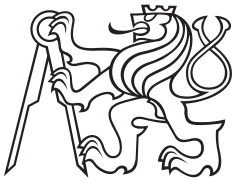
Dissertation Thesis



Czech Technical University in Prague

CIIRC

Czech Institute of Informatics, Robotics and Cybernetics
Department of Artificial Intelligence

Machine Learning for Saturation-Based Theorem Proving

Mgr. Filip Bártek

Supervisor: Mgr. Josef Urban, Ph.D.
Supervisor–specialist: RNDr. Martin Suda, Ph.D.
Field of study: Computer Science
March 2025

Acknowledgements

I thank my supervisor-specialist, Martin Suda, for his practical and informed guidance on research and publication in computer science. I thank my senior supervisor, Josef Urban, for welcoming me into his research group and introducing me to the field of automated reasoning.

I thank the reviewers of my work for numerous valuable suggestions that helped me make the presentation of my research more accessible and complete.

I thank the researchers with whom I have shared the workplace, especially Chad E. Brown, Clémence Chanavat, Anshula Gandhi, Thibault Gauthier, Zarathustra Goertzel, Jonathan Julián Huerta y Munive, Jan Hůla, Miroslav Olšák, Pedro Miguel Orvalho, Adam Pease, Jelle Piepenbrock, and Bartosz Piotrowski, for their welcoming and supportive attitude and for informative discussions. Michael Rawson, David Cerna, and Cezary Kaliszyk provided me with pleasant and inspiring company at conferences, for which I am thankful.

I thank my partner, Markéta Kociánová, for her patience and care. Finally, I am grateful to my parents, Ludmila Bártková and Jiří Bártek, for the financial and emotional support they provided me in my education.

Funding. The research presented in this doctoral thesis was supported by the Czech Science Foundation grants 20-06390Y and 24-12759S, the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement n° 649043), and the Grant Agency of the Czech Technical University in Prague, grant No. SGS20/215/OHK3/3T/37.

Declaration

I hereby declare that I have written this doctoral thesis independently and quoted all sources of information in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has not been submitted or accepted for any other degree.

In Prague, March 21, 2025

Abstract

Automatic theorem proving (ATP) for first-order logic (FOL) addresses a wide array of formalized tasks in domains such as mathematics and software verification. Saturation-based automatic theorem provers, notably the highly optimized Vampire prover, represent the cutting edge in ATP.

State-of-the-art provers employ various heuristics, and the success of a prover on a problem largely depends on the configuration of these heuristics. Traditionally, configuring the heuristics has been the domain of expert users. The vast number of available configurations and the unpredictability of their effects complicate this task, making machine learning (ML) an attractive alternative to human expertise.

This dissertation explores the application of ML in saturation-based ATP, with research focused on enhancing the performance of the Vampire prover through three developed approaches. The first approach involves a system that recommends, for an arbitrary input problem, a symbol precedence to instantiate a simplification ordering on terms in order to prune the proof search efficiently. The second approach leverages a system that recommends symbol weights for the weighted symbol-counting clause selection scheme. Finally, in collaboration with coauthors, a system was developed that autonomously finds complementary strategies (heuristic configurations) and constructs robust strategy schedules that generalize well to previously unseen problems.

All research presented in this thesis has been published in peer-reviewed articles. The thesis presents these articles in their final published forms.

Keywords: automatic theorem proving, machine learning

Abstrakt

Automatické dokazování (ATP) pro predikátovou logiku prvního řádu řeší širokou škálu formalizovaných úloh v oblastech, jako jsou například matematika a verifikace softwaru. Automatické dokazovače založené na saturaci, zejména vysoce optimalizovaný dokazovač Vampire, představují špičku v oblasti ATP.

Současné moderní dokazovače používají různé heuristiky a úspěšnost dokazovače na daném problému do značné míry závisí na konfiguraci těchto heuristik. Konfigurace heuristik je tradičně úkolem expertních uživatelů. Množství dostupných konfigurací a nepředvídatelnost jejich účinků tento úkol komplikují. Proto je strojové učení (ML) slibnou alternativou k lidským odborným znalostem.

Tato disertační práce se zabývá aplikací ML v ATP založeném na saturaci. Výzkum představený v této práci se zaměřuje na zvýšení výkonu dokazovače Vampire pomocí tří přístupů. První přístup je založený na systému, který pro libovolný vstupní problém doporučí precedenci symbolů pro instanciaci zjednodušujícího uspořádání termů, která zajistí efektivní prořezávání grafu odvozených tvrzení. Druhý přístup využívá systém, který doporučuje váhy symbolů pro schéma výběru klauzulí založené na váženém počítáním symbolů. Nakonec byl ve spolupráci se spoluautory vyvinut systém, který samostatně vyhledává komplementární strategie (konfigurace heuristik) a sestavuje z nich robustní rozvrhy, které dobře generalizují na nové problémy.

Veškerý výzkum představený v této práci byl publikován v recenzovaných článcích. V práci jsou tyto články uvedeny v jejich konečné publikované podobě.

Klíčová slova: automatické dokazování, strojové učení

Contents

Chapter 1

Introduction

> In the Truth Mines, though, the tags weren't just references; they included complete statements of the particular definitions, axioms, or theorems the objects represented. The Mines were self-contained: every mathematical result that fleshers and their descendants had ever proven was on display in its entirety.
>
> Greg Egan, Diaspora

Throughout history, humankind has faced a wide array of challenges of an analytical nature. Mathematics rose to prominence as a discipline that studies, on an abstract level, the tools and approaches that demonstrate success in solving such challenges. Mathematical knowledge grew primarily in the directions motivated by natural sciences, engineering, and social sciences. At the same time, the fields of mathematics and logic fascinated humans as the only rigorous disciplines that can, at least in theory, be studied in isolation from the material world we inhabit.

In the nineteenth century, mathematics was considered a mature and important discipline of human intellectual endeavor. Meanwhile, advances in logic opened new possibilities for delimiting and resolving ambiguity in mathematics. The need for clarity in mathematics crystallized in the search for formally defined foundations of the discipline as a whole. In the twentieth century, Zermelo–Fraenkel set theory emerged as the standard foundational theory and classical first-order logic (FOL) was established as a suitable formalism for expressing the axioms and theorems of the theory [8].

In addition to a formal language that allows the expression of complex statements, FOL defines unambiguous notions of valid statement and sound reasoning. In practice, these notions have been applied in various subdomains of mathematics, such as arithmetic, group theory, or graph theory.

Although mathematics played a crucial role in establishing FOL, the expressive power makes FOL suitable for reasoning in other analytical domains. FOL and its extensions have been applied in, for example, software and hardware verification and synthesis [9, 10], common-sense reasoning [11], and legal reasoning [12, 13].

1.1 State of the Art

1.1.1 Reasoning in First-Order Logic

FOL is a general-purpose reasoning paradigm. In this section, I outline how FOL is used in reasoning and automatic theorem proving (ATP). I invite readers interested in more details to consult the relevant literature on logic [14, 15] and automated reasoning [16, 17].

In FOL, a reasoning task is defined by a *signature*, a set of *axioms*, and a *conjecture*. The signature is a set of non-logical (predicate and function) symbols. The axioms and the conjecture are first-order *sentences* (formulas with no free variable occurrences) over the signature. The set of axioms constitutes a *theory*; notable theories axiomatizable in FOL include set theory, group theory, and various fragments of arithmetic.

An *interpretation* specifies the semantics of each of the non-logical symbols of the signature and, by composition, each formula over the signature. If the conjecture is satisfied by every interpretation that satisfies the axioms, the axioms *logically entail* the conjecture. We then call the conjecture a *logical consequence* of the axioms, or a *theorem* of the theory defined by the axioms.[1]

Proof by contradiction is a common technique for certifying the validity of a theorem in mathematics. A direct counterpart exists in formal logic. The task of certifying entailment is reduced to certifying unsatisfiability of a set of formulas as follows: The set of axioms Γ entails conjecture C if the set of formulas $\Gamma \cup \{\neg C\}$ (where $\neg C$ is the negation of C) is unsatisfiable, that is, falsified by every interpretation. In the rest of this text, we call such a set of FOL formulas an *FOL problem*.

By applying De Morgan's laws, Skolemizing, pulling out quantifiers, and naming subformulas (Tseitin transformation), every FOL problem can be transformed into an equisatisfiable set (understood as a conjunction) of universally quantified *clauses* (disjunctions of first-order literals) in polynomial time [18]. We call such a representation of the problem *clause normal form (CNF)* and the process leading to it *clausification*.

Solving a CNF problem amounts to showing that the set of clauses is unsatisfiable. The standard way to define proof of unsatisfiability (or, more generally, proof of entailment) is based on an *inference calculus*—a set of inference rules. A proof, then, is a finite sequence of clauses, each of which is either an input clause or a clause derived from some (typically one or two) of the preceding clauses by an inference rule. If each inference rule in the calculus is *sound*, then each clause in the proof is a logical consequence of the input set of clauses. If the empty clause (a trivial contradiction) appears in the proof, the proof certifies the unsatisfiability of the input set of clauses. In this paradigm, proving unsatisfiability of a set of clauses can be realized as a search for (a derivation of) the empty clause.

[1] Strictly speaking, a theorem is a formula provable in some calculus. Gödel's completeness theorem shows that in FOL, provability in a suitable calculus coincides with logical entailment [14].

Complete inference calculi are of special interest: If an inference calculus is complete and the input clause set is unsatisfiable, then a proof of the unsatisfiability exists. In such a case, a proof (being a finite object) can be found, for example, by exhaustively enumerating clauses derivable from the input clauses. Enumerating the derivable clauses in a way that ensures a fast derivation of the empty clause (provided the input set of clauses is unsatisfiable) is the primary goal of saturation-based theorem proving.

1.1.2 Saturation-Based Theorem Proving

A *saturation-based theorem prover* structures the search for a contradiction (the empty clause) by incrementally expanding two sets of clauses. The *processed set* contains the clauses that have been used exhaustively as premises of the inference rules to derive new clauses. The clauses in the *unprocessed set* have been inferred from the input clauses and have not yet been used as premises of inferences.

The search starts by initializing the processed set as empty and populating the unprocessed set with the input clauses. In each iteration, a clause is selected from the unprocessed set. This clause, referred to as the *given clause*, is moved to the processed set. Then, all the clauses in the processed set are considered as potential partners of the given clause to act as premises of all the inference rules in the calculus implemented by the prover. The new clauses derived by the inferences are added to the unprocessed set.[2]

As soon as the empty clause is derived, the search terminates. The proof of the unsatisfiability of the input problem consists of the empty clause and all the clauses that contributed to its derivation. This shows the importance of clause selection for the efficiency of the proof search: If the prover always selected a clause that ends up in the final proof, the proof search would be highly efficient in terms of iterations of the saturation loop. However, because it is generally hard to recognize these clauses in advance, provers typically derive many clauses that do not contribute to the proof, which makes the clause selection heuristic an attractive target for optimization.

The calculus plays a crucial role in the proof search—it specifies which inferences are available in each iteration. Suppose the calculus is prolific (allowing relatively many inferences on a set of clauses). In that case, the proof search may slow down to a crawl quickly as the number of clauses derived in each iteration increases. For this reason, a successful inference calculus restricts inferences as much as possible while maintaining completeness.

For example, the superposition calculus [21] is parameterized by a *literal selection function*—a function that selects a subset of literals in each clause. Inferences are only made on selected literals. If the selection function is well-behaved, that is, it selects either a negative literal or all literals that are maximal with respect to a term ordering, the restriction preserves complete-

[2] More precisely, if a new clause is subsumed (and thus implied) by another clause derived previously, the new clause may be discarded without loss of completeness. This optimization, known as forward subsumption, is widely used in practice [16, 19, 20].

ness of the calculus. In specific scenarios, it is beneficial to further restrict the inferences in a way that forfeits completeness in exchange for a narrower and faster-converging proof search.

In summary, two main decision points guide saturation-based proof search:

Clause selection Which of the unprocessed clauses should be selected in each iteration of the saturation loop?

Inference restriction Which of the inferences should be applied to the selected clause?

In addition to these, the performance of a saturation-based prover may also depend on the configuration of the preprocessing of the input problem (clausification, premise selection [22, 23]), redundancy elimination (forward and backward subsumption), and other features, such as integration of a propositional satisfiability (SAT) solver [24].

Saturation is the state-of-the-art approach to FOL ATP. This has been demonstrated by the continued success of saturation-based theorem provers such as Vampire [25], E [26], and SPASS [27] in the CADE ATP System Competition (CASC) [28].

1.1.3 Machine Learning for Theorem Proving

Each of the main decision points in a theorem prover is governed by a *heuristic*—a procedure that aims to resolve the decision so as to optimize the prover's performance.[3] Many such heuristics can be configured to specialize the prover to a particular class of input problems.

For example, clause selection is often implemented by interleaving two priority queues ("age" and "weight"). The ratio in which these queues are interleaved, the so-called "age-weight ratio", is configurable. There is no universal best value of the ratio; the optimum depends on the input problem [30] and, in general, on the configuration of the other heuristics.

Traditionally, finding a good configuration of one or more heuristics in a prover is the work of an expert user that draws from their experience with ATP. Modern computing technology has opened new possibilities for the automation of expert decisions in various domains by *machine learning (ML)* [31, 32]. However, to apply ML to ATP, specific considerations need to be taken into account.

One of these considerations is the choice of a suitable representation of the input data. For example, we could try to predict, for an arbitrary input problem, an age-weight ratio that makes the prover likely to solve the problem in a given time limit. To train such a system using standard supervised learning techniques, we would need to map input problems to numeric vectors of fixed length—we might construct the numeric vector representation using human-designed features, such as symbol counts or term

[3] Standard performance measures include success rate and penalized average runtime (PAR) [29]. Each of them is parameterized by a runtime limit and a set of input problems.

walk counts [33], or automate feature extraction using a recursive neural network (RNN) [34, 35] or a graph neural network (GNN) [36, 37]. Each approach presents a specific trade-off between representation faithfulness and computational efficiency.

Since the satisfiability of a set of clauses does not depend on the particular naming of the symbols and variables, we may use an input problem representation that abstracts away from the symbol and variable names. Such representation ensures that we are learning general patterns based on common abstract substructures in the problems rather than idiosyncratic patterns based on naming conventions used by the users. GNNs implement such naming-agnostic representation naturally [36].

Our final goal is to optimize the performance of the prover in terms of, for example, the success rate on an input problem set with a fixed per-execution time limit. For training purposes, we typically need to approximate this measure by a *proxy task*—an optimization task that admits ML and is believed to correlate with the final performance measure.

For example, the task of estimating a good age-weight ratio can be implemented as a supervised learning task, provided that we have a ground truth available in the form of an assignment of age-weight ratios to training problems. To obtain such an assignment, we could, for example, evaluate each training problem with several age-weight ratios and assign the best of the observed values to the problem. This simplification establishes a proxy task: Predicting an age-weight ratio close to the best observed value is a machine-learnable proxy of solving the problem.

As an alternative to configuring one or more heuristics before the proof search starts (offline integration), a trained ML system may be integrated within the proof search procedure as a replacement for a man-made heuristic (online integration). For example, ML-based systems have been trained to perform clause selection directly [33, 34, 36–40]. Such ML systems evaluate all the inferred clauses to prioritize the most promising ones.

Although there is a growing body of research on ML for ATP [41], many avenues remain unexplored. The work presented in this thesis explores several novel approaches, as detailed in the next section.

1.2 Contributions

I designed and implemented several software systems that optimize the performance of the saturation-based prover Vampire. Each of these systems employs a novel approach to learn from executions of Vampire.

The results of my research have been peer-reviewed and published at various conferences and workshops. Chapters 2 to 6 present a selection of these publications. Each of the publications is included in its final published form and is self-contained.

1.2.1 Common Design Decisions

Although the experiments presented in the following have been performed using Vampire, the core techniques are applicable to any saturation-based automatic theorem prover. Furthermore, these techniques are potentially useful in other solvers in domains both within and outside of logic.

In all of the research presented in this thesis, I used the problems in the FOL fragment of Thousands of Problems for Theorem Provers (TPTP) [42], a library of benchmark problems. The problems in TPTP represent several target domains of ATP, such as mathematics and software verification, and have been encoded by various procedures. Each domain may use symbols and concepts that are specific to the domain, and each encoding procedure may use a different symbol naming scheme. For these reasons, the problems do not share a common signature—a symbol may represent different concepts across problems, and a concept may appear under different names. The lack of common signature in TPTP motivated some of the design decisions in my research, namely the preference for signature-agnostic problem representations.

Since Vampire represents the state of the art in ATP [43, 44] and since TPTP represents various domains of reasoning, the results can be understood as exploring and expanding the limits of FOL ATP in general. In all cases, the ML is based on learning from the results of executions of the target prover (Vampire) on problems from the target domain (FOL TPTP).

1.2.2 Symbol Precedence Recommenders

Vampire uses the superposition calculus [21, 25], which is parameterized by a simplification ordering on terms and a literal selection function. These parameters affect two restrictions of the inferences in the calculus: literal selection and equation orienting.

Literal selection Each of the inference rules targets some of the literals of the input clauses; for example, the resolution rule targets (and resolves) a positive literal in a clause and a complementary negative literal in another clause. Not all literals need to be considered as targets: The literal selection function selects one or more literals in each clause, and inferences are only performed on the selected literals. If the selection function is well-behaved (selecting either a negative literal or all literals

that are maximal with respect to the term ordering), the restriction does not compromise the completeness of the calculus [45].

Equation orienting The superposition inference rule generalizes the idea of rewriting terms by replacing equals with equals. The term ordering orients some of the equations, and each oriented equation is only used as a rewrite rule in one of the two possible directions. Informally, complex terms are replaced by simpler ones.

Both literal selection and equation orienting effectively prune the proof search without rendering it incomplete. Choosing a good term ordering for a given input problem can make the pruning more aggressive while still allowing for a short proof, which helps the saturation procedure find a proof quickly.

Knuth-Bendix ordering (KBO), Vampire's default term ordering, is parameterized by a *symbol precedence*—a permutation of the signature of the input problem. I explored two approaches to automatically finding a good symbol precedence to instantiate KBO for an arbitrary input problem.

In my first paper [1] (see chapter 2), I present a *symbol precedence recommender* that orders the symbols based only on six syntactic symbol features. In a follow-up work [2] (see chapter 3), I improved on this by training a GNN-based precedence recommender. In both cases, I trained the recommender to generate a symbol precedence that approximately minimizes the number of iterations the saturation loop takes to solve the input problem. This reduction generally corresponds to a low wallclock runtime needed to solve the problem and an improved probability of solving the problem under an arbitrary time limit.

1.2.3 Clause Selection Guidance

Clause selection is, arguably, the most crucial choice point in a saturation-based prover. The symbol-counting heuristic [30], one of the most common clause selection heuristics, prioritizes clauses that are small in the number of symbol and variable occurrences. In the weighted variant of this heuristic [46], the user specifies a weight for each of the symbols and the weight of a clause is calculated as the weighted sum of the symbol occurrence counts.

In the work included in chapter 4 [3], I modified Vampire to support the weighted symbol-counting clause selection heuristic and trained a GNN to propose a weight for each symbol of any given input problem. Thanks to the use of a GNN, this symbol weight recommender is signature-agnostic, which overcomes the challenge of misaligned signatures present in TPTP. The final evaluation showed that the trained heuristic solves 6.6 % more problems than the baseline.

1.2.4 Strategy Construction and Scheduling

The prominent automatic theorem provers expose various options that control the preprocessing and the proof search. In addition to simplification ordering

on terms and clause selection we discussed above, the options configure premise selection, subformula naming, redundancy elimination, etc. The configuration of all the options of a prover constitutes a *strategy*. Different problems may require different strategies—for example, aggressive premise selection is, in practice, necessary to solve problems with many axioms, while it rarely brings any benefit for small problems.

The research presented in chapters 5 and 6 combines two important tasks that are relevant to automatic theorem proving as well as to other forms of parameterized problem solving:

Automated algorithm configuration [47] is the task of finding a good configuration (strategy) for a distribution of problems.

Budgeted algorithm scheduling [48] is the task of finding a good algorithm schedule (a mapping from algorithms to runtime limits) given a distribution of problems, a portfolio of algorithms, and a runtime budget.

In our initial work on the topic [4] (see chapter 5), my co-authors and I generated a set of strong and mutually complementary strategies for the automatic theorem prover Vampire and combined these strategies into strong schedules. We introduced novel regularization techniques into the scheduling process to ensure good generalization of the resulting schedules. To the best of our knowledge, this is the first principled treatment of generalization of algorithm scheduling.

In a follow-up work [5] (see chapter 6), we investigated the possibility of specializing the schedules to homogeneous classes of problems.

Chapter 2

Learning Precedences from Simple Symbol Features

State-of-the-art automatic theorem provers, including Vampire, use the superposition calculus as the core of the reasoning procedure. Inferences of the superposition calculus are restricted by a simplification ordering on terms. Vampire implements the Knuth-Bendix ordering (KBO) and the lexicographic path ordering (LPO), two popular simplification orderings. Each of these orderings is parameterized by a symbol precedence—permutation of the symbols in the signature.

The standard heuristic methods implemented in Vampire construct the precedence by sorting the symbols by their arity or number of occurrences in the input problem. In the work presented below [1], a precedence is constructed using a combination of six syntactic symbol features that are easy to compute. Arity and number of occurrences are two of these features. This ensures that the system may be trained to represent, besides various other functions of the features, any of the standard heuristics.

Once the system has been trained, we can obtain a precedence for an input problem in two steps:

1. A pairwise preference regressor (Elastic-Net or gradient-boosted decision trees) predicts a preference value for each (ordered) pair of distinct symbols. This value expresses a degree of importance with which the first of the two symbols should precede the second.

2. A greedy algorithm constructs a precedence using the preference values. The algorithm approximately optimizes the cost of the precedence, that is, the sum of preference values of symbol pairs that appear in the precedence as subsequences.

The preference regressor is trained on runs of Vampire with KBO instantiated by random precedences. The training aims to minimize the time it takes Vampire to solve an input problem.

The main contribution of this work is a method of using pairwise ranking as a proxy task in ML for ATP (in this case, to train a symbol precedence recommender). This approach proved especially useful in my subsequent research presented in chapters 3 and 4.

Learning Precedences from Simple Symbol Features

Filip Bártek[a,b], Martin Suda[a]

[a]Czech Technical University in Prague – Czech Institute of Informatics, Robotics and Cybernetics. Jugoslávských partyzánů 1580/3, 160 00 Praha 6 – Dejvice, Czech Republic

[b]Czech Technical University in Prague – Faculty of Electrical Engineering. Technická 2, 166 27 Praha 6 – Dejvice, Czech Republic

Abstract

A simplification ordering, typically specified by a symbol precedence, is one of the key parameters of the superposition calculus, contributing to shaping the search space navigated by a saturation-based automated theorem prover. Thus the choice of a precedence can have a great impact on the prover's performance. In this work, we design a system for proposing symbol precedences that should lead to solving a problem quickly. The system relies on machine learning to extract this information from past successful and unsuccessful runs of a theorem prover over a set of problems and randomly sampled precedences. It uses a small set of simple human-engineered symbol features as the sole basis for discriminating the symbols. This allows for a direct comparison with precedence generation schemes designed by prover developers.

Keywords

saturation-based theorem proving, simplification orderings, symbol precedences, machine learning

1. Introduction

Modern saturation-based automated theorem provers (ATPs) such as E [1], SPASS [2] or Vampire [3] use the superposition calculus [4] as their underlying inference system. Superposition is built around the paramodulation inference [5] crucially constrained by simplification ordering on terms and literals, which is supplied as a parameter of the calculus. Both of the two main classes of simplification orderings used in practice, i.e., the Knuth-Bendix Ordering [6] and the Lexicographic Path Ordering [7], are mainly determined by a *symbol precedence*, a (partial) ordering on the signature symbols.[1]

While the superposition calculus is known [9] to be refutationally complete for any simplification ordering, the choice of the precedence may have a significant impact on how long it takes to solve a given problem. In a well-known example, prioritizing in the precedence the predicates introduced during the Tseitin transformation of an input formula [10] exposes the corresponding literals to resolution inference during early stages of the proof search, with the effect of essentially undoing the transformation and thus threatening with an exponential blow-up that the transformation is designed to prevent [11]. ATPs typically offer a few heuristic

PAAR 2020: Seventh Workshop on Practical Aspects of Automated Reasoning, June 29–30, 2020, Paris, France (virtual)

filip.bartek@cvut.cz (F. Bártek); martin.suda@cvut.cz (M. Suda)

0000-0002-1822-2651 (F. Bártek); 0000-0003-0989-5800 (M. Suda)

[1]KBO is further parameterized by symbol weights, but our reference implementation in Vampire [3] uses for efficiency reasons only weights equal to one [8] and so we do not consider this parameter here.

schemes for generating the symbol precedences. For example, the successful `invfreq` scheme in E [12] orders the symbols by the number of occurrences in the input problem, prioritizing symbols that occur the least often for early inferences. Experiments with random precedences have shown that the existing schemes often fail to come close to the optimum precedence [13], revealing there is a large potential for further improvements.

In this work, we design a system that, when presented with a First-Order Logic (FOL) problem, proposes a symbol precedence that will likely lead to solving the problem quickly. The system relies on the techniques of supervised machine learning and extracts such theorem-proving knowledge from successful (and unsuccessful) runs of the Vampire theorem prover [3] when run over a variety of FOL problems equipped with randomly sampled symbol precedences. We assume that by learning to solve already solvable problems quickly, the acquired knowledge will generalize and help solving problems previously out of reach. As a first step in a more ambitious project, we focus here on representing the symbols in a problem by a fixed set of simple human-engineered features (such as the number of occurrences used by `invfreq` scheme mentioned above)[2] and, to simplify the experimental setup, we restrict our attention to learning precedences for predicate symbols only.[3]

Learning to predict good precedences poses several interesting challenges that we address in this work. First, it is not immediately clear how to characterize a precedence, a permutation of a finite set of symbols, by a real-valued feature vector to serve as an input for a learning algorithm. Additionally, to be able to generalize across problems we need to do it in a way which does not presuppose a fixed signature size. There is also a complication, when sampling different problems, that some problems may be easy to solve for almost every precedence and others hard. In theorem proving, running times typically vary considerably. Finally, even with a regression model ready to predict the prover's performance under a particular precedence π, we still need to solve the task of finding an optimum precedence π^* according to this model, which cannot be simply solved by enumerating all the permutations and running the prediction for each due to their huge number.

Our way of addressing the above sketched challenges lies in using *pairwise symbol preferences* to characterize a precedence, normalizing the target prover run times on a per problem basis, and in the use of "second-order" learning of the preferences for symbols abstracted by their features. These concepts are introduced in Section 3 and later formalized in Section 4. Section 5 presents the results of our experimental evaluation of the proposed technique over the TPTP [14] benchmark. We start our exposition by fixing the notation and basic concepts in Section 2.

2. Preliminaries

We assume that the reader is familiar with basic concepts used in first order logic (FOL) theorem proving. We use this section to recall and formalize the key notions relevant for our work.

Problem A *(first-order) problem* is a pair $P = (\Sigma, Cl)$, where $\Sigma = (s_1, s_2, \ldots, s_n)$ is a list of (predicate and function) symbols called the *signature*, and Cl is a set of first-order clauses

[2] Automatic feature extraction using neural networks is planned for future work.

[3] Our theoretical considerations, however, apply equally to learning function symbol precedences.

built over the symbols of Σ.

The problem is either given directly by the user or could be the result of clausifying a general FOL formula φ, in which case we know which of the symbols were introduced during the clausification (namely during Tseitin transformation and skolemization; see e.g. Nonnengart and Weidenbach [15]) and which occurred in the conjecture (if it was present).

Precedence Given a problem $P = (\Sigma, Cl)$ with $\Sigma = (s_1, s_2, \ldots, s_n)$, a precedence π_P is a permutation, i.e. a bijective mapping, of the set of indices $\{1, \ldots, n\}$. A precedence π_P *determines* a (total) ordering on Σ as follows: $s_{\pi_P(1)} < s_{\pi_P(2)} < \ldots < s_{\pi_P(n)}$.

Simplification Orderings are orderings on terms used to parameterize the superposition calculus [4] employed by modern saturation-based theorem provers. The two classes of simplification orderings most commonly used in practice, the Knuth-Bendix Orderings [6] and the Lexicographic Path Orderings [7], are both defined in terms of a user-supplied (possibly partial) ordering $<$ on the given problem's signature Σ. In this work, we assume that the theorem prover uses a simplification ordering from one of these two classes relying on the ordering on Σ determined by a precedence π_P to construct such a simplification ordering.

Performance measure A saturation-based ATP *solves* a problem $P = (\Sigma, Cl)$ (under a particular fixed strategy and a determined symbol precedence π_P) by either

- deriving from Cl a contradiction in the form of the empty clause, in which case P is shown *unsatisfiable*, or
- finitely saturating the set of clauses Cl without deriving the contradiction, in which case P is shown *satisfiable*.[4]

In both cases, we take the number of iterations of the employed saturation algorithm (see, e.g., Riazanov and Voronkov [16] for an overview) as a measure of the effort that the ATP took to solve the problem. We refer to this measure as the *abstract solving time* and denote it $ast(P, \pi_P)$.[5]

In practice, an ATP can also run out of resources, typically out of the allocated time. In that case, the abstract solving time is *undefined*: $ast(P, \pi_P) = \bot$. While it may happen that running an ATP with the same problem and symbol precedence two times yields a different result each time (namely succeeding one time and failing another time), such cases are rare and we ensure they do not interfere with the learning process by caching the results.

Order matrix Given a permutation π of the set of indices $\{1, \ldots, n\}$, the order matrix $O(\pi)$ is a binary matrix of size $n \times n$ defined in the following manner:

$$O(\pi)_{i,j} = [\![\pi^{-1}(i) < \pi^{-1}(j)]\!] ,$$

[4] We assume a *refutationally complete* calculus and saturation strategy.

[5] The advantage of using abstract solving time is that it does not depend on the hardware used for the computation.

where we use $[\![P]\!]$ to denote the Iverson bracket [17] applied to a proposition P, evaluating to 1 if P is true, and 0 otherwise. In other words, for a symbol precedence π_P, $O(\pi_P)_{i,j} = 1$ if the precedence π_P orders the symbol s_i before the symbol s_j, and $O(\pi_P)_{i,j} = 0$ otherwise.

Flattened matrix Given a matrix M of size $n \times n$, $\overrightarrow{M}$ is the vector of length n^2 obtained by flattening M:

$$\overrightarrow{M}_{(i-1)n+j} = M_{i,j}$$

for every $i, j \in \{1, \ldots, n\}$. For our use the exact way of mapping the matrix elements to the vector indices is not important. We mostly just need a vector representation of the data contained in the given matrix to have access to the dot product operation.

Linear regression is an approach to modeling the relationship between scalar *target* values $y_i \in \mathbb{R}$ and one or more *input* variables $\mathbf{x}_i = (x_i^1, \ldots, x_i^k)$, $i = 1, \ldots, n$. The relationship is modeled using a linear predictor function:

$$\hat{y}_i = \mathbf{x}_i \cdot \mathbf{w} + b,$$

whose unknown model parameters $\mathbf{w} \in \mathbb{R}^k$ and $b \in \mathbb{R}$ are estimated from the data. We call the vector $\mathbf{w}$ the *coefficients* of the model and b the *intercept*. Most commonly, the parameters are picked to minimize the so-called mean squared error:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2,$$

but other norms are also possible [18].

Basic assumptions For the discussion that follows, we assume a fixed ATP that uses the superposition calculus with a simplification ordering parameterized by a symbol precedence. While the practical experiments described in Section 5 use the ATP Vampire [3], the model architecture does not assume a particular ATP and is compatible with any superposition-based ATP such as E [1], SPASS [2] or Vampire. Within the prover a particular saturation strategy is fixed including a time limit.

3. General considerations and overview

The aim of this work is to design a system that learns to suggest good symbol precedences to an ATP from observations of the ATP's performance on a class $\mathcal{P}_{train}$ of problems with randomly sampled precedences. Given a problem $P = (\Sigma, Cl)$ with $|\,\Sigma\,| = n$, we consider a precedence π_P good, if it leads to a low $ast(P, \pi_P)$ among the $n!$ possible precedences for P. Note that for problems with a signature with more than a few symbols, repeatedly running the prover with random precedences represents an effectively infinite source of training data.

Ideally, we would like to learn general theorem proving knowledge, not too dependent on $\mathcal{P}_{train}$, which could be later explained and compared to precedence generation schemes manually designed by the prover developers. Let us quickly recall one such scheme, already mentioned in the introduction, called `invfreq` in E [1]. The prover's manual [12] explains:

> Sort symbols by frequency (frequently occurring symbols are smaller).

What is common to basically all manually designed schemes, is that they pick a certain scalar property of symbols (here it is the symbol frequency, i.e. the number of occurrences of the symbol in the given problem) and obtain a precedence by *sorting* the symbols using that property.

Decomposing We might want our system to also learn a certain property of symbols and use sorting to generate and suggest a precedence. However, it is not clear how to "extract" such property from the observed data, since we only have access to the target values for full precedences. Our idea for "decomposing" these values into pieces that somehow relate to individual symbols (and can thus be "transferred" across problems) is to take a detour using symbol pairs: we assume that the performance of the ATP on P given π_P, i.e. our measure $ast(P, \pi_P)$, can be predicted from a sum of individual contributions corresponding to facts of the form

$$\pi_P \text{ orders the symbol } s_i \text{ before the symbol } s_j.$$

This is in line with how a prover developer could reason about a precedence generating scheme: Even when it is not clear how good or bad a symbol is in absolute terms, one might have an intuition that a symbol from a certain class should preferably come before a symbol from another class in the precedence (e.g., symbols introduced during clausification should typically be smaller than others) and assign some weight to this piece of intuition.

In Section 4.2 we formalize this idea using the notion of *preference matrix* and show how, for each problem in isolation, such preference matrix can be obtained in the form of coefficients learned by linear regression.

Learning across problems Symbol preferences learned on a particular problem are inherently tied to that problem and do not immediately carry over to other problems. The main reason for this is that symbols themselves only appear in the context of a particular problem.[6] That is why we resort to representing symbols by their *features* (cf. Section 4.3.2) when aggregating the learned preferences across different problems. This is in more detail explained further below and, formally, in Section 4.3.

We also strive to ensure that the preference values across problems have possibly the same magnitude. Note that $ast(P, \pi_P)$ may vary a lot for a fixed problem P but all the more so across problems. To obtain commensurable values, we normalize (see Section 4.1) the prover performance data on a per problem basis before learning the preferences. Normalization also deals with supplying a concrete value to those runs which did not finish, i.e. have $ast(P, \pi_P) = \bot$.

[6] On certain benchmarks, such as those coming from translations of mathematical libraries [19], symbols maintain identity and meaning across individual problems. However, since our goal in this work is to learn general theorem proving knowledge, we do not use the assumption of aligned signatures.

"Second-order" regression Once the symbols are abstracted by their feature vectors, we can collect symbol preferences from all the tested problems and turn this collection into another regression task. Note that at this moment, the preferences, which were obtained as the coefficients learned by linear regression, themselves become the regression target. Thus, in a certain sense, we now do second-order learning. It should be stressed though, that while the learning of the preferences *requires* a linear regression model by design, this second-order regression does not need to be linear and more sophisticated models can be experimented with.

The details of this step are given in Section 4.3.

Preference prediction and optimization Once the second-order model has been learned, we can predict preferences for any pair of symbols based on their feature vectors and thus also predict, given a problem P, how many steps will a prover require to solve it using a particular precedence π_P. (For this second step, we reverse the idea of decomposition: we sum up those predicted preferences that correspond to pairs of symbols s_i, s_j such that π_P orders the symbol s_i before the symbol s_j – see Section 4.4 for details).

Having access to an estimate of performance for each precedence π_P, the final step is to look for a precedence π_P^* that ideally minimizes the predicted performance measure over all the $n!$ possible precedences on P's signature. Since finding the true optimum could be computationally hard, we resort to using an approximation algorithm by Cohen et al. [20].

The algorithm is recalled in Section 4.4.2.

4. Architecture

4.1. Values of precedences

We define the base cost value $cost_{base}(P, \pi_P)$ of precedence π_P on problem P according to the outcome of the proof search configured to use this precedence:

- If the proof terminates successfully, $cost_{base}(\pi_P)$ is the number of iterations of the saturation loop started during the proof search: $cost_{base}(\pi_P) = ast(P, \pi_P)$.
- If the proof search fails (meaning that $ast(P, \pi_P) = \bot$), then $cost_{base}(\pi_P)$ is the maximum number of saturation loop iterations encountered in successful training proof searches on this problem: $cost_{base}(\pi_P) = \max_{\pi'_P \in \Pi_P^+} ast(P, \pi'_P)$, where Π_P^+ is the set of all training precedences on problem P that yield a successful proof search.

We further normalize the cost values by the following operations:

1. Logarithmic scaling: For each solvable problem, running proof search with uniformly random predicate precedences reveals a distribution of abstract solving times on successful executions. Examining these distributions for various problems suggests that they are usually approximately log-normal. To make further scaling by standardization reasonable, we first transform the base costs by taking their logarithm.
2. Standardization: Independently for each problem, we apply an affine transformation so that the resulting cost values have the mean 0 and standard deviation 1. This ensures that the values are comparable across problems.

Let $cost_{std}(\pi_P)$ denote the resulting cost value of precedence π_P after the scaling and standardization.

4.2. Problem preference matrix learning

Given a problem P with n symbols, a *preference matrix* W_P is any matrix over $\mathbb{R}$ of size $n \times n$. We define the *proxy cost of precedence* π_P *under preference* W_P to be the sum of the preference values $W_{P_{i,j}}$ of all symbol pairs s_i, s_j ordered by π_P such that s_i comes before s_j:

$$cost_{proxy}(\pi_P, W_P) = \sum_{i,j} [\![\pi_P^{-1}(i) < \pi_P^{-1}(j)]\!] \, W_{P_{i,j}} = \overrightarrow{O(\pi_P)} \cdot \overrightarrow{W_P}$$

where $\overrightarrow{O(\pi_P)} \cdot \overrightarrow{W_P}$ is the dot product of the flattened matrices $O(\pi_P)$ and W_P.

For any given problem we can uniformly sample precedences π_P to form the training set $T = \{(\pi_P^1, cost_{std}(\pi_P^1)), (\pi_P^2, cost_{std}(\pi_P^2)), \ldots, (\pi_P^m, cost_{std}(\pi_P^m))\}$. Having such training set allows us to find a vector $\overrightarrow{W_P}$ that minimizes the mean square error

$$\frac{1}{m} \sum_{(\pi_P, cost_{std}(\pi_P)) \in T} (cost_{proxy}(\pi_P, W_P) - cost_{std}(\pi_P))^2$$

by linear regression.

Minimizing the mean square error directly may lead to overfitting to the training set, especially in problems whose signature is relatively large in comparison to the size of the training set. To improve generalization, we use the Lasso regression algorithm [21] instead of standard linear regression. We use cross-validation to set the value of the regularization hyperparameter.[7]

Another reason to use the Lasso algorithm is that it performs regularization by imposing a penalty on coefficients with large absolute value, effectively shrinking the coefficients that correspond to symbol pairs whose mutual order does not affect the $cost_{std}(\pi_P)$. We can use this property to interpret the absolute value of preference value as a measure of the importance of a given symbol pair.

In the following sections we assume that the preference matrix W_P we find by Lasso regression yields $cost_{proxy}$ that approximates $cost_{std}$ well.

4.3. General preference matrix learning

We proceed to cast the task of finding a good preference matrix W_P for an arbitrary problem as a regression on feature vector representations of symbol pairs. To accomplish this we need to be able to represent each pair of symbols by a feature vector and to know target preference values for pairs of symbols in a training problem set.

4.3.1. Target preference values

For each problem P in the training problem set, we find a problem preference matrix by the method outlined in Section 4.2. The target value of an arbitrary pair of symbols s_i, s_j in P is $W_{P_{i,j}}$.

[7] See the model `LassoCV` in the machine learning library scikit-learn [22].

4.3.2. Symbol pair embedding

We represent each symbol by a numeric *feature vector* that consists of the following components: symbol arity, the number of symbol occurrences in the problem, the number of clauses in the problem that contain at least one occurrence of the symbol, an indicator of occurrence in a conjecture clause, an indicator of occurrence in a unit clause, and an indicator of being introduced during clausification. This choice of symbol features is motivated by the fact that they are readily available in Vampire and that they suffice as a basis for common precedence generation schemes, such as the `invfreq` scheme. We denote the feature vector corresponding to symbol s as $fv(s)$.

We represent a pair of symbols s, t by the concatenation of their feature vectors $[fv(s), fv(t)]$.

4.3.3. Training data

The general preference regressor is trained on samples of the following structure:

- the input: $[fv(s_i), fv(s_j)]$ – the embedding of a symbol pair s_i, s_j in problem P,
- the target: $W_{Pi,j}$ – an element of the preference matrix we learned for problem P corresponding to the symbol pair (s_i, s_j).

We sample problem P from the training problem set with uniform probability.

Thanks to how W_P is constructed (see Section 4.2), preference values close to 0 are associated with symbol pairs whose mutual order has little effect on the outcome of the proof search. To focus the training on the symbol pairs whose order does matter, we weight the samples by the absolute value of the target. More precisely, given a problem P, the probability of sampling the symbol pair i, j is proportional to the absolute target value $|W_{Pi,j}|$. Experiments have shown that using sample weighting improves the performance of the resulting model (see Section 5.2).

We denote the trained model as M and its prediction of the preference value of the symbol pair s_i, s_j as $M([fv(s_i), fv(s_j)])$.

4.4. Precedence construction

When presented a new problem $P = (\Sigma, Cl)$, we propose a symbol precedence by taking the following steps:

1. Estimate a preference matrix $\widehat{W_P}$.
2. Construct a precedence $\widehat{\pi_P}$ that approximately minimizes $cost_{proxy}(\widehat{\pi_P}, \widehat{W_P})$.

4.4.1. Preference matrix construction

To construct a preference matrix $\widehat{W_P}$ for a new problem P, we evaluate the general preference regressor on the feature vectors of all symbol pairs in P. More specifically,

$$\widehat{W_P}_{i,j} = M([fv(s_i), fv(s_j)])$$

for all $s_i, s_j \in \Sigma$.

At this moment, one can use $\widehat{W_P}$ to estimate the cost of an arbitrary symbol precedence.

4.4.2. Precedence construction from preference matrix

The remaining task is, given a preference matrix $\widehat{W_P}$, to find a precedence $\widehat{\pi_P}$ that minimizes $cost_{proxy}(\widehat{\pi_P}, \widehat{W_P})$. Since this task is NP-hard in general [20], we rely in this work on a greedy 2-approximation algorithm proposed by Cohen et al. [20]. The rest of this section provides a brief description of the algorithm.

The algorithm maintains a partially constructed symbol precedence $\mathbf{p} \in \mathbb{N}^*$ (a finite sequence over $\mathbb{N}$; initially empty), a set of available symbols $\Sigma_{avail} \subseteq \Sigma$ (initially the whole Σ) and a *potential value* for each of the symbols $c : \Sigma_{avail} \to \mathbb{R}$. The potential value of a symbol corresponds to the relative increase in proxy cost associated with selecting the symbol as the next to append to the partial precedence:

$$c(s_i) = \sum_{s_j \in \Sigma_{avail}} \widehat{W_P}_{i,j} - \sum_{s_j \in \Sigma_{avail}} \widehat{W_P}_{j,i}$$

In each iteration, a symbol s_i with the smallest potential is selected from Σ_{avail}. This symbol is removed from Σ_{avail} and its index i is appended to the partial precedence $\mathbf{p}$. The potentials of the remaining symbols in Σ_{avail} are updated. This process is repeated until all symbols have been selected, yielding the final $\mathbf{p}$ as $\widehat{\pi_P}$.

5. Evaluation

5.1. Setup

Since the simplification orderings under consideration (LPO and KBO) never use the symbol precedence to compare a predicate symbol with a function symbol, we can break down the symbol precedence into a predicate precedence and a function precedence. In this paper, we restrict our attention to predicate precedences, leaving function symbols to be ordered by the `invfreq` scheme. A more thorough evaluation of both predicate and function precedences and their interaction is left for future work.

We use problems from the TPTP library v7.2.0 [14] for the evaluation. Let $\mathcal{P}_{train}$ be the set of all FOL and CNF problems in TPTP with at most 200 predicate symbols such that at least 1 out of 24 random predicate precedences leads to a successful proof search ($|\mathcal{P}_{train}| = 8217$). Let $\mathcal{P}_{test}$ be the set of all FOL and CNF problems in TPTP with at most 1024 predicate symbols ($|\mathcal{P}_{test}| = 15751$). In each of 5 evaluation iterations (splits), we sample 1000 training problems from $\mathcal{P}_{train}$ and 1000 test problems from $\mathcal{P}_{test}$ uniformly in a way that ensures that the sets do not overlap. We repeat the evaluation 5 times to evaluate the stability of the training.

On each training problem we run Vampire with 100 uniformly random predicate precedences and a strategy fixed up to the predicate precedence.[8] We limit the time to 10 seconds per execution which is in our experience with Vampire sufficient to exhibit interesting behavior. Note that we use a customized version of Vampire to extract a symbol table from each of the problems.[9]

[8] Time limit: 10 seconds, memory limit: 8192 MB, literal comparison mode: predicate, function symbol precedence: `invfreq`, saturation algorithm: discount, age-weight ratio: 1:10, AVATAR: disabled.

[9] https://github.com/filipbartek/vampire/tree/926154f2

Table 1
Experiment results

Case	Successes out of 1000 per split						
	0	1	2	3	4	Mean	Std
Best of 10 random	514	499	509	511	476	501.8	13.85
`invfreq`	494	472	480	481	452	475.8	13.83
Elastic-Net	484	471	479	470	454	471.6	10.21
Gradient Boosting	473	462	475	475	439	464.8	13.78
Elastic-Net without sample weighting	475	454	465	459	453	461.2	8.11
Random	454	455	457	456	430	450.4	10.25

After we fit a preference matrix on each of the training problems (see Section 4.2), we create a batch of 10^6 symbol pair feature vectors with target values to train the general preference regressor (see Section 4.3). We evaluate the trained model by running Vampire on the test problem set with predicate precedences proposed by the trained model, counting the number of successfully solved problems.

A collection of scripts created for the experimental evaluation can be found in the Git repository at https://github.com/filipbartek/vampire-ml/tree/75c693f3. The measurements presented below can be performed by running the script map-reduce/paar2020/run.sh.

5.2. Experimental results

We trained two types of general preference regressors (see Section 4.3):

- Elastic-Net – a linear regression model with L1 and L2 norm regularization; see `ElasticNetCV` in Pedregosa et al. [22]
- Gradient Boosting regressor – see `GradientBoostingRegressor` in Pedregosa et al. [22]

We compared the performance of the regressors with three baseline precedence generation schemes – random precedence, best of 10 random precedences and the `invfreq` scheme. Table 1 shows the results of evaluation on 1000 problems for 5 random choices of training and test problem set (splits).

The case "Elastic-Net without sample weighting" shows the effect of sampling the symbol pairs uniformly. Inspection of the trained feature coefficients reveals that the fitting ends up with an all-zero feature weight vector on splits 1 and 2, signifying a complete failure to learn on these training sets.

Using Elastic-Net for general preference prediction on average nearly matches the performance of Vampire with the `invfreq` precedence scheme. While Elastic-Net performs significantly better than a random precedence generator, it still performs significantly worse than a generator that, given a problem, tries 10 random precedences and chooses the best of these. This suggests that there is space for improvement, possibly with a more sophisticated, non-linear model. Plugging in a Gradient Boosting regressor does not show immediate improvement so more elaborate feature extraction may be necessary.

Table 2
Elastic-Net feature coefficients after fitting on each of the 5 training sets of 1000 problems and on the whole $\mathcal{P}_{train}$. "Frequency" is the number of occurrences of the symbol in the problem. "Unit frequency" is the number of clauses in the problem that contain at least one occurrence of the symbol.

Training set	Left symbol			Right symbol		
	Arity	Frequency	Unit frequency	Arity	Frequency	Unit frequency
0		−0.01		−0.98	0.01	
1		−0.48			0.08	0.44
2			−0.64		0.36	
3	0.88	−0.03	0.01		−0.03	0.05
4		−0.62			0.30	0.07
$\mathcal{P}_{train}$			−0.57		0.43	

5.3. Feature coefficients

Since Elastic-Net is a linear regression model, we can easily inspect the coefficients it assigns to the input features (see Section 4.3.2). In each of the five splits, the final coefficients of the three indicator features (namely the indicators of presence in a conjecture clause, presence in a unit clause and being introduced during clausification) are 0. Table 2 shows the fitted non-zero coefficients of the remaining features. The coefficients were scaled so that their absolute values sum up to 1. Note that scaling the coefficients by a constant does not affect the precedence constructed using the greedy algorithm presented in Section 4.4.2.

It is worth pointing out that the regressor fitted on the whole $\mathcal{P}_{train}$ and on the training sets 1, 2 and 4 assigns a high preference value to symbol pairs (s, t) such that t has a higher frequency and unit frequency than s. Since unit frequency is positively correlated with frequency, minimizing $cost_{proxy}$ using this fitted regressor is consistent with the `invfreq` precedence generating scheme (ordering the symbols by frequency in descending order). Similarly, the model fitted on training sets 0 and 3 corresponds to ordering the symbols by arity in ascending order.

6. Conclusion

This paper is, to the best of our knowledge, a first attempt to use machine learning for proposing symbol precedences for an ATP. This appears to be a potentially highly rewarding task with an access to effectively unlimited amount of training data generated on demand. Nevertheless, the journey from evaluating the prover on random precedences to proposing a good precedence when presented with a new problem is not straightforward and several conceptual gaps need to be bridged to connect these two tasks algorithmically.

In this paper, we proposed a connection using the concept of pairwise symbol preferences that, as we have shown, can be learned as the coefficients of a linear regression model for which an order matrix provides the features of a precedence understood as a permutation. In a second stage, in which symbols are abstracted by their features, the preferences themselves become regression targets.

In our initial experiments reported in this paper, the performance of our system does not

yet reach that of the human-designed heuristic `invfreq`. We believe, however, that further improvements are possible by using a more advanced regression model for the second stage and/or by further hyper-parameter tuning (e.g. of the Gradient Boosting model). Ultimately, we expect to gain the most by using a richer set of symbol features, ideally automatically extracted from the problems using graph neural networks [23].

Acknowledgments

Supported by the ERC Consolidator grant AI4REASON no. 649043 under the EU-H2020 programme, the Czech Science Foundation project 20-06390Y and the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37.

References

[1] S. Schulz, S. Cruanes, P. Vukmirović, Faster, higher, stronger: E 2.3, in: P. Fontaine (Ed.), Automated Deduction - CADE 27, number 11716 in Lecture Notes in Computer Science, Springer, 2019, pp. 495–507. doi:10.1007/978-3-030-29436-6_29.

[2] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, SPASS version 3.5, in: R. A. Schmidt (Ed.), Automated Deduction - CADE-22, number 5663 in Lecture Notes in Computer Science, 2009, pp. 140–145. doi:10.1007/978-3-642-02959-2_10.

[3] L. Kovács, A. Voronkov, First-order theorem proving and Vampire, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–35. doi:10.1007/978-3-642-39799-8_1.

[4] R. Nieuwenhuis, A. Rubio, Paramodulation-based theorem proving, in: J. A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning (in 2 volumes), Elsevier and MIT Press, 2001, pp. 371–443. doi:10.1016/b978-044450813-3/50009-6.

[5] G. Robinson, L. Wos, Paramodulation and theorem-proving in first-order theories with equality, in: J. H. Siekmann, G. Wrightson (Eds.), Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970, Springer Berlin Heidelberg, Berlin, Heidelberg, 1983, pp. 298–313. doi:10.1007/978-3-642-81955-1_19.

[6] D. E. Knuth, P. B. Bendix, Simple word problems in universal algebras, in: J. H. Siekmann, G. Wrightson (Eds.), Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970, Springer Berlin Heidelberg, Berlin, Heidelberg, 1983, pp. 342–376. doi:10.1007/978-3-642-81955-1_23.

[7] S. N. Kamin, J. Lévy, Two generalizations of the recursive path ordering, 1980. URL: http://www.cs.tau.ac.il/~nachumd/term/kamin-levy80spo.pdf, unpublished letter to Nachum Dershowitz.

[8] L. Kovács, G. Moser, A. Voronkov, On transfinite Knuth-Bendix orders, in: N. Bjørner, V. Sofronie-Stokkermans (Eds.), Automated Deduction – CADE-23, number 6803 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 384–399. doi:10.1007/978-3-642-22438-6_29.

[9] L. Bachmair, H. Ganzinger, Rewrite-based equational theorem proving with selection

and simplification, Journal of Logic and Computation 4 (1994) 217–247. doi:10.1093/logcom/4.3.217.

[10] G. S. Tseitin, On the complexity of derivation in propositional calculus, in: J. H. Siekmann, G. Wrightson (Eds.), Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970, Springer Berlin Heidelberg, Berlin, Heidelberg, 1983, pp. 466–483. doi:10.1007/978-3-642-81955-1_28.

[11] G. Reger, M. Suda, A. Voronkov, New techniques in clausal form generation, in: C. Benzmüller, G. Sutcliffe, R. Rojas (Eds.), GCAI 2016. 2nd Global Conference on Artificial Intelligence, volume 41 of *EPiC Series in Computing*, EasyChair, 2016, pp. 11–23. URL: https://easychair.org/publications/paper/XncX. doi:10.29007/dzfz.

[12] S. Schulz, E 2.4 User Manual, 2019. URL: http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/E_DOWNLOAD/V_2.4/eprover.pdf.

[13] G. Reger, M. Suda, Measuring progress to predict success: Can a good proof strategy be evolved?, in: AITP 2017, 2017, pp. 20–21. URL: http://aitp-conference.org/2017/aitp17-proceedings.pdf.

[14] 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. doi:10.1007/s10817-017-9407-7.

[15] A. Nonnengart, C. Weidenbach, Computing small clause normal forms, in: J. A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning (in 2 volumes), Elsevier and MIT Press, 2001, pp. 335–367. URL: https://doi.org/10.1016/b978-044450813-3/50008-4. doi:10.1016/b978-044450813-3/50008-4.

[16] A. Riazanov, A. Voronkov, Limited resource strategy in resolution theorem proving, Journal of Symbolic Computation 36 (2003) 101–115. doi:10.1016/S0747-7171(03)00040-3.

[17] K. E. Iverson, A Programming Language, John Wiley & Sons, Inc., New York, NY, USA, 1962. URL: https://dl.acm.org/doi/book/10.5555/1098666.

[18] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, Springer Series in Statistics, Springer New York Inc., New York, NY, USA, 2009. doi:10.1007/b94608.

[19] C. Kaliszyk, J. Urban, MizAR 40 for Mizar 40, Journal of Automated Reasoning 55 (2015) 245–256. doi:10.1007/s10817-015-9330-8.

[20] W. W. Cohen, R. E. Schapire, Y. Singer, Learning to order things, Journal Of Artificial Intelligence Research 10 (1999) 243–270. doi:10.1613/jair.587. arXiv:1105.5464.

[21] R. Tibshirani, Regression shrinkage and selection via the Lasso, Journal of the Royal Statistical Society. Series B (Methodological) 58 (1996) 267–288. URL: http://www.jstor.org/stable/2346178.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830. URL: https://scikit-learn.org/.

[23] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P. S. Yu, A comprehensive survey on graph neural networks, IEEE Transactions on Neural Networks and Learning Systems (2020) 1–21. doi:10.1109/tnnls.2020.2978386.

Chapter 3

Neural Precedence Recommender

A GNN can be used to extract task-specific features of various syntactic elements of an input problem, including the predicate and function symbols. In the work presented below [2], I trained a GNN to predict a score for each symbol in the signature such that ordering the symbols by their scores yields a good symbol precedence for KBO in Vampire.

A straightforward way to determine the best precedence for a given problem involves evaluating all possible precedences (permutations of the signature of the problem). Such exhaustive evaluation is prohibitively expensive for all but the smallest problems. On the other hand, we can easily compare an arbitrary pair of precedences. Following this observation, we train the symbol score predictor on the proxy task of (pairwise) ranking of precedences.

In summary, the symbol precedence recommender is trained on three nested proxy tasks:

1. Predicting a score of a symbol
2. Predicting a score of a precedence
3. Pairwise ranking of precedences

The experiments presented in the paper demonstrate that a trained precedence recommender outperforms the baseline by more than 4 % in the number of solved problems. This result confirms that the proxy tasks are aligned with the main task of solving as many problems as possible and demonstrates the potential of customizing symbol precedence automatically.

Neural Precedence Recommender

Filip Bártek[1,2] and Martin Suda[1]

[1] Czech Institute of Informatics, Robotics and Cybernetics
[2] Faculty of Electrical Engineering
Czech Technical University in Prague, Czech Republic
{filip.bartek,martin.suda}@cvut.cz

Abstract. The state-of-the-art superposition-based theorem provers for first-order logic rely on simplification orderings on terms to constrain the applicability of inference rules, which in turn shapes the ensuing search space. The popular Knuth-Bendix simplification ordering is parameterized by *symbol precedence*—a permutation of the predicate and function symbols of the input problem's signature. Thus, the choice of precedence has an indirect yet often substantial impact on the amount of work required to complete a proof search successfully.
This paper describes and evaluates a symbol precedence recommender, a machine learning system that estimates the best possible precedence based on observations of prover performance on a set of problems and random precedences. Using the graph convolutional neural network technology, the system does not presuppose the problems to be related or share a common signature. When coupled with the theorem prover Vampire and evaluated on the TPTP problem library, the recommender is found to outperform a state-of-the-art heuristic by more than $4\,\%$ on unseen problems.

Keywords: saturation-based theorem proving · simplification ordering · symbol precedence · machine learning · graph convolutional network

1 Introduction

Modern saturation-based Automatic Theorem Provers (ATPs) such as E [34], SPASS [40], or Vampire [21] employ the superposition calculus [4,24] as their underlying inference system. Integrating the flavors of resolution [5], paramodulation [30], and the unfailing completion [3], superposition is a powerful calculus with native support for equational reasoning. The calculus is parameterized by a simplification ordering on terms and uses it to constrain the applicability of inferences, with a significant impact on performance.

Both main classes of simplification orderings used in practice, the Knuth-Bendix ordering [19] and the lexicographic path ordering [16], are specified with the help of a *symbol precedence*, an ordering on the signature symbols. While the superposition calculus is refutationally complete for any simplification ordering [4], the choice of the precedence has a significant impact on how long it takes to solve a given problem.

It is well known that giving the highest precedence to the predicate symbols introduced as sub-formula names during clausification [25] can immediately make the saturation produce the exponential set of clauses that the transformation is designed to

A. Platzer and G. Sutcliffe (Eds.): CADE 2021, LNAI 12699, pp. 525–542, 2021.
https://doi.org/10.1007/978-3-030-79876-5_30

avoid [29]. Also, certain orderings help to make the superposition a decision procedure on specific fragments of first-order logic (see, e.g., [11,14]). However, the precise way by which the choice of a precedence influences the follow-up proof search on a general problem is extremely hard to predict.

Several general-purpose precedence generating schemes are available to ATP users, such as the successful `invfreq` scheme in E [33], which orders the symbols by the number of occurrences in the input problem. However, experiments with random precedences indicate that the existing schemes often fail to come close to the optimum precedence [28], suggesting room for further improvements.

In this work, we propose a machine learning system that learns to predict for an ATP whether one precedence will lead to a faster proof search on a given problem than another. Given a previously unseen problem, it can then be asked to recommend the best possible precedence for an ATP to run with. Relying only on the logical structure of the problems, the system generalizes the knowledge about favorable precedences across problems with different signatures.

Our recommender uses a relational graph convolutional neural network [32] to represent the problem structure. It learns from the ATP performance on selected problems and pairs of randomly sampled precedences. This information is used to train a *symbol cost model*, which then realizes the recommendation by simply sorting the problem's symbols according to the obtained costs.

This work strictly improves on our previous experiments with linear regression models and simple hand-crafted symbol features [6] and is, to the best of our knowledge, the first method able to propose good symbol precedences automatically using a non-linear transformation of the input problem structure.

The rest of this paper is organized as follows. Section 2 exposes the basic terminology used throughout the remaining sections. Section 3 proposes a structure of the precedence recommender that can be trained on pairs of symbol precedences, as described in Sect. 4. Section 5 summarizes and discusses experiments performed using an implementation of the precedence recommender. Section 6 compares the system proposed in this work with notable related works. Section 7 concludes the investigation and outlines possible directions for future research.

2 Preliminaries

2.1 Saturation-Based Theorem Proving

A *first-order logic (FOL) problem* consists of a set of axiom formulas and a conjecture formula. In a *refutation-based automated theorem prover (ATP)*, proving that the axioms entail the conjecture is reduced to proving that the axioms together with the negated conjecture entail a *contradiction*. The most popular first-order logic (FOL) automated theorem provers (ATPs), such as Vampire [21], E [34], or SPASS [40], start the proof search by converting the input FOL formulas to an equisatisfiable representation in *clause normal form (CNF)* [25,13]. We denote the problem in clause normal form (CNF) as $P = (\Sigma, Cl)$, where Σ is a list of all non-logical (predicate and function) *symbols* in the problem called the *signature*, and Cl is the set of clauses of the problem (including the negated conjecture).

Given a problem P in CNF, a *saturation-based* ATP searches for a refutational proof by iteratively applying the *inference rules* from the given *calculus* to infer new clauses entailed by Cl. As soon as the empty clause, denoted by $\square$, is inferred, the prover concludes that the premises entail the conjecture. The sequence of those inferences leading up from the input clauses Cl to the discovered $\square$ constitutes a proof. If the premises do not entail the conjecture, the proof search continues until the set of inferred clauses is saturated with respect to the inference rules. In the standard setting of time-restricted proof search, a time limit may end the process prematurely.

Since the space of derivable clauses is typically very large, the efficacy of the prover depends on the order in which the inferences are applied. The standard saturation-based ATPs order the inferences by maintaining two classes of inferred clauses: processed and unprocessed [34]. In each *iteration of the saturation loop*, one clause (so-called *given clause*) is combined with all the processed clauses for inferences. The resulting new clauses and the given clause are added to the unprocessed set and the processed set, respectively. Finishing the proof in few iterations of the saturation loop is important because the number of inferred clauses typically grows exponentially during the proof search.

2.2 Superposition Calculus

The *superposition calculus* is of particular interest because it is used in the most successful contemporary FOL ATPs. A *simplification ordering on terms* [4] constrains the inferences of the superposition calculus.

The simplification ordering on terms influences the superposition calculus in two ways. First, the inferences on each clause are limited to the selected literals. In each clause, either a negative literal or all maximal literals are selected. The maximality is evaluated according to the simplification ordering. Second, the simplification ordering orients some of the equalities to prevent superposition and equality factoring from inferring redundant complex conclusions. In each of these two roles, the simplification ordering may impact the direction and, in effect, the length of the proof search.

The *Knuth-Bendix ordering (KBO)* [19], a commonly used simplification ordering scheme, is parameterized by symbol weights and a *symbol precedence*, a permutation[3] of the non-logical symbols of the input problem. In this work, we focus on the task of finding a symbol precedence which leads to a good performance of an ATP when plugged into the Knuth-Bendix ordering (KBO), leaving all the symbol weights at the default value 1 as set by the ATP Vampire.

2.3 Neural Networks

A *feedforward artificial neural network* [12] is a directed acyclic graph of *modules*. Each module is an operation that consumes a numeric (input) *vector* and outputs a numeric vector. Each of the components of the output vector is called a *unit* of the

[3] The definition of KBO does not require the precedence to be total. However, for use in ATPs, the more symbols and thus also terms we can compare, the better.

module. The output of each module is differentiable with respect to the input almost everywhere.

The standard modules include the *fully connected layer*, which performs an affine transformation, and non-linear *activation functions* such as the *Rectified Linear Unit (ReLU)* or *sigmoid*.[4] A fully connected layer with a single unit is called the *linear unit*.

Some of the modules are parameterized by numeric *parameters*. For example, the fully connected layer that transforms the input x by the affine transformation $Wx + b$ is parameterized by the weight matrix W and the bias vector b. If the output of a module is differentiable with respect to a parameter, that parameter is considered *trainable*.

In a typical scenario, the neural network is trained by *gradient descent* on a *training set* of *examples*. In such a setting, the network outputs a single numeric value called *loss* when evaluated on a *batch* of examples. The loss of a batch is typically computed as a weighted sum of the losses of the individual examples. Since each of the modules is differentiable with respect to its input and trainable parameters, the gradient of the loss with respect to all trainable parameters of the neural network can be computed using the *back-propagation* algorithm [12]. The trainable parameters are then updated by taking a small step against the gradient—in the direction that is expected to reduce the loss. An *epoch* is a sequence of iterations that updates the trainable parameters using each example in the training set exactly once.

A *graph convolutional network (GCN)* is a special case of feedforward neural network. The modules of a GCN transform messages that are passed along the edges of a graph encoded in the input example. A particular architecture of a GCN used prominently in this work is discussed in Sect. 3.2.

3 Architecture

A *symbol precedence recommender* is a system that takes a CNF problem $P = (\Sigma, Cl)$ as the input, and produces a precedence π^* over the symbols Σ as the output. For the recommender to be useful, it should produce a precedence that likely leads to a quick search for a proof. In this work, we use the number of iterations of the saturation loop as a metric describing the effort required to find a proof.

The recommender described in this section first uses a neural network to compute a cost value for each symbol of the input problem, and then orders the symbols by their costs in a non-increasing order. In this manner, the task of finding good precedences is reduced to the task of training a good symbol cost function, as discussed in Sect. 4.

The recommender consists of modules that perform specific sub-tasks, each of which is described in detail in one of the following sections (see also Fig. 1).

3.1 Graph Constructor: From CNF to Graphs

As the first step of the recommender processing pipeline, the input problem is converted from a CNF representation to a *heterogeneous (directed) graph* [41]. Each of the nodes of the graph is labeled with a node type, and each edge is labeled with an edge type,

[4] These are, respectively, $f(x) = \max\{0, x\}$ and $g(x) = \frac{1}{1+e^{-x}}$.

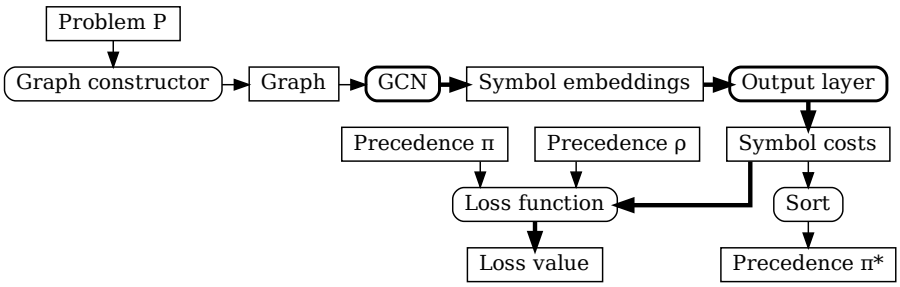


Fig. 1. Recommender architecture overview. When recommending a precedence, the input is problem P and the output is precedence π^*. When training, the input is problem P and precedences π and ρ, and the output is the loss value. The trainable modules and the edges along which the loss gradient is propagated are emphasized by bold lines.

defining the heterogeneous nature of the graph. Each node corresponds to one of the elements that constitute the CNF formula, such as a clause, an atom, or a predicate symbol. Each such category of elements corresponds to one node type. The edges represent the (oriented) relations between the elements, for example, the incidence relation between a clause and one of its (literals') atoms, or the relation between an atom and its predicate symbol. $\mathcal{R}$ denotes the set of all relations in the graph. Figure 2 shows the types of nodes and edges used in our graph representation. Figure 3 shows an example of a graph representation of a simple problem.

The graph representation exhibits, namely, the following properties:

- Lossless: The original problem can be faithfully reconstructed from the corresponding graph representation (up to logical equivalence).
- Signature agnostic: Renaming the symbols and variables in the input problem yields an isomorphic graph.
- For each relation $r \in \mathcal{R}$, its inverse r^{-1} is also present in the graph, typicallyrepresented by a different edge type.
- The polarity of the literals is expressed by the type of the edge (`pos` or `neg`) connecting the respective atom to the clause it occurs in.
- For every non-equality atom and term, the order of its arguments is captured by a sequence of `argument` nodes chained by edges [27].
- The two operands of equality are not ordered. This reflects the symmetry of equality.
- Sub-expression sharing [8,26,27]: Identical atoms and terms share a node representation.

3.2 GCN: From Graphs to Symbol Embeddings

For each symbol in the input problem P, we seek to find a vector representation, i.e., an *embedding*, that captures the symbol's properties that are relevant for correctly ranking the symbol in the symbol precedences over P.

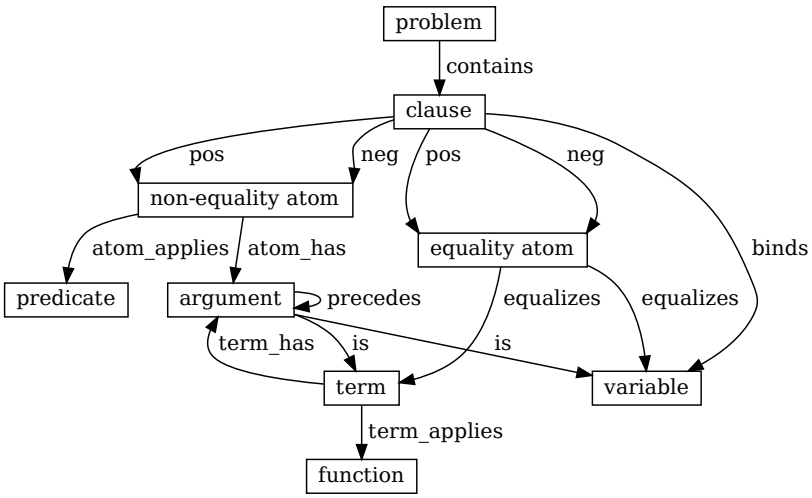


Fig. 2. CNF graph schema

The symbol embeddings are output by a *relational graph convolutional network (R-GCN)* [32], which is a stack of *graph convolutional layers*. Each layer consists of a collection of differentiable modules—one module per edge type. The computation of the GCN starts with assigning each node an initial embedding and then iteratively updates the embeddings by passing them through the convolutional layers.

The initial embedding $h_a^{(0)}$ of a node a is a concatenation of two vectors: a *feature vector* specific for that node (typically empty) and a trainable vector shared by all nodes of the same type. In our particular implementation, feature vectors are used in nodes that correspond to clauses and symbols. Each clause node has a feature vector with a one-hot encoding of the role of the clause, which can be either axiom, assumption, or negated conjecture [38,36]. Each symbol node has a feature vector with two bits of data: whether the symbol was introduced into the problem during preprocessing (most notably during clausification), and whether the symbol appears in a conjecture clause.

One pass through the convolutional layer updates the node embeddings by passing a message along each of the edges. For an edge of type $r \in \mathcal{R}$ going from source node s to destination node d at layer l, the message is composed by converting the embedding of the source node $h_s^{(l)}$ using the module associated with the edge type r. In the simple case that the module is a fully connected layer with weight matrix $W_r^{(l)}$ and bias vector $b_r^{(l)}$, the message is $W_r^{(l)} h_s^{(l)} + b_r^{(l)}$. Each message is then divided by the normalization constant $c_{s,d} = \sqrt{|\mathcal{N}_s^r|}\sqrt{|\mathcal{N}_d^r|}$ [18], where $\mathcal{N}_a^r$ is the set of neighbors of node a under the relation r.

Once all messages are computed, they are aggregated at the destination nodes to form new node embeddings. Each node d aggregates all the incoming messages of a given edge type r by summation, then passes the sum through an activation function

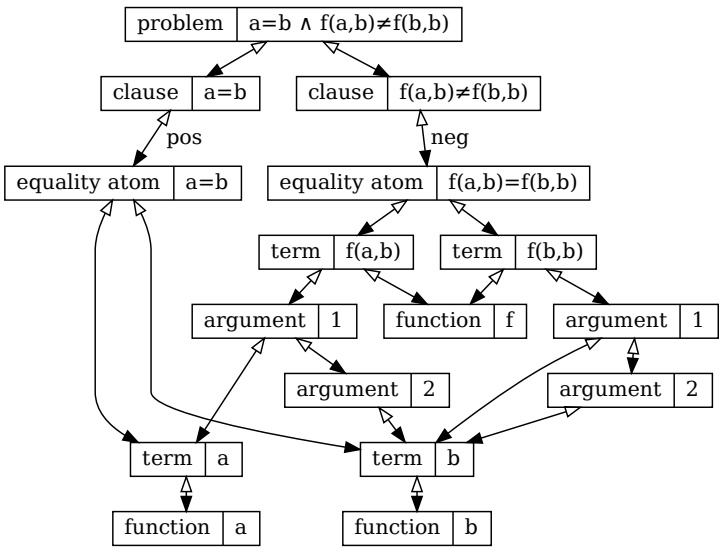


Fig. 3. Graph representation of the CNF formula $a = b \wedge f(a, b) \neq f(b, b)$

σ such as the ReLU, and finally aggregates the messages across the edge types by summation, yielding the new embedding $h_d^{(l+1)}$.

The following formula captures the complete update of the embedding of node d by layer l:

$$h_d^{(l+1)} = \sum_{r \in \mathcal{R}} \sigma\left(\sum_{s \in \mathcal{N}_d^r} \frac{1}{c_{s,d}} (W_r^{(l)} h_s^{(l)} + b_r^{(l)}) \right)$$

3.3 Output Layer: From Symbol Embeddings to Symbol Costs

The symbol cost of each symbol is computed by passing the symbol's embedding through a linear output unit, which is an affine transformation with no activation function.

It is possible to use a more complex output layer in place of the linear unit, e.g., a feedforward network with one or more hidden layers. Our experiments showed no significant improvement when a hidden layer was added, likely because the underlying GCN learns a sufficiently complex transformation.

Let θ denote the vector of all parameters of the whole neural network consisting of the GCN and the output unit. Given an input problem P with signature $\Sigma = (s_1, \ldots, s_n)$, we denote the cost of symbol s_i predicted by the network as $c(i, P; \theta)$. In the rest of this text, we refer to the predicted cost of s_i simply as $c(i)$ because the problem P and the parameters θ are fixed in each respective context.

3.4 Sort: From Symbol Costs to Precedence

The symbol precedence heuristics commonly used in the ATPs sort the symbols by some numeric syntactic property that is inexpensive to compute, such as the number of occurrences in the input problem, or the symbol arity. In our precedence recommender, we sort the symbols by their costs c produced by the neural network described in Sects. 3.2 and 3.3. An advantage of this scheme is that sorting is a fast operation.

Moreover, as we show in Sect. 4, it is possible to train the underlying symbol costs by gradient descent.

4 Training Procedure

In Sect. 3 we described the structure of a recommender system that generates a symbol precedence for an arbitrary input problem. The efficacy of the recommender depends on the quality of the underlying symbol cost function c. In theory, the symbol cost function can assign the costs so that sorting the symbols by their costs yields an optimum precedence. This is because, at least in principle, all the information necessary to determine the optimum precedence is present in the graph representation of the input problem thanks to the lossless property of the graph encoding. Our approach to defining an appropriate symbol cost function is based on statistical learning from executions of an ATP on a set of problems with random precedences.

To train a useful symbol cost function c, we define a precedence cost function C using the symbol cost function c in a manner that ensures that minimizing C corresponds to sorting the symbols by c. Finding a precedence that minimizes C can then be done efficiently and precisely. We proceed to train C on the proxy task of ranking the precedences.

4.1 Precedence Cost

We extend the notion of cost from symbols to precedences by taking the sum of the symbol costs weighted by their positions in the given precedence π:

$$C(\pi) = Z_n \sum_{i=1}^{n} i \cdot c(\pi(i))$$

$Z_n = \frac{2}{n(n+1)}$ is a normalization factor that ensures the commensurability of precedence costs across signature sizes. More precisely, normalizing by Z_n makes the expected value of the precedence cost on a given problem independent of the problem's signature size n, provided the expected symbol cost $\mathbb{E}_i[c(i)]$ does not depend on n:

$$\begin{aligned}\mathbb{E}_\pi[C(\pi)] = \mathbb{E}_\pi\left[Z_n \sum_{i=1}^{n} i \cdot c(\pi(i))\right] &= Z_n \sum_{i=1}^{n} i \cdot \mathbb{E}_\pi[c(\pi(i))] \\ &= Z_n \left(\sum_{i=1}^{n} i\right) \mathbb{E}_i[c(i)] = \frac{2}{n(n+1)} \frac{n(n+1)}{2} \mathbb{E}_i[c(i)] = \mathbb{E}_i[c(i)]\end{aligned}$$

When C is defined in this way, the precedence produced by the recommender (see Sect. 3.4) minimizes C.

Lemma 1. *The precedence cost C is minimized by any precedence that sorts the symbols by their costs in non-increasing order:*

$$\operatorname*{argmin}_{\rho} C(\rho) = \operatorname{argsort}^{-}(c(1), \ldots, c(n))$$

where $\operatorname{argmin}_{\rho} C(\rho)$ is the set of all precedences that minimize precedence cost C for a given symbol cost c, and $\operatorname{argsort}^{-}(x)$ is the set of all permutations π that sort vector x in non-increasing order ($x_{\pi(1)} \geq x_{\pi(2)} \geq \ldots \geq x_{\pi(n)}$).

Proof. We prove direction "$\operatorname{argmin}_{\rho} C(\rho) \subseteq \operatorname{argsort}^{-}(c(1), \ldots, c(n))$" by contradiction. Let π minimize C and let π not sort the costs in non-increasing order. Then there exist $k < l$ such that $c(\pi(k)) < c(\pi(l))$. Let $\bar{\pi}$ be a precedence obtained from π by swapping the elements k and l. Then we obtain

$$\begin{aligned}
\frac{C(\bar{\pi}) - C(\pi)}{Z_n} &= kc(\bar{\pi}(k)) + lc(\bar{\pi}(l)) - kc(\pi(k)) - lc(\pi(l)) \\
&= kc(\pi(l)) + lc(\pi(k)) - kc(\pi(k)) - lc(\pi(l)) \\
&= k(c(\pi(l)) - c(\pi(k))) - l(c(\pi(l)) - c(\pi(k))) \\
&= (k - l)(c(\pi(l)) - c(\pi(k))) \\
&< 0
\end{aligned}$$

The final inequality is due to $k - l < 0$ and $c(\pi(l)) - c(\pi(k)) > 0$. Clearly, $Z_n > 0$ for any $n \geq 0$. Thus, $C(\bar{\pi}) < C(\pi)$, which contradicts the assumption that π minimizes C.

To prove the other direction of the equality, first observe that all precedences π that sort the symbol costs in a non-increasing order necessarily have the same precedence cost $C(\pi)$. Since $\emptyset \neq \operatorname{argmin}_{\rho} C(\rho) \subseteq \operatorname{argsort}^{-}(c(1), \ldots, c(n))$, each of the precedences in $\operatorname{argsort}^{-}(c(1), \ldots, c(n))$ has the cost $\min_{\rho} C(\rho)$. It follows that $\operatorname{argsort}^{-}(c(1), \ldots, c(n)) \subseteq \operatorname{argmin}_{\rho} C(\rho)$. □

4.2 Learning to Rank Precedences

Our ultimate goal is to train the precedence cost function C so that it is minimized by the best precedence, measuring the quality of a precedence by the number of iterations of the saturation loop taken to solve the problem.

Approaching this task directly, as a regression problem, runs into the difficulty of establishing sensible target cost values for the precedences in the training dataset, especially when a wide variety of input problems is covered. Approaching the task as a binary classification of precedences seems possible, but it is not clear which precedences should be a priori labeled as positive and which as negative, to give a guarantee that a precedence minimizing the precedence cost (i.e. the one obtained by sorting) would be among the best in any good sense.

We cast the task as an instance of score-based ranking problem [23,7] by training a classifier to decide which of a *pair* of precedences is better based on their costs. We

train the classifier in a way that ensures that better precedences are assigned lower costs. The motivation for learning to order pairs of precedences is that it allows learning on easy problems, and that it may allow the system to generalize to precedences that are better than any of those seen during training.

Training Data. Each training example has the form (P, π, ρ), where $P = (\Sigma, Cl)$ is a problem and π, ρ are precedences over Σ such that the prover using π solves P in fewer iterations of the saturation loop than with ρ, denoted as $\pi \prec_P \rho$.

Loss Function. Let (P, π, ρ) be a training example ($\pi \prec_P \rho$). The precedence cost classifies this example correctly if $C(\pi) < C(\rho)$, or alternatively $S(\pi, \rho) = C(\rho) - C(\pi) > 0$. We approach this problem as an instance of binary classification with the logistic loss [23], a loss function routinely used in classification tasks in machine learning:

$$\begin{aligned}\ell(P, \pi, \rho) = -\log \operatorname{sigmoid} S(\pi, \rho) &= -\log \operatorname{sigmoid}(C(\rho) - C(\pi)) \\ &= -\log \operatorname{sigmoid} Z_n \sum_{i=1}^{n} i(c(\rho(i)) - c(\pi(i)))\end{aligned}$$

Note that the classifier cannot simply train S to output a positive number on all pairs of precedences because S is defined as a difference of two precedence costs. Intuitively, by training on the example (P, π, ρ) we are pushing $C(\pi)$ down and $C(\rho)$ up.

The loss function is clearly differentiable with respect to the symbol costs, and the symbol cost function c is differentiable with respect to its trainable parameters. This enables the use of gradient descent to find the values of the parameters of c that locally minimize the loss value.

Figure 1 shows how the loss function is plugged into the recommender for training.

5 Experimental Evaluation

To demonstrate the capacity of the trainable precedence recommender described in Sects. 3 and 4, we performed a series of experiments. In this section, we describe the design and configuration of the experiments, and then compare the performance of several trained models to a baseline heuristic.

The scripts that were used to generate the training data and to train and evaluate the recommender are available online.[5]

5.1 Environment

System. All experiments were run on a computer with the CPU Intel Xeon Gold 6140 (72 cores @ 2.30 GHz) and 383 GiB RAM.

[5] `https://github.com/filipbartek/vampire-ml/tree/cade28`

Solver. The empirical evaluation was performed using a modified version of the ATP Vampire 4.3.0 [21]. The prover was used to generate the training data and to evaluate the trained precedence recommender. To generate the training data, Vampire was modified to output CNF representations of the problems and annotated problem signatures in a machine-readable format. For the evaluation of the precedences generated by the recommender, Vampire was modified to allow the user to supply explicit predicate and function symbol precedences for the proof search (normally, the user only picks a precedence generation heuristic). The modified version of Vampire is available online.[6]

We run Vampire with a fixed strategy[7] and a time limit of 10 seconds. To increase the potential impact of predicate precedences, we used a simple transfinite Knuth-Bendix ordering (TKBO) [22,20] that compares atoms according to the predicate precedence first, using the regular KBO to break ties between atoms and to compare terms (using the Vampire option `--literal_comparison_mode predicate`).

5.2 Dataset Preparation

The training data consists of examples of the form (P, π, ρ), where P is a CNF problem and π, ρ are precedences of symbols of problem P such that out of the two precedences, π yields a proof in fewer iterations of the saturation loop (see Sect. 2.1).

Since the TKBO never compares a predicate symbol with a function symbol, two separate precedences can be considered for each problem: a predicate precedence and a function precedence. We trained a predicate precedence recommender separately from a function precedence recommender to simplify the training process and to isolate the effects of the predicate and function precedences. This section describes how the training data for the case of training a *predicate* precedence recommender was generated. Data for training the function precedence recommender was generated analogously.

Base Problem Set. The input problems were assumed to be specified in the CNF or the first-order form (FOF) fragment of the TPTP language [36]. FOF problems were first converted into equisatisfiable CNF problems by Vampire.

We used the problem library TPTP v7.4.0 [36] as the source of problems for training and evaluation of the recommender. We denote the set of all problems available for training and evaluation as $\mathcal{P}_0$ ($|\mathcal{P}_0| = 17\,053$).

Node Feature Extraction. In addition to the signature and the structure of the problem, some metadata was extracted from the input problem to allow training a more efficient recommender. First, each clause was annotated with its role in the problem, which could be either axiom, assumption, or negated conjecture. Second, each symbol was annotated with two bits of data: whether the symbol was introduced into the problem during preprocessing, and whether the symbol appeared in a conjecture clause. This metadata was used to construct the initial embeddings of the respective nodes in the graph representation of the problem (see Sect. 3.2).

[6] `https://github.com/filipbartek/vampire/tree/cade28`

[7] Saturation algorithm: DISCOUNT, age to weight ratio: 1:10, AVATAR [39]: disabled, literal comparison mode: predicate; all other options left at their default values.

Examples Generation. The examples were generated by an iterative sampling of $\mathcal{P}_0$. In each iteration, a problem $P \in \mathcal{P}_0$ was chosen and Vampire was executed twice on P with two (uniformly) random predicate precedences and one common random function precedence. The "background" random function precedence served as additional noise (in addition to the variability contained in TPTP) and made sure that the predicate precedence recommender would not be able to rely on any specificity that would come from fixing function precedences in the training data.

The two executions were compared in terms of performance: the predicate precedence π was recognized as better than the predicate precedence ρ, denoted as $\pi \prec_P \rho$, if the proof search finished successfully with π and if the number of iterations of the saturation loop with π was smaller than with ρ. If one of the two precedences was recognized as better, the example (P, π, ρ) would be produced, where π was the better precedence, and ρ was the other precedence. Otherwise, for example, if the proof search timed out on both precedences, we would go back to sampling another problem.

To ensure the efficiency of the sampling, we interpreted the process as an instance of the Bernoulli multi-armed bandit problem [37], with the reward of a trial being 1 in case an example is produced, and 0 otherwise.

We employed adaptive sampling to balance exploring problems that have been tried relatively scarcely and exploiting problems that have yielded examples relatively often. For each problem $P \in \mathcal{P}_0$, the generator kept track of the number of times the problem has been tried n_P, and the number of examples generated from that problem s_P. The ratio $\frac{s_P}{n_P}$ corresponded to the average reward of problem P observed so far. The problems were sampled using the allocation strategy UCB1 [1] with a parallelizing relaxation.

First, the values of n_P and s_P for each problem P were bootstrapped by sampling the problem a number of times equal to a lower bound on the final value of n_P (at least 1).[8] In each subsequent iteration, the generator sampled the problem P that maximized $\frac{s_P}{n_P} + \sqrt{\frac{2 \ln n}{n_P}}$, where $n = \sum_{P \in \mathcal{P}_0} n_P$ was the total number of tries on all problems. The parallelizing relaxation means that the s_P values were only updated once in 1000 iterations, allowing up to 2000 parallel solver executions.

The sampling continued until $1\,000\,000$ examples were generated when training a predicate precedence recommender, or $800\,000$ examples in the case of a function precedence recommender. For example, while generating $1\,000\,000$ examples for the predicate precedence dataset, 5349 out of the $17\,053$ problems yielded at least one example, while the least explored problem was tried 19 times, and the most exploited problem 504 times.

Validation Split. The $17\,053$ problems in $\mathcal{P}_0$ were first split roughly in half to form the training set and the validation set. Next, both training and validation sets were restricted to problems whose graph representation consisted of at most $100\,000$ nodes to limit the memory requirements of the training. Approximately $90\,\%$ of the problems fit into this limit and there were 7648 problems in the resulting validation set $\mathcal{P}_{\text{val}}$. The training

[8] The number of tries each problem was bootstrapped with is $n_0 = \lceil \frac{2 \log N}{(1+\sqrt{\frac{2 \log N}{N} |\mathcal{P}_0|})^2} \rceil$, where N is the final number of examples to be generated. For example, if $N = 1\,000\,000$ and $|\mathcal{P}_0| = 17\,053$, then $n_0 = 10$.

set $\mathcal{P}_{\text{train}}$ was further restricted to problems that correspond to at least one training example, resulting in 2571 problems when training a predicate precedence recommender, and 1953 problems when training a function precedence recommender.

5.3 Hyperparameters

We used a GCN described in Sect. 3.2 with depth 4, message size 16, ReLU activation function, skip connections [41], and layer normalization [2]. We tuned the hyperparameters by a small manual exploration.

5.4 Training Procedure

A symbol cost model was trained by gradient descent on the precedence ranking task (see Sect. 4.2) using the examples generated from $\mathcal{P}_{\text{train}}$. To avoid redundant computations, all examples generated from any given problem were processed in the same training batch. Thus, each training batch contained up to 128 problems and all examples generated from these problems. The symbol cost model was trained using the Adam optimizer [17]. The learning rate started at 1.28×10^{-3} and was halved each time the loss on $\mathcal{P}_{\text{train}}$ stagnated for 10 consecutive epochs.

The examples were weighted. Each of the examples of problem P contributed to the training with the weight $\frac{1}{s_P}$, where s_P was the number of examples of problem P in the training set. This ensured that each problem contributed to the training to the same degree irrespective of the relative number of examples.

We continued the training until the validation accuracy stopped increasing for 100 consecutive epochs.

5.5 Final Evaluation

After the training finished, we performed a final evaluation of the most promising intermediate trained model on the whole $\mathcal{P}_{\text{val}}$. The model that manifested the best solver performance on a sample of 1000 validation problems was taken as the most promising.

5.6 Results

A predicate precedence recommender was trained on approximately 500 000 examples, and a function precedence recommender was trained on approximately 400 000 examples. For each problem $P \in \mathcal{P}_{\text{val}}$, a predicate and a function precedences were generated by the respective trained recommender, and Vampire was run using these precedences with a wall clock time limit of 10 seconds. The results are averaged over 5 runs to reduce the effect of noise due to the wall clock time limit. As a baseline, the performance of Vampire with the `frequency` precedence heuristic[9] was evaluated with the same time limit. For comparison, the two trained recommenders were evaluated separately, with the predicate precedence recommender using the `frequency` heuristic to generate the function precedences, and vice versa.

[9] This is Vampire's analogue of the `invfreq` scheme in E [33].

To generate a precedence for a problem, the recommender first converts the problem to a machine-friendly CNF format, then converts the CNF to a graph, then predicts symbol costs using the GCN model and finally orders the symbols by their costs to produce the precedence. To simplify the experiment, the time limit of 10 seconds was only imposed on the Vampire run, excluding the time taken by the recommender to generate the precedence. When run with 2 threads, the preprocessing of a single problem took at most 1.26 seconds for $80\,\%$ of the problems by extrapolation from a sample of 1000 problems.[10] Table 1 shows the results of the final evaluation.

Table 1. Results of the evaluation of symbol precedence heuristics based on various symbol cost models on $\mathcal{P}_{\text{val}}$ ($|\mathcal{P}_{\text{val}}| = 7648$). Means and standard deviations over 5 runs are reported. The GCN models were trained according to the description in Sects. 3 to 5. The model Simple is the final linear model from our previous work [6]. The models that used machine learning only for the predicate precedence used the `frequency` heuristic for the function precedence, and vice versa. The frequency model uses the standard `frequency` heuristic for both predicate and function precedence.

Symbol cost model	Successes on $\mathcal{P}_{\text{val}}$		Improvement over baseline	
	Mean	Std	Absolute	Relative
GCN (predicate and function)	3951.6	1.62	+182.0	1.048
GCN (predicate only)	3923.6	2.24	+154.0	1.041
GCN (function only)	3874.2	1.83	+104.6	1.028
Simple (predicate only)	3827.2	1.94	+57.6	1.015
Frequency (baseline)	3769.6	3.07	0.0	1.000

The results show that the GCN-based model outperformed the `frequency` heuristic by a significant margin. Since the predicate precedence recommender was trained with randomly distributed function precedences, it was expected to perform well irrespective of the function precedence heuristic it is combined with, and conversely. Combining the trained recommenders for predicate and function precedences manifested better performance than any of the two in combination with the standard `frequency` heuristic, outperforming the `frequency` heuristic by approximately $4.8\,\%$.

We have confirmed our earlier conjecture [6] that using a graph neural network (GNN) may outperform the "simple" linear predicate precedence heuristic trained in [6].[11]

6 Related Work

Our previous text [6] marked the initial investigation of applying techniques of machine learning to generating good symbol precedences. The neural recommender presented here uses a GNN to model symbol costs, while [6] used a linear combination of symbol features readily available in the ATP Vampire. The GNN-based approach yields more performant precedences at the cost of longer training and preprocessing time.

[10] The remaining $20\,\%$ of the problems either finished preprocessing within 5 seconds, or were omitted from preprocessing due to exceeding the node count limit.

[11] The measurements presented in Table 1 are not directly comparable with those reported in [6] due to differences in the validation problem sets and the computation environments.

In [26], [15] and [27], the authors propose similar GNN architectures to solve tasks on FOL problems. They use the GNNs to solve classification tasks such as premise selection. While our system is trained on a proxy classification task, the main task it is evaluated on is the generation of useful precedences.

The problem of learning to rank objects represented by scores trainable by gradient descent was explored in [7]. Our work can be seen to apply the approach of [7] to rank permutations represented by weighted sums of symbol costs.

7 Conclusion and Future Work

We have described a system that extracts useful symbol precedences from the graph representations of CNF problems. Comparison with a conventional symbol precedence heuristic shows that using a GCN to consider the whole structure of the input problem is beneficial.

A manual analysis of the trained recommender could produce new insights into how the choice of the symbol precedence influences the proof search, which could in turn help design new efficient precedence generating schemes. Indeed, a trained cost model summarizes the observed behaviors of an ATP with random precedences and is able to discover patterns in them (as we know implicitly from its accuracy) despite their seemingly chaotic behavior as perceived by a human observer. The challenge is to extract these patterns in a human-understandable form.

In addition to the symbol precedence, KBO is determined by symbol *weights*. In this work, we keep the symbol weights fixed to the value 1. Learning to recommend symbol weights in addition to the precedences represents an interesting avenue for future research.

The same applies to the idea of learning to recommend both the predicate and function precedences using a single GCN. The joint learning, although more complex to design, could additionally discover interdependencies between the effects of function precedence and predicate precedence on the proof search, while the current setup implicitly assumes that the effects are independent. Finally, a higher training data efficiency could be achieved by considering all pairs of measured executions on a problem in one training batch.

Acknowledgments

This work was generously supported by the Czech Science Foundation project no. 20-06390Y (JUNIOR grant), the project RICAIP no. 857306 under the EU-H2020 programme, and the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multi-armed bandit problem. Machine Learning **47**(2-3), 235–256 (May 2002). https://doi.org/10.1023/A:1013689704352

2. Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization (Jul 2016), http://arxiv.org/abs/1607.06450
3. Bachmair, L., Derschowitz, N., Plaisted, D.A.: Completion without failure. In: Aït-Kaci, H., Nivat, M. (eds.) Rewriting Techniques, pp. 1–30. Academic Press (1989). https://doi.org/10.1016/B978-0-12-046371-8.50007-9
4. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Log. Comput. **4**(3), 217–247 (1994). https://doi.org/10.1093/logcom/4.3.217
5. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson and Voronkov [31], pp. 19–99. https://doi.org/10.1016/b978-044450813-3/50004-7
6. Bártek, F., Suda, M.: Learning precedences from simple symbol features. In: Fontaine et al. [10], pp. 21–33, http://ceur-ws.org/Vol-2752/paper2.pdf
7. Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.: Learning to rank using gradient descent. In: ICML 2005 - Proceedings of the 22nd International Conference on Machine Learning. pp. 89–96. ACM Press, New York, New York, USA (2005). https://doi.org/10.1145/1102351.1102363
8. Chvalovský, K., Jakubův, J., Suda, M., Urban, J.: ENIGMA-NG: Efficient neural and gradient-boosted inference guidance for E. In: Fontaine [9]. https://doi.org/10.1007/978-3-030-29436-6_12
9. Fontaine, P. (ed.): Automated Deduction - CADE 27, LNCS, vol. 11716. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6
10. Fontaine, P., Korovin, K., Kotsireas, I.S., Rümmer, P., Tourret, S. (eds.): Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020). No. 2752 in CEUR Workshop Proceedings, CEUR-WS.org, Aachen (2020), http://ceur-ws.org/Vol-2752
11. Ganzinger, H., de Nivelle, H.: A superposition decision procedure for the guarded fragment with equality. In: 14th Annual IEEE Symposium on Logic in Computer Science. pp. 295–303. IEEE Computer Society (1999). https://doi.org/10.1109/LICS.1999.782624
12. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep Learning. Adaptive computation and machine learning, MIT Press (2016), http://www.deeplearningbook.org/
13. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, Cambridge (2009). https://doi.org/10.1017/CBO9780511576430
14. Hustadt, U., Konev, B., Schmidt, R.A.: Deciding monodic fragments by temporal resolution. In: Nieuwenhuis, R. (ed.) Automated Deduction – CADE-20. LNCS, vol. 3632, pp. 204–218. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11532231_15
15. Jakubův, J., Chvalovský, K., Olšák, M., Piotrowski, B., Suda, M., Urban, J.: ENIGMA Anonymous: Symbol-independent inference guiding machine (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning. LNCS, vol. 12167, pp. 448–463. Springer, Cham (Jul 2020). https://doi.org/10.1007/978-3-030-51054-1_29
16. Kamin, S.N., Lévy, J.: Two generalizations of the recursive path ordering (1980), http://www.cs.tau.ac.il/~nachumd/term/kamin-levy80spo.pdf, unpublished letter to Nachum Dershowitz
17. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization (Dec 2014), http://arxiv.org/abs/1412.6980
18. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017 (Sep 2017), https://openreview.net/forum?id=SJU4ayYgl
19. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Siekmann and Wrightson [35], pp. 342–376. https://doi.org/10.1007/978-3-642-81955-1_23

20. Kovács, L., Moser, G., Voronkov, A.: On transfinite Knuth-Bendix orders. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction – CADE-23. LNCS, vol. 6803, pp. 384–399. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_29
21. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. LNCS, vol. 8044, pp. 1–35. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
22. Ludwig, M., Waldmann, U.: An extension of the Knuth-Bendix ordering with LPO-like properties. In: Dershowitz, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. LNCS, vol. 4790, pp. 348–362. Springer, Berlin, Heidelberg (Oct 2007). https://doi.org/10.1007/978-3-540-75560-9_26
23. Mohri, M., Rostamizadeh, A., Talwalkar, A.: Foundations of Machine Learning. MIT Press, 2 edn. (2018), https://cs.nyu.edu/~mohri/mlbook/
24. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson and Voronkov [31], pp. 371–443. https://doi.org/10.1016/b978-044450813-3/50009-6
25. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson and Voronkov [31], pp. 335–367. https://doi.org/10.1016/b978-044450813-3/50008-4
26. Olšák, M., Kaliszyk, C., Urban, J.: Property invariant embedding for automated reasoning. In: Giacomo, G.D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., Lang, J. (eds.) ECAI 2020 – 24th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 1395–1402. IOS Press (2020). https://doi.org/10.3233/FAIA200244
27. Rawson, M., Reger, G.: Directed graph networks for logical reasoning (extended abstract). In: Fontaine et al. [10], pp. 109–119, http://ceur-ws.org/Vol-2752/paper8.pdf
28. Reger, G., Suda, M.: Measuring progress to predict success: Can a good proof strategy be evolved? In: AITP 2017. pp. 20–21 (2017), http://aitp-conference.org/2017/aitp17-proceedings.pdf
29. Reger, G., Suda, M., Voronkov, A.: New techniques in clausal form generation. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI 2016. 2nd Global Conference on Artificial Intelligence. EPiC Series in Computing, vol. 41, pp. 11–23. EasyChair (2016). https://doi.org/10.29007/dzfz
30. Robinson, G., Wos, L.: Paramodulation and theorem-proving in first-order theories with equality. In: Siekmann and Wrightson [35], pp. 298–313. https://doi.org/10.1007/978-3-642-81955-1_19
31. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning (in 2 volumes). Elsevier and MIT Press (2001)
32. Schlichtkrull, M.S., Kipf, T.N., Bloem, P., van den Berg, R., Titov, I., Welling, M.: Modeling relational data with graph convolutional networks. In: Gangemi, A., Navigli, R., Vidal, M., Hitzler, P., Troncy, R., Hollink, L., Tordai, A., Alam, M. (eds.) The Semantic Web. LNCS, vol. 10843, pp. 593–607. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93417-4_38
33. Schulz, S.: E 2.4 user manual. EasyChair preprint no. 2272, Manchester (2020), https://easychair.org/publications/preprint/8dss
34. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine [9], pp. 495–507. https://doi.org/10.1007/978-3-030-29436-6_29
35. Siekmann, J.H., Wrightson, G. (eds.): Springer, Berlin, Heidelberg (1983)
36. Sutcliffe, G.: The TPTP problem library and associated infrastructure. Journal of Automated Reasoning **59**(4) (Dec 2017). https://doi.org/10.1007/s10817-017-9407-7
37. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, 2 edn. (2018), http://incompleteideas.net/book/the-book-2nd.html

38. TPTP syntax, http://www.tptp.org/TPTP/SyntaxBNF.html
39. Voronkov, A.: AVATAR: The architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46
40. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) Automated Deduction - CADE-22. LNCS, vol. 5663, pp. 140–145. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10
41. Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M.: Graph neural networks: A review of methods and applications (Dec 2018), http://arxiv.org/abs/1812.08434

Chapter 4

A GNN-Advised Clause Selection

In the work presented below [3], we tackled clause selection, arguably the most important heuristic choice point in a saturation-based prover. We used the GNN architecture that proved useful in precedence recommendation (see chapter 3) and trained a neural network that proposes symbol weights to configure the symbol-counting clause selection heuristic [30, 46].

For each problem, the neural network is only evaluated once as a preprocessing step to determine the symbol weights. This approach allows using the optimized clause selection procedure implemented in Vampire with very little modification and negligible computational overhead during the proof search.

The trained system increased the performance of a baseline configuration of Vampire by 6.6 %, which is more than the increase introduced by AVATAR, a powerful technique that delegates some propositional sub-tasks of the FOL proof search to a highly optimized SAT solver. Manual inspection revealed that the system learned to prioritize clauses with symbols that appear in the conjecture (goal-directed guidance), deprioritize clauses with subformula names, and prioritize non-ground clauses—clauses with variable occurrences.

EPiC Series in Computing

Volume 94, 2023, Pages 96–111

Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

How Much Should This Symbol Weigh? A GNN-Advised Clause Selection

Filip Bártek[1,2] and Martin Suda[1]

[1] Czech Institute of Informatics, Robotics and Cybernetics
[2] Faculty of Electrical Engineering
Czech Technical University in Prague, Czech Republic
{filip.bartek,martin.suda}@cvut.cz

Abstract

Clause selection plays a crucial role in modern saturation-based automatic theorem provers. A commonly used heuristic suggests prioritizing small clauses, i.e., clauses with few symbol occurrences. More generally, we can give preference to clauses with a low *weighted symbol occurrence count*, where each symbol's occurrence count is multiplied by a respective symbol weight. Traditionally, a human domain expert would supply the symbol weights.

In this paper, we propose a system based on a graph neural network that learns to predict symbol weights with the aim of improving clause selection for arbitrary first-order logic problems. Our experiments demonstrate that by advising the automatic theorem prover Vampire on the first-order fragment of TPTP using a trained neural network, the prover's problem solving capability improves by 6.6 % compared to uniformly weighting symbols and by 2.1 % compared to a goal-directed variant of the uniformly weighting strategy.

1 Introduction

Clause selection is a major choice point in modern saturation-based provers for first-order logic. It amounts to deciding, at each iteration of the saturation loop, which clause should be the next to activate, i.e., to participate in generating inferences with the previously activated clauses. Since the generating inferences form the backbone of the sought proof, a perfect clause selection oracle would completely alleviate the need for a search. Practical clause selection heuristics in the tightly optimized state-of-the-art provers, however, also need to be efficient to compute. The most widely used general-purpose ones are based on the first-in/first-out principle and simple symbol counting [23].

A promising line of research is to automatically derive new clause selection heuristics from prover successes using machine learning [21, 6]. In recent years, learning systems such as ENIGMA [11, 4, 13] or Deepire [28] have been able, for example, to substantially improve the performance of the base prover on problems exported from a large mathematical corpus [14]. As with classical heuristics, ensuring that the learned part is not too expensive to evaluate was key to success [12, 27].

R. Piskac and A. Voronkov (eds.), LPAR 2023 (EPiC Series in Computing, vol. 94), pp. 96–111

A natural generalization of the symbol counting heuristic, mentioned, for example, in the E prover [24] manual [22] under the name `FunWeight`, allows each symbol to have a user-specified weight to act in the heuristic as a multiplier for the respective symbol's occurrence count. While the extra degrees of freedom make the heuristic, in theory, strictly more powerful than its simple variant, only a true expert on a particular problem domain or on a specific encoding could possibly possess the knowledge to capitalize on the additional power easily. The main reason for this is that the proof search is typically quite fragile [29], making it very hard to draw conclusions about cause and effect. Additionally, the idea of using the extra generality to develop new proving strategies comes with a complication: The symbol weights depend on the problem's signature, so there is no immediate general way of specifying them in advance.

In this paper, we design and implement a learning system that automatically recommends good symbol weights for clause selection for any given problem. The system is based on a graph neural network (NN) that operates on the structure of the input problem's clause normal form. Similarly to ENIGMA or Deepire, we use previous prover successes to train the system, but rely on a novel scheme for balancing positive (coming from a proof) and negative (not coming from proof) training clauses. Because the GNN only computes in a one-time preprocessing phase to recommend the weights, there is no noticeable performance penalty during actual saturation.

We train and evaluate our system on first-order problems from Thousands of Problems for Theorem Provers (TPTP) [30], finding it to substantially improve performance over a baseline strategy with uniform symbol weights. TPTP is a very diverse problem collection, which means that the knowledge the system extracts through training must be, to a large degree, general-purpose and readily useful elsewhere. We attempt to shed some light on what was learned by having a closer look at the recommended weights on several problems that were only solved with our system's help.

The remainder of the paper is organized as follows. Section 2 reviews the basic concepts used throughout the paper. Sections 3 and 4 describe the main task we tackle and our solution—a GNN-based symbol weight recommender. In Section 5, we describe experiments we performed to evaluate our recommender and analyze the results, and in Section 6, we continue the analysis in a case-based manner. We discuss possible modifications and enhancements of our recommender in Section 7 before concluding in Section 8.

2 Background

We assume that the reader is familiar with the basic concepts related to automatic theorem proving. In this section, we provide a brief overview of the terminology relevant to our research with references to detailed explanations.

Formulas in first-order logic (FOL) are built using variables, function and predicate symbols, the equality symbol (which we denote $\approx$), logical connectives, and quantifiers [1, 5]. A FOL problem consists of FOL formulas: a conjecture and zero or more axioms. Solving the problem amounts to finding a proof that the conjecture necessarily follows from the axioms. In refutational proving, we look for a proof by contradiction—we negate the conjecture and proceed to infer a trivial contradiction. A common approach is to convert the problem to an equisatisfiable clause normal form (CNF) problem and to perform the inferences in the clause space, ultimately inferring a trivial contradiction in the form of the empty clause.

A saturation-based automatic theorem prover (ATP), such as Vampire [15] or E [24], relies on a given-clause procedure [18, 23] to orchestrate the processing of inferences. It maintains two sets of clauses: active and passive. At the beginning of a proof search, the passive set is populated with the input clauses. Then, iteratively, a clause is always selected from the passive

C	$W(C)$
$p(X_1, c, X_2) \vee q(X_1)$	$3w(X) + w(p) + w(q) + w(c)$
$g(X_1, h(X_2)) \approx f(g(X_1, X_2), X_1)$	$5w(X) + w(\approx) + w(f) + 2w(g) + w(h)$
$\neg(h(X_1) \approx h(X_2)) \vee X_1 \approx X_2$	$4w(X) + 2w(\approx) + 2w(h)$

Table 1: Examples of clauses and their symbol-counting weights

set and put into the active set. It is made to participate in inferences with all the active clauses, and the newly generated clauses are added to the passive set. The proof search is completed as soon as the empty clause is inferred.

In a typical proof search, selecting inauspicious clauses quickly clutters the active set with useless clauses, slowing the proof search considerably and thus reducing the chance of finding a proof in a reasonable time. On the contrary, accurately selecting proof clauses leads to quick success. Thus, the quality of the clause selection heuristic is crucial for the performance of the prover [23]. Most provers interleave several priority queues of passive clauses, each prioritized by a distinct criterion. Commonly utilized criteria include clause age, which favors early-inferred clauses, and symbol count, often referred to as clause weight. These queues are commonly interleaved at a predetermined ratio. For example, in its default configuration, Vampire interleaves age- and weight-based selection at a ratio known as the age-weight ratio [15].

While the proof search may also conclude by depleting the passive set and thus demonstrating the satisfiability of the input clause set, in this work, we are only concerned with refutation proofs and only consider a proof search successful if it infers the empty clause.

3 Task Definition

3.1 Symbol-Counting Clause Weight

Extended signature. Consider a problem P defined over the signature Σ_P (the set of predicate and function symbols in P). We define the *extended signature* Σ_P^+ by extending Σ_P with two fresh symbols: $\approx$ (for the equality predicate) and X (a generic variable). In the following text, we use the term "symbol" in the broader sense of any member of Σ_P^+.

Symbol occurrence count. Given a clause C over Σ_P, we define $S_C : \Sigma_P^+ \to \mathbb{N}$ as the *symbol occurrence count operator* of C: The values of $S_C(\approx)$ and $S_C(s)$ for $s \in \Sigma_P$ are to be computed in an obvious way and $S_C(X)$ is the total number of positions in C at which a variable occurs.

Symbol and clause weight. Any mapping $w : \Sigma_P^+ \to \mathbb{R}^{\geq 1}$ is a *symbol weight assignment* for P. By extension, $W(C) = \sum_{s \in \Sigma_P^+} S_C(s) \cdot w(s)$ is the *symbol-counting weight* of clause C induced by the symbol weight assignment w.

See examples of symbol-counting clause weights in Table 1.

Fairness. When W is used as a clause selection heuristic in an ATP, a weight-based selection chooses the clause that minimizes W among the passive clauses. While we could, in principle, allow symbol weights smaller than 1 (or even negative), constraining the symbol weights with a lower bound greater than 0 makes the induced clause selection heuristic *fair* [23]: For each

$W' \in \mathbb{R}$, there is only a finite number of clauses (up to variable renaming) that weigh at most W', so no clause inferred during the proof search is denied selection indefinitely.[1]

The standard weight criterion used by Vampire assigns the weight 1 to each symbol [15].

3.2 Main Task

We assume a fixed distribution of FOL problems (the target distribution) and a fixed FOL ATP (the target prover) with a fixed time limit (specified in wall-clock time or CPU instructions). Furthermore, we assume that the target prover is a saturation-based prover that selects at least some clauses by a symbol-counting clause weight W instantiated by a configurable symbol weight assignment w.

Our main goal is to create a system that, when presented with a FOL problem P sampled from the target distribution, produces a symbol weight assignment w such that the target prover is likely to solve P within the time limit when using w.

4 Recommender

To solve the main task, we propose a system based on a neural network (NN) that produces a symbol weight assignment. The system is trained on traces of successful proof searches of the target prover on the target problem distribution.

4.1 Input Problem Format

Given a FOL problem, we first convert it to an equisatisfiable CNF problem [17]. The conversion may introduce new function symbols by Skolemizing the existential quantifiers and new predicate symbols by naming some subformulas. In the remainder of this text, we assume that the input problem is specified in CNF.

In addition to the CNF specification of the problem, some metadata are considered part of the input. Specifically, the recommender is given the role of each clause (axiom, assumption, or negated conjecture), and, for each symbol, whether the symbol was introduced during preprocessing (Skolemization or subformula naming) and whether it appears in the conjecture.

4.2 Prediction

4.2.1 Graph Representation of Problems

Given a problem in CNF, we convert it to a graph. Each node in the graph represents a syntactic element of the input problem, such as a clause, an atom, a term, a predicate or function symbol, or a variable. The edges represent direct relations of the syntactic elements; for example, a clause is connected to all the atoms it contains and an atom is connected to a predicate symbol and zero or more arguments. The nodes and edges are typed according to their meaning: The type of a node denotes the role of the corresponding syntactic element (for example, “clause” or “atom”), and the type of an edge denotes the relation it represents (such as “clause contains atom”) and is typically fully specified by the types of the adjacent nodes.

[1] Strictly speaking, interleaving weight-based selection with age-based selection suffices to ensure fairness. However, in practice, we observed that allowing negative symbol weights may have undesirable effects even when age-based selection is present; see Section 5.5.3.

4.2.2 Graph Neural Network

The graph neural network (NN) that constitutes the predictive core of the recommender is a relational graph convolutional network (GCN) [20]. It maintains an embedding (a numeric vector), in our case of length 16, for each node in the input graph.

Initially, the embedding of a node is a trainable vector common to all nodes of one type. In the clause and symbol nodes, the initial embedding is augmented with the metadata extracted from the input problem.[2]

Four message-passing layers follow: In each of these layers, each node embedding is updated by aggregating messages incoming from the node's neighbors. Each message is the output of a trainable affine transformation of the embedding of the source node. The trainable coefficients of the transformation are shared by all edges of the same type in the same layer.[3]

After all four layers are evaluated, the final node embeddings are available. To obtain an output weight $w(s)$ of symbol $s \in \Sigma_P$, the final embedding of the node that represents s is transformed using an affine transformation, the trainable coefficients of which are shared by all symbols, followed by an output activation function. To obtain an output variable weight $w(X)$, the embeddings of all variable nodes are aggregated by summation and transformed by a trainable affine transformation followed by an output activation function. Similarly, the output weight of equality $w(\approx)$ is calculated by aggregating the embeddings of all equality atom nodes.

We use the output activation function $a(x) = 1+\mathrm{softplus}(x)$, where $\mathrm{softplus}(x) = \log(e^x+1)$. This ensures that every output weight is greater than 1, which ensures fairness of the clause selection, as described in Section 3.1. The empirical experiments described in Section 5.5.3 confirm the efficacy of this activation function.

In the way just described, GNN transforms an input problem graph into a symbol weight assignment w. The quality of the assignment can be empirically assessed by trying to solve the input problem using the target prover with a symbol-counting clause selection heuristic induced by w.

4.3 Training

Next, we explain how the NN is trained. Our approach exploits the fact that the symbol weight assignment w predicted by the NN can be used outside of the prover to compute the weight $W(C)$ of an arbitrary clause C and that $W(C)$ is differentiable with respect to the trainable parameters of the NN.

4.3.1 Training Data Collection

We train the NN on traces of successful proof searches. To collect training data, we run the target prover on the training problem set. We use a fixed clause selection heuristic, namely one that interleaves an age-based queue with symbol-counting with uniform symbol weights at the ratio 1:5. We enforce a time limit to curb impractically long proof searches.

From each successful proof search produced this way, we extract all the selected clauses. We divide the selected clauses into two sets: proof clauses $\mathcal{C}_+$ and nonproof clauses $\mathcal{C}_-$, depending on whether or not the clause contributed to the proof. We limit the size of each of these sets to 10 000 clauses by subsampling if necessary.

[2] More precisely, the initial embedding of a node is the concatenation of a node-specific metadata vector (empty in all nodes except clause and symbol nodes; see Section 4.1), and a trainable vector common to all nodes of one type.

[3] The architecture and hyperparameters of the GNN are the same as those used in our earlier work on recommending symbol precedences [2] which we refer the reader to for additional details.

4.3.2 Training Example

We construct a training example from each pair of a proof clause C_+ and a nonproof clause C_- from the Cartesian product $\mathcal{C}_+ \times \mathcal{C}_-$. We interpret such an example as "C_+ should be selected before C_-". Since we know the occurrence counts of variables and symbols in each of these clauses, we can use the weights predicted by the NN to predict clause weights $W(C_+)$ and $W(C_-)$.

We define the predicted probability of C_+ being preferable to C_- according to the NN as $p(C_+, C_-) = \text{sigmoid}(W(C_-) - W(C_+))$, where $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$ is a standard activation function that maps real numbers to the interval $[0, 1]$.

4.3.3 Accuracy

We consider the example (C_+, C_-) classified correctly if and only if $p(C_+, C_-) > 0.5$. Since this occurs if and only if $W(C_-) > W(C_+)$, this ultimately corresponds to the prover selecting C_+ before C_-.

Given a set of examples, the prediction *accuracy* of the NN is defined as the number of correct classifications divided by the total number of examples. While the main task of the NN remains to achieve a strong performance of the target prover, we train the NN on this proxy task of achieving high accuracy on the training data. The proxy task has two crucial practical advantages:

1. Proxy performance evaluation is relatively inexpensive, so we can quickly assess a trained NN using its proxy performance.

2. Since p is differentiable with respect to the trainable parameters of the NN, we can use gradient descent [8] to optimize the proxy performance of the NN.

4.3.4 Loss Function

To conclude the exposition of the training process, we define the loss on training example (C_+, C_-) as the negative log-likelihood:

$$\ell = -\log p(C_+, C_-) = -\log \text{sigmoid}(W(C_-) - W(C_+)) = \log(1 + e^{W(C_+)-W(C_-)})$$

This is a standard choice for binary classification tasks [10].

4.3.5 Training Procedure

We use minibatch stochastic gradient descent [8] to minimize the total loss on the training set. We use a weighted sum to aggregate the loss values of multiple proof searches of different sizes to reflect our intuition that every successful proof search has the same value. Training proceeds in epochs, each using each training example exactly once.

When optimizing the proxy performance, we are effectively training a clause ranking function W based on training examples of the form "$W(C_+)$ should be smaller than $W(C_-)$". The general task of training a NN to rank arbitrary objects has been explored in RankNet [3]. In our case, unlike RankNet, W is a mere proxy metric, not a final goal, so we need to ensure that W represents our main task sufficiently. This representation is realized by the decomposition of W into the symbol-wise components w.

4.4 Evaluation

The quality of the NN can be assessed by measuring proxy performance or empirical performance. *Proxy performance* is the mean prediction accuracy on a set of proof searches (see Section 4.3.3). To determine the *empirical performance* on a set of problems, we run the target prover using symbol weight assignments predicted by the NN for the respective problems. Empirical performance is measured as the number or proportion of proofs found within a fixed proof search instruction limit.

For the experiments reported in the next section, we implemented a prototype recommender. We did not attempt to optimize the recommender to reduce the computational overhead introduced in addition to the proof search. In the version of the recommender we used in the experiments[4], this overhead for solving a single input problem is of the same order as the time limit we used for empirical evaluation (5×10^{10} CPU instructions). When solving a batch of problems, the overhead can be amortized to a great extent, so the recommender can already be useful in such a setting. Further optimization is necessary to make the recommender practically useful for solving isolated problems with time limits in the order of 10^{10} instructions[5] or less.

5 Experiments

To assess the strength of the architecture and training procedure described in Section 4, we performed a collection of experiments. In the experiments, we targeted the first-order fragment of the Thousands of Problems for Theorem Provers (TPTP) problem library as our target problem distribution and the ATP Vampire as our target prover.

5.1 System

We performed our experiments on a computer with a CPU Intel Xeon Gold 6140 (72 cores @ 2.30 GHz) and 502 GiB random access memory (RAM).

5.2 Target Prover

Vampire [15] is a powerful ATP that has shown success especially in proving FOL theorems. We modified Vampire

- to allow exporting the signature and the graph representation of the input problem,
- to print all the clauses selected in the proof search, and
- to allow the user to override the symbol weight assignment for clause selection.[6]

We used Vampire in its default configuration with the following exceptions: the Discount saturation algorithm, age-weight ratio 1:5, no AVATAR [31], and a limit of 5×10^{10} CPU instructions[7]. Using the Discount saturation algorithm, as opposed to the default limited resource strategy [18], removes a prominent source of nondeterminism from the proof search. With Discount, the

[4] Source code of the recommender: https://github.com/filipbartek/vampire-ml/tree/lpar2023
[5] The order of seconds on a contemporary CPU
[6] Source code of the modified Vampire: https://github.com/filipbartek/vampire/tree/lpar2023
[7] In our experimental setup, 5×10^{10} instructions were executed in approximately 16 seconds of wall-clock time.

age-weight ratio 1:5 is stronger than the default 1:1 and increases the significance of weight-based clause selection. Disabling AVATAR is a convenience that facilitates the parsing of the selected clauses.

5.3 Problem Sets

We first collected all the 17 436 problems annotated as clause normal form (CNF) or first-order form (FOF) in TPTP [30] v8.1.2. We set aside 3487 (approximately 20 %) randomly sampled problems as the test set for the final evaluation. We divided the remaining 13 949 problems in a ratio of approximately 80 to 20 into 11 159 training problems and 2202 validation problems.

The validation set was used to measure the performance of the trained NN during the training to compensate for the possible overfitting of the NN to the training set. Analogously, performing the final evaluation on a separate test set ensures that the observed success is not due to overfitting to the validation set.

Due to practical concerns, we restricted our attention to problems whose graph representation has at most 100 000 nodes. Therefore, the training, validation, and test sets have 10 016, 2492, and 3149 problems, respectively.

5.4 Training

First, we ran Vampire on the training and validation problem sets. The proofs found in this way form the training and validation data. Each proof search trace was subsampled to at most 10 000 proof clauses and at most 10 000 nonproof clauses.

We trained a GNN as described in Section 4. Each training epoch was followed by a proxy evaluation on the training and validation problem sets. Every 10 epochs, we performed an empirical evaluation. After approximately 60 hours (160 epochs) of training, we chose the final model based on the performance observed on the validation set.

Since the accuracy (both training and validation) grew smoothly within the first 80 epochs and turned noticeably noisy afterward, we chose the final version of the GNN among the first 80 epochs to avoid accidental overfitting to the validation problem set. The final version has the highest observed number of proofs found on the validation set.

5.5 Results

5.5.1 Empirical Performance

We evaluated the empirical performance of the final trained NN and the standard symbol weight assignment that assigns the weight 1 to each symbol (denoted as baseline). Table 2 and Figure 1 show the results of the evaluation. The GNN improves the performance by approximately 6.6 % over the baseline. Furthermore, Figure 1 shows that the performance of the trained GNN generalizes to instruction limits up to 4 times as large as the limit used to generate the training data (5×10^{10} instructions).

To get a better sense of the scale of this result, we evaluated the baseline with AVATAR enabled. AVATAR is a powerful and sophisticated Vampire feature that aids saturation-based theorem proving using a propositional SAT solver [31]. When we enabled AVATAR under the same conditions as the baseline, we observed an improvement of 5.9 % over the baseline, which is of a magnitude similar to the improvement brought about by our trained GNN.

Configuration	Proofs found /3149	%	Compared to B +	−	%
Trained GNN	1494	47.4 %	+141	−49	+6.6 %
Baseline (B)	1402	44.5 %	+0	−0	+0.0 %
B + AVATAR	1485	47.2 %			+5.9 %
B + Goal-directed	1463	46.5 %			+4.4 %

Table 2: Results of the final empirical evaluation. The reported performance is the number of proofs found on the test set (3149 problems) within 5×10^{10} CPU instructions per proof search.

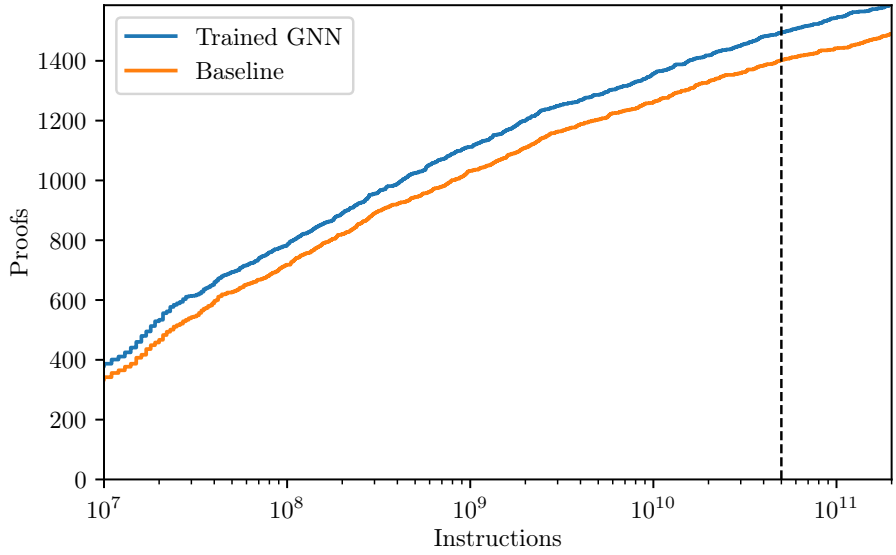


Figure 1: Total number of proofs found on the test set (3149 problems) in a given number of CPU instructions per proof search. The dashed line shows the default evaluation limit of 5×10^{10} instructions that was used to generate the training data. The configurations were evaluated to up to 2×10^{11} instructions.

5.5.2 Iteration Efficiency

Figure 2 shows that the trained GNN is relatively efficient in terms of iterations of the saturation loop. This is an expected outcome: The training indirectly incentivizes the prover to select proof clauses in relatively early iterations, making the proof search conclude in relatively few iterations. There were 1453 test problems that both the GNN and the baseline solved within 2×10^{11} instructions. Of these, 828 were solved in fewer iterations by the GNN, while only 471 were solved in fewer iterations by the baseline.

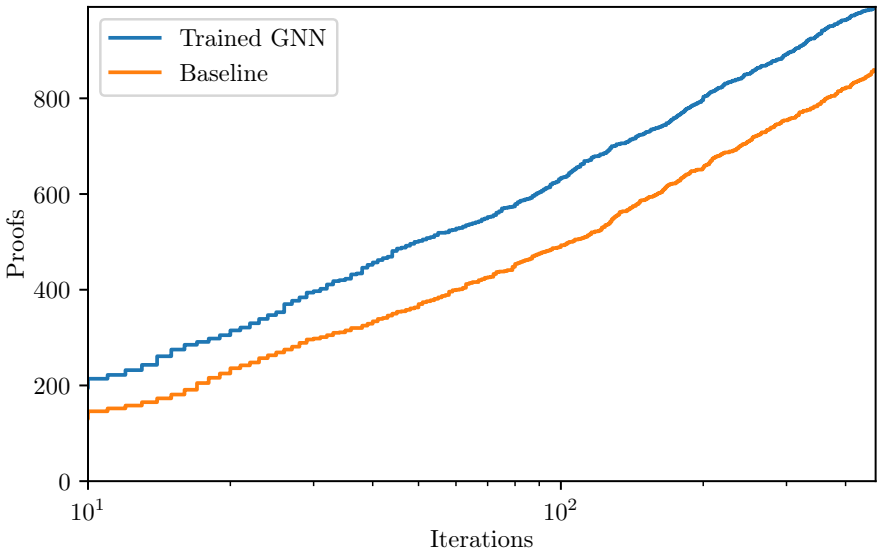


Figure 2: Total number of proofs found on the test set (3149 problems) in a given number of iterations of the saturation loop per proof search. Only proofs found within 2×10^{11} CPU instructions and 463 iterations are included. Proof searches with more than 463 iterations are not shown to ensure that no more than 10 % proofs are obscured from the plot by the instruction limit.

5.5.3 Activation Function Comparison

As described in Section 4.2.2, we forced the output symbol weights to have values greater than 1. To understand what the effect of allowing weights smaller than 1 would be, we trained the GNN for 24 hours with various final activation functions: $a_1(x) = 1 + \text{softplus}(x)$ (default), $a_2(x) = 0.1 + \text{softplus}(x)$ (greater than 0.1), $a_3(x) = \text{softplus}(x)$ (greater than 0), and $a_4(x) = x$ (unconstrained). In almost all observed epochs, the empirical performance on the validation set decreased from a_1 to a_4. Manual inspection of several failed proof searches performed with the weights found with the activation function a_4 confirmed our suspicion that allowing negative symbol weights can often lead to infinite derivation chains of increasingly long clauses with decreasing negative weights.

6 Interpreting the Learned Advice

The details behind the computation performed by a trained network are notoriously hard to interpret. Although this holds for our GNN, too, we at least have the possibility to study its end result—the recommended symbol weights—and to attempt to discover some important features of the recommendations that lead to an efficient proof search. In this section, we report our findings from manually analyzing the GNN symbol weight recommendations and

the corresponding Vampire runs on several TPTP problems. To have the highest chance of noticing impactful features, we selected the problems so that Vampire with recommendations solves each of them fast while the baseline configuration fails (under the limit of 5×10^{10} CPU instructions).

Small conjecture symbols. For most of the problems we examined, symbols appearing in the conjecture were assigned weights very close to the smallest allowed value, 1. At the same time, other symbols were assigned weights ranging from 1 to approximately 10. Having the conjecture symbols weigh less than the rest is a way to make the search goal-directed, a well-known, generally successful strategy [e.g., 23].[8]

We find the fact that the GNN "reinvents" this strategy from the training data quite encouraging. However, it also raises the question of whether this is not all that has been learned. To check this possibility, we reran the baseline configuration in a goal-directed mode.[9] It solved 1463 (46.5 %) problems in the test set, compared to 1494 (47.4 %) solved by the trained GNN (cf. Table 2). In other words, the trained GNN solved 2.1 % more problems than the goal-directed strategy. In summary, while goal-directedness seems to play an important role, it does not explain all the gains obtained through the GNN symbol recommendation.

Bumped-up "Tseitin variables". The TPTP problem `ALG066+1` contains a ground description of a finite algebra with one binary operation `op` on five elements `e0,...,e4` (one of which is a `unit`). Its conjecture is a formula with a complex Boolean structure, for which efficient clausification Vampire introduces 130 new predicate symbols $\texttt{sP}_i$ as names for subformulas [17].

The baseline run on this problem does not finish in a reasonable time, activating thousands of clauses such as `sP126 | sP125 | sP107 | sP88 | sP58`, where the resolution rule systematically "distributes out" parts of the conjecture's Boolean structure (previously abstracted by new names during preprocessing). However, the GNN-advised Vampire finds a proof in 0.25 s and approx. 2500 iterations of the main loop. The search proceeds differently because the GNN assigns high weights (between 2 and 4) to the new predicate symbols $\texttt{sP}_i$.

It seems that the rule discovered here is that it may pay off to postpone working on clauses that are heavy by containing too many subformula names. We note, however, that `ALG066+1`, being ground, becomes trivial also for the baseline when equipped with AVATAR [31], as all the Boolean structure gets then efficiently dealt with by the SAT solver.

Variable weight. We noticed that the trained GNN almost always assigns a low weight (a value very close to 1) to variables. (Recall that in our setting, there is only one global variable occurrence weight for each problem.) This could be interpreted as "Other things being equal, a general clause is more likely to help finish a proof than a specific one.", which is actually obvious when thinking about logical strength and the ultimate goal of finding a contradiction. At the same time, some caution should be in order, as we also know that a clause with more variables provides more opportunities for inferences, which often leads to a more explosive proof search.

Curious to see whether the learned tendency to have variables weigh less than most other symbols was not stopped early by our fairness constraint (stating that any recommended weight must be ≥ 1, cf. Section 4.2.2), we reran Vampire with the GNN symbol weights but overriding

[8] An interesting variant of this idea in the context of equational proving has been given by Smallbone [25].

[9] Specifically, by setting Vampire's *non-goal weight coefficient* option [15] to 5. This makes the search goal-directed by globally multiplying the weight of every clause not derived (directly or indirectly) from the conjecture by 5. The concrete value 5 has been found to lead to good performance in previous experiments on TPTP [26].

variable weight to 0.5. However, this resulted in a decline in overall performance, and the number of problems uniquely solved by this modified configuration was not too high either.

This makes us conclude here that the variable weight of 1 is probably working as a reference value for the GNN recommendations, in the sense that the weights assigned to other symbols are appropriately scaled in relation to this unit, and, in its internal reckoning, they are neither too low nor too high.

Rarely useful axioms? The problem `SET016+1` formalizes the set theory lemma that the first components of equal ordered pairs are equal. The conjecture is stated in the context of 43 common set theory axioms and definitions included from the file `SET005+0.ax`.

Vampire's clausification produces 10 Skolem functions. The last four of these correspond to the universally quantified variables of the conjecture and get recommended weights close to 1 by our GNN. At the other end of the scale, there are the Skolems `sK1` of arity 0 with weight 5.1 and `sK4` of arity 1 with weight 4.1. These correspond to the witnesses from the axioms of infinity and regularity, respectively. We speculate that these relatively "exotic" axioms were rarely, if at all, useful in the proofs used for training (there were 41 training problems that included the axiom file `SET005+0.ax`) and, at the same time, that the GNN was able to recognize them just from their structure and how they relate to the surrounding symbols and other axioms. If these assumptions are correct, assigning the corresponding (uniquely determined) Skolems large weights is a good way to steer the search away from exploring possible inferences with such axioms. However, we did not find a way to easily confirm this rigorously.

7 Future Work

This paper has shown that generalizing the standard clause selection heuristic and automatically determining the symbol weights improves the performance of the prover. Arguably, the clause weight scheme could be generalized further with a negligible computational overhead by a conservative augmentation of the set of clause features used to compute the clause weight, for example, by the number of positive literals[10], the number of distinct variables, or the depth of the clause. Further generalization of the recursively implemented (via summation) symbol-counting clause weight heuristic would yield a recursive neural network (NN) operating on the term structure. An RNN-based clause selection heuristic has been trained by Chvalovský et al. [4] on the proxy task of proof clause classification.

Although allowing clause feature weights smaller than 1 has proven detrimental to the performance of the trained system (see Section 5.5.3), we hypothesize that this effect could be compensated by enriching the training data using failed proof searches from the empirical evaluations.

Using additional or different Vampire base configurations for training data collection and final evaluation, for example, one that enables AVATAR, could yield an even stronger trained prover.

Finally, using constrained logistic regression allows one to find nearly optimal symbol weights (in terms of proxy performance) for one or more successful proof searches with a common signature. It would be interesting to see if we could use such a method to find nearly optimal weights for some interesting domain with a shared signature, such as the Mizar Mathematical Library [9]. In a more general setting, we could collapse the problem-specific symbol counts into a small number of common symbol-counting features such as the number of literals (the sum of

[10] Cf. parameter `pos_mult` of the generic weight functions in E [22].

all predicate and equality occurrence counts), the total number of non-Skolem function symbol occurrences, or the total number of occurrences of predicates introduced in preprocessing. Another way to combine logistic regression with misaligned signatures could involve decomposing the weight of a symbol into common symbol features (such as the arity or an indication of being introduced in preprocessing) and learning, for each symbol category (such as predicate symbols or function symbols), the weights of these symbol features from the data.

8 Conclusion

We proposed and evaluated a symbol weight recommender based on a GNN. The recommender instantiates weights for the weighted symbol occurrence count, a clause selection heuristic we implemented in the FOL ATP Vampire. When trained on a set of proof search traces produced using a baseline Vampire configuration, the recommender outperforms the baseline by 6.6 % in the number of proofs found within a fixed instruction limit. This is, to the best of our knowledge, the first time a machine-learned clause selection heuristic has achieved a comparable improvement on the diverse TPTP problem library.

In this work, we make the following main observations. First, classifying proof-nonproof pairs of clauses is a good proxy task for training a clause selection heuristic. Second, a good symbol weight recommendation can boost the performance of our chosen baseline strategy to a similar degree as enabling the AVATAR architecture. Third, constraining the symbol weights with a nontrivial lower bound helps the performance of a trained recommender significantly. Finally, strong symbol weight assignments tend to assign variable occurrences a small weight.

9 Acknowledgments

We thank Cezary Kaliszyk for discussions. This work was supported by the Czech Science Foundation project no. 20-06390Y (JUNIOR grant), the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/00004, the project RICAIP no. 857306 under the EU-H2020 programme, and the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37. Lastly, we thank the anonymous reviewers for valuable comments.

References

[1] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [19], pages 19–99. ISBN 0-444-50813-9. doi:10.1016/b978-044450813-3/50004-7.

[2] Filip Bártek and Martin Suda. Neural precedence recommender. In Platzer and Sutcliffe [16], pages 525–542. ISBN 978-3-030-79875-8. doi:10.1007/978-3-030-79876-5_30.

[3] Christopher J. C. Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Gregory N. Hullender. Learning to rank using gradient descent. In Luc De Raedt and Stefan Wrobel, editors, *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, volume 119 of *ACM International Conference Proceeding Series*, pages 89–96. ACM, 2005. ISBN 1-59593-180-5. doi:10.1145/1102351.1102363.

[4] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Fontaine [7], pages 197–215. ISBN 978-3-030-29435-9. doi:10.1007/978-3-030-29436-6_12.

[5] Anatoli Degtyarev and Andrei Voronkov. Equality reasoning in sequent-based calculi. In Robinson and Voronkov [19], pages 611–706. ISBN 0-444-50813-9. doi:10.1016/b978-044450813-3/50012-6.

[6] Jörg Denzinger and Stephan Schulz. Learning domain knowledge to improve theorem proving. In Michael A. McRobbie and John K. Slaney, editors, *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, volume 1104 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 1996. ISBN 3-540-61511-3. doi:10.1007/3-540-61511-3_69.

[7] Pascal Fontaine, editor. *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, 2019. Springer. ISBN 978-3-030-29435-9. doi:10.1007/978-3-030-29436-6.

[8] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN 978-0-262-03561-3. URL http://www.deeplearningbook.org/.

[9] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formaliz. Reason.*, 3(2):153–245, 2010. doi:10.6092/issn.1972-5787/1980.

[10] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. ISBN 9780387848570. doi:10.1007/978-0-387-84858-7.

[11] Jan Jakubuv and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2017. ISBN 978-3-319-62074-9. doi:10.1007/978-3-319-62075-6_20.

[12] Jan Jakubuv and Josef Urban. Hammering mizar by learning clause guidance (short paper). In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. ISBN 978-3-95977-122-1. doi:10.4230/LIPIcs.ITP.2019.34.

[13] Jan Jakubuv, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020. ISBN 978-3-030-51053-4. doi:10.1007/978-3-030-51054-1_29.

[14] Cezary Kaliszyk and Josef Urban. Mizar 40 for mizar 40. *J. Autom. Reason.*, 55(3): 245–256, 2015. doi:10.1007/s10817-015-9330-8.

[15] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *lncs*, pages 1–35. Springer, 2013. ISBN 978-3-642-39798-1. doi:10.1007/978-3-642-39799-8_1.

[16] André Platzer and Geoff Sutcliffe, editors. *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, 2021. Springer. ISBN 978-3-030-79875-8. doi:10.1007/978-3-030-79876-5.

[17] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In Christoph Benzmüller, Geoff Sutcliffe, and Raúl Rojas, editors, *GCAI 2016. 2nd Global Conference on Artificial Intelligence, September 19 - October 2, 2016, Berlin, Germany*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016. doi:10.29007/dzfz.

[18] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003. doi:10.1016/S0747-7171(03)00040-3.

[19] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9. URL https://www.sciencedirect.com/book/9780444508133/handbook-of-automated-reasoning.

[20] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, volume 10843 of *Lecture Notes in Computer Science*, pages 593–607. Springer, 2018. ISBN 978-3-319-93416-7. doi:10.1007/978-3-319-93417-4_38.

[21] S. Schulz. Explanation Based Learning for Distributed Equational Deduction. Diplomarbeit in Informatik, Fachbereich Informatik, Universität Kaiserslautern, 1995. URL http://wwwlehre.dhbw-stuttgart.de/~sschulz/PAPERS/Sch95-diplom.ps.gz.

[22] Stephan Schulz. E 2.4 user manual. EasyChair preprint no. 2272, Manchester, 2020. URL https://easychair.org/publications/preprint/8dss.

[23] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2016. ISBN 978-3-319-40228-4. doi:10.1007/978-3-319-40229-1_23.

[24] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. In Fontaine [7], pages 495–507. ISBN 978-3-030-29435-9. doi:10.1007/978-3-030-29436-6_29.

[25] Nicholas Smallbone. Twee: An equational theorem prover. In Platzer and Sutcliffe [16], pages 602–613. ISBN 978-3-030-79875-8. doi:10.1007/978-3-030-79876-5_35.

110

[26] Martin Suda. Aiming for the goal with SInE. In Laura Kovács and Andrei Voronkov, editors, *Vampire 2018 and Vampire 2019. The 5th and 6th Vampire Workshops*, volume 71 of *EPiC Series in Computing*, pages 38–44. EasyChair, 2019. URL https://easychair.org/publications/paper/lZfv.

[27] Martin Suda. Improving enigma-style clause selection while learning from history. In Platzer and Sutcliffe [16], pages 543–561. ISBN 978-3-030-79875-8. doi:10.1007/978-3-030-79876-5_31.

[28] Martin Suda. Vampire with a brain is a good ITP hammer. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 192–209. Springer, 2021. ISBN 978-3-030-86204-6. doi:10.1007/978-3-030-86205-3_11.

[29] Martin Suda. Vampire getting noisy: Will random bits help conquer chaos? (system description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 659–667. Springer, 2022. ISBN 978-3-031-10768-9. doi:10.1007/978-3-031-10769-6_38.

[30] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 59(4), December 2017. ISSN 0168-7433. doi:10.1007/s10817-017-9407-7.

[31] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014. ISBN 978-3-319-08866-2. doi:10.1007/978-3-319-08867-9_46.

Chapter 5

Regularization in Spider-Style Strategy Discovery and Schedule Construction

In the research presented below [4], my co-authors and I created a system that automatically generates strong and mutually complementary strategies (configurations) for the automatic theorem prover Vampire. Using approximately 1000 strategies generated by this system, we constructed strong strategy schedules that generalize well to unseen problems.

A greedy algorithm is at the core of the schedule construction process. The algorithm is parameterized with several regularization options. These new variants of the algorithm and an analysis of their regularizing strength constitute the main contributions presented in the paper.

The data we used to build the schedules (strategies and their performance measurements) has been published [6]. The dataset is potentially interesting for research on static algorithm scheduling and algorithm selection.

Regularization in Spider-Style Strategy Discovery and Schedule Construction

Filip Bártek[1,2( )], Karel Chvalovský[1], and Martin Suda[1]

[1] Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague, Prague, Czech Republic
{filip.bartek,karel.chvalovsky,martin.suda}@cvut.cz
[2] Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic

Abstract. To achieve the best performance, automatic theorem provers often rely on schedules of diverse proving strategies to be tried out (either sequentially or in parallel) on a given problem. In this paper, we report on a large-scale experiment with discovering strategies for the Vampire prover, targeting the FOF fragment of the TPTP library and constructing a schedule for it, based on the ideas of Andrei Voronkov's system Spider. We examine the process from various angles, discuss the difficulty (or ease) of obtaining a strong Vampire schedule for the CASC competition, and establish how well a schedule can be expected to generalize to unseen problems and what factors influence this property.

Keywords: Saturation-based theorem proving · Proving strategies · Strategy schedule construction · Vampire

1 Introduction

In 1997 at the CADE conference, the automatic theorem prover Gandalf [30] surprised its contenders at the CASC-14 competition [29] and won the MIX division there. One of the main innovations later identified as a key to Gandalf's success was the use of multiple theorem proving strategies executed sequentially in a time-slicing fashion [31,32]. Nowadays, it is well accepted that a single, universal strategy of an Automatic Theorem Prover (ATP) is invariably inferior, in terms of performance, to a well-chosen portfolio of complementary strategies, most of which do not even need to be complete or very strong in isolation.

Many tools have already been designed to help theorem prover developers discover new proving strategies and/or to combine them and construct proving schedules [7,9,12,16,21,24,33,34]. For example, Schäfer and Schulz employed genetic algorithms [21] for the invention of strong strategies for the E prover [23], Urban developed BliStr and used it to significantly strengthen strategies for the same prover via iterative local improvement and problem clustering [33],

Manuscript with additional appendices [1]: https://arxiv.org/abs/2403.12869.

C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 194–213, 2024.
https://doi.org/10.1007/978-3-031-63498-7_12

and, more recently, Holden and Korovin applied similar ideas in their HOS-ML system [7] to produce schedules for iProver [14]. The last work mentioned—as well as, e.g., MaLeS [16]—also include a component for *strategy selection*, the task of predicting, based on the input problem's features, which strategy will most likely succeed on it. (Selection is an interesting topic, which is, however, orthogonal to our work and will not be further discussed here.)

For the Vampire prover [15], schedules were for a long time constructed by Andrei Voronkov using a tool called Spider, about which little was known until recently. Its author finally revealed the architectural building blocks of Spider and the ideas behind them at the Vampire Workshop 2023, declaring Spider "a secret weapon behind Vampire's success at the CASC competitions" and "probably the most useful tool for Vampire's support and development" [34]. Acknowledging the importance of strategies for practical ATP usability, we decided to analyze this powerful technology on our own.

In this paper, we report on a large-scale experiment with discovering strategies for Vampire, based on the ideas of Spider (recalled in Sect. 2.1).[1] We target the FOF fragment of the TPTP library [28], probably the most comprehensive benchmark set available for first-order theorem proving. As detailed in Sect. 3, we discover and evaluate (on all the FOF problems) more than 1000 targeted strategies to serve as building blocks for subsequent schedule construction.

Research on proving strategies is sometimes frowned upon as mere "tuning for competitions". While we briefly pause to discuss this aspect in Sect. 4, our main interest in this work is to establish how well a schedule can be expected to generalize to unseen problems. For this purpose, we adopt the standard practice from statistics to randomly split the available problems into a train set and a test set, construct a schedule on one, and evaluate it on the other. In Sect. 6, we then identify several techniques that *regularize*, i.e., have the tendency to improve the test performance while possibly sacrificing the training one.

Optimal schedule construction under some time budget can be expressed as a mixed integer program and solved (given enough time) using a dedicated tool [7,24]. Here, we propose to instead use a simple heuristic from the related set cover problem [3], which leads to a polynomial-time greedy algorithm (Sect. 5). The algorithm maintains the important ability to assign different time limits to different strategies, is much faster than optimal solving (which may overfit to the train set in some scenarios), allows for easy experimentation with regularization techniques, and, in a certain sense made precise later, does not require committing to a single predetermined time budget.

In summary, we make the following main contributions:

– We outline a pragmatic approach to schedule construction that uses a greedy algorithm (Sect. 5), contrasting it with optimal schedules in terms of the quality of the schedules and the computational resources required for their construction (Sect. 6.2). In particular, our findings demonstrate a relative efficacy of the greedy approach for datasets similar to our own.

[1] Not claiming any credit for these, potential errors in the explanation are ours alone.

– Leveraging the adaptability of the greedy algorithm, we introduce a range of regularization techniques aimed at improving the robustness of the schedules in unseen data (Sect. 6). To the best of our knowledge, this represents the first systematic exploration into regularization of strategy schedules.
– The strategy discovery and evaluation is a computationally expensive process, which in our case took more than twenty days on 120 CPU cores. At the same time, there are further interesting questions concerning Vampire's strategies than we could answer in this work. To facilitate research on this paper's topic, we made the corresponding data set available online [2].

2 Preliminaries

The behavior of Vampire is controlled by approximately one hundred *options*. These options configure the preprocessing and clausification steps, control the saturation algorithm, clause and literal selection heuristics, determine the choice of generating inferences as well as redundancy elimination and simplification rules, and more. Most of these options range over the Boolean or a small finite domain, a few are numeric (integer or float), and several represent ratios.

Every option has a *default* value, which is typically the most universally useful one. Some option settings make Vampire incomplete. This is automatically recognized, so that when the prover finitely saturates the input without discovering a contradiction, it will report "unknown" (rather than "satisfiable").

A *strategy* is determined by specifying the values of all options. A *schedule* is a sequence $(s_i, t_i)_{i=1}^{n}$ of strategies s_i together with assigned time limits t_i, intended to be executed in the prescribed order. We stress that in this work we do not consider schedules that would branch depending on problem features.

2.1 Spider-Style Strategy Discovery and Schedule Construction

We are given a set of problems P and a prover with its space of available strategies $\mathbb{S}$. *Strategy discovery* and *schedule construction* are two separate phases. We work under the premise that the larger and more diverse a set of strategies we first collect, the better for later constructing a good schedule.

Strategy discovery consists of three stages: random probing, strategy optimization, and evaluation, which can be repeated as long as progress is made.

Random Probing. We start strategy discovery with an empty pool of strategies $S = \emptyset$. A straightforward way to make sure that a new strategy substantially contributes to the current pool S is to always try to solve a problem not yet solved (or *covered*) by any strategy collected so far. We repeatedly pick such a problem and try to solve it using a *randomly sampled* strategy out of the totality of all available strategies $\mathbb{S}$. The sampling distribution may be adapted to prefer option values that were successful in the past (cf. Sect. 3.3). This stage is computationally demanding, but can be massively parallelized.

Strategy Optimization. Each newly discovered strategy s, solving an as-of-yet uncovered problem p, will get *optimized* to be as fast as possible at solving p. One explores the strategy neighborhood by iterating over the options (possibly in several rounds), varying option values, and committing to changes that lead to a (local) improvement in terms of solution time or, as a tie-breaker, to a default option value where time differences seem negligible. We evaluate the impact of this stage in Sect. 3.4.

Strategy Evaluation. In the final stage of the discovery process, having obtained an optimized version s' of s, we evaluate s' on all our problems P. (This is another computationally expensive, but parallelizable step.) Thus, we enrich our pool and update our statistics about covered problems. Note that every strategy s' we obtain this way is associated with the problem $p_{s'}$ for which it was originally discovered. We will call this problem the *witness problem* of s'.

Schedule Construction can be tried as soon as a sufficiently rich (cf. Sect. 3) pool of strategies is collected. Since we, for every collected strategy, know how it behaves on each problem, we can pose schedule construction as an optimization task to be solved, e.g., by a (mixed) integer programming (MIP) solver.

In more detail: We seek to allocate time slices $t_s > 0$ to some of the strategies $s \in S$ to cover as many problems as possible while remaining in sum below a given time budget T [7,24]. Alternatively, we may try to cover all the problems known to be solvable in as little total time as possible.[2] In this paper, we describe an alternative schedule construction method based on a greedy heuristic, with a polynomial running time guarantee and other favorable properties (Sect. 5).

2.2 CPU Instructions as a Measure of Time

We will measure computation time in terms of the number of user instructions executed (as available on Linux systems through the `perf` tool). This is, in our experience, more precise and more stable (on architectures with many cores and many concurrently running processes) than measuring real time.[3]

In fact, we report *megainstructions (Mi)*, where 1 Mi = 2^{20} instructions reported by `perf`. On contemporary hardware, 2000 Mi will typically get used up in a bit less than a second and 256 000 Mi in around 2 min of CPU time. We also set 1 Mi as the granularity for the time limits in our schedules.

3 Strategy Discovery Experiment

Following the recipe outlined in Sect. 2.1, we set out to collect a pool of Vampire (version 4.8) strategies covering the first-order form (FOF) fragment of the

[2] Strictly speaking, these only give us a set of strategy-time pairs (as opposed to a sequence). However, the strategies can be ordered heuristically afterward.

[3] For a more thorough motivation, see Appendix A of [26].

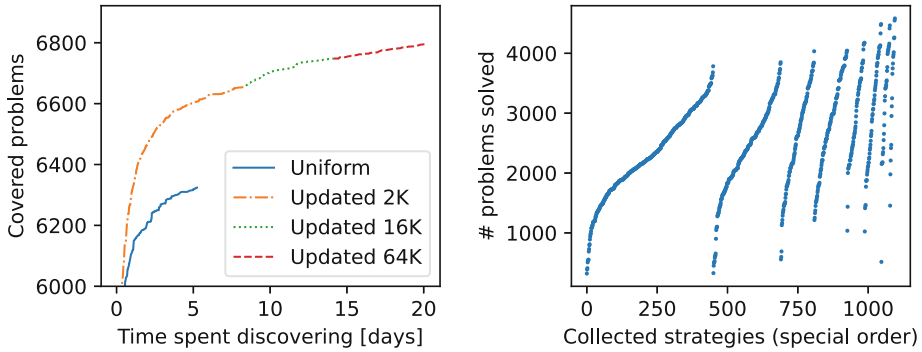


Fig. 1. Strategy discovery. *Left:* problem coverage growth in time (uniform strategy sampling distribution vs. an updated one). *Right:* collected strategies ordered by limit (2000, 4000, ..., 256 000 Mi) and, secondarily, by how many problems can each solve.

TPTP library [28] version 8.2.0. We focused only on proving, so left out all the problems known to be satisfiable, which left us with a set P of 7866 problems. Parallelizing the process where possible, we strived to fully utilize 120 cores (AMD EPYC 7513, 3.6 GHz) of our server equipped with 500 GB RAM.

We let the process run for a total of 20.1 days, in the end covering 6796 problems, as plotted in Fig. 1 (left). The effect of diminishing returns is clearly visible; however, we cannot claim we have exhausted all the possibilities. In the last day alone, 8 strategies were added and 9 new problems were solved.

The rest of Fig. 1 is gradually explained in the following as we cover some important details regarding the strategy discovery process.

3.1 Initial Strategy and Varying Instruction Limits

We seeded the pool of strategies by first evaluating Vampire's default strategy for the maximum considered time limit of 256 000 Mi, solving 4264 problems out of the total 7866.

To save computation time, we did not probe or evaluate all subsequent strategies for this maximum limit. Instead, to exponentially prefer low limits to high ones, we made use of the Luby sequence[4] [18] known for its utility in the restart strategies of modern SAT solvers. Our own use of the sequence was as follows.

The lowest limit was initially set to 2000 Mi and, multiplying the Luby sequence members by this number, we got the progression 2000, 2000, 4000, 2000, 2000, 4000, 8000, ... as the prescribed limits for subsequent probe iterations. This sequence reaches 256 000 Mi for the first time in 255 steps. At that point, we stopped following the Luby sequence and instead started from the beginning (to avoid eventually reaching limits higher than 256 000 Mi).

After four such cycles, the lowest, that is 2000 Mi, limit probes stopped producing new solutions (a sampling timeout of 1 h per iteration was imposed).

[4] https://oeis.org/A182105.

Here, after almost 8.5 d, the "updated 2K" plot ends in Fig. 1 (left). We then increased the lowest limit to 16 000 Mi and continued in an analogous fashion for 155 iterations and 5.7 more days ("updated 16K") and eventually increased the lowest limit to 64 000 Mi ("updated 64K") until the end.

Figure 1 (right) is a scatter plot showing the totality of 1096 strategies that we finally obtained and how they individually perform. The primary order on the x axis is by the limit and allows us to make a rough comparison of the number of strategies in each limit group (2000 Mi, 4000 Mi, ..., 256 000 Mi, from left to right). It is also clear that many strategies (across the limit groups) are, in terms of problem coverage, individually very weak, yet each at some point contributed to solving a problem considered (at that point) challenging.

3.2 Problem Sampling

While the guiding principle of random probing is to constantly aim for solving an as-of-yet unsolved problem, we modified this criterion slightly to produce a set of strategies better suited for an unbiased estimation of schedule performance on unseen problems (as detailed in the second half of this paper).

Namely, in each iteration i, we "forgot" a random half P_i^F of all problems P, considered only those strategies (discovered so far) whose witness problem lies in the remaining half $P_i^R = P \setminus P_i^F$, and aimed for solving a random problem in P_i^R not covered by any of these strategies. This likely slowed the overall growth of coverage, as many problems would need to be covered several times due to the changing perspective of P_i^R. However, we got a (probabilistic) guarantee that any (not too small) subset $P' \subseteq P$ will contain enough witness problems such that their corresponding strategies will cover P' well.

3.3 Strategy Sampling

We sampled a random strategy by independently choosing a random value for each option. The only exception were dependent options. For example, it does not make sense to configure the AVATAR architecture (changing options such as `acc`, which enables congruence closure under AVATAR) if the main AVATAR option (`av`) is set to `off`. Such complications can be easily avoided by following, during the sampling, a topological order that respects the option dependencies. (For example, we sample `acc` only after the value `on` has been chosen for `av`.)

Even under the assumption of option independence, the mean time in which a random strategy solves a new problem can be strongly influenced by the value distributions for each option. This is because some option values are rarely useful and may even substantially reduce the prover performance, for example, if they lead to a highly incomplete strategy.[5] Nevertheless, not to preclude the

[5] An extreme example is turning off *binary resolution*, the main inference for non-equational reasoning. This can still be useful, for instance when replaced by *unit resulting resolution* [15], but our sampling needs to discover this by chance.

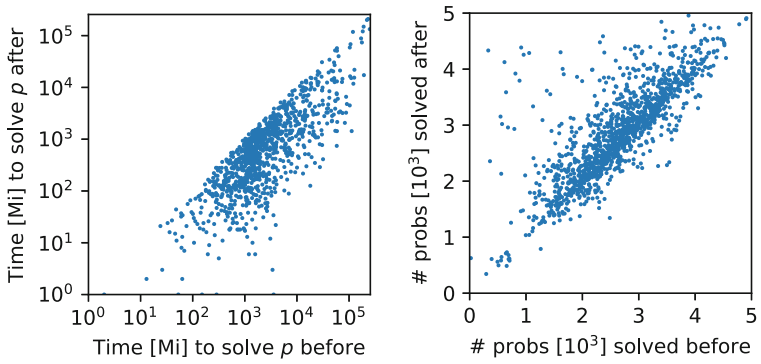


Fig. 2. Strategy optimization scatter plots. *Left:* time needed to solve strategy's witness problem (a log–log plot). *Right:* the total number of problems (in thousands) solved.

possibility of discovering arbitrarily wild strategies, we initially sampled every option uniformly where possible.[6]

Once we collected enough strategies,[7] we updated the frequencies for sampling finite-domain options (which make up the majority of all options) by counting how many times each value occurred in a strategy that, at the moment of its discovery, solved a previously unsolved problem. (This was done before a strategy got optimized. Otherwise the frequencies would be skewed toward the default, especially for option values that rarely help but almost never hurt.)

The effect of using an updated sampling distribution for strategy discovery can be seen in Fig. 1 (left). We ran two independent versions of the discovery process, one with the uniform distribution and one with the updated distribution. We abandoned the uniform one after approximately 5 d, by which time it had covered 6324 problems compared to 6607 covered with the help of the updated distribution at the same mark. We can see that the rate at which we were able to solve new problems became substantially higher with the updated distribution.

3.4 Impact of Strategy Optimization

Once random probing finds a new strategy s that solves a new problem p, the task of optimization (recall Sect. 2.1) is to search the option-value neighborhood of s for a strategy s' that solves p in as few instructions as possible and preferably uses default option values (where this does not compromise performance on p).

The impact of optimization is demonstrated in Fig. 2. On the left, we can see that, almost invariably, optimization substantially improves the performance of the discovered strategy on its witness problem p. The geometric mean of the improvement ratio we observed was 4.2 (and the median 3.2). The right

[6] Exceptions were: 1. The ratios: e.g., for `age_weight_ratio` we sampled uniformly its binary logarithm (in the range −10 and 4) and turned this into a ratio afterward (thus getting values between 1 : 1024 and 16 : 1); 2. Unbounded integers (an example being the *naming threshold* [20]), for which we used a geometric distribution instead.

[7] This was done in an earlier version of the main experiment.

scatter plot shows the overall performance of each strategy.[8] Here, the observed improvement is ×1.09 on average (median 1.03), and the improvement is solely an effect of setting option values to default where possible (without this feature, we would get a geometric mean of the improvement 0.84 and median 0.91). In this sense, the tendency to pick default values regularizes the strategies, making them more powerful also on problems other than their witness problem.

3.5 Parsing Does Not Count

When collecting the performance data about the strategies, we decided to ignore the time it takes Vampire to parse the input problem. This was also reflected in the instruction limiting, so that running Vampire with a limit of, e.g., 2000 Mi would allow a problem to be solved if it takes at most 2000 Mi on top of what is necessary to parse the problem.

The main reason for this decision is that Vampire, in its strategy scheduling mode, starts dispatching strategies only after having parsed the problem, which is done only once. Thus, from the perspective of individual strategies, parsing time is a form of a sunk cost, something that has already been paid.

Although more complex approaches to taking parse time into account when optimizing schedules are possible, we in this work simply pretend that problem parsing always costs 0 instructions. This should be taken into account when interpreting our simulated performance results reported next (in Sect. 4, but also in Sect. 6.2).

4 One Schedule to Cover Them All

Having collected our strategies, let us pretend that we already know how to construct a schedule (to be detailed in Sect. 5) and use this ability to answer some imminent questions, most notably: How much can we now benefit?

Figure 3 plots the cumulative performance (a.k.a. "cactus plot") of schedules we could build after 2 h, 6 h, 1 day, and full 20.1 days of the strategy discovery. The dashed vertical line denotes the time limit of 256 000 Mi, which roughly corresponds to a 2-minute prover run. For reference, we also plot the behavior of Vampire's default strategy. We can see that already after two hours of strategy discovery, we could construct a schedule improving on the default strategy by 26% (from 4264 to 5403 problems solved). Although the added value per hour spent searching gradually drops, the 20.1 days schedule is still 4% better than the 1 day one (improving from 6197 to 6449 at 256 000 Mi).

The plot's x-axis ends at 8·256 000 Mi, which roughly corresponds to the time limit used by the most recent CASC competitions [27] in the FOF division (i.e., 2 min on 8 cores). The strongest schedule shown in the figure manages to

[8] In a previous version of the main experiment, we evaluated each strategy both before and after optimization, which gave rise to this plot.

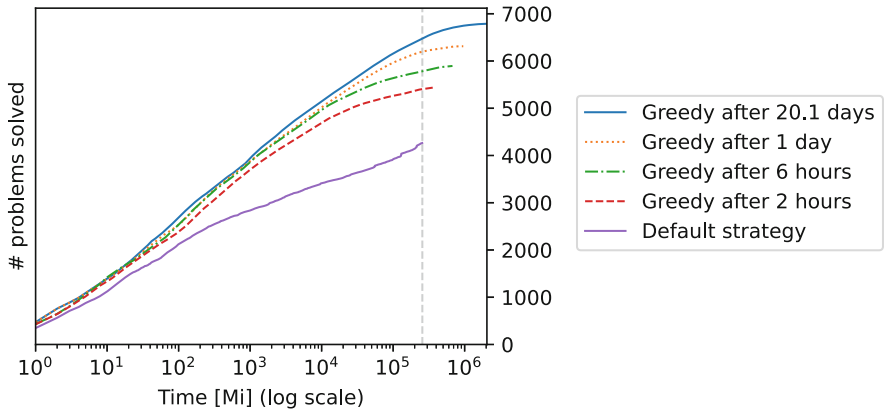


Fig. 3. Cumulative performance of several greedy schedules, each using a subset of the discovered strategies as gathered in time, compared with Vampire's default strategy

solve 6789 problems (of the 6796 covered in total) at that mark.[9] We remark that this schedule, in the end, employs only 577 of the 1096 available strategies, which points towards a noticeable redundancy in the strategy discovery process.

One way to fit all the solvable problems below the CASC budget would be to use a standard trick and split the totality of problems P into two or more easy-to-define syntactic classes (e.g., Horn problems, problems with equality, large problems, etc.) and construct dedicated schedules for each class in isolation. The prover can then be dispatched to run an appropriate schedule once the input problem features are read. We do not explore this option here. Intuitively, by splitting P into smaller subsets, the risk of overfitting to just the problems for which the strategies were discovered increases, and we mainly want to explore here the opposite, the ability of a schedule to generalize to unseen problems.

5 Greedy Schedule Construction

Having collected a set of strategies S and evaluated each on the problems in P, let us by $E^s_p : S \times P \to \mathbb{N} \cup \{\infty\}$ denote the *evaluation matrix*, which records the obtained solutions times (and uses ∞ to signify a failure to solve a problem within the evaluation time limit used). Given a time budget T, the *schedule construction problem* (SCP) is the task of assigning a time limit $t_s \in \mathbb{N}$ to every strategy $s \in S$, such that the number of covered problems

$$\left| \bigcup_{s \in S} \{p \in P \mid E^s_p \leq t_s\} \right|,$$

subject to the constraint $\sum_{s \in S} t_s \leq T$, is maximized.

[9] It is possible to solve all 6796 covered problems with a schedule that spans 2 582 228 Mi. This is the optimum length – no shorter schedule solving all covered problems exists.

Algorithm 1. Greedy schedule construction (base version)

Input: Problems P, strategies S, solution times E_p^s, time budget T
Output: Pre-schedule $t_s : S \to \mathbb{N}$

1: $t_s \leftarrow \mathbf{0}$ ▷ Start with zeros everywhere
2: $T' \leftarrow T, P' \leftarrow P$ ▷ Remaining budget, remaining problems
3: **while** the maximum just below is positive **do**
4: $s, t \leftarrow \operatorname{argmax}_{s \in S, t \in \mathbb{N}, 0 < (t - t_s) \leq T'} |\{p \in P' \mid E_p^s \leq t\}|/(t - t_s)$
5: $T' \leftarrow T' - (t - t_s)$ ▷ Update the remaining budget
6: $t_s \leftarrow t$ ▷ Extend the pre-schedule
7: $P' \leftarrow \{p \in P' \mid E_p^s > t\}$ ▷ Remove covered problems from P'

To obtain a schedule as a sequence (as defined in Sect. 2), we would need to order the strategies having $t_s > 0$. This can, in practice, be done in various ways, but since the order does not influence the predicted performance of the schedule under the budget T, we keep it here unspecified (and refer to the mere time assignment t_s as a *pre-schedule* where the distinction matters).

Although it is straightforward to encode SCP as a mixed integer program and attempt to solve it exactly (though it is an NP-hard problem), an adaptation of a greedy heuristic from a closely related (budgeted) maximum coverage problem [3,13] works surprisingly well in practice and runs in time polynomial in the size of E_p^s. The key idea is to greedily maximize the number of newly covered problems *divided* by the amount of time this additionally requires.

Algorithm 1 shows the corresponding pseudocode. It starts from an empty schedule t_s and iteratively extends it in a greedy fashion. The key criterion appears on line 4. Note that this line corresponds to an iteration over all available strategies S and, for each strategy s, all meaningful time limits (which are only those where a new problem gets solved by s, so their number is bounded by $|P|$).

Algorithm 1 departs from the obvious adaptation of the above-mentioned greedy algorithm for the set covering problem [3] in that we allow extending a slice of a strategy s that is already included in the schedule (that is, has $t_s > 0$) and "charge the extension" only for the additional time it claims (i.e., $t - t_s$). This *slice extension* trick turns out to be important for good performance.[10]

5.1 Do We Need a Budget?

A budget-less version of Algorithm 1 is easy to obtain (imagine T being very large). When running on real-world E_p^s (from evaluated Vampire strategies), we noticed that the length of a typical extension $(t - t_s)$ tends to be small relative to the current used-up time $\sum_{s \in S} t_s$ and that the presence of a budget starts affecting the result only when the used-up time comes close to the budget.

As a consequence, if we run a budget-less version and, after each iteration, record the pair $(\sum_{s \in S} t_s, |P \setminus P'|)$, we get a good estimate (in a single run) of how the algorithm would perform for a whole (densely inhabited) sequence of

[10] The degree of importance of slice extension can be observed in Fig. 5 in Appendix A.

relevant budgets. This is how the plot in Fig. 3 was obtained. Note that this would be prohibitively expensive to do when trying to solve the SCP optimally.

We can also use this observation in an actual prover. If we record and store a journal of the budget-less run, remembering which strategy got extended in each iteration and by how much, we can, given a concrete budget T, quickly replay the journal just to the point before filling up T, and thus get a good schedule for the budget T without having to optimize specifically for T.

6 Regularization in Schedule Construction

To estimate the future performance of a constructed schedule on previously unseen problems, we adopt the standard methodology used in statistics, randomly split our problem set P into a train set P_{train} and a test set P_{test}, construct a schedule for the first, and evaluate it on the second.

To reduce the variance in the estimate, we use many such random splits and average the results. In the experiments reported in the following, we actually compute an average over several rounds of 5-fold cross-validation [6]. This means that the size of P_{train} is always 80.0 % and the size P_{test} 20.0% of our problem set P. However, we *re-scale* the reported train and test performance back to the size of the whole problem set P to express them in units that are immediately comparable. We note that the reported performance is obtained though simulation, i.e., it is only based on the evaluation matrix E^s_p.

Training Strategy Sets. We retroactively simulate the effect of discovering strategies only for current training problems P_{train}. Given our collected pool of strategies S, we obtain the training strategy set S_{train} by excluding those strategies from S whose witness problem lies outside P_{train} (cf. Sect. 3.2). When a schedule is optimized on the problem set P_{train}, the training data consists of the results of the evaluations of strategies S_{train} on problems P_{train}.

6.1 Regularization Methods

We propose several modifications of greedy schedule construction (Algorithm 1) with the aim of improving its performance on unseen problems (the test set performance) while possibly sacrificing some of its training performance.

With the base version, we observed that it could often solve more test problems by assigning more time to strategies introduced into the schedule in early iterations, at the expense of strategies added later (the latter presumably covering just a few expensive training problems and being over-specialized to them). Most of the modifications described next assign more time to strategies added during early iterations, each according to a different heuristic.

Slack. The most straightforward regularization we explored extends each non-zero strategy time limit t_s in the schedule by multiplying it by the multiplicative slack $w \geq 1$ and adding the additive slack $b \in \{0, 1, \ldots\}$. For each

$t_s > 0$, the new limit t'_s is therefore $t_s \cdot w + b$. To avoid overshooting the budget, we keep track of the total length of the extended schedule during the construction (implementation details are slightly more complicated but not immediately important). The parameters w and b control the degree of regularization, and with $w = 1$ and $b = 0$, we get the base algorithm.

Temporal Reward Adjustment. In each iteration of the base greedy algorithm, we select a combination of strategy s and time limit t that maximizes the number of newly solved problems n per time t. Intuitively, the relative degree to which these two quantities influence the selection is arbitrary. To allow stressing n more or less with respect to t, we exponentiate n by a regularization parameter $\alpha \geq 0$, so the decision criterion becomes $\frac{n^\alpha}{t}$.

For small values of α, the algorithm values the time more and becomes eager to solve problems early. For large values of α, on the other hand, the algorithm values the problems more and prefers longer slices that cover more problems. For example, for $\alpha = 1.5$, the algorithm prefers solving 2 problems in 5000 Mi to solving 1 problem in 2000 Mi. Compare this to $\alpha = 1$ (the base algorithm), which would rank these slices the other way around.

Diminishing Problem Rewards. By covering a training problem with more than one strategy, we cover it robustly: When a similar testing problem is solved by only one of these strategies, the schedule still manages to solve it. However, the base greedy algorithm does not strive to cover any problem more than once: as soon as a problem is covered by one strategy, this problem stops participating in the scheduling criterion. This is the case even when covering the problem again would cost relatively little time.

Regularization by diminishing problem rewards covers problems robustly by rewarding strategy s not only by the number of *new* problems it covers but also by the problems covered by s that are already covered by the schedule. This is achieved by modifying the slice selection criterion. Instead of maximizing the number of new problems solved per time, we maximize the total reward per time, which is defined as follows: Each problem contributes the reward β^k, where k is the number of times the schedule has covered the problem and β is a regularization parameter ($0 \leq \beta \leq 1$). We define $0^0 = 1$ so that $\beta = 0$ preserves the original behavior of the base algorithm.

For example, for $\beta = 0.1$, each problem contributes the reward 1 the first time it is covered, 0.1 the second time, 0.01 the third time, etc. Informally, the algorithm values covering a problem the second time in time t as much as covering a new problem in time $10 \cdot t$.

These modifications are independent and can be arbitrarily combined.

6.2 Experimental Results

We evaluated the behavior of the previously proposed techniques using three time budgets: 16 000 Mi (≈8 s), 64 000 Mi (≈32 s), and 256 000 Mi (≈2 min).

Optimal Schedule Constructor. In the existing approaches to the construction of strategy schedules [7,24], it is common to encode the SCP (see Sect. 5) as a mixed-integer program and use a MIP solver to find an exact solution. We implemented such an optimal schedule construction (OSC) by encoding the problem[11] in Gurobi [5] (ver. 10.0.3) and compared OSC to the base greedy schedule construction (Algorithm 1) on 10 random 80 : 20 splits.

For the budget of 256 000 Mi, it takes Gurobi over 16 h to find an optimal schedule, whereas the greedy algorithm finds a schedule in less than a minute. The optimal schedule solves, on average, 45.0 (resp. 8.5) more problems than the greedy schedule on P_{train} (resp. on P_{test}) when re-scaled to $|P|$. For the 16 000 Mi and 64 000 Mi budgets, Gurobi does not solve the optimal schedule within a reasonable time limit. For example, after 24 h, the relative gaps between the lower and upper objective bound are 5.38 % and 1.43 %, respectively. This makes the OSC impractical to use as a baseline for our regularization experiments.[12]

Regularization of the Greedy Algorithm. To estimate the performance of the proposed regularization methods, we evaluated each variant on 50 random splits (10 times 5-fold cross-validation). We assessed the algorithm's response to each regularization parameter in isolation. For each parameter, we evaluated regularly spaced values from a promising interval covering the default value ($b = 0$, $w = 1$, $\alpha = 1$, $\beta = 0$). Figure 4 demonstrates the effect of these variations on the train and test performance for the budget 64 000 Mi.[13]

Temporal reward adjustment was the most powerful of the regularizations, improving test performance for all the evaluated values of α between 1.1 and 2.0. Surprisingly, the values 1.1 and 1.2 also improved the train performance, suggesting that the default greedy algorithm is too time-aggressive on our dataset.

Table 1 compares the performance of notable configurations of the greedy algorithm. Specifically, we include evaluations of the base greedy algorithm and the best of the evaluated parameter values for each of the regularizations. The table also illustrates the effect of regularizations on the computational time of the greedy schedule construction: $\beta > 0$ slows the procedure down and $\alpha > 1$ speeds it up.

In a subsequent experiment, we searched for a strong combination of regularizations by local search from the strongest single-parameter regularization ($\alpha = 1.7$). This yielded a negligible improvement over $\alpha = 1.7$: The best observed test performance was 5707 ($\alpha = 1.7$ and $b = 30$), compared to 5704 of $\alpha = 1.7$.

Finally, we briefly explored the interactions between the budget and the optimal values of the regularization parameters. For each of the three budgets of interest and each of the regularization parameters, we identified the best param-

[11] We used a straightforward encoding similar to the encoding described by Holden and Korovin [7].

[12] A better encoding and solver settings may improve on this. However, we suspect the problem to be hard; we tried some modifications with similar (or worse) results.

[13] This budget seems to be the most practically relevant (e.g., for the application in interactive theorem provers). The other two budgets are detailed in Appendix A.

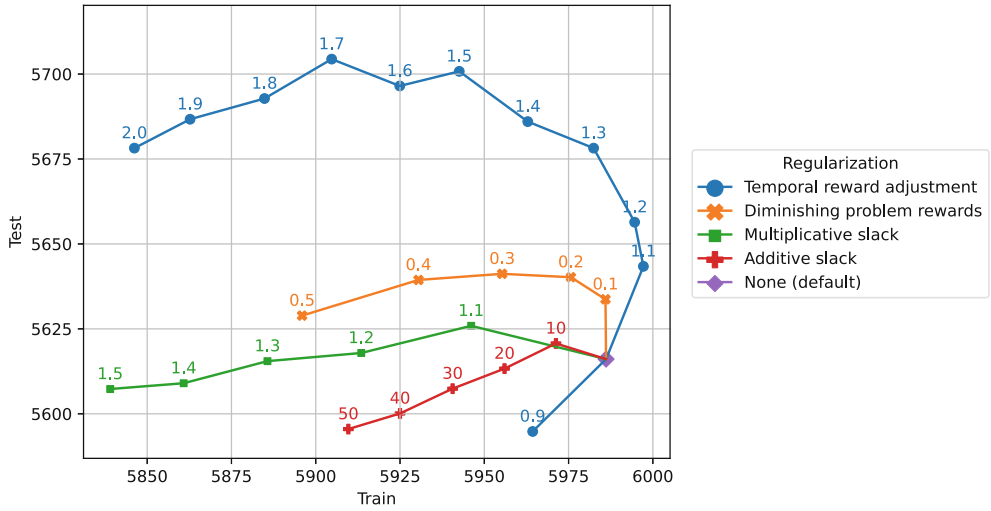


Fig. 4. Train and test performance of various regularizations of the greedy schedule construction algorithm for the budget 64 000. Performance is the mean number of problems solved out of 7866 across 50 splits. The label of each point denotes the value of the respective regularization parameter.

eter value from the evaluation grid. Table 2 shows that the best configurations of all the regularizations except multiplicative slack vary across budgets.[14]

7 Related Work

Outside the realm of theorem proving, strategy discovery belongs to the topic of *algorithm configuration* [22], where the task is to look for a strong configuration of a parameterized algorithm automatically. Prominent general-purpose algorithm configuration procedures include ParamILS [8] and SMAC [17].

To gather a portfolio of complementary configurations, Hydra [36] searches for them in rounds, trying to maximize the marginal contribution against all the configurations identified previously. Cedalion [25] is interesting in that it maximizes such contribution *per unit time*, similarly to our heuristic for greedy schedule construction. Both have in common that they, a priori, consider all the input problems in their criterion. BliStr and related approaches [7,9,11,12,33], on the other hand, combine strategy improvement with problem clustering to breed strategies that are "local experts" on similar problems. Spider [34] is even more radical in this direction and optimizes each strategy on a single problem.[15]

[14] See a more detailed comparison in Appendix A.

[15] Although the preference for default option values as a secondary criterion, at the same time, helps to push for good general performance (see Sect. 3.4).

Table 1. Comparison of regularizations of the greedy schedule construction algorithm for the budget 64 000 Mi. Performance is the mean number of problems solved out of 7866 across 50 splits. Time to fit is the mean time to construct a schedule in seconds.

Regularization	Performance		Time to fit [s]
	Test	Train	
$\alpha = 1.7$	5704	5905	4
$\beta = 0.3$	5641	5955	67
$w = 1.1$	5626	5946	19
$b = 10$	5621	5971	20
None (default)	5616	5986	20

Table 2. Best observed values of regularization parameters for various budgets

Budget [Mi]	Best parameter value			
	b	w	α	β
16000	0	1.1	1.3	0.2
64000	10	1.1	1.7	0.3
256000	20	1.1	1.4	0.2

Once a portfolio of strategies is known, it may be used in one of several ways to solve a new input problem: execute all strategies in parallel [36], select a single strategy [9], select one of pre-computed schedules [7], construct a custom strategy schedule [19], schedule strategies dynamically [16], or use a pre-computed static schedule [12,24]. The latter is the approach we explored in this work.

A popular approach to construct a static schedule (besides solving SCP optimally [7,24]) is to greedily stack uniformly-timed slices [12].[16] Regularization in this context is discussed by Jakubuv et al. [10]. Finally, a different greedy approach to schedule construction was already proposed in *p-SETHEO* [35].

8 Conclusion

In this work, we conducted an independent evaluation of Spider-style [34] strategy discovery and schedule creation. Focusing on the FOF fragment of the TPTP library, we collected over a thousand Vampire proving strategies, each a priori optimized to perform well on a single problem. Using these strategies, it is easy to construct a single monolithic schedule which covers most of the problems

[16] However, uniformly-timed slices only get close to the performance of our greedy schedule at a small region depending on the slice time limit used.

known to be solvable within the budget used by the CASC competition. This suggests that for CASC not to be mainly a competition in memorization, using a substantial set of previously unseen problems each year is essential.

To construct strong schedules using the discovered strategies, we proposed a greedy schedule construction procedure, which can compete with optimal approaches. For a time budget of approximately 2 min, the greedy algorithm takes less than a minute to produce a schedule that solves more than 99.0% as many problems as an optimal schedule, which takes more than 16 h to generate. For shorter time budgets, optimal schedule construction is no longer feasible, while greedy construction still produces relatively strong schedules.

This surprising strength of the greedy scheduler can be further reinforced by various regularization mechanisms, which constitute the main contribution of this work. An appropriately chosen regularization allows us to outperform the optimal schedule on unseen problems. Finally, the runtime speed and simplicity of the greedy schedule construction algorithm and the regularization techniques make them attractive for reuse and further experimentation.

Acknowledgment. This work was supported by the Czech Science Foundation project no. 20-06390Y (JUNIOR grant), the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the project RICAIP no. 857306 under the EU-H2020 programme, the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37, and the Czech Science Foundation project no. 24-12759S.

A Experimental Results on Various Budgets

In addition to the budget of 64 000 Mi (approx 32 s), which we discussed in Sect. 6.2, we evaluated the schedule construction algorithms on the budgets of 16 000 Mi (approx. 8 s) and 256 000 Mi (approx. 2 min). Figure 5 shows the results of these evaluations. It shows namely that temporal reward adjustment is the most powerful of the regularizations under consideration for all of these budgets, and that the optimal values of most of the regularization parameters vary across budgets.

To demonstrate the effect of the slice extension trick described in Sect. 5, we also include two weaker versions of the base greedy algorithm: one without slice extension and one with the slice extension restricted to the most recent slice ("conservative slice extension"). Both of these modifications allow including any single strategy in the schedule more than once, which is implemented in a straightforward fashion.

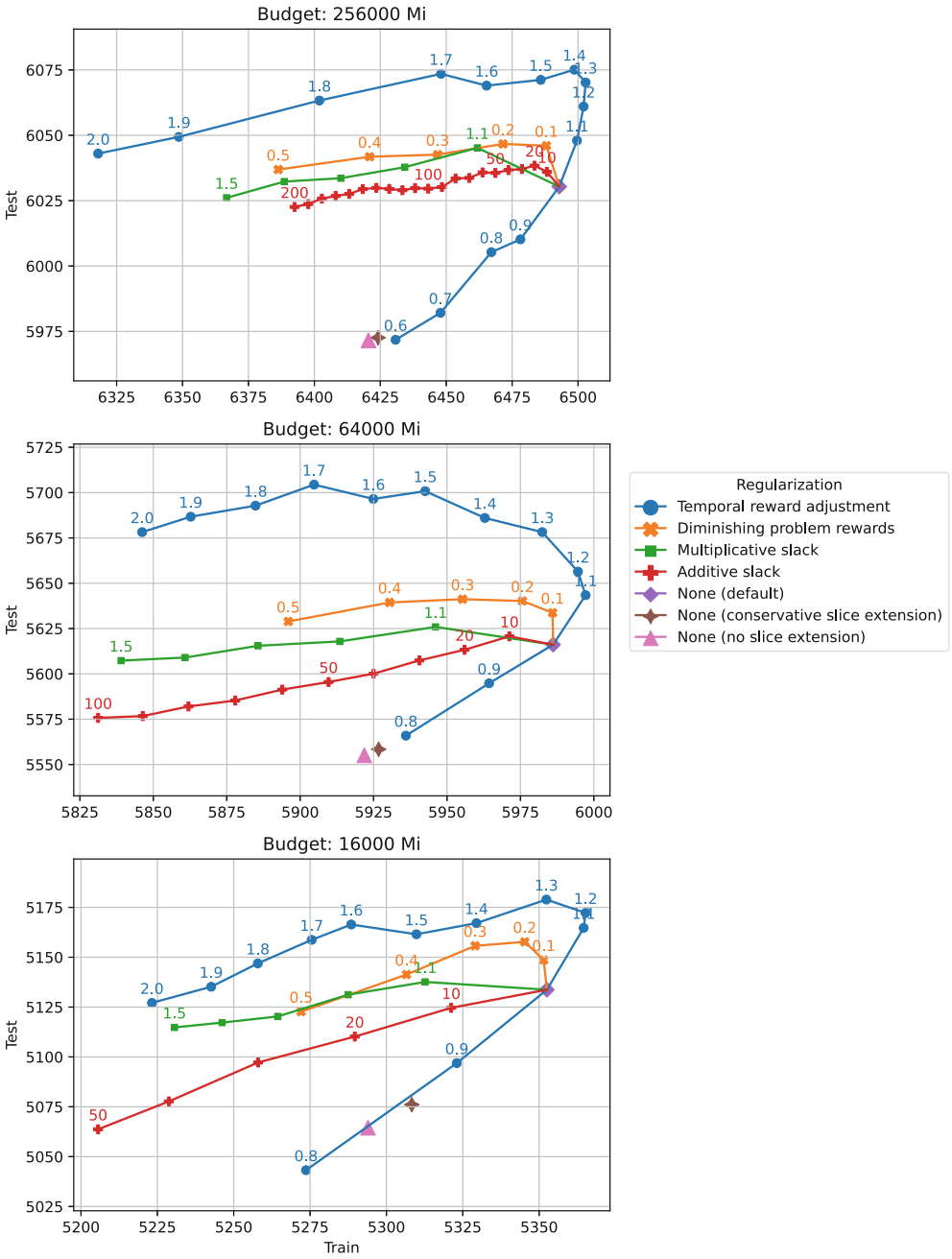


Fig. 5. Train and test performance of various regularizations of the greedy schedule construction algorithm for the budgets 256 000 Mi (*top*), 64 000 Mi (*middle*), and 16 000 Mi (*bottom*). Performance is mean number of problems solved out of 7866 across 50 splits. The label of each point denotes the value of the respective regularization parameter.

References

1. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction (2024). https://doi.org/10.48550/arXiv.2403.12869
2. Bártek, F., Suda, M.: Vampire strategy performance measurements (2024). https://doi.org/10.5281/zenodo.10814478
3. Chvátal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979). https://doi.org/10.1287/moor.4.3.233
4. Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.): Global Conference on Artificial Intelligence, GCAI 2015, Tbilis, 16–19 October 2015, EPiC Series in Computing, vol. 36. EasyChair (2015). https://easychair.org/publications/volume/GCAI_2015
5. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual (2023). https://www.gurobi.com
6. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning, 2nd edn. SSS, Springer, New York (2009). https://doi.org/10.1007/978-0-387-84858-7
7. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: Kamareddine, F., Coen, C.S. (eds.) CICM 2021. LNCS, vol. 12833, pp. 107–123. Springer, Cham (2021).https://doi.org/10.1007/978-3-030-81097-9_8
8. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**, 267–306 (2009). https://doi.org/10.1613/JAIR.2861
9. Hula, J., Jakubuv, J., Janota, M., Kubej, L.: Targeted configuration of an SMT solver. In: Buzzard, K., Kutsia, T. (eds.) CICM 2022. LNCS, vol. 13467, pp. 256–271. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16681-5_18
10. Jakubuv, J., et al.: MizAR 60 for Mizar 50. In: Naumowicz, A., Thiemann, R. (eds.) 14th International Conference on Interactive Theorem Proving, ITP 2023, 31 July to 4 August 2023, Białystok. LIPIcs, vol. 268, pp. 19:1–19:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ITP.2023.19
11. Jakubuv, J., Suda, M., Urban, J.: Automated invention of strategies and term orderings for Vampire. In: Benzmüller, C., Lisetti, C.L., Theobald, M. (eds.) GCAI 2017, 3rd Global Conference on Artificial Intelligence, Miami, 18–22 October 2017. EPiC Series in Computing, vol. 50, pp. 121–133. EasyChair (2017). https://doi.org/10.29007/XGHJ
12. Jakubuv, J., Urban, J.: BliStrTune: hierarchical invention of theorem proving strategies. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, 16–17 January 2017, pp. 43–52. ACM (2017).https://doi.org/10.1145/3018610.3018619
13. Khuller, S., Moss, A., Naor, J.: The budgeted maximum coverage problem. Inf. Process. Lett. **70**(1), 39–45 (1999). https://doi.org/10.1016/S0020-0190(99)00031-9
14. Korovin, K.: iProver—an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_24
15. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1

16. Kühlwein, D., Urban, J.: MaLeS: a framework for automatic tuning of automated theorem provers. J. Autom. Reason. **55**(2), 91–116 (2015). https://doi.org/10.1007/s10817-015-9329-1
17. Lindauer, M., et al.: SMAC3: a versatile bayesian optimization package for hyperparameter optimization. J. Mach. Learn. Res. **23**, 54:1–54:9 (2022). http://jmlr.org/papers/v23/21-0888.html
18. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993). https://doi.org/10.1016/0020-0190(93)90029-9
19. Mangla, C., Holden, S.B., Paulson, L.C.: Bayesian ranking for strategy scheduling in automated theorem provers. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 559–577. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_33
20. Reger, G., Suda, M., Voronkov, A.: New techniques in clausal form generation. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI 2016. 2nd Global Conference on Artificial Intelligence, 19 September–2 October 2016, Berlin. EPiC Series in Computing, vol. 41, pp. 11–23. EasyChair (2016). https://doi.org/10.29007/DZFZ
21. Schäfer, S., Schulz, S.: Breeding theorem proving heuristics with genetic algorithms. In: Gottlob et al. [4], pp. 263–274. https://doi.org/10.29007/gms9
22. Schede, E., et al.: A survey of methods for automated algorithm configuration. J. Artif. Intell. Res. **75**, 425–487 (2022). https://doi.org/10.1613/jair.1.13676
23. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE 27. LNCS, vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
24. Schurr, H.: Optimal strategy schedules for everyone. In: Konev, B., Schon, C., Steen, A. (eds.) Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, 11–12 August 2022. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022). https://ceur-ws.org/Vol-3201/paper8.pdf
25. Seipp, J., Sievers, S., Helmert, M., Hutter, F.: Automatic configuration of sequential planning portfolios. In: Bonet, B., Koenig, S. (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, 25–30 January 2015, Austin, pp. 3364–3370. AAAI Press (2015). https://doi.org/10.1609/AAAI.V29I1.9640
26. Suda, M.: Vampire getting noisy: will random bits help conquer chaos? (system description). EasyChair Preprint no. 7719 (2022). https://easychair.org/publications/preprint/CSVF
27. Sutcliffe, G.: The CADE ATP system competition - CASC. AI Mag. **37**(2), 99–101 (2016). https://doi.org/10.1609/AIMAG.V37I2.2620
28. Sutcliffe, G.: The TPTP problem library and associated infrastructure: from CNF to TH0, TPTP v6.4.0. J. Automat. Reason. **59**(4), 483–502 (2017). https://doi.org/10.1007/s10817-017-9407-7
29. Suttner, C.B., Sutcliffe, G.: The CADE-14 ATP system competition. J. Autom. Reason. **21**(1), 99–134 (1998). https://doi.org/10.1023/A:1006006930186
30. Tammet, T.: Gandalf. J. Autom. Reason. **18**(2), 199–204 (1997). https://doi.org/10.1023/A:1005887414560
31. Tammet, T.: Gandalf c-1.1 (1998). https://www.tptp.org/CASC/15/SystemDescriptions.html#Gandalf. Accessed 25 Jan 2023
32. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) CADE-15. LNCS, vol. 1421, pp. 427–441. Springer, Cham (1998). https://doi.org/10.1007/BFb0054276

33. Urban, J.: BliStr: The blind strategymaker. In: Gottlob et al. [4], pp. 312–331. https://doi.org/10.29007/8n7m
34. Voronkov, A.: Spider: learning in the sea of options (2023). https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833. Unpublished. Paper accepted at Vampire23: The 7th Vampire Workshop
35. Wolf, A., Letz, R.: Strategy parallelism in automated theorem proving. In: Cook, D.J. (ed.) Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference, 18–20 May 1998, Sanibel Island, pp. 142–146. AAAI Press (1998). http://www.aaai.org/Library/FLAIRS/1998/flairs98-027.php
36. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hydra: automatically configuring algorithms for portfolio-based selection. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, 11–15 July 2010. AAAI Press (2010). http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1929

Chapter 6

Cautious Specialization of Strategy Schedules

In chapter 5, my co-authors and I focused on building a monolithic strategy schedule that performs well on the whole target set of problems. The performance of the prover can be further increased by dividing the target set of problems into classes and constructing a specialized schedule for each of the classes. Each class should be identified by problem features that are easy to compute so that a branching system of schedules can be used on an unseen problem efficiently.

As long as the classes are relatively homogeneous in performance of the strategies, the specialization is expected to improve performance on the training problems. However, aggressive specialization can lead to a decrease in performance on unseen problems.

In the extended abstract presented below [5], we expanded on the work introduced in chapter 5 by investigating the problem of specialization of strategy schedules. We performed two initial experiments:

1. First, we compared a hand-tweaked split of the problem space (dividing problems into three classes by the number of atoms) to a random split.
2. Second, we experimented with training a collection of boosting trees to effectively tailor a schedule for an arbitrary input problem.

While the first experiment showed that the choice of problem features to split on may have a large impact on generalization, the second experiment represents a viable solution to the schedule specialization task as a whole.

Cautious Specialization of Strategy Schedules (Extended Abstract)

Filip Bártek[1,2], Karel Chvalovský[1] and Martin Suda[1]

[1]Czech Technical University in Prague – Czech Institute of Informatics, Robotics and Cybernetics. Jugoslávských partyzánů 1580/3, 160 00 Prague 6 – Dejvice, Czech Republic

[2]Czech Technical University in Prague – Faculty of Electrical Engineering. Technická 2, 166 27 Prague 6 – Dejvice, Czech Republic

Abstract

Combining theorem proving strategies into schedules is a well-known recipe for improving prover performance, as a well-chosen set of complementary strategies in practice covers many more problems within a predetermined time budget than a single strategy could. A strategy schedule can be a monolithic sequence, but may also consist of multiple schedule branches, intended to be selected based on certain problem's features and thus to *specialize* in solving the corresponding problem classes. When allowing schedule branching, there is an intuitive trade-off between covering the known (training) problems quickly, by splitting them into many classes, and the *generalization* of the obtained branching schedule, i.e., its ability to solve previously unseen (test) problems.

In this paper, we attempt to shed more light on this trade-off. Starting off with a large set of proving strategies of the automatic theorem prover Vampire evaluated on the first-order part of the TPTP library, we report on two preliminary experiments with building branching schedules while keeping track of how well they generalize to test problems. We hope to attract more attention to the exciting topic of schedule construction by this work-in-progress report.

Keywords

strategy schedules, schedule specialization, regularization, Vampire

1. Introduction

Proof search in most automatic theorem provers (ATPs) relies on parameterized heuristics. Strong configurations (*strategies*) of an ATP are, typically, specialized to certain classes of input problems – no configuration performs well on all possible problems of interest. Selecting a strong strategy for a given input problem is, in general, hard. This is namely due to the chaotic behavior of the provers [1]: Using a fixed strategy, a small change in the input problem may lead to a substantially different proof search.

Typically, if a strategy solves a problem, the solution is reached in a relatively short time [2]. In contexts where sufficient time or parallelism is available, running a portfolio of diverse strategies is a pragmatic approach to increase the power of a prover. Such portfolio, or *schedule* in case a time limit is assigned to each of the strategies, may be either hand-crafted [3] or constructed automatically [4, 5].

Further gain in performance can be achieved by *specializing* the schedule: We partition the possible input problems into classes and construct a specialized schedule for each class. We prefer the classes to be homogeneous in terms of the problems' response to strategies, so that a schedule specialized for a class needs less time to cover the same percentage of problems. Increasing the granularity of the classes increases the specialization and the associated potential performance gain.

Decision trees [6] provide a suitable architecture to capture such gradual specialization. A set of features, such as the number of atoms, the presence of certain syntactic traits, or the membership in standard logical fragments, are computed for the input problem. A binary tree is then traversed from the root to a leaf. Each internal node is labeled with a criterion that partitions the domain of one of the features into two parts. When a node is traversed, the next node is selected depending on the corresponding feature of the problem. The schedule inhabiting the final leaf is then used to attempt to solve the problem.

PAAR'24: 9th Workshop on Practical Aspects of Automated Reasoning, July 2, 2024, Nancy, France

filip.bartek@cvut.cz (F. Bártek); karel.chvalovsky@cvut.cz (K. Chvalovský); martin.suda@cvut.cz (M. Suda)

0000-0002-1822-2651 (F. Bártek); 0000-0002-0541-3889 (K. Chvalovský); 0000-0003-0989-5800 (M. Suda)

The ATP Vampire [7] won the prestigious First-Order Form theorems (FOF) division of the CADE ATP System Competition (CASC) [8] every year from 2002 to 2023 [9]. Arguably, this continued success is partially thanks to the famous "CASC mode" of the prover and the corresponding use of strong strategy schedules. The schedules were constructed by Andrei Voronkov with the help of a system called Spider [10] and arranged in a shallow decision tree for specialization.[1]

When the specialization problem classes are chosen in a way that optimizes the performance on a training set of problems, we run the risk of *overfitting*, that is increasing performance on the training problems at the cost of decreasing performance on unseen problems. *Regularization* is the process of reducing the expressive power of a machine-learned model to prevent overfitting.

In our previous work [4], we studied the regularization of monolithic schedule construction, i.e., the case where the algorithm's output is a single sequence with no specialization to problem features. We discovered and evaluated more than a thousand Vampire strategies targeting the first-order fragment of the TPTP library [11], which are now available as an independent data set for experimentation [12].

In this work, we use the same data set to start shedding some light on regularization during schedule specialization. Namely, after brief preliminaries (Sect. 2), we first discuss (and measure) how a single branching criterion (i.e., a single problem feature) can affect the generalization of the simplest branching schedule with just one level of branching (Sect. 3). We complement this view by an experiment with gradient boosted trees (Sect. 4), which explores the opposite extreme case where every problem can — based on its features — be, in principle, assigned its unique schedule. Our paper also briefly overviews related work (Sect. 5) and offers several concluding remarks (Sect. 6).

We are far from any definite answer on how specialization in schedule construction should be best done. Yet we hope to spark more interest in this exciting topic by this work-in-progress report.

2. Preliminaries

2.1. Training and evaluation data

In our previous work [4], we generated a collection of 1096 strong and mutually complementary strategies for the ATP Vampire. We evaluated the performance of each of these strategies on a set of 7866 first-order logic (FOL) problems from TPTP [11] v8.2.0. The results of these evaluations constitute a dataset suitable for experimenting with schedule construction and specialization [12].

We denote the set of strategies as S ($|S| = 1096$), the set of problems as P ($|P| = 7866$), and the *evaluation matrix* as $E_p^s : S \times P \to \mathbb{N} \cup \{\infty\}$. The value E_p^s is the time at which strategy s solves problem p, or ∞ in case strategy s failed to solve problem p within the time limit.

The time is measured in whole CPU *megainstructions (Mi)*, where $1\,\mathrm{Mi} = 2^{20}$ instructions reported by the tool `perf` and $2000\,\mathrm{Mi}$ take approximately 1 second of wallclock time on our hardware.

2.2. Vampire's CASC-mode problem features

For our schedule specialization experiments, we use problem features already available in Vampire and used there by the CASC mode. This feature set consists mainly of various syntactic counters (the number of: goal clauses, axiom clauses, Horn clauses, equational clauses, etc.), most prominent of which is the number of atoms (roughly corresponding to problem size); problem class according to TPTP classification (NEQ, HEQ, …, EPR, UEQ); and a number of bitfields checking for properties such as "has $X = Y$", "has functional definitions", or "has inequality resolvable with deletion".

These features are not included in the mentioned dataset [12], but can be obtained easily by running a modified version of Vampire[2] in "profile mode" via `--mode profile <problem_name>`.

[1] E.g., https://github.com/vprover/vampire/blob/c7564c1d65020771079f29787ca2d5d7743f5d6a/CASC/Schedules.cpp#L5676.
[2] https://github.com/vprover/vampire/tree/ourProfile

2.3. Strategy schedules

A strategy *schedule* is a mapping $t_s : S \to \mathbb{N}$. The value t_s is the time limit assigned to strategy s. A schedule may be used to attack an arbitrary problem with Vampire: The strategies are run sequentially with the assigned time limits, each attempting to solve the problem, until a solution is found or every strategy has depleted its allocated time. In the rest of this work, instead of running the prover, we use the evaluation matrix to simulate the evaluation of schedules, restricting our scope to problems in P.

Similarly to strategies, schedules are typically evaluated in a time-constrained setting. Schedule t_s satisfies (time) *budget* $T \in \mathbb{N}$ if and only if $\sum_{s \in S} t_s \leq T$.

In our previous work [4], we proposed a greedy algorithm that constructs a strategy schedule from an evaluation matrix and a budget. The algorithm allows finding a relatively strong (with respect to the evaluation matrix) schedule satisfying the budget quickly.

A *strategy schedule recommender* produces a schedule for an arbitrary input problem. To evaluate a schedule recommender on a set of problems, we estimate, using the evaluation matrix, the proportion of problems solved by the respective schedules produced by the recommender.

2.4. Overfitting and regularization

Extreme schedule specialization is discouraged in CASC [13]:

> All techniques used must be general purpose, and expected to extend usefully to new unseen problems. [...] If machine learning procedures are used to tune a system, the learning must ensure that sufficient generalization is obtained so that no there is no specialization to individual problems.

In machine learning (ML), it is common for a trained system to perform better on the data it was trained on than on previously unseen data. The system is said to *overfit* to the training data. In most applications, we are ultimately interested in optimizing the performance on unseen data, so overfitting is undesirable. A system that does not overfit is said to *generalize* (to unseen data).

To evaluate generalization of our ML training procedures, we repeatedly randomly split the set of problems P into a training set ($80\,\%$) and a test set ($20\,\%$), train the system on the training set, and evaluate the trained system on the test set. The final score is the mean of the per-split test scores (in our case, schedule recommender success rates). This way, we estimate the performance on unseen data.

Moreover, before training we exclude strategies that have their "witness problem" in the test set. Roughly speaking, this prunes the set of strategies only to those that could have been discovered if a given test set was fixed beforehand for the strategy discovery phase. See [4, Section 6] for more details.

3. Single branching point

Creating a schedule can be seen as an act of "covering" the training problems with the help of the known solutions of the previously evaluated strategies. It is clear that if many strategies need to be involved with long running times in order to eventually cover every training problem, the running time of the whole schedule will need to be high.[3] Schedule specialization comes to rescue to reduce this running time, as it allows the covering process to focus, within each branch, on a different subset of the training problems. In a nutshell, on each branch, there is less covering work to do and during evaluation, only the "right" branch is always used.

However, from the perspective of generalization, i.e., the effort of ensuring that our specializing schedule performs well also on problems unseen in training, splitting the work into branches may be detrimental. Indeed, if many training problems end up in their respective subset for mostly a random

[3] Here we conceptually depart from the position where there is a fixed budget and we are trying to solve as many problems as possible within that budget. This is because we aim, in this section, to present an analysis which answers a question for all budgets at once. We can do it thanks to the anytime nature of the mentioned greedy algorithm we employ here [4].

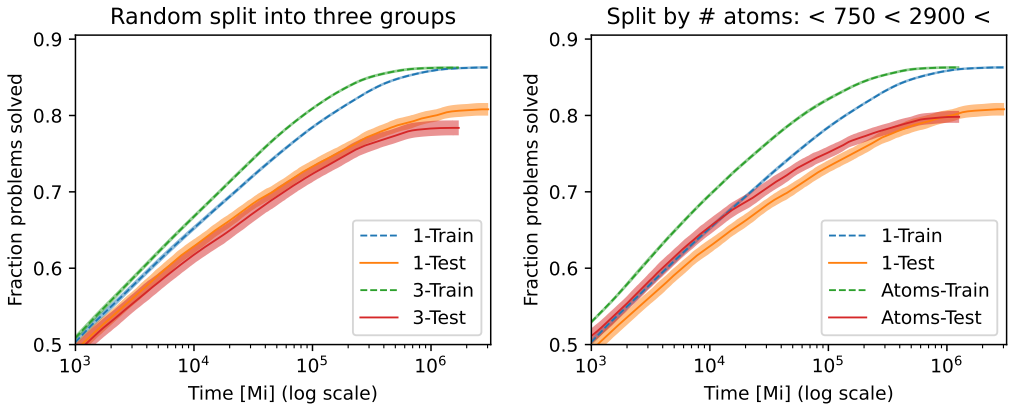


Figure 1: Monolithic ("1") vs single branching schedule performance. Problems split into three random subsets on the left ("3") and into small-medium-large subsets based on the number of atoms ("Atoms") on the right. Showing averages over 100 random $80 : 20$ train/test split runs. Shading marks the zone of one standard deviation around the averages (and got four times larger for the Test runs compared to the Train runs just from the effect of normalizing the averages from the unequal train/test set sizes to the reported fractions).

reason, the chances of a similar testing problem (that could be solved by the same strategy) being classified into the same subset and thus subjected to the same schedule branch are low.

In this experiment, we set out to demonstrate this phenomenon at its extreme. Our method of covering is the greedy algorithm presented in previous work [4] and we run it repetitively over random $80 : 20$ train/test splits of our problem set P. We compare the average performance of a monolithic (non-branching) schedule, a schedule branching into three groups based on the number of problem's atoms, and a schedule branching into three random equally-sized groups. For the branching on atoms, we split the problems into *small* (number of atoms < 750), *large* (number of atoms > 2900) and *medium* (the rest). The cutoff points were chosen *not* to split the problems into equally-sized subsets, but by aiming for a split for which each subset can be fully covered within approximately the same budget.

The results are presented in cactus plot form in Fig. 1. The results of the monolithic schedule building (1-Train/1-Test) are the same in both the left and the right plot (and roughly correspond to Fig. 7 in Appendix E of our previous work [4]). The monolithic schedule requires approximately 2 million Mi to cover all the training problems and the test performance converges to around $93\,\%$ of the training one. The train performance of both the random 3-split (3-Train) and the atom split (Atom-Train) are at all times higher than that of the monolithic schedule until they (necessarily) converge to the same value and stabilize. (The atom split tends to finish its covering job a bit earlier than the random split.) However, the pictures for the test performance of the branching schedules differ. While splitting on atoms improves over monolithic even in test performance (up until around the 1 million Mi mark), and for the lower budgets even improves over the monolithic's train performance, the test performance of the random 3-split (3-Test) is at all times worse.

It is perhaps not surprising that random features are useless from the perspective of branching schedule generalization, however, we find the "backwards version" of this perspective to be quite revealing, namely the realization that even reasonable-sounding problem features may differ in how useful (useless, or even detrimental) they could be in this regard. Schedule construction is sometimes organized by *first* fixing the problem groups based on features and only *then* looking for strategies in each group separately (one can save computational resources by not having to evaluate each strategy on every training problem; evaluating just within the relevant group would be enough). However, if such initial split is only based on educated guessing of what problems reasonably belong together, opportunities for better (or even the best possible) generalization can be missed.

Ideally, we would like to come up with generalization-conducive splits of the problems beforehand,

by just studying our evaluation matrix and asking a question such as "What could be a nice, easy-to-delineate subset of problems for which certain strategies work well together and complement each other, while elsewhere they are mostly useless?" We remark that such splits might be defined by other properties than those expressible by our current fixed feature set. Discovering such new properties that would still be easy to compute (and thus easy to understand) would be an interesting instance of the explainable AI endeavour. At this moment, however, we do not have a good way of doing this and leave this research direction for future work.

4. Gradient boosting

Unlike in the previous section, where a hand-crafted split based on the number of atoms occurring in the problem is discussed, we want to produce finer splits of the problems based on Vampire's CASC-mode problem features here. In other words, we want to learn more complicated decision trees, or as in our case ensembles of small decision trees. Hence we want to train a model that for a given problem p produces a schedule $t_1, \ldots, t_{|S|}$, where $\sum_{s \in S} t_s \leq T$, such that there is a strategy $s \in S$ that solves $p \in P$ in time at most t_s.

Although we want to obtain schedules, we may characterize them by probability distributions. A discrete probability mass function $\mathbf{p}\colon S \to [0, 1]$ defines a schedule by setting $t_s = T \cdot \mathbf{p}_s$. Hence we can use standard approaches for multi-class classification, where usually we get a probability distribution over possible classes and select the class as the one with the highest estimated probability. Here, however, we take the estimated probability distribution and compute a schedule based on it.

This of course immediately leads to the question of what is the true probability distribution we want to estimate. One possible approach is to assign, for example, probability 1 to the best strategy for the problem. However, we likely need a more fine-grained approach as we are interested in generalization. First, we are happy to split the budget among more strategies, as long as at least one of them solves the problem. Second, we do not require that the problem be solved by the best available strategy.

Hence instead of having, for each (training) problem, a fixed true distribution that we want to learn, we may produce a "true" distribution iteratively during the training based on our previous estimates and change it as necessary. For example, we have a training problem p and an estimated probability $\mathbf{p}_1, \ldots, \mathbf{p}_{|S|}$ for it. We may want to produce a "true" distribution for p by assigning $\mathbf{p}_i = 0$ if the strategy i does not solve the problem in the given budget T, and hence increase $\mathbf{p}_j$ if the strategy j solves the problem. In other words, we want to split the budget only among strategies that can possibly solve the problem. Moreover, we can boost the strategy that is closest to solving the problem; the strategy requiring the smallest increase in the estimated budget to solve the problem. And there are many other ways how to get a new "true" distribution. It is worth noticing that we may obtain this "true" distribution in different ways: one way for those training examples that our model predicts correctly (estimated schedule solves the problem) and another for those that are predicted incorrectly.

4.1. Model

Gradient boosting for decision trees, see, e.g., [14], fits nicely into this picture. Loosely speaking, it combines weak models that iteratively improve their predictions by correcting the previous mistakes. In particular, we learn decision trees that split our problems into different subsets based on the static Vampire's features. Or, more precisely, in each iteration, we learn $|S|$ regression trees (multi-class problem); one per strategy. When we want to estimate the probability distribution after i iterations, we combine the output of all i regression trees learned for each strategy, and get the probability distribution by the softmax function. In the next iteration, we fix the "true" distribution based on the newly estimated distribution and iterate this process until we get a satisfactory model or reach the limit on the number of iterations.

Table 1
We compare the success rate of the split into small-medium-large subsets based on the number of atoms ("Atoms", see Sect. 3) and the schedule produced by gradient boosting.

Budget	Atoms		Gradient boosting	
	Train	Test	Train	Test
1000	0.5297	0.5113	0.6196	0.5660
10000	0.6965	0.6547	0.7650	0.6998
100000	0.8219	0.7519	0.8358	0.7807

4.2. Initial experiments

We use LightGBM [15], a framework for gradient boosting decision trees, with custom objective functions. We get probability estimates from our models by the softmax function and use the cross-entropy between predicted probabilities and "true" probabilities as the loss function.

We report some preliminary results, as we have not performed much of hyper-parameter tuning.[4] We use the learning rate 0.05, the maximal number of bins that feature values are bucketed in is 8, the number of boosting iterations is 500, and, most importantly, we restrict the number of leaves to 2; hence we use decision stumps as weak learners.

Our custom objective functions create "true" probabilities $\mathbf{p}'_1, \ldots, \mathbf{p}'_{|S|}$ from the estimated probabilities $\mathbf{p}_1, \ldots, \mathbf{p}_{|S|}$ for a problem p. We have tried various combinations of the following procedures:

(a) $\mathbf{p}'_i = 0$ if the strategy i does not solve p in the time budget T, otherwise $\mathbf{p}'_j = \mathbf{p}_j$,

(b) $\mathbf{p}'_i = 1$ if the strategy i is the best available strategy[5] and $\mathbf{p}'_j = 0$ for $i \neq j$,

(c) boost[6] the best available strategy,

and then L1-normalize $\mathbf{p}'_1, \ldots, \mathbf{p}'_{|S|}$. It is even possible to combine different ways how to create "true" probabilities based on whether the predicted schedule solves the training problem or not. Surprisingly, (a) works quite well even though it can get stuck in a local minima. Hence, for simplicity, all the subsequent results use this objective function. It takes roughly $10\,\mathrm{s}$, using one core, to train one iteration of the model with our non-optimized custom objective function written in Python.

We train our models using 10 random 80:20 train/test splits over the dataset described in Section 2.1. In Table 1, we report the success rate. The best iteration is selected based on the number of solved problems on the training set. For shorter budgets, the gradient boosting model significantly outperforms the simple split into small-medium-large subsets based on the number of atoms in the problem. As expected, the gap is narrowing as the budget is increasing.

Interestingly, when we inspect the produced schedules, the majority of problems (both on the train and test sets) are solved by multiple strategies in the schedule. Moreover, it is unlikely that a problem is solved by the overall best available strategy for the problem, supporting our idea behind the careful creation of "true" probabilities.

As the "true" probability that we aim to estimate evolves, it is possible that the majority of trees learned do not offset incorrect estimates from previous iterations but rather compensate for the changing "true" probability. Note that we often add the same tree (with different leaf values) multiple times.

These initial experiments show that it is possible to automatically train a model that both leverages the available feature space and keeps a reasonable level of generalization.

[4] The performance of LightGBM is sensitive to choices of hyper-parameters. Moreover, our custom objective functions make these choices even more significant.

[5] There are many possible definitions here. For example, we have experimented with selecting the strategy that has the maximal difference between the time assigned to the strategy and the time necessary to solve the problem by the strategy.

[6] We tried two variants. First, in combination with (a), we split "saved time" not among all strategies that solve the problem, but we only boost the best available strategy. Second, we increase the allocation to the best available strategy just enough for it to solve the problem.

5. Related work

Strategy schedule specialization is closely related to *strategy selection* (also referred to as "heuristic selection"), a special case of algorithm selection [16, 17]. Automatic mode in the E prover [18] uses hand-crafted problem classes to select a strategy for the input problem [19]. Strategy selection for E has also been approached by machine learning [20]. Given an input problem, MaLeS [19] predicts solving time for all strategies in a portfolio and selects the strategy that minimizes the predicted time. Similarly, Grackle [21] selects a strategy using a trained LightGBM model that predicts a ranking of strategies in a pre-computed portfolio.

Strategy schedules have been constructed manually in Gandalf [3] and automatically in E-SETHEO [22], Spider for Vampire [10], CPhydra [23], BliStrTune [24], HOS-ML [5], and in our previous work [4]. Given an input problem, HOS-ML [5] selects a strategy schedule automatically. The problem's static features are used to identify a problem class, for which a schedule was optimized during the training. CPhydra [23] identifies 10 training problems that are the most similar to the input problem and optimizes a schedule for these problems using a pre-computed evaluation matrix.

6. Conclusion

Working with a dataset of 1096 Vampire strategies, which were evaluated on the first-order part of TPTP, and a selection of syntactic problem features, which Vampire computes efficiently for its input problem at startup, we presented two experiments with strategy schedule construction and schedule specialization. In the first, we noticed that the selection of the splitting features not only influences how fast a specialized schedule can cover the training problems, but, more interestingly, how well it can generalize to previously unseen problems. In the second, we demonstrated that a good performance can be achieved by using gradient boosted decision trees. The second approach is interesting in that it departs from the concept of problem covering and instead develops a schedule in the form of a probability distribution over the available strategies. This requires the use of a custom target distribution which develops during the learning process.

There are many avenues for future research. Our next target for experimentation is the grow-and-prune technique for decision tree construction and regularization [6] and its adaptation to cautious schedule specialization. However, we would also be more than happy for others to join us in experimenting with our strategy data [12] to see which method could eventually lead to the most useful approach.

Acknowledgments

The work on this paper was supported by the Czech Science Foundation grant 24-12759S and the project RICAIP no. 857306 under the EU-H2020 programme.

References

[1] M. Suda, Vampire getting noisy: Will random bits help conquer chaos? (system description), in: J. Blanchette, L. Kovács, D. Pattinson (Eds.), Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings, volume 13385 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 659–667. URL: https://doi.org/10.1007/978-3-031-10769-6_38. doi:10.1007/978-3-031-10769-6_38.

[2] G. Reger, Boldly going where no prover has gone before, in: M. Suda, S. Winkler (Eds.), Proceedings of the Second International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, ARCADE@CADE 2019, Natal, Brazil, August 26, 2019, volume 311 of *EPTCS*, 2019, pp. 37–41. URL: https://doi.org/10.4204/EPTCS.311.6. doi:10.4204/EPTCS.311.6.

[3] T. Tammet, Towards efficient subsumption, in: C. Kirchner, H. Kirchner (Eds.), Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings, volume 1421 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 427–441. URL: https://doi.org/10.1007/BFb0054276. doi:10.1007/BFb0054276.

[4] F. Bártek, K. Chvalovský, M. Suda, Regularization in Spider-style strategy discovery and schedule construction, 2024. URL: https://arxiv.org/abs/2403.12869. doi:10.48550/arXiv.2403.12869. arXiv:2403.12869.

[5] E. K. Holden, K. Korovin, Heterogeneous heuristic optimisation and scheduling for first-order theorem proving, in: F. Kamareddine, C. S. Coen (Eds.), Intelligent Computer Mathematics - 14th International Conference, CICM 2021, Timisoara, Romania, July 26-31, 2021, Proceedings, volume 12833 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 107–123. URL: https://doi.org/10.1007/978-3-030-81097-9_8. doi:10.1007/978-3-030-81097-9_8.

[6] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification and Regression Trees, Wadsworth, 1984.

[7] L. Kovács, A. Voronkov, First-order theorem proving and Vampire, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 1–35. URL: https://doi.org/10.1007/978-3-642-39799-8_1. doi:10.1007/978-3-642-39799-8_1.

[8] G. Sutcliffe, The CADE ATP System Competition - CASC, AI Magazine 37 (2016) 99–101.

[9] G. Sutcliffe, The CADE ATP system competition, 2024. URL: https://tptp.org/CASC/, [Online; accessed 26-April-2024].

[10] A. Voronkov, Spider: Learning in the sea of options, 2023. URL: https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833, unpublished. Paper accepted at Vampire23: The 7th Vampire Workshop.

[11] G. Sutcliffe, The TPTP problem library and associated infrastructure, Journal of Automated Reasoning 59 (2017). doi:10.1007/s10817-017-9407-7.

[12] F. Bártek, M. Suda, Vampire strategy performance measurements, 2024. URL: https://doi.org/10.5281/zenodo.10814478. doi:10.5281/zenodo.10814478.

[13] G. Sutcliffe, CASC design and organization, 2024. URL: https://tptp.org/CASC/J12/Design.html, [Online; accessed 15-April-2024].

[14] J. H. Friedman, Greedy function approximation: A gradient boosting machine., The Annals of Statistics 29 (2001) 1189–1232. URL: https://doi.org/10.1214/aos/1013203451. doi:10.1214/aos/1013203451.

[15] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, LightGBM: A highly efficient gradient boosting decision tree, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 30, Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[16] J. R. Rice, The algorithm selection problem, Adv. Comput. 15 (1976) 65–118. URL: https://doi.org/10.1016/S0065-2458(08)60520-3. doi:10.1016/S0065-2458(08)60520-3.

[17] P. Kerschke, H. H. Hoos, F. Neumann, H. Trautmann, Automated algorithm selection: Survey and perspectives, Evol. Comput. 27 (2019) 3–45. URL: https://doi.org/10.1162/evco_a_00242. doi:10.1162/EVCO_A_00242.

[18] S. Schulz, S. Cruanes, P. Vukmirovic, Faster, higher, stronger: E 2.3, in: P. Fontaine (Ed.), Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings, volume 11716 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 495–507. URL: https://doi.org/10.1007/978-3-030-29436-6_29. doi:10.1007/978-3-030-29436-6_29.

[19] D. Kühlwein, J. Urban, MaLeS: A framework for automatic tuning of automated theorem provers, J. Autom. Reason. 55 (2015) 91–116. URL: https://doi.org/10.1007/s10817-015-9329-1. doi:10.1007/s10817-015-9329-1.

[20] J. P. Bridge, S. B. Holden, L. C. Paulson, Machine learning for first-order theorem proving - learning to select a good heuristic, J. Autom. Reason. 53 (2014) 141–172. URL: https://doi.org/10.1007/s10817-014-9301-5. doi:10.1007/S10817-014-9301-5.

[21] J. Hůla, J. Jakubův, M. Janota, L. Kubej, Targeted configuration of an SMT solver, in: K. Buzzard, T. Kutsia (Eds.), Intelligent Computer Mathematics - 15th International Conference, CICM 2022, Tbilisi, Georgia, September 19-23, 2022, Proceedings, volume 13467 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 256–271. URL: https://doi.org/10.1007/978-3-031-16681-5_18. doi:10.1007/978-3-031-16681-5_18.

[22] S. Schulz, E-SETHEO, http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/e-setheo.html, 2024. [Online; accessed 26-April-2024].

[23] D. Bridge, E. O'Mahony, B. O'Sullivan, Case-based reasoning for autonomous constraint solving, in: Y. Hamadi, É. Monfroy, F. Saubion (Eds.), Autonomous Search, Springer, 2012, pp. 73–95. URL: https://doi.org/10.1007/978-3-642-21434-9_4. doi:10.1007/978-3-642-21434-9_4.

[24] J. Jakubův, J. Urban, BliStrTune: hierarchical invention of theorem proving strategies, in: Y. Bertot, V. Vafeiadis (Eds.), Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017, ACM, 2017, pp. 43–52. URL: https://doi.org/10.1145/3018610.3018619. doi:10.1145/3018610.3018619.

Chapter 7
Conclusion

7.1 Summary

ML provides an attractive direction for improving ATP because provers are complex, heuristic-driven systems whose behavior is difficult to optimize by hand and because, typically, training data can be generated in large amounts, limited only by computational resources. In this thesis, I presented several novel techniques for applying ML to improve saturation-based ATP for FOL. By using the prover Vampire and the TPTP problem library in my experiments, I demonstrated the degree to which these techniques can be used to improve the state of the art in ATP.

The first broad approach I explored is the configuration of a chosen heuristic on a per-problem basis: Given an input problem, a trained recommender configures a heuristic in Vampire to maximize the chance that the problem is solved quickly. I configured two heuristics this way: simplification ordering on terms (via symbol precedence; see chapters 2 and 3) and weighted symbol-counting clause selection (see chapter 4).

Among the ML models I experimented with, the GNNs were prominently successful in configuring the heuristics, yielding an improvement of 4 % and 6.6 % over a baseline, respectively, in the expected percentage of problems solved in a fixed time limit. Notably, using a GNN allows training a signature-agnostic recommender—one that learns from the structure of the training problems in a manner invariant to symbol renaming.

Each of the GNN-based recommenders is trained on a proxy task of ranking objects (precedences or clauses, respectively) with training data in the form of ordered pairs of objects. Notably, the designs of these recommenders demonstrate how methods proposed in the area of learning to rank can be applied in ATP.

The second approach (see chapters 5 and 6) configures a large number of heuristics jointly. Instead of recommending a single prover strategy (configuration) on a per-problem basis, I constructed a static schedule of complementary strategies to optimize Vampire's performance on a whole distribution of problems. The strategies and performance measurements that served as the basis for the schedule construction have been published.

The main innovation this branch of my research introduced is the rigorous treatment of generalization of the schedules to unseen problems. To this effect, I proposed a regularized greedy algorithm for constructing the schedules.

7.2 Future Work

Various models and associated training algorithms have been proposed as approaches to ML. In my research, I used neural networks trained by stochastic gradient descent, Elastic-Net (regularized linear regression), and gradient-boosted decision trees. The neural networks trained on pairwise ranking proxy tasks demonstrated the greatest success in optimization of the prover heuristics (see chapters 3 and 4). Different attractive trade-offs between theorem proving performance, training efficiency, ease of deployment, and explainability might be achieved by using different models or proxy tasks.

In each of the problem domains that admit encoding in FOL, such as mathematics (formalized, for example, in the Mizar Mathematical Library [49, 50] or Archive of Formal Proofs [51, 52]) or software verification, a common set of symbols—a shared signature—can be identified. Configuring a term ordering or the clause selection procedure for such a domain could be achieved by combining relevant parts of my research with approaches suitable for operating with a fixed signature, such as convex optimization.

While saturation-based proving is well established and researched in the context of FOL, the paradigm, with appropriate modifications, has also recently been successfully applied to higher-order logic (HOL) [53–55]. All of the techniques for optimization of heuristics I introduced in this thesis could potentially be adapted and applied in saturation-based proving in HOL.

Highly parameterized solvers are common in various areas of complex problem solving besides FOL and HOL. Alternatively, two or more solvers may be available for a problem area in a setting known as algorithm selection [29]. My work on regularized strategy schedule construction (see chapters 5 and 6) could be applied to an arbitrary parameterized solver or a collection of solvers regardless of the problem area. It would be particularly exciting to explore whether and how the specifics of the problem area influence the effect of the regularization.

Declaration on Generative AI

During the preparation of this work, the author used Grammarly and Writefull to improve the readability and language of the work (Grammar and spelling check, Improve writing style, Paraphrase and reword), and DeepL Translator to translate the abstract from English to Czech (Text Translation).[1] After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

[1]The contributions are categorized in accordance with the CEUR-WS GenAI Usage Taxonomy [56].

Bibliography

[1] Filip Bártek and Martin Suda. Learning precedences from simple symbol features. In Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tourret, editors, *Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual)*, volume 2752 of *CEUR Workshop Proceedings*, pages 21–33. CEUR-WS.org, 2020. URL `https://ceur-ws.org/Vol-2752/paper2.pdf`.

[2] Filip Bártek and Martin Suda. Neural precedence recommender. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 525–542. Springer, 2021. ISBN 978-3-030-79875-8. doi:10.1007/978-3-030-79876-5_30.

[3] Filip Bártek and Martin Suda. How much should this symbol weigh? A GNN-advised clause selection. In Ruzica Piskac and Andrei Voronkov, editors, *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023*, volume 94 of *EPiC Series in Computing*, pages 96–111. EasyChair, 2023. doi:10.29007/5F4R.

[4] Filip Bártek, Karel Chvalovský, and Martin Suda. Regularization in Spider-style strategy discovery and schedule construction. In Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, volume 14739 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2024. ISBN 978-3-031-63497-0. doi:10.1007/978-3-031-63498-7_12.

[5] Filip Bártek, Karel Chvalovský, and Martin Suda. Cautious specialization of strategy schedules (extended abstract). In Christopher W. Brown, Daniela Kaufmann, Cláudia Nalon, Alexander Steen, and Martin Suda,

editors, *Joint Proceedings of the 9th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 9th Satisfiability Checking and Symbolic Computation Workshop (SC-Square), 2024 co-located with the 12th International Joint Conference on Automated Reasoning (IJCAR 2024), Nancy, France, July 2, 2024*, volume 3717 of *CEUR Workshop Proceedings*, pages 28–36. CEUR-WS.org, 2024. URL `https://ceur-ws.org/Vol-3717/short2.pdf`.

[6] Filip Bártek and Martin Suda. Vampire strategy performance measurements, March 2024. doi:10.5281/zenodo.10814478.

[7] Filip Bártek, Karel Chvalovský, and Martin Suda. Regularization in Spider-style strategy discovery and schedule construction. *CoRR*, abs/2403.12869, 2024. doi:10.48550/ARXIV.2403.12869.

[8] José Ferreirós. The road to modern logic - an interpretation. *Bull. Symb. Log.*, 7(4):441–484, 2001. doi:10.2307/2687794.

[9] Vijay Victor D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 27(7):1165–1178, 2008. doi:10.1109/TCAD.2008.923410.

[10] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. ISBN 978-3-319-49811-9. doi:10.1007/978-3-319-49812-6.

[11] Adam Pease and Geoff Sutcliffe. First order reasoning on a large ontology. In Geoff Sutcliffe, Josef Urban, and Stephan Schulz, editors, *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories, Bremen, Germany, 17th July 2007*, volume 257 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007. URL `https://ceur-ws.org/Vol-257/07_Pease.pdf`.

[12] Henry Prakken, Adam Z. Wyner, Trevor J. M. Bench-Capon, and Katie Atkinson. A formalization of argumentation schemes for legal case-based reasoning in ASPIC+. *J. Log. Comput.*, 25(5):1141–1166, 2015. doi:10.1093/LOGCOM/EXT010.

[13] Tomer Libal and Tereza Novotná. Towards transparent legal formalization. In Davide Calvaresi, Amro Najjar, Michael Winikoff, and Kary Främling, editors, *Explainable and Transparent AI and Multi-Agent Systems - Third International Workshop, EXTRAAMAS 2021, Virtual Event, May 3-7, 2021, Revised Selected Papers*, volume 12688 of *Lecture Notes in Computer Science*, pages 296–313. Springer, 2021. ISBN 978-3-030-82016-9. doi:10.1007/978-3-030-82017-6_18.

[14] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. ISBN 978-0-201-53082-7.

[15] Melvin Fitting. *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science. Springer, 1996. ISBN 978-1-4612-7515-2. doi:10.1007/978-1-4612-2360-3.

[16] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. ISBN 978-0-521-89957-4.

[17] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9. URL `https://www.sciencedirect.com/book/9780444508133/handbook-of-automated-reasoning`.

[18] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [17], pages 335–367. ISBN 0-444-50813-9. doi:10.1016/B978-044450813-3/50008-4.

[19] William McCune. OTTER 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003. doi:arXiv:cs/0310056.

[20] Andrei Voronkov. The anatomy of Vampire implementing bottom-up procedures with code trees. *J. Autom. Reason.*, 15(2):237–265, 1995. doi:10.1007/BF00881918.

[21] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994. doi:10.1093/LOGCOM/4.3.217.

[22] Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2011. ISBN 978-3-642-22437-9. doi:10.1007/978-3-642-22438-6_23.

[23] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2012. ISBN 978-3-642-31364-6. doi:10.1007/978-3-642-31365-3_30.

[24] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of*

the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014. ISBN 978-3-319-08866-2. doi:10.1007/978-3-319-08867-9_46.

[25] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013. ISBN 978-3-642-39798-1. doi:10.1007/978-3-642-39799-8_1.

[26] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 495–507. Springer, 2019. ISBN 978-3-030-29435-9. doi:10.1007/978-3-030-29436-6_29.

[27] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009. ISBN 978-3-642-02958-5. doi:10.1007/978-3-642-02959-2_10.

[28] Geoff Sutcliffe. The CADE ATP system competition - CASC. *AI Mag.*, 37(2):99–101, 2016. doi:10.1609/AIMAG.V37I2.2620.

[29] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger H. Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. ASlib: A benchmark library for algorithm selection. *Artif. Intell.*, 237:41–58, 2016. doi:10.1016/J.ARTINT.2016.04.003.

[30] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2016. ISBN 978-3-319-40228-4. doi:10.1007/978-3-319-40229-1_23.

[31] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. ISBN 9780387848570. doi:10.1007/978-0-387-84858-7.

[32] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. doi:10.1016/J.NEUNET.2014.09.003.

[33] Jan Jakubův and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2017. ISBN 978-3-319-62074-9. doi:10.1007/978-3-319-62075-6_20.

[34] Karel Chvalovský, Jan Jakubův, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2019. ISBN 978-3-030-29435-9. doi:10.1007/978-3-030-29436-6_12.

[35] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL `https://openreview.net/forum?id=SkZxCk-0Z`.

[36] Jan Jakubův, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020. ISBN 978-3-030-51053-4. doi:10.1007/978-3-030-51054-1_29.

[37] Ibrahim Abdelaziz, Maxwell Crouse, Bassem Makni, Vernon Austel, Cristina Cornelio, Shajith Ikbal, Pavan Kapanipathi, Ndivhuwo Makondo, Kavitha Srinivas, Michael Witbrock, and Achille Fokoue. Learning to guide a saturation-based theorem prover. *IEEE Trans. Pattern Anal. Mach. Intell.*, 45(1):738–751, 2023. doi:10.1109/TPAMI.2022.3140382.

[38] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017. doi:10.29007/8MWC.

[39] Martin Suda. Improving ENIGMA-style clause selection while learning from history. In André Platzer and Geoff Sutcliffe, editors, *Automated*

Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings, volume 12699 of *Lecture Notes in Computer Science*, pages 543–561. Springer, 2021. ISBN 978-3-030-79875-8. doi:10.1007/978-3-030-79876-5_31.

[40] Martin Suda. Vampire with a brain is a good ITP hammer. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 192–209. Springer, 2021. ISBN 978-3-030-86204-6. doi:10.1007/978-3-030-86205-3_11.

[41] Lasse Blaauwbroek, David M. Cerna, Thibault Gauthier, Jan Jakubův, Cezary Kaliszyk, Martin Suda, and Josef Urban. Learning guided automated reasoning: A brief survey. In Venanzio Capretta, Robbert Krebbers, and Freek Wiedijk, editors, *Logics and Type Systems in Theory and Practice - Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*, volume 14560 of *Lecture Notes in Computer Science*, pages 54–83. Springer, 2024. ISBN 978-3-031-61715-7. doi:10.1007/978-3-031-61716-4_4.

[42] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 59(4), 2017. ISSN 0168-7433. doi:10.1007/s10817-017-9407-7.

[43] Geoff Sutcliffe. The 12th IJCAR automated theorem proving system competition–CASC-J12. *The European Journal on Artificial Intelligence*, 38(1):3–20, 2025. doi:10.1177/30504554241305110.

[44] Geoff Sutcliffe and Martin Desharnais. The 11th IJCAR automated theorem proving system competition - CASC-J11. *AI Commun.*, 36(2): 73–91, 2023. doi:10.3233/AIC-220244.

[45] Krystof Hoder, Giles Reger, Martin Suda, and Andrei Voronkov. Selecting the selection. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2016. ISBN 978-3-319-40228-4. doi:10.1007/978-3-319-40229-1_22.

[46] Stephan Schulz. E 2.4 user manual. EasyChair preprint no. 2272, Manchester, 2020. URL `https://easychair.org/publications/preprint/8dss`.

[47] Elias Schede, Jasmin Brandt, Alexander Tornede, Marcel Wever, Viktor Bengs, Eyke Hüllermeier, and Kevin Tierney. A survey of methods for automated algorithm configuration. *J. Artif. Intell. Res.*, 75:425–487, 2022. doi:10.1613/JAIR.1.13676.

[48] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evol. Comput.*, 27(1):3–45, 2019. doi:10.1162/EVCO_A_00242.

[49] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formaliz. Reason.*, 3(2):153–245, 2010. doi:10.6092/ISSN.1972-5787/1980.

[50] Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reason.*, 37(1-2):21–43, 2006. doi:10.1007/S10817-006-9032-3.

[51] Jasmin Christian Blanchette, Max W. Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2015. ISBN 978-3-319-20614-1. doi:10.1007/978-3-319-20615-8_1.

[52] Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. ISBN 978-3-95977-252-5. doi:10.4230/LIPICS.ITP.2022.8.

[53] Alexander Steen. Extensional paramodulation for higher-order logic and its effective implementation Leo-III. *Künstliche Intell.*, 34(1):105–108, 2020. doi:10.1007/S13218-019-00628-8.

[54] Petar Vukmirovic, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making higher-order superposition work. *J. Autom. Reason.*, 66(4):541–564, 2022. doi:10.1007/S10817-021-09613-Z.

[55] Ahmed Bhayat and Martin Suda. A higher-order Vampire (short paper). In Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, volume 14739 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 2024. ISBN 978-3-031-63497-0. doi:10.1007/978-3-031-63498-7_5.

[56] Angelo Salatino, Fabrizio Fornari, and Jelena Zdravkovic. GenAI usage taxonomy, December 2024. URL `https://ceur-ws.org/GenAI/Taxonomy.html`. [Online; accessed 20. Dec. 2024].

[57] ICORE conference portal, January 2025. URL `https://portal.core.edu.au/conf-ranks`. [Online; accessed 13. Jan. 2025].

[58] Amy Brand, Liz Allen, Micah Altman, Marjorie M. K. Hlava, and Jo Scott. Beyond authorship: attribution, contribution, collaboration, and credit. *Learn. Publ.*, 28(2):151–155, 2015. doi:10.1087/20150211.

Appendix A

Author's Publications

This section lists the main publications I contributed to during my doctoral studies. All of these publications are related to the topic of this dissertation thesis and contribute to the research presented in the thesis. Table A.1 shows an overview of the conference and workshop publications.

Event	Year	CORE[1]	WoS[2]	Citations All	Citations Ext.[3]	Chapter	Bibliography
PAAR	2020			7	5	2	[1]
CADE	2021	A	✓	9	6	3	[2]
LPAR	2023	A		3	1	4	[3]
IJCAR	2024	A	✓	5	2	5	[4]
PAAR	2024			0	0	6	[5]

Table A.1: Author's conference and workshop publications. Each publication is identified by the event (conference or workshop) at which it was published.

The numbers of citations were collected from Google Scholar on March 21, 2025. In this summary, an *external citation* is a citation from a publication with no common author.

For each of the publications included in this chapter, the contributions of individual authors are structured according to the CRediT contributor role taxonomy [58]. The authors are identified by their initials. Where appropriate, contributors other than authors are credited.

[1] CORE conference rank [57]
[2] Whether or not the publication is indexed in Web of Science
[3] External citations – excluding self-citations

A.1 Publications Indexed in Web of Science

Neural Precedence Recommender

Authors *Filip Bártek* and Martin Suda

Title Neural Precedence Recommender [2]

Conference Conference on Automated Deduction (CADE) 2021[4]

Public acceptance Number of external citations: 6

This paper is included in chapter 3.

Author contributions. Conceptualization: M.S. and *F.B.*; Formal analysis: *F.B.*; Funding acquisition: M.S., *F.B.*, and Josef Urban; Investigation: *F.B.*; Methodology: *F.B.* and M.S.; Project administration: M.S.; Resources: Josef Urban; Software: *F.B.* and M.S.; Supervision: M.S.; Visualization: *F.B.*; Writing – original draft: *F.B.* and M.S.; Writing – review & editing: *F.B.* and M.S.

Regularization in Spider-Style Strategy Discovery and Schedule Construction

Authors *Filip Bártek*, Karel Chvalovský, and Martin Suda

Title Regularization in Spider-Style Strategy Discovery and Schedule Construction [4]

Conference International Joint Conference on Automated Reasoning (IJCAR) 2024[5]

Public acceptance Number of external citations: 2

This paper is included in chapter 5.

Author contributions. Conceptualization: M.S., *F.B.*, and K.C.; Data curation: *F.B.* and M.S.; Formal analysis: *F.B.* and M.S.; Funding acquisition: M.S., Josef Urban, and *F.B.*; Investigation: *F.B.*, M.S., and K.C.; Methodology: M.S., *F.B.*, and K.C.; Project administration: M.S.; Resources: Josef Urban; Software: *F.B.*, M.S., and K.C.; Supervision: M.S.; Validation: *F.B.*; Visualization: *F.B.* and M.S.; Writing – original draft: M.S., *F.B.*, and K.C.; Writing – review & editing: *F.B.*, M.S., and K.C.

[4]CORE2021 conference rank: A
[5]CORE2023 conference rank: A

A.2 Other Publications

A.2.1 Conference and Workshop Papers

Learning Precedences from Simple Symbol Features

Authors *Filip Bártek* and Martin Suda

Title Learning Precedences from Simple Symbol Features [1]

Conference Workshop on Practical Aspects of Automated Reasoning (PAAR) 2020

Public acceptance Number of external citations: 5

This paper is included in chapter 2.

Author contributions. Conceptualization: M.S. and *F.B.*; Formal analysis: *F.B.*; Funding acquisition: Josef Urban, M.S., and *F.B.*; Investigation: *F.B.*; Methodology: M.S. and *F.B.*; Project administration: M.S.; Resources: Josef Urban; Software: *F.B.* and M.S.; Supervision: M.S.; Visualization: *F.B.*; Writing – original draft: *F.B.* and M.S.; Writing – review & editing: *F.B.* and M.S.

A GNN-Advised Clause Selection

Authors *Filip Bártek* and Martin Suda

Title How much should this symbol weigh? A GNN-Advised Clause Selection [3]

Conference International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR) 2023[6]

Public acceptance Number of external citations: 1

This paper is included in chapter 4.

Author contributions. Conceptualization: M.S., *F.B.*, and Cezary Kaliszyk; Formal analysis: *F.B.* and M.S.; Funding acquisition: M.S., Josef Urban, and *F.B.*; Investigation: *F.B.*; Methodology: *F.B.* and M.S.; Project administration: M.S.; Resources: Josef Urban; Software: *F.B.*; Supervision: M.S.; Visualization: *F.B.*; Writing – original draft: *F.B.* and M.S.; Writing – review & editing: *F.B.* and M.S.

[6] CORE2021 conference rank: A

Cautious Specialization of Strategy Schedules

Authors *Filip Bártek*, Karel Chvalovský, and Martin Suda

Title Cautious Specialization of Strategy Schedules (Extended Abstract) [5]

Conference Workshop on Practical Aspects of Automated Reasoning (PAAR) 2024

This extended abstract is included in chapter 6.

Author contributions. Conceptualization: M.S., K.C., and *F.B.*; Data curation: M.S. and K.C.; Formal analysis: M.S. and K.C.; Funding acquisition: M.S.; Investigation: K.C. and M.S.; Methodology: M.S. and K.C.; Project administration: M.S. and *F.B.*; Resources: Josef Urban; Software: K.C. and M.S.; Supervision: M.S.; Visualization: M.S.; Writing – original draft: M.S., K.C., and *F.B.*; Writing – review & editing: *F.B.*, M.S., and K.C.

A.2.2 Datasets

Vampire strategy performance measurements

Authors *Filip Bártek* and Martin Suda

Title Vampire strategy performance measurements [6]

This dataset is a supplement to our paper Regularization in Spider-Style Strategy Discovery and Schedule Construction [4] included in chapter 5.

Author contributions. Conceptualization: *F.B.* and M.S.; Data curation: *F.B.* and M.S.; Formal analysis: *F.B.*; Funding acquisition: M.S., Josef Urban, and *F.B.*; Investigation: *F.B.*; Methodology: *F.B.*; Project administration: M.S.; Resources: Josef Urban; Software: *F.B.*; Supervision: M.S.; Visualization: *F.B.*; Writing – original draft: *F.B.*; Writing – review & editing: M.S.

Návrh na vypsání rámcového tématu

disertační práce v DSP na ČVUT FEL.

Jméno a příjmení školitele (vč. Titulů): Mgr. Josef Urban, PhD., spoluškolitel: RNDr. Martin Suda, PhD.

Navrhované rámcové téma disertační práce: Machine learning for saturation-based theorem proving

Vybraná publikace školitele, která se váže k navrhovanému tématu:
Karel Chvalovský, Jan Jakubuv, Martin Suda, Josef Urban:
ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E. CoRR abs/1903.03182 (2019), accepted to CADE 2019.
Cezary Kaliszyk, Josef Urban, Henryk Michalewski, Miroslav Olsák:
Reinforcement Learning of Theorem Proving. NeurIPS 2018: 8836-8847 .
Jan Jakubuv, Martin Suda, Josef Urban:
Automated Invention of Strategies and Term Orderings for Vampire. GCAI 2017: 121-133.
Krystof Hoder, Giles Reger, Martin Suda, Andrei Voronkov:
Selecting the Selection. IJCAR 2016: 313-329.
Giles Reger, Martin Suda, Andrei Voronkov:
Playing with AVATAR. CADE 2015: 399-415.

Počet citací (školitele) dle WOS[1]: 414 H-index[2]: 14 (WOS)

Podrobnější popis tématu:

The goal of the project is to investigate the topic of combining machine learning methods with state-of-the-art saturation-based Automated Theorem Provers. The Ph.D. candidate will apply modern machine learning techniques such as deep neural networks and reinforcement learning to improve the quality of prover's decisions at key heuristic choice points such as premise selection, clause selection, and the choice of strategy parameters. The work includes the design of new learning architectures particularly suited for dealing with logical formulas and related objects.

Datum a podpis školitele: 29.4. 2019

Vyjádření vedoucího školicího pracoviště (uveďte kód pracoviště):

Datum a výsledek projednání ORO (uveďte název oboru):

[1] S vyloučením autocitací.
[2] Citovaná i citující práce musí být v časopise excerpovaném SCI Expanded, nebo v odpovídající databázi v případě humanitních oborů.
Školitel předloží prostřednictvím vedoucího školicího pracoviště ke schválení příslušné oborové radě. Pokud oborová rada téma schválí, předá dokument v elektronické formě i písemně oddělení VVZS.
Pokud má školitel školit současně více než 5 doktorandů, nebo pokud je jeho časopisecký H-index nulový, předloží ORO nejpozději společně s tímto formulářem i příslušné návrhy na výjimky.