Why Reinforcement Learning Struggles with Expression Simplification: A Reward Analysis

Oleksii Shuhailo[1] **Karel Chvalovský**[2] **Tomáš Pevný**[1]
[1] Artificial Intelligence Center, Czech Technical University. Czech Republic
[2] Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University. Czech Republic

Abstract

Expression simplification is a central task in both mathematics and computer science, with applications ranging from algebraic reasoning to compiler optimization. The successes of reinforcement learning (RL) in various domains have spurred attempts to apply it to symbolic reasoning tasks. However, RL-based methods frequently underperform relative to specialized solutions. This paper theoretically shows that one source of failure might be a poorly designed reward function.

1 Introduction

Expression simplification is a process of reducing the complexity of an expression or equation by applying various mathematical rules and techniques to obtain a simpler, more manageable form. This concept is crucial not only in formal math but also in computer science, as various optimizations of modern compilers can be viewed as expression simplification.

Reinforcement Learning (RL) has achieved remarkable success across a wide range of domains, from games [20] to robotics [24] and combinatorial optimization [4]. These breakthroughs have inspired growing interest in applying RL to symbolic reasoning tasks such as automated theorem proving [9, 25, 1, 14, 21], term rewriting, and expression simplification. In these tasks, the search must navigate large, structured search spaces to derive correct solutions. At first glance, these domains seem to be a good fit for RL, as they are sequential decision-making processes and thus align well with the traditional structure of an RL environment. However, the state-of-the-art methods based on RL still fall short compared to specialized solutions, such as computer algebra systems [8, 15] or systems leveraging e-graphs [6, 23], and to methods directly maximizing efficiency of the tree-search [17, 5, 7].

This work theoretically analyzes one possible source of failure: the reward function. We show that for rewards depending only on the last state, e.g., size of the expression, a corresponding heuristic derived from the optimal value function is constant for all states on trajectories to the optimum. Experiments confirm our analyses but show that RL can perform better than random search. We explain that this is consistent with the theory.

2 Problem formulation

The expression simplification task aims to find the most reduced form of an expression, where the "reduction" may correspond to the smallest syntactic size, the lowest computational complexity, or the fastest variant. This form is obtained by applying a sequence of algebraic rules (at given positions) starting from the input expression.

It is common [4, 19, 26, 16], to formulate this problem as a Markov Decision Process (MDP), defined by a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, t, r, \gamma \rangle$, where each state $s \in \mathcal{S}$ corresponds to an expression, action

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

$a \in \mathcal{A}$ corresponds to an algebraic rule to be applied at a particular position within the expression, transition function t corresponds to the application of the rule on an expression, r is the reward, and $\gamma \in [0, 1]$ is the discount factor. This formulation has several important properties. First, the set of actions available at each state (an expression) is not static; it depends on the expression (naturally, larger expressions have a larger set of available actions). This work adopted the set of algebraic rules from [11]. Second, the transition function t is deterministic. Third, the reward function r is deterministic, and it can be chosen freely as long as it aligns with the objective. For example [4] proposed $r(s, a) = |s| - |t(s, a)|$, where $|\cdot|$ is the length of the expression (or some other measure of quality). Its important property is that for $\gamma = 1$ and a given sequence of states $(s_0, s_1, \ldots, s_n)$ and actions $(a_0, a_1, \ldots, a_{n-1})$ such that $(\forall i)(s_{i+1} = t(s_i, a_i)$ it holds that $\sum_{i=0}^{n-1} r(s_i, a_i) = |s_0| - |s_n|$. This means that the reward depends on the final state, which is well aligned with the original objective, but it does not unnecessarily penalize long paths. This penalization can be achieved by setting $\gamma < 1$, but this might negatively affect the main objective, as discussed in Section 4. By abuse of notation, we also define the reward $r(s_i, s_{i+i}) = |s_i| - |s_{i+1}|$, where $s_{i+1} = t(s_i, a_i)$ for some action a_i.

3 Background

This section reviews two principally different approaches to solving the aforementioned MDP. The first is based on methods of (deep) reinforcement learning [4] with the intuition that the optimal value function would perform well in tree-search as well, while the second approach learns a heuristic value maximizing [17, 5, 7] efficiency of search in symbolic planning.

3.1 Learning value function

Reinforcement Learning (RL) has extensive previous work, we review the key concepts important for this paper, focusing on Value learning as our approach to solving the previously described MDP. While many methods used Q-learning or policy learning, we frame the problem in terms of Value learning for several reasons: (i) it simplifies the theoretical analysis below; (ii) in deterministic MDPs Value learning is closely related to Q-learning, as the Q-function can be expressed as $Q(s, a) = r(s, a) + \gamma V(t(s, a))$, where $V(s)$ is the value transition function; (iii) policy learning can be viewed as regularized Q-learning [18]; (iv) and the value function in RL and heuristic function in planning can be realized by the same model, which makes experimental comparison fairer.

The objective of Value learning in RL is to compute the optimal value function $V^\star(s)$ that maximizes the expected cumulative discounted reward [22]. The optimal value function $V^\star(s)$ satisfies the Bellman optimality equation [3]

$$V^\star(s) = \max_a \left(r(s, a) + \gamma V^\star(t(s, a))\right). \tag{1}$$

A parameterized value function $V_\theta(s)$ is usually found by minimizing a loss function encouraging $V_\theta(s)$ to satisfy Eq. (1). The value of learning loss is defined as

$$\ell_{\mathrm{VL}}(\theta) = \sum_{s \in \mathcal{S}} \left[\left(V_\theta(s) - y(s)\right)^2\right], \tag{2}$$

where target values $y(s) = \max_a(r(s, a) + \gamma V_{\theta'}(t(s, a)))$ with θ' being parameters of the value function delayed by time (often referred to as the target network parameters in deep RL). This regression-based formulation ensures that the value function produces numerically accurate estimates of expected returns, aligning the predicted value of each state with the Bellman optimality condition.

If the sum in Eq. (2) is over all states $\mathcal{S}$, the method corresponds to value iteration. This is usually in practice infeasible due to the size of $\mathcal{S}$; therefore, it is approximated by samples from search rollouts, a technique used in Real-Time Dynamic Programming (RTDP) [2]. The value function V_θ is used to derive an optimal policy by selecting actions that maximize the right-hand side of Eq. (1), i.e., $\pi(s) = \arg\max_a \left(r(s, a) + \gamma V_\theta(t(s, a))\right)$. For the purpose of planning, the heuristic function of state s_i is

$$h(s_i) = -\left(\sum_{j=0}^{i-1} r(s_j, s_{j+1}) + V_\theta(s_i)\right), \tag{3}$$

where $(s_0, s_1, \ldots, s_i)$ is the path from root s_0 to the node s_i. The term $\sum_{j=0}^{i-1} r(s_j, s_{j+1})$ is required to make values comparable (see Appendix A for details).

3.2 L^* loss

Contrary to the above Value learning, L^* loss (concurrently proposed in [5, 17, 7]) directly optimizes the efficiency of a heuristic tree-search, which is optimal, if in every iteration the states on the optimal trajectory have smaller heuristic values than all other states in the open list. The method therefore optimizes **relative order** of heuristic values, which determines the efficiency of the tree-search.

Assuming an (optimal) trajectory $\pi = (s_0, s_1, s_2, \ldots, s_l)$ and a heuristic function $h_\theta(s)$ with parameters $\theta \in \mathbf{\Theta}$, the L^* is defined as

$$\mathrm{L}^*(h, \pi) = \sum_{s_i \in \mathcal{S}^\pi} \sum_{s_j \in \mathcal{O}_i \setminus \mathcal{S}^{\pi:i}} \ell(h_\theta(s_i) - h_\theta(s_j)), \tag{4}$$

where $\mathcal{O}_i$ is an Open list in the i^{th} iteration of the forward search expanding only states on the trajectory π; $\mathcal{S}^{\pi:i}$ are states on the sub-trajectory ending by state s_i, therefore $\mathcal{S}^{\pi:i} = \{s_0, s_1, \ldots, s_i\}$, and ℓ is a suitable differentiable loss function, in case of this paper $\ell(x) = \log(1 + \exp(x))$.

The L^* loss defined by (4) solves the ranking problem, whereas Value learning loss (2) solves the regression problem. As discussed in [5], the ranking problem is more sample efficient than the regression problem. But more importantly, L^* does not prescribe the exact values, giving heuristic functions more freedom to realize goals. While the above formulation assumes knowing the optimal path in advance, the loss function can be used in Bootstrapped methods by extracting the best solution from search rollouts.

4 Theoretical analysis of Value learning

We now present the main theoretical result of the work.

Theorem 1. *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, t, r, \gamma \rangle$ be such that the reward $r(s, a) = |s| - |t(s, a)|$ and $\gamma = 1$. Let s be a node, whose actions a', a'' create children s', s'' that can be reduced to expressions of the same minimal size. Then for an optimal value function V satisfying the Bellman equation* (1) *it holds that $r(s, a') + V(s') = r(s, a'') + V(s'')$.*

The proof is in Appendix B. The theorem states that if there are two trajectories to the solution, the heuristic derived from an optimal value function cannot determine which is better, even if they have different lengths, and hence provides no information to the search. The number of such paths can be very large, since many rewriting systems contain associativity and commutativity rules, which rapidly produce numerous essentially equivalent states, and rewriting steps can usually be permuted to varying degrees, further complicating the situation.

Corollary 1. *Let's assume conditions of Theorem 1 hold, and furthermore let's assume that from every state exists a trajectory to every other state through some sequence of actions. Then for every state $s \in \mathcal{S}$ and for all actions a' and a'' applicable in s it holds that $r(s, a') + V(t(s, a')) = r(s, a'') + V(t(s, a''))$.*

According to the corollary above, for some rewriting systems, the heuristic function derived from the optimal value function is uninformative.

Theorem 2. *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, t, r, \gamma \rangle$ be such that the reward $r(s, a) = |s| - |t(s, a)|$ and $\gamma = 1$. Let $(s_0, s_1, s_2, \ldots, s_n)$ be a trajectory reducing node s_0 to a node of minimal size s_n, and $(a_0, \ldots, a_{n-1})$ be the corresponding sequence of actions. Then for the heuristic function* (3) *derived from optimal value function holds $\sum_{i=0}^{n-1} r(s_i, a_i) + V(s_n) = |s_0| - |s_n|$ and hence is constant.*

The theorem states that the heuristic function derived from the optimal value function is not correlated with "closeness" to the goal, further complicating the search.

The above theory was derived for $\gamma = 1$. Setting $\gamma < 1$ might possibly solve the problem, as it will penalize long solutions. But selecting the right γ is difficult, and it might not even exist, because some expression needs to be first made longer before they can be reduced to their minimal form. If $\gamma \ll 1$, then the reward from the reduction phase might be diminished, and a longer expression reachable in a few steps will be preferred. Clearly, for large and complicated expressions, there might be several phases where the size needs to increase, which requires γ to be close to one.

Table 1: The average reduction of expression size of different variants for heuristic function. Best / worst results are blue / red. Standard deviations are estimated from three repetitions of experiment.

	Target	Training set					Testing set				
	Value	γ=0.8	γ=0.9	γ=0.95	γ=0.99	γ=1	γ=0.8	γ=0.9	γ=0.95	γ=0.99	γ=1
VL Tree	$\lvert\cdot\rvert$	6.55 ± 0.69	6.56 ± 0.43	6.48 ± 0.12	6.52 ± 0.37	6.45 ± 0.51	5.25 ± 0.93	6.32 ± 2.04	5.61 ± 0.92	6.46 ± 0.99	7.17 ± 1.25
VL DG	$\lvert\cdot\rvert$	4.41 ± 0.47	4.13 ± 1.02	5.81 ± 0.47	4.91 ± 0.28	4.34 ± 0.95	5.33 ± 1.24	5.28 ± 0.56	5.54 ± 0.33	4.56 ± 0.44	3.86 ± 0.58
RTDP Tree	V	6.48 ± 0.54	6.37 ± 0.52	6.14 ± 0.21	6.28 ± 0.5	6.2 ± 0.61	6.69 ± 1.17	6.5 ± 0.8	6.09 ± 0.51	6.05 ± 0.44	6.23 ± 0.37
RTDP DG	V	6.11 ± 0.69	6.29 ± 0.82	5.64 ± 0.21	6.05 ± 0.54	6.14 ± 0.86	6.92 ± 0.74	6.83 ± 2.41	6.19 ± 1.65	5.76 ± 0.61	5.95 ± 0.89
L^*	—			10.44 ± 0.99					9.92 ± 0.09		
Expression size	—			3.591					3.786		

5 Experiments

The experiments were designed to rigorously compare heuristic optimization using L^* and a heuristic derived by Equation (3) from the Value function in Greedy Best-First Search (GBFS). The experiments were performed on a subset of a dataset from [4] and the rewrite rules from [14]. The optimization of L^* was implemented as in [5], but optimizing the Value function is not straightforward. First, it is not clear which transitions from the search graph should be used to create training samples. It can be pruned to a Tree by keeping only the shortest path to the root (usually done in GBFS) or maintained as a directed graph (DG). Second, it is not clear if the values of leaves should be derived from the true expression size (denoted as VL for Value learning) or from the value function (denoted as RTDP). We have therefore tested all variants. Furthermore, since the discount factor γ can significantly influence the quality of the learned heuristic, we have tested values $\{0.8, 0.9, 0.95, 0.99, 1.0\}$. Due to lack of space, the experimental settings are detailed in Appendix C.

The reduction in expression size was measured as a difference between the length of the initial expression and the shortest expression found during the search. All compared methods are shown in Table 1. Consistent with the above theory, L^* method directly maximizing efficiency of the search is achieving the best score. However, contrary to the theory, almost all variants of Value learning and RTDP deliver performance better than greedy minimization of expression size (denoted as *Expression size* in Table 1), albeit worse than L^*. This phenomenon, seemingly contradicting the theory, is caused by biases (discussed below) introduced by the creation of training samples from the search tree. Notice, though, that the Value learning with directed graph (VL DG), where the bias consists only of the limited exploration, performs the worst. For all methods, higher γ leads to better results, which experimentally validates our analysis in Section 4.

One source of bias is the limits of the number of expanded nodes and depth during GBFS, which means that only a small subset of trajectories can be observed, which causes states not in them to receive lower values (they can possibly reduce the same results, but using longer chains of operations, which fall outside limits). The next source of bias comes from pruning of the search graph to the search tree. The tree contains only one trajectory (shortest) from the root to the smallest expression, and therefore conditions of Theorem 1. This pruning introduces a discrepancy between the real transition and the transition system used to compute new target values according to Bellman Equation (1).

6 Conclusion

This work has analyzed the Value learning method of reinforcement learning and L^* loss from planning in the domain of symbolic expression simplification. We have shown that Value learning, optimizing the size of the expression, which seems like a natural goal, is destined to fail, because the heuristic function derived from the optimal value function can be non-informative for the search. The experimental comparison has further clarified that the observed success of Value learning methods is caused by biases introduced by experimental settings. Specifically, by i) limited exploration of the search caused by limits on depth and number of expanded nodes, and ii) by search tree keeping only one (shortest) trajectory from the best solution to the root. The L^* loss suffers from the first problem, but completely sidesteps the latter, which, according to our experiments, seems to be more severe.

The theoretical analysis was done in the setting of a Greedy Best-First Search. While we assume the results to hold for currently popular Monte-Carlo tree search and policy gradient methods, it is something we would like to investigate in the future. A perpendicular line of interest is the design of a better reward, which does not suffer from the problems reported above.

Acknowledgements

The authors acknowledge the support of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics". KC's work was supported by the Czech Science Foundation project 24-12759S.

References

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

[2] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81-138, 1995.

[3] Richard Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[4] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. *Advances in neural information processing systems*, 32, 2019.

[5] Leah Chrestien, Stefan Edelkamp, Antonin Komenda, and Tomas Pevny. Optimize planning heuristics to rank, not to estimate cost-to-goal. *Advances in Neural Information Processing Systems*, 36:25508–25527, 2023.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[7] Mingyu Hao, Felipe Trevizan, Sylvie Thiébaux, Parick Ferber, and Jörg Hoffmann. Learned pairwise rankings for greedy best-first search. In *Proc. ICAPS Workshop on Reliable Data-Driven Planning and Scheduling*, 2024.

[8] Wolfram Research, Inc. Mathematica, Version 14.3. Champaign, IL, 2025.

[9] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. Caviar: an e-graph based trs for automatic code optimization. *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, page 54–64, 2022.

[12] Šimon Mandlík, Tomáš Pevný, Václav Šmídl, and Lukáš Bajer. Malicious internet entity detection using local graph inference. *IEEE Transactions on Information Forensics and Security*, 19:3554–3566, 2024.

[13] Šimon Mandlík, Matěj Račinský, Viliam Lisý, and Tomáš Pevný. Jsongrinder.jl: automated differentiable neural architecture for embedding arbitrary json data. *Journal of Machine Learning Research*, 23:1–5, 2022.

[14] Jack McKeown and Geoff Sutcliffe. Reinforcement learning for guiding the E theorem prover. In Michael Franklin and Soon Ae Chun, editors, *Proceedings of the Thirty-Sixth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2023, Clearwater Beach, FL, USA, May 14-17, 2023*. Florida Online Journals, 2023.

[15] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[16] Flavio Petruzzellis, Alberto Testolin, and Alessandro Sperduti. A neural rewriting system to solve algorithmic problems. 2024.

[17] Jelle Piepenbrock, Tom Heskes, Mikoláš Janota, and Josef Urban. Guiding an automated theorem prover with neural rewriting. In *International Joint Conference on Automated Reasoning*, pages 597–617. Springer International Publishing Cham, 2022.

[18] John Schulman, Xi Chen, and Pieter Abbeel. Equivalence between policy gradients and soft q-learning. *arXiv preprint arXiv:1704.06440*, 2017.

[19] Ali Shehper, Anibal M. Medina-Mardones, Lucas Fagan, Bartłomiej Lewandowski, Angus Gruen, Yang Qiu, Piotr Kucharski, Zhenghan Wang, and Sergei Gukov. What makes math problems hard for reinforcement learning: a case study. 2025.

[20] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[21] Martin Suda. Efficient neural clause-selection reinforcement. 2025.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[23] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[24] Andy Zeng, Shuran Song, Daniel Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *Science Robotics*, 4(28):eaau8910, 2019.

[25] Zsolt Zombori, Adrián Csiszárik, Henryk Michalewski, Cezary Kaliszyk, and Josef Urban. Towards finding longer proofs. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6-9, 2021, Proceedings*, volume 12842 of *Lecture Notes in Computer Science*, pages 167–186. Springer, 2021.

[26] Zsolt Zombori, Josef Urban, and Chad E. Brown. Prolog technology reinforcement learning prover. *Automated Reasoning*, pages 489–507, 2020.

A Why Value function cannot be used as a heuristic

In Section 3.1, we posit that the heuristic function of state s_i in search cannot be negative of the (optimal) Value function maximizing the Bellman equation, but it should be equal to

$$h(s) = -\left(\sum_{j=0}^{i-1} r(s_j, s_{j+1}) + V_\theta(s_i)\right), \tag{5}$$

where $(s_0, s_1, \ldots, s_j)$ is the path from root s_0 to the node s_i, V_θ is the (optimal) value function, and $r(s_j, s_{j+1})$ is the reward equal to $r(s_j, s_{j+1}) = |s_j| - |s_j + 1|$. The term $\sum_{j=0}^{i-1} r(s_j, s_{j+1})$ is required to make values comparable, otherwise, states that were first made large would be preferred, as they have higher potential to be reduced.

Let's illustrate the problem with an example. Let's assume a heuristic $\bar{h}(s) = -V_\theta(s)$ and let's consider an expression $1 \leq 2$. This expression can be trivially reduced to 1 with the value zero, $V_\theta(1) = 0$, as it cannot be reduced further. The same expression can be also rewritten to $!(1 > 2)$ of value three, $V_\theta(!(1 > 2)) = 3$, because it can be simplified to 1 as well. The heuristic function $\bar{h}$ equal to the negative of the Value function of these states would be, therefore equal to $\bar{h}(1) = 0$ and $\bar{h}(!(1 > 2)) = -3$, and Greedy Best-First Search would therefore prefer to expand the latter expression. The problem can be even worse if the set of rewrite rules contains a rule $x \to x + 0$, which can be applied indefinitely.

Including the reward $\sum_{j=0}^{i-1} r(s_j, s_{j+1})$ into the heuristic function (5) fixes the problem. But as discussed in the main text, it makes the heuristic values of both states (1 and $!(1 > 2)$) the same, equal to two.

B Proof of theorem

Theorem 1. *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, t, r, \gamma \rangle$ be such that the reward $r(s, a) = |s| - |t(s, a)|$ and $\gamma = 1$. Let s be a node, whose actions a', a'' create children s', s'' that can be reduced to expressions of the same minimal size. Then for an optimal value function V satisfying the Bellman equation* (1) *it holds that $r(s, a') + V(s') = r(s, a'') + V(s'')$.*

Proof. We proceed by contradiction. Suppose there exists a state $s \in \mathcal{S}$ with two actions $a', a'' \in \mathcal{A}(s)$ producing children $s' = t(s, a')$ and $s'' = t(s, a'')$ such that $r(s, a') + V(s') \neq r(s, a'') + V(s'')$, where V is an optimal value function. By assumption, both s' and s'' can be reduced (through some sequences of actions) to expressions of the same minimal size, call these states s'_g and s''_g. We know $|s'_g| = |s''_g|$. For s', let $(a'_1, \ldots, a'_n)$ be a sequence reducing $s' = s'_1$ to $s'_{n+1} = s'_g$. Then the cumulative reward is $\sum_{i=1}^{n} r(s'_i, a'_i) = |s'| - |s'_g|$. Thus, $r(s, a') + V(s') = |s| - |s'| + |s'| - |s'_g| = |s| - |s'_g|$. Similarly, for s'' we obtain $r(s, a'') + V(s'') = |s| - |s''| + |s''| - |s''_g| = |s| - |s''_g|$. Hence $r(s, a') + V(s') = r(s, a'') + V(s'')$ as $|s'_g| = |s''_g|$, contradicting our assumption. □

The above proof states that if there are two trajectories to the solution, the heuristic derived from an optimal value function cannot decide which one is better, despite their different length, hence providing no information to the search. The number of such paths can be very large, since many rewriting systems contain associativity and commutativity rules, which rapidly produce numerous essentially equivalent states, and rewriting steps can usually be permuted to varying degrees, further complicating the situation.

C Experimental settings

The experiments aim to verify deviations of the above theory from reality. We compare Greedy Best-First Search with a heuristic function optimized by variants of RTDP and by L^* loss. An emphasis was put on making the comparison fair, such that the difference is only in the loss functions used to optimize the heuristic function. The heuristic function is optimized by following the protocol

1. Randomly initialize weights of the model.

2. Run tree-search for every expression in the training set.
3. Extract training data from search trees obtained in step 2.
4. Train the model for 10 epochs on training data from step 3 and return to step 2.

Steps 1–4 are repeated 20 times.

The parametrized model implementing the heuristic function is realized by a tree-structured neural network and is described in the Appendix in Section C.5.

The tree search in step 2 is a Greedy Best-First Search with at most 1000 expansions and a maximum depth 100. These parameters were chosen with respect to our limited computation resources. ϵ-greedy strategy with exponential decay, reducing ϵ by a factor of 0.8 after each epoch, with a minimum threshold of 0.05, was used to encourage exploration. The optimizer in step 4 was Adam [10] with default settings.

C.1 L^* loss

The L^* is implemented and used exactly as described in the original publication [5]. Since in this paper, the trajectory to the solution for creating the training data is not available, from every search tree obtained by GBFS with the current version of the heuristic, the trajectory to the smallest expression was used instead. This makes the protocol equal to that of Value learning and RTDP. We have maintained the best solution found across epochs, such that we would not degrade the training data.

C.2 Value learning

The Value learning uses Greedy Best-First Search for rollouts. We have investigated three different variants of how to treat multiple trajectories from the root to the same node. The motivation for exploring these variants is that multiple trajectories affect the *transition system observed during training.*

The first variant, called **Tree** is the vanilla tree-search algorithm, which maintains only one shortest found path from the root to each node. Therefore, the transition system extracted from the search tree differs from the real transition system. The second variant, called **Directed Graph (DG)** does not delete any edges. Therefore, the resulting search structure is no longer a tree but a graph. This captures the full complexity of the transition system, but the value backup might need to be modified to handle loops.

We have compared two methods to compute target values $y(s)$ used for training the neural network (see Equation (2). The first, inspired by Real Time Dynamic Programming (abbreviated **RTDP**),

$$y(s) = \max_a(r(s,a) + \gamma V_\theta(t(s,a))), \tag{6}$$

computes the target value of a node using estimates of that of children using the current value function V_θ. This works with all three variants of search structures without any modification.

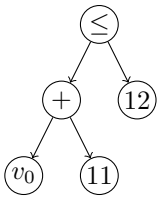
The second, inspired by **Value iteration** (abbreviated as **VL**) uses true reward $y(s_g) = |s_0| - |s_g|$ for leaf nodes (nodes without children), where s_0 is the initial expression and s_g is a leaf expression. The target values of inner-nodes are computed recursively according to

$$y(s) = \max_a(r(s,a) + \gamma y(t(s,a))). \tag{7}$$

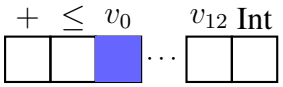
While this is possible for the Tree variant of search structure, it is impossible in DG due to the presence of cycles. In this case, we have replaced the not-yet-computed values with the estimate V_θ.

C.3 Settings of the search during tests

The evaluation uses the same tree search pipeline as in training, but reduces the maximum number of expansions from 1000 to 100. This constraint allows us to better verify if the trained networks have genuinely learned to guide the search, or they rely on brute-force trying a large number of expansions to discover a solution. The exploration is disabled, i.e. $\epsilon = 0$ in ϵ-greedy. The performance was always measured on the training, validation, and testing datasets to be able to estimate the overfitting and generalization.



(a) Expression tree: $v_0 + 11 \leq 12$.



(b) One-hot encoding of symbols.

Figure 1: Illustration of expression representation. (a) Expression tree structure. (b) Corresponding one-hot encoding of vocabulary symbols.

C.4 Discount factor

In reinforcement learning, the discount factor γ determines how much emphasis should be put on immediate and future rewards. Ranging between 0 and 1, a γ close to 0 prioritizes immediate rewards, while a value near 1 encourages long-term planning by emphasizing future returns. Since the setting of a discount factor is important for the quality of the learned value function, we have tested several values $\{0.8, 0.9, 0.95, 0.99, 1.0\}$, though we believe that the domain requires values closer to one, since the greedy decrease of the expression size is certainly not optimal.

C.5 Representation of expression in neural networks

Neural architecture. The architecture of the neural network realizing value / heuristic function is inspired by [13, 12], it consists of four components:

$$m = (f_{\text{head}}, f_{\text{args}}, \psi, h),$$

where f_{head} embeds function symbols, f_{args} embeds argument tuples with positional information, ψ aggregates argument embeddings, and h is the final heuristic network. The embedding dimension is fixed to $d = 64$. Below, all components are described in more detail.

Leaf encoding. A leaf corresponds to a symbol $s \in \mathcal{V}$, optionally annotated with a scalar value $v \in \mathbb{R}$. Its one-hot encoding $x \in \mathbb{R}^{|\mathcal{V}|}$ is defined by

$$x_i = \begin{cases} v & \text{if } i = \text{index}(s), \\ 0 & \text{otherwise.} \end{cases}$$

A leaf embedding is computed as

$$z_{\text{leaf}} = f_{\text{head}}(x) \parallel f_{\text{args}}(\mathbf{0}),$$

where $\mathbf{0} \in \mathbb{R}^d$ is a dummy argument placeholder and $\parallel$ denotes concatenation.

Internal node encoding. An internal node with head symbol s and $k \in \{1, 2\}$ arguments already embedded as $u_1, \ldots, u_k \in \mathbb{R}^d$ is encoded as follows:

$$z_{\text{args}} = f_{\text{args}}\big([u_1 \parallel \cdots \parallel u_k] \parallel p_k\big),$$

where $p_k \in \mathbb{R}^2$ is a positional encoding distinguishing unary ($k = 1$) vs. binary ($k = 2$) operators. The head symbol is embedded as

$$z_{\text{head}} = f_{\text{head}}(x_s),$$

with x_s the one-hot encoding of s. The node representation is then

$$z_{\text{node}} = f_{\text{head}}\big(z_{\text{head}} \parallel z_{\text{args}}\big).$$

Aggregation. The function $\psi : (\mathbb{R}^d)^k \to \mathbb{R}^d$ aggregates argument embeddings. In our implementation, ψ is realized as $\sum$.

Heuristic output. The final expression embedding $z_{\text{expr}} \in \mathbb{R}^d$ is mapped to a scalar heuristic value via

$$m(\text{expr}) = h(z_{\text{expr}}),$$

where $h : \mathbb{R}^d \to \mathbb{R}$ is a two-layer MLP.

Architecture choices. In the experiments used in this paper, we have used the following settings, which we have found sufficient in our preliminary experiments.

- f_{head}: a product model consisting of two parts: (i) a one-layer feed-forward network (`Dense(`$|\mathcal{V}|$`, 64, GELU)`) for symbols, (ii) a one-layer network (`Dense(64, 64, GELU)`) for arguments, combined and projected by another dense layer `Dense(128, 64, GELU)`.
- f_{args}: a product model with (i) `Dense(64, 64, GELU)` for argument tuples, (ii) `Dense(2, 64)` for positional encodings, combined and projected by `Dense(128, 64, GELU)`.
- Aggregation: $\psi = \sum$.
- Heuristic head h: a two-layer MLP

$$h(z) = \text{Dense}(64, 64, \text{ReLU}) \circ \text{Dense}(64, 1)(z).$$

C.6 Dataset

The dataset used in our experiments was introduced in [4], where it was generated using the Halide system pipeline to produce a large collection of examples. It is based on a fixed vocabulary of symbols. For our setup, both the training and testing datasets consist of 1000 expressions each.

D Rules used

For the sake of completeness, we list the rewriting rules used in the system in Table 2.

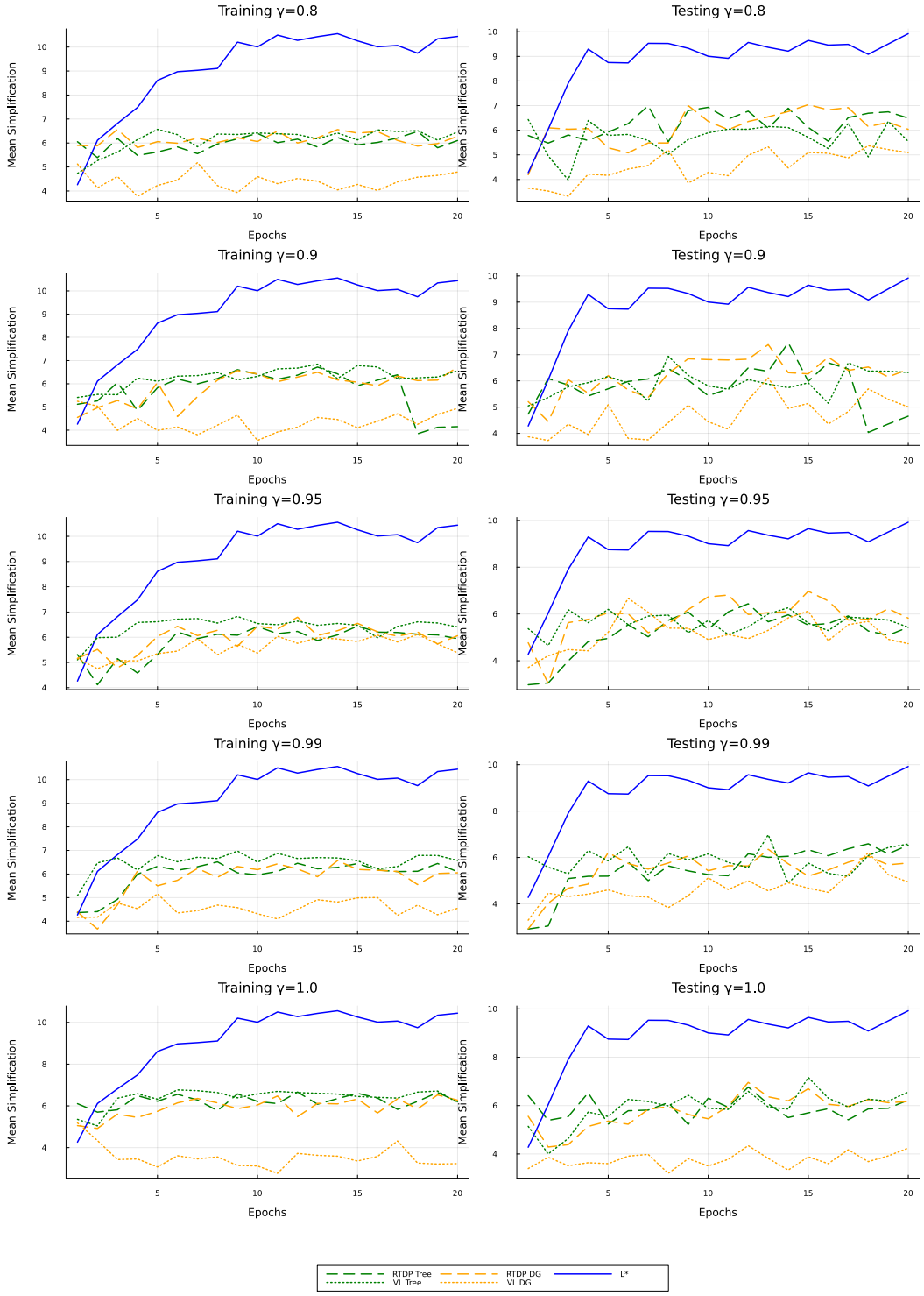


Figure 2: Comparison of training and testing results across different methods.

Table 2: Rewriting rules

Pattern	Rewrite / Condition
a::Number + b::Number	a + b
a::Number - b::Number	a - b
a::Number * b::Number	a * b
a + b	b + a
a + (b + c)	(a + b) + c
(a + b) + c	a + (b + c)
(a + b) - c	a + (b - c)
a + (b - c)	(a + b) - c
(a - b) - c	a - (b + c)
a - (b + c)	(a - b) - c
(a - b) + c	a - (b - c)
a - (b - c)	(a - b) + c
a + 0	a
a * (b + c)	a*b + a*c
a*b + a*c	a * (b + c)
(a / b) + c	(a + (c * b)) / b
(a + (c * b)) / b	(a / b) + c
x / 2 + x % 2	(x + 1) / 2
x * a + y * b	((a / b) * x + y) * b
a && b	b && a
a && (b && c)	(a && b) && c
1 && a	a
a && 1	a
a && a	1
a && !a	0
!a && a	0
(a == c::Number) && (a == x::Number)	c != x ? 0 : :($a == $x)
a::Number && b::Number	a != b ? 0 : 1
(a < y) && (a < b)	a < min(y, b)
a < min(y, b)	(a < y) && (a < b)
(a <= y) && (a <= b)	a <= min(y, b)
a <= min(y, b)	(a <= y) && (a <= b)
(a > y) && (a > b)	x > max(y, b)
a > max(y, b)	(a > y) && (a > b)
(a >= y) && (a >= b)	a >= max(y, b)
a >= max(y, b)	(a >= y) && (a >= b)
(a::Number > b) && (c::Number < b)	a < c ? 0 : nothing
(a::Number >= b) && (c::Number <= b)	a < c ? 0 : nothing
(a::Number >= b) && (c::Number < b)	a <= c ? 0 : nothing
a && (b || c)	(a && b) || (a && c)
a || (b && c)	(a || b) && (a || c)
b || (b && c)	b
0 / a	0
a / a	1
(-1 * a) / b	a / (-1 * b)
a / (-1 * b)	(-1 * a) / b
-1 * (a / b)	(-1 * a) / b
(-1 * a) / b	-1 * (a / b)
(a * b) / c	a / (c / b)
a / (c / b)	(a * b) / c
(a / b) * c	a / (b / c)
a / (b / c)	(a / b) * c
(a + b) / c	(a / c) + (b / c)

Pattern	Rewrite / Condition
`((a * b) + c) / d`	`((a * b) / d) + (c / d)`
`x == y`	`y == x`
`x == y`	`y != 0 && x != 0 ? :(($x - $y) == 0) : nothing`
`x + y == a`	`x == a - y`
`x == x`	`1`
`x*y == 0`	`(x == 0) || (y == 0)`
`max(x,y) == y`	`x <= y`
`min(x,y) == y`	`y <= x`
`y <= x`	`min(x,y) == y # creates huge number of expand nodes`
`x != y`	`!(x == y)`
`x > y`	`y < x`
`a < a`	`0`
`a <= a`	`1`
`a + b < c`	`a < c - b`
`a - b < a`	`0 < b`
`0 < a::Number`	`0 < a ? 1 : 0`
`a::Number < 0`	`a < 0 ? 1 : 0`
`min(a , b) <= a`	`b <= a`
`min(a, b) <= min(a , c)`	`b <= min(a, c)`
`max(a, b) <= max(a , c)`	`max(a ,b) <= c`
`min(a, b)`	`min(b, a)`
`min(min(a, b), c)`	`min(a, min(b, c))`
`min(a,a)`	`a`
`min(max(a, b), a)`	`a`
`min(max(a, b), max(a, c))`	`max(min(b, c), a)`
`min(max(min(a,b), c), b)`	`min(max(a,c), b)`
`min(a + b, c)`	`min(b, c - a) + a`
`min(a, b) + c`	`min(a + c, b + c)`
`min(a + c, b + c)`	`min(a, b) + c`
`min(c + a, b + c)`	`min(a, b) + c`
`min(a + c, b + c)`	`min(a, b) + c`
`min(c + a, c + b)`	`min(a, b) + c`
`min(a, a + b::Number)`	`b > 0 ? :($a) : nothing`
`min(a ,b) * c::Number`	`c > 0 ? :(min($a * $c, $b * $c)) : nothing`
`min(a * c::Number, b * c::Number)`	`c > 0 ? :(min($a ,$b) * $c) : nothing`
`min(a, b) / c::Number`	`c > 0 ? :(min($a / $c, $b / $c)) : nothing`
`min(a / c::Number, b / c::Number)`	`c>0 ? :(min($a, $b) / $c) : nothing`
`max(a , b) / c::Number`	`c < 0 ? :(max($a / $c , $b / $c)) : nothing`
`max(a / c::Number, b / c::Number)`	`c < 0 ? :(max($a, $b) / $c) : nothing`
`min(max(a,b::Number), c::Number)`	`c <= b ? :($c) : nothing`
`min(a % b::Number, c::Number)`	`c >= b - 1 ? :($a % $b) : nothing`
`min(a % b::Number, c::Number)`	`c <= 1 - b ? :($c) : nothing`
`min(max(a, b::Number), c::Number)`	`b <= c ? :(max(min($a, $c), $b)) : nothing`
`max(min(a, c::Number), b::Number)`	`b <= c ? :(min(max($a, $b), $c)) : nothing`
`min(a , b::Number) <= c::Number`	`a <= c || b <= c`
`max(a , b::Number) <= c::Number`	`a <= c && b <= c`
`c::Number <= max(a , b::Number)`	`c <= a || c <= b`

Pattern	Rewrite / Condition
`c::Number <= min(a , b::Number)`	`c <= a && c <= b`
`min(a * b::Number, c::Number)`	`c != 0 && b % c == 0 && b > 0 ? :(min($a, $b / $c) * $c) : nothing`
`min(a * b::Number, d * c::Number)`	`c != 0 && b % c == 0 && b > 0 ? :(min($a, $d * ($c/$b))*$b) : nothing`
`min(a * b::Number, c::Number)`	`c != 0 && b % c == 0 && b < 0 ? :(max($a, $b / $c) * $c) : nothing`
`min(a * b::Number, d * c::Number)`	`c != 0 && b % c == 0 && b < 0 ? :(max($a, $d * ($c/$b))*$b) : nothing`
`max(a * c::Number, b * c::Number)`	`c < 0 ? :(min($a, $b) * $c) : nothing`
`a % 0`	`0`
`a % a`	`0`
`a % 1`	`0`
`(a * -1) % b`	`-1 * (a % b)`
`-1 * (a % b)`	`(a * -1) % b`
`(a - b) % 2`	`(a + b) % 2`
`((a * b::Number) + d) % c::Number`	`c != 0 && b % c == 0 ? :($b % $c) : nothing`
`(b::Number * a) % c::Number`	`c != 0 && b % c == 0 ? 0 : nothing`
`a * b`	`b * a`
`a * (b * c)`	`(a * b) * c`
`a * 0`	`0`
`0 * a`	`0`
`a * 1`	`a`
`1 * a`	`a`
`max(a,b) * min(a, b)`	`a * b`
`min(a,b) * max(a, b)`	`a * b`
`(a * b) / b`	`a`
`(b * a) / b`	`a`
`x <= y`	`!(y < x)`
`!(y < x)`	`x <= y`
`x >= y`	`!(x < y)`
`!(x == y)`	`x != y`
`!(!x)`	`x`
`x || y`	`!((!x) && (!y))`
`y || x`	`x || y`
`a || 1`	`1`
`1 || a`	`1`
`a::Number <= b::Number`	`a<=b ? 1 : 0`
`a <= b - c`	`a + c <= b`
`a + c <= b`	`a <= b - c`
`a <= b + c`	`a - c <= b`
`a - c <= b`	`a <= b + c`
`a <= b + c`	`a - b <= c`
`a - b <= c`	`a <= b + c`
`a - a`	`0`
`min(a::Number, b::Number)`	`a >= b ? b : a`
`max(a::Number, b::Number)`	`a >= b ? a : b`
`a <= min(b,c)`	`a<=b && a<=c`
`min(b,c) <= a`	`b<=a || c<=a`
`a <= max(b,c)`	`a<=b || a<=c`
`max(b,c) <= a`	`b<=a || c<=a`

Pattern	Rewrite / Condition
a<=b && c<=a	c <= b
b<=a && a<=c	b <= c
a + b - c	a - c + b