Empirical Properties of Term Orderings for Superposition

Stephan Schulz[1]

[1]DHBW Stuttgart, Stuttgart, Germany

Abstract

Term orderings play a critical role in most modern automated theorem proving systems, in particular for systems dealing with equality. In this paper we report on a set of experiments with the superposition theorem prover E, comparing the performance of the system under Knuth-Bendix Ordering and Lexicographic Path Ordering with a number of different precedence and weight generation schemes. Results indicate that relative performance under a short time limit seems to be a good predictor for longer run times and that KBO generally outperforms LPO. Also, if other strategy elements (especially clause selection) are independent of the ordering, performance of a given ordering instance with one strategy is a decent predictor of performance with a different strategy.

1. Introduction

The root cause of difficulty in automated theorem proving is the size of the search space, i.e. the space of possible derivations. Many improvements in calculi have introduced ways to reduce the branching factor of this search space, by identifying strictly unnecessary inferences. One of the more important techniques in the context, in particular for equational reasoning, is the use of term orderings, to break the symmetry of equality, and also to impose additional constraints on possible inferences. This was pioneered by Knuth and Bendix in their paper on completion [1], which also introduced what we today call the Knuth-Bendix-Ordering (KBO), still one of the most important orderings for automated theorem proving. The completion approach was continued to a complete theorem proving method for unit equality problems [2, 3], and finally lifted to full clause logic with equality in the superposition calculus [4, 5], which still forms the basis of most modern saturating calculi and most high-performance theorem provers. In these calculi, term orderings are used in two principal ways: First, equalities can only be applied in (at least potentially) decreasing order, and secondly, many inferences can be restricted to be only applied on maximal literals.

When these theoretical results are translated to practical reasoning systems, there are additional challenges. Orderings need to be efficiently implemented, and they need to be instantiated for any given proof problem. Both the KBO and the LPO (Lexicographic Path Ordering), the two most frequently used orderings, are actually ordering schemata.

PAAR'22: 8th Workshop on Practical Aspects of Automated Reasoning, August 11–12, 2022, Haifa, Israel

In the concrete case, both need a symbol precedence. The KBO additionally needs an assignments of weights to function (and predicate) symbols.

While e.g. Otter [6] pioneered an *automatic mode* that would also set ordering parameters, other systems, such as DISCOUNT [7]) and early Waldmeister [8], left the task to the user, who, at that time, was assumed to be familiar with the domain and have the necessary expertise. However, the CADE ATP System Competition [9], starting with CASC-13 in 1996 [10], established a paradigm where the provers were given only the logic formula proper, and had to determine all search control parameters themselves. While nearly all competitive ATP systems have implemented some method, to our knowledge there has never been a systematic practical evaluation of different ways to solve the order generation problem.

Over time, the theorem prover E[11, 12] has accumulated a number of different ways to generate symbol precedence and weight generation schemes, based only on simple syntactic features of the symbols and the proof problems. It also features an efficient implementation of both KBO and LPO, based on Löchner's excellent and possibly definitive work [13, 14].

In this paper, we want to address a number of questions regarding the performance of these different schemes.

1. How do they compare for overall performance?
2. Does good performance of an ordering in one setting translate to different settings? In particular, if we replace only the ordering, can (relative) performance be extrapolated from one search strategy to another?
3. Similarly, can (relative) performance be extrapolated from shorter to longer run times?

Most of our experiments were done only on unit-equational examples, where the influence of the term ordering is more important. However, we have also ran one experiment on all usable TPTP [15] first-order problems. That allows us to address one additional question:

4. How does the relative performance on UEQ compare to that on the larger FOF/TFF set?

Finally, most theorem provers we are aware of implement rewriting with unorientable unit equations in a way that is strictly weaker than supported by theory. In E, we have introduced rewriting with *strong instances*, a technique that often allows for more simplification than these conventional implementations. We also analyse the data with respect to this new technique.

We start the paper with a short definition of the most important terms and concepts, including the idea of rewriting with strong instances. In the next section, we discuss the experimental setting, including the different ordering generation schemes and applications to simplification. In the experimental section, we discuss the primary results. Finally, we mention some related and future work, and conclude.

$$\text{(SP)}\ \frac{C \vee s \simeq t \qquad D \vee u \not\simeq v}{(C \vee D \vee u[p \leftarrow t] \not\simeq v)\sigma} \quad \text{if}$$

- $u|_p$ is not a Variable, $\sigma = mgu(u|_p, s)$
- $s\sigma \not< t\sigma$, $u\sigma \not< v\sigma$
- $(s \simeq t)\sigma$ is strictly maximal in $(C \vee s \simeq t)\sigma$
- $(u \not\simeq v)\sigma$ is maximal in $(D \vee u \not\simeq v)\sigma$

$$\text{(RP)}\ \frac{s \simeq t \qquad u \simeq v \vee C}{s \simeq t \qquad u[p \leftarrow t\sigma] \simeq v \vee C} \quad \text{if}$$

- $u|_p = s\sigma$, $s\sigma > t\sigma$
- $u \simeq v$ is not maximal or $u \not> v$ or $u|_p \neq u$

Note that the double line in (RP) denotes a simplification rule, i.e. a rule in which the premises are *replaced* by the results.

Figure 1: Core inferences of the superposition calculus

2. Preliminaries

We assume the standard first-order notions of a signature, consisting of function symbols with associated arities, and variable symbols. Terms are either variables, or are constructed from a function symbol with the proper number of arguments. Constants are a special case of function symbols with arity zero (and hence a constant is a term). Terms without variables are called ground terms. Equations are unordered pairs of terms, literals are either equations or negated equations, and clauses are multi-sets of literals[1] interpreted (and written) as disjunctions. Substitutions replace variables by terms and can be applied to variables, terms, literals and clauses. We use s, t, u, v to denote terms, C, D to denote (partial) clauses, and σ for substitutions. We use $t|_p$ to denote the subterm of t at position p and $t[p \leftarrow s]$ for term t with the subterm at position p replaced by s.

Saturating theorem provers try to show that a set of clauses is unsatisfiable by the process of saturation, i.e. by using inference rules to derive new clauses from existing clauses with the aim of eventually producing the empty clause. The proof search is usually organised by some variant of the *given-clause-algorithm.* This algorithm maintains the proof state in the form of two separate sets of clauses. The set P of *processed clauses* is initially empty. The set U of *unprocessed clauses* initially contains all clauses of the proof problem. The algorithm repeatedly selects a *given clause* g from U and performs all inferences in which g is one premise and all other premises are from P. The given clause is then moved into P, the resulting new clauses into U. The algorithm terminates if the empty clause is derived (representing a proof that the original clause set is unsatisfiable), or if the set U runs empty (in which case the proof search fails). In addition to this basic operations, a significant proportion of time is spent on *simplification*, removing redundant clauses and replacing clauses with, in a certain sense, smaller and simpler clauses. The selection of the given clause has to be *fair* (i.e. no clause can be delayed infinitely long), and is also one of the most important heuristic choice points for the proof search.

The most important inference in modern calculi is *superposition*, a restricted version of *paramodulation.* A paramodulation inference has two premises: the main premise, an

[1]Non-equational literals can be encoded as equations with a reserved constant $\top$ with a separate Boolean sort.

instance of which forms the core of the newly inferred clause, and a side premise, which contains at least one positive equational literal and is used as a (lazy) conditional rewrite rule to modify the instance of the main premise. To perform an inference, one side of a positive literal of the side premise is overlapped into a subterm of the main premise, followed by instantiation and replacement of the overlapped term by the instance of the other side of the inference literal in the side premise. The instances of other literals of the side premise are inherited by the newly generated child clause. Paramodulation inferences can be restricted in certain ways. In particular, we can use a *simplification ordering* on terms, lift it to literals, and restrict inferences to maximal literals. In this case, we call such an inference a *superposition* inference. Fig 1 shows the inference rule (SP) for superpositions into a positive literal. Term orderings also allow us to use strictly decreasing instances of positive unit clauses for simplification (i.e. destructive rewriting, in which a parent clause is replaced by the result). Rule RP in figure 1 describes rewriting of a positive literals. Both rules have very similar counterparts for overlapping into or rewriting negative literals. See e.g. [4] for details and [11] for a more practice-oriented exposition.

As described above, the two most frequently used classes of orderings are the KBO [1] and the LPO [16] (but see [14, 13] for excellent modern treatments).

The KBO requires a weight assignment to function symbols, and a precedence on these symbols. It compares terms first by weight (i.e. numerical comparison of the summed weight of all symbols in the term), breaking ties by top symbol (using the precedence), and further breaking ties by lexicographic comparison of subterms. It additionally requires each variable to occur in the smaller term no more often than in the larger term.

The LPO requires only a precedence on function symbols, and essentially compares terms lexicographically, recursively ensuring that no larger subterms are hidden deep in the term structure of the potentially smaller term.

3. Term Orderings in Practice

As stated above, both LPO and KBO need to be instantiated with parameters - a symbol precedence in the case of LPO, both a symbol precedence and a weight assignment for KBO. In E, we have implemented 13 different mature precedence generation schemes, and 26 different mature weight assignment schemes. All are based on simple features of the signature (primarily arity) and the number of occurrences in the original input specification (frequency). E always generates the symbol precedence first, and the weight assignment scheme has the symbol precedence available as an extra input parameter.

Since any precedence can be combined with any weight assignment, this represents 13 different LPO instances, and 13 times 26, or 338 different KBO instances.

3.1. Term Ordering Generation Schemes

The complete list of precedence generation schemes used in the experiments is as follows: `unary_first`, `unary_freq`, `arity`, `invarity`, `const_max`, `const_min`, `freq`, `invfreq`, `invconjfreq`, `invfreqconjmax`, `invfreqconjmin`, `invfreqconstmin`, `invfreqhack`.

We will discuss some selected schemes in some detail - see [17] for more detailed notes. While naming is not perfectly consistent, there are some general rules. Any scheme starting with `inv` will just order symbols in the opposite (or *inverse*) way of the corresponding non-`inv` scheme.

As a simple and long-lived example, the `arity` function just orders symbols by arity, making symbols with a bigger arity greater. Since all our schemes generate total precedences, ties are broken by the order of appearance in the signature table (which corresponds to the order in which symbols occur in the input). The `unary_first` scheme is a variant that will make unary function symbols maximal. This allows the prover to generate the ordering originally suggested in [1] for the completion of the group axioms. It also turns out to perform very well for UEQ problems in general. Another useful scheme, in particular for general problems, is `invfreq`. This makes rare symbols bigger then more frequent symbols.

The 26 different weight assignment schemes are: `firstmaximal0`, `arity`, `aritymax0`, `modarity`, `modaritymax0`, `aritysquared`, `aritysquaredmax0`, `invarity`, `invaritymax0`, `invaritysquared`, `invaritysquaredmax0`, `precedence`, `invprecedence`, `precrank5`, `precrank10`, `precrank20`, `freqcount`, `invfreqcount`, `freqrank`, `invfreqrank`, `invconj-freqrank`, `freqranksquare`, `invfreqranksquare`, `invmodfreqrank`, `invmodfreqrank-max0`, `constant`.

An interesting case is `firstmaximal0`, which complements `unary_first` to generate the original KBO for group completion. It assigns a weight of 1 to all symbols, except to the first (non-constant) maximal symbol in the signature, which is assigned weight 0. A more complex example is `invfreqconjmin`. In this scheme, symbols are ordered by inverse frequency, but symbols that occur in the problem's conjecture (if any) are smaller than symbols that occur only in axioms. Weights are then assigned sequentially, starting with 1 for the most frequent conjecture symbol. The `precedence` scheme assigns weights according to the (total) precedence, with symbols that are larger in the precedence also getting higher weights.

KBO also has the constraint that constants must have at least the weight of variables. In our experiments, we always assigned a fixed weight of 1 to both constants and variables.

3.2. Rewriting with strong instances

One of the major advantages of superposition-based calculi is that they allow for strong redundancy elimination. One of the most important simplification techniques in practice is unconditional rewriting. It allows the use of a positive unit-equational clause (i.e. a single equation) to potentially replace a term in another clause by a smaller term. For this, two conditions have to be met[2]: On the one hand, one side of the equation (called the left-hand side or lhs) must match the subterm to be rewritten. And on the other hand, the instance used for rewriting must be decreasing, i.e. the instantiated other side (rhs) of the equation has to be strictly smaller than the replaced term.

[2]Rewriting is also restricted for a small set of term positions by the calculus, but that is not our concern here.

Most provers combine the matching and the instantiation operation. Thus, if an lhs matches a candidate term, the corresponding substitution is applied to both sides of the equation, and then the ordering condition is checked. This is often sufficient, however, the test will automatically fail if the rhs contains a variable not occurring in the lhs, because such a free variable is always incomparable with any term in which it does not occur. To be able to still use these equations, we have introduced the concept of a *strong instance* of an equation. In a strong instance, we dynamically instantiate all variables of the equation before performing the ordering test. Variables in the lhs of an equation are automatically bound by the match against the term to be rewritten. Additionally, we substitute any remaining unbound rhs variables with the smallest constant (of the proper sort). As an example, consider the equation $R = f(x, a) \simeq g(x, y)$ (where f is greater than g) applied to the term $t = f(a, a)$ and let $\perp$ be the smallest constant in the signature. The lhs $f(x, a)$ matches t with $\sigma = \{x \mapsto a\}$. But $\sigma(R) = f(a, a) \simeq g(a, y)$ is unorientable. If we extend σ to also replace the unbound variable y by $\perp$, we do get $f(a, a) \simeq g(a, \perp)$, which is orientable, and hence can be used to reduce the term. This is sound, because variables are universally quantified, i.e. the equality holds for all instances. We have implemented this simple concept in E, and it has shown a significant and consistent improvement in performance.

4. Experimental results

4.1. Experimental settings

We have employed four different search strategies. All normal search parameters were fixed to (usually) known good values. The only difference between the four strategies was in the clause selection heuristic. We broadly describe the 4 strategies below, see [18] for a detailed exposition on clause selection in E.

SC: This is a pure symbol-counting strategy. All function, predicate, and variable symbol occurrences are counted with a weight of one, and the lightest clauses are processed first. The exact command line options for search control options are available in the online supporting material (see below).

20:1: This selection strategy interleaves selecting the lightest and the oldest clause, with a ratio of 20 to 1.

EVO: This clause selection strategy has been honed through various optimizations [19, 20]. It interleaves a number of goal-directed and conventional selection queues, and is one of the best heuristics on TPTP we have found so far.

UEQ: This is similar to **EVO** in structure and intent, but optimised only towards unit-equational problems.

We ran tests on all suitable problems from TPTP 7.5.0. Most experiments were run on unit-equational problems. This set contained 1497 different problems. Some tests used all suitable first-order problems from TPTP (i.e. all CNF and FOF problems). This set was more than 10 times larger, containing 17283 problems. Run time per problem was limited to 5 seconds of CPU time.

Tests were run on a machine with four Intel Xeon Gold 6130 CPUs, running a total of 64 physical cores at a nominal frequency of 2.10GHz and with 512GB of RAM. RAM size was sufficient to guarantee no contention or swapping between parallel processes.

We have run 351 different ordering schemes with 4 different search strategies and two different rewrite settings, i.e. 2808 different strategies, on 1497 UEQ problems, for a total of 4 203 576 individual UEQ test runs. We have run the same 351 strategies, with only the 20:1 clause selection strategy and conventional rewriting, on all 17283 problems for 6 066 333 individual test runs for the general case. Thus, the total corpus has data on a bit over 10 million test runs.

A version of the prover suitable for reproducing the actual dataset runs, as well as the summarized results of the test runs, is available as an online supplement at http://eprover.eu/E-eu/Term_Orderings.html. The protocols included there also specify the exact command line options and contain the lists of problems used. TPTP 7.5.0, the source of the test problems, can be downloaded from http://www.tptp.org. The latest official release of E (and future updates) can be found at https://www.eprover.org.

4.2. Performance on unit-equational problems

We will first discuss the performance of the different orderings with conventional rewriting on the 1497 UEQ problems. Table 1 gives a general overview of the data. Each row corresponds to a search strategy (differentiated by clause selection strategy and use of rewriting with strong instances). For each row, we show the number of problems solved by the best, worst, and median term ordering variant, and by the union of all variants, separately for LPO, KBO, and both.

We can see that the difference between the best and the worst ordering is roughly 110-120 successes, and about comparable to the difference between the weakest and strongest search strategy, which is roughly 130-145. We can also already see that for UEQ problems, the best LPO strategy is systematically worse than the worst KBO strategy. However, for each search strategy the union of both LPO and KBO strategies solves more problems than either alone.

To get a better understanding of the distribution of results, we have plotted the data in Fig. 2. In this figure, each individual measurement is a lightly coloured dot. The corresponding thin line is a gliding average. For each search strategy, the individual orderings are sorted in ascending order by number of successes. This results in a very distinct *hockey stick* graph, with an strongly ascending "blade", followed by a more slowly rising "handle". The blade consists of the results for the 13 different LPO instances, the handle of the $13 \cdot 26 = 338$ KBO instances. While this graph suggests very similar distribution of performance for each search strategies, it does not tell us if we see the same orderings in the same order, or if the different ordering variants appear in different order for the different search strategies.

To remedy this, we have replotted the same data in Fig. 3. In this figure, we use the SC strategy as the reference that determines the order of ordering variants for all 4 search

Strategy	**All orderings**				**KBO only**				**LPO only**			
	Min	Median	Max	Total	Min	Median	Max	Total	Min	Median	Max	Total
SC	610	707	719	782	691	708	719	740	610	655	668	733
20:1	680	783	798	857	762	784	798	824	680	720	740	820
EVO	773	872	883	920	848	872	883	897	773	802	820	877
UEQ	742	852	865	923	834	852	865	896	742	774	809	890
SC-SI	577	729	718	796	700	729	718	747	577	651	627	747
20:1-SI	679	803	789	867	771	803	790	833	679	729	709	826
EVO-SI	752	877	865	929	850	877	865	910	752	809	771	881
UEQ-SI	769	887	873	929	849	887	873	902	769	820	804	892
20:1-All	6523	7427	7298	8527	6792	7427	7306	8244	6523	7262	6998	8199

Remark: Strategies marked with SI uses rewriting with strong instances. The last line contains data for the full problem set, not just UEQ problems.

Table 1
Performance summary (number of successes) on UEQ and all problems

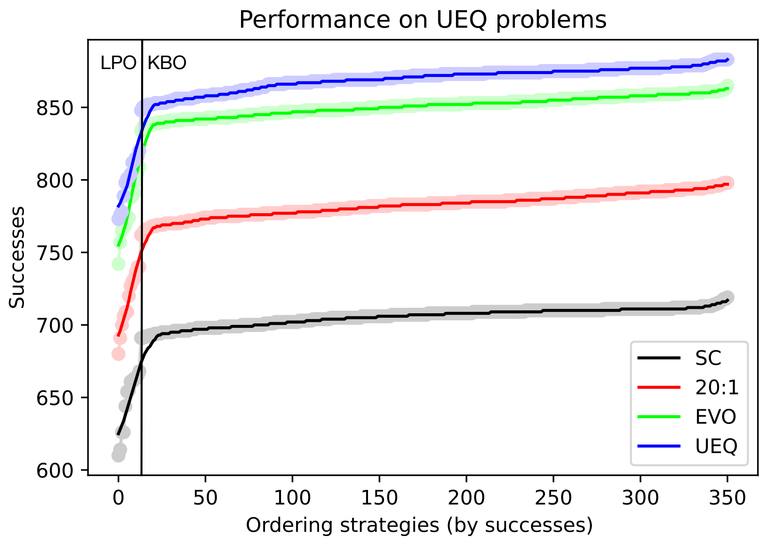
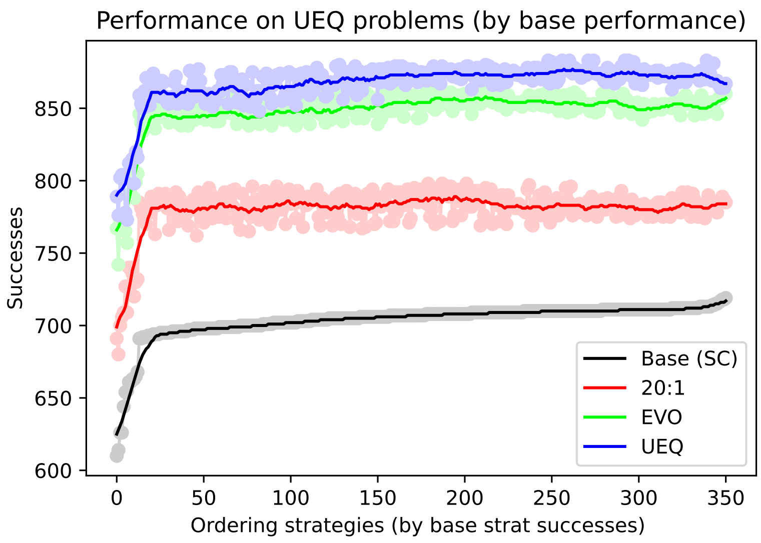


Figure 2: Performance spectrum on UEQ problems. The vertical black line corresponds to the transition from the 13 LPO ordering variants (on the left) to the KBO variants (on the right)

strategies. In other words, the ordering variants are represented by a fixed order on the X-axis for all different search strategies. We can see that the basic hockey stick shape remains, however, there is significant local variation, and the increase in the “handle” part of the graph is much less pronounced.

To quantify how far the order of strategies has changed between the different strategies, we compute Kendall’s Tau value [21]. This metric is based on the number of bubble-sort style swaps needed to bring two rankings in agreement, and measures correspondence between them (in this case, the two rankings of orderings induced by the performance data with different strategies). The result lies in $[1, -1]$, with 1 indicating perfect agreement and -1 completely opposite rankings. The algorithm also provides a p-value for the



Rankings	Kendall's Tau	p-Value
SC vs 20:1	0.084	1.84e-02
SC vs EVO)	0.321	2.67e-19
SC vs UEQ)	0.358	1.41e-23

Figure 3: Relative Performance on UEQ problems

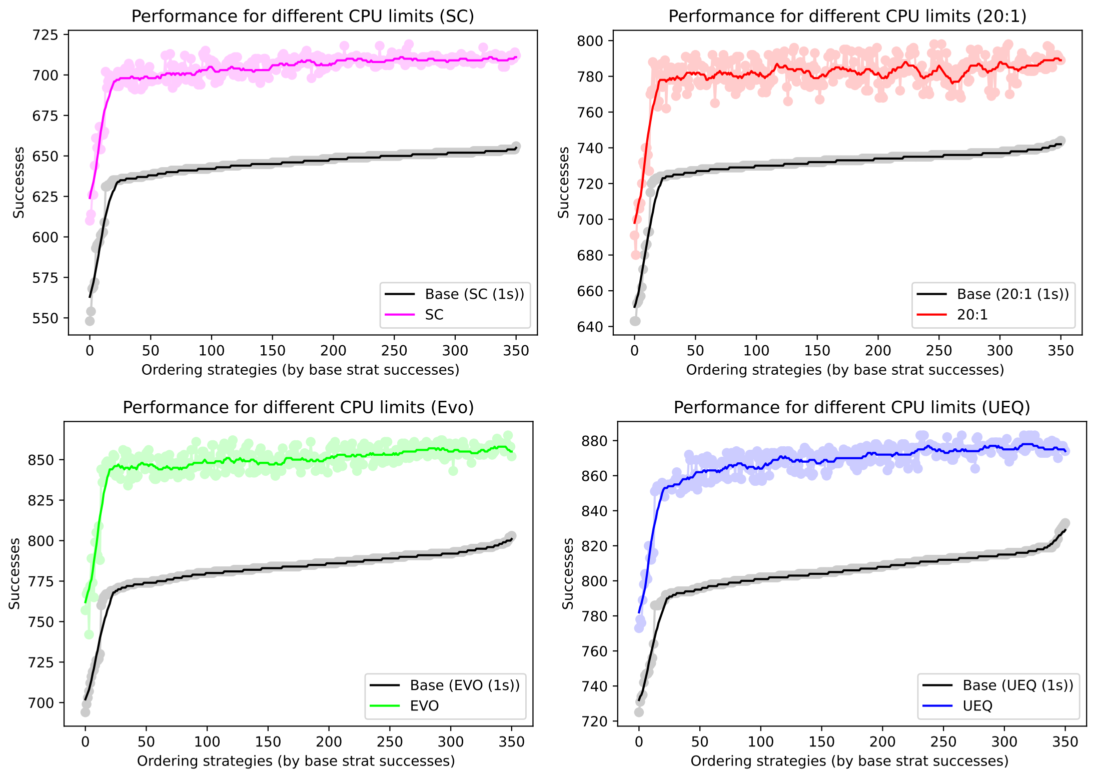
null-hypothesis that the two rankings have been produced by uncorrelated processes.

All rankings are related to the SC base case. The correlation is quite weak for 20:1, but much stronger for the other strategies. The null-hypothesis can be comfortably rejected for all three pairings.

We can identify a number of consistently badly performing orderings. For all 4 search strategies, LPO with precedence generation schemata `invarity` and `const_max` are always among the worst 5 orderings. It's less easy to pick out the winners, because the curves are fairly flat at the top. However, all strong orderings are KBO instances, and precedence schemata `unary_first`, `const_min` and `invfreqconstmin` appear frequently among the top strategies. For weight generation, `invaritymax0` and `invaritysquaredmax0` appear most frequently. The absolute best single combination is `unary_first` combined with `firstmaximal0`, however, there are 5 strategies with essentially the same performance.

4.3. Different CPU limits

Figure 4 shows the relative performance of the orderings for a CPU limit of 1 second, compared to a CPU limit of 5 seconds. For all 4 base strategies, there is good similarity between the curves, and for all but 20:1 there is a strong correlation. In all cases, the null-hypothesis is strongly rejected. Thus, it seems possible to extrapolate ordering performance from shorter to longer run times in most cases.



Rankings	Kendall's Tau	p-Value
SC 1s vs 5s	0.554	3.30e-54
20:1 1s vs 5s	0.259	4.23e-13
EVO 1s vs 5s	0.502	8.98e-45
UEQ 1s vs 5s	0.627	6.99e-69

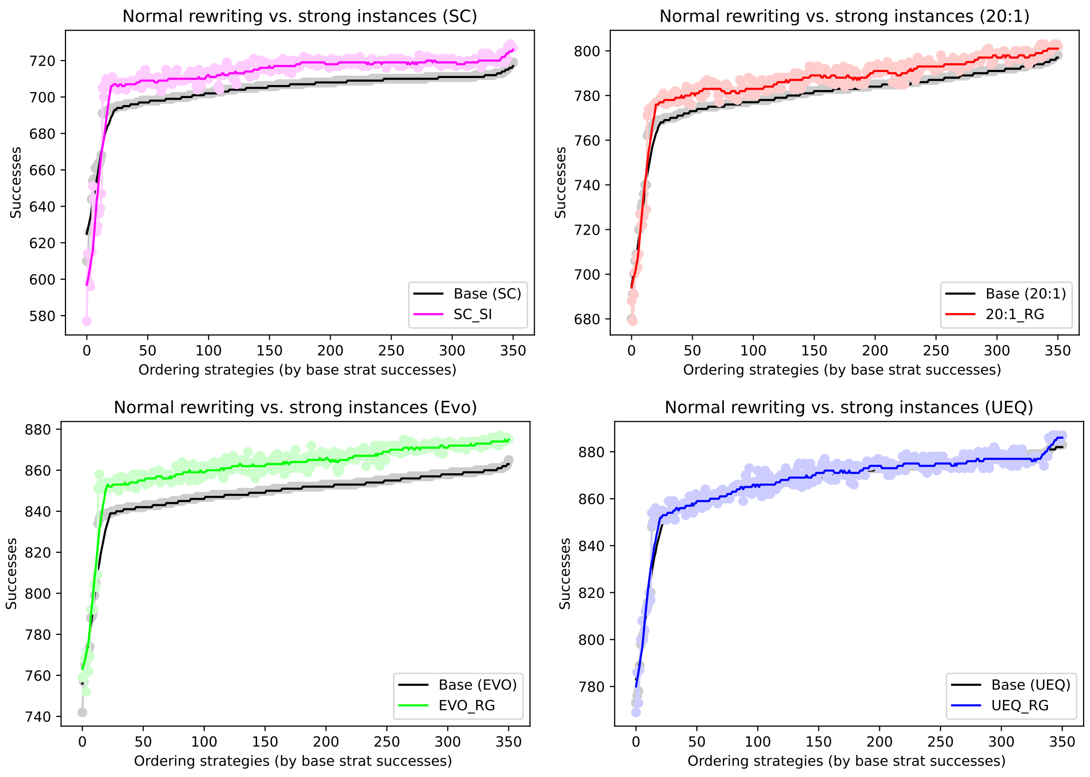
Figure 4: Comparison of performance for 1s and 5s CPU limits

4.4. Rewriting with strong instances

We have plotted the relative performance of the different orderings and strategies in Figure 5. As already visible from the summary data in Table 1, rewriting with strong instances performs consistently better than conventional rewriting. The difference is significant for all base strategies but UEQ, where it is minuscule. In all 4 cases we have an excellent agreement between relative performance of normal and strong instance variants, indicating that strong instance can probably be added to any prover and yield an improvement without disrupting existing strategies.

4.5. Performance on full first-order problems

Finally, we can compare the performance of all orderings on the full set of first-order problems, not just unit-equational problems. Since the number of problems is over 10 times larger now, we had to restrict the experiment to only one clause selection strategy.



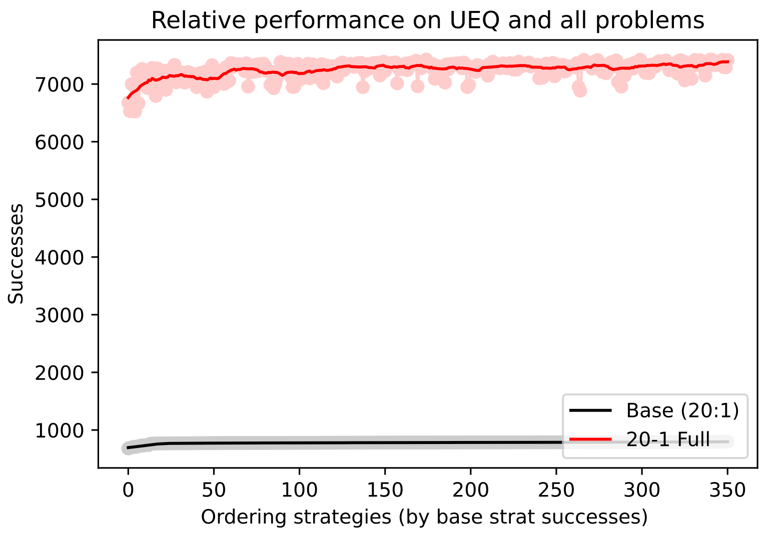
Rankings	Kendall's Tau	p-Value
SC vs. SC-SI	0.685	7.12e-82
20:1 vs 20:1-SI	0.813	1.51e-114
EVO vs EVO-SI	0.808	4.58e-113
UEQ vs UEQ-SI	0.790	3.32e-108

Figure 5: Performance with conventional and strong-instance rewriting

We picked 20:1, because it has decent performance and is easy to transfer to most other provers. The result is plotted in Figure 6. While the overall number of solutions for each ordering variant is several times higher for the full set (since the number of problems is higher), we can see that again performance on all problems correlates to performance on unit-equational problems, if not quite as pronounced as for some of the other comparisons.

5. Related and Future Work

The long term aim of this line of research is to automatically find good term orderings for any given problem. Our current approach is driven by the very successful standard saturation approach based on the given-clause loop. For the unit-equational case, there has been a lot of impressive work done for the Waldmeister prover, some of which is described in [22], but significant parts of which are only available as folklore. In particular, Waldmeister recognised domains by matching axioms on higher-order patterns, and



Rankings	Kendall's Tau	p-Value
UEQ vs All	0.345	4.72e-22

Figure 6: Relative performance on UEQ an all problems

instantiated ordering schemes based on the domain. Generalising this approach to full clausal logic, and also automatising the knowledge acquisition, would be a major step into the future.

One step in this direction is the work by Bartek and Suda [23], which describes an attempt to learn good precedences for KBO from features of the proof problem using a neural network architecture.

There has been some work about orienting a static system of equations with a KBO, e.g. [24]. However, this was a theoretical result with no experiments about the practical effects of computing such a static ordering before starting the proof search. Based on a similar premise is MædMax [25], a completion-based system for unit-equality. It is not built on the given-clause loop, but rather on a kind of level-saturation. At each iteration of its main loop, the system tries to find an ordering to orient as many equations as possible before the next round, before computing and normalising all critical pairs of the resulting system.

Our work with *strong instances* is quite conservative, in that it does not modify the clause set or inference rules, but only makes the implementation correspond more closely to the theoretical case. Twee [26] has an interesting alternative approach. It splits such unorientable equations into an orientable part, and a permutation equation. It would be interesting to see a direct comparison of both techniques.

Finally, saturating theorem provers are moving into higher-order [27] . One significant question is how many of these techniques can be lifted to this case, and how well they will work there.

6. Conclusion

Term orderings are a critical component of the search strategies of modern theorem provers. We have presented a first systematic overview of the performance of relatively naive term ordering generation schemes. With respect to our research questions, we can say the following:

1. KBO performs almost uniformly better than LPO. However, both KBO and LPO instances are necessary to solve all solvable problems. Differences between different LPO instances are much bigger than those between different KBO instances, but the latter are still significant.
2. There is some correlation between the performance of different orderings under different other settings. However, relative performance between similarly performing strategies cannot be reliably predicted.
3. It is, on the other hand, quite possible to predict relative performance for longer runtimes from shorter runtimes. That is particularly valuable for system tuning, since tuning experiments can be done with relatively short run times.
4. Maybe surprisingly, there is a quite good correlation between performance of an ordering in the UEQ domain and over all problems, despite the fact that orderings play additional roles for reasoning with general clauses.

The raw data is available for further evaluation and experiments, and there is a clear path to both generate more diverse and stronger orderings, and to couple ordering generation to the presence of certain axioms and axiom structures.

Acknowledgments

The various anonymous reviewers of this paper have been extremely helpful, identifying, in addition to several language issues and typos, a wrong number in Table 1 that slipped in during editing.

References

[1] D. Knuth, P. Bendix, Simple Word Problems in Universal Algebras, in: J. Leech (Ed.), Computational Algebra, Pergamon Press, 1970, pp. 263–297.

[2] J. Hsiang, M. Rusinowitch, On Word Problems in Equational Theories, in: Proc. of the 14th ICALP, Karlsruhe, volume 267 of *LNCS*, Springer, 1987, pp. 54–71.

[3] L. Bachmair, N. Dershowitz, D. Plaisted, Completion Without Failure, in: H. Ait-Kaci, M. Nivat (Eds.), Resolution of Equations in Algebraic Structures, volume 2, Academic Press, 1989, pp. 1–30.

[4] L. Bachmair, H. Ganzinger, Rewrite-Based Equational Theorem Proving with Selection and Simplification, Journal of Logic and Computation 3 (1994) 217–247.

[5] R. Nieuwenhuis, A. Rubio, Paramodulation-Based Theorem Proving, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, volume I, Elsevier Science and MIT Press, 2001, pp. 371–443.

[6] W. McCune, L. Wos, Otter: The CADE-13 Competition Incarnations, Journal of Automated Reasoning 18 (1997) 211–220. Special Issue on the CADE 13 ATP System Competition.

[7] J. Denzinger, M. Kronenburg, S. Schulz, DISCOUNT: A Distributed and Learning Equational Prover, Journal of Automated Reasoning 18 (1997) 189–198. Special Issue on the CADE 13 ATP System Competition.

[8] T. Hillenbrand, A. Buch, R. Vogt, B. Löchner, WALDMEISTER. High Performance Equational Deduction, Journal of Automated Reasoning 18 (1997) 265–270. Special Issue on the CADE 13 ATP System Competition.

[9] G. Sutcliffe, The CADE ATP System Competition – CASC, AI Magazine 37 (2016) 99–101. doi:10.1609/aimag.v37i2.2620.

[10] F. J. Pelletier, G. Sutcliffe, C. Suttner, Conclusions about the CADE-13 ATP system competition, Journal of Automated Reasoning 18 (1997) 287–296.

[11] S. Schulz, E – A Brainiac Theorem Prover, Journal of AI Communications 15 (2002) 111–126.

[12] S. Schulz, S. Cruanes, P. Vukmirović, Faster, higher, stronger: E 2.3, in: P. Fontaine (Ed.), Proc. of the 27th CADE, Natal, Brasil, number 11716 in LNAI, Springer, 2019, pp. 495–507.

[13] B. Löchner, Things to Know when Implementing KBO, Journal of Automated Reasoning 36 (2006) 289–310.

[14] B. Löchner, Things to Know When Implementing LPO, International Journal on Artificial Intelligence Tools 15 (2006) 53–80.

[15] G. Sutcliffe, The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0, Journal of Automated Reasoning 59 (2017) 483–502.

[16] S. Kamin, J.-J. Levy, Two generalizations of the recursive path ordering, Technical Report, Departement of Computer Science, University of Illinois, Urbana, IL, 1980.

[17] S. Schulz, E 2.4 User Manual, EasyChair preprint no. 2272, 2019. URL: https://easychair.org/publications/preprint/RjDx.

[18] S. Schulz, M. Möhrmann, Performance of clause selection heuristics for saturation-based theorem proving, in: N. Olivetti, A. Tiwari (Eds.), Proc. of the 8th IJCAR, Coimbra, volume 9706 of *LNAI*, Springer, 2016, pp. 330–345.

[19] S. Schäfer, S. Schulz, Breeding theorem proving heuristics with genetic algorithms, in: G. Gottlob, G. Sutcliffe, A. Voronkov (Eds.), Proc. of the Global Conference on Artificial Intelligence, Tibilisi, Georgia, volume 36 of *EPiC*, EasyChair, 2015, pp. 263–274.

[20] J. Urban, Blistr: The blind strategymaker, in: G. Gottlob, G. Sutcliffe, A. Voronkov (Eds.), Proc. of the Global Conference on Artificial Intelligence, Tibilisi, Georgia, volume 36 of *EPiC*, EasyChair, 2015, pp. 312–319.

[21] M. G. Kendall, Rank Correlation Methods, 4th ed., Charles Griffin & Co., 1970.

[22] T. Hillenbrand, A. Jaeger, B. Löchner, System Abstract: Waldmeister – Improvements in Performance and Ease of Use, in: H. Ganzinger (Ed.), Proc. of the 16th CADE, Trento, volume 1632 of *LNAI*, Springer, 1999, pp. 232–236.

[23] F. Bártek, M. Suda, Neural Precedence Recommender, in: A. Platzer, G. Sutcliffe (Eds.), Proc. of the 28th CADE, Pittsburgh, volume 12699 of *LNAI*, Springer, 2021,

pp. 525–542.

[24] K. Korovin, A. Voronkov, Orienting rewrite rules with the Knuth–Bendix order, Journal of Information and Computation 183 (2003) 165–186.

[25] S. Winkler, G. Moser, MædMax: A maximal ordered completion tool, in: D. Galmiche, S. Schulz, R. Sebastiani (Eds.), Proc. of the 9th IJCAR, Oxford, volume 10900 of *LNAI*, Springer, 2018, pp. 472–480.

[26] N. Smallbone, Twee: An Equational Theorem Prover, in: A. Platzer, G. Sutcliffe (Eds.), Proc. of the 28th CADE, Pittsburgh, volume 12699 of *LNAI*, Springer, 2021, pp. 602–613.

[27] P. Vukmirović, J. C. Blanchette, S. Cruanes, S. Schulz, Extending a Brainiac Prover to Lambda-free Higher-Order Logic, International Journal on Software Tools for Technology Transfer (2021). doi:10.1007/s10009-021-00639-7.