 \vee x_3 \vee x_9) \wedge$
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$
$(\neg x_5 \vee \neg x_6) \wedge$
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$
$(x_1 \vee x_8) \wedge$
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$

.....

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$
$(\neg x_1 \vee x_3 \vee x_9) \wedge$
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$
$(\neg x_5 \vee \neg x_6) \wedge$
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$
$(x_1 \vee x_8) \wedge$
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$

.....

$\{\ldots, \neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}, \ldots\}$ (initial assignment)

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$
$(\neg x_1 \vee x_3 \vee x_9) \wedge$
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$
$(\neg x_5 \vee \neg x_6) \wedge$
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$ removed
$(x_1 \vee x_8) \wedge$ removed
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$

.....

$\{\ldots, \neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}, \ldots, x_1\}$ (branch on x_1)

(unit x_2, x_3)

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$ removed

$(\neg x_1 \vee x_3 \vee x_9) \wedge$ removed

$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$

$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$

$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$

$(\neg x_5 \vee \neg x_6) \wedge$

$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$ removed

$(x_1 \vee x_8) \wedge$ removed

$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$

.....

$\{\ldots, \neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}, \ldots, x_1, x_2, x_3\}$

(unit x_4)

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$ removed
$(\neg x_1 \vee x_3 \vee x_9) \wedge$ removed
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$ removed
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$
$(\neg x_5 \vee \neg x_6) \wedge$
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$ removed
$(x_1 \vee x_8) \wedge$ removed
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$

.....

$\{\ldots, \neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}, \ldots, x_1, x_2, x_3, x_4\}$

(unit x_5, x_6)

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$	removed
$(\neg x_1 \vee x_3 \vee x_9) \wedge$	removed
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$	removed
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$	removed
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$	removed
$(\neg x_5 \vee \neg x_6) \wedge$	conflict
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$	removed
$(x_1 \vee x_8) \wedge$	removed
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$	

.....

$\{\ldots, \neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}, \ldots, x_1, x_2, x_3, x_4, x_5, x_6\}$

Conflict set: $\{\neg x_9, \neg x_{10}, \neg x_{11}, x_1\} \Rightarrow$ backtrack to x_1

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$ removed
$(\neg x_1 \vee x_3 \vee x_9) \wedge$ removed
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$
$(\neg x_5 \vee \neg x_6) \wedge$
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$
$(x_1 \vee x_8) \wedge$
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$

.....

$\{\ldots, \neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}, \ldots, \neg x_1\}$ (branch on $\neg x_1$)

(unit x_7, x_8)

Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$ removed
$(\neg x_1 \vee x_3 \vee x_9) \wedge$ removed
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$
$(\neg x_5 \vee \neg x_6) \wedge$
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$ removed
$(x_1 \vee x_8) \wedge$ removed
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$ conflict

.....

$\{\ldots, \neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}, \ldots, \neg x_1, x_7, x_8\}$

Conflict set: $\{x_{12}, x_{13}, \neg x_1\}$

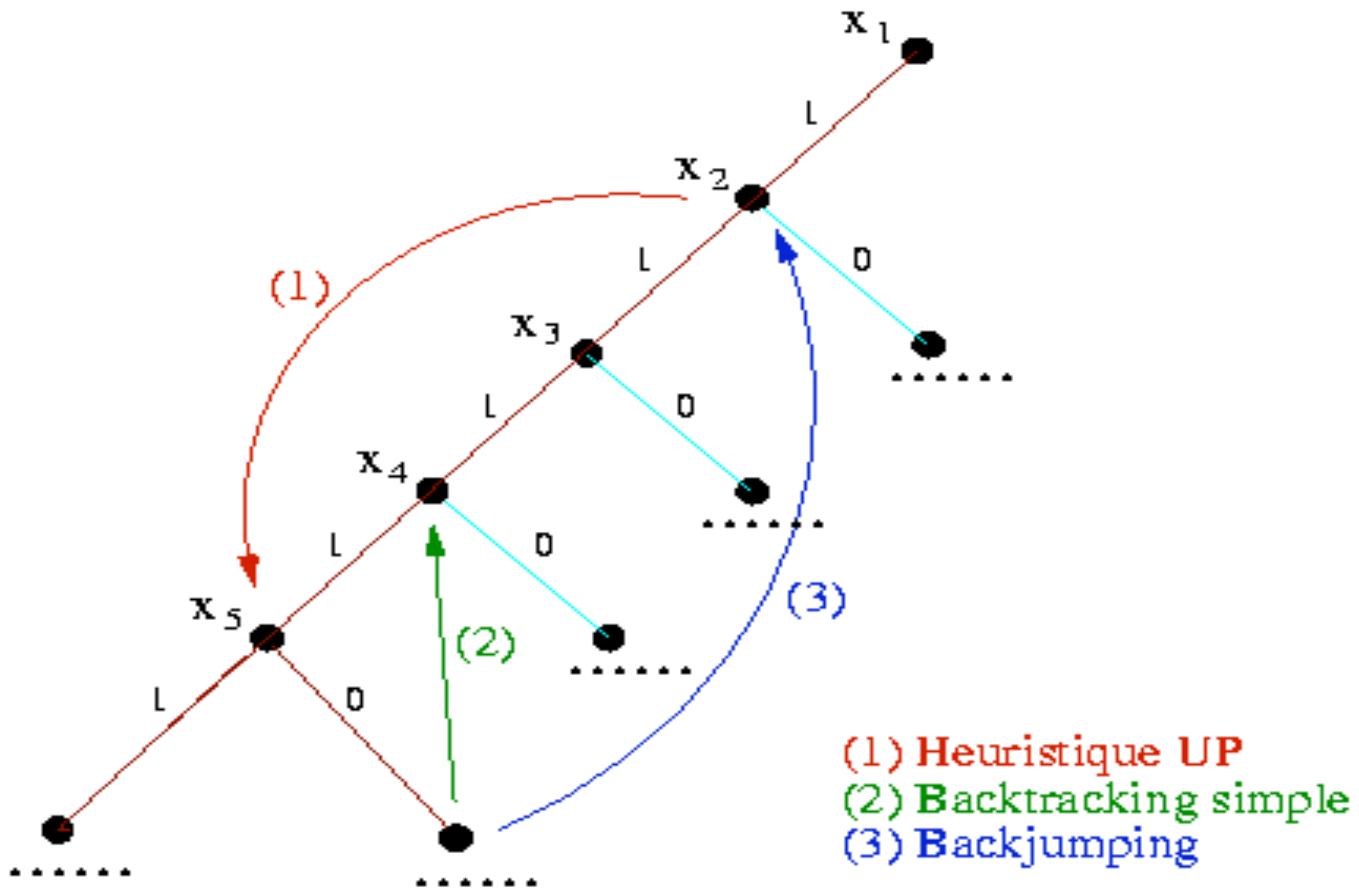
Backjumping: an example

$(\neg x_1 \vee x_2) \wedge$	removed
$(\neg x_1 \vee x_3 \vee x_9) \wedge$	removed
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$	
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$	
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$	
$(\neg x_5 \vee \neg x_6) \wedge$	
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$	removed
$(x_1 \vee x_8) \wedge$	removed
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$	**conflict**

Conflict set: $\{x_{12}, x_{13}, \neg x_1\} \vee \{\neg x_9, \neg x_{10}, \neg x_{11}, x_1\}$

$\Rightarrow \{\neg x_9, \neg x_{10}, \neg x_{11}, x_{12}, x_{13}\} \Rightarrow$ backtrack to x_{13}

Look-Ahead vs. Look-Back



Learning

- Idea: when a conflict set C is revealed, then $\neg C$ can be added to the clause set
  - DPLL will never again generate an assignment containing C.
- May avoid a lot of redundant search.
- Problem: may cause a blow up in space
  - Techniques to control learning and to drop learned clauses when necessary.
- Learning is very effective in pruning the search space for structured problems.

Learning: an example

$(\neg x_1 \vee x_2) \wedge$	removed
$(\neg x_1 \vee x_3 \vee x_9) \wedge$	removed
$(\neg x_2 \vee \neg x_3 \vee x_4) \wedge$	removed
$(\neg x_4 \vee x_5 \vee x_{10}) \wedge$	removed
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge$	removed
$(\neg x_5 \vee \neg x_6) \wedge$	**conflict**
$(x_1 \vee x_7 \vee \neg x_{12}) \wedge$	removed
$(x_1 \vee x_8) \wedge$	removed
$(\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge$	
.....	
$(x_9 \vee x_{10} \vee x_{11} \vee \neg x_1)$	learned clause

Conflict set: $\{\neg x_9, \neg x_{10}, \neg x_{11}, x_1\}$

Learn: $(x_9 \vee x_{10} \vee x_{11} \vee \neg x_1)$

NATIONAL ICT AUSTRALIA

Pseudocode of MiniSat

Algorithm 1 MiniSat

```
 1: loop
 2:    propagate()
 3:    if not conflict then
 4:       if all variables assigned then
 5:          return SATISFIABLE
 6:       else
 7:          decide()
 8:    else
 9:       analyze()
10:       if top-level conflict found then
11:          return UNSATISFIABLE
12:       else
13:          backtrack()
```

Decision Heuristic in CDCL-based Solvers

- VSIDS = Variable State Independent Decaying Sum
  - It keeps a score for each phase of a variable. Initially, the scores are the number of occurrences of a literal in the initial formula. VSIDS increases the score of a variable by a constant whenever an added clause contains the variable. Moreover, as the search progresses, periodically all scores are divided by a constant number.VSIDS will choose a free variable with the highest score to branch.
  - Used in zChaff
  - An improved version is used in MiniSat where variable activities are decayed 5% after each conflict.
- VMTF = Variable Move To Front
  - The initial order of the list is sorted by the occurrence of the variables in the formula. Every time a new clause is learnt and then added to the database, a constant number of the variables from the clause are moved to the front of the list. The list is resorted according to the occurrence of variables in clauses database every time that the restart occurs.
  - Used in Siege

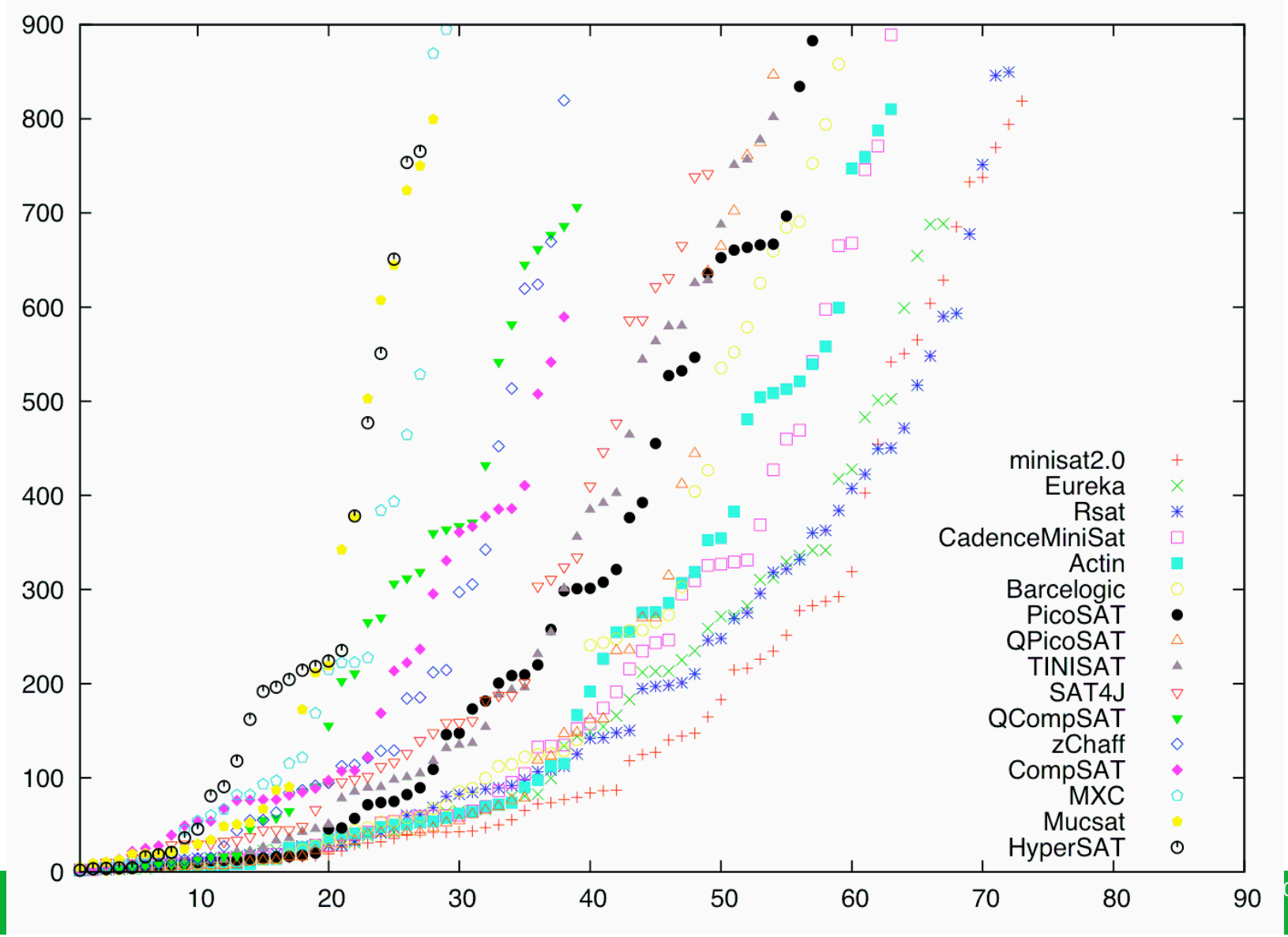
Restart Policy in CDCL-based Solvers

- Abandon the current search space and restart a new one after exceeding certain conditions, such as number of backtracks.
- Increase the backtrack cutoff value by a constant amount to allow solving unsatisfiable formula.
- The clauses learned prior to the restart are considered in the new search. They will help to prune the search space.
- The effect of restarts on the efficiency of clause learning [Huang, 2007]

Watched Literals Mechanism

- For efficient unit propagation and backtrack processes
- Using 2-literal watching in each clause
- The Quest for Efficient Boolean Satisfiability Solver [Zhang and Malik, 2002]

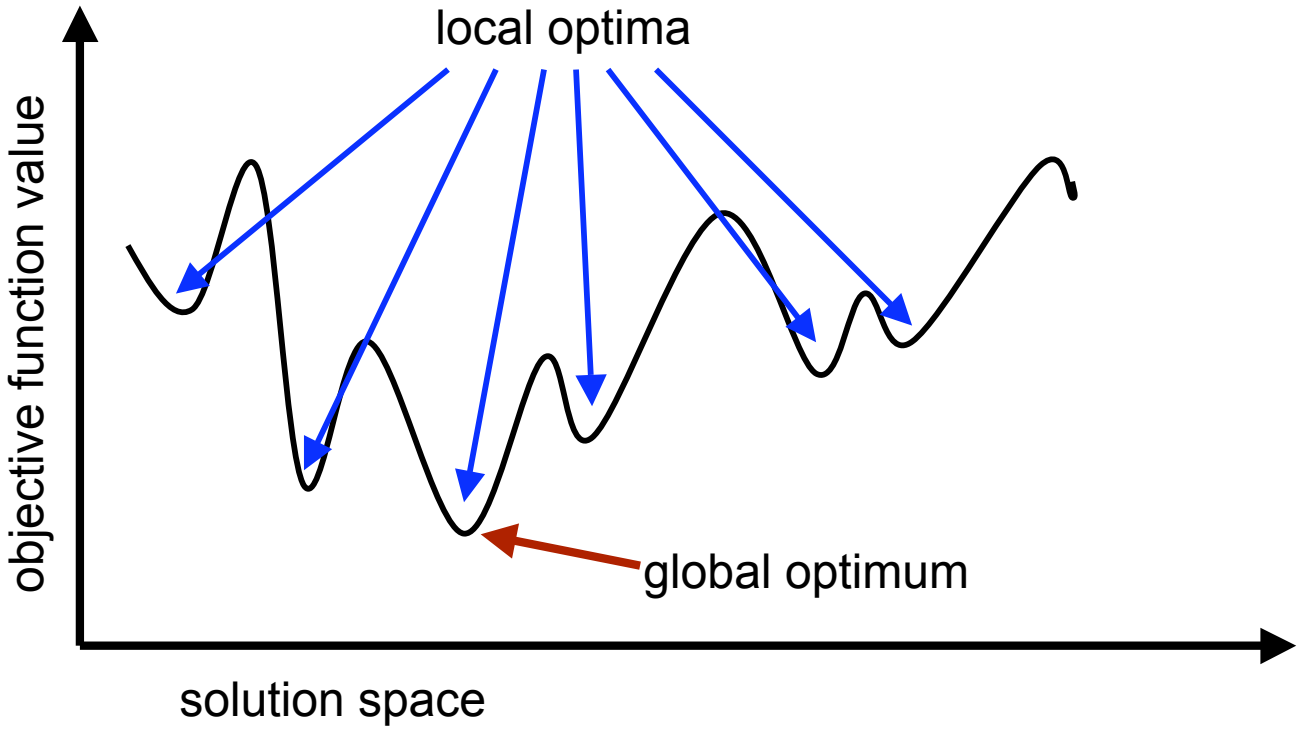
Results of SAT-Race 2006



Stochastic Methods: motivation

- DPLL can reliably solve hard random problems with up to 700 variables.
- … but problems commonly arising in practice often need 100000s - millions of variables.
- We might need "anytime answers" which can provide a "best guess" at any point we stop the algorithm.
- SLS Algorithms:
  - GSAT
  - Random Walk: WalkSAT, AdaptNovelty$^+$, g2wsat
  - Clause Weighting: SAPS, PAWS, DDFW, DDFW$^+$
- Other approach: SP (Survey Propagation)

Local Optima and Global Optimum in SLS



Flipping Coins: The "Greedy" Algorithm

- This algorithm is due to Koutsopias and Papadimitriou
- Main idea: flip variables till you can no longer increase the number of satisfied clauses.

```
Procedure greedy(F)
    T = random (F)    // random assignment
    repeat until no improvement possible
        T = T with variable flipped that increases
                the number of satisfied clauses
    end
```

The GSAT Procedure

- This algorithm is due to Selman, Levesque and Mitchell
- Adds restarts to the simple "greedy" algorithm, and also allows sideways flips.

```
Procedure GSAT(F, MAX_TRIES, MAX_FLIPS)
    for i=1 to MAX_TRIES              // these are the restarts
        T = random(F)                 // random assignment
        for j=1 to MAX_FLIPS          // to ensure termination
            if T satisfies F then return T
            Flip any variable in T that results in greatest increase
                 in number of satisfied clauses
                // it does not matter if the number does not increase.
                // This are the sideways flips
        end
    end
    return "No satisfying assignment found"
End GSAT
```

The WALKSAT Procedure

- The procedure is due to Selman, Kautz and Cohen

```
Procedure WalkSAT(F, MAX_TRIES, MAX_FLIPS, VSH)
    for i=1 to MAX_TRIES      // these are the restarts
        T = random(F)         // random assignment
        for j=1 to MAX_FLIPS  // to ensure termination
            if T satisfies F then return T.
            choose unsatisfied clause C ∈ F at random.
            choose a variable x ∈ C according to VSH.
            T = T with variable x flipped.
        end
    end
    return "No satisfying assignment found"
End WalkSAT
```

AdaptNovelty+

- WalkSAT variants depend on the setting of their noise parameter.
- Noise parameter: to control the degree of greediness in the variable selection process. It takes value between zero and one.
- AdaptNovelty+ is for adaptively tuning the noise level based on the detection of stagnation.

Dynamic Local Search: The basic idea

- Use clause weighting mechanism
  - Increase weights on unsatisfied clauses in local minima in such a way that further improvement steps become possible
  - Adjust weights periodically when no further improvement steps are available in the local neighborhood

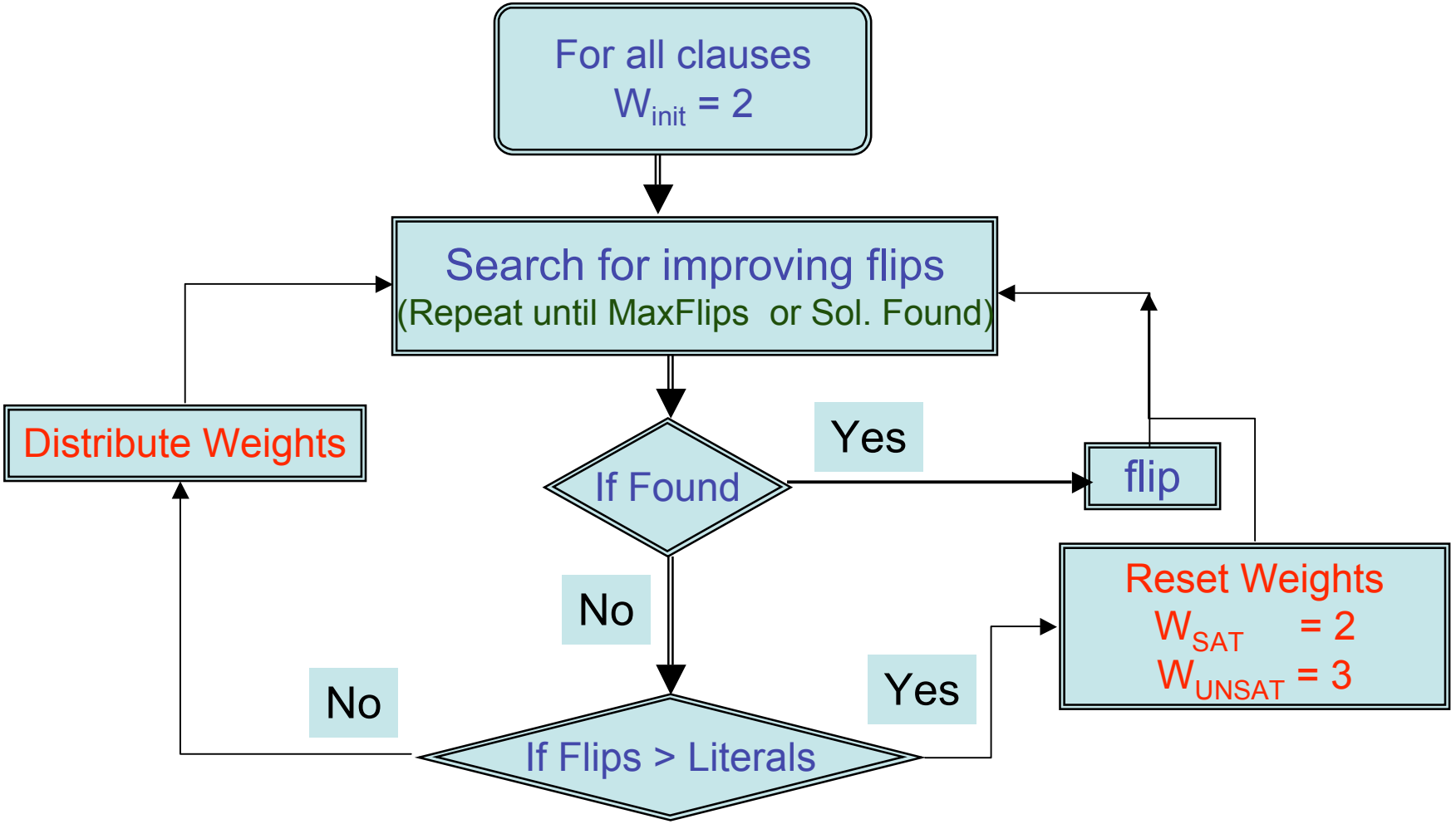
Dynamic Local Search: A brief history

- Breakout Method [Morris, 1993]
- Weighted GSAT [Selman and Kautz, 1993]
- Learning short-term clause weights for GSAT [Frank, 1997]
- Discrete Lagrangian Method (DLM) [Wah and Shang, 1997]
- Smoothed Descent and Flood [Schuurmans and Southey, 2000]
- Scaling and Probabilistic Smoothing (SAPS) [Hutter, Tompkins, and Hoos, 2002]
- Pure Additive Weighting Scheme (PAWS) [Thornton *et al.*, 2004]
- Divide and Distribute Fixed Weight (DDFW) [Ishtaiwi *et al.*, 2005]
- Adaptive DDFW (DDFW^{+}) [Ishtaiwi *et al.*, 2006]

DDFW+

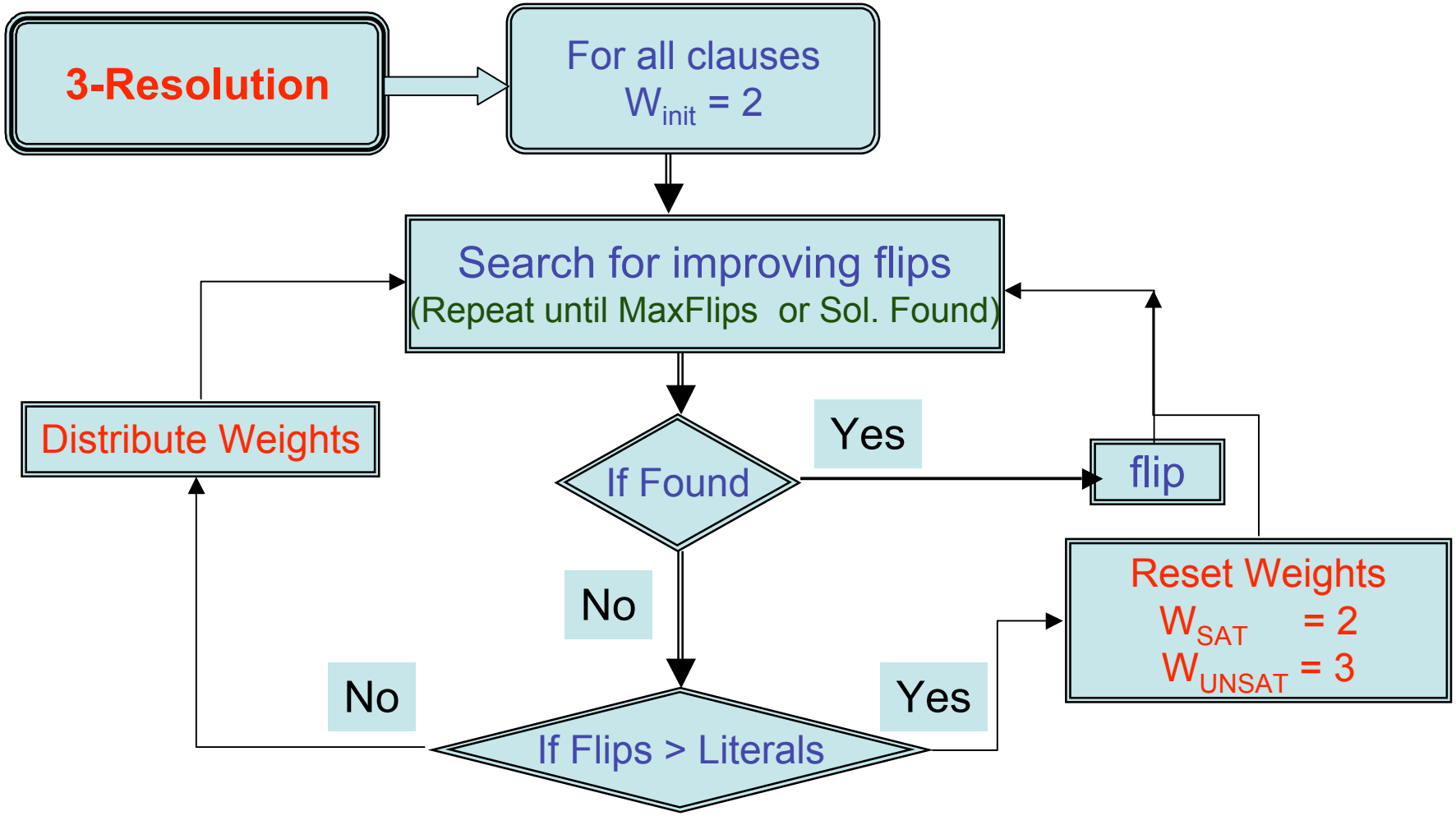
- Adaptive DDFW
- No parameter tuning
- Dynamically alters the total amount of weight that DDFW distributes according to the degree of stagnation in the search.
- The weight initialization value is set at 2 and could be altered during the search between 2 and 3.
- R+DDFW+ is the current best SLS solver for solving random and structured problems

DDFW+



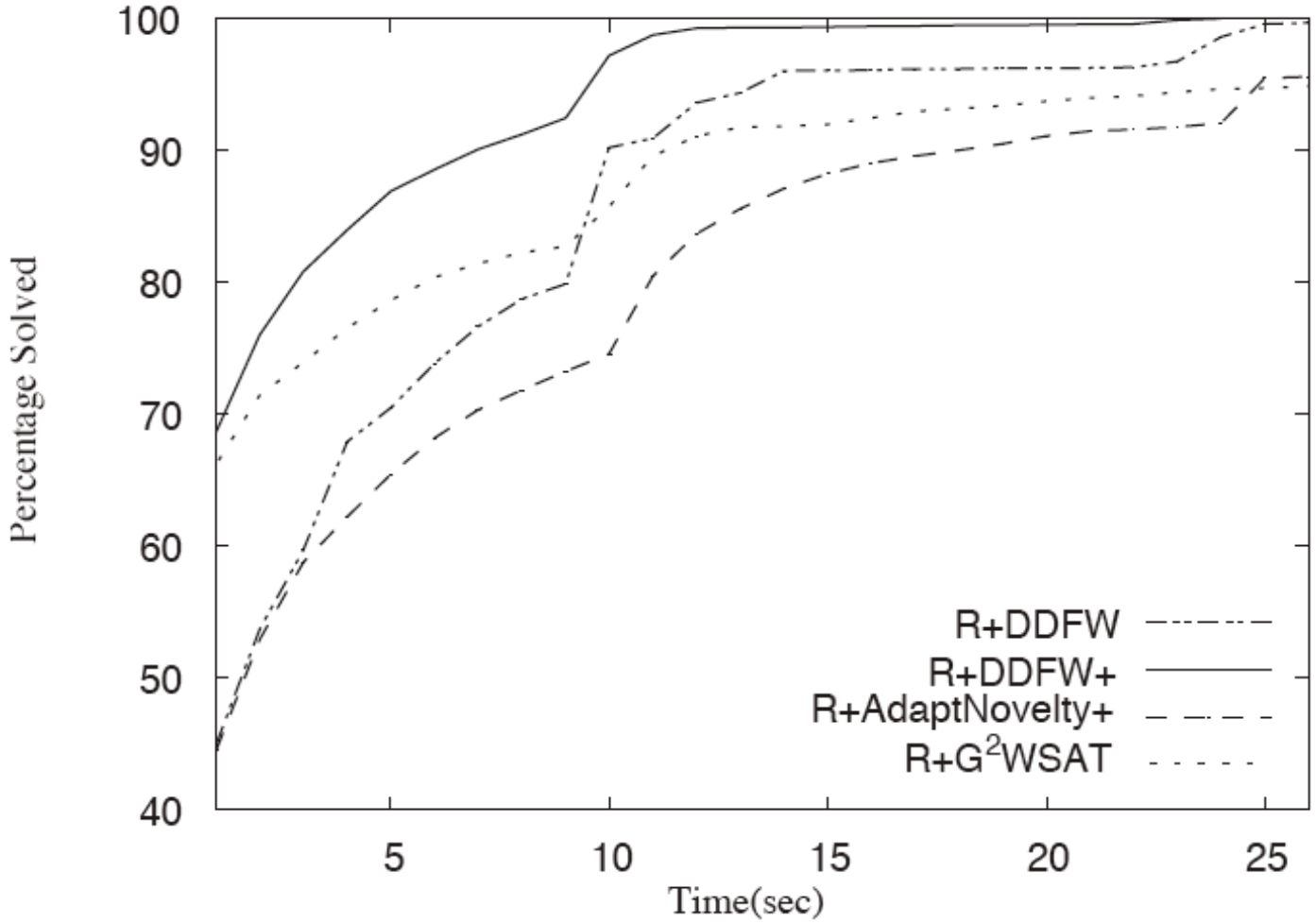
Taken from Abdul Sattar's presentation slide at CP'06

R+DDFW+



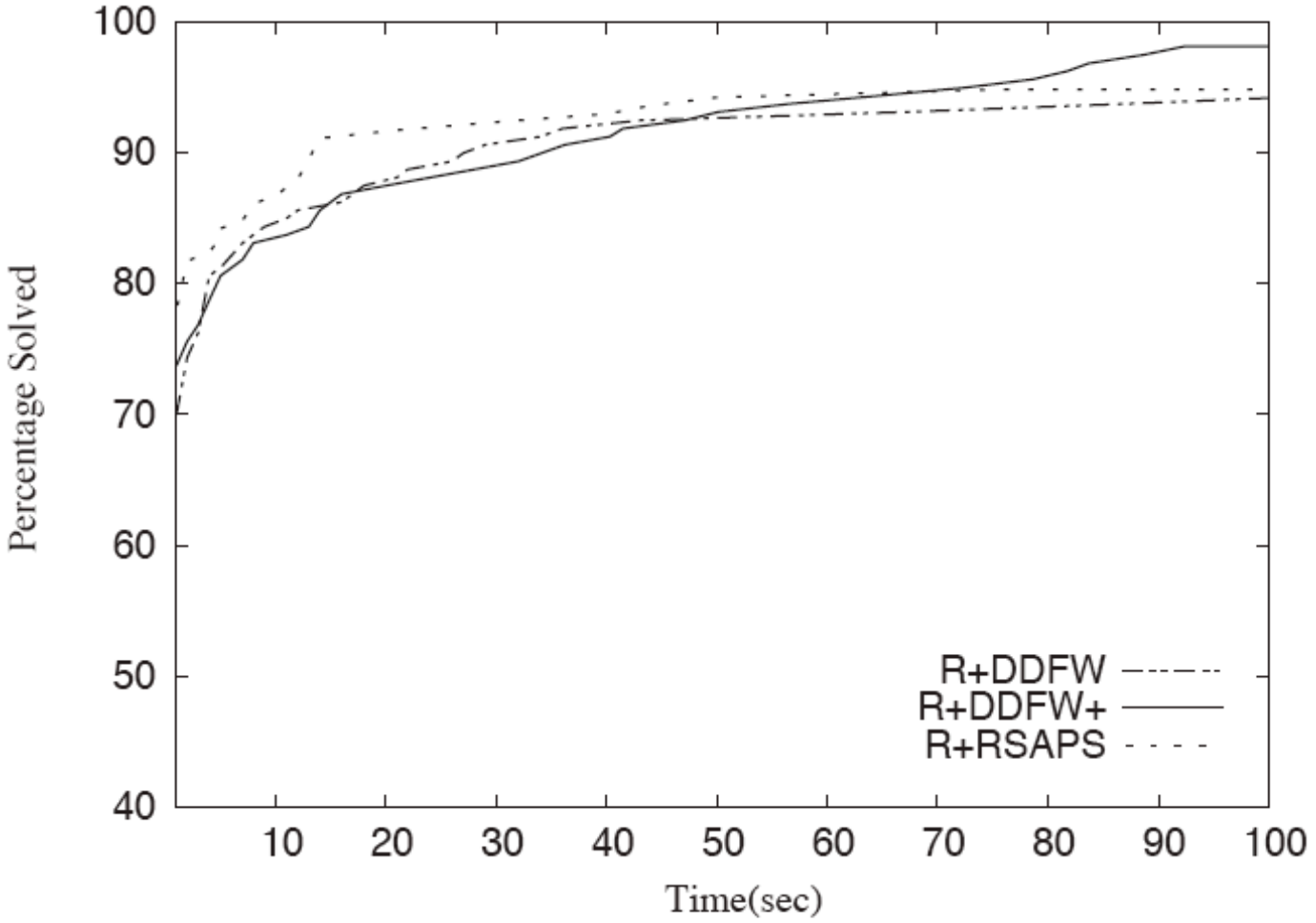
Taken from Abdul Sattar's presentation slide at CP'06

Comparison Results of SLS on Random Problems



a. Random 3SAT problems (50 Instances)

Comparison Results of SLS on Ferry Planning Problems



b. Industrial Ferry problems (16 Instances)

Old Resolution meets Modern SLS

- Adding restricted resolution as a preprocessor
- See [Anbulagan et al., 2005]

Algorithm 1 ComputeResolvents()

```
1:  for each clause c1 of length ≤ 3 in F do
2:    for each literal l of c1 do
3:      for each clause c2 of length ≤ 3 in F s.t. l̄ ∈ c2 do
4:        Compute resolvent r = (c1\{l}) ∪ (c2\{l̄});
5:        if r is empty then
6:          return "unsatisfiable";
7:        else
8:          if r is of length ≤ 3 then
9:            F := F ∪ {r};
10:         end if
11:       end if
12:     end for
13:   end for
14: end for
```

$(x_1 \vee x_2 \vee x_3) \wedge$
$(\neg x_1 \vee x_2 \vee x_4) =$
$(x_2 \vee x_3 \vee \neg x_4)$

From 2005 International SAT Competition

- R+AdaptNovelty$^+$ won the Gold Medal.
- Joint work with IIIS-Griffith University
- Solves 209 of 285 random SAT problems.
- The 2nd and 3rd place are g2wsat (178) and VW (170).
- The 2004 winner, AdaptNovelty$^+$ could only solve 119 problems

Boosting SLS using Resolution

- Old resolution meets modern SLS
  - [Anbulagan et al., AAAI-2005]
  - Limited by using only the 3-Resolution preprocessor.

Resolution-based Preprocessors

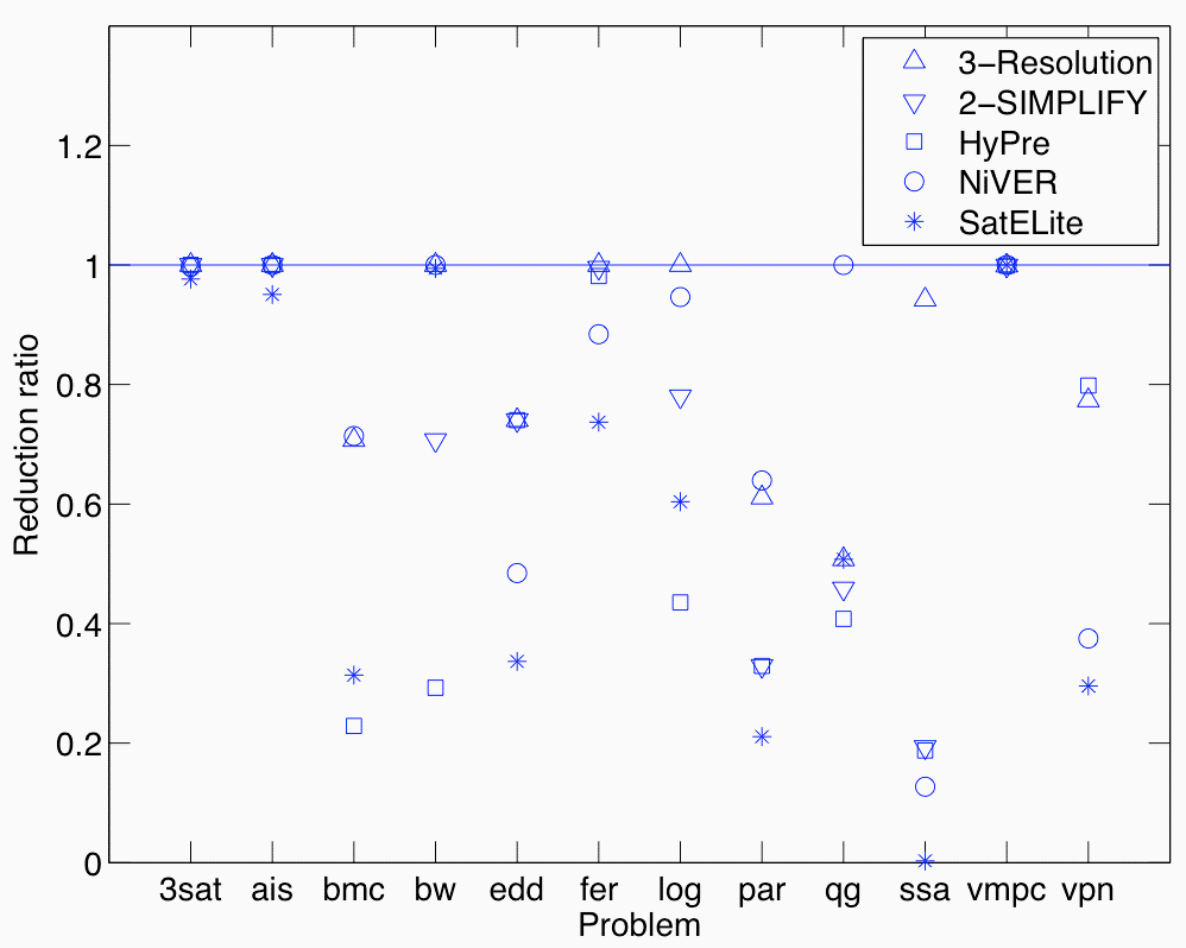
- 3-Resolution [Li and Anbulagan, CP-1997]: computes resolvents for all pairs of clauses of length ≤ 3
- 2-SIMPLIFY [Brafman, IJCAI-2001]: constructs an implication graph from all binary clauses of a problem instance and uses a restricted variant of hyper-resolution.
- HyPre [Bacchus and Winter, SAT-2003]: reasons with binary clauses and do full hyper-resolution.
- NiVER [Subbarayan and Pradhan, SAT-2004]: Non increasing Variable Elimination Resolution.
- SatELite [Eén and Biere, SAT-2005]: improved NiVER with a variable elimination by substitution rule.

Problems

- Hard random 3-SAT (3sat), 10 instances, SAT2005
- Quasigroup existence (qg), 10 instances, SATLIB
- 10 Real-world domains
  - All interval series (ais), 6 instances, SATLIB
  - BMC-IBM (bmc), 3 instances, SATLIB
  - BW planning (bw), 4 instances, SATLIB
  - Job-shop scheduling e*ddr* (edd), 6 instances, SATLIB
  - Ferry planning (fer), 5 instances, SAT2005
  - Logistics planning (log), 4 instances, SATLIB
  - Parity learning par16* (par), 5 instances, SATLIB
  - "single stuck-at" (ssa), 4 instances, SATLIB
  - Cryptographic problem (vmpc), 5 instances, SAT2005
  - Models generated from Alloy (vpn), 2 instances, SAT2005
- Problem instance size
  - The smallest (ais6) contains 61 variables and 581 clauses
  - The largest (vpn-1962) contains 267,766 variables and 1,002,957 clauses

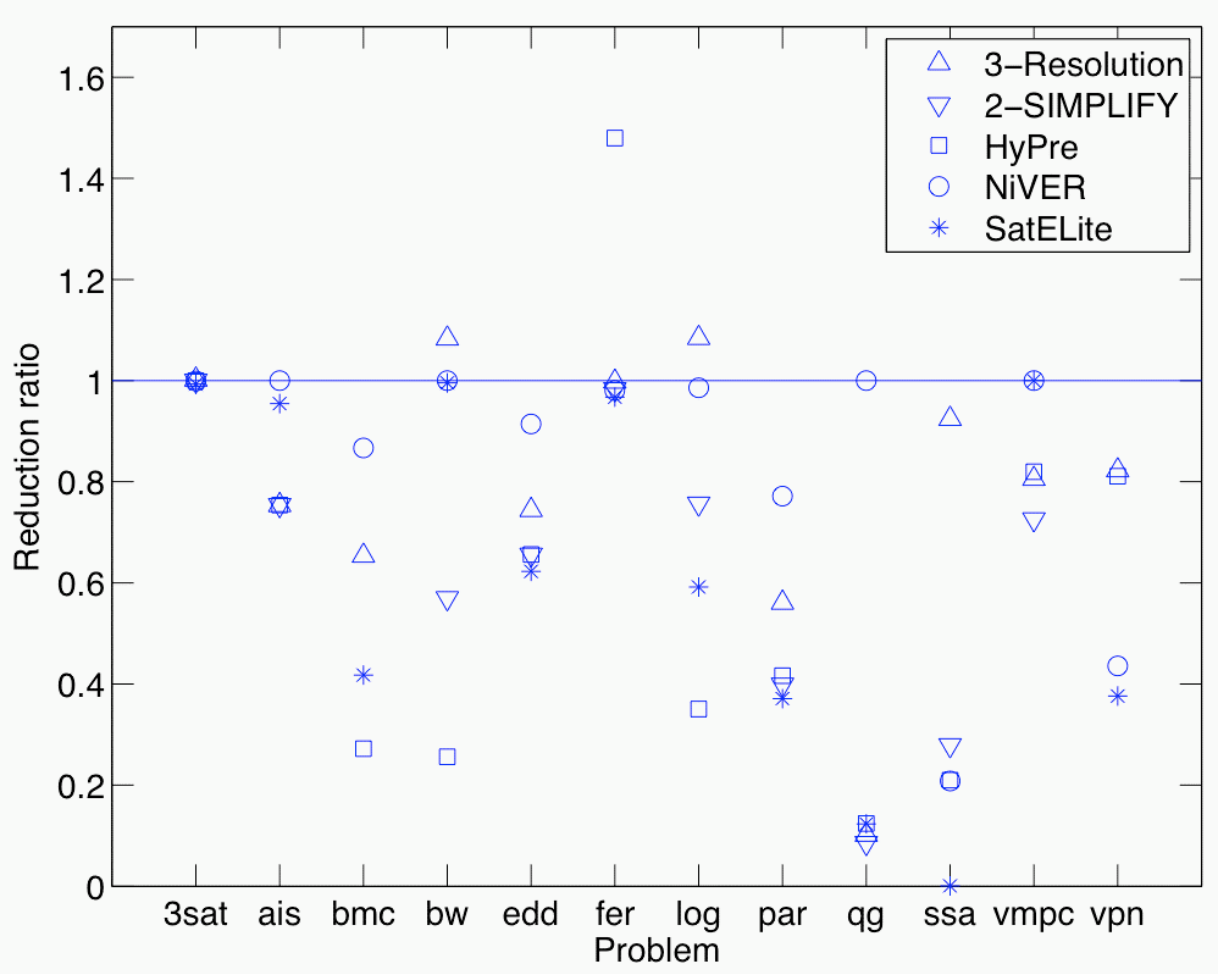
The Impact of Preprocessor

Variables Reduction



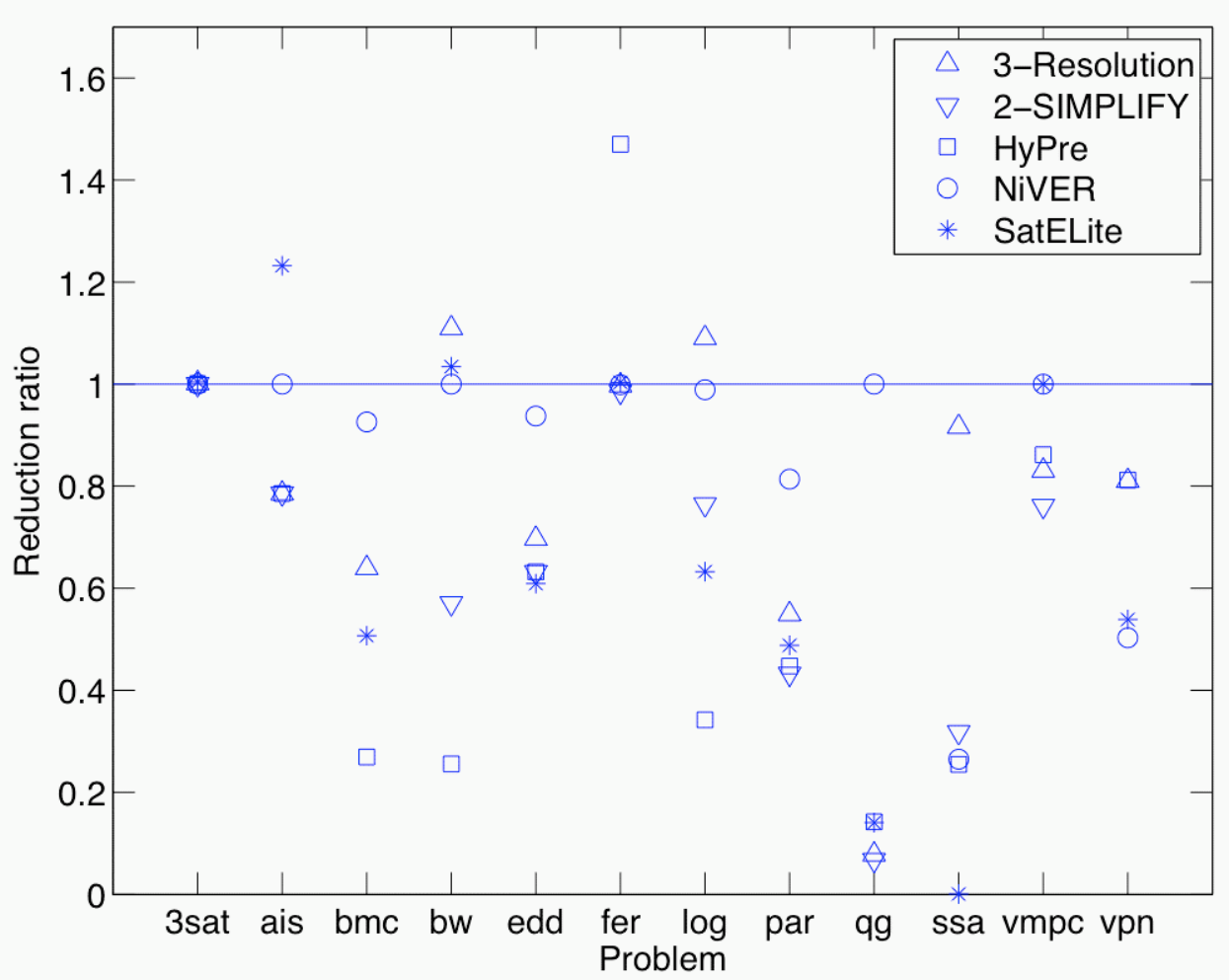
The Impact of Preprocessor

Clauses Reduction



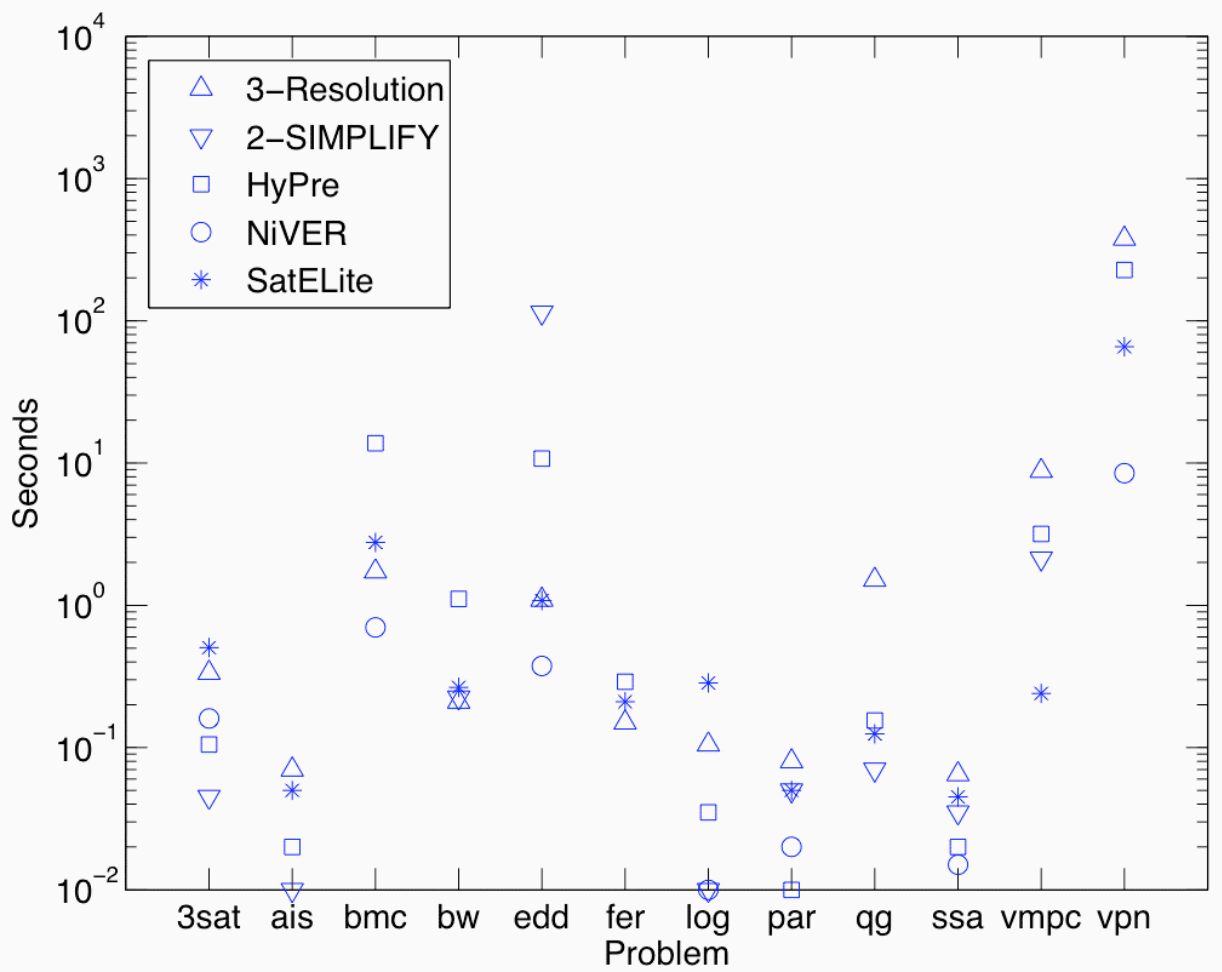
The Impact of Preprocessor

Literals Reduction



The Impact of Preprocessor

Preprocessing Time



SLS Solvers

- Random-Walk:
  - AdaptNovelty+ [Hoos, AAAI-2002]: enhancing Novelty+ with adaptive noise mechanism.
  - g2wsat [Li and Huang, SAT-2005]: deterministically picks the best promising decreasing variable to flip.

- Clause Weighting:
  - $PAWS_{10}$: PAWS [Thornton et al., AAAI-2004] with smooth parameter fixed to 10
  - RSAPS: reactive version of SAPS [Hutter et al., CP-2002]

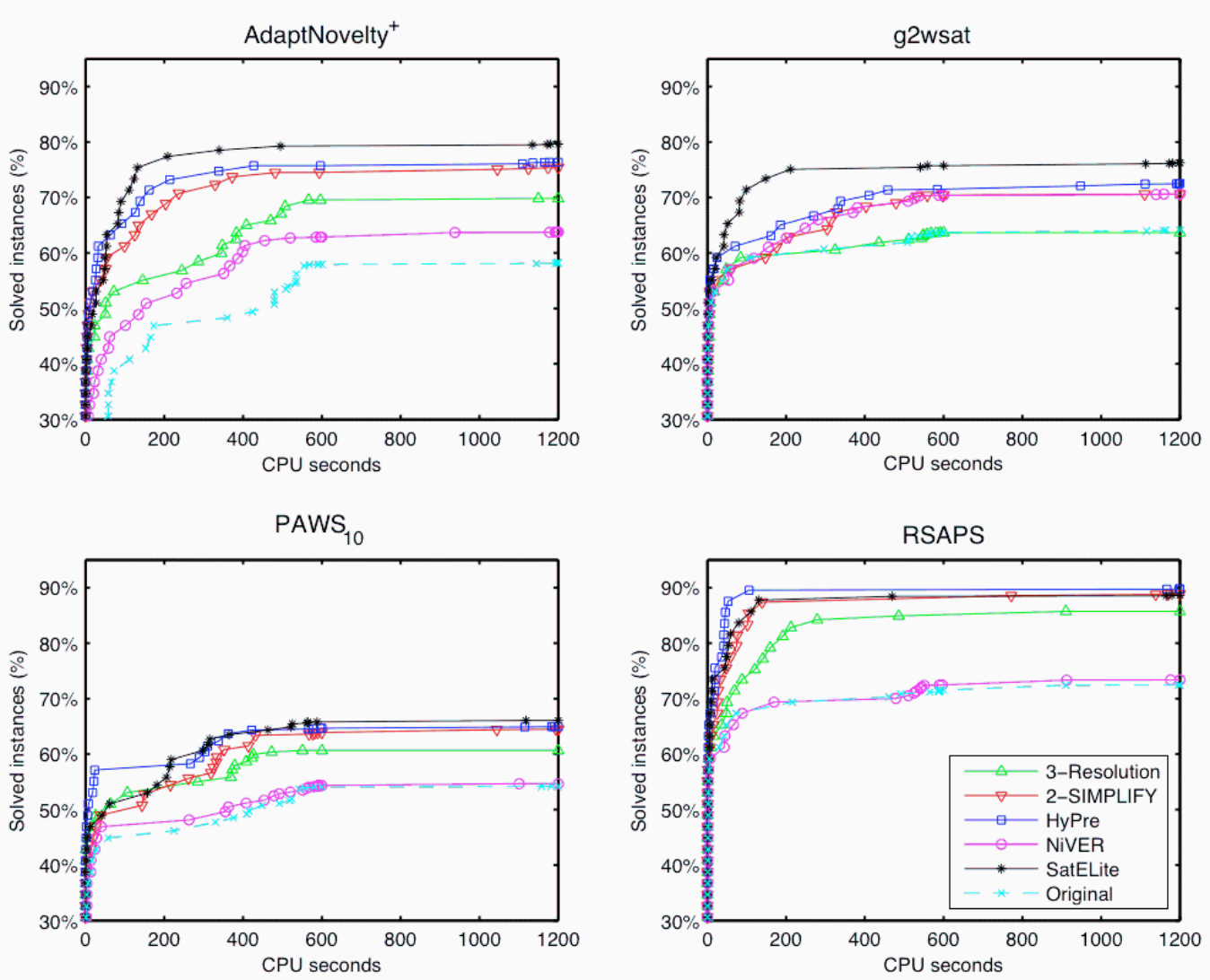
Empirical Study

- 12 classes of problems: random, quasigroup, real-world
  - 64 problem instances
- 5 resolution-based preprocessors
- 4 SLS solvers: random walk vs. clause weighting
- The total of 153,600 runs
  - 100 runs for each instance
  - 128,000 runs on preprocessed instances
  - 25,600 runs on original instances
- Time limit for each run is 1200 seconds for random, ferry, and cryptographic problems and 600 seconds for the other ones.
- On Linux Pentium IV computer with 3.00GHz CPU and 1GB RAM

Empirical Results

Instances	Prep.	#Vars/#Cls/#Lits	Ptime	g2wsat		AdaptNovelty$^+$		PAWS$_{10}$		RSAPS	
				%	Stime	%	Stime	%	Stime	%	Stime
ais16	origin	481/10621/25261	n/a	1	596.01	2	591.38	**75**	**329.96**	100	48.11
	2-SIM	481/10621/25261	0.06	10	558.64	3	593.65	66	352.13	**100**	**25.72**
	HyPre	481/10621/25261	0.04	5	583.78	1	596.47	66	362.62	100	43.77
	SatELite	465/10365/34140	0.17	**96**	**147.93**	**100**	**50.42**	2	588.07	99	79.73
bw_large_d	origin	6325/131973/294118	n/a	0	600.00	99	165.36	24	518.42	45	460.53
	3-Res	6325/141421/322458	0.85	0	600.00	88	244.74	15	551.18	34	485.78
	2-SIM	4644/82453/183302	1.57	0	600.00	37	481.55	98	149.17	99	139.24
	HyPre	3572/86285/188327	15.08	**51**	**410.81**	75	338.64	**100**	**21.36**	**100**	**41.97**
	SatELite	6307/131653/303116	1.22	0	600.00	**99**	**123.38**	25	525.46	35	468.96
ferry8-ks99a-4004	origin	1259/15259/31167	n/a	58	508.44	2	1197	86	543.26	100	0.08
	3-Res	1241/15206/31071	0.11	64	435.55	12	1150	98	285.26	100	0.04
	2-SIM	1233/14562/29783	0.00	56	531.99	26	1046	94	431.19	100	0.07
	HyPre	1209/20906/42471	0.13	30	947.43	8	1136	**100**	**1.34**	100	0.04
	NiVER	976/14952/30817	0.00	58	510.05	40	937.56	100	28.14	**100**	**0.03**
	SatELite	813/14720/34687	0.19	**84**	**210.65**	**100**	**209.11**	100	4.14	100	0.03
par16-1	origin	1015/3310/8788	n/a	12	534.30	35	479.82	0	600.00	5	588.42
	3-Res	607/1815/4713	0.04	9	548.27	99	145.44	0	600.00	97	88.02
	2-SIM	317/1266/3652	0.03	74	312.23	100	54.26	10	567.43	100	27.92
	HyPre	317/1324/3790	0.01	63	331.39	100	32.47	8	566.36	99	18.37
	NiVER	632/2512/7058	0.02	81	200.98	97	153.91	3	587.52	19	530.89
	SatELite	201/1173/4121	0.05	**100**	**22.99**	**100**	**17.59**	**93**	**214.34**	**100**	**5.91**
qg7-13	origin	2197/97072/256426	n/a	21	518.52	0	600.00	0	600.00	3	587.76
	3-Res	1412/45362/115164	2.06	27	511.52	49	381.89	99	106.34	99	159.64
	2-SIM	1333/41647/106430	0.75	75	326.24	84	135.23	100	37.97	100	58.96
	HyPre	1207/41110/105189	0.95	**100**	**24.57**	**99**	**27.32**	**100**	**8.23**	**100**	**19.69**
	SatELite	1412/45967/116374	0.35	17	541.08	55	339.55	98	157.78	100	130.90
3sat-1648	origin	10000/42000/126000	n/a	68	744.99	16	1145	78	399.52	0	1200
	3-Res	10000/42081/126243	0.40	66	747.21	**44**	**1022**	80	477.08	0	1200
	NiVER	9982/41981/125993	0.43	**78**	**600.19**	30	1082	**90**	**287.00**	0	1200
	SatELite	9775/41747/126601	0.66	70	672.08	22	1133	74	463.08	0	1200

RTDs on Structured Problems



Multiple Preprocessing and Preprocessor Ordering

Instances	Preprocessor	#Vars/#Cls/#Lits	Ptime	Succ. rate	CPU Time		Flips	
					median	mean	median	mean
ferry7-ks99i-	origin	1946/22336/45706	n/a	100	192.92	215.27	55,877,724	63,887,162
4001	SatELite	1286/21601/50644	0.27	100	4.39	5.66	897,165	1,149,616
	HyPre	1881/32855/66732	0.19	100	2.34	3.26	494,122	684,276
	HyPre & Sat	1289/29078/76551	0.72	100	2.17	3.05	359,981	499,964
	Sat & HyPre	1272/61574/130202	0.59	100	0.83	1.17	83,224	114,180
ferry8-ks99i-	origin	2547/32525/66425	n/a	42	1,200.00	910.38	302,651,507	229,727,514
4005	SatELite	1696/31589/74007	0.41	100	44.96	58.65	7,563,160	9,812,123
	HyPre	2473/48120/97601	0.29	100	9.50	19.61	1,629,417	3,401,913
	HyPre & Sat	1700/43296/116045	1.05	100	5.19	10.86	1,077,364	2,264,998
	Sat & HyPre	1680/92321/194966	0.90	100	2.23	3.62	252,778	407,258
par16-4	origin	1015/3324/8844	n/a	4	600.00	587.27	273,700,514	256,388,273
	HyPre	324/1352/3874	0.01	100	10.14	13.42	5,230,084	6,833,312
	SatELite	210/1201/4189	0.05	100	5.25	7.33	2,230,524	3,153,928
	Sat & HyPre	210/1210/4207	0.05	100	4.73	6.29	1,987,638	2,655,296
	HyPre & Sat	198/1232/4352	0.04	100	1.86	2.80	1,333,372	1,995,865

Table 3. RSAPS performance on ferry planning and par16-4 instances.

Boosting DPLL using Resolution-based Preprocessing

Empirical Results on Parity and Planning

Instance	Prep.	#Var/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
par32-4	Orig	3176/10313/27645	n/a	>15000	n/a	>15000	n/a
	3Res	2385/7433/19762	0.08	10,425	10,036,154	>15000	n/a
	Sat	849/5160/18581	0.21	12,820	18,230,746	>15000	n/a
	Hyp+3Res	1331/6055/16999	0.36	9,001	17,712,997	>15000	n/a
	3Res+Hyp	1331/5567/16026	0.11	5,741	10,036,146	>15000	n/a
	Niv+3Res	1333/5810/16503	0.34	6,099	10,036,154	>15000	n/a
	3Res+Niv	1290/5297/15481	0.10	14,003	25,092,756	>15000	n/a
	3Res+Sat	850/5286/18958	0.35	3,552	7,744,986	>15000	n/a
	Sat+3Res	849/5333/19052	0.38	3,563	7,744,986	>15000	n/a
	Sat+2Sim	848/5154/18565	0.26	12,862	18,230,746	>15000	n/a
ferry10_ks99a	Orig	1977/29041/59135	n/a	>15000	n/a	0.03	710
	3Res	1955/28976/59017	0.13	>15000	n/a	0.03	827
	Hyp	1915/40743/82551	0.29	>15000	n/a	0.04	563
	Niv	1544/28578/58619	0.02	>15000	n/a	0.01	0
	Sat	1299/28246/66432	0.44	>15000	n/a	0.03	909
	2Sim	1945/27992/57049	0.05	>15000	n/a	0.05	1,565
	Sat+2Sim	1299/69894/149728	0.69	0.28	1	0.04	419
	3Res+2Sim+Niv	1793/21099/43369	0.43	0.08	0	0.06	1,278
	Niv+Hyp+2Sim+3Res	1532/24524/50463	0.54	5.19	3,949	0.02	454

Empirical Results on BMC

Instance	Prep.	#Var/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
BMC-IBM-12	Orig	39598/194778/515536	n/a	>15000	n/a	8.41	11,887
	Hyp	12205/87082/228241	92	>15000	n/a	0.74	1,513
	Niv	27813/168440/476976	0.69	>15000	n/a	4.46	8,702
	3Res	32606/160555/419341	2.77	>15000	n/a	6.77	10,243
	Sat	15176/109121/364968	4.50	>15000	n/a	2.37	6,219
	Niv+Hyp+3Res	12001/100114/253071	86	106	6	0.76	1,937
BMC-alpha-25449	Orig	663443/3065529/7845396	n/a	>15000	n/a	6.64	502
	Sat	12408/76025/247622	129	6.94	7	0.06	1
	Sat+Hyp	9091/61789/203593	566	7.82	2	0.10	109
	Sat+Niv	12356/75709/246367	130	4.48	2	0.06	1
	Sat+3Res	12404/77805/249192	130	8.84	1	0.06	1
	Sat+2Sim	10457/71128/229499	131	6.37	10	0.10	133
BMC-alpha-4408	Orig	1080015/3054591/7395935	n/a	>15000	n/a	5,409	587,755
	Sat	23657/112343/364874	47	>15000	n/a	1,266	820,043
	Sat+Hyp	13235/88976/263053	56	>15000	n/a	8,753	4,916,981
	Sat+Niv	22983/108603/351369	49	>15000	n/a	2,137	1,294,590
	Sat+3Res	23657/117795/380389	48	>15000	n/a	946	618,853
	Sat+2Sim	17470/129245/375444	52	>15000	n/a	804	561,529
	Sat+2Sim+3Res	16837/98726/305057	53	>15000	n/a	571	510,705

Empirical Results on FPGA Routing

Instance	Dew_Satz			MiniSat		
	#Solved	Stime	#BackT	#Solved	Stime	#Conflict
bart (21 SAT)	21	18	1,536,966	8	7,203	119,782,466
homer (15 UNSAT)	15	2,662	109,771,200	14	22,183	143,719,166

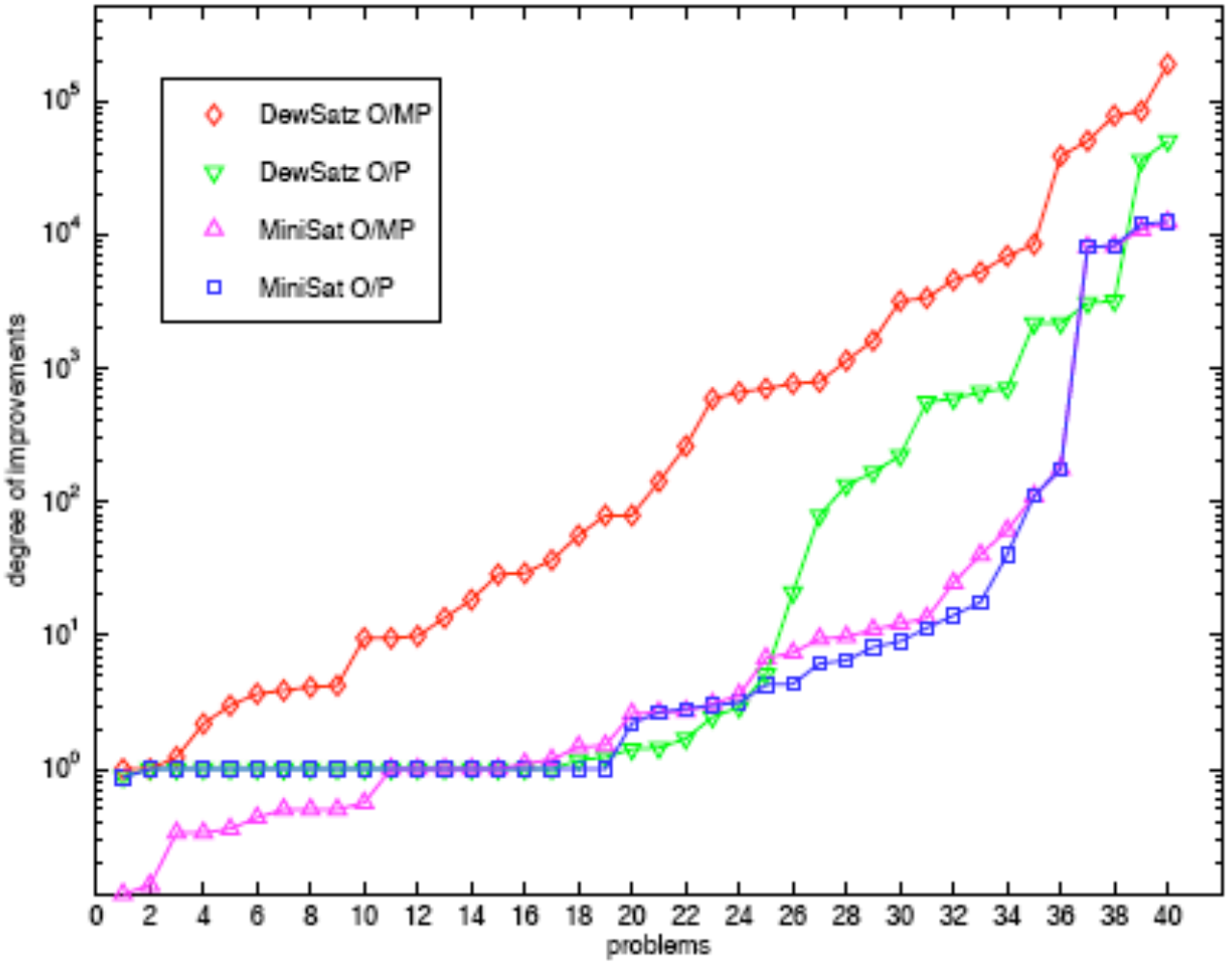
Instance	Prep.	#Var/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
bart-28	Orig	428/2907/7929	n/a	0	0	>15,000	n/a
	Sat	413/2892/11469	0.06	0.02	0	>15,000	n/a
	Sha	1825/8407/27003	0.37	0.06	9	198	775,639
	Sha+3Res	1764/7702/24400	0.46	0.04	1	2,458	7,676,459
	Sha+Hyp	1764/8349/26138	0.41	0.05	20	>15,000	n/a
	Sha+Niv	1781/8358/26759	0.38	0.05	6	5.46	53,683
	Sha+Sat	1728/8254/30422	0.53	0.10	0	115	684,272
	Sha+2Sim	1750/7892/24682	0.39	0.05	17	19.12	150,838
homer-20	Orig	440/4220/8800	n/a	941	19,958,400	>15,000	n/a
	Sat	400/4180/15200	0.08	1,443	6,982,425	11,448	57,302,582
	Sha	1999/10340/29988	0.28	369	350,610	1.83	22,950
	Sha+3Res	1907/8793/25027	0.37	362	405,059	1.41	18,273
	Sha+Hyp	1905/10527/29129	0.34	1,306	1,451,567	1.10	13,927
	Sha+Niv	1941/10276/29671	0.29	379	349,842	0.91	13,543
	Sha+Sat	1723/9420/30986	0.54	822	300,605	1.00	13,831
	Sha+2Sim	1879/9419/26188	0.31	114	120,297	0.40	6,612

Multiple Preprocessing and Preprocessor Ordering

Using Dew_Satz

Instance	Prep.	#Var/#Cls/#Lits	Ptime	Stime	#BackT
BMC-IBM-12	Hyp+3Res+Niv	10805/83643/204679	96.11	>15,000	n/a
	Niv+Hyp+3Res	12001/100114/253071	85.81	106	6
	3Res+Hyp+Niv	10038/82632/221890	89.56	>15,000	n/a
	3Res+Niv+Hyp	11107/99673/269405	58.38	>15,000	n/a
ferry10_ks99a	2Sim+Niv+Hyp+3Res	1518/32206/65806	0.43	>15,000	n/a
	Niv+3Res+2Sim+Hyp	1532/25229/51873	0.49	11,345	17,778,483
	3Res+2Sim+Niv+Hyp	1793/20597/42365	0.56	907	1,172,964
	Niv+Hyp+2Sim+3Res	1532/24524/50463	0.54	5	3,949
ferry10_ks99a	2Sim+Niv	1518/27554/56565	0.08	>15,000	n/a
	2Sim+Niv+2Sim	1518/18988/39433	0.27	3,197	6,066,241
	2Sim+Niv+ 2Sim+Niv	1486/18956/39429	0.29	129	290,871
	2Sim+Niv+ 2Sim+Niv+2Sim	1486/23258/48033	0.48	7,355	8,216,100

Preprocessing + DPLL



Conclusion: on SAT Algorithms

- Complete method
  - LA-based DPLL
  - CDCL-based DPLL
- Incomplete method (SLS)
  - Random Walk
  - DLS (clause weighting SLS)
  - Resolution+SLS
- Resolution + SLS

Conclusion: on Complete SAT Algorithms

3 classes of SAT solvers in terms of their capability for solving problems:

- High performance on random problems
  - Kcnfs
  - Based on look-ahead
- High performance on most of structured problems
  - zChaff, Jerusat, Berkmin, Siege, MiniSat, Tinisat, etc…
  - Based on CDCL
- Good performance on random & high performance on some classes of structured problems
  - Satz, Dew_Satz, march_dl, etc…
  - Based on look-ahead

Conclusion: SAT Algorithms Performance Comparison

Algo / Prob	Complete SAT solvers		Incomplete SAT solvers	
	DPLL	CDCL	Random Walk	Clause Weighting
random	Can reach 700 vars	10 - 40 times slower than DPLL	10000 vars with 68% sr (g2wsat)	10000 vars with 0% sr (RSAPS)
structured real-world	better than CDCL on 30%	better than DPLL on 70%	better than CW on 30%	better than RW on 70%

R+DDFW^{+} is the current best SLS solver for both random and realistic problems

Questions……..