Performance of Clause Selection Heuristics for Saturation-Based Theorem Proving

Stephan Schulz and Martin Möhrmann

DHBW Stuttgart, Germany
schulz@eprover.org, moehrmann@eprover.org

Abstract. We analyze the performance of various clause selection heuristics for saturating first-order theorem provers. These heuristics include elementary first-in/fist-out and symbol counting, but also interleaved heuristics and a complex heuristic with goal-directed components.
We can both confirm and dispel some parts of developer folklore. Key results include: (1) Simple symbol counting heuristics beat first-in/first-out, but by a surprisingly narrow margin. (2) Proofs are typically small, not only compared to all generated clauses, but also compared to the number of selected and processed clauses. In particular, only a small number of *given clauses* (clauses selected for processing) contribute to any given proof. However, the results are extremely diverse and there are extreme outliers. (3) Interleaving selection of the given clause according to different clause evaluation heuristics not only beats the individual elementary heuristics, but also their union - i.e. it shows a synergy not achieved by simple strategy scheduling. (4) Heuristics showing better performance typically achieve a higher ratio of *given-clause* utilization, but even a fairly small improvement leads to better outcomes. There seems to be a huge potential for further progress.

1 Introduction

Saturating theorem provers for first-order logic try to show the unsatisfiability of a clause set by systematically enumerating direct consequences and adding them to the clause set, until either no new (non-redundant) clauses can be generated, or the empty clause as an explicit witness of inconsistency is found.

At this time, the most powerful provers for first-order logic with equality are based on saturation. These provers implement saturation by variants of the *given-clause algorithm*. In this algorithm, clauses are selected for inferences one at a time. The order of selection of the *given clause* for each iteration of the main loop is a major choice point in the algorithm. While there is significant folklore about this choice point, we are not aware of a systematic evaluation of different heuristics for this choice point.

There is also little understanding of the properties of proofs and the proof induced by different strategies. Previous work was restricted to unit-equational logic and much smaller search spaces [4].

In this paper, we compare different classical and modern clause selection heuristics. In particular, we consider the following questions:

- How powerful are different heuristics on different classes of problems?
- How well do different heuristics perform compared to a perfect oracle that finds the same proofs? Which proportion of selected clauses is contributing to a given proof?
- How do different heuristics interact when interleaved?
- Can commonly held beliefs about clause selection be supported by data?
- What are typical properties of proofs found by a modern theorem prover?

To obtain data on these questions, we have instrumented the prover E to efficiently collect data about the ongoing proof search and to print out an analysis of both the proof object and the complete proof search graph at termination.

This paper is organized as follows. First, we introduce the concept of saturation and briefly describe the given-clause algorithm. We also discuss the basics of clause evaluation and E's flexible implementation of clause selection heuristics. In section 3 we describe the design of the experiments and the particular clause selection heuristics analyzed. Section 4 contains results on the performance of different heuristics and their analysis, as well as information on properties and structures of proofs and proof search. We then conclude the paper.

2 Saturating Theorem Proving

Modern saturating theorem proving started with resolution [17]. It was also a natural framework for completion-based equational reasoning [7, 6, 1]. The confluence of resolution and completion, implemented e.g. in Otter [12, 13], the first modern-style high-performance theorem prover, lead to the the still current equality-based superposition calculus, definitively described by Bachmair and Ganzinger [2]. Today, systems based on superposition and saturation like Vampire [15, 8], Prover9 [11], SPASS [23] and E [19, 20] define the state of the art in theorem proving for first-order logic with equality.

Saturating calculi for first-order logic are based on a refutational paradigm, i.e. the axioms and conjecture are converted into a clause set that is unsatisfiable if and only if the conjecture is logically implied by the axioms. The calculus defines a series of inference rules which take one or more (most often two) existing clauses as premises and produce a new clause as the conclusion. This new clause is added to the original clause set and is available as a premise for future inferences. The process stops when either no new non-redundant clause can be derived (in this case, the clause set is *saturated* up to redundancy), or when the empty clause as an explicit witness of unsatisfiability is derived.

Current calculi also include simplification rules which allow the replacement of some clauses by simpler (and often syntactically smaller) clauses, or even the complete removal of redundant clauses. Examples include in particular *rewriting* (replacement of terms by smaller terms), subsumption (discarding of a clause implied by a more general clause) and tautology deletion.

In most cases, saturation can, in principle, derive an infinite number of consequences. In these cases, *completeness* of the proof search requires a certain notion of *fairness*, namely that no non-redundant inference is delayed infinitely.

Search state: (U, P)
U contains *unprocessed* clauses, P contains *processed* clauses.
Initially, P is empty and all clauses are in U.
The *given clause* is denoted by g.

```
while U ≠ {}
  g = extract_best(U)
  g = simplify(g, P)
  if g == □
    SUCCESS, Proof found
  if g is not subsumed by any clause in P (or otherwise redundant w.r.t. P)
    P = P\{c ∈ P | c subsumed by (or otherwise redundant w.r.t.) g}
    T = {c ∈ P | c can be simplified with g}
    P = (P\T) ∪ {g}
    T = T ∪ generate(g, P)
    foreach c ∈ T
      c = cheap_simplify(c, P)
      if c is not trivial
        U = U ∪ {c}
SUCCESS, original U is satisfiable
```

Remarks: extract_best(U) finds and extracts the clause with the best heuristic evaluation from U. This is the choice point we are particularly interested in this paper.

Fig. 1: The *given-clause* algorithm as implemented in E

The superposition calculus is the current state of the art in saturating theorem proving. It subsumes earlier calculi like resolution, paramodulation, and unfailing completion. In the superposition calculus, inferences can be restricted to maximal terms of maximal literals using a *term ordering*, and optionally to selected negative literals using a *literal selection* scheme. All systems we are currently aware of determine a fixed term ordering and literal selection scheme before saturation starts, either by user input or automatically after analyzing the problem.

2.1 Saturation Algorithms

Saturation algorithms handle the problem of organizing the search through the space of all possible derivations. The simplest and obviously fair algorithm is *level saturation*. Given a clause set C_0, level saturation computes the set of all direct consequences D_0 of clauses in C_0. The union $C_1 = C_0 \cup D_0$ then forms the basis for the next iteration of the algorithm. Level saturation does not support heuristic guidance, and we are not aware of any current or competitive system built on the basis of level saturation. To our knowledge level saturation has never been implemented with modern redundancy elimination techniques.

At the other extreme, a *single step* algorithm performs just one inference at a time, adding the consequence to the set and making it available for further

inferences (and potential simplification). The major disadvantage of the single-step algorithm is the book-keeping necessary. Moreover, while search heuristics can work at the finest possible granularity, the objects of heuristic evaluations are potential inferences, not concrete clauses. We are not aware of any system that uses a per-inference evaluation for search guidance, although e.g. Vampire's *limited resource strategy* [16] discards some potential inferences up-front, based on a very cursory evaluation.

The most widely used saturation algorithms are variants of the *given-clause* algorithm. They split the set of all clauses into two subsets U of *unprocessed* clauses and P of *processed* clause (initially empty). In each iteration, the algorithm selects one clause g from U and adds it to P, computing all inferences in which g is at least one premise and all other premises are from P. The algorithm adds the resulting new clauses to U, maintaining the invariant that all inferences between clauses in P have been performed.

Variants of the *given-clause* algorithm are at the heart of most of today's saturating theorem provers. The two main variants are the so-called *Otter loop* and the *DISCOUNT loop*, popularized by the eponymous theorem provers [13, 3]. In the Otter loop, all clauses are used for simplification. In particular, newly generated clauses are used to back-simplify both processed and unprocessed clauses. In the DISCOUNT loop, unprocessed clauses are truly passive, i.e. only clauses that are selected for processing are used for back-simplification. As a result, the Otter loop can typically find proofs in less iterations of the main loop, but each iteration takes longer. In the DISCOUNT loop, contradictory clauses in U may not be discovered until selected for processing. However, each individual iteration of the main loop results in less work. In both variants, selection of the given clause is the main heuristic choice point. In the DISCOUNT loop this control is at a finer level of granularity, since each iteration of the main loop represents a smaller part of the proof search.

In addition to Otter, the Otter loop is implemented in Prover9, SPASS and Vampire. The DISCOUNT loop historically was implemented in systems specializing in equational reasoning, including Waldmeister [10] and E. It was also added as an alternative loop to both SPASS and Vampire. There is little evidence that one or the other variant has a systematic advantage. A comparison in Vampire [16] showed some advantage for the DISCOUNT loop over the plain Otter loop, but also some advantage of the Otter loop in combination with the *limited resource strategy* (which sacrifices completeness for efficiency by discarding some new clauses) over Vampire's DISCOUNT loop.

Fig. 1 depicts the DISCOUNT loop as implemented in E. The given-clause selection is represented by the `extract_best()` function.

2.2 Clause Selection Heuristics

Once term ordering and literal selection scheme are fixed, clause selection, i.e. the order of processing of the unprocessed clauses, is the main choice point. The standard implementation assigns a heuristic weight to each clause, and processes

clauses in ascending order of weight, i.e. at each iteration of the main loop the clause with the lowest weight is selected.

Most modern provers allow at least the interleaving of a best-first (lowest weight) and breadth-first (oldest clause) search, where the weight is usually based on (weighted) symbol counting. The ratio of clauses picked by size to clauses picked by age is also known as the *pick-given ratio* [12]. E generalizes this concept. It supports a large number of different parameterized clause evaluation functions and allows the user to specify an arbitrary number of priority queues and a weighted round-robin scheme that determines how many clauses are picked from each queue. This enables us to configure the prover to use nearly arbitrarily complex clause selection heuristics and makes it possible to simulate nearly any conventional clause selection heuristic.

In this study, we are, in particular, concerned with the properties of conventional clause selection schemes. Thus, we look at the following basic clause evaluation heuristics:

- *First-in/First-out* or *FIFO* clause selection always prefers the oldest unprocessed clause. In E, this is realized by giving each new clause a pseudo-evaluation based on a counter that is increased each time a new clause is generated. If one ignores simplification, a pure FIFO strategy will emulate *level saturation*, i.e. it will generate all clauses of a given level before clauses of the next level. In this case, it should find the shortest possible proof (by number of generating inferences). Integration of simplification complicates the issue, although we would still expect FIFO to find short proves. FIFO is an obviously fair heuristic.
- *Symbol counting* or *SC* clause evaluation counts the number of symbols in a clause, and prefers small clauses. Function symbols and variables can have uniform or different weights. There are several intuitive reasons why this should be a good strategy. On the most obvious level, the goal of the saturation is the derivation of the empty clause, which has zero symbols. Moreover, clauses with fewer symbols are more general, hence allowing the system to remove more redundancy via subsumption and rewriting. And finally, clauses with fewer symbols also have fewer positions, and hence likely fewer successors, keeping explosion of the search spaces lower than large clauses. As long as all symbols (or at least all function symbols with non-zero arity) have positive weight, *SC*-based strategies are fair (there are only finitely many different clauses below any given weight).
- *Ordering-aware* evaluation functions are symbol-counting variants that are designed to prefer clauses with few maximal terms and maximal literals. In the general case, this reduces the number of inference positions (and hence potential successors), decreasing the branching factor in the search space. In the unit-equational case it will also prefer orientable equations (*rules*) to unorientable equations. Rules are much cheaper to apply for simplification. In E, the *refined weight* (*RW*) heuristic achieves the desired effect by multiplying the weight of maximal terms and maximal literals by user-selectable constant factors.

- A major feature of E is the use of *goal-directed* evaluation functions (*GD*). These give a lower weight to symbols that occur in the conjecture, and a higher weight to other symbols, thus preferring clauses which are more likely to be applicable for inferences with the conjecture.

Most of our experiments look at simple heuristics employing only one or two clause evaluation functions - see the experimental design section. However, for comparison we also include the globally best clause selection heuristic for E known to us. This scheme was created via genetic algorithm from a population of random heuristics spanning the parameter space of manually created heuristics developed over the last 15 years [18].

In addition to clause selection based on the syntactic form of the clause, the system can also select clauses based on their origin. In particular, a common recommendation is to first process all the initial clauses, before any of the derived clauses is picked.

3 Experimental Design

We added the ability to efficiently record compact internal proof objects in E 1.8. The overhead for proof recording is minimal and barely measurable [20]. We have now slightly extended the internal representation of the proof search to be able to record all processed given clauses, thus enabling the prover to provide more detailed statistics on the quality of clause selection. Other statistics were obtained by analyzing the existing proof object, and by counting operations and inferences performed during the proof search. The code is part of E version 1.9.1 (pre-release) and will be included in the next release of the prover.

3.1 Computing Environment and Test Set

We used problems from the TPTP [22] library, version v6.3.0. Since we are interested in the performance of the heuristics for proof search, and since several of our statistical measures only make sense for proofs, we restricted the problem set to full first-order (FOF) and clause normal form (CNF) problems that should be provable, i.e. CNF problems with status *Unsatisfiable* or *Unknown*[1] and FOF problems with status *Unsatisfiable*, *ContradictoryAxioms*, or *Theorem*.[2]

This selection left 13774 problems, 7082 FOF and 6692 CNF problems. FOF problems were translated to CNF by E dynamically, with (usually short) translation time included in the reported times.

We report performance results separately for unit problems (all clauses are unit), Horn problems (all clauses are Horn and at least one clause is a non-unit Horn clause) and general (there is at least one non-Horn clause), with and

---

[1] Status *Unknown* is assigned to problems which should be provable, but for which no machine proof is known.

[2] Two trivial syntactic test examples were excluded. They tested floating point syntax features that at the time of the experiments were incorrectly handled by E.

Table 1: Clause selection heuristics used

Heuristic	Description
FIFO	First-in/First-out, i.e. oldest clause first
SC12	Symbol counting, function symbols have weight 2, variables have weight 1
SC11	Symbol counting, both function symbols and variables have weight 1
SC21	Symbol counting, function symbols have weight 2, variables have weight 1
RW212	Symbol counting, function symbols have weight 2, variables have weight 1, maximal terms receive double weight.
2SC11/FIFO	Interleaved selection: Select 2 out of every 3 clauses according to SC11, the remaining one with FIFO
5SC11/FIFO	Ditto, with a selection ration of 5:1. This is inspired by Larry Wos comment on Otter ("The optimal pick-given ratio is five")
10SC11/FIFO	Ditto, selection ratio 10:1
15SC11/FIFO	Ditto, selection ratio 15:1
GD	Individual goal-directed heuristic, extracted from *Evolved* below
5GD/FIFO	GD interleaved 5:1 with FIFO
SC11-PI	As SC11, but always process initial clauses first
10SC11/FIFO-PI	As 10SC11/FIFO, but always process initial clauses first
Evolved	Evolved heuristic, combining 2 goal-directed evaluation functions, two symbol-counting heuristics, and FIFO. See [18]

without equality. The classification of problems into these types refers to the clausified form and was performed by E after clausification.

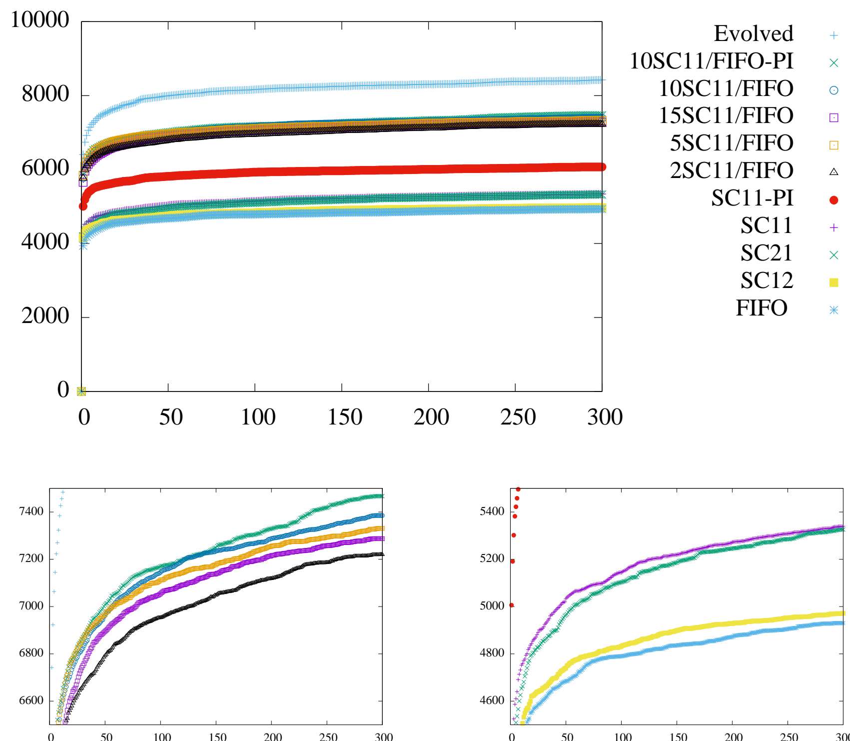
The StarExec Cluster [21] was used for all benchmark runs. Each problem was executed alone and single threaded on an Intel Xeon E5-2609 processor running at 2.4 GHz base clock speed. Each node had at least 128 Gigabyte RAM. We ran the experiments with a per-problem time limit of 300 seconds and, given the amount of RAM available, without enforced memory limit.

3.2 Claus Selection Heuristics

We tested 40 different clause selection heuristics. From these we selected the 14 heuristics described in Table 1 as sufficiently distinct and reasonably covering the parameter space we are interested in.

The 14 selected heuristics include basic FIFO and symbol counting, ordering-aware and goal-directed heuristics, as well as combinations of symbol counting variants with FIFO. We also tested the performance of a preference for initial clauses, and include the *Evolved* heuristic as a benchmark that represents the current state of the art.

The full data for all 40 strategies and the exact parameters for the provers are archived and available at `http://www.eprover.eu/E-eu/Heuristics.html`.



The large plot shows overall performance (vertical axis is number of proofs found up to a given time, horizontal axis is run time in seconds). The smaller plots scale interesting sections of the y-axis to differentiate strategies with similar overall performance.

Fig. 2: Solutions over time for different clause selection heuristics

4 Results

4.1 Global Search Performance

Table 2 summarizes the performance of the 14 different strategies on the full test set. The *Rank* column shows the ranking of strategies by total number of successes within the time limit. The third column shows the number of successes, as an absolute number and as a fraction of all 13774 problems. The next column shows how many problems were solved by the corresponding strategy only, not by any of the other strategies. Finally, the last column shows how many problems are solved within a 1 second search time, and the fraction of total successes by that strategy this number represents. Figure 2 visualizes the performance of a selected subset of strategies over time.

From this data, we can already draw a number of conclusions:

Table 2: Global search performance

Heuristic	Rank	Successes		Successes within 1s	
		total	unique	absolute	of column 3
FIFO	14	4930 (35.8%)	17	3941	79.9%
SC12	13	4972 (36.1%)	5	4155	83.6%
SC11	9	5340 (38.8%)	0	4285	80.2%
SC21	10	5326 (38.7%)	17	4194	78.7%
RW212	11	5254 (38.1%)	13	5764	79.8%
2SC11/FIFO	7	7220 (52.4%)	24	5846	79.7%
5SC11/FIFO	5	7331 (53.2%)	3	5781	78.3%
10SC11/FIFO	3	7385 (53.6%)	1	5656	77.6%
15SC11/FIFO	6	7287 (52.9%)	6	5006	82.5%
GD	12	4998 (36.3%)	12	5856	78.4%
5GD/FIFO	4	7379 (53.6%)	62	4213	80.2%
SC11-PI	8	6071 (44.1%)	13	4313	86.3%
10SC11/FIFO-PI	2	7467 (54.2%)	31	5934	80.4%
Evolved	1	8423 (61.2%)	593	6406	76.1%

- All performance curves are similar in basic shape, and all strategies find the bulk of their proofs within the first few seconds. Indeed, most strategies reach around 80% of their successes within the first second, and even for the *Evolved* strategy, more than three quarter of the successes are achieved within one second.
- *FIFO* is the weakest of the search strategies. However, even *SC11*, the best simple symbol counting heuristic, proves less than 10% more than *FIFO*.
- There is no evidence that using different weights for function symbols and variables increases overall performance. Indeed, using a higher weight for variables markedly decreases performance. However, it changes the part of the search space explored early, potentially adding more solutions to the performance of the ensemble of all strategies.
- The ordering-aware *RW212* has slightly lower global performance than the corresponding simple symbol-counting heuristics. This is surprising, since this and similar strategies have for a long time been major contributors to E's collection of standard heuristics.
- All four strategies interleaving simple symbol counting and FIFO perform much better than the corresponding pure symbol-counting strategy, with the best one solving more than 2000 extra problems, an increase of nearly 40%. On the other hand, the spread of performance over the *pick-given ratios* from 2 to 15 is very small, varying by only about 2%. The best ratio in our tests for E is not 5 as sometimes anecdotally reported for the Otter loop, but 10.
- For the union of solutions found by SC11 and FIFO (with 300 second time limit for each), the prover finds only 6329 proofs. Thus, there is real synergy in the interleaved strategies, which beat not only the individual components but also their union. We believe this is due to two effects: Symbol counting selection builds a compact representation of the theory induced by the ax-

Table 3: Number of problems solved in 300 seconds for different problem classes

Type	general		Horn		unit	
Equational	eq	non-eq	eq	non-eq	eq	non-eq
Heuristic/Size	(8626)	(1607)	(1011)	(1432)	(1037)	(61)
FIFO	2421 (28%)	907 (56%)	371 (37%)	835 (58%)	335 (32%)	61 (100%)
SC12	2160 (25%)	842 (52%)	432 (43%)	828 (58%)	649 (63%)	61 (100%)
SC11	2369 (27%)	918 (57%)	465 (46%)	853 (60%)	674 (65%)	61 (100%)
SC21	2410 (28%)	978 (61%)	428 (42%)	800 (56%)	649 (63%)	61 (100%)
RW212	2336 (27%)	972 (60%)	429 (42%)	800 (56%)	656 (63%)	61 (100%)
2SC11/FIFO	3809 (44%)	1199 (75%)	576 (57%)	953 (67%)	622 (60%)	61 (100%)
5SC11/FIFO	3798 (44%)	1200 (75%)	606 (60%)	983 (69%)	683 (66%)	61 (100%)
10SC11/FIFO	3803 (44%)	1192 (74%)	617 (61%)	989 (69%)	723 (70%)	61 (100%)
15SC11/FIFO	3732 (43%)	1187 (74%)	612 (61%)	967 (68%)	728 (70%)	61 (100%)
GD	2271 (26%)	819 (51%)	431 (43%)	821 (57%)	595 (57%)	61 (100%)
5GD/FIFO	3860 (45%)	1153 (72%)	606 (60%)	967 (68%)	732 (71%)	61 (100%)
SC11-PI	2894 (34%)	968 (60%)	523 (52%)	913 (64%)	712 (69%)	61 (100%)
10SC11/FIFO-PI	3929 (46%)	1142 (71%)	631 (62%)	986 (69%)	718 (69%)	61 (100%)
Evolved	4477 (52%)	1201 (75%)	712 (70%)	1204 (84%)	768 (74%)	61 (100%)

ioms, thus enabling the prover to traverse larger parts of search space, while FIFO ensures that no part of the search space is unduly delayed.

- The goal-directed heuristic on its own is not particularly powerful. Its performance is in line with the symbol-counting heuristics. However, it profits even more from the addition of a FIFO component than the other strategies.
- Processing initial clauses first does indeed boost performance of a strategy. However, the effect is much stronger for the pure symbol-counting heuristic than for a strategy that interleaves FIFO selection. The intuitive explanation is that FIFO selection will bring in all initial clauses relatively early anyways.
- The *Evolved* strategy significantly outperforms even the best other strategy.

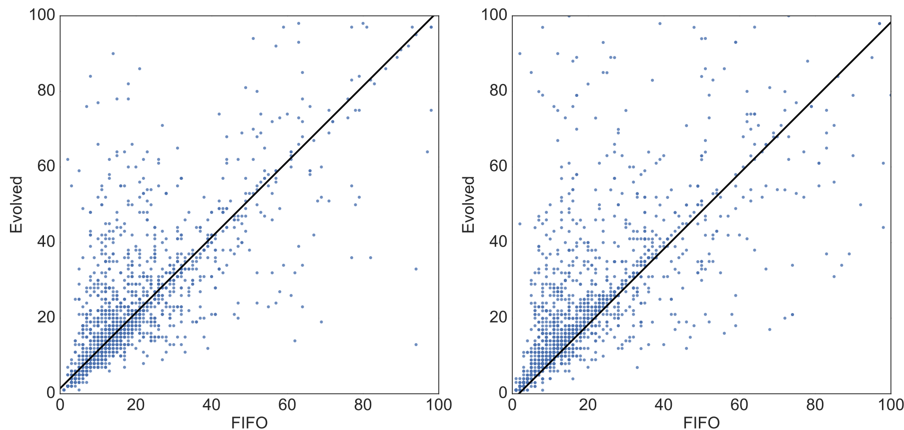
4.2 Search Performance by Problem Class

Table 3 breaks down the performance of the different heuristics by problem class. Interesting observations are in particular in the unit categories. First, all strategies solved all non-equational unit problems. This is not surprising, since this category is decidable and comprises only the task of finding one pair of complementary unifiable literals. In the unit-equality category, *FIFO* is comparatively much weaker than in the other categories. Likewise, *GD* is weak, but makes a strong showing in the combination with *FIFO*. Most of the results in the non-unit problems are in line with the general performance discussed in the previous section. We do notice that general (i.e. non-Horn) problems with equality are the hardest class for the tested strategies.

4.3 Proof Size and Structure

We are interested in the properties of proofs actually found by the prover. Particular properties we are interested in are:

- Is there a substantial difference between proofs found by different heuristics?
- How many of the initial clauses are used in the proof? I.e. what is the size of the unsatisfiable core of the axioms (and negated conjecture) that the prover found? In addition to the general interest, this value also provides important information for tuning pre-search axiom pruning techniques [9] like SInE [5] and MePo [14].
- How many inferences are in a typical proof, and how many search decisions contribute to it?



Proof size scatter plots. Each dot corresponds to one solved problem, with the size of the proof found by *Evolved* on the y-axis and the size of the *FIFO* proof on the X-axis. Proof size measure in the left is number of *given clauses* in the proof object, on the right it is total number of inferences in the proof. Both diagrams where cut off at 100 on each axis for better visibility. The left plot covers 93.8% and the right plot covers 90% of all data points. Only proofs where both strategies need at least 0.02 seconds are represented. The linear regression lines are $1.0x + 1.44$ for the left and $1.0x - 1.72$ for the right plot.

Fig. 3: Comparison of proof sizes

In principle, we would expect *FIFO* to find shorter proofs, since the underlying search is breadth-first. However, simplification may complicate this, and symbol-counting heuristics are likely to find more compact representations of the equational theory earlier, thus using fewer rewrite steps in normalization.

Figure 3 shows a comparison of the size of individual proof objects for proofs found with *FIFO* and *Evolved*, the two strategies with the widest difference in

Table 4: Number of clauses in proofs and proof searches

Heuristic	Proofs found	Given clauses in proof search Mean	Minimum	First quartile	Median	Third Quartile	Maximum
FIFO	4930	2302.5	1	28	157	875	209154
Evolved	8423	3598.7	1	38	188	1506	190309

Heuristic	Proofs found	Total clauses generated Mean	Minimum	First quartile	Median	Third Quartile	Maximum
FIFO	4930	342422.4	0	28	582	16951	21822536
Evolved	8423	356893.0	0	37	1023	38327	26187659

performance. While relative proof sizes are distributed over the whole diagram, there is a distinct increase in density towards the diagonal, and the computed regression is very close to the diagonal indeed. On average, *FIFO* proofs have slightly smaller number of given clauses, in line with our expectations. *Evolved* proofs have slightly fewer inferences. The difference is indeed due to the number of simplification steps. However, neither effect is very strong, and on average the proofs found by both heuristics seem to be of very similar sizes.

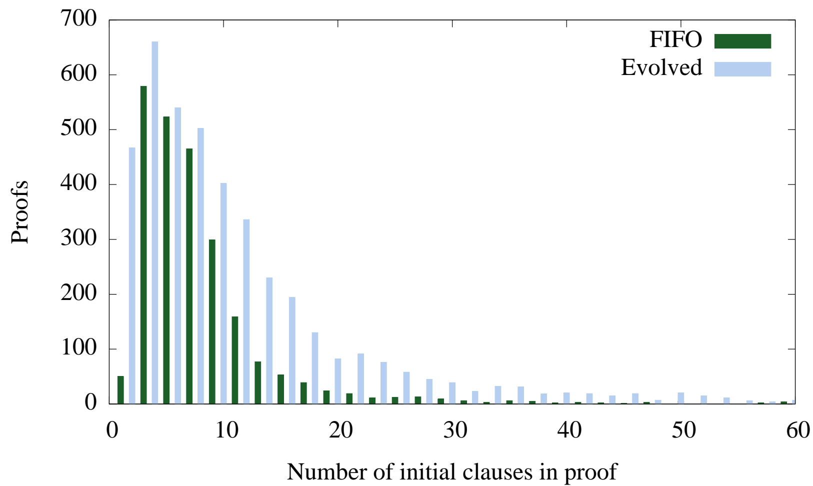
Figure 4 (top) shows the distribution of the number of initial clauses in proof objects. On average, there are 12.7 clauses in a non-trivial *FIFO* proof, and nearly 50% more initial clauses in an *Evolved* proof. Note that this statistic is based on all proofs found by either strategy, not on the subset of problems solved by both strategies. The bulk of the weight of the distribution is towards small numbers of initial axioms, with the mean very much influenced by a small number of combinatorial problems that need over 1000 clauses.

A similar observation holds for the actual proof size as shown in Figure 4 (bottom). By median, *Evolved* proofs are nearly twice as large than *FIFO* proofs, and at the third quartile, *Evolved* proofs are nearly 3 times as long as *FIFO* proofs. Thus, quite a lot of non-trivial proofs can be found. The mean proof size is again strongly influenced by a small number of combinatorial problems that require nearly a million inferences.

4.4 Proof Search Statistics and Performance

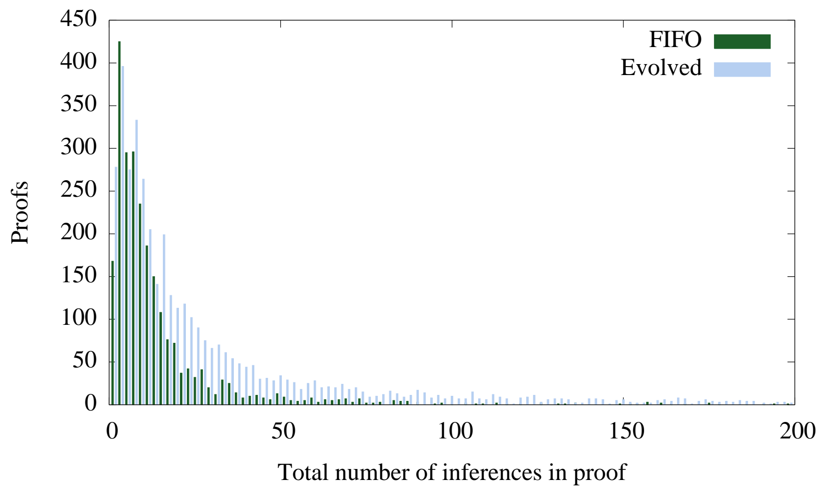
Table 4 shows the size of the search space constructed and traversed during the proof search. Comparing this with Fig. 4, we see that the number of *given clauses* actually processed to find a proof is orders of magnitude greater than the number of such clauses in the proof object. However, we also see that the number of clauses generated is again much larger, i.e. for non-trivial proofs many clauses derived by the inference engine are never processed.

An interesting measure is the fraction of processed *given clauses* that end up in the proof object, i.e. that represent good search decisions that contributed to the proof. We have plotted this *given-clause utilization* in Figure 5 (comparing different heuristics pairwise) and in Figure 6 (showing the distribution of the



Heuristic	Proofs found	Initial clauses in proof					
		Mean	Minimum	First quartile	Median	Third quartile	Maximum
FIFO	4930	12.7	1	4	6	9	1330
Evolved	8423	18.3	1	5	9	15	1330

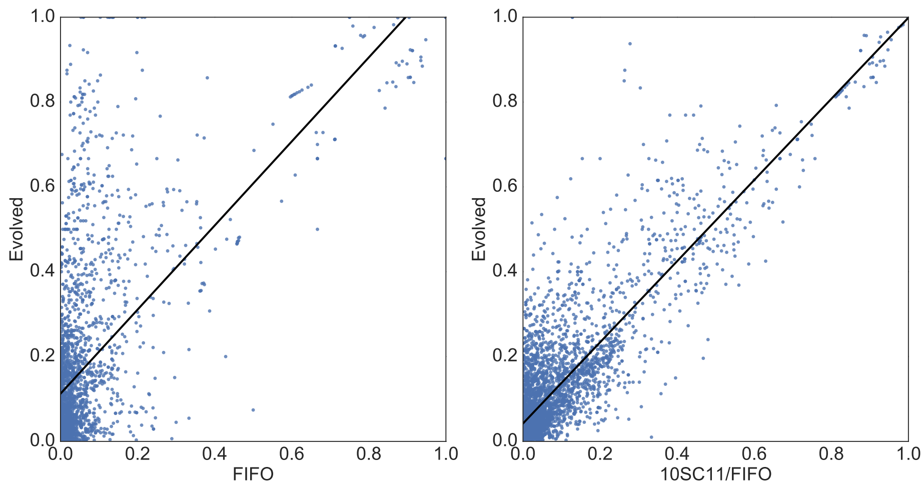
For *FIFO*, there are 104 proofs with more than 60 initial clauses in the proof object, i.e. the diagram covers 97.9% of all proofs. For *Evolved* there are 326 proofs with more than 60 initial clauses, i.e. the diagram covers 96.1% of all proofs.



Heuristic	Proofs found	Given clauses in proof					
		Mean	Minimum	First quartile	Median	Third Quartile	Maximum
FIFO	4930	784.8	1	4	9	17	933822
Evolved	8423	587.2	1	7	17	49	933819

There are 171 *FIFO* proofs with more than 200 inferences, i.e. the diagram covers 96.5% of all proofs. There are 658 *Evolved* proofs with more than 200 inferences, i.e. the diagram covers 92.2% of all proofs.

Fig. 4: Distribution of the number of initial clauses and inferences in proofs



Given-clause utilization rate scatter plot. The vertical axis shows the given-clause utilization for *Evolved*, the horizontal axis for *FIFO* (left) and *10SC11/FIFO* (right). Only proofs where both strategies need at least 0.02 seconds are represented. The linear regression lines are $0.992x + 0.111$ for the left and $0.957x + 0.043$ for the right plot.

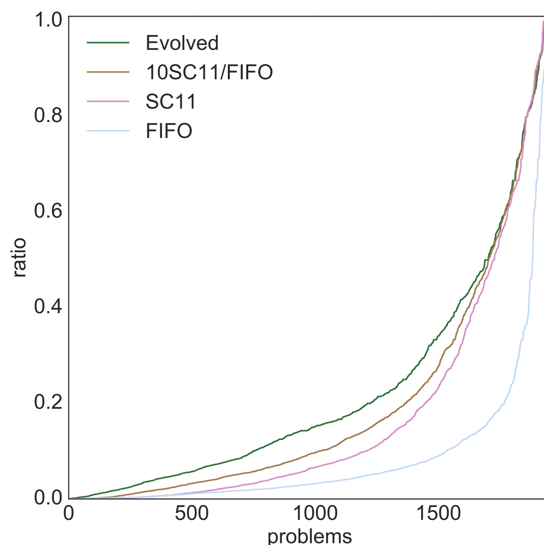
Fig. 5: Comparison of given-clause utilization ratios

ratio over the set of problems solved by four representative strategies). In both diagrams it is clear that the *given-clause utilization* is, on average, quite low. Also, Figure 6 strongly suggests that given-clause utilization is a good predictor for overall performance, with stronger strategies showing significantly better ratios.

5 Conclusion

Our analysis has shown the comparative performance of several classical and modern clause selection heuristics. We can confirm that interleaving symbol-counting and FIFO selection shows significantly better performance than either individually. We also found that preferring initial clauses is, on average, a significant advantage, and that goal-directed heuristics seem to work best in combination with other heuristics.

Proofs found by different heuristics for the same problem seem to be similar in size and complexity, however, stronger heuristics are able to find longer and more complex proofs. The average *given-clause utilization* as a measure of the quality of search decisions seems to correlate well with performance. It also shows us that even the best heuristics are far from optimal, or, to state it positively, that there still is a lot of room for improvement.



Given clause utilization rate for problems solved by four different strategies (only proofs for problems that are solved by all four strategies and where each needs at least 0.02 seconds are considered). The graph shows how many problems are solved with a given-clause utilization no better than the value on the vertical axis.

Fig. 6: Given clause utilization ratios over problem set

An open question is how far these results can be transferred to provers which employ the Otter loop, which places more priority to immediate simplification.

Acknowledgements We thank the StarExec [21] team for providing the community infrastructure making these experiments possible.

References

1. Bachmair, L., Dershowitz, N., Plaisted, D.: Completion Without Failure. In: Ait-Kaci, H., Nivat, M. (eds.) Resolution of Equations in Algebraic Structures. vol. 2, pp. 1–30. Academic Press (1989)
2. Bachmair, L., Ganzinger, H.: Rewrite-Based Equational Theorem Proving with Selection and Simplification. Journal of Logic and Computation 3(4), 217–247 (1994)
3. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT: A Distributed and Learning Equational Prover. Journal of Automated Reasoning 18(2), 189–198 (1997), (Special Issue on the CADE 13 ATP System Competition)
4. Denzinger, J., Schulz, S.: Recording and Analysing Knowledge-Based Distributed Deduction Processes. Journal of Symbolic Computation 21(4/5), 523–541 (1996)
5. Hoder, K., Voronkov, A.: Sine Qua Non for Large Theory Reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Proc. of the 23rd CADE, Wroclav. LNAI, vol. 6803, pp. 299–314. Springer (2011)

6. Hsiang, J., Rusinowitch, M.: On Word Problems in Equational Theories. In: Proc. of the 14th ICALP, Karlsruhe. LNCS, vol. 267, pp. 54–71. Springer (1987)
7. Knuth, D., Bendix, P.: Simple Word Problems in Universal Algebras. In: Leech, J. (ed.) Computational Algebra, pp. 263–297. Pergamon Press (1970)
8. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) Proc. of the 25th CAV, LNCS, vol. 8044, pp. 1–35. Springer (2013)
9. Kühlwein, D., van Laarhoven, T., Tsivtsivadze, E., Urban, J., Heskes, T.: Overview and evaluation of premise selection techniques for large theory mathematics. In: Gramlich, B., Sattler, U., Miller, D. (eds.) Proc. of the 6th IJCAR, Manchester, LNAI, vol. 7364, pp. 378–392. Springer (2012)
10. Löchner, B., Hillenbrand, T.: A Phytography of Waldmeister. Journal of AI Communications 15(2/3), 127–133 (2002)
11. McCune, W.W.: Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/` (2005–2010), (acccessed 2016-03-29)
12. McCune, W.: Otter 3.0 Reference Manual and Guide. Tech. Rep. ANL-94/6, Argonne National Laboratory (1994)
13. McCune, W., Wos, L.: Otter: The CADE-13 Competition Incarnations. Journal of Automated Reasoning 18(2), 211–220 (1997), (Special Issue on the CADE 13 ATP System Competition)
14. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. Journal of Applied Logics 7(1), 41–57 (2009)
15. Riazanov, A., Voronkov, A.: The Design and Implementation of VAMPIRE. Journal of AI Communications 15(2/3), 91–110 (2002)
16. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. Journal of Symbolic Computation 36(1–2), 101–115 (2003)
17. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1), 23–41 (1965)
18. Schäfer, S., Schulz, S.: Breeding theorem proving heuristics with genetic algorithms. In: Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.) Proc. of the Global Conference on Artificial Intelligence, Tibilisi, Georgia. EPiC, vol. 36, pp. 263–274. EasyChair (2015)
19. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications 15(2/3), 111–126 (2002)
20. Schulz, S.: System Description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) Proc. of the 19th LPAR, Stellenbosch. LNCS, vol. 8312, pp. 735–743. Springer (2013)
21. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A Cross-Community Infrastructure for Logic Solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Proc. of the 7th IJCAR, Vienna. LNCS, vol. 8562, pp. 367–373. Springer (2014)
22. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
23. Weidenbach, C., Schmidt, R., Hillenbrand, T., Topić, D., Rusev, R.: SPASS Version 3.0. In: Pfenning, F. (ed.) Proc. of the 21st CADE, Bremen. LNAI, vol. 4603, pp. 514–520. Springer (2007)