# Learning Proof Path Selection Policies in Neural Theorem Proving

Matthew Morris[1,*], Pasquale Minervini[2] and Phil Blunsom[1,3]

[1]Computer Science Department, University of Oxford
[2]UCL Centre for Artificial Intelligence, University College London
[3]DeepMind, London

Abstract
Neural Theorem Provers (NTPs) are neural relaxations of the backward-chaining logic reasoning algorithm. They can learn continuous representations for predicates and constants, induce interpretable rules, can provide logic explanations for their predictions, and show strong systematic generalisation properties. However, since they enumerate all possible proof paths for proving a goal, they suffer from high computational complexity, and are thus unsuitable for complex reasoning tasks. Conditional Theorem Provers (CTPs) try to overcome this issue by generating relevant rules on-the-fly based on the goal, rather than considering all possible rules. Nonetheless, CTPs suffer from similar computational constraints, as they still have to consider multiple proof paths while reasoning. We propose Adaptive CTPs (ACTPs), where CTPs are augmented with a learned policy to dynamically select the most promising proof paths. This allows the model designer to specify the number of proof paths to consider, to conform to the computational constraints of their use case, while retaining all of the benefits of CTPs. By evaluating on the CLUTRR dataset, we provide evidence for the computational issues in existing CTP models, show that ACTPs alleviate these issues, and demonstrate that, in certain scenarios, the accuracy achieved by ACTPs is higher than CTPs while retaining the same computational complexity.

**Keywords**
reasoning, neuro-symbolic, adaptive computation, reinforcement learning

## 1. Introduction

Promising work has been done around integrating neural models and symbolic reasoning, as their complementary strengths and weaknesses make for powerful models when combined [1, 2, 3, 4, 5]. In this paper, we consider such a technique: Neural Theorem Provers [6].

**Neuro-symbolic Reasoning** The approach of Rocktäschel and Riedel [6] is to keep variable binding symbolic, but compare predicates and constants using their sub-symbolic representations. They introduce Neural Theorem Provers (NTPs): end-to-end differentiable provers for theorems formulated as queries to a knowledge base (KB). Prolog's backward chaining algorithm [7] is used as a blueprint for constructing neural networks in a recursive manner,

*NeSy 2022, 16th International Workshop on Neural-Symbolic Learning and Reasoning, Cumberland Lodge, Windsor, UK*
*Corresponding author.
matthewthemorris@gmail.com (M. Morris); p.minervini@cs.ucl.ac.uk (P. Minervini); phil.blunsom@cs.ox.ac.uk (P. Blunsom)
0000-0003-3337-7229 (M. Morris); 0000-0002-8442-602X (P. Minervini); 0000-0003-4558-2457 (P. Blunsom)

which can prove queries to a KB using vector representations of symbols. These proofs are given success scores, which are differentiable with respect to the sub-symbolic representations, allowing the model to learn representations that maximise the proof scores. Using the same process, rules of pre-defined structures are also learnt.

NTPs can learn representations of symbols in a KB like neural link prediction models and learn rules which hold in the KB. They also allow one to incorporate already known rules into the reasoning process, as one simply needs to include them in the knowledge base. NTPs are also naturally interpretable, since they induce sub-symbolic rules that can be decoded to human-readable symbolic rules. Finally, Minervini et al. [8] demonstrate that NTPs have the ability to perform systematic generalisation, learning how to solve complex reasoning tasks while only being trained on simpler examples. In contrast, many neural models appear not to generalise robustly on tasks requiring systematic generalisation [9, 10, 11, 12].

However, NTPs have a significant computational footprint, as they consider all possible rules for proving a goal or sub-goal. This means they cannot scale to settings with a large number or rules or reasoning steps. To solve this problem, Minervini et al. [8] propose Conditional Theorem Provers (CTPs), an extension of NTPs that learn to dynamically generate a set of rules for proving the current goal or sub-goal. This is implemented by a `select` module which, given a goal, returns multiple rules that can be used to prove that goal. It consists of multiple neural networks, which we refer to as *reformulators*, each of which can represent several rules in a knowledge base. This module is end-to-end differentiable, and can be trained end-to-end via backpropagation.

However, CTPs can end up suffering from computational issues in a similar fashion to NTPs, since they still need to consider multiple proof paths during the reasoning process. For complex datasets in which there are many ways to prove a given goal, more proof paths need to be checked. This, in conjunction with the high reasoning depth often required for such datasets, causes CTPs to become infeasibly slow. This is especially problematic when CTPs are applied in settings where the inference time is critical.

**Objectives** In this paper, we aim to address the computational shortcomings of CTPs by extending them to ACTPs (*Adaptive* CTPs). Specifically, we augment CTPs with a policy trained to select the proof paths that are most likely to succeed. This allows one to control the amount of exploration in the space of proof paths for a given goal, depending on the computational requirements of the task. An alternative to addressing the computational issues of CTPs is to simply reduce the number of reformulators trained, leading to fewer proof paths being considered. However, this makes the model less expressive, meaning it will likely be unable to capture all of the rules in a knowledge base. Thus, for ACTPs to be useful, an ACTP model expanding only $k$ proof paths should achieve higher accuracy than a CTP model with only $k$ reformulators. In the following, we 1) motivate for CTP models sometimes requiring a large number of reformulators and reasoning depth; 2) concretely establish the computational issues that CTPs suffer from, using both empirical results and a theoretical analysis; 3) define a framework for ACTPs using policy gradient descent; 4) empirically demonstrate on the CLUTRR dataset that ACTPs are an improvement upon CTPs, in both their respective accuracies and evaluation times.

## 2. Neural Theorem Proving

**Backward Chaining** Prolog [7] is a logic programming language that finds use in contemporary work. It has been used for a variety of tasks, including automated theorem proving [13], expert systems [14], and natural language processing [15]. A Prolog KB consists of *rules* and *facts*. Queries are passed to the KB, with Prolog returning whether or not the queries are entailed by the KB. The restrictive syntax of Prolog allows one to answer queries using Prolog's *backward chaining algorithm* [7]. Given a goal, such as $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$, which is constructed from a query, Prolog tries to find substitutions for the goal by using the rules in the KB. The process of checking if the head of a rule matches a goal is called *unification*. If unification succeeds, then the goal is replaced with the atoms from the body of the rule, giving a new set of sub-goals. The same process is then applied to each of these sub-goals, continuing recursively until all sub-goals are found as facts in the KB or there are no more rules to apply.

**Neural Theorem Provers** Neural Theorem Provers (NTPs), proposed by Rocktäschel and Riedel [6], are a continuous relaxation of the backward chaining algorithm. NTPs can be trained end-to-end, by calculating the gradient of proof successes with respect to vector representations of symbols, and are defined in terms of modules [16]. The recursive expansion upon the goal is kept track of in *proof states*, which each contain a neural network that outputs the success score of the proof so far, and the substitution set. The network is recursively built upon, with new nodes being added as rules are applied. Every different proof path will have a different associated proof state. However, as noted by Rocktäschel and Riedel [6], NTPs suffer from severe computational limitations. In standard backward chaining, a proof path can be aborted when unification with a rule fails, but this happens far less in neural backward chaining, since unification only fails when predicates do not have matching arities. Given a goal such as $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$, the prover should only consider rules such as the first one below, and not the second.

$$\texttt{sister}(X, Y) \leftarrow \texttt{father}(X, Z) \wedge \texttt{daughter}(Z, Y)$$
$$\texttt{father}(Y, Z) \leftarrow \texttt{son}(Y, X) \wedge \texttt{grandfather}(X, Z)$$

**Conditional Theorem Provers** To address the computational limitations of NTPs, Minervini et al. [8] propose introducing a new module into the system, `select`, to reduce the number of rules being considered when expanding upon a goal. In the `or` module, instead of considering every rule $H \leftarrow B \in \mathscr{K}$ in the KB $\mathscr{K}$, they only consider each $H \leftarrow B \in \texttt{select}_\theta(G)$. This *select* module can be implemented by a sequence of differentiable parameterized functions $\texttt{select}^1_\theta(G), \texttt{select}^2_\theta(G), ..., \texttt{select}^n_\theta(G)$ that each, given a goal, produces a sequence of sub-goals. We refer to each of these functions as a *goal reformulation module*, or simply a *reformulator*.

In summary, a CTP is composed of $n$ reformulators, each of which can represent multiple rules, and it learns representations for predicates and constants. All model parameters are initialised randomly, and trained end-to-end via backpropagation. During inference, a CTP model starts with a goal $p(c_1, c_2)$, and initialises $G = \{p(c_1, c_2)\}$ as the set of sub-goals. Then, recursively up to the given reasoning depth $d$, the model applies each reformulator to each

**Table 1**
CTP Test Accuracy for a Varying Number of Reformulators

| Number | 1.4 AVG | 1.4 MAX | 1.10 AVG | 1.10 MAX |
|---|---|---|---|---|
| 1 | 35.5 | 41.6 | 42.7 | 54.1 |
| 2 | 51.8 | 74.0 | 49.6 | 58.2 |
| 3 | 69.1 | 81.8 | 66.6 | 83.6 |
| 4 | 72.6 | 83.1 | 75.9 | **90.2** |
| 5 | 73.5 | 87.0 | 74.8 | 86.1 |
| 6 | 77.2 | 88.3 | 73.8 | 86.1 |
| 7 | 76.6 | 84.4 | 76.1 | 86.9 |
| 8 | **79.9** | **89.6** | **77.9** | 86.9 |

of the sub-goals in $G$, generating a new set of sub-goals from the output of the reformulators with each recursive step. At each recursive step, the model produces a new proof path for each reformulator. The model maximises scores over proof paths. At every depth up to the reasoning depth $d$, the model unifies every sub-goal from the proof path with the knowledge base of facts, given in the dataset task. These similarity values are propagated up, with the score of the atom in the head of a reformulator being set to the minimum similarity of all the atoms in its body.

**Evaluation Dataset** CLUTRR [12] is a system for constructing artificial datasets modelling family relationships. Given a set of family relations, the task is to infer the relationship between two family members whose relationship is not explicit in the set. To solve this, an agent ought to be able to induce the logical rules that govern family relationships, and use those rules to infer the relationship of the query members from the given relations. In particular, CLUTRR allows for testing an agent's ability to perform *systematic generalization* [17, 12, 18]. Sinha et al. [12] published several generated dataset groups alongside the CLUTRR system, which we make use of for training and evaluation in this paper. We refer to the training dataset as *1.2,1.3,1.4_train* and the testing datasets as *1.2_test*, *1.3_test*, ..., *1.10_test*.

## 3. Expressivity and Complexity

The expressivity of a relational learning model is an important consideration for its viability [19, 20]. A more expressive model can capture more types of data and relations than a less expressive one, meaning it can be applied to more complex datasets. Due to the restrictive nature of the syntax of Prolog, CTPs are already quite limited when it comes to the structure of expressions upon which they can reason. However, these limitations go further, as the rules that a CTP model can capture depend on the structure and number of reformulators in the model.

**Number of Reformulators** A single reformulator can capture any number of rules, provided that for each rule, the positions of the variables in the rule correspond to the positions of their representations in the reformulator. In addition, given an atom for the head of a rule, a

**Table 2**
CTP Test Accuracy for a Varying Reasoning Depth

| Depth | 1.4 AVG | 1.4 MAX | 1.10 AVG | 1.10 MAX |
|---|---|---|---|---|
| 1 | 18.61 | 22.08 | 15.57 | 16.39 |
| 2 | 31.17 | 41.56 | 22.95 | 26.23 |
| 3 | 71.00 | 84.42 | 45.90 | 51.64 |
| 4 | 71.00 | 84.42 | 68.31 | 76.23 |
| 5 | **71.86** | **84.42** | **75.96** | **86.89** |

reformulator can only capture one rule with the given head. For example, a single reformulator would be unable to capture both of the following rules, even though the positions of the variables are the same in both:

$$\begin{aligned}\texttt{sister}(X,Y) &\leftarrow \texttt{father}(X,Z) \wedge \texttt{daughter}(Z,Y)\\ \texttt{sister}(X,Y) &\leftarrow \texttt{sister}(X,Z) \wedge \texttt{sister}(Z,Y)\end{aligned}$$

This is because a reformulator is a function that maps from $A \in \mathbb{R}^k \times (\mathbb{R}^k \cup V) \times (\mathbb{R}^k \cup V)$, meaning that each $A$ has to map to a unique output. As such, to fully capture all of the rules in a knowledge base, we need as many reformulators as there are the maximum number of rules with the same head atom. We refer to this number as the *minimal full expressivity bound.* For the CLUTRR datasets, this number is 5. To see the effect of increasing the number of reformulators, we fix the train and test reasoning depths, and evaluate the average/maximum test accuracy of CTPs on on *1.4_test* and *1.10_test*, using a number of reformulators varying from 1 to 8. The results are shown in Table 1.

**Reasoning Depth** CTPs reason up to a pre-defined reasoning depth and then unify the existing sub-goals with the facts in the knowledge base. This means that, even if the CTP model has perfectly learned to represent all of the rules in the knowledge base with reformulators, it still needs a sufficient reasoning depth to prove the goal. The required reasoning depth for a valid proof path is the minimum number of recursive steps down, such that every sub-goal appears in the knowledge base of facts. The reasoning depth needed to solve a task is thus the minimum required reasoning depth across all possible proof paths. For the most complex CLUTRR instance, this number is at most 5. In Table 2, we demonstrate the effect of using different test reasoning depths.

**Computational Issues** While we have demonstrated that it is rarely harmful and often beneficial to increase the reasoning depth and number of reformulators in terms of predictive accuracy, doing so can lead to higher computational costs. In Appendix A, we provide a theoretical analysis of the time complexity of CTPs. We consider two base operations that we wish to count: the number of reformulator applications (data being passed through a neural network) and the number of sub-goals that need unifying with the knowledge base after the reasoning depth is reached (comparisons with all fact embeddings using a RBF kernel). The remainder of the operations in CTPs are either tied into one of these two, or take constant time. We count these operations with respect to the number of reformulators used $n$, the reasoning depth $d$, and the maximum number of atoms in the body of any reformulator used $m$. We find that the time complexity of CTPs as a whole is $\mathcal{O}((nm)^d)$. We illustrate the issues resulting from this empirically in Appendix B.

## 4. Optimization via REINFORCE

**Outline of Solution** Since the time complexity of CTPs is $\mathcal{O}((nm)^d)$, optimizing CTP evaluation time consists of trying to keep each of these variables as low as possible. The maximum number of atoms in the body of any reformulator used $m$ is almost impossible to reduce, as it is completely determined by the rules the model is trying to capture in the knowledge base. The depth $d$ is also challenging to reduce, as a certain depth is required for full expressivity on the test datasets. Our approach is to keep the number of reformulators $n$ used at each expansion step as low as possible, minimizing the number of proof paths that need to be considered.

**Choosing Reformulators** Given that the time complexity of CTPs is $\mathcal{O}((nm)^d)$, having $n$ reformulators and only selecting $k$ at each expansion step would lead to a time complexity of $\mathcal{O}((km)^d)$ instead. Ideally, we would just be able to use $k = 1$ and learn to select the perfect reformulator at every expansion step. However, initial experiments demonstrated this to be an almost impossible task since, for each goal and sub-goal in CLUTRR, we do not know in advance which proof path will allow us to prove it. Rather, we use an hyperparameter $k \in \mathbb{N}$, tuned for maximising accuracy and satisfying the computational constraints of the test datasets. This means that $k$ instead of $n$ reformulators are selected and used at every expansion step in the reasoning. We refer to the module making these selection decisions as the *selection module*.

**Implementation** We adopt REINFORCE [21] to train the reformulation selection module. Let us first define the corresponding deterministic Markov decision process $(S, A, \delta, R)$:

**States.** A state in the model represents which sub-goal we are currently considering for expansion in the proof. It is thus an atom $A$ where $A \in \mathbb{R}^k \times (\mathbb{R}^k \cup V) \times (\mathbb{R}^k \cup V)$. In order that the policy estimator may be implemented by a neural network, we define $S := \mathbb{R}^k \times \mathbb{R}^k \times \mathbb{R}^k = \mathbb{R}^{3k}$. We fixed the value of a variable to be $\{0\}^k$.

**Actions.** The set of possible actions from any state is the reformulators that could be used to expand upon the sub-goal. With $n$ reformulators, we thus have $A = \{1, ..., n\}$. Note that as we are choosing $k$ reformulators for expansion, multiple actions are chosen for a given state by sampling without repetition from the probability distribution $\pi_\theta(s)$.

**Transition Function.** After some subset of reformulators is chosen, the CTP model proceeds as normal, expanding out into a different proof path for each reformulator. Thus, we have a deterministic transition function $\delta$ defined by this process.

**Rewards**. The proof score is an obvious choice for reward signal, as higher proof scores correspond to the model performing better on positive tasks. However, rather than discounting future rewards, we can use the fact that proof scores are propagated back up through the CTP model to have access to the exact score that choosing a reformulator leads to. The reward given for choosing a reformulator is thus the maximum proof score across all proof paths originating from the reformulator applied to the current sub-goal.

The policy network is implemented by a neural network with a single hidden layer, containing 30 hidden nodes. We use a Rectified Linear Unit (ReLU) activation function before the hidden layer: $\mathrm{ReLU}(x) := \max(0, x)$. The output of the policy network is thus

$$\pi_\theta(s) := \mathrm{softmax}(W_2 \times \mathrm{ReLU}(W_1 \times s))$$

while the loss is given by $L(\theta) = -\frac{1}{B}\sum_{b=1}^{B} R_b \times \ln(\pi_\theta(s_b)_{a_b})$, where $B$ is the batch size, $a_b$ is the action chosen in a particular task in the batch, $R_b$ is the proof score that resulted from the action, and $\pi_\theta(s_b)_{a_b}$ is the probability that action $a_b$ has in the distribution $\pi_\theta(s_b)$. The loss is applied separately for each of the $k$ actions (reformulators) chosen. Rather than having episodes, we simply execute the CTP model as usual, applying the policy and calculating the loss at every expansion step in the reasoning. We refer to this model as *ACTP* and the original baseline CTP model as *CTP*.

**ACTP Speedup** We see that if ACTPs are used, there is at least a theoretical speedup from the baseline of CTPs. Ideally this would translate into a speedup in wall-clock time as well, but this is not guaranteed for all datasets and values of $m$ and $d$. The larger the values of $m$ and $d$, the greater the effect of choosing $k$ from $n$ reformulators will have; this follows directly from the computational complexity $\mathcal{O}((km)^d)$. Furthermore, larger and more complex datasets will also see this effect being more pronounced, as they contain more facts that need to be unified with the $\mathcal{O}((km)^d)$ sub-goals once the test depth is reached.

Counteracting this is the overhead that comes from having to do the selection at each reasoning step, instead of just applying every reformulator. If the overhead is high enough, then better wall-clock times might not present themselves for the evaluations we do on CLUTRR. However, even if this is not the case, we argue that the theoretical speedup of this method proves its usefulness regardless: eventually the dataset complexity and reasoning depths will be high enough that the resulting theoretical speedup overcomes the overhead that comes with the method. Serious computational concerns for CTPs will occur more often for complex datasets and high reasoning depths, which is exactly when the theoretical speedup becomes an advantage.

## 5. Related Work

Other works have already explored models that learn to traverse a knowledge base. Das et al. [22], Xiong et al. [23] use reinforcement learning to learn inference paths in large knowledge bases. Both of these works are based upon the path ranking algorithm [24], which uses random walks with restarts to perform several upper-bounded depth-first searches to find paths along relations. When combined with elastic-net based learning, the algorithm can learn to choose paths which are more likely to complete the inference.

Das et al. [22] propose MINERVA, a method for searching a knowledge graph for answer-providing paths using reinforcement learning, conditioned on the query. Given a knowledge graph, it attempts to learn a policy which, given a query of the form $\texttt{predicate}(\mathrm{c}, X)$, starts from c and walks over relations (edges in the knowledge graph), choosing a relation at each step, conditioned upon the query predicate and the walk so far. This is done with reinforcement learning by trying to maximize the reward: reaching the correct answer constant. Xiong et al. [23] adopt a similar but slightly simpler approach with DeepPath, which also uses reinforcement learning to find paths between pairs of constants. However, in contrast to Das et al. [22], they also condition upon the answer constant while traversing the graph.

While our proposed method is not exactly what MINERVA and DeepPath do, their existence and success at least indicate that the problem of learning which reasoning steps to take in a knowledge base is a solvable one. Our work also runs parallel to that of Asgharbeygi et al. [25], Crouse et al. [26], both of which use reinforcement learning as a search heuristic to optimise reasoning, albeit in different settings. Finally, we also draw inspiration for our method from the work of Li et al. [27], who provide motivation for training large transformer models and then heavily compressing them before testing. In a similar manner, our method aims to utilize a large number of reformulators when training, and then only use the selected ones during evaluation.

## 6. Experiments

**Experiment Design** We adopt the the hyperparameters in Minervini et al. [8] to train our CTP models. We adopt the procedure of first training the reformulators, and then training the selection module. This means the selection module is learning to select proof paths over actual rules in the knowledge base, and we can independently control how long the reformulators and selection module are trained for. For optimizing hyperparameters, we adopt the same approach to evaluation and test sets as Minervini et al. [8]. We perform two different optimizations: the first has hyperparameters tuned on an evaluation set of *1.3_test* and is tested on all other datasets, and the second is tuned on an evaluation set of *1.9_test*.

**Reformulator Strength** Since all reformulators have different random initializations, they all converge to different local optima. Thus, it is clear that some reformulators will capture more rules than others, and that some will learn a particular rule better than the others. Moreover, certain subsets of reformulators are likely to contribute more to proofs than others, with subsets of the reformulators that better cover the range of rules in the knowledge base giving higher accuracies when used for evaluation. We demonstrate this empirically by training 5 and 8 reformulators respectively, across 10 different seeds and then using 4 random subsets of 3 reformulators each for evaluation. The average accuracies of the highest and lowest scoring subsets are 17% and 51% for 5 reformulators, and 11% and 31% for 8 reformulators.

We see that there is a large discrepancy between the performance of different subsets, indicating that when it comes to maximizing accuracy during evaluation, there are reformulators whose inclusion in the model is far more important.We also present the following hypothesis: as the number of reformulators increases, individual reformulators become weaker. More formally: as we use more reformulators during training, the expected accuracy when using a fixed-size subset of the reformulators during evaluation decreases. This is an important hypothesis to note and prove, as it means that the task of selecting the best reformulators for a proof becomes harder as the total number of reformulators increases. This hypothesis is supported empirically

**Table 3**
CTP accuracy across all datasets, using 8 and 5 reformulators during training, evaluating using 4 random subsets of 3 reformulators each

| Dataset | 8 Ref. | 5 Ref. |
|---|---|---|
| *1.2_test* | 31.3 | **48.8** |
| *1.3_test* | 22.3 | **41.0** |
| *1.4_test* | 20.4 | **33.6** |
| *1.5_test* | 26.1 | **43.5** |
| *1.6_test* | 23.0 | **40.8** |
| *1.7_test* | 21.8 | **36.7** |
| *1.8_test* | 19.3 | **32.9** |
| *1.9_test* | 18.0 | **31.5** |
| *1.10_test* | 17.1 | **30.1** |

by the results in Table 3: we trained CTP models with 5 and 8 reformulators respectively, reporting the average accuracy when 4 random subsets of 3 reformulators were used for evaluation.

**Adaptive CTP Evaluation** Let $n$ be the number of reformulators trained and $k$ the number of proof paths expanded during evaluation. We chose to evaluate ACTPs with $n \in \{5, 8\}$ to measure their effectiveness when more (and individually weaker) reformulators are used and with $k \in \{2, 3\}$ to measure the effectiveness of ACTPs when they are allowed to expand fewer proof paths. We refer to such an ACTP model as ACTP-*n*C*k*. This yielded 4 different scenarios for evaluation.

Only when using $n := 5$ and $k := 3$ were ACTPs an improvement upon CTPs, the results of which are shown in Table 4. Further to this, we see that ACTPs perform significantly worse when only 2 reformulators are chosen instead of 3. ACTP models that only choose 2 reformulators are outperformed by the baseline across every dataset. We hypothesise that, for CLUTRR, the task of learning which two reformulators are the most promising is one that is just too difficult for the model to find a solution to.

As shown in Section 6, as the number of reformulators increases, individual reformulators become weaker. Thus, the task of choosing the optimal reformulators for expansion becomes more difficult as the number of reformulators increases. This means that, all else being constant, the accuracy of ACTP models will drop as more reformulators are trained. This effect is offset by the increasing expressivity of the model as more reformulators are used. Hence, as expected, ACTPs consistently perform worse when more reformulators are trained. This is confirmed by our experimental findings, where ACTP-5C*k* models outperform ACTP-8C*k* models in every scenario and across every dataset.

We also note that tuning hyperparameters on *1.9_test* instead of *1.3_test* causes baseline CTP performance to decrease for the simpler datasets but increase for the more complex datasets. For ACTPs however, this effect appears far less pronounced, with the accuracy of ACTP-5C3 models even dropping slightly for the more complex test datasets, when tuning on *1.9_test* instead of *1.3_test*. This indicates that ACTPs are not learning the reasoning patterns needed for

**Table 4**
One-tailed unpaired t-test between the baseline of CTPs with 3 reformulators and ACTPs with 5 choosing 3 reformulators. Hyperparameters tuned on *1.3_test*

| | **CTP** | | **ACTP** | | |
|---|---|---|---|---|---|
| | $\mu$ | $\pm\,\sigma$ | $\mu$ | $\pm\,\sigma$ | **P-Value** |
| *1.2* | **80.00** | 24.49 | 78.95 | 25.84 | 0.525 |
| *1.3* | 90.65 | 7.24 | **94.58** | 1.09 | 0.147 |
| *1.4* | 76.10 | 4.97 | **81.82** | 4.57 | 0.048 |
| *1.5* | 87.03 | 3.31 | **89.73** | 5.34 | 0.185 |
| *1.6* | 83.81 | 3.41 | **89.52** | 4.82 | 0.033 |
| *1.7* | 78.71 | 1.82 | **86.06** | 5.01 | 0.014 |
| *1.8* | 73.93 | 5.20 | **78.82** | 6.50 | 0.114 |
| *1.9* | 68.87 | 7.58 | **73.87** | 5.94 | 0.140 |
| *1.10* | 67.05 | 4.96 | **69.67** | 6.43 | 0.246 |

the more complex datasets, even when tuned on such a dataset. It is also possible that ACTPs could be overfitting to the evaluation set when tuned on *1.9_test*. However, since the accuracy of ACTPs on *1.9_test* does not even increase that much when tuning on the dataset, we find the former explanation to be more likely. As a final point, we note that ACTPs do not appear to exhibit higher instability in their accuracies than CTPs, with the models showing comparable levels of variance.

## 7. Conclusions

In this paper, we provided motivation for cases in which CTP models would be required to have a large number of reformulators and a high reasoning depth, as well as demonstrating how this leads to computational complexity concerns both theoretically and empirically. We defined a framework for ACTPs as an extension to CTPs, in which reinforcement learning is used to learn to select optimal reformulators for expansion during a proof. This allows the model designer to scale down the number of selected reformulators, such that the computational constraints of the use case may be met. We noted that certain subsets of reformulators perform significantly better than others, and that individual reformulators tend to become weaker as the number of reformulators used in a CTP model increases. This means that the task of selecting reformulators becomes more difficult as the number of reformulators increases. We evaluated ACTPs in 4 separate scenarios, which vary in regard to the number of reformulators trained and the number of reformulators selected. In 1 of these 4 scenarios, we found that ACTPs outperformed CTPs on 8 out of 9 test datasets. The results demonstrate the usefulness of ACTPs over CTPs in certain situations, but also highlight their failing to be a categorical improvement upon CTPs.

## Acknowledgments

This work was mostly done during Matthew's MSc in Computer Science at Oxford. As a result, the following has been written by him. Thanks are first and foremost due to my supervisor, Phil Blunsom. Thank you for introducing me to the topic of this thesis, for sharing so many ideas and papers with me, and for the constant and gentle encouragement. To Pasquale Minervini, the primary creator of CTPs and an outstanding academic at UCL: I convey my most sincere gratitude. I am still amazed that you were willing to spend so long helping me through all my technical and theoretical difficulties. You owed nothing to me, which makes me so much more thankful for all that you did. Thanks are also due to the Oxford Whiteson Research Lab, for allowing me to use their servers for running experiments, and to Luisa in particular, for helping me get everything set up. Finally, thank you to Skye Foundation and The Oppenheimer Memorial Trust for fully funding my degree and expenses while in the UK.

## References

[1] A. d. Garcez, T. R. Besold, L. De Raedt, P. Földiak, P. Hitzler, T. Icard, K.-U. Kühnberger, L. C. Lamb, R. Miikkulainen, D. L. Silver, Neural-symbolic learning and reasoning: contributions and challenges, in: 2015 AAAI Spring Symposium Series, 2015.

[2] F. Yang, Z. Yang, W. W. Cohen, Differentiable learning of logical rules for knowledge base reasoning, Advances in Neural Information Processing Systems 30 (2017) 2316–2325.

[3] R. Evans, E. Grefenstette, Learning explanatory rules from noisy data, Journal of Artificial Intelligence Research 61 (2018) 1–64.

[4] A. Sadeghian, M. Armandpour, P. Ding, D. Z. Wang, Drum: End-to-end differentiable rule mining on knowledge graphs, Proc. of NIPS (2019).

[5] P. Minervini, M. Bošnjak, T. Rocktäschel, S. Riedel, E. Grefenstette, Differentiable reasoning on large knowledge bases and natural language, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, 2020, pp. 5182–5190.

[6] T. Rocktäschel, S. Riedel, End-to-end differentiable proving, in: NIPS, 2017, pp. 3788––3800.

[7] H. Gallaire, J. Minker, Logic and data bases, symposium on logic and data bases, centre d'études et de recherches de toulouse, 1977, Advances in Data Base Theory (1978).

[8] P. Minervini, S. Riedel, P. Stenetorp, E. Grefenstette, T. Rocktäschel, Learning reasoning strategies in end-to-end differentiable proving, in: ICML, volume 119 of *Proceedings of Machine Learning Research*, PMLR, 2020, pp. 6938–6949.

[9] J. Johnson, B. Hariharan, L. Van Der Maaten, L. Fei-Fei, C. Lawrence Zitnick, R. Girshick, Clevr: A diagnostic dataset for compositional language and elementary visual reasoning, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 2901–2910.

[10] D. Bahdanau, S. Murty, M. Noukhovitch, T. H. Nguyen, H. de Vries, A. Courville, Systematic generalization: what is required and can it be learned?, ICLR (2019).

[11] B. Lake, M. Baroni, Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks, in: International conference on machine learning, PMLR, 2018, pp. 2873–2882.

[12] K. Sinha, S. Sodhani, J. Dong, J. Pineau, W. L. Hamilton, Clutrr: A diagnostic benchmark for inductive reasoning from text, Empirical Methods of Natural Language Processing (EMNLP) (2019).

[13] M. E. Stickel, A prolog technology theorem prover: Implementation by an extended prolog compiler, Journal of Automated reasoning 4 (1988) 353–380.

[14] D. Merritt, Building expert systems in Prolog, Springer Science & Business Media, 2012.

[15] A. Lally, P. Fodor, Natural language processing with prolog in the ibm watson system, The Association for Logic Programming (ALP) Newsletter 9 (2011).

[16] J. Andreas, M. Rohrbach, T. Darrell, D. Klein, Learning to compose neural networks for question answering, NAACL (2016).

[17] N. Chomsky, Logical structures in language, American Documentation (pre-1986) 8 (1957) 284.

[18] N. Gontier, K. Sinha, S. Reddy, C. Pal, Measuring systematic generalization in neural proof generation with transformers, NeurIPS'20 (2020).

[19] S. Natarajan, T. Khot, K. Kersting, B. Gutmann, J. Shavlik, Gradient-based boosting for

statistical relational learning: The relational dependency network case, Machine Learning 86 (2012) 25–56.

[20] T. Trouillon, J. Welbl, S. Riedel, É. Gaussier, G. Bouchard, Complex embeddings for simple link prediction, in: International Conference on Machine Learning, PMLR, 2016, pp. 2071–2080.

[21] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine learning 8 (1992) 229–256.

[22] R. Das, S. Dhuliawala, M. Zaheer, L. Vilnis, I. Durugkar, A. Krishnamurthy, A. Smola, A. McCallum, Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning, Proceedings of the 7th International Conference on Learning Representations (2018).

[23] W. Xiong, T. Hoang, W. Y. Wang, Deeppath: A reinforcement learning method for knowledge graph reasoning, Conference on Empirical Methods in Natural Language Processing (EMNLP) (2017).

[24] N. Lao, T. Mitchell, W. Cohen, Random walk inference and learning in a large scale knowledge base, in: Proceedings of the 2011 conference on empirical methods in natural language processing, 2011, pp. 529–539.

[25] N. Asgharbeygi, N. Nejati, P. Langley, S. Arai, Guiding inference through relational reinforcement learning, in: International Conference on Inductive Logic Programming, Springer, 2005, pp. 20–37.

[26] M. Crouse, I. Abdelaziz, B. Makni, S. Whitehead, C. Cornelio, P. Kapanipathi, K. Srinivas, V. Thost, M. Witbrock, A. Fokoue, A deep reinforcement learning approach to first-order logic theorem proving, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, 2021, pp. 6279–6287.

[27] Z. Li, E. Wallace, S. Shen, K. Lin, K. Keutzer, D. Klein, J. E. Gonzalez, Train large, then compress: Rethinking model size for efficient training and inference of transformers, in: Proceedings of the 37th International Conference on Machine Learning (ICML 2020), 2020, pp. 5958–5968.

## A. Computational Complexity of CTPs

We provide a theoretical analysis of the time complexity of CTPs. We consider two base operations that we wish to count: the number of reformulator applications (data being passed through a neural network) and the number of sub-goals that need unifying with the knowledge base after the reasoning depth is reached (comparisons with all fact embeddings using a RBF kernel). The remainder of the operations in CTPs are either tied into one of these two, or take constant time. We count these operations with respect to the number of reformulators used $n$, the reasoning depth $d$, and the maximum number of atoms in the body of any reformulator used $m$. We find that the time complexity of CTPs as a whole is $\mathcal{O}((nm)^d)$.

Let $s_i$ be the number of sub-goals after the model has applied $i$ recursive expansions upon the goal. With each recursive step down, $n$ reformulators are applied to every sub-goal, with each reformulator generating $\mathcal{O}(m)$ new sub-goals to be proved. This means $\mathcal{O}(nm)$ new sub-goals for each existing sub-goal, so $s_{i+1} = \mathcal{O}(s_i \times nm)$. Noting that $s_0 = 1$, representing the query passed to the model, we see that the number of sub-goals after the test reasoning depth has been reached is: $s_d = s_0 \times \mathcal{O}((nm)^d) = \mathcal{O}((nm)^d)$. Furthermore, at depth $i$ with $s_i$ sub-goals, there are $ns_i$ reformulator applications. Thus, the total number of reformulator applications until the reasoning depth has been reached is:

$$r_d = ns_0 + ns_1 + ns_2 + \ldots + ns_{d-1} = \mathcal{O}(n) + \mathcal{O}((nm)^1 n) + \ldots + \mathcal{O}((nm)^{d-1} n) = \mathcal{O}((nm)^d d)$$

Then since $n > 1$ and $m > 1$ for any CTP model of non-trivial complexity, $r_d = \mathcal{O}((nm)^d)$. Thus, as $s_d = r_d$, we state that the time complexity for CTPs as a whole is $\mathcal{O}((nm)^d)$. The main issue with this time complexity is the raising to the power of $d$. For a fixed depth $d$, the time complexity is a $d$-degree polynomial in $n$ and $m$, which can also cause difficulties for a suitably large value of $d$. We have already provided motivation for using a test reasoning depth of 5 on CLUTRR, meaning that this would become degree 5 polynomial.

## B. Computational Issues of CTPs

To demonstrate the computational issues of CTPs in practice, we compute the average evaluation time across all CLUTRR datasets, noting that the results are not entirely stable due to them being evaluated and aggregated across a variety of machines. We show this with respect to the number of reformulators in Fig. 1, and with respect to the test reasoning depth in Fig. 2. For reference, the longest evaluation time was for a test reasoning depth of 5, with 5 reformulators, on the *1.10_test* dataset. It took 16.7 hours to evaluate.

## C. ACTP Speedup

**Wall-clock Speedup** In Fig. 3, we compare the evaluation times of ACTPs and baseline CTPs, with the baseline operating on 3 and 8 reformulators respectively. As expected, the overhead caused by the copying, masking, and other operations needed in an ACTP model led to it taking significantly longer to evaluate than CTP-3. However, the overhead was low enough for ACTP-8C3 to take less time to evaluate than CTP-8 across all datasets. The effect becomes

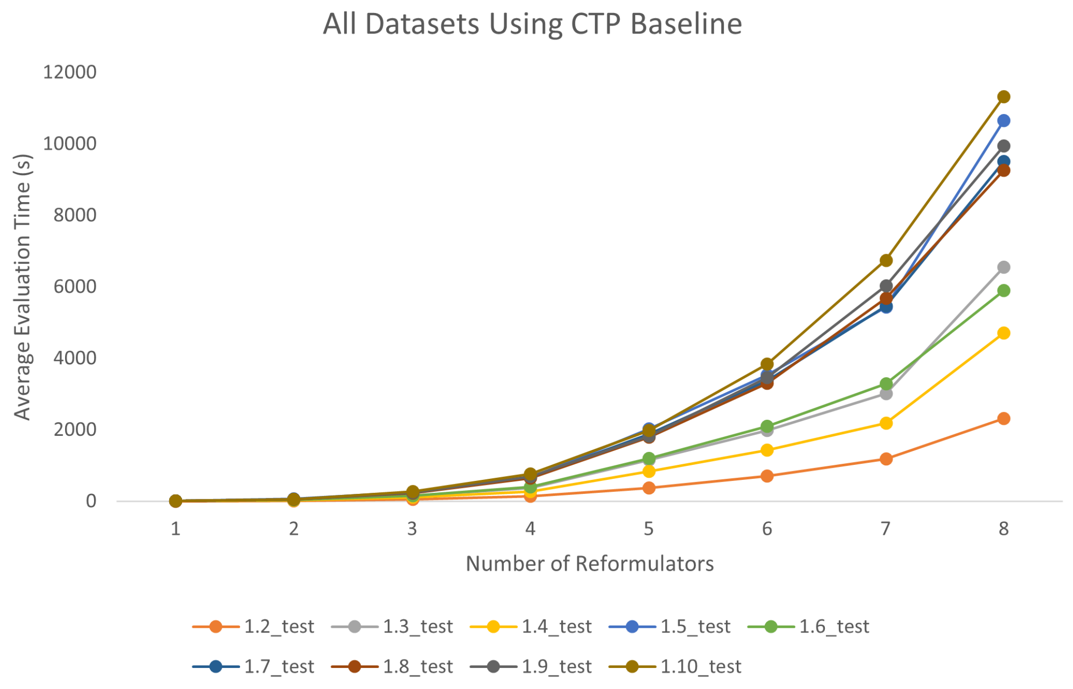

**Figure 1:** CTP average evaluation time across all datasets for a varying number of reformulators

more pronounced as the complexity of the dataset increases, since the number of facts in the knowledge base to unify the sub-goals with increases, which has a significant effect on the computational complexity of the model.

As the dataset complexity continues to increase. the overhead will become more negligible, leading the evaluation time of ACTP-*n*C*k* models to tend to those of CTP-*k* models. The increasing gap between the evaluation time of ACTP-8C3 and CTP-8 in Fig. 3 is a clear visual illustration of this trend. Datasets always taking longer to evaluate on than others is explained by the size of the datasets. For example, despite *1.6_test* being a more complex dataset than *1.5_test*, it only contains 104 tasks, compared to the 184 of *1.5_test*.

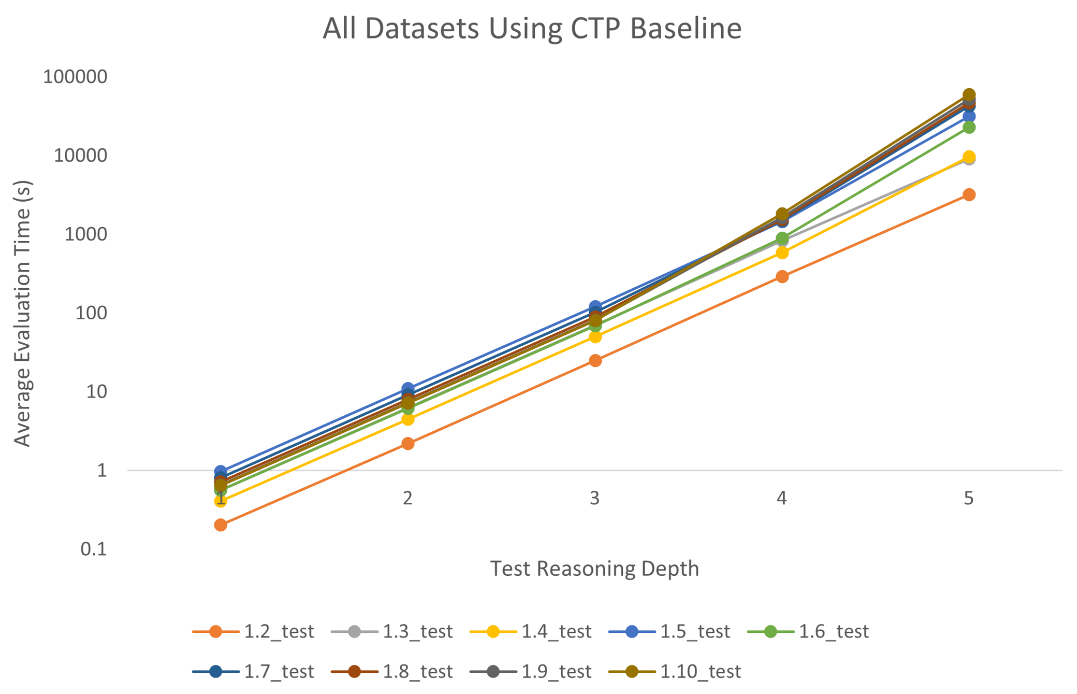

**Figure 2:** CTP average evaluation time across all datasets for a varying test reasoning depth, using a logarithmic scale

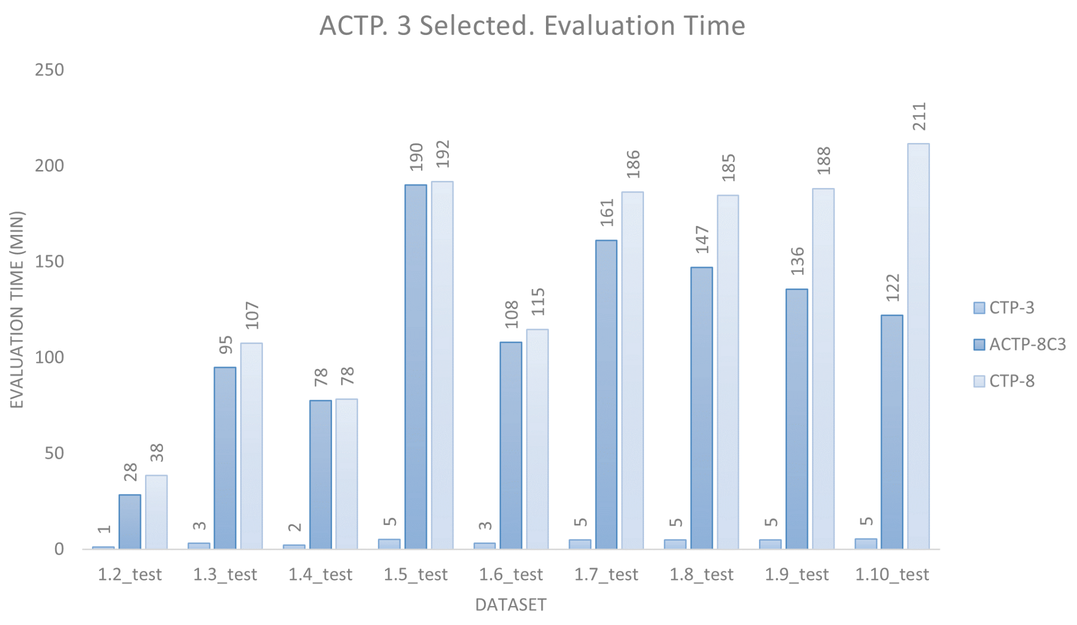

**Figure 3:** Comparison across all datasets of evaluation time when using CTPs with 3 reformulators, CTPs with 8 reformulators, and ACTPs with 8 choosing 3 reformulators

## D. Model Hyperparameters

In this appendix, for reproducibility, we provide the hyperparameters used for each of our evaluations.

### D.1. Fixed CTP Hyperparameters

| Name | Value |
|---:|:---|
| *batch-size* | 16 |
| *embedding-size* | 50 |
| *epochs* | 20 |
| *evaluate-every* | 100 |
| *init* | random |
| *init-size* | 1 |
| *k-max* | 5 |
| *learning-rate* | 0.1 |
| *max-depth* | 2 |
| *nb-rules* | 512 |
| *optimizer* | adagrad |
| *ref-init* | random |
| *reformulator* | attentive |
| *scoring-type* | concat |
| *slope* | 1 |
| *test* | *ALL TEST DATASETS* |
| *test-batch-size* | 1 |
| *tnorm* | min |
| *train* | *TRAIN DATASET* |

## D.2. Number of Reformulators

| Name | Values |
|---:|---|
| *hops* | 1-8 reformulators, each with 2 atoms in the body. For example, 4 reformulators is denoted by: "2 2 2 2" |
| *seed* | 1-30 |
| *test-max-depth* | 4 |

## D.3. Reasoning Depth

| Name | Values |
|---:|---|
| *hops* | 5 reformulators, each with 2 atoms in the body, denoted by: "2 2 2 2 2" |
| *seed* | 1-3 |
| *test-max-depth* | 1-5 |

## D.4. Reformulator Subsets

| Name | Values |
|---:|---|
| *hops* | 5, 8 reformulators, each with 2 atoms in the body, denoted by: "2 2 2 2 2" and "2 2 2 2 2 2 2 2" respectively |
| *seed* | 1-10 |
| *test-max-depth* | 4 |
| *subset* | Use reformulators with the following indices for evaluation: { [0 1 2], [2 3 4], [0 3 4], [1 2 4] } |

### D.5. ACTPs 5 Reformulators

| Name | Values |
|---:|---|
| *hops* | 5 reformulators, each with 2 atoms in the body, denoted by: “2 2 2 2 2” |
| *seed* | 1-30 |
| *test-max-depth* | 4 |
| *rl-actions-selected* | 2, 3 |
| *rl-epochs* | 3 |
| *rl-learning-rate* | 0.001, 0.01 |

### D.6. ACTPs 8 Reformulators

| Name | Values |
|---:|---|
| *hops* | 8 reformulators, each with 2 atoms in the body, denoted by: “2 2 2 2 2 2 2 2” |
| *seed* | 1-30 |
| *test-max-depth* | 4 |
| *rl-actions-selected* | 2, 3 |
| *rl-epochs* | 3 |
| *rl-learning-rate* | 0.005, 0.01 |

## E. Algorithm Pseudocode

Here follows the pseudocode for some of the algorithms used in this paper.

**Algorithm 1:** Backward chaining

In the code, $K$ is the knowledge base containing the rules and facts, sets are denoted with curly brackets, lists are denoted with square brackets, an underscore matches any argument, $G$ refers to a goal, $\hat{G}$ to a set of sub-goals, $S$ to a substitution set, $B$ to the body of a rule, and $H$ to the head of a rule. To check if a goal $G_1$ holds true, one needs to get the output of or$(G_1, [])$. If the output contains a substitution set, then the query is true, otherwise it will only contain the value FAIL and the query cannot be proven.

```
1: $\text{or}(G, S) = \{S' \mid S' \in \text{and}(B, \text{unify}(H, G, S)) \text{ for each } H \leftarrow B \in K\}$
2: $\text{and}(\_, \text{FAIL}) = \text{FAIL}$
3: $\text{and}([], S) = S$
4: $\text{and}(G : \hat{G}, S) = \{S'' \mid S'' \in \text{and}(\hat{G}, S'')\ \forall S' \in \text{or}(\text{substitute}(G, S), S)\}$
5: $\text{unify}(\_, \_, \text{FAIL}) = \text{FAIL}$
6: $\text{unify}([], [], S) = S$
7: $\text{unify}([], \_, \_) = \text{FAIL}$
8: $\text{unify}(\_, [], \_) = \text{FAIL}$
9:
   $$\text{unify}(h : H, g : G, S) = \text{unify}(H, G \left\{ \begin{array}{cc} S \cup \{h/g\} & \text{if } h \in V \\ S \cup \{g/h\} & \text{if } g \in V, h \notin V \\ S & \text{if } g = h \\ \text{FAIL} & \text{otherwise} \end{array} \right\})$$
10: $\text{substitute}([], \_) = []$
11:
   $$\text{substitute}(g : G, S) = \left\{ \begin{array}{cc} x & \text{if } g/x \in S \\ g & \text{otherwise} \end{array} \right\}$$
```

**Algorithm 2:** Neural backward chaining

The code is based on the summary by Minervini et al. [8]. $G$ is a goal, $d$ is the reasoning depth, $H$ is the head of a rule, $B$ is the body, $\mathcal{K}$ is a knowledge base containing rules and facts, $K$ is the RBF kernel, $S$ is a proof state, $S_\psi$ is a substitution set, $S_\rho$ is a proof score, and $V$ is a set of variables.

```
def or($G, d, S$)
    for $H \leftarrow B \in \mathcal{K}$                    /* Try use any rule in KB to prove */
    do
        for $S \in and(B, d, unify(H, G, S))$ do
            yield $S$
        end
    end
end

def and($B, d, S$)
    if $B = []$ or $d = 0$              /* Empty body or reasoning depth reached */
    then
        yield $S$
    else
        for $S' \in or(sub(B_0, S_\psi), d-1, S)$ /* Apply substitution to body, then try
         to prove each atom within */
        do
            for $S'' \in and(B_{1:}, d, S')$ do
                yield $S''$
            end
        end
    end
end

def unify($H, G, S = (S_\psi, S_\rho)$)
    $T_i = \left\{ \begin{array}{cc} \{H_i/G_i\} & \text{if } H_i \in V \\ \{G_i/H_i\} & \text{if } G_i \in V, H_i \notin V \\ \emptyset & \text{otherwise} \end{array} \right\}$
    $S'_\psi = S_\psi \bigcup T_i$                          /* Extend the substitution set */

    $S'_\rho = \min\{S_\rho\} \bigcup_{H_i, G_i \notin V} \{K(\theta_{H_i}, \theta_{G_i})\}$        /* Similarity value is the minimum
     similarity across all representations */

    return $S' = (S'_\psi, S'_\rho)$
end
```

**Algorithm 3:** REINFORCE

In this algorithm: $\theta$ represents the model parameters, $N$ is the number of episodes, $K$ is the number of episodes per batch, $T$ is the number of steps in an episode, $\gamma$ is the *discount factor* used to make further away rewards worth less, $\pi_\theta(s)_a$ gives the probability of action $a$ from the distribution produced by $\pi_\theta$ applied to $s$, and $\alpha$ is the learning rate.

```
n = 0
while $n < N$ do
    for $K$ episodes in batch do
        Generate episode $s_0, a_0, R_0, ..., s_T, a_T, r_T$ using the policy $\pi_\theta$ to output probability
          distributions which are then sampled from to get actions
        for $t \in 1, ..., T$ do
            Calculate discounted rewards from each state: $G_t := \sum_{i=t}^{T} \gamma^i R_i$
        end
        $n := n + 1$
    end
    Calculate policy loss for entire batch: $L(\theta) := -\frac{1}{K} \sum_{t=1}^{K} \ln(G_t \pi_\theta(s)_{a_t})$
    Update policy: $\theta := \theta + \alpha \nabla L(\theta)$
end
```

**Algorithm 4:** Model training procedure

```
def get_best_model(model, train_set, hyperparameter_grid, evaluation_set)
    best_model = None
    best_accuracy = -1
    for hyperparameter_set ∈ hyperparameter_grid do
        train_model(model, train_set)
        accuracy_value = accuracy(model, evaluation_set)
        if accuracy_value > best_accuracy then
            best_accuracy = accuracy_value
            best_model = model
        end
    end
    return best_model
end

def get_best_seeded_model(model, train_set, hyperparameter_grid, evaluation_set)
    pick 6 seeds $S = \{s_0, ..., s_5\}$ at random
    include seeds $S$ in hyperparameter_grid
    return get_best_model(model, train_set, hyperparameter_grid, evaluation_set)
end

def get_model_accuracy(model, train_set, evaluation_set, test_sets)
    initialize hyperparameter_grid
    total_accuracy = 0
    for 5 iterations do
        best_model = get_best_seeded_model(model, train_set, hyperparameter_grid,
          evaluation_set)
        test_accuracies = accuracy(best_model, test_sets)
        total_accuracy += test_accuracies
    end
    return total_accuracy / 5
end

get_model_accuracy(model, train_set, 1.3_test, test_sets)
get_model_accuracy(model, train_set, 1.9_test, test_sets)
```

**Algorithm 5:** ACTP selection module

$S$ is a batch of states, $A$ is a batch of actions, and $R$ a batch of rewards. $\pi$ denotes the policy estimator and $k$ is the number of reformulators the module ought to select. *sample_with_replacement*$(S, p, k)$ draws $k$ samples from $S$ with replacement, using the probability distribution $p$.

```
def get_actions(S)
    $P := e^{\pi(S)}$                    /* A batch of probability distributions */
    selected_reformulators = []
    reformulator_counts = []

    for each probability distribution $p \in P$ do
        indices = []
        if n_positive_entries(p) $< k$ then
            indices = indices of the positive entries in $p$
        else
            indices = sample_with_replacement($\{0, ..., n-1\}, p, k$)
        end

        selected_reformulators.append(indices)
        update reformulator_counts using indices
    end

    return selected_reformulators, reformulator_counts
end

def apply_reward(S, A, R)
    $L$ := get_loss($S, A, R$)
    optimizer.apply_loss($L$, retain_graph = True)
end
```