# Evaluating CDCL Variable Scoring Schemes

Armin Biere  Andreas Fröhlich

Johannes Kepler University, Linz, Austria

18th International Conference on

Theory and Applications of Satisfiability Testing

SAT'15

The University of Texas at Austin
Austin, TX, USA

Wednesday, 24th September, 2015

# Motivation



- VSIDS/EVSIDS important
  - empirically
  - no formal argument “why it works”
- reconsider simpler alternatives
  - particularly variable move to front schemes (VMTF)
  - requires careful data structure design
  - formalization of these heuristics
  - empirical evaluation
- so a step towards “Trying to Understand the Power of VSIDS”

# Decision Heuristics



- decision heuristics consist of
  - variable selection: which variable to assign next?
  - phase selection: assign variable to which phase (true or false)?
- *phase saving* [PipatsrisawatDarwiche'07]
  - select phase to which variable was assigned before
  - initialized by static one-side heuristics [JeroslowWang'90]
  - very effective and thus default in state-of-the-art solvers
- we consider only *variable selection* as decision heuristic for this study
  - clause based heuristics less effective (BerkMin, CMTF)
  - same applies to literal based heuristics (using literal scores)
- variable selection and **decision heuristic** boils down to
  - compute and maintain heuristic scores for variables
  - **select variable with highest score**

# Scores in CDCL Loop



```
for (;;) {
  if (bcp ()) {          // [X] reorder assigned variables
                         // eagerly is considered too costly
    if (restarting ())
      restart ();        // [E] reuse trail if possible thus
                         // need to know next decision variable
                         // [C] unassigning variables might
                         // lead to reordering to support (B)

    if (!decide ())      // [B] find next decision variable
      return SAT;        // (unassigned one with highest score)
  } else {
    if (!analyze ())     // [A] learn clause from conflict and bump
      return UNSAT;      // relevant variables  (increase score)
  }                      // [D] optionally rescore all variables
}                        // (explicitly or to avoid overflow)
                         // [C] backtrack and unassign variables
```

# Variable Scoring Schemes



- how to compute scores
  - static or dynamic
  - **bump** variables: when to *increase* scores and by how much
  - **rescore** variables: when to *decrease* scores and by how much
  - state-of-the-art: VSIDS (from Chaff)

    more precisely the exponential variant (EVSIDS) of MiniSAT

- data structures for finding decision variables
  - eager or lazy update of “order”
  - state-of-the-art: priority queue of variables ordered by score (MiniSAT)

- data structure depends on how scores are computed and vice versa

# Zero, First and Second Order Variable Scoring Schemes

- *zero* order scheme $=$ static scores
  - computed for instance once during preprocessing
  - still needs search for “best” unassinged variable
  - only total orders considered so far
- *first* order schemes $=$ dynamic but state less
  - for instance: score $=$ pos occs $\times$ neg occs
  - independent of how search reached current branch / search node
  - might be quite expensive to compute / update (linear in CNF size)
- *second* order schemes: variable score depends on history of search
  - first order $+$ learning $\Rightarrow$ second order
  - but can also be used to speed up search for “best” variable
  - goal is logarithmic or even constant algorithm for variable selection

# VSIDS



- VSIDS appeared in seminal Chaff paper from Princeton (2001)
- bump variables occurring in learned clauses
  - bumping means incrementing an integer VSIDS score

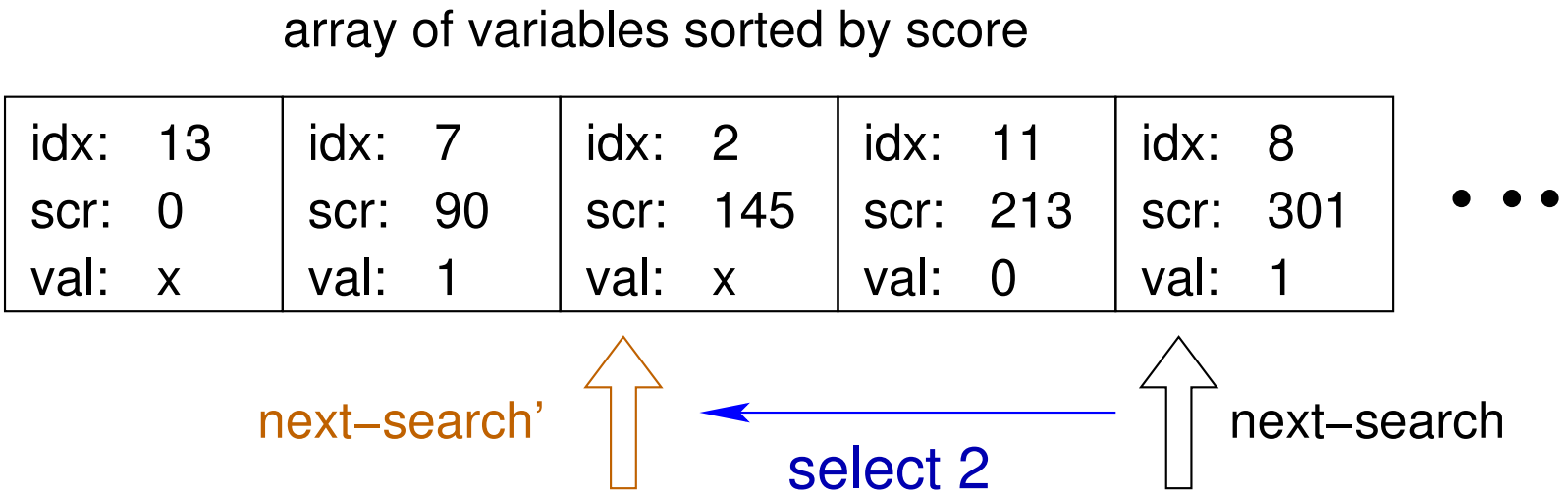
    current state-of-the-art: bump *all* variables used to derive learned clause
  - thus independent of state of clauses (satisfied or not)
- rescoring gives focus on recently used variables
  - scores are “decayed”, e.g., originally *divided by two* every 256th conflict
  - “low pass filter” on “use frequency” of variables

    `http://youtu.be/MOjhFywLre8`
- search for next unassigned variable with largest score
  - keeps an array of variables sorted by score
  - only re-sorts it w.r.t. score during rescoring (every 256th conflict)
  - uses right-most unassigned variable, thus original implementation **imprecise**

# ZChaff's Optimized Search for Next Decision Variable

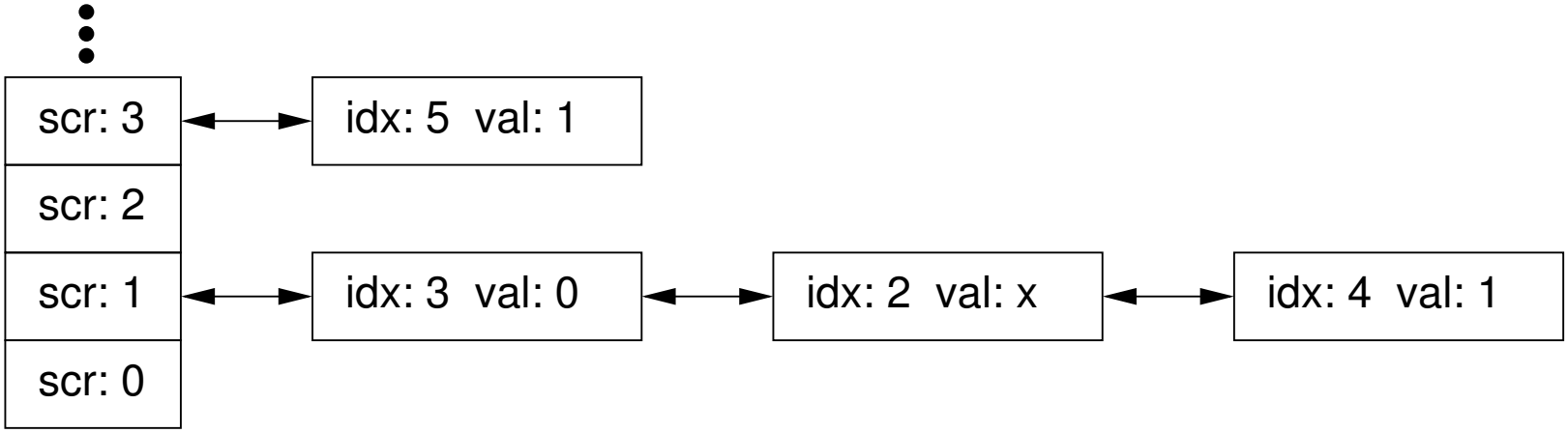


ZChaff's implementation actually sorted the other way around, largest score first




- basically start search at largest score variable
  - scan array towards smaller positions with smaller score
  - until *unassigned* variable is found, e.g., with idx 2 in the example
  - easily leads to accumulated quadratic search cost (for decisions)
- avoided by "caching" position of last found decision variable in "next-search"
  - next search can start from there
  - if variable right to next-search becomes assigned move next-search
  - no update of next-search during bumping (in ZChaff)

- original ZChaff implementation imprecise in three aspects
  - integer scores
  - effect of bumping not immediate
  - selected unassigned variable not necessarily the one with largest score
- JeruSAT [Nadel'02] partial fix was to
  - array of doubly linked lists of variables with same score
  - was still imprecise since it still used integer scores
  - costly eager update of data structure during (un)assigning, bumping and rescoring



# Variable Move To Front



- Siege SAT solver [Ryan'04] used *variable move to front* (VMTF)
  - bumped variables moved to head of *doubly linked list*
  - search for unassigned variable starts at head
  - variable selection is an online sorting algorithm of scores
  - classic "move-to-front" strategy achieves good amortized complexity
- original implementation severely restricted
  - only moved a subset of bumped variables
  - details about caching the search not described
    no source code published either
  - not exactly the same as VSIDS
- as consequence VMTF not used in state-of-the-art solvers

# MiniSAT's Exponential VSIDS (EVSIDS) from 2004



- floating point scores
  - allows fine grained rescore at every conflict
  - consider multiplying by $f = 0.9$ every score at each conflict
- actually, instead of updating scores of all variables (at every conflict)
  - only increase score of bumped variables by $g^i$
  - with $i$ the *conflict-index*, and $g = 1/f$
  - non-bumped variables not touched
- priority queue of variables ordered by score
  - implemented as binary heap with update (bubble up)
  - lazy assigned variable removal
    - remove largest score variable from heap until unassigned one found
    - put unassigned variables not on the heap back (logarithmic complexity)
- normalized VSIDS (NVSIDS) $\in [0, 1]$ as (theoretical) model [Biere'08]

# Summary Variable Scoring Schemes

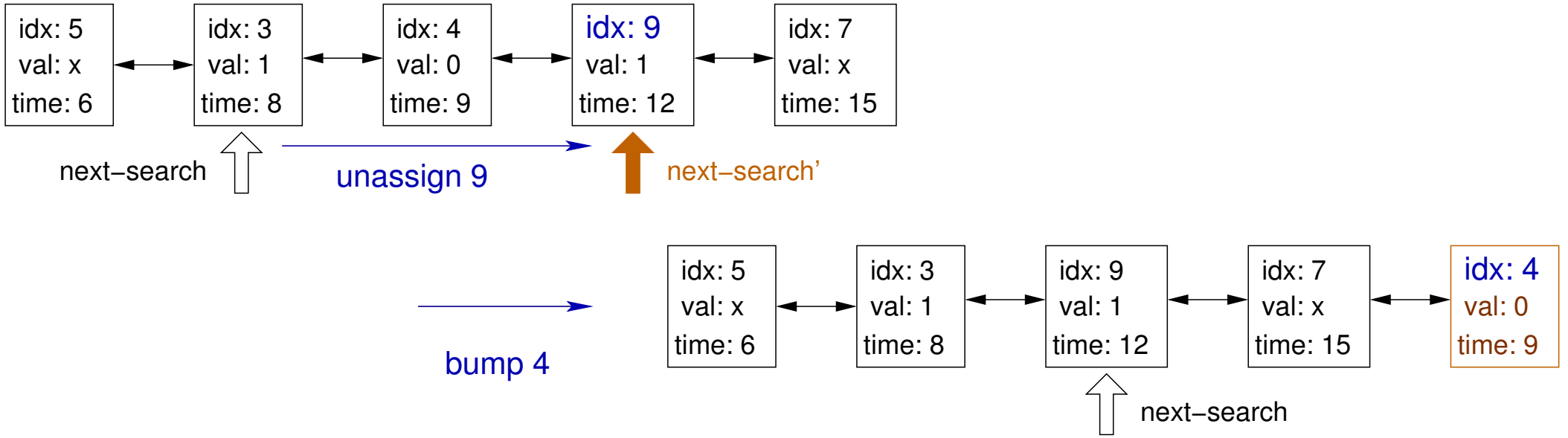
$s$ old score $s'$ new score

| | variable score $s'$ after $i$ conflicts | | |
|---|---|---|---|
| | bumped | not-bumped | |
| STATIC | $s$ | $s$ | static decision order |
| INC | $s+1$ | $s$ | increment scores |
| **SUM**[new] | $s+i$ | $s$ | sum of conflict-indices |
| VSIDS | $h_i^{256} \cdot s + 1$ | $h_i^{256} \cdot s$ | original implementation in Chaff |
| NVSIDS | $f \cdot s + (1-f)$ | $f \cdot s$ | normalized variant of VSIDS |
| EVSIDS | $s+g^i$ | $s$ | exponential MiniSAT dual of NVSIDS |
| **ACIDS**[new] | $(s+i)/2$ | $s$ | average conflict-index decision scheme |
| VMTF | $i$ | $s$ | variable move-to-front |

$0 < f < 1$ $\quad g = 1/f,$ $\quad h_i^m = 0.5$ if $m$ divides $i$, $\quad h_i^m = 1$ otherwise

# Fast VMTF Implementation

- fast simple implementation for caching searches in VMTF
  - doubly linked list does not have positions as an ordered array
  - bump = move-to-front = *dequeue* then *insertion* at the head
- time-stamp list entries with “insertion-time”
  - insertion-time same role as variable positions in ZChaff
  - maintained invariant: all variables right of next-search are assigned
  - requires (constant time) update to next-search while unassigning variables
  - occassionally (32-bit) time-stamps will overflow: update all time stamps

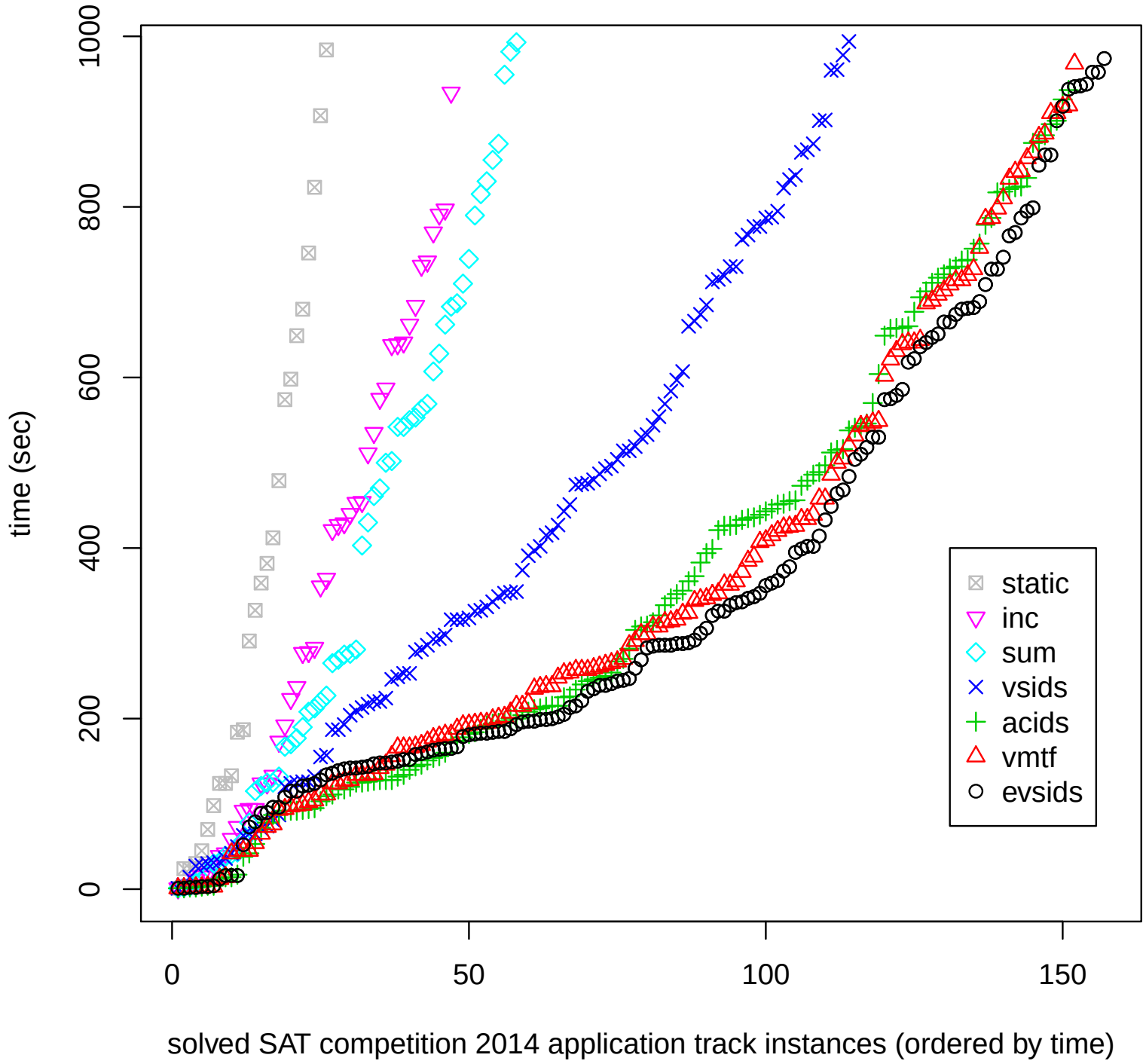


# Fast Generic Queue Implementation

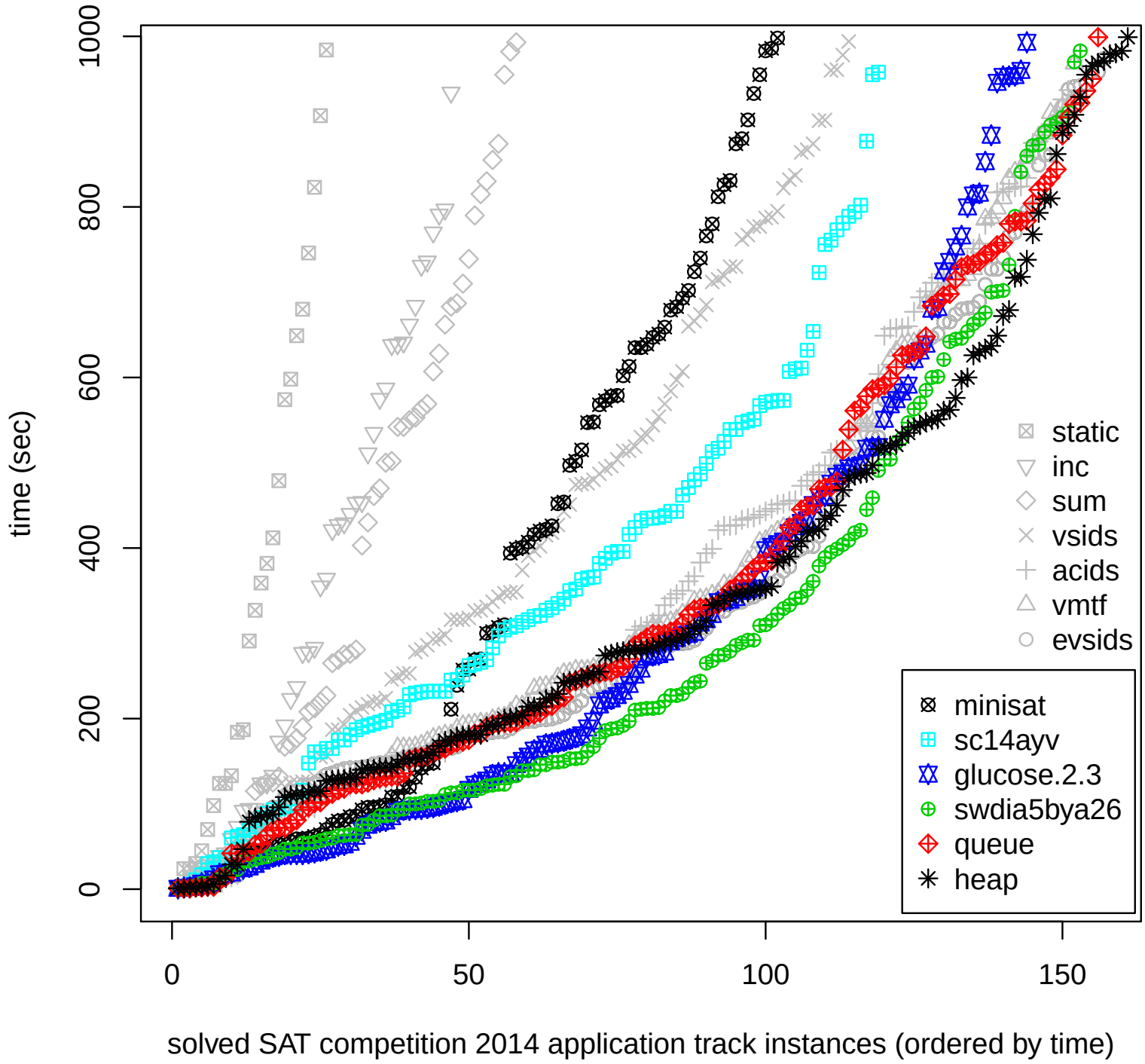


- one implementation working reasonably well for all schemes
- our preliminary results presented at BANFF'14 suggested VMTF not working
  - profiling revealed updating binary heap as a bottle neck
  - without time-stamping VMTF produces too much overhead
- array of variables split into doubly linked list of variables with same score **exponent**
  - similar to JeruSAT but for floating points
  - can be considered as a 1st level of **radix** / **bucket sort**
  - exponent-next-search position as in ZChaff
- each doubly linked list sorted by **mantissa** of scores
  - mantissa-next-search and insertion-times as in our fast VMTF implementation
  - key optimization is to add another **radix** / **bucket sort** level

    based on some bits of the mantissa
  - see paper for more optimization to avoid otherwise pathological cases
  - empirically not a bottle-neck anymore


time (sec)
0
200
400
600
800
1000
0
50
100
150
solved SAT competition 2014 application track instances (ordered by time)
static
inc
sum
vsids
acids
vmtf
evsids

| | evsids | vmtf | acids | vsids | sum | inc | static |
|---|---|---|---|---|---|---|---|
| solved | 157 | 152 | 151 | 114 | 58 | 47 | 26 |
| unsatisfiable | 87 | 85 | 82 | 51 | 22 | 17 | 9 |
| satisfiable | 70 | 67 | 69 | 63 | 36 | 30 | 17 |
| reductions (1e3 #) | 8 | 8 | 8 | 10 | 8 | 8 | 8 |
| restarts (1e3 #) | 5826 | 6000 | 5678 | 4491 | 2612 | 2387 | 5593 |
| rescored (1e3 #) | 253 | 0 | 0 | 2338 | 0 | 0 | 0 |
| conflicts (1e6 #) | 488 | 476 | 444 | 604 | 527 | 540 | 463 |
| decisions (1e6 #) | 3691 | 3581 | 3889 | 4263 | 2603 | 2567 | 21503 |
| simp (1e3 sec) | 29.7 | 30.0 | 29.4 | 32.6 | 34.5 | 34.1 | 31.2 |
| search (1e3 sec) | 143.1 | 146.4 | 147.9 | 174.9 | 203.9 | 209.7 | 226.7 |
| bump (1e3 sec) | 7.8 | 6.2 | 16.0 | 16.9 | 34.6 | 37.2 | 0.0 |
| decide (1e3 sec) | 2.3 | 2.5 | 2.6 | 2.8 | 1.7 | 1.7 | 12.9 |
| rescore (1e3 sec) | 0.2 | 0.0 | 0.0 | 2.6 | 0.0 | 0.0 | 0.0 |


time (sec)
0
200
400
600
800
1000
0
50
100
150
solved SAT competition 2014 application track instances (ordered by time)
static
inc
sum
vsids
acids
vmtf
evsids
minisat
sc14ayv
glucose.2.3
swdia5bya26
queue
heap

| | heap | queue | swd ia5by a26 | glu cose 2.3 | sc14 ayv | mini sat |
|---|---|---|---|---|---|---|
| solved | 161 | 156 | 153 | 144 | 119 | 101 |
| unsatisfiable | 90 | 86 | 81 | 79 | 60 | 41 |
| satisfiable | 71 | 70 | 72 | 65 | 59 | 60 |
| reductions (1e3 #) | 8 | 8 | 59 | 10 | 30 | — |
| restarts (1e3 #) | 5870 | 6003 | 3210 | 3846 | 7948 | 1782 |
| rescored (1e3 #) | 241 | 0 | — | — | 393 | — |
| conflicts (1e6 #) | 463 | 474 | 650 | 728 | 760 | 1090 |
| decisions (1e6 #) | 3874 | 3566 | 5868 | 6818 | 5002 | 8388 |
| simp (1e3 sec) | 29.2 | 29.7 | 0.8 | 0.8 | 32.4 | 2.2 |
| search (1e3 sec) | 141.8 | 144.6 | 165.4 | 172.5 | 164.4 | 206.5 |
| bump (1e3 sec) | 3.8 | 4.9 | — | — | 3.3 | — |
| decide (1e3 sec) | 4.9 | 2.5 | — | — | 6.4 | — |
| rescore (1e3 sec) | 0.1 | 0.0 | — | — | 0.0 | — |

# Conclusion



- surveyed and classified variable selection / scoring schemes
  - and came up with a new one ACIDS (as well as SUM)
  - EVSIDS, VMTF, ACIDS comparable in performance
  - with a generic fast queue implementation
- VMTF was considered to be obsolete
  - can be made effective (with less code than EVSIDS)
  - needs proper optimized implementation: time-stamping with insertion-time
  - VMTF might be easier to reason about in proof complexity
- threads to validity
  - unclear whether VMTF only works in combination with Glucose restarts
    see also our POS'15 paper and talk
  - benchmark selection in recent SAT competitions controversial
- future work: implement simple SAT solver only based on VMTF