Where does SAT not work?

Armin Biere
Johannes Kepler University
Linz, Austria

Theoretical Foundations of Applied SAT Solving

BIRS

Banff International Research Station
for Mathematical Innovation and Discovery

Banff, Canada

Monday, 20 January, 2014

Overview

Even though SAT solving is quite successful in many application there are still open issues, where SAT does not work or at least we do not know how or why it works.

- learning definitions through extended resolution
- speeding up CDCL by local search effectively
- arithmetic reasoning on the CNF level
- data-flow based algorithms for SAT
- parallel SAT solving
- reencoding CNF: extensions and theoretical questions
- data structures for next generation SAT solvers
- how and why does VSIDS work?

Learning Definitions Through Extended Resolution

- extended resolution (ER) more powerful than resolution
  - "ER allows to simulate execution of any algorithm / circuit"
  - how to find *good definitions*?
  - simplest version just defines AND gate over two existing variables $n = l \wedge k$
  - other definitions are conceivable too $(\bar{l} \vee \bar{k} \vee n)(l \vee \bar{n})(k \vee \bar{n})$
- not much practical work on extended resolution
  - [SinzBiere'06] use extended resolution to simulate BDD operations
    original motivation was to simulate Gaussian elimination
  - [AudemardKatsirelosSimon'10] [Huang'10]
    introduce abbreviations for common parts in learned clauses
  - [MantheyHeuleBiere'12] reencode CNF through *bounded variable addition* (BVA)
- challenges for making ER work
  - learn more than abbreviations for learned clauses
  - more powerful reencodings for in/pre-processing
  - garbage collection of definitions

Speeding up CDCL by Local Search Effectively

- Balint showed (empirically) that CDCL can help local search
  - idea is to use CDCL solver as oracle inside local search
  - CDCL checks environment around current point
  - some improvements using this idea on random instances
- challenges in combining CDCL with local search
  - more synergies using CDCL for local search
  - effectively using local search in CDCL
  - is it possible to connect the two in a proof theoretic sense
- BTW, new generation of local search solver with substantial progress
  - *Sparrow*, *CCASat*, *ProbSAT*, ...
  - *ProbSAT* is elegant and simple and quite competitive
  - they solve some hard (satisfiable) "combinatorial" instances

Arithmetic Reasoning on the CNF Level

- bit-blasting $a \cdot b = b \cdot a$ with a, b 32-bit bit-vectors
  - assume bit-blasting without any word level rewriting (simplification)
  - produces *and inverter graph* (AIG) with 7277 nodes and 64 inputs
  - results in CNF with 7341 variables and 21654 clauses
  - after CNF level preprocessing 3355 variables and 15493 clauses
  - extremely hard for current state-of-the-art SAT solvers (working on CNF)
- related important practical problem
  - equivalence checking of arithmetic circuits
  - no intermediate equivalent literals
  - SAT sweeping even on AIG level does not help
    - see our recent LPAR-19 paper for SAT sweeping on the CNF level
  - some structural bit level techniques exists
    - still incomparable in speed to pattern based word level simplification
  - nothing on the CNF level

Data Flow Algorithms for SAT

- most paradigms for SAT solving are control-dominated:
  - such as variants of CDCL, WalkSAT, or Look-Ahead based algorithms
  - hard to port to highly parallel computing architectures like:
    - bit-parallel operations on streaming units (SSE, AVX ops with 128 bit - 256 bit)
    - multi-core systems with say 96 or even more cores
    - clusters / grid / clouds with 128 - 100000 cores
    - GPUs with more than 2000 cores
  - control flow dominated algorithms have a hard to time to achieve memory locality
- conjecture is that data-flow orientation allows memory locality
  - challenge is to come up with SAT algorithms organized around data-flow
  - find other ways to change algorithms / machines to become more “local”
- our (unpublished) experiences with bit-parallel SAT and GPU’s are rather negative
  - only focused on preprocessing sofar
  - positive effect for few crafted instances
  - usually way slower

- dominating approach: portfolio with clause sharing
  - *ManySAT*, *Plingeling*, *Penelope*, ...
  - successful in the application track of the competition
  - portfolio already gives substantial speed-up
  - clause sharing of “good” clauses gives another boost
- search space splitting
  - originally used on clusters / grids
    - guiding path principle [ZhangBonacinaHsiang’96]
    - revisited and extended recently [HyvärinenJunttilaNiemelä’10]
  - can be combined with look-ahead
    - Cube & Conquer approach [HeuleKullmannWieringaBiere’11]
    - works well on multi-core as well
    - Treengeling won parallel combinatorial track in SAT Competition 2013
- how to merge these two approaches?
- scalability for many cores and larger clusters / grids / cloud

Reencoding

- reencode CNF through *bounded variable addition* [MantheyHeuleBiere'12] (BVA)

  Replace $(a \vee d)\ (a \vee e)$ $(b \vee d)\ (b \vee e)$ $(c \vee d)\ (c \vee e)$ by $(\bar{x} \vee a)\ (\bar{x} \vee b)\ (\bar{x} \vee c)$ $(x \vee d)\ (x \vee e)$

- "reverse" of *bounded variable elimination* (BVE) [DP/Satelite]
  - BVE eliminates variable if number of clauses does not increase
  - BVA adds variable to decrease number of clauses
  - surprisingly simulates using better encodings for cardinality
    - particularly works well for at-most-one constraints constraints
  - we only know how to implement restricted variant **simple** BVA
- practical questions: inprocessing? other re-encodings? multiple variables?
- theoretical side: properties of this class(es) of re-encodings?

- one of Lingeling's main design goals: compactness of data structures
  - results in smaller memory foot print
  - smaller working set thus less pressure on the memory system
  - implicitly increases speed
- particularly useful if *multiple instances* of the SAT solver run on the same machine
  - allows more solver instances within the same amount of memory
  - cube & conquer (Treengeling)
  - incremental approaches with push & pop semantics
  - portfolio style SAT solvers (which copy clauses e.g. Plingeling)
- still in all these scenarios many copies of identical clauses exists
  - redundant copies = wasted memory
  - new management scheme to physically share these redundant copies
  - avoid congestion and too much synchronization overhead

VSIDS Decision Heuristic

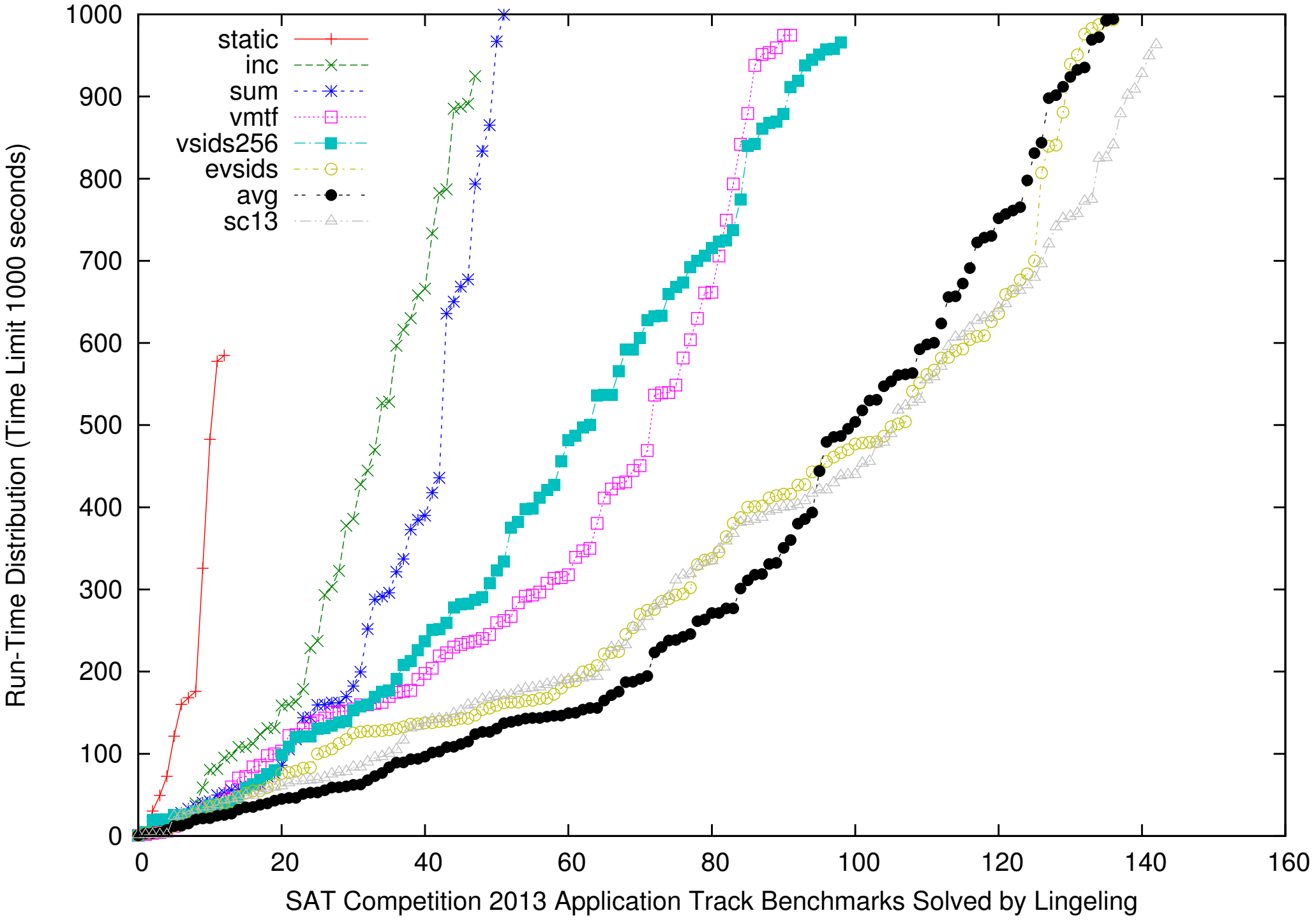
- original idea in *Chaff* [Moskewicz…'01] was to "bump" literals in learned clauses
  - *Variable State Independent Decaying Sum* (VSIDS)
  - originally simply incremented a counter/score (one per variable)
    same effect as DLIS in *Grasp*
  - every 256th conflict the counter divided by 2
    "filtering" is the novel part in Chaff, it adds state and focus
- state-of-the-art is the *Exponential VSIDS* (EVSIDS) scheme of MiniSAT
  - use a global increment g which is added to the score
  - increase this increment exponentially at every conflict (increase by say 5%)
  - implementation uses a priority queue with lazy removal of assignment variables
  - better bump all variables used in deriving the conflict
- alternative variants (in my experience less efficient)
  - *BerkMin* uses variables in most recent still unsatisfied learned clause
  - *Siege* used Variable Move To Front (VMTF) strategy
  - *HaifaSAT* combines both by a Clause Move To Front (VMTF) strategy
    same as in *PrecoSAT* but with separate queues for each glucose level (LBD)

How to Compute the Score?

- SAT solver picks unassigned variable with largest score as next decision
  - consider only change of the score s_i of one variable v during i-th conflict
  - let $\beta_i = 1$ if v is *bumped* in the i-th conflict otherwise 0
- some possible variable score update functions:
  - **static** $s_{i+1} = s_i$ initialize score statically and do not change it
  - **inc** $s_{i+1} = s_i + \beta_i$ this is in essence DLIS from Grasp
  - **vmtf** $s_{i+1} = i$
  - **sum** $s_{i+1} = s_i + i \cdot \beta_i$ emphasis on recent conflicts unpublished
  - **vsids** $s_{i+1} = d \cdot s_i + \beta_i$ decay $d \in [0,1)$ e.g. $d = 0.95$
  - **evsids** $s_{i+1} = s_i + g_i \cdot \beta_i, \quad g_{i+1} = e \cdot g_i$ factor $e \in [1,2)$ e.g. $e = 1.05$
  - **avg** $s_{i+1} = s_i + \beta_i \cdot (i - s_i)/2$ another filter function unpublished
- last four share the idea of “low-pass filtering” of the involvement of variables
  - for this interpretation see our SAT’08 paper and the video
  - important practical issue: number of bumped variables is usually small

Why does Bumping & Smoothing work?

- CDCL SAT solving is actually a very focused “local search”
  - focus on recently learned clauses
  - increases chance to learn clauses relevant to already learned clauses
  - this “relevance” increases the chance of “overlap” and small LBD
  - which in turn allows to find short proofs
- more speculation about “smoothing” or “low-pass filtering”
  - VMTF / BerkMin seems to be too aggressive more like a jumpy kid
  - filtering through the “decaying” part actually leads to *intensification*
- not aware of any work on **really** explaining either of them
  - except maybe for the insight behind glucose levels (LBD)
  - is there a way to formalize these intuitions?
  - do we need more empirical experiments?
  - maybe later this seminar . . .


static
inc
sum
vmtf
vsids256
evsids
avg
sc13
Run-Time Distribution (Time Limit 1000 seconds)
SAT Competition 2013 Application Track Benchmarks Solved by Lingeling
0
100
200
300
400
500
600
700
800
900
1000
0
20
40
60
80
100
120
140
160

Arithmetic Reasoning on the CNF Level

- bit-blasting $a \cdot b = b \cdot a$ with a, b 32-bit bit-vectors
  - assume bit-blasting without any word level rewriting (simplification)
  - produces *and inverter graph* (AIG) with 7277 nodes and 64 inputs
  - results in CNF with 7341 variables and 21654 clauses
  - after CNF level preprocessing 3355 variables and 15493 clauses
  - extremely hard for current state-of-the-art SAT solvers (working on CNF)
- related important practical problem
  - equivalence checking of arithmetic circuits
  - no intermediate equivalent literals
  - SAT sweeping even on AIG level does not help
    - see our recent LPAR-19 paper for SAT sweeping on the CNF level
  - some structural bit level techniques exists
    - still incomparable in speed to pattern based word level simplification
  - nothing on the CNF level

Data Flow Algorithms for SAT

- most paradigms for SAT solving are control-dominated:
  - such as variants of CDCL, WalkSAT, or Look-Ahead based algorithms
  - hard to port to highly parallel computing architectures like:
    - bit-parallel operations on streaming units (SSE, AVX ops with 128 bit - 256 bit)
    - multi-core systems with say 96 or even more cores
    - clusters / grid / clouds with 128 - 100000 cores
    - GPUs with more than 2000 cores
  - control flow dominated algorithms have a hard to time to achieve memory locality
- conjecture is that data-flow orientation allows memory locality
  - challenge is to come up with SAT algorithms organized around data-flow
  - find other ways to change algorithms / machines to become more "local"
- our (unpublished) experiences with bit-parallel SAT and GPU's are rather negative
  - only focused on preprocessing sofar
  - positive effect for few crafted instances
  - usually way slower

Reencoding

- reencode CNF through *bounded variable addition* [MantheyHeuleBiere'12] (BVA)

  Replace $(a \vee d)\ (a \vee e)\ (b \vee d)\ (b \vee e)\ (c \vee d)\ (c \vee e)$ by $(\bar{x} \vee a)\ (\bar{x} \vee b)\ (\bar{x} \vee c)\ (x \vee d)\ (x \vee e)$
- "reverse" of *bounded variable elimination* (BVE) [DP/Satelite]
  - BVE eliminates variable if number of clauses does not increase
  - BVA adds variable to decrease number of clauses
  - surprisingly simulates using better encodings for cardinality
    - particularly works well for at-most-one constraints constraints
  - we only know how to implement restricted variant **simple** BVA
- practical questions: inprocessing? other re-encodings? multiple variables?
- theoretical side: properties of this class(es) of re-encodings?

How to Compute the Score?

- SAT solver picks unassigned variable with largest score as next decision
  - consider only change of the score s_i of one variable v during i-th conflict
  - let $\beta_i = 1$ if v is *bumped* in the i-th conflict otherwise 0
- some possible variable score update functions:
  - **static** $s_{i+1} = s_i$ initialize score statically and do not change it
  - **inc** $s_{i+1} = s_i + \beta_i$ this is in essence DLIS from Grasp
  - **vmtf** $s_{i+1} = i$
  - **sum** $s_{i+1} = s_i + i \cdot \beta_i$ emphasis on recent conflicts unpublished
  - **vsids** $s_{i+1} = d \cdot s_i + \beta_i$ decay $d \in [0,1)$ e.g. $d = 0.95$
  - **evsids** $s_{i+1} = s_i + g_i \cdot \beta_i$, $g_{i+1} = e \cdot g_i$ factor $e \in [1,2)$ e.g. $e = 1.05$
  - **avg** $s_{i+1} = s_i + \beta_i \cdot (i - s_i)/2$ another filter function unpublished
- last four share the idea of "low-pass filtering" of the involvement of variables
  - for this interpretation see our SAT'08 paper and the video
  - important practical issue: number of bumped variables is usually small