Keynote on

Incremental Solvers for Formal Reasoning

Armin Biere Univ. Freiburg

25th Forum on Specification & Design Languages

FDL'22

Linz, Austria

September 14, 2022

Incremental Inprocessing in SAT Solving

best student Paper

SAT'19

Katalin Fazekas[1] (✉), Armin Biere[1], Christoph Scholl[2]

[1] Johannes Kepler University, Linz, Austria
katalin.fazekas@jku.at, armin.biere@jku.at
[2] Albert–Ludwigs–University, Freiburg, Germany
scholl@informatik.uni-freiburg.de

Abstract. Incremental SAT is about solving a sequence of related SAT problems efficiently. It makes use of already learned information to avoid repeating redundant work. Also preprocessing and inprocessing are considered to be crucial. Our calculus uses the most general redundancy property and extends existing inprocessing rules
solving. It allows to automatically reverse earlier
which are inconsistent with literals in new increme
Our approach to incremental SAT solving not only
inprocessing but also substantially improves solvin

1 Introduction

Solving a sequence of related SAT problems incremen
the efficiency of SAT based model checking [5,6,7,8]
domains [9,10,11,12]. Utilizing the effort already spe
keeping learned information (such as variable scores an
nificantly speed-up solving similar problems. Equally i
plification techniques such as variable elimination, su
resolution, and equivalence reasoning [13,14,15,16].

These simplifications are not only applied before

Mining Definitions in Kissat with Kittens

POS'21

Mathias Fleury and Armin Biere

Johannes Kepler University Linz, Austria
mathias.fleury@jku.at armin.biere@jku.at

Abstract

Bounded variable elimination is one of the most important preprocessing techniques in SAT solving. It benefits from discovering functional dependencies in the form of definitions encoded in the CNF. While the original approach relied on syntactic pattern matching our new approach uses cores produced by an embedded SAT solver. In contrast to a similar semantic approach in Lingeling based on BDD algorithms, our new approach is able to generate DRAT proofs. We further discuss design choices for our embedded SAT solver Kitten. Experiments with Kissat show the effectiveness of this approach.

Single Clause Assumption without Activation Literals to Speed-up IC3

Nils Froleyks
nils.froleyks@jku.at
Johannes Kepler University, Linz, Austria

Armin Biere
biere@cs.uni-freiburg.de
Albert–Ludwigs–University, Freiburg, Germany

FMCAD'21

***Abstract*—We extend the well-established assumption-based interface of incremental SAT solvers to clauses, allowing the addition of a temporary clause that has the same lifespan as literal assumptions. Our approach is efficient and easy to implement in modern CDCL-based solvers. Compared to previous approaches, it does not come with any memory overhead and does not slow down the solver due to disabled activation literals, thus eliminating the need for algorithms like IC3 to restart the SAT**

values greater than five are unreachable. A typical query asks "is state six reachable from any other state?", expressed as $SAT?[T \wedge (\neg b_2 \vee \neg b_1 \vee b_0) \wedge b_2' \wedge b_1' \wedge \neg b_0']$, where T encodes the transition system for one step from $b_2b_1b_0$ to $b_2'b_1'b_0'$. It is unsatisfiable, telling us that state six is in fact unreachable. We can try to generalize this result to a set of states by considering a *cube* – an assignment to a subset of

Incremental SAT

$F_0 = C_0$ — Satisfiable?

$F_1 = C_0 \wedge C_1$ — Satisfiable?

⋮

$F_K = C_0 \wedge \dots \wedge C_K$ — Satisfiable?

shared

K queries

1 SAT SOLVER!

SAT, SAT, ..., SAT, UNSAT

typical example Optimization but also CEGAR / Localization

C_0 hard constraint if SAT ⇝ UB
upper bound

$C_1, \ldots, C_k$ restrictions,

i.e., C_1 sol. with UB − 1

C_k sol. with UB − k

if C_k SAT but C_{k+1} UNSAT ⇒ UB − k Optimum

UNSAT, UNSAT, ..., UNSAT, SAT

typical example BMC

bounded Model Checking

but also core based MaxSAT

bad state ~~constraint~~ assumptions

B_0 C_0 initial state constraints

B_1 C_1 1st step constraints

⋮

B_k C_k k^{th} step constraints

$$F_k \equiv \bigwedge C_i \wedge B_k$$

assumptions

Minimal Unsatisfiable Set (MUS)

not unique

$$M \subseteq \{ C_0, C_1, \ldots, C_k \}$$

with $\bigwedge M$ UNSAT

and $\bigwedge M'$ SAT for all $M' \subsetneq M$

Simple Destructive MUS Algorithm

$W = M$

$M' = M$

while $\exists C \in W$

  $T = M' \setminus \{C\}$

  $W = W \setminus \{C\}$

  if $\bigwedge T$ unsatisfiable

    $M' = T$

SAT, UNSAT, SAT, UNSAT, SAT, SAT, UNSAT, ... ~~SAT~~ UNSAT

could be improved by binary search

QuickCheck, DeltaDebugging

MiniSAT Assumption Interface

* for each temporary clause

  $L_1 \vee \ldots \vee L_m$

  add Activation literal A

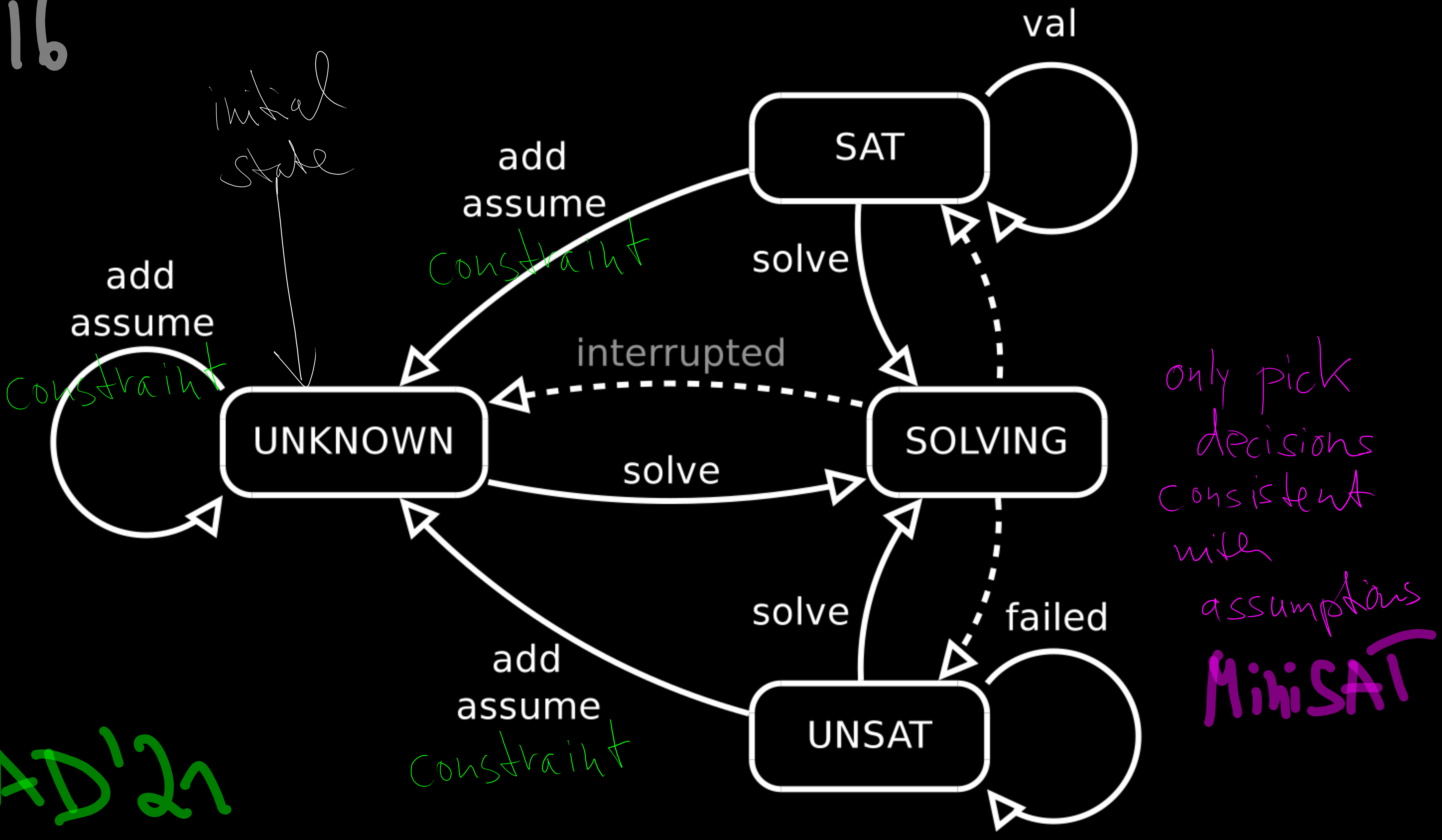
  $L_1 \vee \ldots \vee L_m \vee \overline{A}$

* solve under assumption A
* if unsatisfiable determine assumption core
* if clause not needed anymore add unit $\overline{A}$

State of the art

IPASIR Model

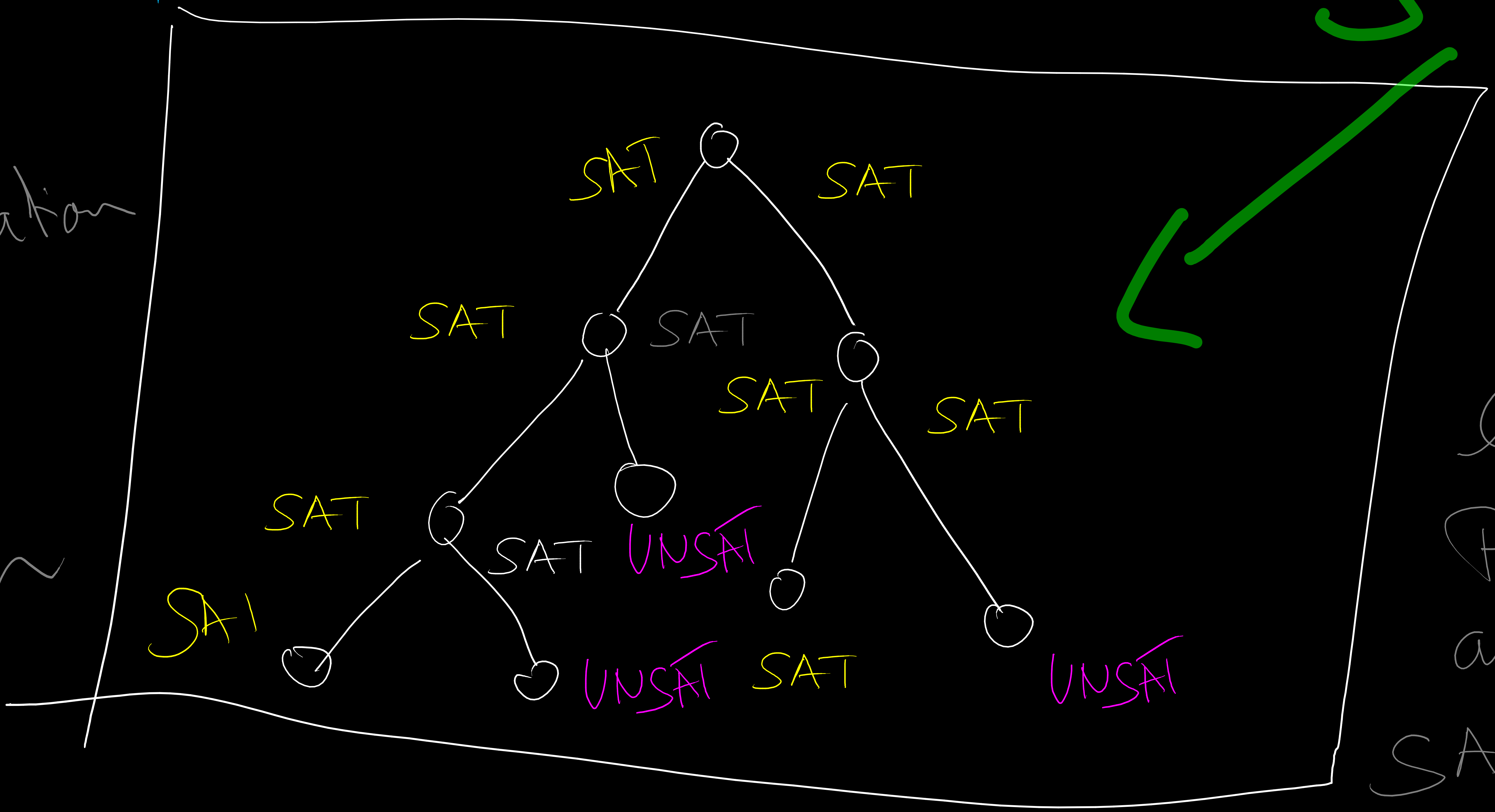
AIJ'16



FMCAD'21

Symbolic Execution

Explore Computation Tree of Program



Stack Like

Branches lead to Path conditions and two SAT Checks

so similar to SAT, SAT, ..., UNSAT but with backtracking

SMT Style Push/Pop Interface

* "push" new context
* add clauses/constraints as usual
* solve context with new clauses
* "pop" context undoes clause additions

hard to implement efficiently

Relative Inductive Clause Generalization in IC3

$$C \wedge T(s, s') \models C'$$

C' next state copy of C

minimize $D \subseteq C$ with

$$D \wedge T(s, s') \models D'$$

remove l_2?

$$(l_1 \vee l_2 \vee \ldots \vee l_n) \wedge T(S,S') \wedge \bar{l}_1 \wedge \bar{l}_2 \wedge \ldots \wedge \bar{l}_n$$

assumed constraint? — assumptions

instead add

$$\bar{A} \vee l_1 \vee l_3 \vee \ldots \vee l_n$$

assume A — fresh variable $\Rightarrow$ **Recycle**

Cabodi, G. and Camurati, P. E. and Mishchenko, A. and Palena, M. and Pasini, P., "SAT Solver Management Strategies in IC3: An Experimental Approach," *Formal Methods in System Design*, vol. 50, pp. 39–74, mar 2017.

decide ( )

```
if level < |assumptions|
    ℓ = assumptions[level]
    if val(ℓ) = false
        analyzeFinal()
    else if val(ℓ) = true
        level++ // pseudo decision level
    else trail[level++] = ℓ
else if level = |assumptions|
    unassignedLit = 0
    for ℓ in constraint
        if val(ℓ) = true
            level++ // pseudo decision level
        else if val(ℓ) = unassigned
            unassigendLit = ℓ
    if unassigendLit = 0
        analyzeFinalConstraint() // cannot be satisfied
    else trail[level++] = unassigendLit
else
    ℓ = literalSelectionHeuristic()
    trail[level++] = ℓ
```

Original

	PAR-2					Res.	Calls		TpC	
	Di	Og	Ca	Co	De	Ca	Ca	Co	Ca	Co
Mean	80	46	16	8.93	**8.21**	61	19	**15**	0.61	**0.51**
beemTele6Int	136	7200	**53**	181	101	520	**157**	574	**0.24**	0.27
toyLock4	7200	483	1731	**357**	359	7459	2251	**1098**	0.42	**0.25**
visArraysField5	7200	1.6	**0.58**	51	34	1	**1**	113	0.53	**0.41**
nan	208	421	163	158	**140**	1381	**420**	423	**0.29**	0.32
beemColl6Int	241	258	322	133	**108**	398	123	**91**	2.31	**1.24**
cal110	213	168	130	**110**	122	191	59	**42**	**1.96**	2.39
cal109	179	197	102	117	**86**	110	**34**	44	2.71	**2.44**
cal93	186	136	121	**118**	140	206	63	**58**	**1.69**	1.8
cal94	127	160	115	**95**	131	171	52	**41**	**1.94**	2.1
cal100	112	**42**	67	67	54	148	45	**44**	**1.23**	1.29
cal131	46	**44**	77	58	60	136	42	**35**	1.58	**1.41**
cal146	47	39	71	42	**38**	131	41	**23**	**1.51**	1.55
cal136	**34**	46	59	43	35	100	31	**23**	1.62	**1.59**
cal128	52	38	46	**37**	40	99	31	**25**	1.29	**1.27**
beemExit5Int	51	17	26	16	**15**	357	110	**86**	0.18	**0.15**
cal134	38	47	50	48	**36**	79	**25**	26	1.72	**1.57**
cal132	39	36	48	42	**32**	83	26	**24**	1.57	**1.54**
cal144	**30**	34	41	33	42	64	20	**17**	1.7	**1.64**
beemLampNat5Int	26	23	**23**	35	31	193	**61**	102	**0.28**	0.3
cal89	16	**14**	32	33	25	68	22	**18**	**1.23**	1.6
beemRether4Bstep	13	**4.29**	16	7.16	6.99	91	29	**13**	**0.42**	0.49
beemBrp2Int	16	5.1	3.6	0.76	**0.74**	86	29	**7**	0.08	**0.07**
beemFrogs2Bstep	**2.47**	2.53	12	5.59	4.74	31	10	**4**	**1.12**	1.27
beemAdding5Int	1.78	3.9	2.07	1.12	**1.09**	53	17	**11**	0.08	**0.07**
visArraysTwo	1.35	2.89	3.89	0.57	**0.55**	99	30	**5**	0.09	**0.07**
Heap	2.02	1.9	3.38	1.68	**1.63**	57	22	**13**	0.11	**0.09**

Disable restarts, **Orig**inal version of ABC, **Ca**DiCaL backend, **Co**nstraint interface used, **De**fer model reconstruction

recycle solver

FMCAD'21

Incremental Inprocessing in SAT Solving

[IJCAR'12]

SAT'19 best student paper

$$\frac{\varphi\,[\rho]\,\sigma}{\varphi\,[\rho \wedge C]\,\sigma}\ \boxed{\sharp} \qquad \frac{\varphi\,[\rho \wedge C]\,\sigma}{\varphi\,[\rho]\,\sigma} \qquad \frac{\varphi\,[\rho \wedge C]\,\sigma}{\varphi \wedge C\,[\rho]\,\sigma} \qquad \frac{\varphi\,[\rho]\,\sigma}{\varphi \wedge \Delta\,[\rho]\,\sigma}\ \boxed{\mathcal{I}}$$

LEARN$^-$ FORGET STRENGTHEN ADDCLAUSES

new context $\Delta = C_i$

$$\frac{\varphi \wedge C\,[\rho]\,\sigma}{\varphi\,[\rho]\,\sigma \cdot (\omega : C)}\ \boxed{\flat} \qquad \frac{\varphi \wedge C\,[\rho]\,\sigma}{\varphi\,[\rho]\,\sigma}\ \boxed{\emptyset} \qquad \frac{\varphi\,[\rho]\,\sigma \cdot (\omega : C) \cdot \sigma'}{\varphi \wedge C\,[\rho]\,\sigma \cdot \sigma'}\ \boxed{\partial}$$

WEAKEN$^+$ DROP RESTORE

clean up reconstruction stack for new "solve" call

where $\boxed{\sharp}$ is $\varphi \wedge \rho \models C$, $\boxed{\flat}$ is $\varphi \wedge C \equiv^{\omega}_{sat} \varphi$, $\boxed{\emptyset}$ is $\varphi \models C$,

$\boxed{\partial}$ is C is clean w.r.t. σ' and $\boxed{\mathcal{I}}$ is that each clause in Δ is clean w.r.t. σ

RestoreAddClauses (new clauses Δ, reconstruction stack σ)

$(\omega_1 : C_1) \cdots (\omega_n : C_n) := \sigma$

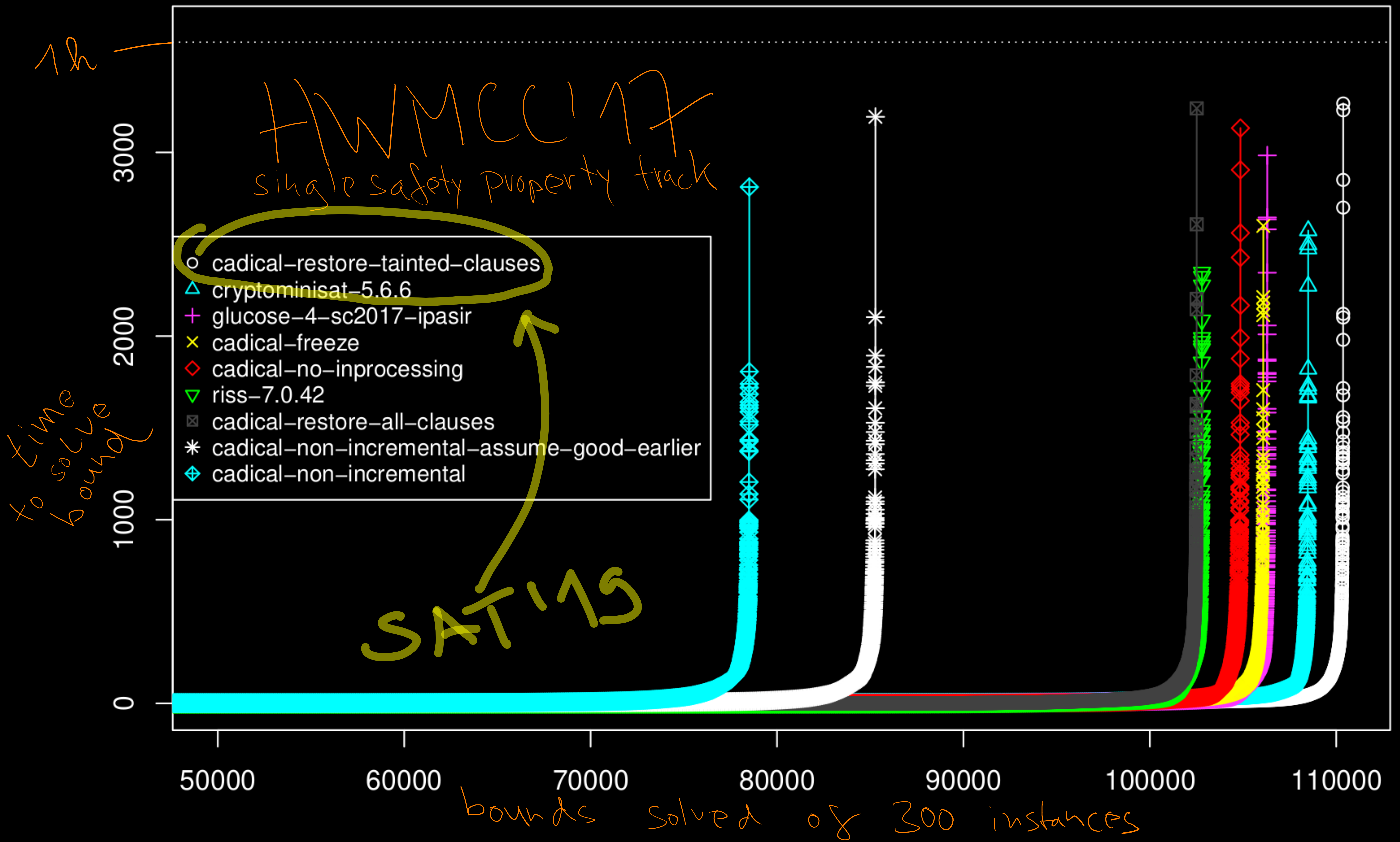
for i **from** 1 **to** n

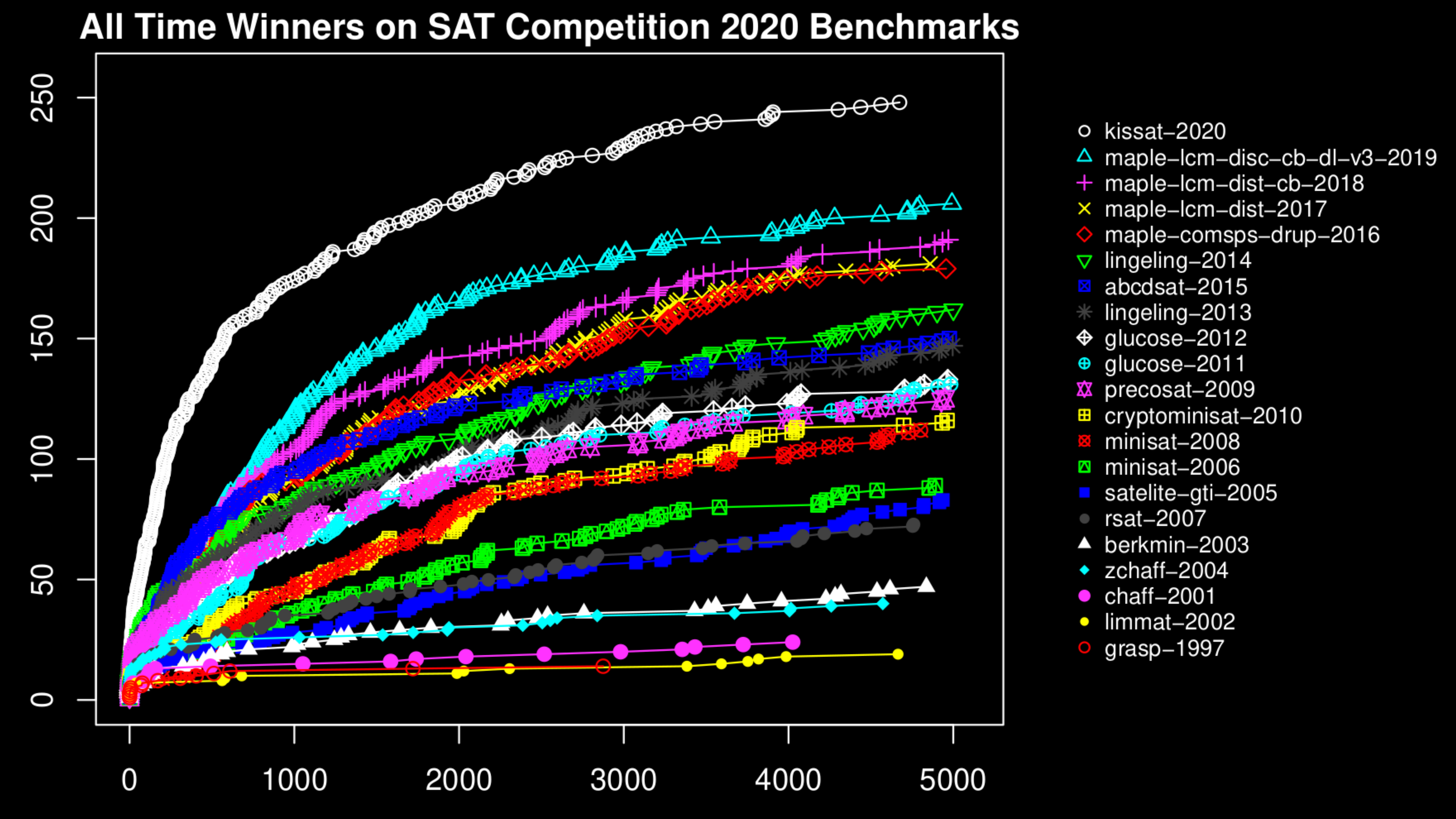
  if exists $\ell \in \omega_i$ where $\neg\ell$ occurs in Δ **then**

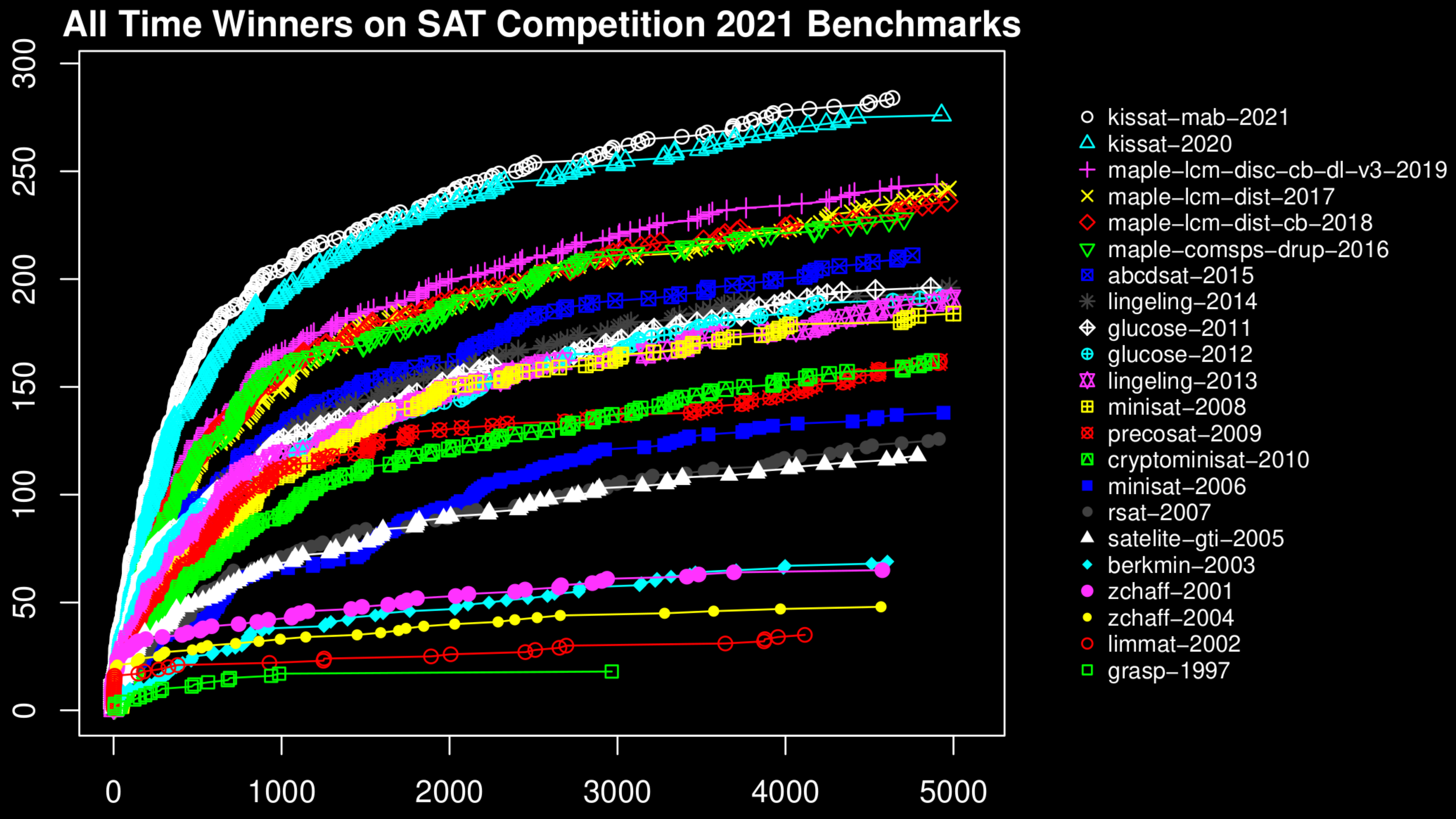
    $\Delta := \Delta \cup C_i$, $\sigma := \sigma \setminus (\omega_i : C_i)$

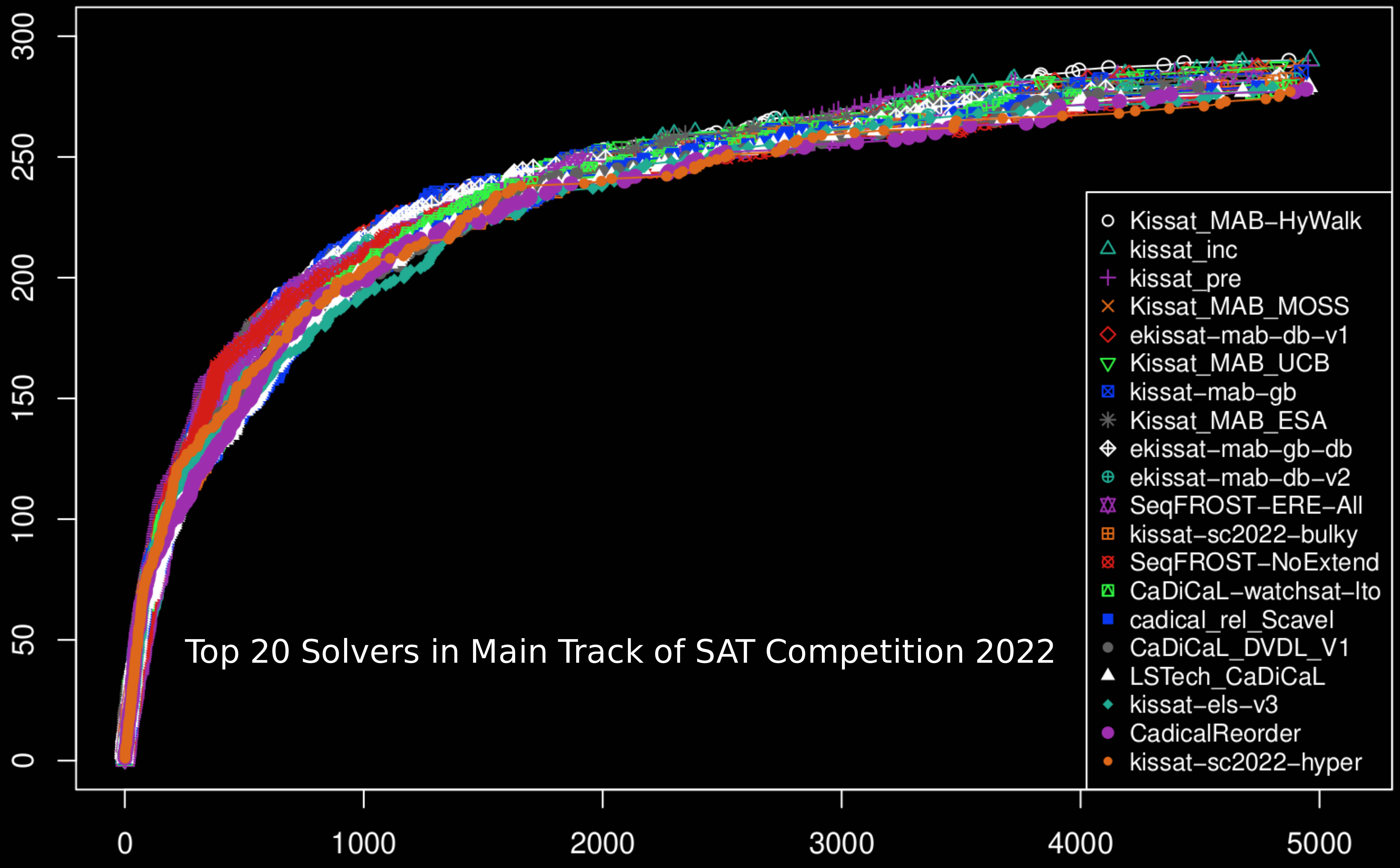
return $\langle \Delta, \sigma \rangle$

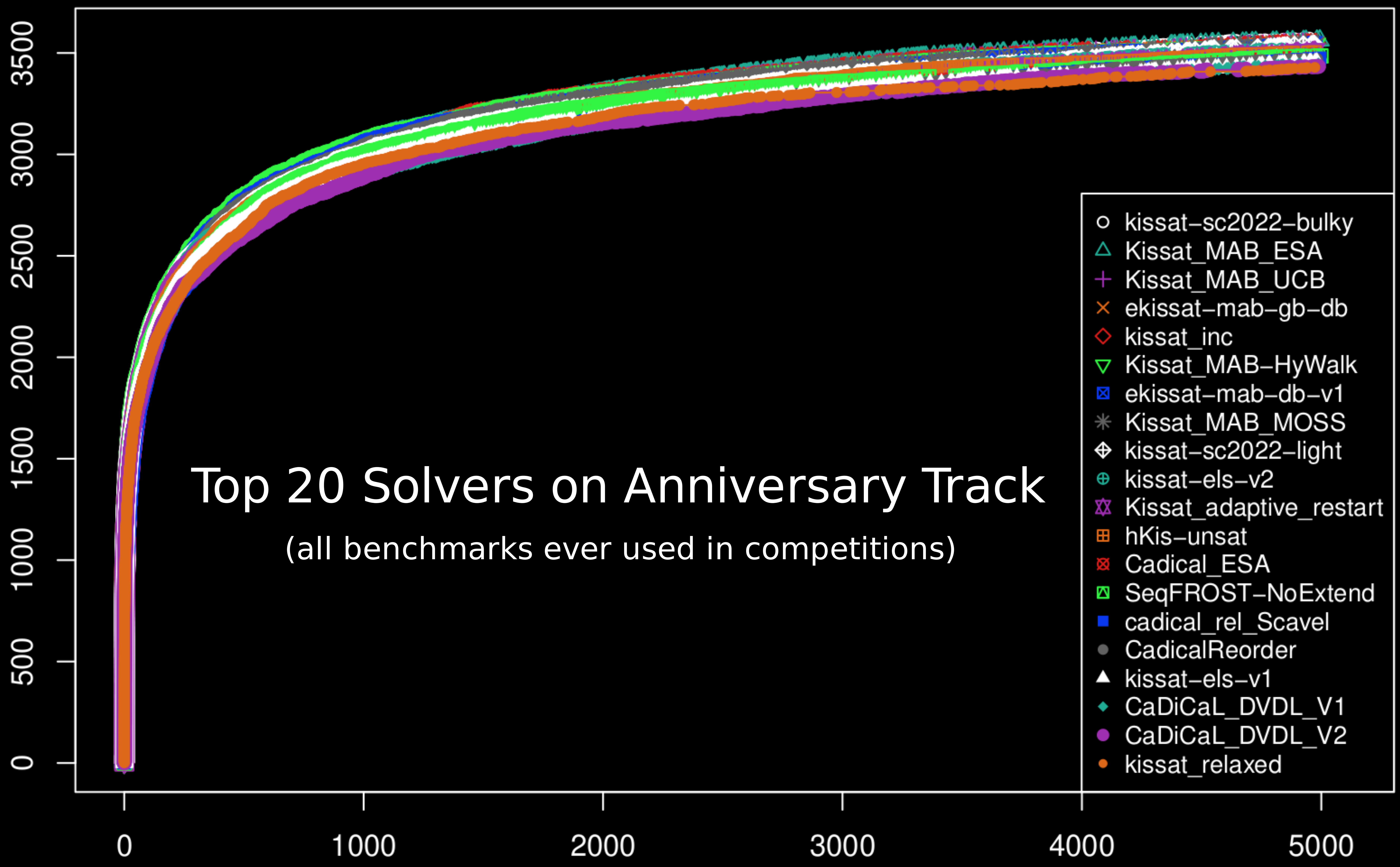
Algorithm RestoreAddClauses to identify and restore all tainted clauses

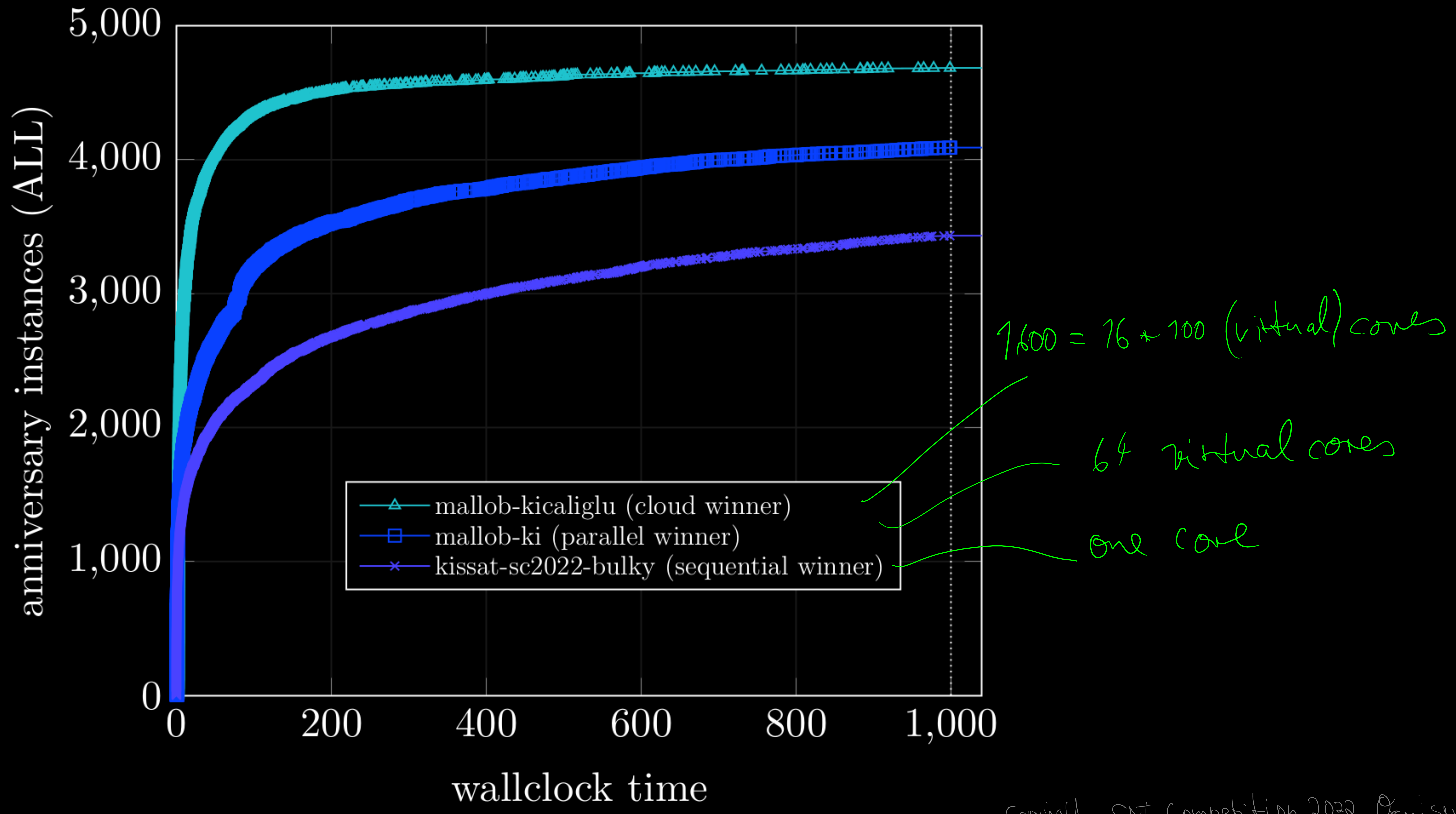

1h
HWMCC'17
single safety property track
cadical–restore–tainted–clauses
cryptominisat–5.6.6
glucose–4–sc2017–ipasir
cadical–freeze
cadical–no–inprocessing
riss–7.0.42
cadical–restore–all–clauses
cadical–non–incremental–assume–good–earlier
cadical–non–incremental
SAT'19
time to solve bound
0
1000
2000
3000
50000
60000
70000
80000
90000
100000
110000
bounds solved of 300 instances

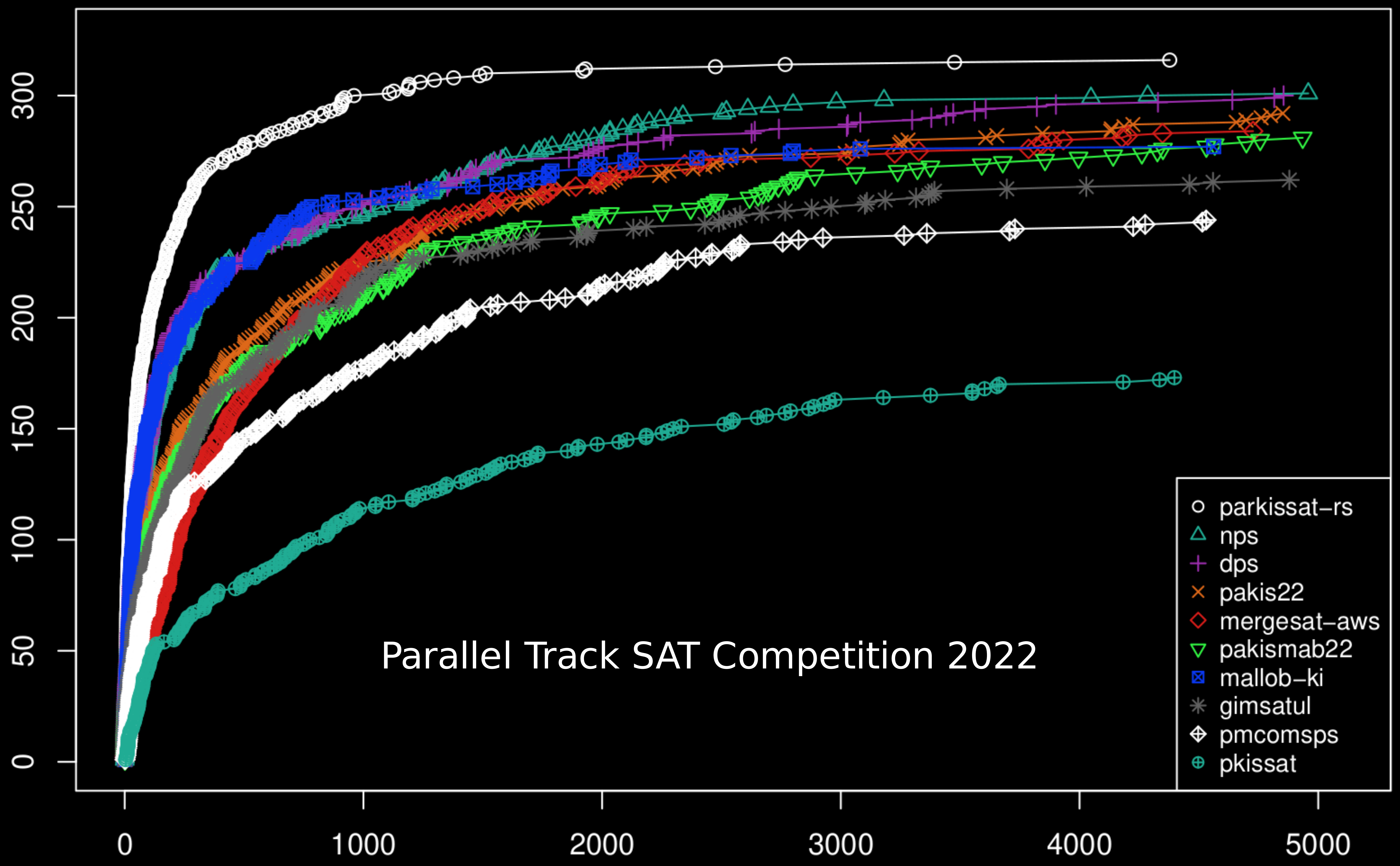

All Time Winners on SAT Competition 2020 Benchmarks
0
50
100
150
200
250
0
1000
2000
3000
4000
5000
kissat−2020
maple−lcm−disc−cb−dl−v3−2019
maple−lcm−dist−cb−2018
maple−lcm−dist−2017
maple−comsps−drup−2016
lingeling−2014
abcdsat−2015
lingeling−2013
glucose−2012
glucose−2011
precosat−2009
cryptominisat−2010
minisat−2008
minisat−2006
satelite−gti−2005
rsat−2007
berkmin−2003
zchaff−2004
chaff−2001
limmat−2002
grasp−1997


All Time Winners on SAT Competition 2021 Benchmarks
0
50
100
150
200
250
300
0
1000
2000
3000
4000
5000
kissat–mab–2021
kissat–2020
maple–lcm–disc–cb–dl–v3–2019
maple–lcm–dist–2017
maple–lcm–dist–cb–2018
maple–comsps–drup–2016
abcdsat–2015
lingeling–2014
glucose–2011
glucose–2012
lingeling–2013
minisat–2008
precosat–2009
cryptominisat–2010
minisat–2006
rsat–2007
satelite–gti–2005
berkmin–2003
zchaff–2001
zchaff–2004
limmat–2002
grasp–1997


300
250
200
150
100
50
0
0
1000
2000
3000
4000
5000
Top 20 Solvers in Main Track of SAT Competition 2022
Kissat_MAB−HyWalk
kissat_inc
kissat_pre
Kissat_MAB_MOSS
ekissat−mab−db−v1
Kissat_MAB_UCB
kissat−mab−gb
Kissat_MAB_ESA
ekissat−mab−gb−db
ekissat−mab−db−v2
SeqFROST−ERE−All
kissat−sc2022−bulky
SeqFROST−NoExtend
CaDiCaL−watchsat−lto
cadical_rel_Scavel
CaDiCaL_DVDL_V1
LSTech_CaDiCaL
kissat−els−v3
CadicalReorder
kissat−sc2022−hyper

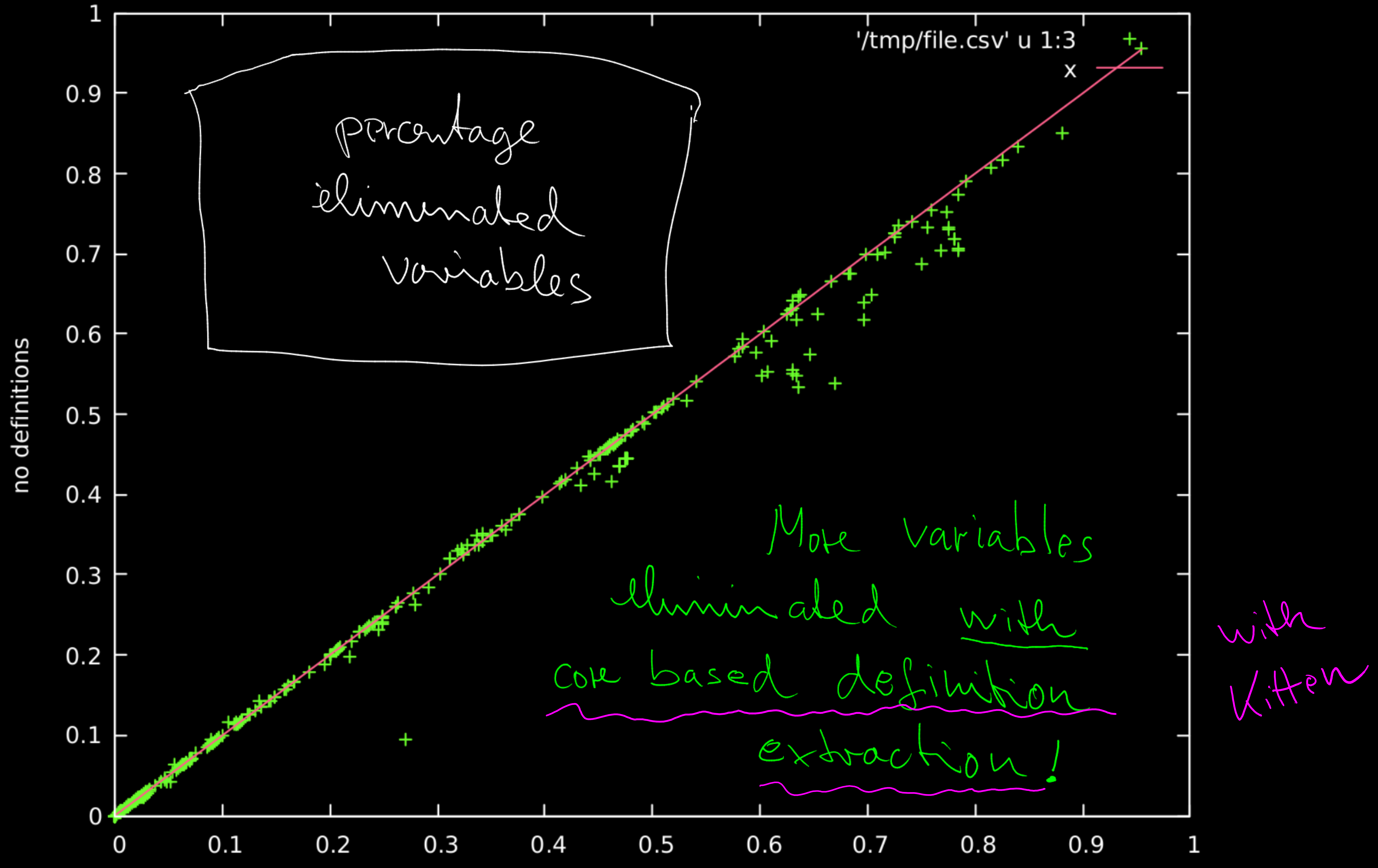

Top 20 Solvers on Anniversary Track
(all benchmarks ever used in competitions)
kissat–sc2022–bulky
Kissat_MAB_ESA
Kissat_MAB_UCB
ekissat–mab–gb–db
kissat_inc
Kissat_MAB–HyWalk
ekissat–mab–db–v1
Kissat_MAB_MOSS
kissat–sc2022–light
kissat–els–v2
Kissat_adaptive_restart
hKis–unsat
Cadical_ESA
SeqFROST–NoExtend
cadical_rel_Scavel
CadicalReorder
kissat–els–v1
CaDiCaL_DVDL_V1
CaDiCaL_DVDL_V2
kissat_relaxed
0
500
1000
1500
2000
2500
3000
3500
0
1000
2000
3000
4000
5000

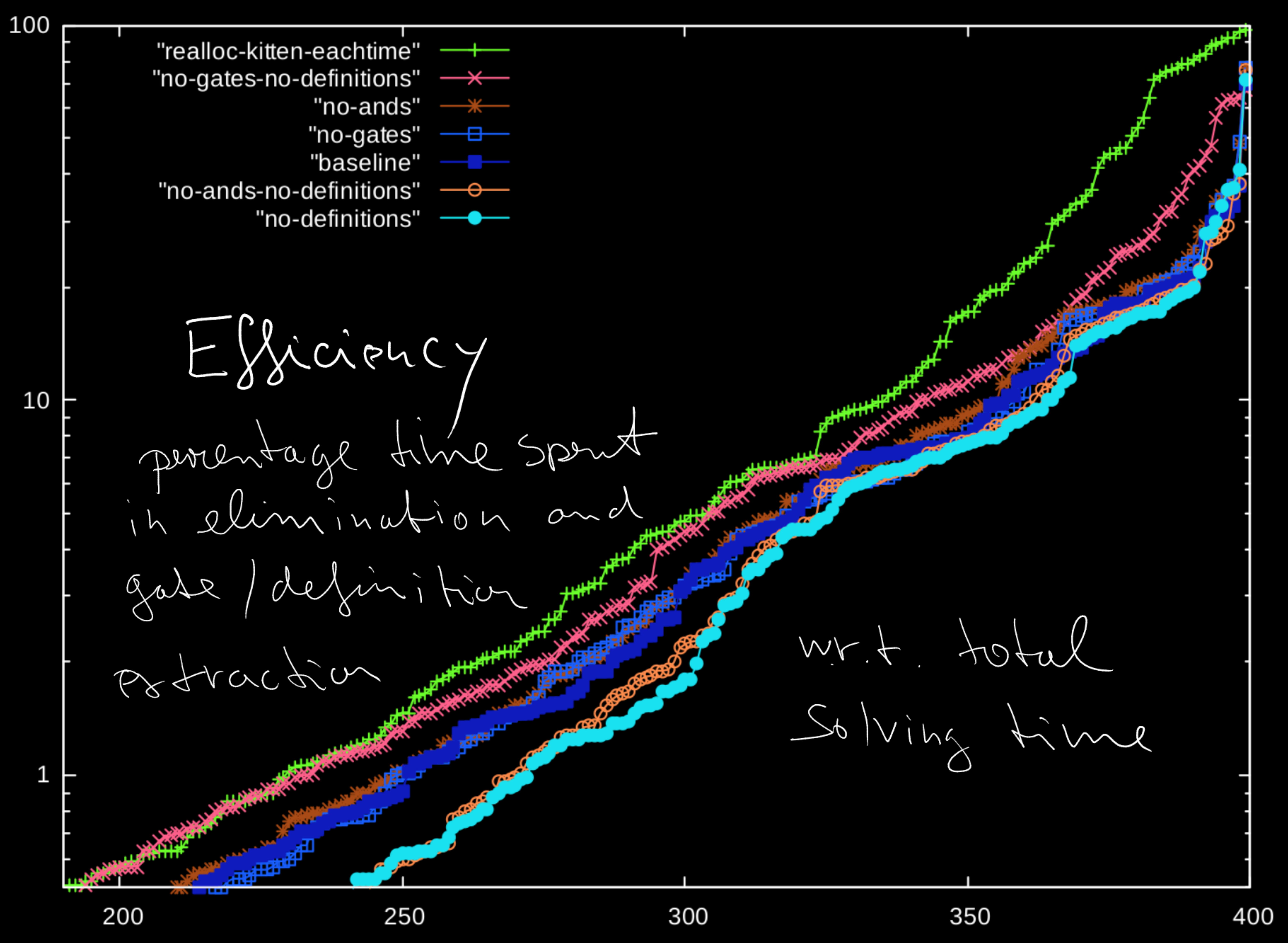

5,000
4,000
3,000
2,000
1,000
0
anniversary instances (ALL)
0
200
400
600
800
1,000
wallclock time
mallob-kicaliglu (cloud winner)
mallob-ki (parallel winner)
kissat-sc2022-bulky (sequential winner)
1,600 = 16 * 100 (virtual) cores
64 virtual cores
one core



Parallel Track SAT Competition 2022
parkissat-rs
nps
dps
pakis22
mergesat-aws
pakismab22
mallob-ki
gimsatul
pmcomsps
pkissat
0
50
100
150
200
250
300
0
1000
2000
3000
4000
5000


Big
KISSAT'21
coredef
POS'21
small embedded
SAT solver
KITTEN
Little
Sweep
in preparation


'/tmp/file.csv' u 1:3
x
no definitions
percentage eliminated variables
More variables eliminated with core based definition extraction!
with Kitten
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Efficiency
percentage time spent in elimination and gate / definition extraction
w.r.t. total solving time
"realloc-kitten-eachtime"
"no-gates-no-definitions"
"no-ands"
"no-gates"
"baseline"
"no-ands-no-definitions"
"no-definitions"
100
10
1
200
250
300
350
400

```c
kitten *kitten_init (void);
void kitten_clear (kitten *);
void kitten_release (kitten *);

void kitten_track_antecedents (kitten *);

void kitten_shuffle_clauses (kitten *);
void kitten_flip_phases (kitten *);
void kitten_randomize_phases (kitten *);

void kitten_assume (kitten *, unsigned lit);

void kitten_clause (kitten *, size_t size, unsigned *);
void kitten_unit (kitten *, unsigned);
void kitten_binary (kitten *, unsigned, unsigned);

void kitten_clause_with_id_and_exception (kitten *, unsigned id,
                                          size_t size, const unsigned *,
                                          unsigned except);

void kitten_no_ticks_limit (kitten *);
void kitten_set_ticks_limit (kitten *, uint64_t);
```

only clear memory aka std::vector.clear

reallocate if release every time

diversify solutions / cores

produce clausal cores in memory

limitted solving

```c
int kitten_solve (kitten *);
int kitten_status (kitten *);

signed char kitten_value (kitten *, unsigned);
bool kitten_failed (kitten *, unsigned);
bool kitten_flip_literal (kitten *, unsigned);

unsigned kitten_compute_clausal_core (kitten *, uint64_t * learned);
void kitten_shrink_to_clausal_core (kitten *);

void kitten_traverse_core_ids (kitten *, void *state,
                               void (*traverse) (void *state, unsigned id));

void kitten_traverse_core_clauses (kitten *, void *state,
                                   void (*traverse) (void *state,
                                                     bool learned, size_t,
                                                     const unsigned *));
```

new secret sauce for sweeping

clausal cores !

Conclusion

- ✡ Assumptions and Assumed Constraints
- ✡ Incremental Inprocessing: Add + Restore
  + tainting
- ✡ Big-Little SAT Solving

Opportunities for making incremental Solving FASTER?

Incremental Inprocessing in SAT Solving

Katalin Fazekas[1](✉), Armin Biere[1], Christoph Scholl[2]

[1] Johannes Kepler University, Linz, Austria
katalin.fazekas@jku.at, armin.biere@jku.at
[2] Albert–Ludwigs–University, Freiburg, Germany
scholl@informatik.uni-freiburg.de

Abstract. Incremental SAT is about solving a sequence of related SAT problems efficiently. It makes use of already learned information to avoid repeating redundant work. Also preprocessing and inprocessing are considered to be crucial. Our calculus uses the most general redundancy property and extends existing inprocessing rules
solving. It allows to automatically reverse earlier
which are inconsistent with literals in new increme
Our approach to incremental SAT solving not only
inprocessing but also substantially improves solvin

1 Introduction

Solving a sequence of related SAT problems incremen
the efficiency of SAT based model checking [5,6,7,8]
domains [9,10,11,12]. Utilizing the effort already spe
keeping learned information (such as variable scores ar
nificantly speed-up solving similar problems. Equally i
plification techniques such as variable elimination, su
resolution, and equivalence reasoning [13,14,15,16].

These simplifications are not only applied before

Mining Definitions in Kissat with Kittens

Mathias Fleury and Armin Biere

Johannes Kepler University Linz, Austria
mathias.fleury@jku.at armin.biere@jku.at

Abstract

Bounded variable elimination is one of the most important preprocessing techniques in SAT solving. It benefits from discovering functional dependencies in the form of definitions encoded in the CNF. While the original approach relied on syntactic pattern matching our new approach uses cores produced by an embedded SAT solver. In contrast to a similar semantic approach in Lingeling based on BDD algorithms, our new approach is able to generate DRAT proofs. We further discuss design choices for our embedded SAT solver Kitten. Experiments with Kissat show the effectiveness of this approach.

Single Clause Assumption without Activation Literals to Speed-up IC3

Nils Froleyks
nils.froleyks@jku.at
Johannes Kepler University, Linz, Autstria

Armin Biere
biere@cs.uni-freiburg.de
Albert–Ludwigs–University, Freiburg, Germany

***Abstract*—We extend the well-established assumption-based interface of incremental SAT solvers to clauses, allowing the addition of a temporary clause that has the same lifespan as literal assumptions. Our approach is efficient and easy to implement in modern CDCL-based solvers. Compared to previous approaches, it does not come with any memory overhead and does not slow down the solver due to disabled activation literals, thus eliminating the need for algorithms like IC3 to restart the SAT**

values greater than five are unreachable. A typical query asks “is state six reachable from any other state?”, expressed as $SAT?[T \wedge (\neg b_2 \vee \neg b_1 \vee b_0) \wedge b_2' \wedge b_1' \wedge \neg b_0']$, where T encodes the transition system for one step from $b_2b_1b_0$ to $b_2'b_1'b_0'$. It is unsatisfiable, telling us that state six is in fact unreachable. We can try to generalize this result to a set of states by considering a *cube* – an assignment to a subset of