Smarter Features, Simpler Learning?

Georg Moser and Sarah Winkler*

[1] University of Innsbruck
georg.moser@uibk.ac.at
[2] University of Innsbruck
sarah.winkler@uibk.ac.at

Abstract

Earlier work on machine learning for automated reasoning mostly relied on simple, syntactic features combined with sophisticated learning techniques. Building on ideas exploited in the software verification competition (SV-COMP), we propose the investigation of more complex, structural features to learn from. These may be exploited to either learn suitable strategies for tools, or build a portfolio solver which chooses the most suitable tool for a given problem. We propose concrete features for term rewrite systems, and give some ideas for the area of theorem proving.

1 Introduction

The idea to exploit machine learning in automated reasoning has been a natural one, given the success of AI methods in areas like Jeopardy[1] and Go.[2]

In theorem proving, learning has been investigated to improve heuristics for different tasks. Most prominently, these include learning parameters of the proof process [7], learning selection of the objects (like clauses) to process next [15, 10], premise selection [12] and the use of learning to guide proof search in a more general sense, such as which branch to follow in a tableau proof [11]. Work done in the area so far typically exploited syntactic features.[3] On the one hand, static features such as symbol counts, the number of clauses of a certain type, clause depth and weight were investigated, along with dynamic features that record similar features during the proof process [7]. Alternatively, term walks are used as features, where terms are represented as symbol strings [15, 10]. In another experiment, the entire problem was fed into a neural network [14].

These approaches tend to use elaborate machine learning models with rather simple features. A contrasting approach following the paradigm of *smart features, simple learning* has been pursued by Demyanova *et al* to design a portfolio solver for the software verification competition (SV-COMP) [9]: Based on metrics of programs related to loop types, variable use, and control flow, the authors applied machine learning techniques to decide which solver to use. The resulting portfolio solver would have won the software competition in three consecutive years. The applied machine learning model is comparatively simple (support vector machines), the success of the method is thus attributed to the sophisticated metrics. Besides providing a powerful combined solver, the work thus also delivers a taxonomy of problems, along with insights which techniques and tools work best on them.

*Supported by Austrian Science Fund project T789.

[1]Watson, IBM, https://researcher.watson.ibm.com/researcher/view_group.php?id=2099

[2]AlphaGo, DeepMind, https://deepmind.com/research/alphago.

[3]Since space limitations do not allow us to present a comprehensive related work, we only mention some examples as representatives of common approaches.

We propose to apply a similar approach to the area of term rewriting. Both in the termination[4] and the confluence competition[5] a variety of tools competes annually on thousands of benchmarks. Each tool has its own strengths on problems of a certain kind, but it can typically also be run with a vast variety of different strategies that accommodate better to some problems than others. Though some tools are restricted to problems with certain syntactic characteristics, like string rewrite systems, many tools work on the standard categories and offer a variety of proof strategies. The same holds for the area of theorem proving and the CADE Annual System Competition (CASC)[6].

We propose to investigate meaningful structural features that, possibly together with syntactic features, allow machine learning models to predict on the one hand which tool works best, and on the other hand which tool strategy is most suitable for a problem.

In Section 2 we summarize the program metrics exploited in [9]. We propose corresponding term rewrite system characteristics in Section 3. Subsequently, we consider possible extensions to problems from theorem proving in Section 4 and conclude.

2 Metrics for Programs

The metrics of programs proposed in [9] are based on three characteristics: variable roles, loop patterns, and control flow. For the former, 27 variable roles are defined, including e.g., pointers, loop bounds, and counters. An occurrence count vector holding the number of variables of each role, normalized by the total number of variables, is included in the features used for learning.

Second, four loop patterns are defined based on whether the number of iterations in a program execution can be bounded or not. In particular, the first two patterns correspond to loops with a constant number of iterations, and loops corresponding to a FOR-loop where termination is ensured. Again, the number of loops of each kind gets counted, and an occurrence count vector (normalized by the number of loops overall) is included in the feature list.

Finally, control flow metrics include the number of basic blocks and the maximal indegree of a basic block for intraprocedural control flow, as well as the number of (recursive) call expressions an the involved parameters to account for interprocedural control flow.

3 Features of Term Rewrite Systems

There are obviously certain syntactic features which exclude the application of certain tools of techniques. These include e.g. the type of rewrite systems (string, conditional, or constrained) but also syntactic features such as e.g. variable duplication for the Knuth-Bendix order.

That said, we here want to discuss more structural features which might influence applicability of certain tools and methods, as outlined in Section 2. While programs are certainly more structured than term rewrite systems (TRSs), many properties outlined above are reflected on the level of TRSs.

Variable Roles. The role of a program variable corresponds to the role of the argument position of a function symbol. Consider a symbol f of arity n occurring in a TRS $\mathcal{R}$, with argument position $i \leq n$. Properties of interest may include the following:

[4] http://termination-portal.org/wiki/Termination_Competition
[5] http://project-coco.uibk.ac.at
[6] http://www.tptp.org/CASC

- i is a *projection argument* if $\mathcal{R}$ contains a rule $f(t_1, \dots, t_n) \to t_i$
- argument i is *decreasing* in presence of a rule $f(t_1, \dots, s(t_i), \dots t_n) \to f(t_1, \dots, t_i, \dots t_n)$ (or *increasing* if the rule is reversed)
- i is *recursive* if some rule $f(t_1, \dots, t_n) \to f(t_1, \dots, f(u_1, \dots, u_n), \dots t_n)$ features a recursive call in the ith position
- i is a *pattern matching argument* if $\mathcal{R}$ has rules $f(t_1, \dots, c_1, \dots, t_n) \to r_1$ and $f(t_1, \dots, c_2, \dots, t_n) \to r_1$ with different constructors c_1 and c_2 occuring at position i
- i is a *duplication argument* if in a rule $f(t_1, \dots, t_n) \to r$ term t_i occurs at least twice in r

Note that some properties have likely correspondance to roles of program variables, for instance a decreasing or increasing argument position models a counter variable in a term rewrite system.

Recursion Patterns. In particular for termination and complexity analysis, recursion in TRSs constitutes the key difficulty. The first two loop patterns considered in [9] is strongly connected to the idea of tiering and safe recursion [16, 4, 13], where the essential idea is that thearguments of a function are separated into normal and safe arguments, and recursion is limited to safe arguments. Such a problem classification is also used in runtime complexity analysis of term rewrite systems [2, 1]. More generally, restricted loop structures have been considered in the context of decidable classes of resource or termination analysis, cf. [5, 6, 8].

Control Flow. Control flow in TRSs is commonly captured by the dependency graph (DG). Estimations of this graph are, for instance, commonly used in termination tools. Natural features of this graph are given by the number of nodes, edges, and strongly connected components (SCCs) of this graph. Though the control flow is in general less obvious for TRSs than for programs, some of the program metrics mentioned above have quite direct correspondents: The number of recursive calls is reflected by the number of paths in the DG between nodes with the same root symbol. Mutually recursive functions are reflected by edges between nodes and SCCs of different root symbols. The indegree of nodes in the DG could be seen as related to the indegree of basic blocks.

However, the DG is typically used to analyze termination: its nodes are thus dependency pairs and edges correspond to potential recursive calls. For the sake of control flow analysis, one could also consider a more general call graph where nodes stand for rewrite rules rather than dependency pairs, and terminating computations are covered as well.

Finally, we mention some further features for term rewrite systems which carry some semantics. One may consider properties such as orthogonality, the number of critical pairs, or whether it is a constructor system. But also more complex properties such as simple (non-)termination and local (non-)confluence can be easily decided for some cases, and might be indicative for the complexity of an input TRS and hence appropriate strategies. Moreover, also the presence of certain algebraic substructures such as associative and commutative operators, groups, lattices, or distributivity can be taken into account.

4 Conclusion

In presence of equality literals, all features described in Section 3 can be considered for theorem proving problems as well. Some of the discussed features are, moreover, not limited to equality literals: A dependency graph can also be conceived to analyze the dependencies of predicates on certain function symbols, for instance. Similarly, the presence of certain substructures can be exploited, as it e.g. already been done in Waldmeister [3].

However, these problem features constitute only some initial ideas. We think it would make for an interesting topic for ARCADE to discuss whether (a) the approach outlined in this abstract is considered of interest, (b) whether the proposed problem metrics could be of use, and (c) which other features could be of use in the various areas of automated theorem proving.

References

[1] M. Avanzini, N. Eguchi, and G. Moser. A new order-theoretic characterisation of the polytime computable functions. *Theoretical Computer Science*, 585:3–24, 2015.

[2] M. Avanzini and G. Moser. Polynomial path orders. *Logical Methods in Computer Science*, 9(4), 2013.

[3] J. Avenhaus, T. Hillenbrand, and B. Lchner. On using ground joinable equations in equational theorem proving. *JSC*, 36(1–2):217–233, 2003.

[4] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Comput. Complex.*, 2(2):97–110, 1992.

[5] A. Ben-Amram. Monotonicity constraints for termination in the integer domain. *Log. Meth. Comput. Sci.*, 7(3), 2011.

[6] A. Ben-Amram and A. Pineles. Flowchart programs, regular expressions, and decidability of polynomial growth-rate. In *Proc. of the Fourth International Workshop on Verification and Program Transformation*, volume 216 of *EPTCS*, pages 24–49, 2016.

[7] J. P. Bridge, S. Holden, and L. Paulson. Machine learning for first-order theorem proving. *Journal of Automated Reasoning*, 53(2):141–172, Aug 2014.

[8] T. Colcombet, L. Daviaud, and F. Zuleger. Automata and program analysis. In *Proc. of the 21st International Symposium on Fundamentals of Computation Theory*, volume 10472 of *LNCS*, pages 3–10, 2017.

[9] Y. Demyanova, T. Pani, H. Veith, and F. Zuleger. Empirical software metrics for benchmarking of verification tools. *Form. Methods Syst. Des.*, 50(2-3):289–316, 2017.

[10] J. Jakubův and J. Urban. ENIGMA: efficient learning-based inference guiding machine. In *Proc. CICM 2017*, volume 10383 of *LNCS*, pages 292–302, 2017.

[11] C. Kaliszyk and J. Urban. FEMaLeCoP: Fairly efficient machine learning connection prover. In *Proc. LPAR 2015*, volume 9450 of *LNCS*, pages 88–96, 2015.

[12] A. Kucik and K. Korovin. Premise selection with neural networks and distributed representation of features. *CoRR*, abs/1807.10268, 2018.

[13] D. Leivant. Predicative recurrence and computatinal complexity I: Word recurrence and polytime. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.

[14] S. M. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. In *Proc. 21st LPAR*, pages 85–105, 2017.

[15] S. Schulz. Learning search control knowledge for equational theorem proving. In *Proc. KI 2001*, volume 2174 of *LNCS*, pages 320–334, 2001.

[16] H. Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27(2):177–188, 1988.