CDCL SAT Solving and Applications to Optimization Problems

Alexander Nadel, Intel & Technion, Israel

April 17, 2023

Satisfiability: Theory, Practice, and Beyond Workshop

Simons Institute, UC Berkeley, Berkeley, CA, USA

Agenda

1. Core CDCL SAT Solving
   - Review of the foundations
2. Solving Complex Optimization Problems with SAT
   - Opportunity for the future
     - based on my own industrial experience

Introduction

$$F = \overbrace{(a \vee b)}^{\text{clause \#1}} \wedge \overbrace{(\neg a \vee \neg b \vee c)}^{\text{clause \#2}}$$

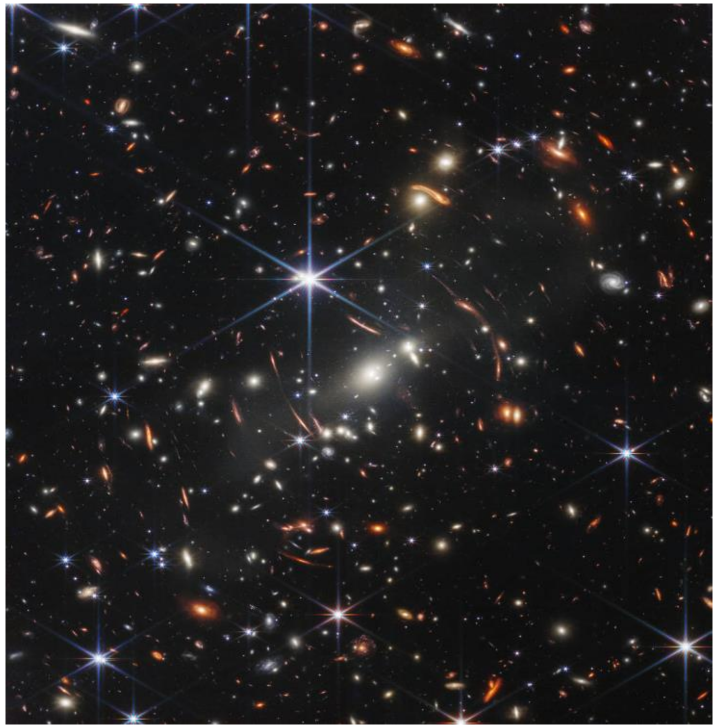
Literals

SAT: determine if a Boolean formula in Conjunctive Normal Form (CNF) satisfiable

The original NP-Complete problem: the famous Cook-Levin theorem (early 70s)

SAT has exponential complexity unless P = NP -- whether or not P = NP is frequently called the most important outstanding question in CS

- Check a solution is easy → is it also easy to solve?



SAT is an unresolved mystery

Yet, SAT solvers are scalable widely used tools, how come?!

SAT Fundamentals: Backtrack Search

The baseline algorithm in modern SAT solvers is backtrack search

Called DPLL or DLL

Davis, Martin; Logemann, George; Loveland, Donald: "A Machine Program for Theorem Proving". Communications of the ACM. 5 (7): 394–397. (1961).

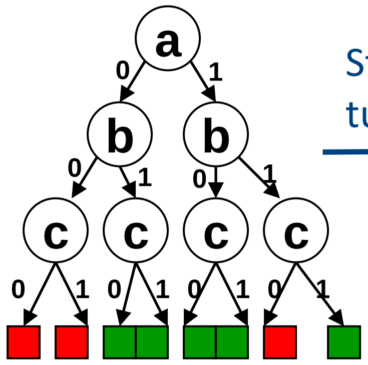
Davis, Martin; Putnam, Hilary: A computing procedure for quantification theory. Journal of the ACM 7 (1960)

From Enumeration to DPLL

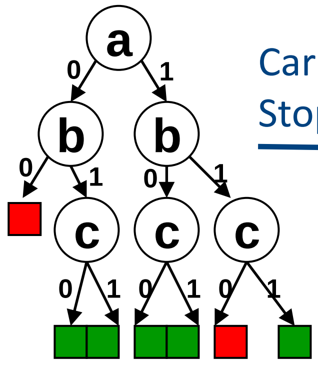
clause #1 clause #2

$$F = (a \lor b) \land (\neg a \lor \neg b \lor c)$$

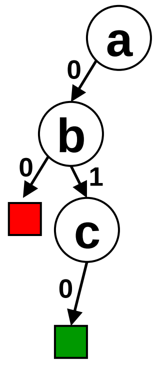
Literals



Stop when a clause turns UNSAT

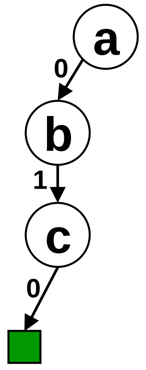


Carry out backtrack search. Stop when a model is found



Apply the unit clause rule till fixed-point aka **Boolean Constraint Propagation (BCP)**

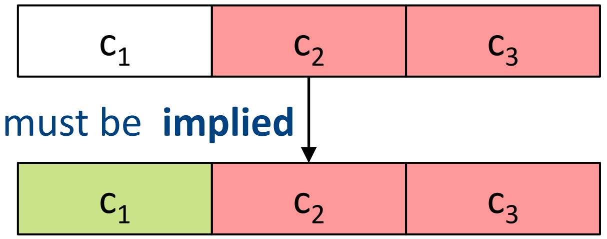
Implied in **parent** clause #1:



The **unit clause rule**: the unassigned literal in a unit clause must be 1

A ***unit*** clause -- one unassigned, rest falsified:

The unassigned literal c_1 must be **implied**



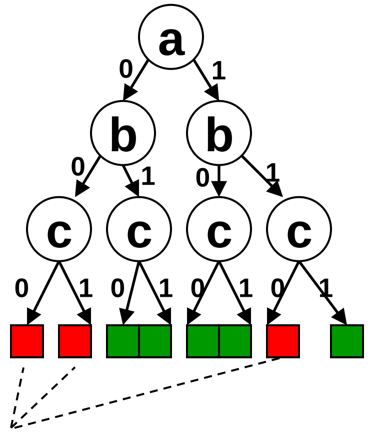
Falsified literal: 🟥 Satisfied literal: 🟩 Unassigned literal: ⬜

The Mystery of SAT Solver Scalability

DPLL could handle formulas with <2,000 clauses

Modern SAT solvers cope with industrial instances of 100,000,000's clauses

The introduction of Conflict-Driven-Clause-Learning (CDCL) or, simply, Conflict-driven Solving was the birth of modern highly-scalable SAT solving



Learn from conflicts to drive & prune backtrack search

CDCL: the Intuitive Principles

Learning and pruning

- Block already explored sub-spaces

Locality

- Focus the search on the relevant data
- Learn strong clauses from the local context

Well-engineered data structures

- Extremely fast Boolean Constraint Propagation (BCP)

Beyond CDCL

- Inprocessing
- Local search integration

Today’s Focus for the 1st Part of the Talk

In-depth dive into the “core of the core”

- Conflict analysis loop
- Boolean Constraint Propagation (BCP)

Conflict-driven SAT Solving: Seminal Work

1996: GRASP by *Joao P. Marques-Silva* and *Karem A. Sakallah*

João P. Marques Silva, Karem A. Sakallah: *GRASP - a new search algorithm for satisfiability*. *ICCAD 1996: 220-227*

2001: Chaff by *Matthew W. Moskewicz*, *Conor F. Madigan*, *Ying Zhao*, *Lintao Zhang* and *Sharad Malik*

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik: Chaff: Engineering an Efficient SAT Solver. DAC 2001: 530-535

CAV Award
2009
Computer Aided Verification
awarded to
Conor F. Madigan
Sharad Malik
João P. Marques-Silva
Matthew W. Moskewicz
Karem A. Sakallah
Lintao Zhang
Ying Zhao
for
fundamental contributions to the development of high-performance Boolean satisfiability solvers

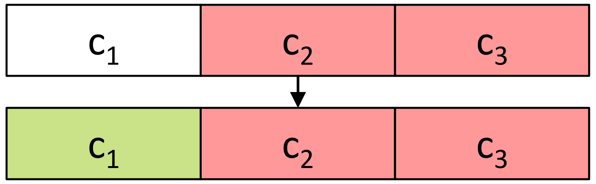
Abstract The 2009 CAV (Computer-Aided Verification) award was presented to seven individuals who made major advances in creating high-performance Boolean satisfiability solvers. This annual award recognizes a specific fundamental contribution or series of outstanding contributions to the CAV field.

Boolean Constraint Propagation (BCP) Essentials

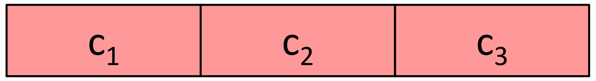
BCP consumes 80-90% of SAT run-time

What?

- Identify and propagate in unit clauses (performance)



- Identify and report any conflicts (correctness)



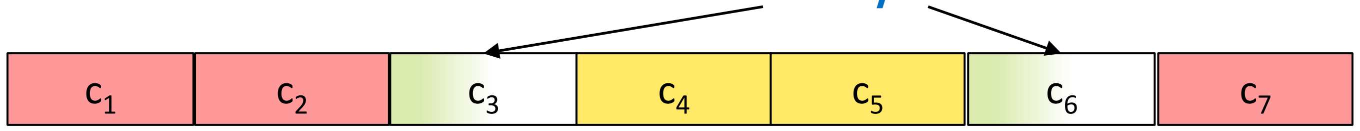
How?

- Every literal *l* holds a *Watch List -- WL(l)* with all the clauses where *l* is *watched*
- When literal l is falsified, visit all the clauses in WL($\neg$l)

Falsified literal: 🟥 Satisfied literal: 🟩 Unassigned literal: ⬜

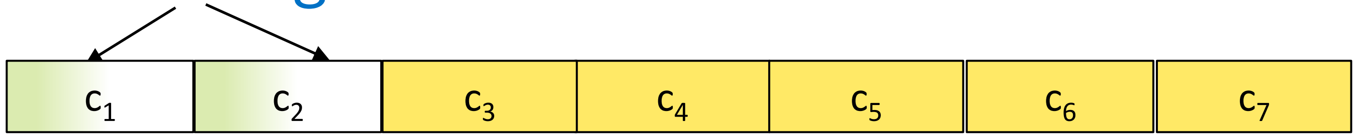
Efficient Data Structure for BCP

- GRASP watched all the literals
- It is sufficient to watch two *non-falsified* literals: SATO's Head/Tail!



Hantao Zhang: SATO: An Efficient Propositional Prover. CADE 1997: 272-275

- Chaff's 2WL: watching the first two literals – no need to visit during backtracking!



  - as long as: decision-level(falsified watch) ≥ decision-level(falsified non-watch)

- Caching one literal inside the watches & inlining binary clauses

Sörensson, N., Eén, N.: MiniSAT 2.1 and MiniSAT++ 1.0 - SAT race Editions. SAT, Competitive Event Booklet (2008) *(caching one literal)*
Geoffrey Chu, Aaron Harwood, Peter J. Stuckey: Cache Conscious Data Structures for Boolean Satisfiability Solvers. J. Satisf. Boolean Model. Comput. 6(1-3): 99-120 (2009) *(caching one literal & inlining binary clauses)*

Falsified literal:	🟥	Satisfied literal:	🟩	Unknown literal:	🟨
Unassigned literal:	⬜	Non-falsified literal:	▨	Non-satisfied literal:	▨

Conflict Analysis Loop in Chaff

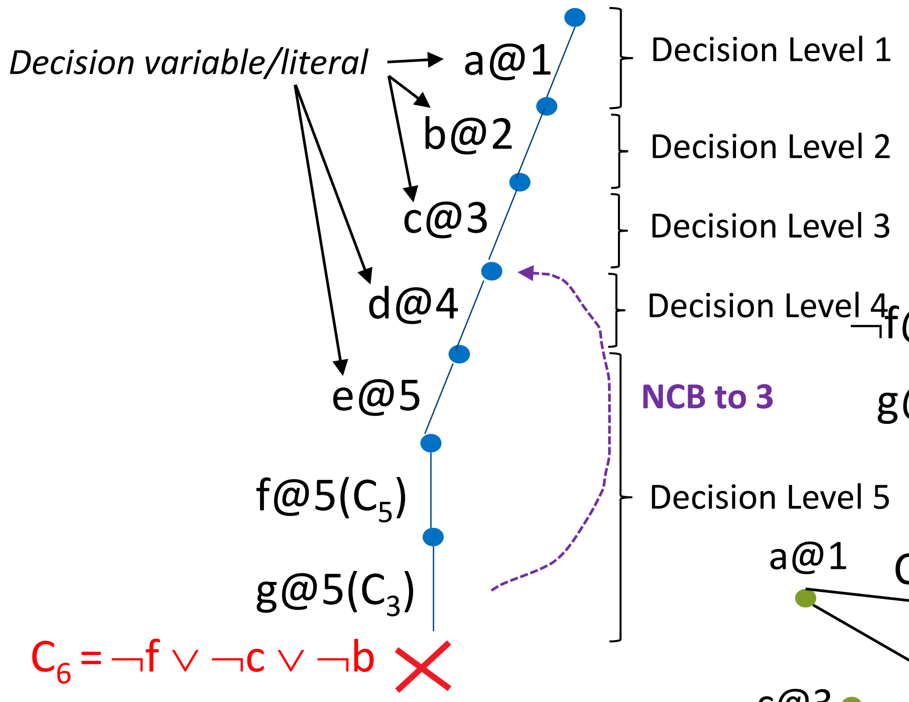
$C_1 = \neg a \vee f \vee g$

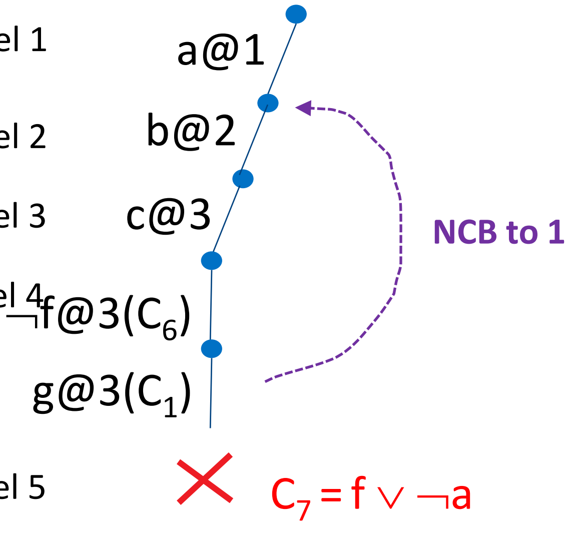
$C_2 = \neg a \vee f \vee \neg g$

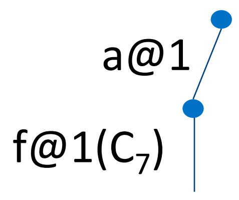
$C_3 = \neg c \vee \neg f \vee g$

$C_4 = \neg b \vee \neg f \vee \neg g$

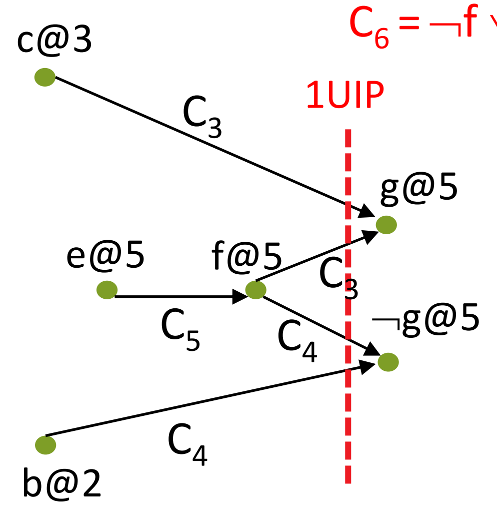
$C_5 = \neg e \vee f$

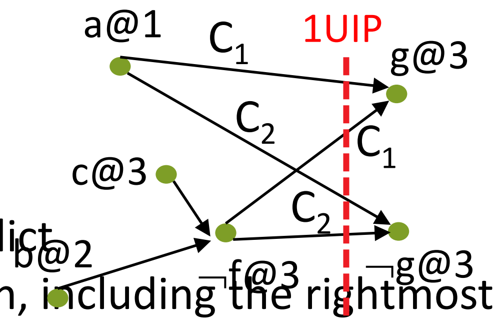






Implication graph





Right-hand side: the conflict

Left-hand side: the reason, including the rightmost Unique Implication Point (UIP) of the last level

- Learn a falsified *asserting* clause $C=[c_1^{@\delta}, c_2^{@\beta<\delta}, c_3^{@\leq\beta}, \ldots, c_{|C|}^{@\leq \beta}]$
    - 1UIP clause in Chaff
- Backtrack to level β: called Non-Chronological Backtracking (NCB) in Chaff → C becomes unit
- Flip & imply c_1 in its parent C and run BCP

Conflict Analysis Loop in GRASP

$C_1 = \neg a \vee f \vee g$

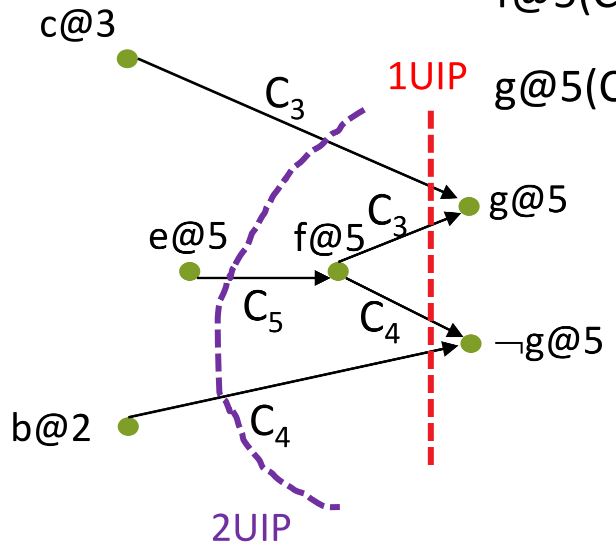
$C_2 = \neg a \vee f \vee \neg g$

$C_3 = \neg c \vee \neg f \vee g$

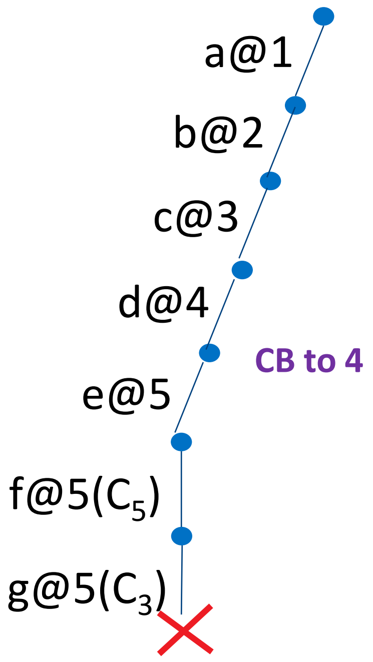
$C_4 = \neg b \vee \neg f \vee \neg g$

$C_5 = \neg e \vee f$

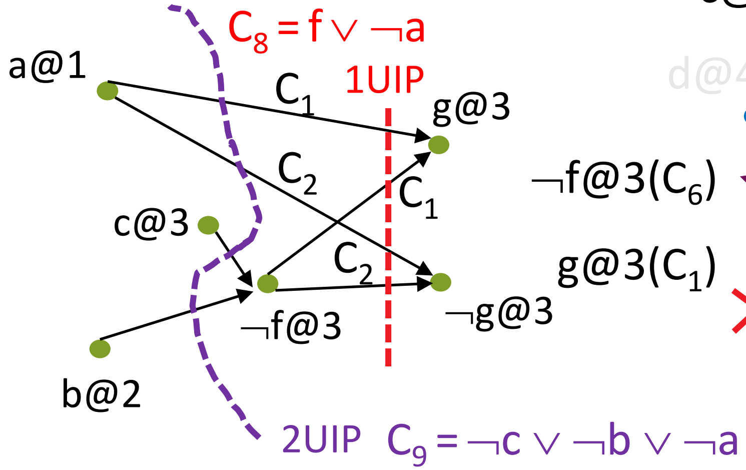
$C_6 = \neg f \vee \neg c \vee \neg b$



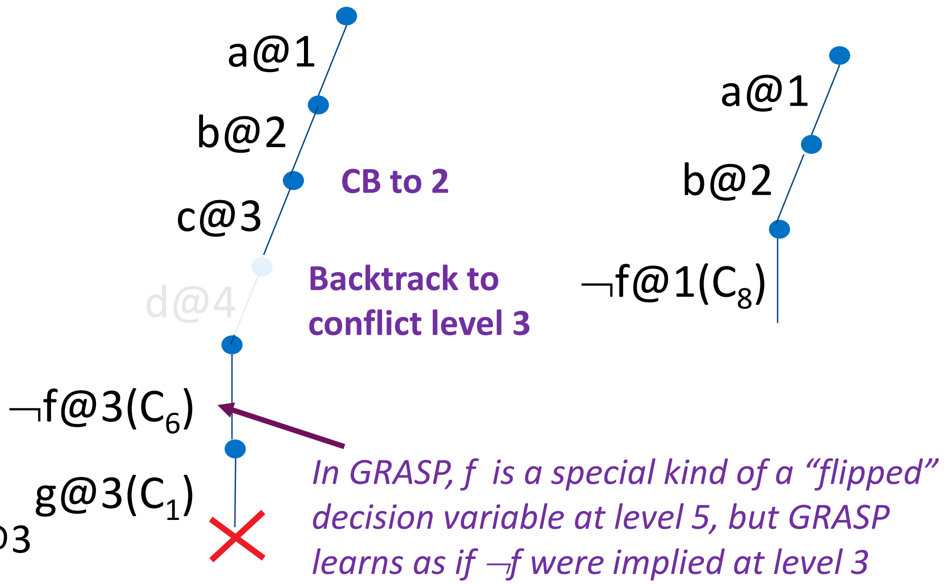
$C_7 = \neg e \vee \neg c \vee \neg b$



$C_8 = f \vee \neg a$



$C_9 = \neg c \vee \neg b \vee \neg a$



In GRASP, f is a special kind of a "flipped" decision variable at level 5, but GRASP learns as if ¬f were implied at level 3

- Backtrack to the conflict level δ: called NCB in GRASP
- Learn a falsified asserting 1UIP clause $C=[c_1^{@\delta}, c_2^{@\beta<\delta}, c_3^{@\leq\beta}, \ldots, c_{|C|}^{@\leq\beta}]$
- Learn a clause per every other UIP of the last level
- Backtrack to level δ-1: called Chronological Backtracking (CB) in later literature!
- Flip & imply c_1 in its parent C and run BCP

Up-to-date Conflict Analysis Loop Algorithm Covers GRASP & Chaff & Modern Solvers

1. Backtrack before conflict analysis: backtrack to the conflict level δ, if required
   - Required in GRASP and called Non-Chronological Backtracking (NCB) in GRASP
   - Not required in Chaff: current decision level $\equiv$ conflict level
2. Learn an asserting clause $C=[c_1{}^{@\delta}, c_2{}^{@\beta<\delta}, c_3@^{@\leq\beta}, \ldots, c_i@^{@\leq\beta}, \ldots, c_{|C|}{}^{@\leq\beta}]$
   - 1UIP clause in both GRASP & Chaff
3. Optionally, learn other clauses
   - GRASP: a clause for every other UIP of the conflict decision level
4. Backtrack: backtrack to a level in $[\beta, \beta+1, \ldots, \delta-1]$ -- makes the asserting clause unit
   - GRASP -- always δ-1: Chronological Backtracking (CB) in today's terminology
   - Chaff -- always β: Non-Chronological Backtracking (NCB) in today's terminology
5. Flip c_1 by implying it in C and run BCP

Conflict Analysis Loop Evolvement

1996 GRASP

2001 Chaff

Chaff's alg. (one 1UIP cls. & NCB) is the state-of-the-art

2018 Maple_LCM_Dist_ChronoBT

2019 Cadical'19

Maple_LCM_Dist_ChronoBT (MapleCB): the return of Chronological Backtracking (CB)

Alexander Nadel, Vadim Ryvchin: Chronological Backtracking. *SAT 2018: 111-121*

- A backtracking heuristic choosing between CB and NCB
  - Today: Maple-based solvers, Cryptominisat, Kissat alter between CB and NCB
- CB algorithm is similar to GRASP's
- Integrating CB with post-GRASP data structures for BCP (Watch Lists) turned out to be highly non-trivial
  - Because of simultaneous propagation at several levels
  - BCP must be adjusted to prevent correctness & performance issues
  - Useful BCP invariants are still violated

Cadical'19: custom (score-based) backtracking

Sibylle Möhle, Armin Biere: Backing Backtracking. *SAT 2019: 250-266*

- Backtrack to the decision level with the highest variable score
- Applied by Cadical & IntelSAT

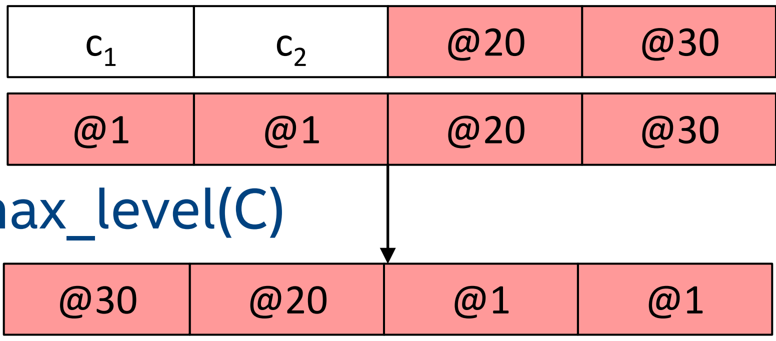
CB & BCP integration: implemented, but not discussed

Falsified literal: 🟥 Unassigned literal: ☐

Satisfied literal: 🟩

Integrating CB and BCP

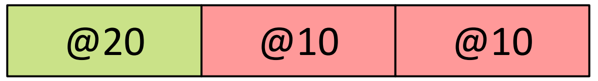
Example of a necessary adjustment



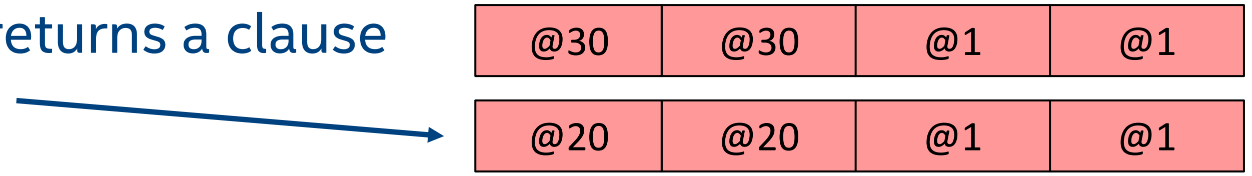
- $\neg c_1$ and $\neg c_2$ are assigned @1 < max_level(C)
  - Can't happen with NCB, where the assigned level is always ≥ max_level(C)
- Must swap lit's & update WL's to watch two highest falsified lit's
- Essential for correctness – in order not to miss conflicts after backtracking!

Useful invariants are still violated even with the adjustments:

- **lowest implication**: no assigned literal can be implied at a lower level ✗



- **lowest conflict**: every conflict, BCP returns a clause falsified at the lowest possible level



Intel® SAT Solver (IntelSAT): lowest implication & lowest conflict ensured!

Intel® SAT Solver (IntelSAT)

An open-source CDCL solver written from scratch in C++20

- Alexander Nadel: Introducing Intel® SAT Solver. SAT 2022.
- Alexander Nadel, “Introducing Intel® SAT Solver” [video], MIAO Seminars. February 2023.

License: MIT (free)

Public repository: https://github.com/alexander-nadel/intel_sat_solver

Tuned towards incremental applications with mostly SAT queries

- Paper: anytime unweighted MaxSAT
- @Intel: optimization problems – placement, scheduling, etc.

Optimization in SAT

OptSAT(F, ψ): given a propositional formula F in CNF and a Pseudo-Boolean objective function ψ, return a model to F which minimizes ψ

- A Pseudo-Boolean (PB) function: a mapping from every full assignment to a real number

Example: F = (a + b) (a + ¬c) (¬a + c)

F has 3 models:

- M_1={a=0, b=1, c=0}
- M_2={a=1, b=0, c=1}
- M_3={a=1, b=1, c=1}

a	b	c	ψ
0	0	0	2.3
0	0	1	3.5
0	1	0	8
0	1	1	100.1
1	0	0	96.3
1	0	1	75
1	1	0	1.35
1	1	1	20.4

Optimization in SAT

OptSAT(F, ψ): given a propositional formula F in CNF and a Pseudo-Boolean objective function ψ, return a model to F which minimizes ψ

- A Pseudo-Boolean (PB) function: a mapping from every full assignment to a real number

Example: $F = (a + b)(a + \neg c)(\neg a + c)$

F has 3 models:

- **$M_1=\{a=0, b=1, c=0\}$**
- $M_2=\{a=1, b=0, c=1\}$
- $M_3=\{a=1, b=1, c=1\}$

Best model

a	b	c	ψ
0	0	0	2.3
0	0	1	3.5
0	1	0	8
0	1	1	100.1
1	0	0	96.3
1	0	1	75
1	1	0	1.35
1	1	1	20.4

Solving OptSAT(F, ψ) Instances in Real-life

Is ψ is a linear PB function: $\psi = w_{n-1}*t_{n-1} + \ldots + w_1*t_1 + \ldots + w_0*t_0$?

- t_i's are Boolean variables
- w_i's are strictly positive integer coefficients
- Example: $\psi = 2*t_2 + 5*t_1 + 7*t_0$

Yes

MaxSAT: a rich well-established field!

No

Scarce research

Our contribution

- Polosat algorithm: simulate local search with SAT (Nadel, FMCAD'20)
  - Very efficient and simple to implement
  - In focus today
- High-level local search with SAT/Polosat as an oracle (Cohen&Nadel&Ryvchin, TACAS'22)

Polosat: Black-Box Optimization in SAT

Polosat: minimize a black-function $\psi(V)$, given the SAT formula $F(V)$

How to use Polosat:

1. Similarly to SAT, create the CNF formula by adding clauses
2. Call the SAT solver (possibly under assumptions), but also provide ψ as a callback function
   - The solver will query ψ, when all the variables are assigned
   - The solver expects to get back a number

No need to bit-blast ψ into clauses: calculate ψ in the callback instead!

In practice:

- ψ depends only on observables $B= \{b_{n-1}, \dots , b_0\} \subseteq V$, and
- ψ is strictly monotone in B: $b_i \in B$ flipped from 1 to 0 $\rightarrow$ ψ is decreased
  - Example: MaxSAT, where $\psi = w_{n-1}*b_{n-1} + \dots + w_1*b_1 + \dots + w_0*b_0$
- these restrictions can be lifted

Polosat Algorithm: Simulate Local Search with SAT

Polosat (F, ψ, T)

- M := SAT(F)
- External loop: run the following internal loop until M is not improved anymore
  - Internal loop: go over all the **bad** observables (bad: never assigned 0 in any model)
    - – Try to **flip** the current bad observable b:
      - » M' := SAT(F, $\{\neg b\}$) ($\neg$t is an assumption)
    - – If (satisfiable and $\psi(M') < \psi(M)$) M := M'
- Return M

No model can be rediscovered by construction

Making Polosat work in practice:

- Apply **polarity-fixing** in all the SAT invocations to **simulate local search**
  - TORC heuristic: fix the observables to 0 and the rest to the best model so far M
- Use a **conflict threshold** N (N=1000): limit every SAT call (except for the 1st one) by N conflicts
- Initial **boost to the VSIDS scores** of the observables can also be useful

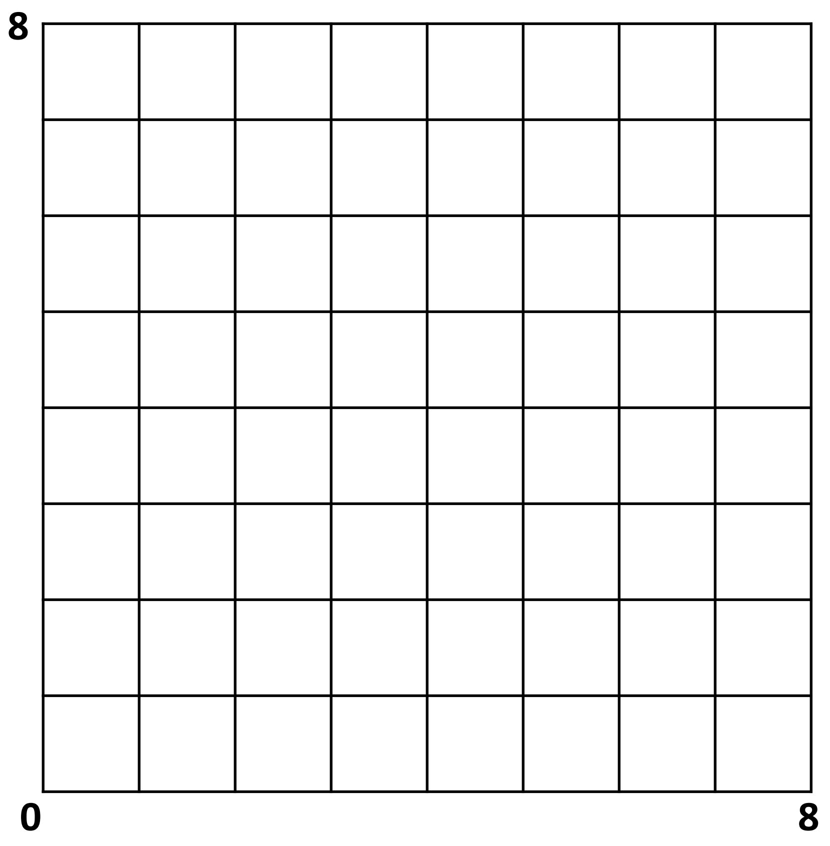
Polosat: Incomplete vs. Complete

Polosat is an incomplete algorithm

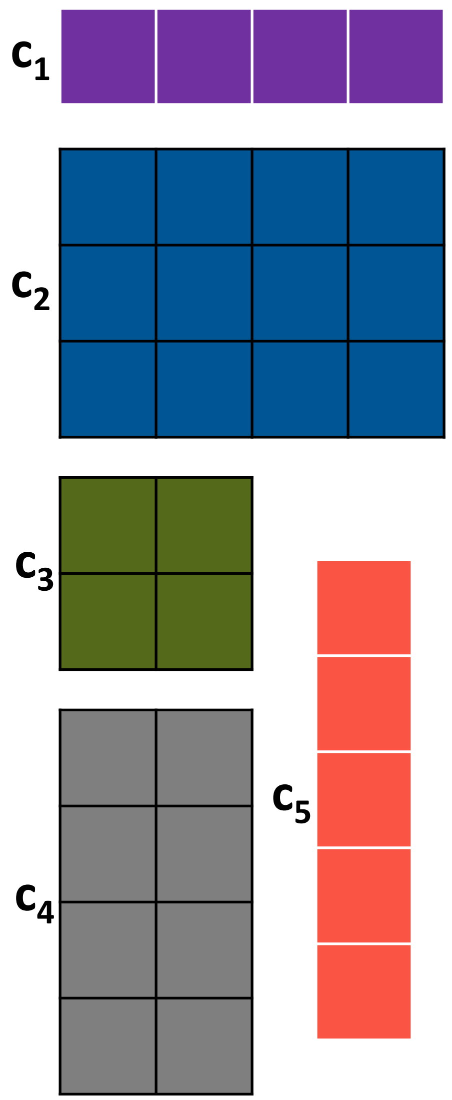
Polosat can be integrated into a high-level complete algorithm by **replacing** SAT queries to Polosat queries

Cell Placement without Optimization: Input

The grid where to place the cells

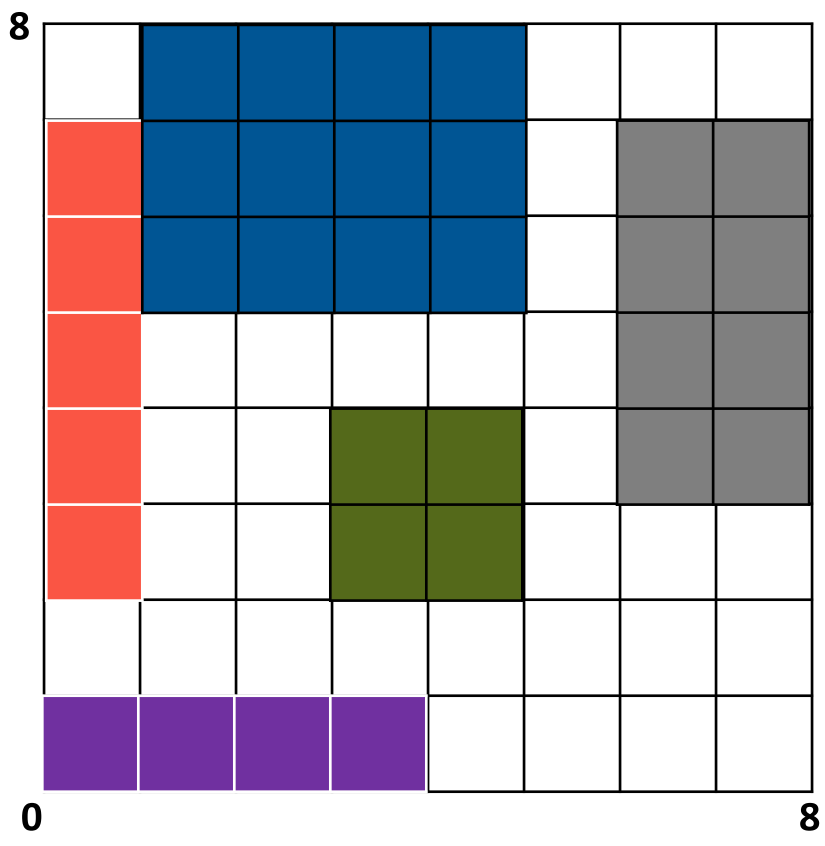


The cells to be placed



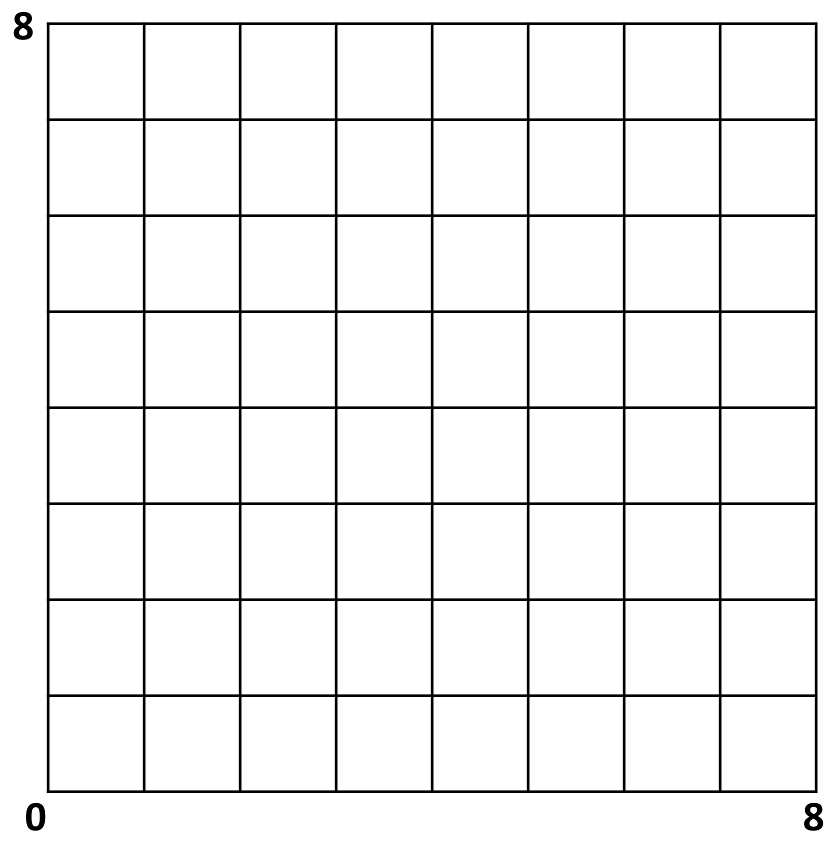
Cell Placement w/o Opt.: Output

- The cells are placed
- Already NP-complete!



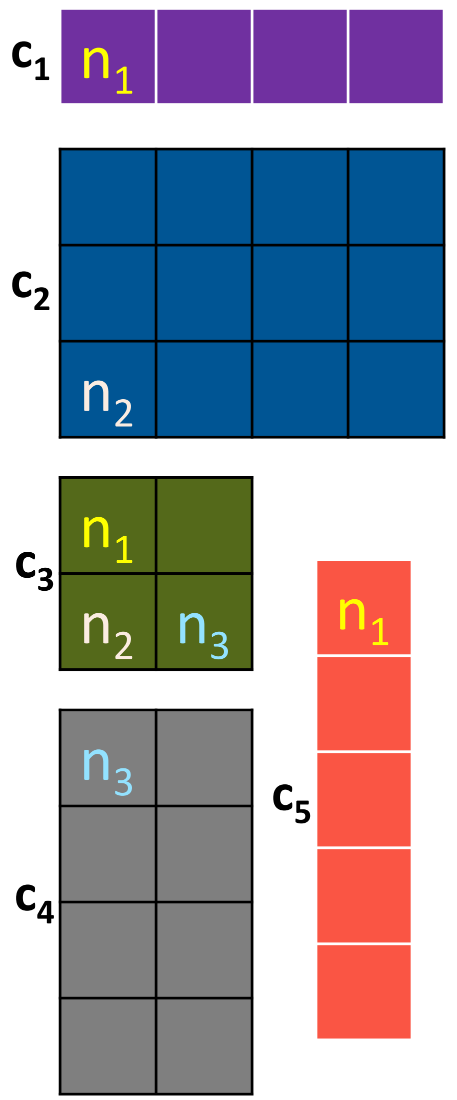
Cell Placement Input

The grid where to place the cells



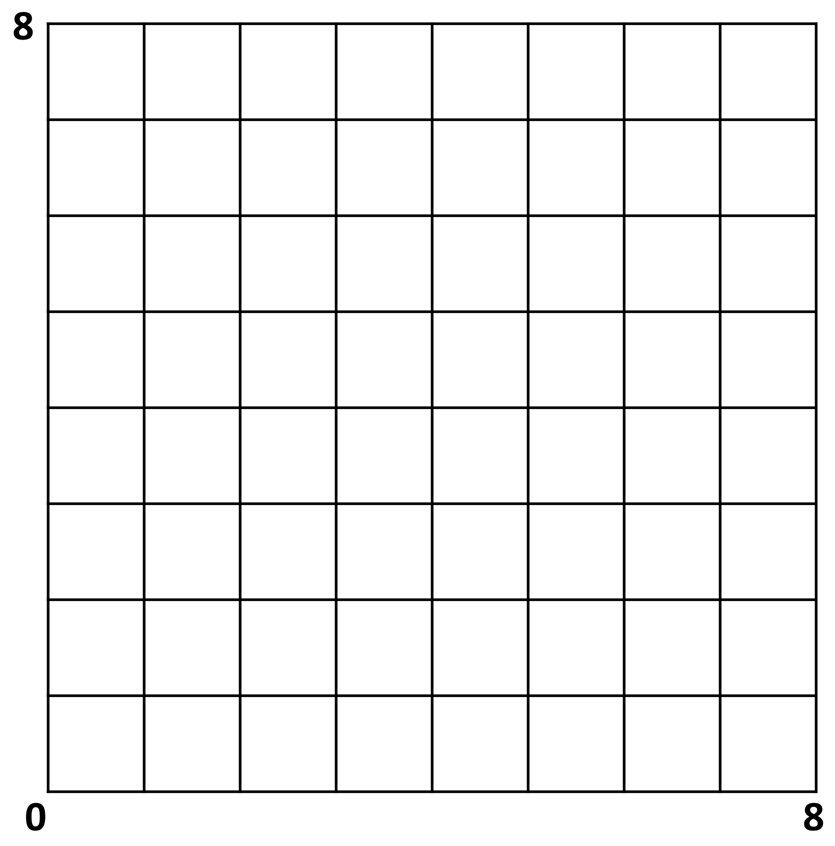
N nets: $n_1=\{c_1,c_3,c_5\}$; $n_2=\{c_2,c_3\}$; $n_{N=3}=\{c_3,c_4\}$

The cells to be placed



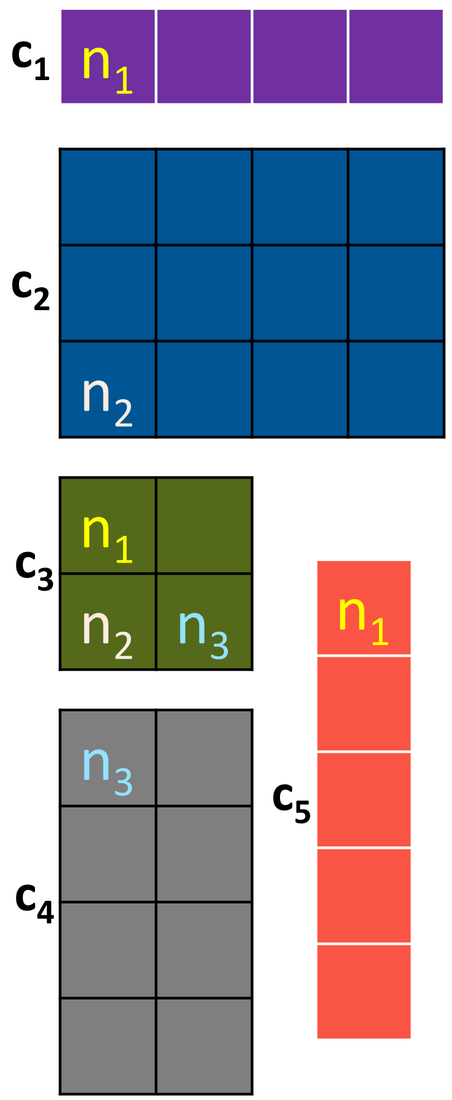
Cell Placement Input

The grid where to place the cells

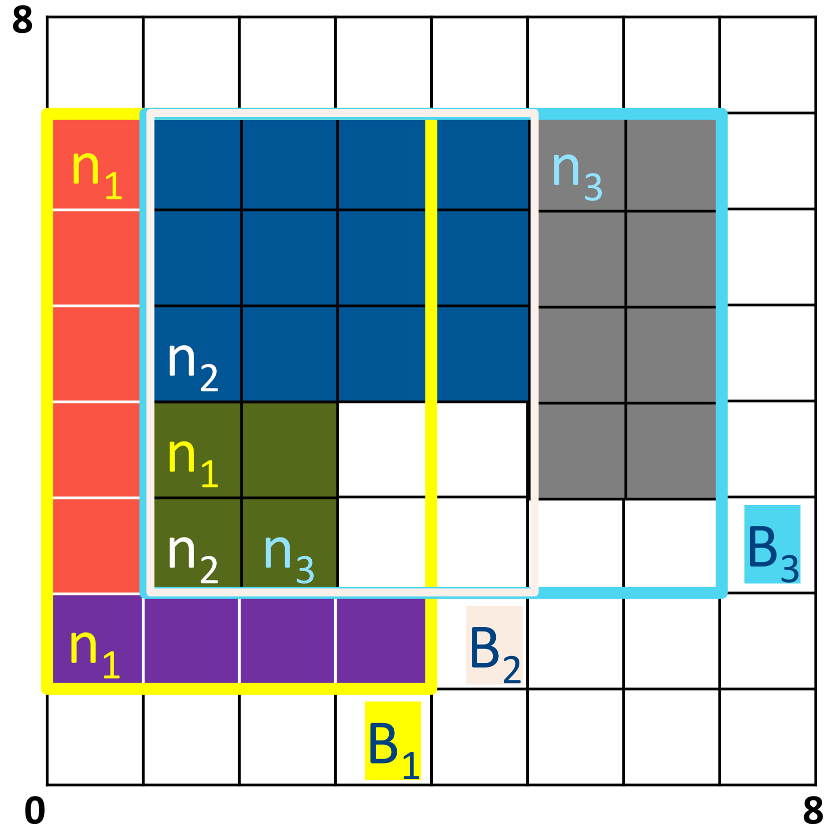


N nets: $n_1=\{c_1,c_3,c_5\}$; $n_2=\{c_2,c_3\}$; $n_{N=3}=\{c_3,c_4\}$

The cells to be placed

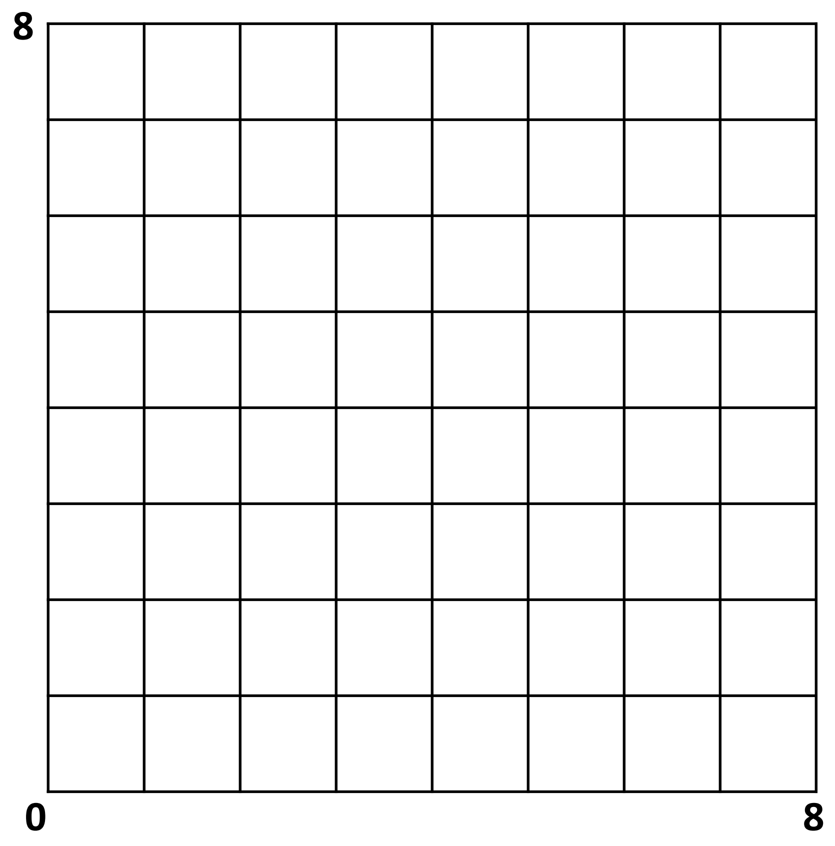


- Net size $|n_i|$ of n_i: the perimeter of n_i's bounding box B_i

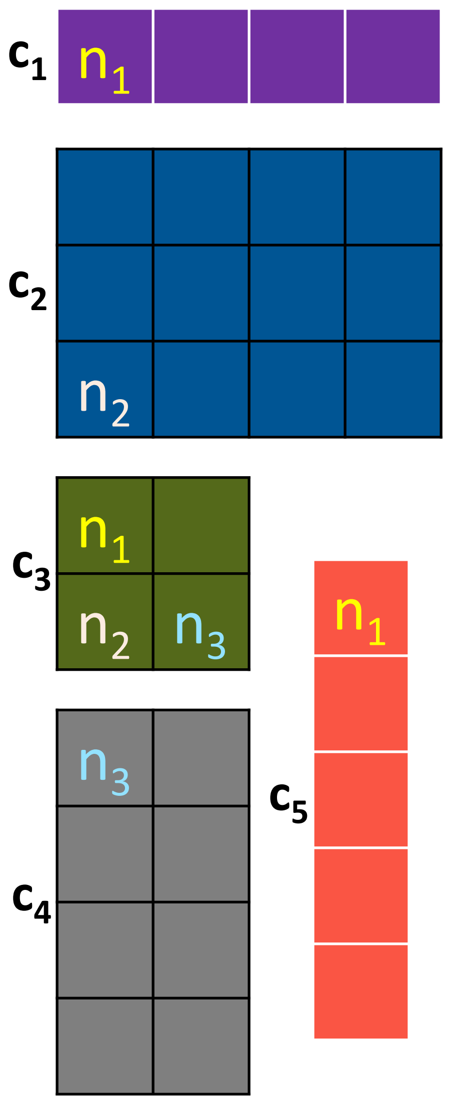


Cell Placement Input

The grid where to place the cells



The cells to be placed

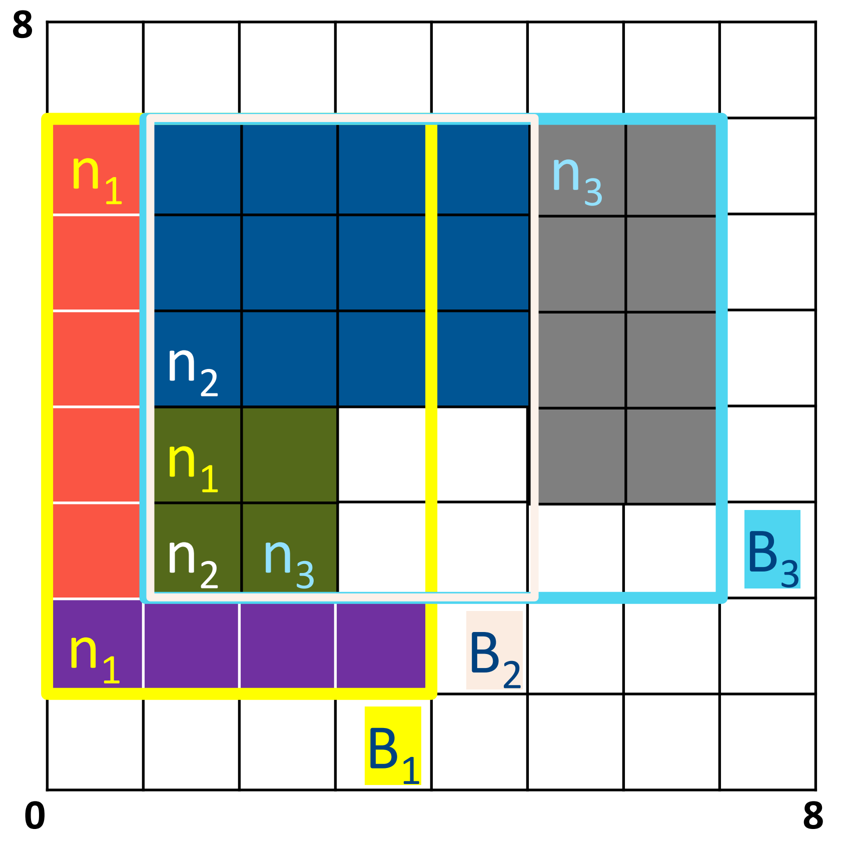


N nets: $n_1=\{c_1,c_3,c_5\}$; $n_2=\{c_2,c_3\}$; $n_{N=3}=\{c_3,c_4\}$

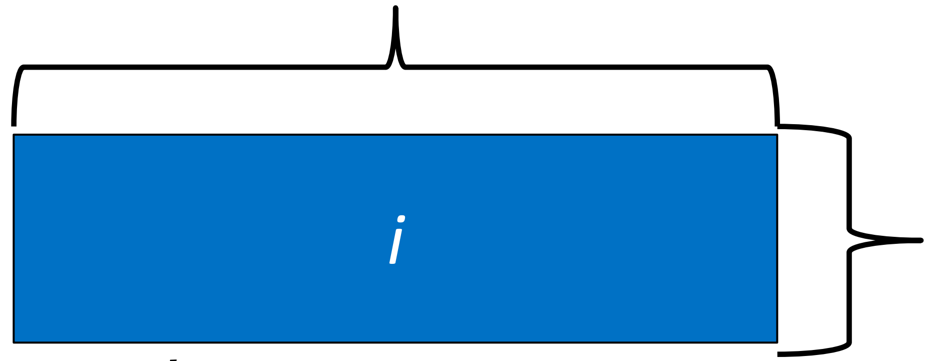
Industrial practice: additional constraints!

Cell Placement with Opt. Output

- Net size $|n_i|$ of n_i: the perimeter of n_i's bounding box B_i
- Minimize placement size: the sum of all the net sizes



Cell Placement → BV/SAT : Constraints



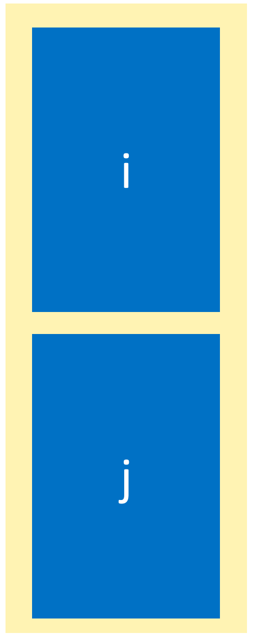
c_i^{width}: the constant width

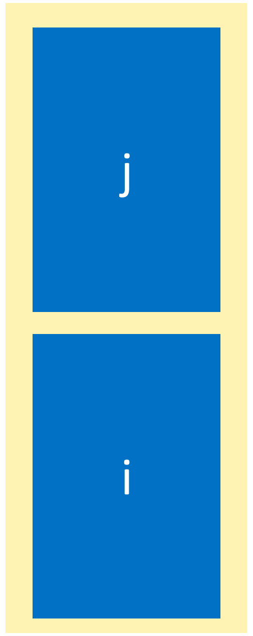
c_i^{height} : the constant height

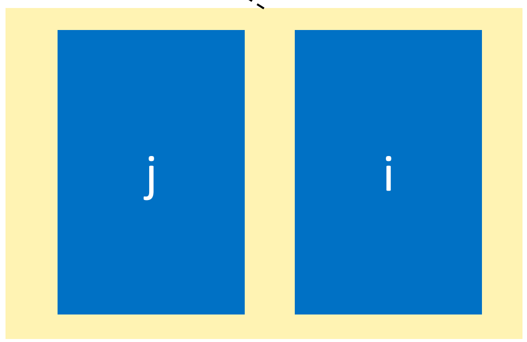
$(c_i^{west}, c_i^{south})$: two bit-vectors, representing the bottom-most corner
(widths' determination is skipped here)

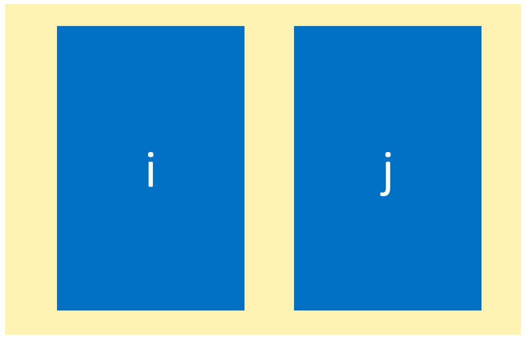
To find a solution: ensure there is no overlap between each pair of cells
and all the cells are placed inside the grid (skipped here)

$\forall i, j\!: 1 \le i < j \le N\!: (c_i^{west} \ge c_j^{east}) \lor (c_j^{west} \ge c_i^{east}) \lor (c_i^{south} \ge c_j^{north}) \lor (c_j^{south} \ge c_i^{north})$



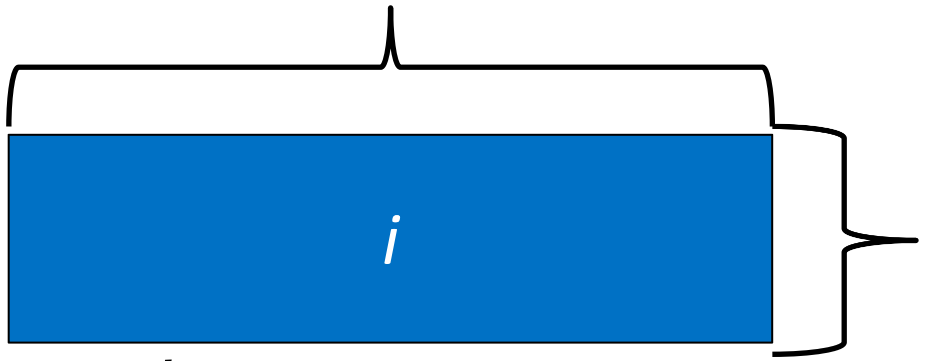






Solving Placement with Optimization Modulo Bitvectors (OBV)

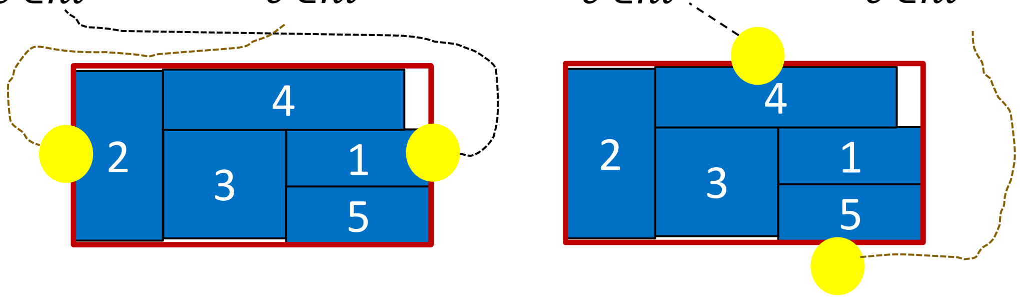
c_i^{width}: the constant width



c_i^{height} : the constant height

$(c_i^{west}, c_i^{south})$: two bit-vectors, representing the bottom-most corner for each cell

A BV variable for the net size for each net: $n_i = \left(\max_{c \in ni} c^{east} - \min_{c \in ni} c^{west}\right) + \left(\max_{c \in ni} c^{north} - \min_{c \in ni} c^{south}\right)$



The OBV target T = $n_1+n_2+\ldots+n_N$

OBV goal: minimize T

Solving OBV(F,T) with SAT-based Linear Search

```
$solver.Assert(F); \mu := solver.Sat()$    ▷ assert $F$ and find the first solution
**while** $\mu$ is a solution **do**    ▷ while there is still a solution
    $solver.Assert(T < \mu(T))$    ▷ block all the solutions with cost $\geq \mu(T)$
    $\mu := solver.Sat()$    ▷ can we improve our solution?
**return** $\mu$    ▷ $\mu$ is guaranteed to be $T$-minimal
```

+ Complete anytime algorithm
- anytime algorithm: finds better and better solutions, the longer it keeps running

+ Outperforms other OBV algorithms (binary-search-based)

- Still, gets stuck far from the optimum on industrial placement instances
- bit-vector addition (in the target) is too heavy

Polosat for Placement

Integration: Polosat invocations **replace** SAT invocations inside linear search

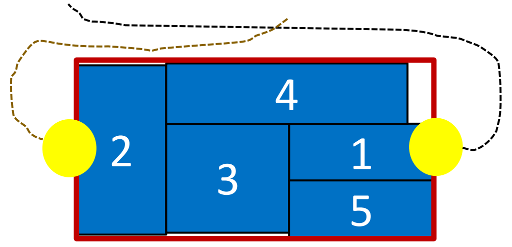
ψ simply returns the value of T under the current model

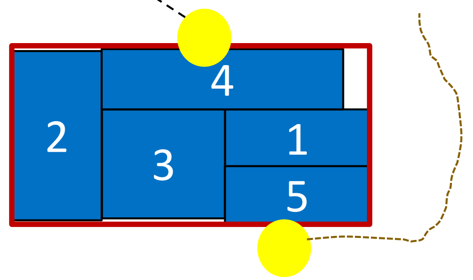
Observables B = all the bits of $\{n_1, n_2, \ldots, n_N\}$; ψ is monotone in B

```
1: $solver.Assert(F); \mu := solver.Sat()$          ▷ assert $F$ and find the first solution
2: while $\mu$ is a solution do                       ▷ while there is still a solution
3:     $solver.Assert(T < \mu(T))$                    ▷ block all the solutions with cost $\geq \mu(T)$
4:     $\mu := solver.Sat()$                          ▷ can we improve our solution?
5: return $\mu$                                       ▷ $\mu$ is guaranteed to be $T$-minimal
```

A BV variable for the net size for each net: $n_i = \left(\max_{c \in ni} c^{east} - \min_{c \in ni} c^{west}\right) + \left(\max_{c \in ni} c^{north} - \min_{c \in ni} c^{south}\right)$

The OBV target $T = n_1 + n_2 + \ldots + n_N$

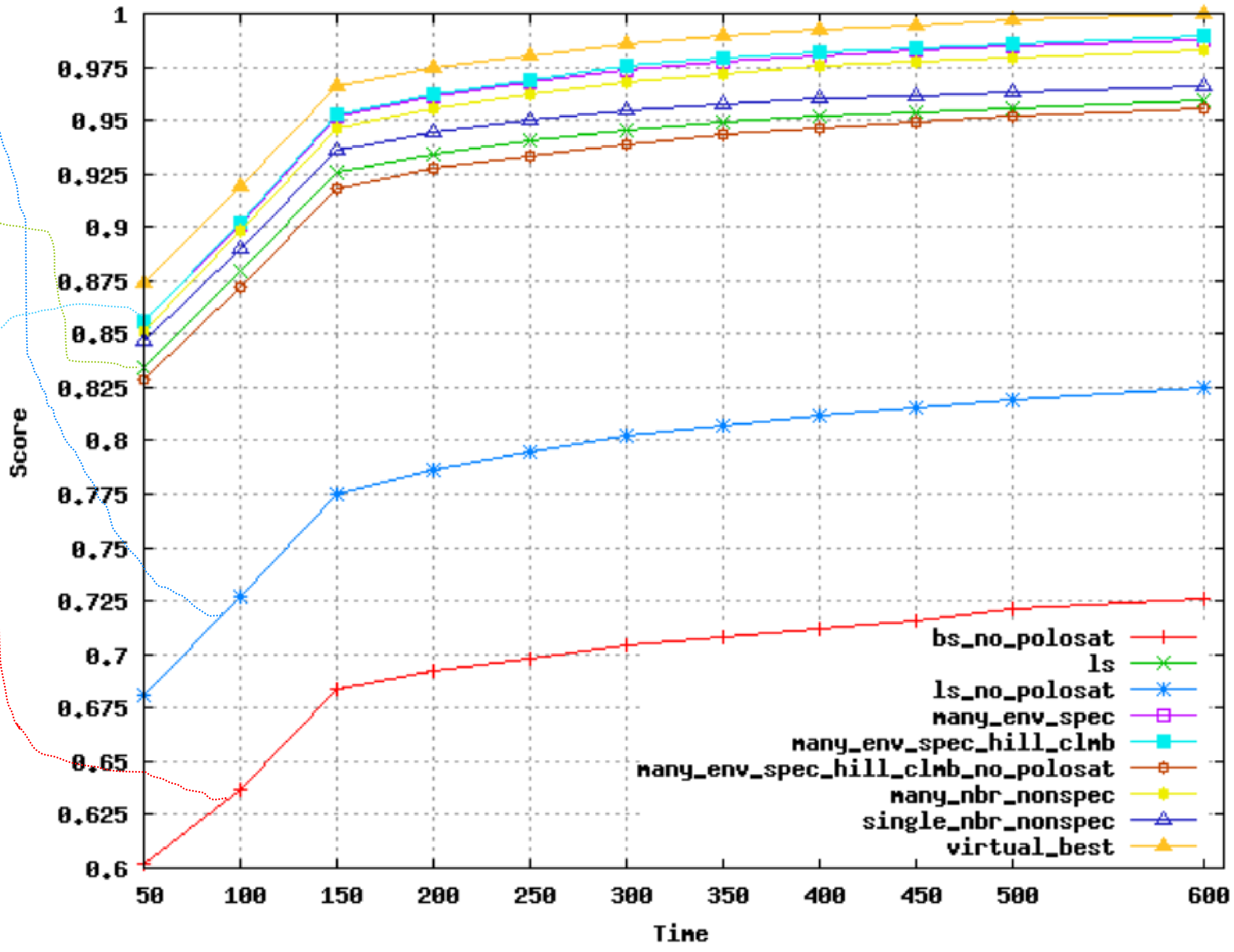




Experimental Results: Industrial Cell Placement Benchmarks (our TACAS'22 paper)

- 1200 proprietary industrial designs of various sizes and complexities.
- Algorithms:
  - {OBV with Binary search, OBV with Linear search, LSSO} x {SAT, Polosat}
    - LSSO: Local Search with SAT/Polosat as an Oracle (our TACAS'22 paper)
- Maximal timeout: 600 sec.
  - Intermediate timeouts: 50, 100, 150, 200, 250, 300, 350, 400, 450, 500
- Score = best Σ(net-sizes) / my Σ(net-sizes)
  - normalized to [0, 1] for every timeout
  - 1: the absolutely best result within the timeout (the virtual best)
  - the closer to 1 the better

- Polosat impact
  - Linear search with SAT (ls_no_polosat) outperforms binary search with SAT (bs_no_polosat)
  - Linear search with Polosat (ls) outperforms linear search with SAT (ls_no_polosat)
- LSSO with SAT/Polosat as an Oracle
  - **Best LSSO with Polosat** outperforms the linear search with Polosat
  - Out tool has been successfully productized at Intel!



Polosat for MaxSAT: our FMCAD'20 Paper

Integrated into the anytime MaxSAT solver TT-Open-WBO-Inc

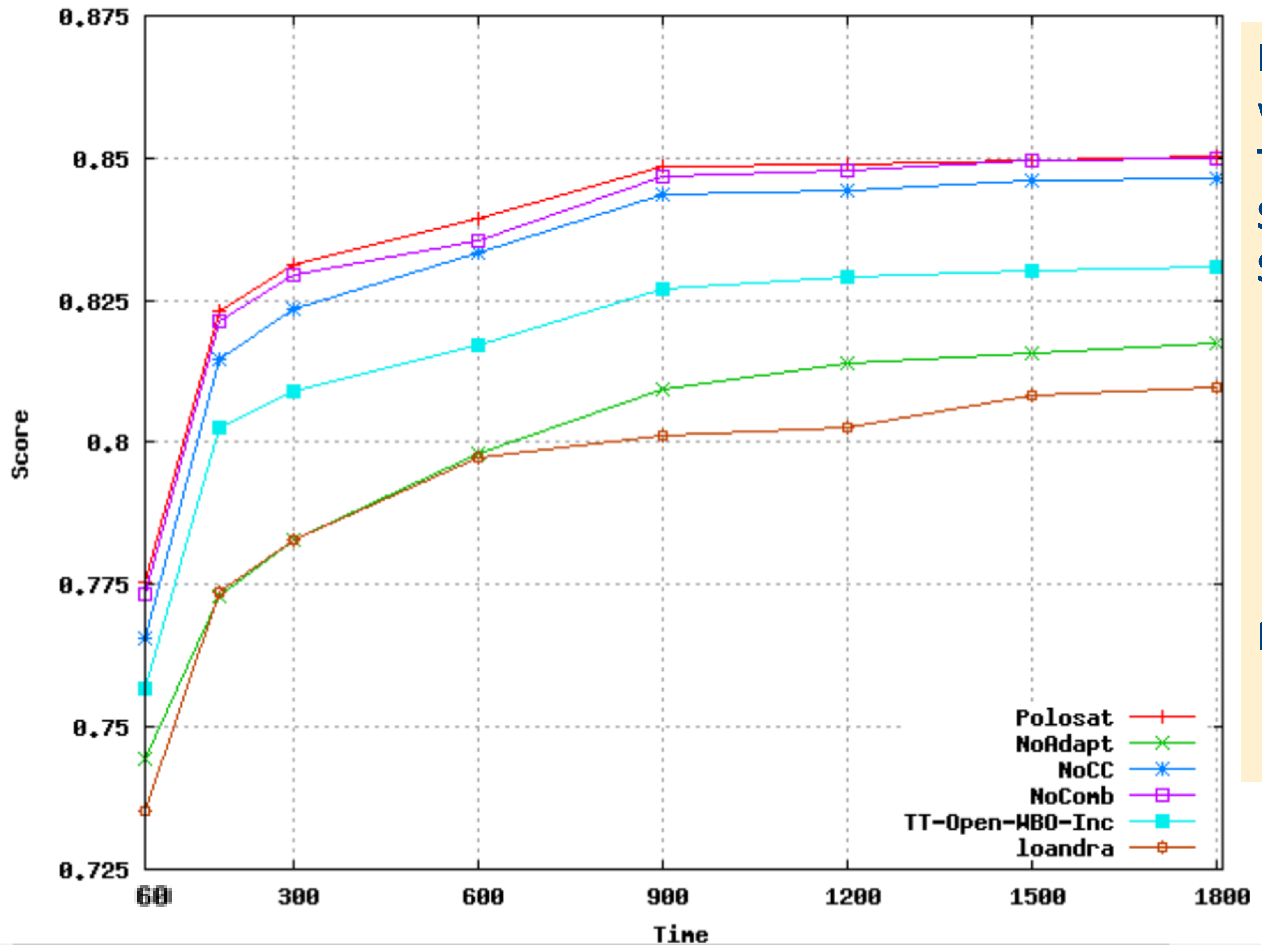
- Winner of MSE'19 in both the weighted, incomplete categories

Integration: replaced SAT invocations by Polosat invocations

Used adaptive strategy to stop Polosat forever, when it gets too slow

  - Gets too slow: generates less than 1 new model per second

Polosat for MaxSAT: Results



Benchmarks: 297 MSE'19 benchmarks in weighted, incomplete categories

Timeout: 1800 sec.

Score: [0, 1]: 1 is the best

Solvers:

- Polosat
- NoComb: A Polosat variation
- NoCC: A Polosat variation
- TT-Open-WBO-Inc: MSE'19 winner
- NoAdapt: No adaptive strategy
- Loandra: MSE'19 runner-up

Main Observation:

Polosat substantially improves TT-Open-WBO-Inc!

Polosat: Status

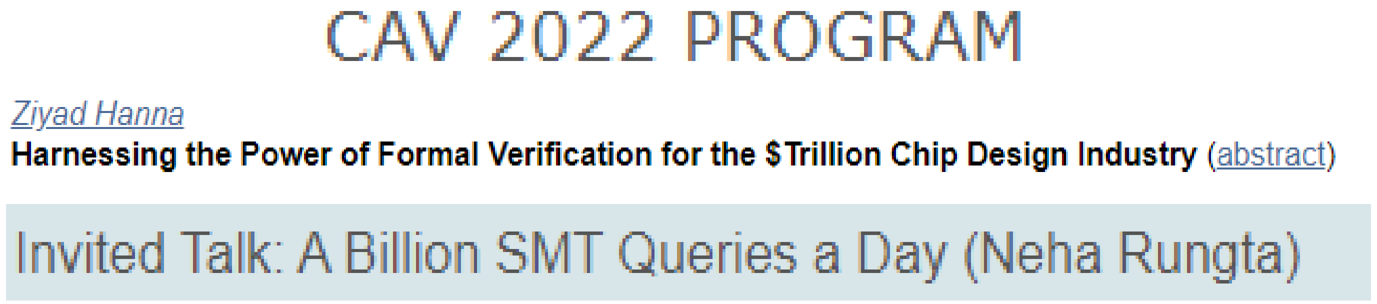
Polosat is an enabler for solving industrial optimization problems at Intel

Polosat was used by the winner of the MaxSAT Evaluation 2022 in all the incomplete categories

- NuWLS-c: preprocessing with local search + TT-Open-WBO-Inc
- Categories: {weighted,unweighted} X {60 sec. to, 300 sec. to}

Solving Complex (Non-Linear) Optimization Problems is an Opportunity for the SAT Community!

Currently, SAT-based verification comprises SAT's heaviest industrial usage



CAV 2022 PROGRAM

Ziyad Hanna
Harnessing the Power of Formal Verification for the $Trillion Chip Design Industry (abstract)

Invited Talk: A Billion SMT Queries a Day (Neha Rungta)

But what about industrial optimization problems?

Alternative: mixed-integer nonlinear programming (MINP)

- Might not work in the presence of complex Boolean constraints, where SAT-based solutions are expected to be a better fit (similarly to MaxSAT vs. ILP in linear case)
- Optimizing black-box functions in MINP is not explored

A SAT-based solution is already productized @ Intel for placement and scheduling!

Solving Complex Optimization Problems with SAT: Ideas for Future Research

Classic (non-SAT-based) local search

Dedicated algorithms for sub-classes of optimization functions

Going beyond SAT constraints (Pseudo-Boolean)

Finding more applications

Backup

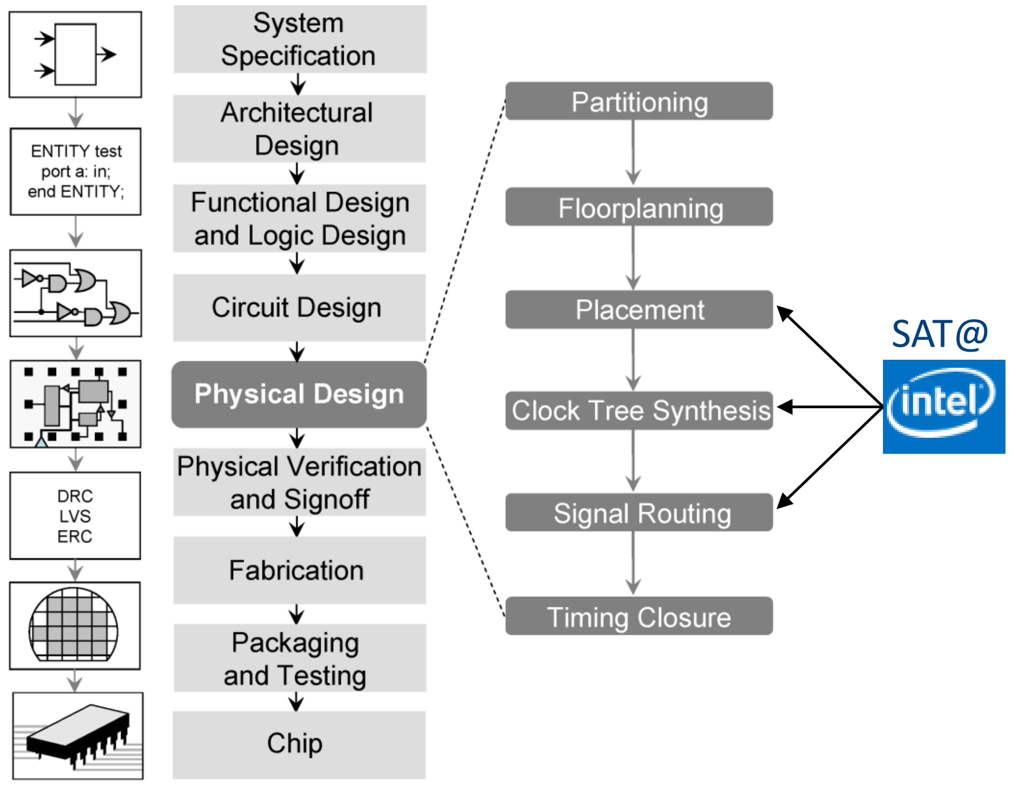
SAT Application Examples

CAV 2022 PROGRAM

Ziyad Hanna

Harnessing the Power of Formal Verification for the $Trillion Chip Design Industry (abstract)

Invited Talk: A Billion SMT Queries a Day (Neha Rungta)



MAY 30, 2016 REPORT

Computer generated math proof is largest ever at 200 terabytes

by Bob Yirka , Phys.org

Credit: Victorgrigas/Wikideia/ CC BY-SA 3.0

(Phys.org)—A trio of researchers has solved a single math problem by using a supercomputer to grind through over a trillion color combination possibilities, and in the process has generated the largest math proof ever—the text of it is 200 terabytes in size. In their paper uploaded to the preprint server *arXiv*, Marijn Heule with the University of Texas, Oliver Kullmann with Swansea University and Victor Marek with the University of Kentucky outline the math problem, the means by which a supercomputer was programmed to solve it, and the answer which the proof was asked to provide.

The math problem has been named the boolean Pythagorean Triples problem and was first proposed back in the 1980's by mathematician Ronald Graham. In looking

IntelSAT Concepts

Incremental Lazy Backtracking (ILB)

- In-between incremental queries, backtrack only when necessary and to the highest possible level
  - Other solvers backtrack all the way to level 0

Reimplication: new core SAT algorithm

- Reimplies assigned literals at lower levels without backtracking
- Enables ILB
- Independently of ILB, restores the two core BCP invariants, broken by Chronological Backtracking (CB)

Trail implemented as doubly-linked list (rather than stack) to facilitate CB & reimplication

- No pointers, efficient array-based implementation

New heuristics (query-driven tuning, subsumption-based flipped clause filtering, incr. score reboot)

No heavy algorithms, such as, inprocessing and vivification