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 $g(x) = \frac{1}{1+e^{-x}}$.

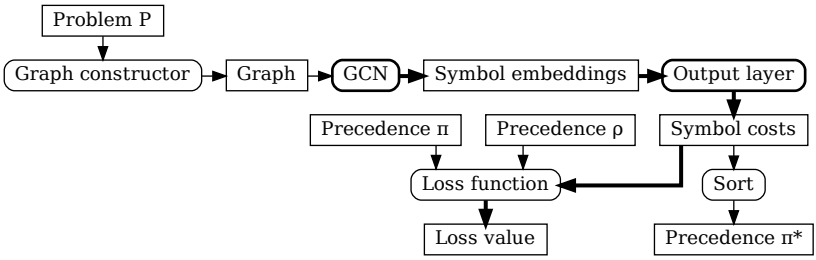

Fig. 1. Recommender architecture overview. When recommending a precedence, the input is problem P and the output is precedence π^*. When training, the input is problem P and precedences π and ρ, and the output is the loss value. The trainable modules and the edges along which the loss gradient is propagated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

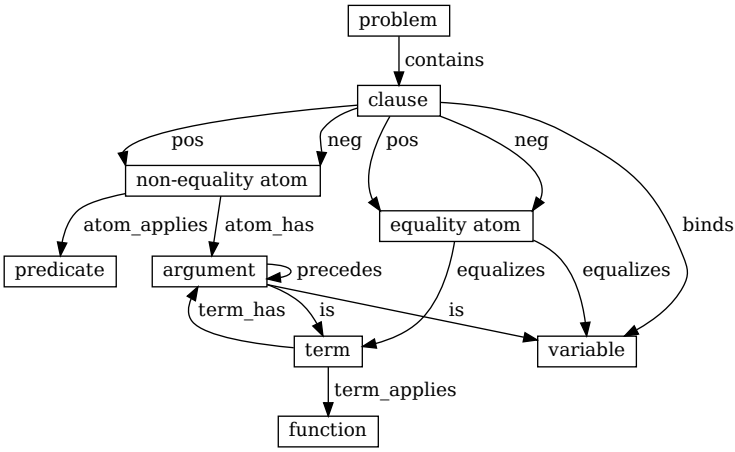

Fig. 2. CNF graph schema

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

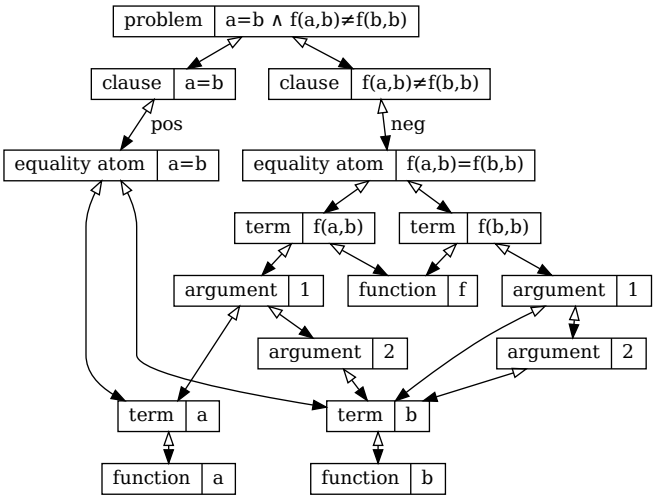

Fig. 3. Graph representation of the CNF formula $a = b \wedge f(a, b) \neq f(b, b)$

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 multiarmed 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