

---

# Learning to Perform Local Rewriting for Combinatorial Optimization

---

**Xinyun Chen** \*  
UC Berkeley  
xinyun.chen@berkeley.edu

**Yuandong Tian**  
Facebook AI Research  
yuandong@fb.com

## Abstract

Search-based methods for hard combinatorial optimization are often guided by heuristics. Tuning heuristics in various conditions and situations is often time-consuming. In this paper, we propose *NeuRewriter* that learns a policy to pick heuristics and rewrite the local components of the current solution to iteratively improve it until convergence. The policy factorizes into a region-picking and a rule-picking component, each parameterized by a neural network trained with actor-critic methods in reinforcement learning. *NeuRewriter* captures the general structure of combinatorial problems and shows strong performance in three versatile tasks: expression simplification, online job scheduling and vehicle routing problems. *NeuRewriter* outperforms the expression simplification component in Z3 [15]; outperforms DeepRM [33] and Google OR-tools [19] in online job scheduling; and outperforms recent neural baselines [35, 29] and Google OR-tools [19] in vehicle routing problems.<sup>2</sup>

## 1 Introduction

Solving combinatorial problems is a long-standing challenge and has a lot of practical applications (e.g., job scheduling, theorem proving, planning, decision making). While problems with specific structures (e.g., shortest path) can be solved efficiently with proven algorithms (e.g. dynamic programming, greedy approach, search), many combinatorial problems are NP-hard and rely on manually designed heuristics to improve the quality of solutions [1, 40, 27].

Although it is usually easy to come up with many heuristics, determining when and where such heuristics should be applied, and how they should be prioritized, is time-consuming. It takes commercial solvers decades to tune to strong performance in practical problems [15, 44, 19].

To address this issue, previous works use neural networks to predict a complete solution from scratch, given a complete description of the problem [50, 33, 29, 21]. While this avoids search and tuning, a direct prediction could be difficult when the number of variables grows.

Improving iteratively from an existing solution is a common approach for continuous solution spaces, e.g. trajectory optimization in robotics [34, 47, 31]. However, such methods relying on gradient information to guide the search, is not applicable for discrete solution spaces due to indifferenciability.

To address this problem, we directly learn a neural-based policy that improves the current solution by iteratively rewriting a local part of it until convergence. Inspired by the problem structures, the policy is factorized into two parts: the region-picking and the rule-picking policy, and is trained end-to-end with reinforcement learning, rewarding cumulative improvement of the solution.

We apply our approach, *NeuRewriter*, to three different domains: expression simplification, online job scheduling, and vehicle routing problems. We show that *NeuRewriter* is better than strong

---

\*Work partially done when interning at Facebook AI Research.

<sup>2</sup>The code is available at <https://github.com/facebookresearch/neural-rewriter>.

heuristics using multiple metrics. For expression simplification, `NeuRewriter` outperforms the expression simplification component in Z3 [15]. For online job scheduling, under a controlled setting, `NeuRewriter` outperforms Google OR-tools [19] in terms of both speed and quality of the solution, and DeepRM [33], a neural-based approach that predicts a holistic scheduling plan, by large margins especially in more complicated setting (e.g., with more heterogeneous resources). For vehicle routing problems, `NeuRewriter` outperforms two recent neural network approaches [35, 29] and Google OR-tools [19]. Furthermore, extensive ablation studies show that our approach works well in different situations (e.g., different expression lengths, non-uniform job/resource distribution), and transfers well when distribution shifts (e.g., test on longer expressions than those used for training).

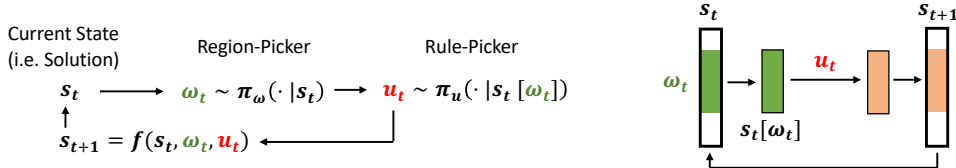


Figure 1: The framework of our neural rewriter. Given the current state (i.e., solution to the optimization problem)  $s_t$ , we first pick a region  $\omega_t$  by the region-picking policy  $\pi_\omega(\omega_t | s_t)$ , and then pick a rewriting rule  $u_t$  using the rule-picking policy  $\pi_u(u_t | s_t[\omega_t])$ , where  $\pi_u(u_t | s_t[\omega_t])$  gives the probability distribution of applying each rewriting rule  $u \in \mathcal{U}$  to the partial solution. Once the partial solution is updated, we obtain an improved solution  $s_{t+1}$  and repeat the process until convergence.

## 2 Related Work

**Methods.** Using neural network models for combinatorial optimization has been explored in the last few years. A straightforward idea is to construct a solution directly (e.g., with a Seq2Seq model) from the problem specification [50, 6, 33, 28]. However, such approaches might meet with difficulties if the problem has complex configurations, as our evaluation indicates. In contrast, our paper focuses on iterative improvement of a *complete* solution.

Trajectory optimization with local gradient information has been widely studied in robotics with many effective techniques [34, 9, 51, 47, 32, 31]. For discrete problems, it is possible to apply continuous relaxation and apply gradient descent [10]. In contrast, we *learn the gradient* from previous experience to optimize a complete solution, similar to data-driven descent [49] and synthetic gradient [26].

At a high level, our framework is closely connected with the local search pipeline. Specifically, we can leverage our learned RL policy to guide the local search, i.e., to decide which neighbor solution to move to. We will demonstrate that in our evaluated tasks, our approach outperforms several local search algorithms guided by manually designed heuristics, and softwares supporting more advanced local search algorithms, i.e., Z3 [15] and OR-tools [19].

**Applications.** For expression simplification, some recent work use deep neural networks to discover equivalent expressions [11, 2, 52]. In particular, [11] trains a deep neural network to rewrite algebraic expressions with supervised learning, which requires a collection of ground truth rewriting paths, and may not find novel rewriting routines. We mitigate these limitations using reinforcement learning.

Job scheduling and resource management problems are ubiquitous and fundamental in computer systems. Various work have studied these problems from both theoretical and empirical sides [8, 20, 3, 42, 48, 33, 13]. In particular, a recent line of work studies deep reinforcement learning for job scheduling [33, 13] and vehicle routing problems [29, 35].

Our approach is tested on multiple domains with extensive ablation studies, and could also be extended to other closely related tasks such as code optimization [41, 12], theorem proving [25, 30, 4, 24], text simplification [14, 37, 18], and classical combinatorial optimization problems beyond routing problems [16, 28, 7, 50, 27], e.g., Vertex Cover Problem [5].

## 3 Problem Setup

Let  $\mathcal{S}$  be the space of all feasible solutions in the problem domain, and  $c : \mathcal{S} \rightarrow \mathbb{R}$  be the cost function. The goal of optimization is to find  $\arg \min_{s \in \mathcal{S}} c(s)$ . In this work, instead of finding a solution from scratch, we first construct a feasible one, then make incremental improvement by iteratively applying *local rewriting rules* to the existing solution until convergence. Our rewriting formulation is especially suitable for problems with the following properties: (1) a feasible solution

is easy to find; **(2)** the search space has well-behaved local structures, which could be utilized to incrementally improve the solution. For such problems, a complete solution provides a full context for the improvement using a rewriting-based approach, allowing additional features to be computed, which is hard to obtain if the solution is generated from scratch; meanwhile, different solutions might share a common routine towards the optimum, which could be represented as local rewriting rules. For example, it is much easier to decide whether to postpone jobs with large resource requirements when an existing job schedule is provided. Furthermore, simple rules like swapping two jobs could improve the performance.

Formally, each solution is a *state*, and each local region and the associated rewriting rule is an *action*.

**Optimization as a rewriting problem.** Let  $\mathcal{U}$  be the rewriting ruleset. Suppose  $s_t$  is the current solution (or state) at iteration  $t$ . We first compute a state-dependent *region set*  $\Omega(s_t)$ , then pick a region  $\omega_t \in \Omega(s_t)$  using the *region-picking policy*  $\pi_\omega(\omega_t|s_t)$ . We then pick a rewriting rule  $u_t$  applicable to that region  $\omega_t$  using the *rule-picking policy*  $\pi_u(u_t|s_t[\omega_t])$ , where  $s_t[\omega_t]$  is a subset of state  $s_t$ . We then apply this rewriting rule  $u_t \in \mathcal{U}$  to  $s_t[\omega_t]$ , and obtain the next state  $s_{t+1} = f(s_t, \omega_t, u_t)$ . Given an initial solution (or state)  $s_0$ , our goal is to find a sequence of rewriting steps  $(s_0, (\omega_0, u_0)), (s_1, (\omega_1, u_1)), \dots, (s_{T-1}, (\omega_{T-1}, u_{T-1})), s_T$  so that the final cost  $c(s_T)$  is minimized.

To tackle a rewriting problem, rule-based rewriters with manually-designed rewriting routines have been proposed [23]. However, manually designing such routines is not a trivial task. An incomplete set of routines often leads to an inefficient exhaustive search, while a set of kaleidoscopic routines is often cumbersome to design, hard to maintain and lacks flexibility.

In this paper, we propose to train a neural network instead, using reinforcement learning. Recent advance in deep reinforcement learning suggests the potential of well-trained models to discover novel effective policies, such as demonstrated in Computer Go [43] and video games [36]. Moreover, by leveraging reinforcement learning, our approach could be extended to a broader range of problems that could be hard for rule-based rewriters and classic search algorithms. For example, we can design the reward to take the validity of the solution into account, so that we can start with an infeasible solution and then move towards a feasible one. On the other hand, we can also train the neural network to explore the connections between different solutions in the search space. In our evaluation, we demonstrate that our approach (1) mitigates laborious human efforts, (2) discovers novel rewriting paths from its own exploration, and (3) finds better solution to optimization problem than the current state-of-the-art and traditional heuristic-based software packages tuned for decades.

## 4 Neural Rewriter Model

In the following, we present the design of our rewriting model, i.e., *NeuRewriter*. We first provide an overview of our model framework, then present the design details for different applications.

### 4.1 Model Overview

Figure 1 illustrates the overall framework of our neural rewriter, and we describe the two key components for rewriting as follows. More details can be found in Appendix C.

**Score predictor.** Given the state  $s_t$ , the score predictor computes a score  $Q(s_t, \omega_t)$  for every  $\omega_t \in \Omega(s_t)$ , which measures the benefit of rewriting  $s_t[\omega_t]$ . A high score indicates that rewriting  $s_t[\omega_t]$  could be desirable. Note that  $\Omega(s_t)$  is a problem-dependent region set. For expression simplification,  $\Omega(s_t)$  includes all sub-trees of the expression parse trees; for job scheduling,  $\Omega(s_t)$  covers all job nodes for scheduling; and for vehicle routing, it includes all nodes in the route.

**Rule selector.** Given  $s_t[\omega_t]$  to be rewritten, the rule-picking policy predicts a probability distribution  $\pi_u(s_t[\omega_t])$  over the entire ruleset  $\mathcal{U}$ , and selects a rule  $u_t \in \mathcal{U}$  to apply accordingly.

### 4.2 Training Details

Let  $(s_0, (\omega_0, u_0)), \dots, (s_{T-1}, (\omega_{T-1}, u_{T-1})), s_T$  be the rewriting sequence in the forward pass.

**Reward function.** We define  $r(s_t, (\omega_t, u_t))$  as  $r(s_t, (\omega_t, u_t)) = c(s_t) - c(s_{t+1})$ , where  $c(\cdot)$  is the task-specific cost function in Section 3.

**Q-Actor-Critic training.** We train the region-picking policy  $\pi_\omega$  and rule-picking policy  $\pi_u$  simultaneously. For  $\pi_\omega(\omega_t|s_t; \theta)$ , we parameterize it as a softmax of the underlying  $Q(s_t, \omega_t; \theta)$  function:

$$\pi_\omega(\omega_t|s_t; \theta) = \frac{\exp(Q(s_t, \omega_t; \theta))}{\sum_{\omega_t} \exp(Q(s_t, \omega_t; \theta))} \quad (1)$$

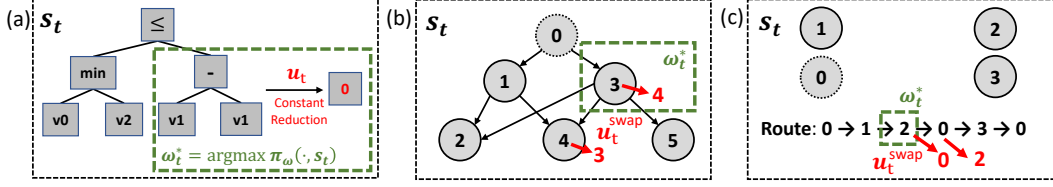


Figure 2: The instantiation of `NeuRewriter` for different domains: **(a)** expression simplification; **(b)** job scheduling; and **(c)** vehicle routing. In **(a)**,  $s_t$  is the expression parse tree, where each square represents a node in the tree. The set  $\Omega(s_t)$  includes every sub-tree rooted at a non-terminal node, from which the region-picking policy selects  $\omega_t \sim \pi_\omega(\omega_t | s_t)$  to rewrite. Afterwards, the rule-picking policy predicts a rewriting rule  $u_t \in \mathcal{U}$ , then rewrites the sub-tree  $\omega_t$  to get the new tree  $s_{t+1}$ . In **(b)**,  $s_t$  is the dependency graph representation of the job schedule. Each circle with index greater than 0 represents a job node, and node 0 is an additional one representing the machine. Edges in the graph reflect job dependencies. The region-picking policy selects a job  $\omega_t$  to re-schedule from all job nodes, then the rule-picking policy chooses a moving action  $u_t$  for  $\omega_t$ , then modifies  $s_t$  to get a new dependency graph  $s_{t+1}$ . In **(c)**,  $s_t$  is the current route, and  $\omega_t$  is the node selected to change the visit order. Node 0 is the depot, and other nodes are customers with certain resource demands. The region-picking policy and the rule-picking policy work similarly to the job scheduling ones.

and instead learn  $Q(s_t, \omega_t; \theta)$  by fitting it to the cumulative reward sampled from the current policies  $\pi_\omega$  and  $\pi_u$ :

$$L_\omega(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} \left( \sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{t'}, (\omega_{t'}, u_{t'})) - Q(s_t, \omega_t; \theta) \right)^2 \quad (2)$$

Where  $T$  is the length of the episode (i.e., the number of rewriting steps), and  $\gamma$  is the decay factor.

For rule-picking policy  $\pi_u(u_t | s_t[\omega_t]; \phi)$ , we employ the Advantage Actor-Critic algorithm [45] with the learned  $Q(s_t, \omega_t; \theta)$  as the critic, and thus avoid boot-strapping which could cause sample insufficiency and instability in training. This formulation is similar in spirit to soft-Q learning [22]. Denoting  $\Delta(s_t, (\omega_t, u_t)) \equiv \sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{t'}, (\omega_{t'}, u_{t'})) - Q(s_t, \omega_t; \theta)$  as the advantage function, the loss function of the rule selector is:

$$L_u(\phi) = - \sum_{t=0}^{T-1} \Delta(s_t, (\omega_t, u_t)) \log \pi_u(u_t | s_t[\omega_t]; \phi) \quad (3)$$

The overall loss function is  $L(\theta, \phi) = L_u(\phi) + \alpha L_\omega(\theta)$ , where  $\alpha$  is a hyper-parameter. More training details can be found in Appendix D.

## 5 Applications

In the following sections, we discuss the application of our rewriting approach to three different domains: expression simplification, online job scheduling, and vehicle routing. In expression simplification, we minimize the expression length using a well-defined semantics-preserving rewriting ruleset. In online job scheduling, we aim to reduce the overall waiting time of jobs. In vehicle routing, we aim to minimize the total tour length.

### 5.1 Expression Simplification

We first apply our approach to expression simplification domain. In particular, we consider expressions in Halide, a domain-specific language for high-performance image processing [39], which is widely used at scale in multiple products of Google (e.g., YouTube) and Adobe Photoshop. Simplifying Halide expressions is an important step towards the optimization of the entire code. To this end, a rule-based rewriter is implemented for the expressions, which is carefully tuned with manually-designed heuristics. The grammar of the expressions considered in the rewriter is specified in Appendix A.1. Notice that the grammar includes a more comprehensive operator set than previous works on finding equivalent expressions, which consider only boolean expressions [2, 17] or a subset of arithmetic operations [2]. The rewriter includes hundreds of manually-designed rewriting templates. Given an expression, the rewriter checks the templates in a pre-designed order, and applies those rewriting templates that match any sub-expression of the input.

After investigating the rewriting templates in the rule-based rewriter, we find that a large number of rewriting templates enumerate specific cases for an *uphill rule*, which lengthens the expression first

and shortens it later (e.g., “min/max” expansion). Similar to momentum terms in gradient descent for continuous optimization, such rules are used to escape a local optimum. However, they should only be applied when the initial expression satisfies certain *pre-conditions*, which is traditionally specified by manual design, a cumbersome process that is hard to generalize.

Observing these limitations, we hypothesize that a neural network model has the potential of doing a better job than the rule-based rewriter. In particular, we propose to only keep the core rewriting rules in the ruleset, remove all unnecessary pre-conditions, and let the neural network decide which and when to apply each rewriting rule. In this way, the neural rewriter has a better flexibility than the rule-based rewriter, because it can learn such rewriting decisions from data, and has the ability of discovering novel rewriting patterns that are not included in the rule-based rewriter.

**Ruleset.** We incorporate two kinds of templates from Halide rewriting ruleset. The first kind is simple rules (e.g.,  $v - v \rightarrow 0$ ), while the second one is the uphill rules after removing their manually designed pre-conditions that do not affect the validity of the rewriting. In this way, a ruleset with  $|\mathcal{U}| = 19$  categories is built. See Appendix B.1 for more details.

**Model specification.** We use expression parse trees as the input, and employ the N-ary Tree-LSTM designed in [46] as the input encoder to compute the embedding for each node in the tree. Both the score predictor and the rule selector are fully connected neural networks, taken the LSTM embeddings as the input. More details can be found in Appendix C.1.

## 5.2 Job Scheduling Problem

We also study the job scheduling problem, using the problem setup in [33].

**Notation.** Suppose we have a machine with  $D$  types of resources. Each job  $j$  is specified as  $v_j = (\rho_j, A_j, T_j)$ , where the  $D$ -dimensional vector  $\rho_j = [\rho_{jd}]$  denotes the required portion  $0 \leq \rho_{jd} \leq 1$  of the resource type  $d$ ,  $A_j$  is the arrival timestep, and  $T_j$  is the duration. In addition, we define  $B_j$  as the scheduled beginning time, and  $C_j = B_j + T_j$  as the completion time.

We assume that the resource requirement is fixed during the entire job execution, each job must run continuously until finishing, and no preemption is allowed. We adopt an online setting: there is a pending job queue that can hold at most  $W$  jobs. When a new job arrives, it can either be allocated immediately, or be added to the queue. If the queue is already full, to make space for the new job, at least one job in the queue needs to be scheduled immediately. The goal is to find a time schedule for every job, so that the average waiting time is as short as possible.

**Ruleset.** The set of rewriting rules is to re-schedule a job  $v_j$  and allocate it after another job  $v_{j'}$  finishes or at its arrival time  $A_j$ . See Appendix B.2 for details of a rewriting step. The size of the rewriting ruleset is  $|\mathcal{U}| = 2W$ , since each job could only switch its scheduling order with at most  $W$  of its former and latter jobs respectively.

**Representation.** We represent each schedule as a directed acyclic graph (DAG), which describes the dependency among the schedule time of different jobs. Specifically, we denote each job  $v_j$  as a node in the graph, and we add an additional node  $v_0$  to represent the machine. If a job  $v_j$  is scheduled at its arrival time  $A_j$  (i.e.,  $B_j = A_j$ ), then we add a directed edge  $\langle v_0, v_j \rangle$  in the graph. Otherwise, there must exist at least one job  $v_{j'}$  such that  $C_{j'} = B_j$  (i.e., job  $j$  starts right after job  $j'$ ). We add an edge  $\langle v_{j'}, v_j \rangle$  for every such job  $v_{j'}$  to the graph. Figure 2(b) shows the setting, and we defer the embedding and graph construction details to Appendix C.2.

**Model specification.** To encode the graphs, we extend the Child-Sum Tree-LSTM architecture in [46], which is similar to the DAG-structured LSTM in [53]. Similar to the expression simplification model, both the score predictor and the rule selector are fully connected neural networks, and we defer the model details to Appendix C.2.

## 5.3 Vehicle Routing Problem

In addition, we evaluate our approach on vehicle routing problems studied in [29, 35]. Specifically, we focus on the Capacitated VRP (CVRP), where a single vehicle with limited capacity needs to satisfy the resource demands of a set of customer nodes. To do so, we construct multiple routes starting and ending at the depot, i.e., node 0 in Figure 2(c), so that the resources delivered in each route do not exceed the vehicle capacity, while the total route length is minimized.

We represent each vehicle routing problem as a sequence of the nodes visited in the tour, and use a bi-directional LSTM to embed the routes. The ruleset is similar to the job scheduling, where each node can swap with another node in the route. The architectures of the score predictor and rule selector are similar to job scheduling. More details can be found in Appendix C.3.

## 6 Experiments

We present the evaluation results in this section. To calculate the inference time, we run all algorithms on the same server equipped with 2 Quadro GP100 GPUs and 80 CPU cores. Only 1 GPU is used when evaluating neural networks, and 4 CPU cores are used for search algorithms. We set the timeout of search algorithms to be 10 seconds per instance. All neural networks in our evaluation are implemented in PyTorch [38].

### 6.1 Expression Simplification

**Setup.** To construct the dataset, we first generate random pipelines using the generator in Halide, then extract expressions from them. We filter out those irreducible expressions, then split the rest into 8/1/1 for training/validation/test sets respectively. See Appendix A.1 for more details.

**Metrics.** We evaluate the following two metrics: (1) *Average expression length reduction*, which is the length reduced from the initial expression to the rewritten one, and the length is defined as the number of characters in the expression; (2) *Average tree size reduction*, which is the number of nodes decreased from the initial expression parse tree to the rewritten one.

**Baselines.** We examine the effectiveness of NeuRewriter against two kinds of baselines. The first kind of baselines are heuristic-based rewriting approaches, including Halide-rule (the rule-based Halide rewriter in Section 3) and Heuristic-search, which applies beam search to find the shortest rewriting with our ruleset at each step. Note that NeuRewriter does not use beam search.

In addition, we also compare our approach with Z3, a high-performance theorem prover developed by Microsoft Research [15]. Z3 provides two tactics to simplify the expressions: Z3-simplify performs some local transformation using its pre-defined rules, and Z3-ctx-solver-simplify traverses each sub-formula in the input expression and invokes the solver to find a simpler equivalent one to replace it. This search-based tactic is able to perform simplification not included in the Halide ruleset, and is generally better than the rule-based counterpart but with more computation. For Z3-ctx-solver-simplify, we set the timeout to be 10 seconds for each input expression.

**Results.** Figure 3a presents the main results. We can notice that the performance of Z3-simplify is worse than Halide-rule, because the ruleset included in this simplifier is more restricted than the Halide one, and in particular, it can not handle expressions with “max/min/select” operators. On the other hand, NeuRewriter outperforms both the rule-based rewriters and the heuristic search by a large margin. In particular, NeuRewriter could reduce the expression length and parse tree size by around 52% and 59% on average; compared to the rule-based rewriters, our model further reduces the average expression length and tree size by around 20% and 15% respectively. We observe that the main performance gain comes from learning to apply uphill rules appropriately in ways that are not included in the manually-designed templates. For example, consider the expression  $5 \leq \max(\max(v0, 3) + 3, \max(v1, v2))$ , which could be reduced to *True* by expanding  $\max(\max(v0, 3) + 3, \max(v1, v2))$  and  $\max(v0, 3)$ . Using a rule-based rewriter would require the need of specifying the pre-conditions recursively, which becomes prohibitive when the expressions become more complex. On the other hand, heuristic search may not be able to find the correct order of expanding the right hand side of the expression when more “min/max” are included, which would make the search less efficient.

Furthermore, NeuRewriter also outperforms Z3-ctx-solver-simplify in terms of both the result quality and the time efficiency, as shown in Figure 3a and Table 1a. Note that the implementation of Z3 is in C++ and highly optimized, while NeuRewriter is implemented in Python; meanwhile, Z3-ctx-solver-simplify could perform rewriting steps that are not included in the Halide ruleset. More results can be found in Appendix G.

**Generalization to longer expressions.** To measure the generalizability of our approach, we construct 4 subsets of the training set:  $Train_{\leq 20}$ ,  $Train_{\leq 30}$ ,  $Train_{\leq 50}$  and  $Train_{\leq 100}$ , which only include expressions of length at most 20, 30, 50 and 100 in the full training set. We also build  $Test_{>100}$ , a subset of the full test set that only includes expressions of length larger than 100. The statistics of these datasets can be found in Appendix A.1.

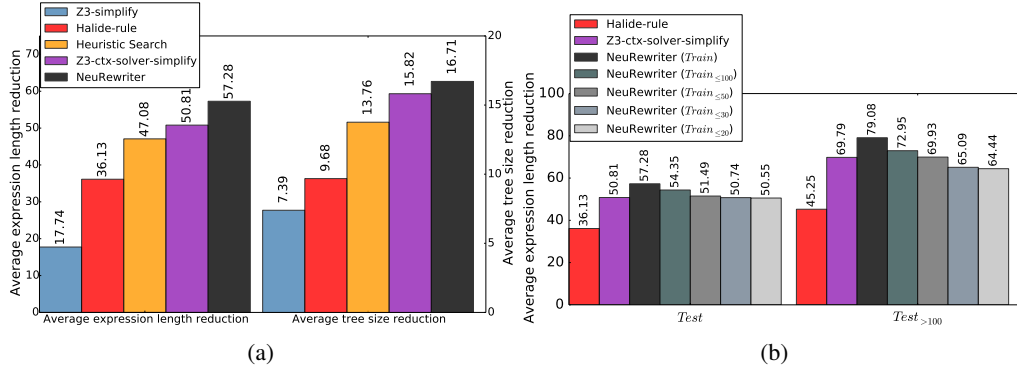


Figure 3: Experimental results of the expression simplification problem. In (b), we train NeuRewriter on expressions of different lengths (described in the brackets).

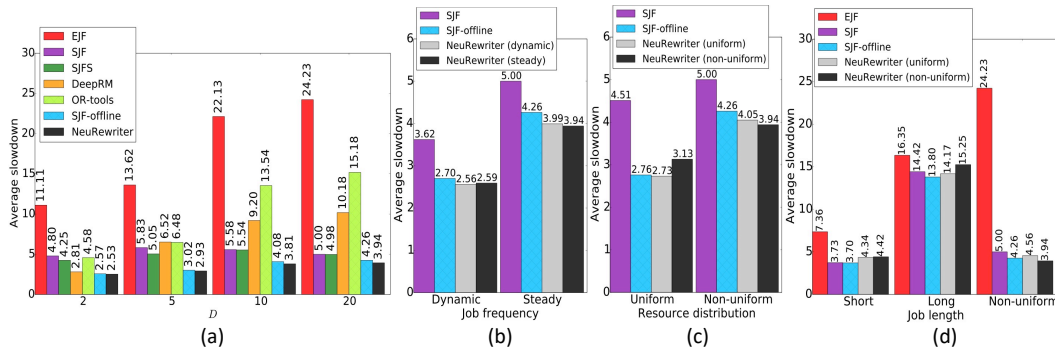


Figure 4: Experimental results of the job scheduling problem varying the following aspects: (a) the number of resource types  $D$ ; (b) job frequency; (c) resource distribution; (d) job length. For NeuRewriter, we describe training job distributions in the brackets. Workloads in (a) are with steady job frequency, non-uniform resource distribution, and non-uniform job length. In (b), (c) and (d),  $D = 20$ . In (b) and (c), we omit the comparison with some approaches because their results are significantly worse; for example, the average slowdown of EJF is 14.53 on the dynamic job frequency, and 11.06 on the uniform resource distribution. More results can be found in Appendix E.

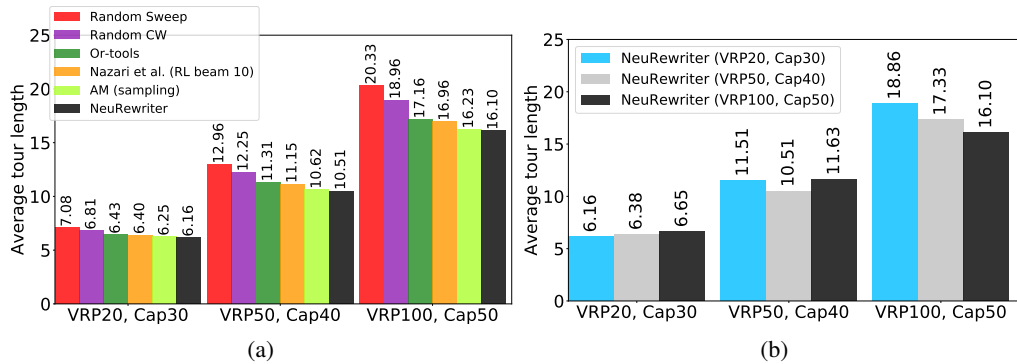


Figure 5: Experimental results of the vehicle routing problem with different number of customer nodes and vehicle capacity; e.g., VRP100, Cap50 means there are 100 customer nodes and the vehicle capacity is 50. (a) NeuRewriter outperforms multiple baselines and previous works [29, 35]. More results can be found in Appendix F. (b) We evaluate the generalization performance of NeuRewriter on problems from different distributions, and we describe the training problem distributions in the brackets.

We present the results of training our model on different datasets above in Figure 3b. Even trained on short expressions, `NeuRewriter` is still comparable with the Z3 solver. Thanks to local rewriting rules, our approach can generalize well even when operating on very different data distributions.

## 6.2 Job Scheduling Problem

**Setup.** We randomly generate 100K job sequences, and use 80K/10K/10K for training, validation and testing. Typically each job sequence includes  $\sim 50$  jobs. We use an online setting where jobs arrive on the fly with a pending job queue of length  $W = 10$ . Unless stated otherwise, we generate initial schedules using *Earliest Job First* (EJF), which can be constructed with negligible overhead.

When the number of resource types  $D = 2$ , we follow the same setup as in [33]. The maximal job duration  $T_{max} = 15$ , and the latest job arrival time  $A_{max} = 50$ . With larger  $D$ , except changing the resource requirement of each job to include more resource types, other configurations stay the same.

**Metric.** Following DeepRM [33], we use the *average job slowdown*  $\eta_j \equiv (C_j - A_j)/T_j \geq 1$  as our evaluation metric. Note that  $\eta_j = 1$  means no slow down.

**Job properties.** To test the stability and generalizability of `NeuRewriter`, we change job properties (and their distributions): (1) *Number of resource types  $D$* : larger  $D$  leads to more complicated scheduling; (2) *Average job arrival rate*: the probability that a new job will arrive, *Steady job frequency* sets it to be 70%, and *Dynamic job frequency* means the job arrival rate changes randomly at each timestep; (3) *Resource distribution*: jobs might require different resources, where some are *uniform* (e.g., half-half for resource 1 and 2) while others are *non-uniform* (see Appendix A.2 for the detailed description); (4) *Job lengths*: *Uniform job length*: length of each job in the workload is either [10, 15] (long) or [1, 3] (short), and *Non-uniform job length*: workload has both short and long jobs. We show that `NeuRewriter` is fairly robust under different distributions. When trained on one distribution, it can generalize to others without performance collapse.

We compare `NeuRewriter` with three kinds of baselines.

**Baselines on Manually designed heuristics:** *Earliest Job First* (EJF) schedules each job in the increasing order of their arrival time. *Shortest Job First* (SJF) always allocates the shortest job in the pending job queue at each timestep, which is also used as a baseline in [33]. *Shortest First Search* (SJFS) searches over the shortest  $k$  jobs to schedule at each timestep, and returns the optimal one. We find that other heuristic-based baselines used in [33] generally perform worse than SJF, especially with large  $D$ . Thus, we omit the comparison.

**Baselines on Neural network.** We compare with DeepRM [33], a neural network also trained with RL to construct a solution from scratch.

**Baselines on Offline planning.** To measure the optimality of these algorithms, we also take an offline setting, where the entire job sequence is available before scheduling. Note that this is equivalent to assuming an unbounded length of the pending job queue. With such additional knowledge, this setting provides a strong baseline. We tried two offline algorithms: (1) *SJF-offline*, which is a simple heuristic that schedules each job in the increasing order of its duration; and (2) Google OR-tools [19], which is a generic toolbox for combinatorial optimization. For OR-tools, we set the timeout to be 10 seconds per workload, but we find that it can not achieve a good performance even with a larger timeout, and we defer the discussion to Appendix E.

**Results on Scalability.** As shown in Figure 4a, `NeuRewriter` outperforms both heuristic algorithms and the baseline neural network DeepRM. In particular, while the performance of DeepRM and `NeuRewriter` are similar when  $D = 2$ , with larger  $D$ , DeepRM starts to perform worse than heuristic-based algorithms, which is consistent with our hypothesis that it becomes challenging to design a schedule from scratch when the environment becomes more complex. On the other hand, `NeuRewriter` could capture the bottleneck of an existing schedule that limits its efficiency, then progressively refine it to obtain a better one. In particular, our results are even better than offline algorithms that assume the knowledge of the entire job sequence, which further demonstrates the effectiveness of `NeuRewriter`. Meanwhile, we present the running time of OR-tools, DeepRM and `NeuRewriter` in Table 1b. We can observe that both DeepRM and `NeuRewriter` are much more time-efficient than OR-tools; on the other hand, the running time of `NeuRewriter` is comparable to DeepRM, while achieving much better results. More discussion can be found in Appendix E.

**Results on Robustness.** As shown in Figure 4, `NeuRewriter` excels in almost all different job distributions, except when the job lengths are uniform (short or long, Figure 4d), in which case



	Time (s)
Z3-solver	1.375
NeuRewriter	0.159

(a)

	Time (s)
OR-tools	10.0
DeepRM	0.020
NeuRewriter	0.037

(b)

	VRP20	VRP50	VRP100
OR-tools	0.010	0.053	0.231
Nazari et al.	0.162	0.232	0.445
AM	0.036	0.168	0.720
NeuRewriter	0.133	0.211	0.398

(c)

Table 1: Average runtime (per instance) of different solvers (OR-tools [19] and the tactic `Z3-ctx-solver-simplify` of Z3 [15]) and RL-based approaches (`NeuRewriter`, `DeepRM` [33], Nazari et al. [35] and `AM` [29]) over the test set of: **(a)** expression simplification; **(b)** job scheduling; **(c)** vehicle routing.

existing methods/heuristics are sufficient. This shows that `NeuRewriter` can deal with complicated scenarios and is adaptive to different distributions.

**Results on Generalization.** Furthermore, `NeuRewriter` can also generalize to different distributions than those used in training, without substantial performance drop. This shows the power of local rewriting rules: using local context could yield more generalizable solutions.

### 6.3 Vehicle Routing Problem

**Setup and Baselines.** We follow the same training setup as [29, 35] by randomly generating vehicle routing problems with different number of customer nodes and vehicle capacity. We compare with two neural network approaches, i.e., `AM` [29] and Nazari et al. [35], and both of them train a neural network policy using reinforcement learning to construct the route from scratch. We also compare with `OR-tools` and several classic heuristics studied in [35].

**Results.** We first demonstrate our main results in Figure 5a, where we include the variant of each baseline that performs the best, and defer more results to Appendix F. Note that the initial routes generated for `NeuRewriter` are even worse than the classic heuristics; however, starting from such sub-optimal solutions, `NeuRewriter` is still able to iteratively improve the solutions and outperforms all the baseline approaches on different problem distributions. In addition, for `VRP20` problems, we can compute the exact optimal solutions, which provides an average tour length of 6.10. We observe that the result of `NeuRewriter` (i.e., 6.16) is the closest to this lower bound, which also demonstrates that `NeuRewriter` is able to find solutions with better quality.

We also compare the runtime of the most competitive approaches in Table 1c. Note that the `OR-Tools` solver for vehicle routing problems is highly tuned and implemented in C++, while the RL-based approaches in comparison are implemented in Python. Meanwhile, following [35], to report the runtime of RL models, we decode a single instance at a time, thus there is potential room for speed improvement by decoding multiple instances per batch. Nevertheless, we can still observe that `NeuRewriter` achieves a better balance between the result quality and the time efficiency, especially with a larger problem scale.

**Results on Generalization.** Furthermore, in Figure 5b, we show that `NeuRewriter` can generalize to different problem distributions than training ones. In particular, they still exceed the performance of the classic heuristics, and are sometimes comparable or even better than the `OR-tools`. More discussion can be found in Appendix F.

## 7 Conclusion

In this work, we propose to formulate optimization as a rewriting problem, and solve the problem by iteratively rewriting an existing solution towards the optimum. We utilize deep reinforcement learning to train our neural rewriter. In our evaluation, we demonstrate the effectiveness of our neural rewriter on multiple domains, where our model outperforms both heuristic-based algorithms and baseline deep neural networks that generate an entire solution directly.

Meanwhile, we observe that since our approach is based on local rewriting, it could become time-consuming when large changes are needed in each iteration of rewriting. In extreme cases where each rewriting step needs to change the global structure, starting from scratch becomes preferable. We consider improving the efficiency of our rewriting approach and extending it to more complicated scenarios as future work.

## References

- [1] M. Affenzeller and R. Mayrhofer. Generic heuristics for combinatorial optimization problems. In *Proc. of the 9th International Conference on Operational Research*, pages 83–92, 2002.
- [2] M. Allamanis, P. Chanthirasegaran, P. Kohli, and C. Sutton. Learning continuous semantic representations of symbolic expressions. In *International Conference on Machine Learning*, pages 80–88, 2017.
- [3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
- [4] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
- [5] R. Bar-Yehuda and S. Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981.
- [6] A. Bay and B. Sengupta. Approximating meta-heuristics with homotopic recurrent neural networks. *arXiv preprint arXiv:1709.02194*, 2017.
- [7] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.
- [8] J. Błażewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European journal of operational research*, 93(1):1–33, 1996.
- [9] S. J. Bradtke, B. E. Ydstie, and A. G. Barto. Adaptive linear quadratic control using policy iteration. In *Proceedings of the American control conference*, volume 3, pages 3475–3475. Citeseer, 1994.
- [10] R. R. Bunel, A. Desmaison, P. K. Mudigonda, P. Kohli, and P. Torr. Adaptive neural compilation. In *Advances in Neural Information Processing Systems*, pages 1444–1452, 2016.
- [11] C.-H. Cai, Y. Xu, D. Ke, and K. Su. Learning of human-like algebraic reasoning using deep feedforward neural networks. *Biologically Inspired Cognitive Architectures*, 25:43–50, 2018.
- [12] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. Learning to optimize tensor programs. *NIPS*, 2018.
- [13] W. Chen, Y. Xu, and X. Wu. Deep reinforcement learning for multi-resource multi-machine job scheduling. *arXiv preprint arXiv:1711.07440*, 2017.
- [14] T. A. Cohn and M. Lapata. Sentence compression as tree transduction. *Journal of Artificial Intelligence Research*, 34:637–674, 2009.
- [15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [16] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau. Learning heuristics for the tsp by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181. Springer, 2018.
- [17] R. Evans, D. Saxton, D. Amos, P. Kohli, and E. Grefenstette. Can neural networks understand logical entailment? *ICLR*, 2018.
- [18] D. Feblowitz and D. Kauchak. Sentence simplification as tree transduction. In *Proceedings of the Second Workshop on Predicting and Improving Text Readability for Target Reader Populations*, pages 1–10, 2013.
- [19] Google. Google or-tools. <https://developers.google.com/optimization/>, 2019.
- [20] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2015.
- [21] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.
- [22] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. In *ICML*, pages 1352–1361. JMLR. org, 2017.
- [23] Halide. Halide simplifier. <https://github.com/halide/Halide>, 2018.

- [24] J. Hsiang, H. Kirchner, P. Lescanne, and M. Rusinowitch. The term rewriting approach to automated theorem proving. *The Journal of Logic Programming*, 14(1-2):71–99, 1992.
- [25] D. Huang, P. Dhariwal, D. Song, and I. Sutskever. Gamepad: A learning environment for theorem proving. *arXiv preprint arXiv:1806.00608*, 2018.
- [26] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.
- [27] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.
- [28] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.
- [29] W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.
- [30] G. Lederman, M. N. Rabe, and S. A. Seshia. Learning heuristics for automated reasoning through deep reinforcement learning. *arXiv preprint arXiv:1807.08058*, 2018.
- [31] S. Levine and P. Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.
- [32] S. Levine and V. Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.
- [33] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016.
- [34] D. Q. MAYNE. Differential dynamic programming—a unified approach to the optimization of dynamic systems. In *Control and Dynamic Systems*, volume 10, pages 179–254. Elsevier, 1973.
- [35] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takac. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9861–9871, 2018.
- [36] OpenAI. Openai dota 2 bot. <https://openai.com/the-international/>, 2018.
- [37] G. H. Paetzold and L. Specia. Text simplification as tree transduction. In *Proceedings of the 9th Brazilian Symposium in Information and Human Language Technology*, 2013.
- [38] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.
- [39] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.
- [40] C. R. Reeves. *Modern heuristic techniques for combinatorial problems. Advanced topics in computer science*, volume 15. Mc Graw-Hill, 1995.
- [41] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.
- [42] Z. Scully, G. Blelloch, M. Harchol-Balter, and A. Scheller-Wolf. Optimally scheduling jobs with multiple tasks. *ACM SIGMETRICS Performance Evaluation Review*, 45(2):36–38, 2017.
- [43] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.
- [44] N. Sorensson and N. Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.
- [45] R. S. Sutton, A. G. Barto, et al. *Reinforcement learning: An introduction*. 1998.
- [46] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2015.

- [47] Y. Tassa, T. Erez, and E. Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4906–4913. IEEE, 2012.
- [48] D. Terekhov, D. G. Down, and J. C. Beck. Queueing-theoretic approaches for dynamic scheduling: a survey. *Surveys in Operations Research and Management Science*, 19(2):105–129, 2014.
- [49] Y. Tian and S. G. Narasimhan. Hierarchical data-driven descent for efficient optimal deformation estimation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2288–2295, 2013.
- [50] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.
- [51] D. Vrabie, O. Pastravanu, M. Abu-Khalaf, and F. L. Lewis. Adaptive optimal control for continuous-time linear systems based on policy iteration. *Automatica*, 45(2):477–484, 2009.
- [52] W. Zaremba, K. Kurach, and R. Fergus. Learning to discover efficient mathematical identities. In *Advances in Neural Information Processing Systems*, pages 1278–1286, 2014.
- [53] X. Zhu, P. Sobhani, and H. Guo. Dag-structured long short-term memory for semantic compositionality. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 917–926, 2016.

```

    <Expr> ::= <AlgExpr> | <BoolExpr>
    <BoolExpr> ::= <AlgExpr> <<AlgExpr>
                | <AlgExpr> <= <AlgExpr>
                | <AlgExpr> == <AlgExpr>
                | (!<BoolExpr>)
                | (<BoolExpr> && <BoolExpr>)
                | (<BoolExpr> || <BoolExpr>)
    <AlgExpr> ::= <Term>
                | (<AlgExpr> + <Term>)
                | (<AlgExpr> - <Term>)
                | (<AlgExpr> * <Term>)
                | (<AlgExpr> / <Term>)
                | (<AlgExpr> % <Term>)
    <Term> ::= <Var> | <Const>
            | max(<AlgExpr>, <AlgExpr>)
            | min(<AlgExpr>, <AlgExpr>)
            | select(<BoolExpr>, <AlgExpr>, <AlgExpr>)

```

Figure 6: Grammar of the Halide expressions in our evaluation. “select ( $c$ ,  $e1$ ,  $e2$ )” means that when the condition  $c$  is satisfied, this term is equal to  $e1$ , otherwise is equal to  $e2$ . In our dataset, all constants are integers ranging in  $[-1024, 1024]$ , and variables are from the set  $\{v0, v1, \dots, v12\}$ .

Number of expressions in the dataset	Length of expressions	Size of expression parse trees
Total: 1.36M	Average: 106.84	Average: 27.39
Train/Val/Test: 1.09M/136K/136K	Min/Max: 10/579	Min/Max:3/100
<i>Train</i> <sub>≤20</sub> : 17K	Average: 16.76	Average: 4.66
<i>Train</i> <sub>≤30</sub> : 48K	Average: 22.91	Average: 6.43
<i>Train</i> <sub>≤50</sub> : 170K	Average: 35.62	Average: 10.18
<i>Train</i> <sub>≤100</sub> : 588K	Average: 63.49	Average: 18.72
<i>Test</i> <sub>&gt;100</sub> : 53K	Average: 142.22	Average: 42.20

Table 2: Statistics of the dataset for expression simplification.

## A More Details of the Dataset

### A.1 Expression Simplification

Figure 6 presents the grammar of Halide expressions in our evaluation. We use the random pipeline generator in the Halide repository to build the dataset<sup>3</sup>. Table 2 presents the statistics of the datasets.

### A.2 Job Scheduling

**Description of different resource distributions.** For each job  $j$ , we define *dominant resources*  $d_{\text{dom}}$  as the resources with  $0.5 \leq \rho_j d_{\text{dom}} \leq 1$ , and *auxiliary resources*  $d_{\text{aux}}$  as those with  $0.1 \leq \rho_j d_{\text{aux}} \leq 0.2$ . We refer to a job with both dominant and auxiliary resources as a job with *non-uniform resources*. We also evaluate on workloads including only jobs with *uniform resources*, where each job only includes either dominant resources or auxiliary resources.

### A.3 Vehicle Routing

Our data generation follows the setup in [35, 29]. The positions of the depot and customer nodes are uniformly randomly sampled from the unit square  $[0, 1] \times [0, 1]$ . Each node is denoted as  $v_j = ((x_j, y_j), \delta_j)$ , where  $(x_j, y_j)$  is the position, and  $\delta_j$  is the resource demand. We set  $\delta_0 = 0$  for the depot (i.e., node 0), and  $\delta_j \in \{1, 2, \dots, 9\}$  for customer nodes (i.e.,  $j > 0$ ).

<sup>3</sup>[https://github.com/halide/Halide/tree/new\\_autoschedule\\_with\\_new\\_simplifier/apps/random\\_pipeline](https://github.com/halide/Halide/tree/new_autoschedule_with_new_simplifier/apps/random_pipeline).

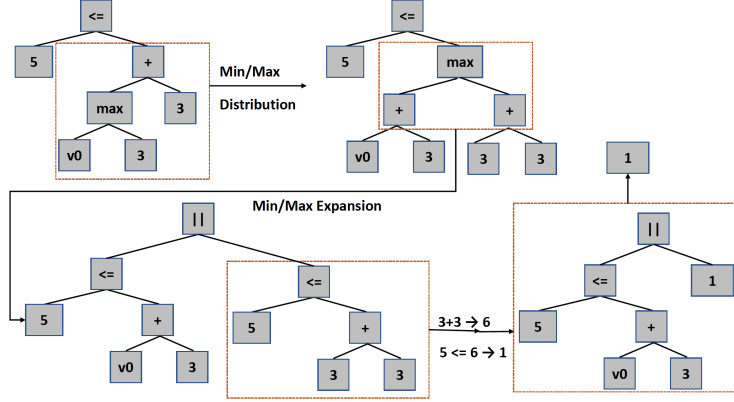


Figure 7: An example of the rewriting process for Halide expressions. The initial expression is  $5 \leq \max(v0, 3) + 3$ , which could be reduced to 1, i.e., *True*.

---

**Algorithm 1** Algorithm of a Single Rewriting Step for Job Scheduling Problem

---

```

1: function REWRITE( $v_j, v_{j'}, s_t$ )
2:   if  $C_{j'} < A_j$  or  $C_{j'} == B_j$  then
3:     return  $s_t$ 
4:   end if
5:   if  $j' \neq 0$  then  $B'_j = C_{j'}$  else  $B'_j = A_j$  fi
6:    $C'_j = B'_j + T_j$ 
7:
8:   //Resolve potential resource occupation overflow within  $[B'_j, C'_j]$ 
9:    $J =$  all jobs in  $s_t$  except  $v_j$  that are scheduled within  $[B'_j, C'_j]$ 
10:  Sort  $J$  in the topological order
11:  for  $v_i \in J$  do
12:     $B'_i =$  the earliest time that job  $v_i$  can be scheduled
13:     $C'_i = B'_i + T_i$ 
14:  end for
15:  For  $v_i \notin J$ ,  $B'_i = B_i$ ,  $C'_i = C_i$ 
16:   $s_{t+1} = \{(B'_i, C'_i)\}$ 
17:  return  $s_{t+1}$ 
18: end function

```

---

## B More Details on the Rewriting Ruleset

### B.1 More Details for Expression Simplification Problem

The ruleset implemented in the Halide rule-based rewriter can be found in their public repository <sup>4</sup>.

**More discussions about the uphill rules.** A commonly used type of uphill rules is “min/max” expansion, e.g.,  $\min(a, b) < c \rightarrow a < c \parallel b < c$ . Dozens of templates in the ruleset of the Halide rewriter are describing conditions when a “min/max” expression could be simplified. Notice that although applying this rewriting rule has no benefit in most cases, since it will increase the expression length, it is necessary to include it in the ruleset, because when either  $a < c$  or  $b < c$  is always true, expanding the “min” term could reduce the entire expression to a tautology, which ends up simplifying the entire expression. Figure 7 shows an example of the rewriting process using uphill rules properly.

### B.2 More Details for Job Scheduling Problem

Algorithm 1 describes a single rewriting step for job scheduling problem.

<sup>4</sup> <https://github.com/halide/Halide>.

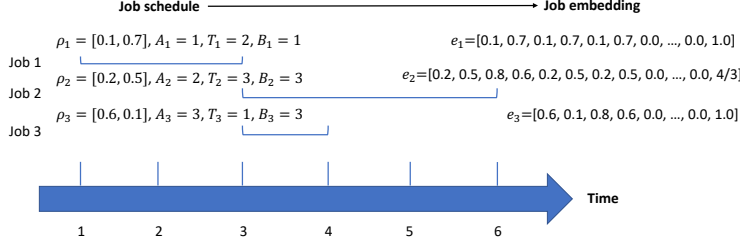


Figure 8: An example to illustrate the job embedding approach for the job scheduling problem.

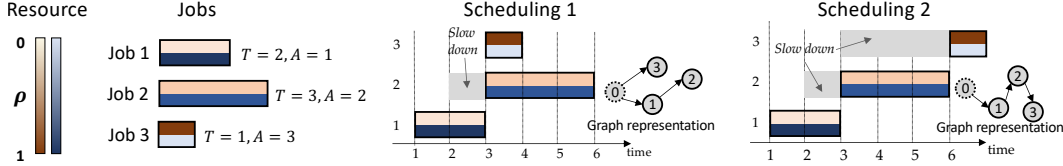


Figure 9: An example to illustrate two possible job schedules on a single machine and their corresponding graph representations. Node 0 was added to represent the start of the scheduling process. For multiple machines, multiple node 0 will be added.

## C More Details on Model Architectures

### C.1 Model Details for Expression Simplification

**Input embedding.** Notice that in this problem, each non-terminal has at most 3 children. Thus, let  $x$  be the embedding of a non-terminal,  $(h_L, c_L), (h_M, c_M), (h_R, c_R)$  be the LSTM states maintained by its children nodes, the LSTM state of the non-terminal node is computed as

$$(h, c) = \text{LSTM}([h_L; h_M; h_R], [c_L; c_M; c_R], x) \quad (4)$$

Where  $[a; b]$  denotes the concatenation of vectors  $a$  and  $b$ . For non-terminals with less than 3 children, the corresponding LSTM states are set to be zero. We use  $d$  to represent the size of  $h$  and  $c$ , i.e., the hidden size of the LSTM.

**Input representation.** For each sub-tree  $\omega_i$ , its input to both the score predictor and the rule-picking policy is represented as a  $2d$ -dimensional vector  $[h_0; h_i]$ , where  $h_0$  is the embedding of the root node encoding the entire tree. The reason why we include  $h_0$  in the input is that looking at the sub-tree itself is sometimes insufficient to determine whether it is beneficial to perform the rewriting. For example, consider the expression  $\max(a, b) + 2 < a + 2$ , by looking at the sub-expression  $\max(a, b) + 2$  itself, it does not seem necessary to rewrite it as  $\max(a + 2, b + 2)$ . However, given the entire expression, we can observe that this rewriting is an important step towards the simplification, since the resulted expression  $\max(a + 2, b + 2) < a + 2$  could be reduced to *False*. We have tried other approaches of combining the parent information into the input, but we find that including the embedding of the entire tree is the most efficient way.

**Score predictor.** The score predictor is an  $L_P$ -layer fully connected network with a hidden size of  $N_P$ . For each sub-tree  $\omega_i$ , its input to the score predictor is represented as a  $2d$ -dimensional vector  $[h_0; h_i]$ , where  $h_0$  embeds the entire tree.

**Rule selector.** The rule selector is an  $L_S$ -layer fully connected network with the hidden size  $N_S$ , and its input format is the same as the score predictor. A  $|\mathcal{U}|$ -dimensional softmax layer is used as the output layer.

### C.2 More Details for Job Scheduling Problem

**Job embedding.** We embed each job into a  $(D \times (T_{max} + 1) + 1)$ -dimensional vector  $e_j$ , where  $T_{max}$  is the maximal duration of a job. This vector encodes the information of the job attributes and the machine status during its execution. We describe the details of job embedding as follows. Consider

a job  $v_j = (\rho_j, A_j, T_j)$ . We denote the amount of resources occupied by all jobs at each timestep  $t$  as  $\rho'_t = (\rho'_{t1}, \rho'_{t2}, \dots, \rho'_{tD})$ . Each job  $v_j$  is represented as a  $(D \times (T_{max} + 1) + 1)$ -dimensional vector, where the first  $D$  dimensions of the vector are  $\rho_j$ , representing its resource requirement. The following  $D \times T_j$  dimensions of the vector are the concatenation of  $\rho'_{B_j}, \rho'_{B_j+1}, \dots, \rho'_{B_j+T_j-1}$ , which describes the machine usage during the execution of the job  $v_j$ . When  $T_j < T_{max}$ , the following  $D \times (T_{max} - T_j)$  dimensions are zero. The last dimension of the embedding vector is the slowdown of the job in the current schedule. We denote the embedding of each job  $v_j$  as  $e_j$ . The embedding of the machine (i.e.,  $v_0$ ) is a zero vector  $e_0 = \mathbf{0}$ . Figure 8 shows an example of our job embedding approach, and Figure 9 illustrates an example of the graph construction.

**Model specification.** To encode the graphs, we extend the Child-Sum Tree-LSTM architecture in [46], which is similar to the DAG-structured LSTM in [53]. Specifically, for a job  $v_j$ , suppose  $(h_1, c_1), (h_2, c_2), \dots, (h_p, c_p)$  are the LSTM states of all parents of  $v_j$ , then its LSTM state is:

$$(h, c) = \text{LSTM}\left(\left(\sum_{i=1}^p h_i, \sum_{i=1}^p c_i\right), e_j\right) \quad (5)$$

For each node, the  $d$ -dimensional hidden state  $h$  is used as the embedding for other two components.

**Score predictor.** This component is an  $L_P$ -layer fully connected neural network with a hidden size of  $N_P$ , and the input to the predictor of job  $v_j$  is  $h_j$ .

**Rule selector.** The rewriting rules are equivalent to moving the current job  $v_j$  to be a child of another job  $v_{j'}$  or  $v_0$  in the graph, which means allocating job  $v_j$  after job  $v_{j'}$  finishes or at its arrival time  $A_j$ . Thus, the input to the rule selector not only includes  $h_j$ , but also  $h_{j'}$  of all other  $v_{j'}$  that could be used for rewriting. The rule selector has two modules. The first module is an  $L_S$ -layer fully connected neural network with a hidden size of  $N_S$ . For each job  $v_j$ , let  $N_j$  be the number of jobs that could be the parent of  $v_j$ , and  $\{v_{j'_k}\}$  denotes the set of such jobs. For each  $v_{j'_k}$ , the input is  $[h_j; h_{j'_k}]$ , and this module computes a  $d$ -dimensional vector  $h'_k$  to encode such a pair of jobs. The second module of the rule selector is another  $L_S$ -layer fully connected neural network with a hidden size of  $N_S$ . For this module, the input is a  $(|\mathcal{U}| \times d)$ -dimensional vector  $[h'_1; h'_2; \dots; h'_{|\mathcal{U}|}]$ , where  $|\mathcal{U}| = 2W$ . When  $N_j < |\mathcal{U}|$ ,  $h'_{N_j+1}, h'_{N_j+2}, \dots, h'_{|\mathcal{U}|}$  are set to be zero. The output layer of this module is a  $|\mathcal{U}|$ -dimensional softmax layer, which predicts the probability of each different move of  $v_j$ .

### C.3 More Details for Vehicle Routing Problem

**Node embedding.** We embed each node into a 7-dimensional vector  $e_j$ . This vector encodes the information of the node position, node resource demand, and the current status of the vehicle. We describe the details of node embedding as follows. Consider a node  $v_j = ((x_j, y_j), \delta_j)$ , where  $(x_j, y_j)$  is the position, and  $\delta_j$  is the resource demand. We set  $\delta_0 = 0$  for the depot (i.e., node 0). Denote  $Cap$  as the vehicle capacity. The first three dimensions of  $e_j$  are  $x_j, y_j$ , and  $\delta_j/Cap$ . The next three dimensions of  $e_j$  are the coordinates of the node visited at the previous step (set as the depot position for the first visited node) and the Euclidean distance between  $v_j$  and the previous node. The last dimension is the amount of remaining resources carried by the vehicle at the current step, which is also normalized by the vehicle capacity.

**Score predictor.** This component is an  $L_P$ -layer fully connected neural network with a hidden size of  $N_P$ , and the input to the predictor of the node  $v_j$  is  $h_j$ , where  $h_j$  is the output of the bi-directional LSTM used to encode each node in the route.

**Rule selector.** The rewriting rules are equivalent to moving a node in the route  $v_j$  after another node  $v_{j'}$ , similar to the job scheduling setting. However, different from job scheduling, the number of such nodes  $v_{j'}$  varies among different problems. Thus, we train an attention module to select  $v_{j'}$ , with a similar design to the pointer network [50].



---

**Algorithm 2** Forward Pass Algorithm for the Neural Rewriter during Training

---

**Require:** initial state  $s_0$ , hyper-parameters  $\epsilon, p_c, T_{iter}, T_\omega, T_u$

- 1: **for**  $t = 0 \rightarrow T_{iter} - 1$  **do**
- 2:     **for**  $i = 1 \rightarrow T_\omega$  **do**
- 3:         Sample  $\omega_t \sim \pi_\omega(\omega_t|s_t; \theta)$ , where  $\omega_t \in \Omega(s_t)$ ,  $\pi_\omega(\omega_t|s_t; \theta) = \frac{\exp(Q(s_t, \omega_t; \theta))}{\sum_{\omega_t} \exp(Q(s_t, \omega_t; \theta))}$
- 4:         **if**  $Q(s_t, \omega_t; \theta) < \epsilon$  **then**
- 5:             Re-sample  $\omega'_t \sim \pi_\omega(\omega'_t|s_t; \theta)$  with a probability of  $1 - p_c$
- 6:             **if** Re-sampling is not performed **then break fi**
- 7:         **else**
- 8:             **break**
- 9:         **end if**
- 10:     **end for**
- 11:     **for**  $i = 1 \rightarrow T_u$  **do**
- 12:         Sample  $u_t \sim \pi_u(u_t|s_t[\omega_t]; \phi)$
- 13:         **if**  $u_t$  can be applied to  $s_t[\omega_t]$  **then break fi**
- 14:     **end for**
- 15:     **if**  $u_t$  does not applied to  $s_t[\omega_t]$  **then break fi**
- 16:      $s_{t+1} = f(s_t, \omega_t, u_t)$
- 17: **end for**

---

#### C.4 Model hyper-parameters

For both the expression simplification and job scheduling tasks,  $L_S = L_P = 1$ . For the vehicle routing task,  $L_S = L_P = 2$ . For all the three tasks,  $N_S = N_P = 256$ ,  $d = 512$ .

## D More Details on Training

Algorithm 2 presents the details of the forward pass during training. The forward pass during evaluation is similar, except that we compute  $\omega_t$  and  $u_t$  as  $\omega_t = \arg \max_{\omega} \pi_\omega(\omega|s_t; \theta)$  and  $u_t = \arg \max_u (\pi_u(u|s_t[\omega_t]; \phi))$ , and the inference immediately terminates when  $Q(s_t, \omega_t; \theta) < \epsilon$  or  $u_t$  does not apply.

**Hyper-parameters.** For all tasks in our evaluation, in Algorithm 2,  $\epsilon = 0.0$ ,  $T_\omega = 10$ ,  $T_u = 10$ .  $T_{iter} = 50$  for both the expression simplification and the job scheduling tasks. For the vehicle routing task, we set  $T_{iter} = 200$ , because we find that applying 50 rewriting steps could be insufficient for finding a competitive solution, especially when the number of customer nodes is large. For all tasks in our evaluation,  $p_c$  is initialized with 0.5, and is decayed by 0.8 for every 1000 timesteps until  $p_c = 0.01$ , where it is not decayed anymore. In the training loss function,  $\alpha = 10.0$ . The decay factor for the cumulative reward is  $\gamma = 0.9$ . The initial learning rate is  $1e - 4$ , and is decayed by 0.9 for every 1000 timesteps. Batch size is 128. Gradients with  $L_2$  norm larger than 5.0 are scaled down to have the norm of 5.0. The model is trained using Adam optimizer. All weights are initialized uniformly randomly in  $[-0.1, 0.1]$ .

## E More Results for Job Scheduling Problem

We observe that while OR-tools is a high-performance solver for generic combinatorial optimization problems, it is less effective than both heuristic-based scheduling algorithms and neural network approaches on our job scheduling problem, especially with more resource types. After looking into the schedules computed by OR-tools, we find that they often prioritize long jobs over short jobs, while swapping the scheduling order between them would clearly decrease the job waiting time. On the other hand, both our neural rewriter and heuristic algorithms based on the job length would usually schedule short jobs very soon after their arrival, which results in better schedules.

Table 3 and 4 present the results of ablation study on job frequency and resource distribution respectively.

	Dynamic Job Frequency	Steady Job Frequency
Earliest Job First (EJF)	14.53	24.23
Shortest Job First (SJF)	3.62	5.00
SJF-offline	2.70	4.26
NeuRewriter (dynamic)	<b>2.56</b>	3.99
NeuRewriter (steady)	2.59	<b>3.94</b>

Table 3: Experimental results of the job scheduling problem with different distribution of job frequency.

	Uniform Job Resources	Non-uniform Job Resources
Earliest Job First (EJF)	11.06	24.23
Shortest Job First (SJF)	4.51	5.00
SJF-offline	2.76	4.26
NeuRewriter (uniform)	<b>2.73</b>	4.05
NeuRewriter (non-uniform)	3.13	<b>3.94</b>

Table 4: Experimental results of the job scheduling problem with different distribution of job resources.

Initial average slowdown	$\leq 10$	10 – 25	$> 25$
Final average slowdown	3.88	3.90	4.06
Earliest Job First (EJF)	24.23		
Shortest Job First (SJF)	5.00		
Shortest First Search (SJFS)	4.98		
DeepRM	10.18		
OR-tools	15.18		
SJF-offline	4.26		
NeuRewriter	<b>3.94</b>		

Table 5: Experimental results of the job scheduling problem using initial schedules with different average slowdown. The number of resource types  $D = 20$ .

To examine how the initial schedules affect the final results, besides earliest-job-first schedules, we also evaluate initial schedules with different average slowdown. Specifically, for each job sequence, we generate different initial schedules by randomly allocating one job at a time.

In Table 5, we present the results with  $D = 20$  types of resources. For each job sequence, we randomly generate 10 different initial schedules. We can observe that although the effectiveness of initial schedules affects the final schedules, the performance is still consistently better than other baseline approaches, which demonstrates that our neural rewriter is able to substantially improve the initial solution regardless of its quality.

## F More Discussion of the Evaluation on Vehicle Routing Problem

We generate the initial routes for `NeuRewriter` in the following way: starting from the depot, at each timestep, the vehicle visits the nearest node that is either: (1) a customer node that has not been visited yet, and its resource demand can be satisfied; or (2) the depot node, and the resources carried by the vehicle is less than its capacity. See Figure 10 for examples of the initial solutions. In this way, the average tour length is 7.74 for VRP20, 13.47 for VRP50, and 20.36 for VRP100. Note that these results are even worse than the classic heuristics compared in Table 6.

Table 6 presents more results for vehicle routing problems, and Figure 10 shows an example of the rewriting steps performed by `NeuRewriter`.

For generalization results, note that after training on VRP50, `NeuRewriter` achieves an average tour length of 17.33 on VRP100 (See Figure 5b in the mainbody of the paper). This is better than 18.00 reported in [35], suggesting that our approach could adapt better to different problem distributions.

Model	VRP20, Cap30	VRP50, Cap40	VRP100, Cap50
NeuRewriter	<b>6.16</b>	<b>10.51</b>	<b>16.10</b>
AM-Greedy	6.40	10.98	16.80
AM-Sampling	6.25	10.62	16.23
Nazari et al. (RL-Greedy)	6.59	11.39	17.23
Nazari et al. (RL-BS(5))	6.45	11.22	17.04
Nazari et al. (RL-BS(10))	6.40	11.15	16.96
CW-Greedy	7.22	12.85	19.72
CW-Rnd(5,5)	6.89	12.35	19.09
CW-Rnd(10,10)	6.81	12.25	18.96
SW-Basic	7.59	13.61	21.01
SW-Rnd(5)	7.17	13.09	20.47
SW-Rnd(10)	7.08	12.96	20.33
OR-Tools	6.43	11.31	17.16
Gurobi (optimal)	6.10	-	-

Table 6: Experimental results of the vehicle routing problems.

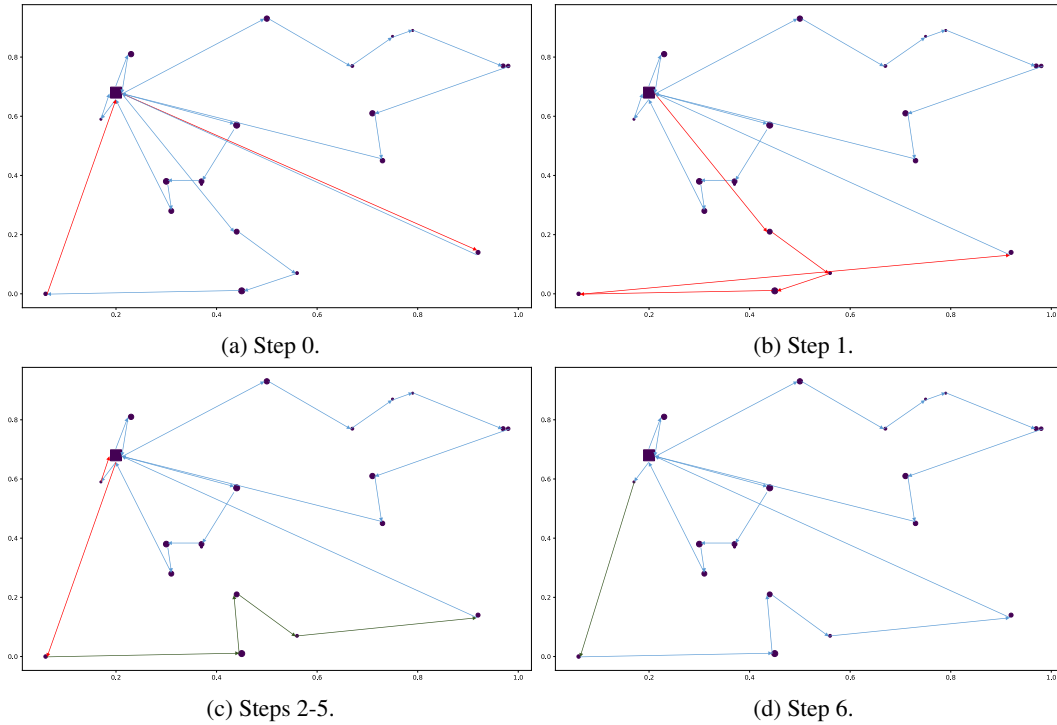


Figure 10: An example of the rewriting steps for a VRP20 problem. The square is the depot, and circles are customer nodes. The customer node sizes are proportional to their resource demands. At each stage, red edges are to be rewritten at the next step, and green edges are rewritten ones. The tour length of the initial route is 7.31, and the final tour length after rewriting is 5.98.

## G More Results for Expression Simplification

In Figures 11 and 12, we present some success cases of expression simplification, where we can simplify better than both the Halide rule-based rewriter and the Z3 solver.

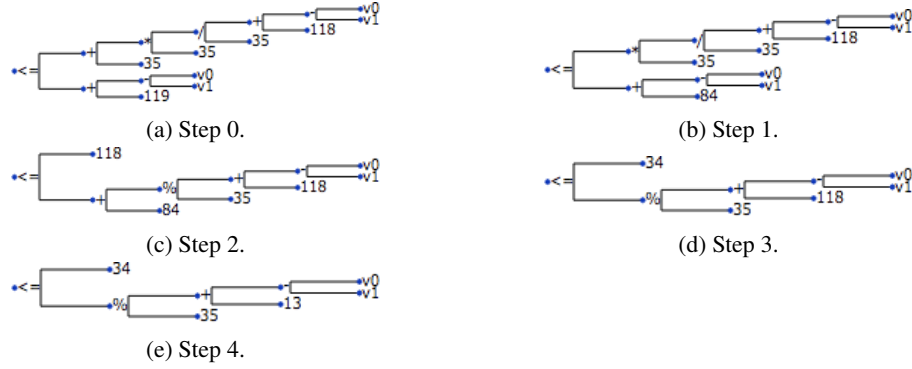


Figure 11: The rewriting process that simplifies the expression  $((v_0 - v_1 + 18)/35 * 35 + 35) \leq v_0 - v_1 + 119$  to  $34 \leq (v_0 - v_1 + 13)\%35$ .

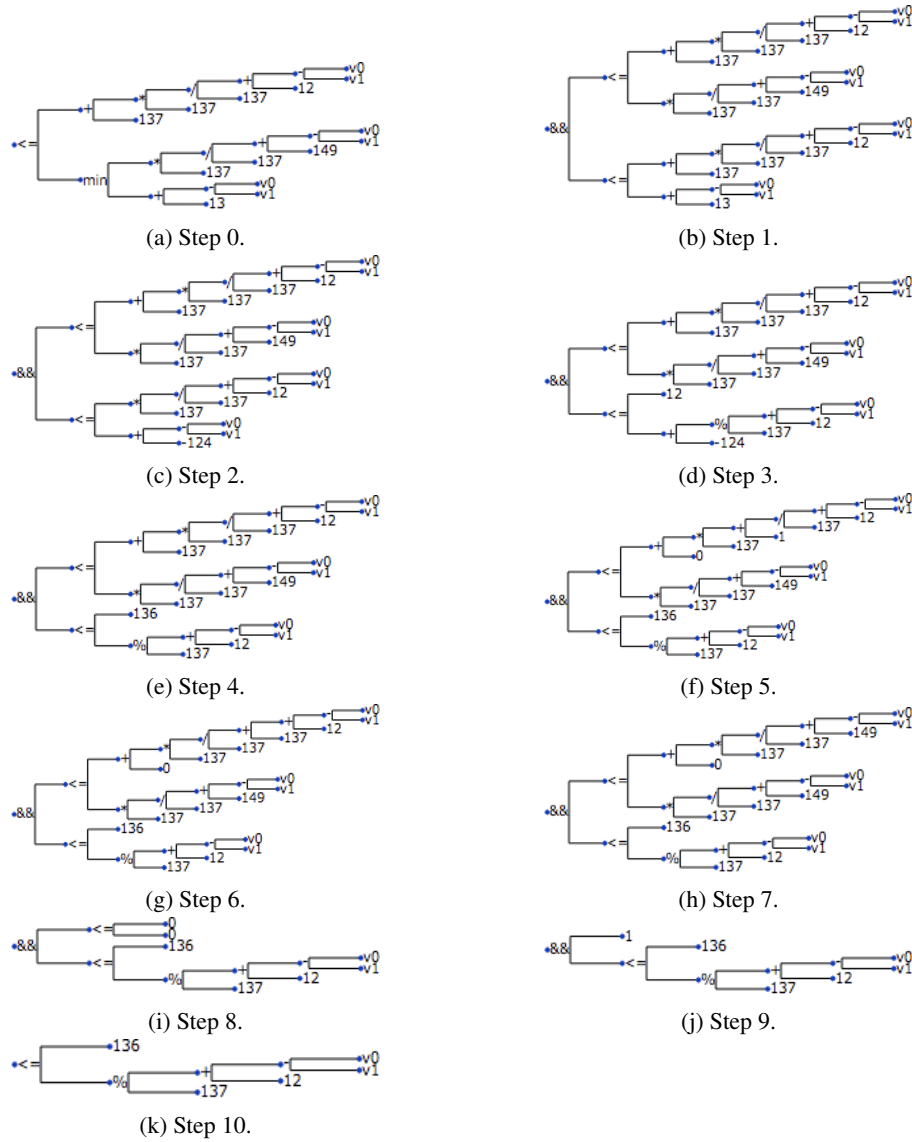


Figure 12: The rewriting process that simplifies the expression  $((v_0 - v_1 + 12)/137 * 137 + 137) \leq \min((v_0 - v_1 + 149)/137 * 137, v_0 - v_1 + 13)$  to  $136 \leq (v_0 - v_1 + 12)\%137$ .