UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Deep Reinforcement Learning for Equality Saturation

Zak Singh

Clare Hall

June 2022

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 45

Main chapters (excluding front-matter, references and appendix): 41 pages (pp 1–41)

Main chapters word count: 14,848

Methodology used to generate that word count:

```
$ make wordcount
gs -q -dSAFER -sDEVICE=txtwrite -o - \
   -dFirstPage=1 -dLastPage=41 main.pdf | \
egrep '[A-Za-z]{3}' | wc -w
```

Abstract

An essential part of modern compilation systems is *term rewriting*, which translates the user-specified program into an equivalent format which is faster to execute. For example, many compilers will replace instances of $x \times 2$ with the faster bit-shift operation $x \ll 1$. Typically, rewrites are applied according to a pre-defined ordering created by the compiler designer; in such a system, each rewrite operates on the output of its predecessor. This is often problematic, as rewrites are *destructive*, meaning that a poorly chosen rewrite rule can prevent future rules from finding better optimizations. In other words, the order in which rewrites are applied affects the final outcome; this is known as the *phase ordering* problem.

Equality Saturation is an algorithm for term rewriting which aims to solve this problem by simultaneously exploring all possible variants of a program that can be derived from the set of rewrite rules before extracting the best one. This is achieved by building an equivalence graph (*e-graph*) data structure which can efficiently encode many equivalent programs into a single graph. As each rewrite is applied, nodes can only be added to the graph, creating new possible equivalent programs; once the application of rules no longer modifies the e-graph, it is said to have reached *saturation* as it encodes every possible program representation, and terminates. While equality saturation is promising, and has been proven on a number of domains ranging from CAD design [28] to neural network optimization [48], challenges remain.

Namely, we find the saturation condition to be an unfeasible stopping point in even simple tasks, as the e-graph, while efficient, may expand indefinitely under a variety of circumstances. This can prevent equality saturation from finding good solutions, as memory limits are reached before useful rule sequences can be applied. Because of this, we propose *Omelette,* a Reinforcement Learning system which learns rewrite rule application orderings to intelligently build the e-graph such that superior solutions can be found compared to modern approaches.

We demonstrate that Omelette can improve solution quality by up to 100% over the current state-of-the-art equality saturation solver across both propositional logic and mathematical simplification tasks (chosen to represent both symbolic and numerical domains), and can do so while staying within the bounds of tight e-graph memory restrictions.

Acknowledgements

This project would not have been possible without the wonderful support and patience of my supervisor, Eiko Yoneki, and co-supervisor, Sami Alabed, throughout the development of this work and its many ups-and-downs.

Contents

List of Figures

Chapter 1

Introduction

Traditional compilation systems apply optimizations sequentially, with each optimization operating on the output of its predecessor. A major drawback of this approach is that the quality of the generated code is dependent on the order in which these optimizations are applied. This issue, known as the *phase ordering problem*, causes compilers to produce sub-optimal programs when an optimization earlier in the chain inadvertently eliminates the opportunity for a superior optimization in a later pass. Compiler designers must therefore carefully tune the ordering of optimizations to maximize performance.

To address this challenge, Tate et al. [40] propose *Equality Saturation*, an algorithm which simultaneously computes all possible optimized versions of the input program before selecting the best candidate, thereby eliminating the threat of phase ordering altogether. To enable representing potentially millions of equivalent programs within memory, equality saturation relies on the use of the E-Graph data structure [30] which leverages the notion of equivalence classes to merge and de-duplicate shared terms between programs. However, despite its promise, equality saturation's practical use has been limited by its implementation complexity and high computational demands.

The recent *egg (e-graphs good)* [45] equality saturation framework has partially addressed these challenges by providing much-needed performance optimizations to the internals of the equality saturation algorithm. In doing so, it has enabled its use in domains ranging from linear algebra kernels [43] and deep learning computation graphs [48] to 3D CAD programs [28], greatly improving over traditional optimization systems and proving the efficacy of equality saturation's core approach. Challenges remain, however, as many of these works have needed to implement complex domain-specific mechanisms to further work around the nuances of equality saturation to make their problem domains solvable under finite memory and time constraints. If a *general* solution to these obstacles could be incorporated into the equality saturation algorithm, it could theoretically be applied to many more domains without the need for significant adaptation.

In this work, we have sought to first explore the *theoretical* and *practical* limitations which

currently inhibit the success of equality saturation. In doing so, we observe that equality saturation can unexpectedly fail in scenarios where a destructive rewrite system would readily succeed due to the unique properties of the e-graph data structure. Following this, we observe that equality saturation could be greatly improved by directing its search through the space of possible rewrites in order to reach lower cost solutions within the finite memory space allocated to the algorithm.

To accomplish this, we propose the use of Reinforcement Learning (RL) to guide the application of rewrite rules to avoid rapid E-Graph expansion while simultaneously ensuring that the rules applied enable the optimal equivalent program to be found. While RL has previously been used to select rewrite rules in traditional compiler pipelines [8], the unique properties of the E-Graph structure enable interesting design opportunities and have the potential to make RL a more effective solution for this task. The exact contributions of this work are enumerated below:

1. The identification of theoretical and practical limitations of equality saturation which restrict the performance of state-of-the-art solvers.
2. The first formulation of equality saturation as a Reinforcement Learning task.
3. We present *Omelette*, an RL agent flexible enough to solve arbitrary equality saturation tasks given only a language and a set of rewrite rules. To demonstrate this ability, we evaluate Omelette upon both symbolic and numerical domains in the form of propositional logic and equation simplification in order to test the agent's flexibility.
4. An extensive comparison between Omelette and the state-of-the-art equality saturation solver, egg, across these two domains. In doing so, we show that Omelette is strictly superior to the egg solver in terms of final solution quality on the domains tested; in some cases, solutions can be improved by up to 100% compared to egg within the same e-graph memory constraints.

Chapter 2

Background and Related Work

The purpose of this chapter is to provide the background in equality saturation and reinforcement learning necessary to contextualize our proposed solution, *Omelette*. We begin with the underlying theory of equality saturation and RL before progressing to the design of the current state-of-the-art equality saturation solver, egg. Finally, we discuss related work in applying RL to graph optimization tasks.

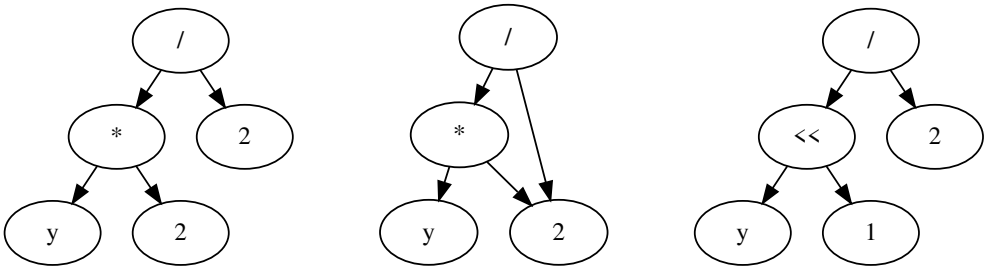
2.1 Equality Saturation Overview

Equality saturation emerged from the inherent ordering-based complexities of code optimization. Informally, it is best understood as an attempt to search over *all* potential optimization orderings at once in a more efficient manner than exhaustive enumeration. This section presents the root problem addressed by equality saturation as well as the benefits underlying this technique.

2.1.1 Term Rewriting

Term rewriting [7] refers to a process by which expressions (known as *terms*) are transformed according to a set of *rules* into alternative representations which are beneficial by some cost metric. For example, the simplification of a mathematical expression can be thought of as a term rewriting task which many are familiar with. Just as simplifying an equation makes it easier to interpret by a human, term rewriting is used at the heart of compilers [15], logic programming [22], proof assistants [10], and computer algebra systems [6] to improve efficiency for machines. However, unlike shortening equations, rewriting computer programs generally requires optimizing a more complex metric related to code execution time.

Beginning with a term t and a set of binary *rewrite relations* $R = \{l \to r\}$, the optimizer searches for the *pattern* l in t, producing a list of *matches* σ which map variables in l to



(a) AST (b) Term graph (c) Term graph after rewrite

Figure 2.1: AST and term graph representations of the expression $(y \times 2)/2$. Note how the term graph is able to share the subterm 2.

subterms of t. The values of σ are then substituted into r, denoted as $r\,[\sigma]$. To illustrate this process, consider the following problem:

$$\begin{aligned} t &= (y \times 2)/2 \\ R &= \{(x \times 2) \to (x \ll 1)\} \end{aligned} \tag{2.1}$$

First, the left-hand side of the singular rule we've defined is searched for within t, yielding a match upon the subterm $(y \times 2)$. The right hand side of the rewrite rule's values are concretized with $\sigma = \{x : 2\}$, yielding $r\,[\sigma] = y \ll 1$, which is then substituted into t to form a revised $t' = (y \ll 2)/2$.

This rewrite rule can be found in many compilers, and is known as a *strength reduction*, a type of optimization in which an expensive operation is replaced with an equivalent yet less expensive one. However, this example illustrates a key challenge with term rewriting systems: the order of rule application matters, as each substitution is *destructive*. Considering our prior example, a superior optimization would be to "cancel out" the multiplication and division by two through *constant folding*. However, the rewrite we performed has hidden this opportunity by introducing a bit-shift.

This is a classic example of the *phase ordering* problem in compiler literature, in which the order of optimizations must be carefully considered in order to prevent optimization passes from interfering by obscuring each other's optimization opportunities. A significant corpus of work treats phase ordering as a combinatorial optimization problem, and seeks to solve it through methods such as exhaustive search via backtracking, genetic algorithms [16], integer linear programming [17], and reinforcement learning [8], with varying success. Nonetheless, phase ordering remains a crucial challenge in term rewriting systems.

Term Graphs

Term rewriting systems most commonly represent expressions as a graph data structure, known as a *term graph*, rather than the more familiar Abstract Syntax Tree (AST). Unlike

ASTs, term graphs can share sub-expressions between nodes. On the other hand, each node in an AST can have only one parent, resulting in the redundant computation of identical terms. Figure 2.1 highlights this difference.

Termination and Confluence

When constructing a term rewriting system, it is surprisingly easy to define rules which enable themselves, creating recursive rule applications which can be repeated indefinitely. According to Dershowitz and Jouannaud [7], rewrite systems—as successors to lambda calculus—are known to be Turing-complete in their most general form. Thus, they inherent their predecessors potential for non-termination as well as several key concepts which we shall now discuss.

Formally, an object l is said to be *reducible* if there exists some r in R such that $l \to r$; otherwise l is said to be in *normal form*. A rewriting system is *terminating* if there exists no infinite chain $t_0 \to t_1 \to t_2 \to \cdots$. For a system to be terminating, every object must have at least one normal form. A rewriting systems is said to be *confluent* if for a term with multiple reductions ($M \to M_1 \land M \to M_2 \ldots$), all produced terms are guaranteed to *eventually* converge to an identical term following a series of further reductions ($M_1 \to^* M_c \land M_2 \to^* M_c \ldots$).

Importantly, if a term rewriting system is confluent and terminating, it can be used to to prove equality between expressions. Taking two expressions s and t, apply equalities from left to right as long as possible on each term to reach s' and t', which, if s and t are equivalent, will also be equal. Crucially, this method does not depend on the order in which rewrite rules are applied; confluence ensures that every sequence will lead to the same result.

Term Rewriting and Formal Grammars

By labelling terms as *terminal* or *nonterminal*, a link between rewriting systems and formal grammars can be established. A formal grammar can be thought of as a special case of rewriting system where the alphabet is divided into terminal and nonterminal symbols, and all inference rules are local and involve a nonterminal on the LHS. The computation starts from a specific initial nonterminal and halts when the word contains only terminals. The set of possible resulting words is the language defined by the grammar. Thus, given this correspondence we can talk about the *language* defined by a given term rewriting system as the set of words which may be generated by its equivalent grammar.

2.1.2 E-Graphs

An *equality graph* (e-graph) is a data structure capable of efficiently encoding a congruence relation over many expressions [30]. Formally, an e-graph contains a set of equivalence

classes (*e-classes*), with each containing a set of equivalent *e-nodes*. Each e-node consists of an operator (such as $\times$, $\ll$, or terminals like 2) and edges connect e-nodes to their e-class children. Of course, the relevance of such an e-graph to the term rewriting problem depends on what exactly is meant by *equivalent*. Formally, two nodes are said to be equivalent if they satisfy an equivalence relation—a binary, reflexive, symmetric, and transitive relation according to Rosen [34, sec. 9.5]—defined for the term rewriting system. For the rest of this work, two e-nodes will be considered equivalent if the substitution of either one with the other does not change the meaning of the expression they are constituents of. In the case of compilers, this would imply that the compiled code produces the same output regardless of which node in an e-class is used.

Notice how an e-graph in which each e-class is a singleton (Figure 2.2a) is functionally identical to a term graph (Figure 2.1a) and has the same properties. Thus, an e-graph brings the same advantage of being able to contain cycles over standard Abstract Syntax Trees. Therefore, they are able to represent infinitely many expressions, such as $(a \times 1)$, $(a \times 1 \times 1)$, and so on, all within a fixed memory space.

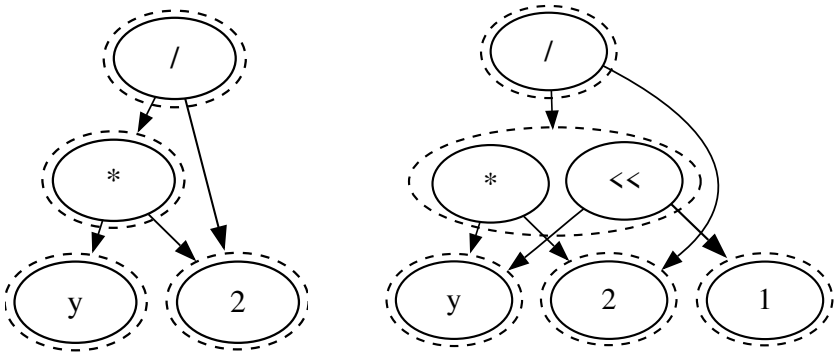
Although originally developed for use in automated theorem provers (ATPs), recent work has utilized e-graphs to implement term rewriting systems which address the phase-ordering problem through a method known as *equality saturation* [40]. As we will see, equality saturation can greatly reduce the combinatorial explosion of rewrite sequences seen with classic term rewriting systems.

2.1.3 Equality Saturation

Having previously discussed the phase-ordering problem, we can now consider how to solve it. A brute-force method would involve generating all possible sub-expressions repeatedly in a breadth-first manner with memorization of previously seen expressions until no new version is generated. This process suffers from combinatorial explosion, however, it serves as the basis upon which equality saturation, shown in Listing 2.1.3, improves upon.

```
1  def equality_saturation(expr, rewrites):
2      egraph = initial_egraph(expr)
3
4      while not egraph.is_saturated_or_timeout():
5          for rw in rewrites:
6              for (subst, eclass) in egraph.match(rw.lhs):
7                  eclass2 = egraph.add(rw.rhs.subst(subst))
8                  egraph.merge(eclass, eclass2)
9
10     return egraph.extract_best()
```

Listing 2.1: The Equality Saturation algorithm.



(a) E-Graph (b) E-Graph after rewrite

Figure 2.2: The same rewrite rule applied to an E-Graph. Dotted lines represent e-classes, and edges connects e-nodes to their e-class children. Notice that applying the rewrite rule $x \times 2 \rightarrow x \ll 1$ is purely *additive*; no information is lost, thus eliminating the phase-ordering problem.

Equality saturation receives an input program p, and constructs an e-graph E that represents a set of expressions equivalent to p, before extracting the "best" program from E according to a defined cost metric (typically performance). To construct the e-graph, pattern-based rewrite rules are repeatedly applied. Notably, these rewrites are exclusively additive; they can never remove information from the e-graph. Theoretically, this property eliminates the need for the careful ordering of rules seen in current compilers, as any sequence will result in the same e-graph. Equality saturation thus enables e-graphs to represent an exponential number of equivalent programs simultaneously in a memory-efficient manner.

Upon reaching *saturation*, the point at which applying additional rewrite rules has no further effect on the e-graph, it can be said that E represents *every equivalent form* of p with regards to the rewrite rules available. Consider the commutative rewrite $x+y \rightarrow y+x$. After applying it once, the re-applying it results in no new information being added to the e-graph, as it already contains the x, $+$, and y terms, as well as the edges between them.

Extraction

After the e-graph has been saturated, an *extraction* procedure selects the optimal equivalent term according to a cost function. While a bottom-up greedy traversal of the e-graph suffices for simple cost functions, more complex domains have necessitated the use of Integer Linear Programs (ILPs) and genetic algorithms to extract the best solution [45].

Due to this extraction procedure, an E-Graph can also be seen as an AND-OR Directed Acyclic Graph (DAG) over expressions. Each E-Class is an OR node whose children are equivalent expressions from which the optimizer chooses from. Each operator is an AND node whose children must all be picked if the operator itself is picked. We favor the terms E-Graph and E-Class to emphasize that each OR node represents an equivalence class.

2.2 Reinforcement Learning

Reinforcement Learning (RL) [39] refers to a body of techniques used in optimizing sequential decision making processes. A common formalism for RL tasks is the Markov Decision Process (MDP), which is defined as a 4-tuple (S, A, R, T), which constitutes:

States. S is the *state space*. For example, in a chess environment, the state space would be every possible configuration of pieces on the board.

Actions. A is the set of actions available at each state.

Rewards. R is a reward function $R(s, a)$, mapping each state-action pair to a real value used to inform agent behavior by incentivizing good actions while punishing negative ones.

Transition Function. T is a transition function $T : S \times A \to S$, specifying how applying an action at a given state transitions the environment to a new state.

Notably, MDPs assume the *Markov Property* holds true, meaning that the effects of an action taken in a state depend only on that state and not any prior history.

The goal of reinforcement learning is to learn a policy π which maps each element of S to an action in A in order to maximize the total reward accumulated by an agent throughout each *episode*, a sequence of states and actions between an initial and terminal state.

While an agent seeks reward, if it simply chose the highest reward action at each state, it would constitute no more than a greedy search, missing potential better outcomes which require transitioning through lower reward states to reach a higher one. Therefore an RL agent does not optimize for reward directly, but rather for *expected value*, the discounted sum of all future rewards presuming the agent follows an optimal policy from the current state onwards. The standard way to model this value balances reward for the current state with the value of future states,

$$\begin{aligned} G_t &= R_{t+1} + \gamma G_{t+1} \\ 0 &\leq \gamma < 1, \end{aligned} \tag{2.2}$$

where G_t is the current state value, R_{t+1} is the reward given by the next action, γ is the discount factor, and G_{t+1} is the future-state value defined recursively. From this recursive formulation it can be seen how the agent chooses to prioritize the current state compared to its future according to the discount factor γ. The geometric fall-off of γ is not coincidental as it guarantees that the infinite series defined by Eq. (2.2) converges to a finite number for $\gamma \in [0, 1)$.

$$\lim_{n\to\infty} \sum_{k=0}^{n} \gamma^k R_{t+k+1} = \frac{R^t}{1-\gamma} \tag{2.3}$$

2.2.1 Approximate Solution Methods

While an MDP can be solved to find the optimal policy through the use of dynamic programming to evaluate every possible action sequence, this is not practical in environments with large state spaces. For example, the number of possible camera images is larger than the number of atoms in the universe; in such cases the optimal policy or value function cannot be found even in the limit of infinite time and data [39].

Furthermore, in such tasks it is likely that many states encountered by the agent will never have been seen before; therefore to make sensible decisions the agent must generalize from prior encounters with similar states. In other words, experience with a limited subset of the state space must be generalized to approximate over a much larger space. To accomplish this, *neural networks* may be used to approximate the policy and value functions.

However, the use of deep learning techniques bring with them new challenges when applied to RL; for example, supervised learning typically relies on the concept of *independent and identically distributed (i.i.d)* data, which assumes that training data is sampled from the same distribution. In an RL task, we have no way of ensuring this, as training data is limited to whatever states the agent has experienced through its actions, which are likely to be focused in certain sectors of the state space; this heightens the risk of the agent getting stuck in local optima.

2.2.2 Proximal Policy Optimization

Proximal Policy Optimization [36] is a policy-gradient RL algorithm which learns a stochastic policy over the environment. Notably, PPO is an example of an *actor critic* model, which means it is constituted of two separate networks: a *actor* policy network mapping observations to actions, and a *critic* network mapping observations to predicted values.

PPO's primary innovation is its *stability*; building off Trust Region Policy Optimization (TRPO) [35], PPO ensures that the agent's policy evolves smoothly over the course of training by *clipping* policy updates which are too drastic; this is a crucial innovation, as not doing so results in *destructive policy updates* in which an overly-large policy update can shift the agent into a worse portion of the search space which it can then never escape from, ruining the training run altogether. By constraining policy adjustments, PPO ensures the agent can recover from bad states throughout its training.

This recoverability property makes PPO an excellent candidate for usage in novel domains where RL has never been attempted before and environment dynamics are unknown. Therefore, while PPO is no longer the state-of-the-art in terms of absolute performance [9], it remains well-suited for novel RL applications such as ours.

2.3 Related Work

This section briefly describes relevant works which have influenced the overall design of Omelette. We begin with a presentation and discussion of egg [45], the current state-of-the-art equality saturation solver, as it serves as the primary benchmark of comparison for this work. This is followed by an overview of the nascent field of Graph Reinforcement Learning (GRL) and its applications to the related topic of combinatorial optimization.

2.3.1 The egg Equality Saturation Framework

The recent *egg (e-graphs good)* [45] framework has revitalized the field of equality saturation by providing a modern and well-optimized toolkit for its usage across different domains. As egg has become synonymous with equality saturation, and is currently the only mature library available, it is used as the basis for this work and is the primary means of comparison for our proposed method.

Willsey et al. [45] motivate the design of egg with two goals in mind. The first is constructing a domain-independent equality saturation system which allows *domain-specific extensions* to be injected with little modification. This was necessary as several previous works [31, 40, 46] had to re-implement equality saturation in its entirety due to lacking such an ability. The second objective of egg is to make equality saturation feasible by tremendously increasing performance via a novel technique known as *rebuilding*. Rebuilding is an amortized means of restoring e-graph invariants between iterations of the loop shown in Listing 2.1.3 in a manner capable of providing *asymptotic* speed improvements over previous approaches.

While the details of these additions are beyond the scope of this work, their benefits were sufficient to allow egg to become the de-facto standard equality saturation system. Willsey et al. [45] show that egg provides several orders of magnitude of improvement over the previous work of Panchekha et al. [31] on floating point rewrite optimizations. Furthermore, they show egg yields a $1000x$ speed up to the CAD deconstruction system of Nandi et al. [27].

Discussion Egg is a significant improvement over previous equality saturation solvers in terms of both efficiency and flexibility. However, its implementation requires several ad-hoc solutions to memory constraints. The most relevant of these are the addition of a node limit and a timeout. Both of these changes to the theoretical equality saturation algorithm have ramifications in terms of the execution speed of egg and its ability to find optimal solutions. These limitations will be further discussed in the following chapter.

2.3.2 Graph Reinforcement Learning

Recent lines of work have explored the use of RL for combinatorial optimization tasks, particularly on graph domains. These include the Traveling Salesman Problem, the Maximum Cut Problem, and the Bin Packing Problem (see Mazyavkina et al. [21] for an overview). While these methods are not directly applicable to an equality saturation-based RL solution operating on E-Graphs, they can provide insight into the strengths and weaknesses of RL when exploring large graph state spaces.

A combinatorial optimization problem is a *discrete* optimization problem defined over a set of elements V with a cost function $f : V \to R$. While the set V may be finite, many combinatorial optimization problems have exceptionally large state spaces depending on initial parameters, with no known means of efficiently finding the global optimum. According to Mazyavkina et al. [21], RL has been applied most frequently to the Traveling Salesman Problem (TSP) and the Capacitated Vehicle Routing Problem (CVRP). In the case of the CVRP, the RL design of Lu et al. [19] was shown to outperform all baselines considered by Mazyavkina et al. [21]. This success is not universal, however, it does serve as evidence for the overall ability of RL to handle complex optimization problems on graphs, providing validation for our proposed agent which operates upon E-Graphs.

Mazyavkina et al. [21] also indicate several limitations of the previous applications of RL to combinatorial optimization. The first of these is that learning based solutions only outperform Google's OR-Tools [32] Linear Programming solver for small TSP graphs; this suggests that RL struggles with the larger action spaces which occur when scaling up the graph. The second relevant limitation is that the training times of RL methods are much longer than algorithmic approaches, undermining its real-world practicality.

Of particular interest is the graph-based nature of these combinatorial optimization problems, as they serve as one of the primary applications of Graph Reinforcement Learning (GRL). According to the GRL survey of Mingshuo et al. [24], graph data is particularly difficult to handle in ML generally and RL in particular. The primary causes of this challenge are the overall scale, discrete nature, and diversity of graph data. To handle this, GRL employs Graph Neural Networks, described in the next section.

Graph Neural Networks

Traditional Deep Learning methods such as Multi-Layer Perceptrons (MLPs) struggle with graph data, as graphs features complex relationships not expressible in Euclidean space. Graph Neural Networks (GNNs) [50] seek to address this problem through architectures designed specifically for graphs by utilizing relationships *between nodes* in their predictions. MLPs, on the other hand, have no means of taking advantage of this *structural* information which can be crucial to many graph-related tasks.

While simple GNN architectures such as Graph Convolutional Networks (GCNs) [14]

convolve each node's feature vector with those of its neighbors evenly, this method is limited in expressivity, as nodes which have identical neighborhood structures will be mapped to identical embeddings, resulting in a loss of higher-level graph information.

Graph Attentional Networks (GATs) [42], used in this work, improve upon GCNs by weighing each neighboring node into the final node embedding using attention mechanisms. This allows specific *features* of neighboring nodes to be weighed independently into a node's embedding, providing finer-grained control of each node's representation and thereby increasing expressivity.

While many GNN tasks perform classification or regression on a per-node basis, this thesis involves using GNNs to make *graph-level* classifications. To enable this type of higher-level prediction, *pooling* layers are typically used to combine individual learned node representations into a single value [23]. Popular examples of this method include global average pooling and sum pooling; the trade-offs of these approaches will be discussed in the design of our agent.

Chapter 3

Omelette: Deep RL For Equality Saturation

3.1 Overview

This section begins with an exploration of the theoretical limitations of equality saturation by formulating scenarios in which it can be shown to unexpectedly never terminate where a traditional term rewriting system would have, thus providing valuable insights for those considering the use of equality saturation in their domains. Following this, the practical limitations of current equality saturation systems which must operate within finite memory and time requirements are discussed, providing solver designers with new considerations on how the trade-offs they implement may be improved. Crucially, we identify that both the theoretical and practical concerns of equality saturation arise from the fact that *saturation* is not an achievable termination condition in real-world scenarios, and that therefore *phase ordering* is re-introduced at the e-graph construction level. By not accounting for this, equality saturation leaves significant performance opportunities on the table.

Following this discussion of the limitations of existing equality saturation techniques, we present *Omelette*, a novel Reinforcement Learning approach to equality saturation which seeks to address these challenges in a domain-agnostic way. Unlike classical equality saturation methods which exhaustively apply all available rewrite rules in every iteration until saturation, Omelette discards the saturation objective and is therefore capable of both choosing *which* rules to run and *prioritizing* them in order to avoid non-termination or explosive growth of the e-graph. On this basis, Omelette is postulated to better account for the aforementioned theoretical and practical limitations of equality saturation than existing solvers, while simultaneously improving upon already saturable tasks by reducing the number of rewrite rule applications necessary to find the optimal solution.

In describing Omelette's design, we provide the first-ever formulation of equality saturation as an RL task before presenting a collection of hypotheses with regards to its expected behavior. Finally, the criteria by which we will evaluate the agent are discussed.

3.2 Limitations of Equality Saturation

As previously mentioned, both the theoretical and practical limitations of equality saturation arise from the unexpected re-emergence of the phase ordering problem within the realm of e-graph construction. However, while the challenges presented are related, there is an important distinction to be made: the theoretical limitations described apply to an equality saturation solver even with *unlimited* computation time and memory space, while the practical limitations discussed are caused by the methods used by existing e-graph solvers to handle *finite* memory and time constraints *even in scenarios in which equality saturation will eventually terminate.*

3.2.1 Theoretical Limitations

While the difference between equality saturation and term rewriting with regards to their ability to terminate remains an open research question, we will provide our observations in this section to illuminate scenarios where their behaviors diverge unexpectedly. The critical question is to identify why *reasonable* rule sets can lead to the unbounded expansion of the e-graph and therefore non-termination during equality saturation.

As term rewriting systems may be as powerful as Lambda calculus, they inherit not only its non-termination potential but also the inherent difficulty of *predicting* non-termination, as codified in the Halting Problem [38]. As such, this work is not concerned with non-termination as a phenomenon generally present in all rewriting systems. Instead, we highlight scenarios in which a non-terminating rewrite system which *terminates* under a given rewrite ordering using destructive term rewriting methods fails to do so when applied with equality saturation. These situations are unintuitive and not easily predicted, and form a core yet under-discussed pitfall of equality saturation.

Building on the work of Willsey et al. [44], we present a scenario which demonstrates that equality saturation will never terminate even when using a rewrite order which terminates in an equivalent destructive rewrite system; this is due to the fact that the equality saturation produces an unbounded number of e-classes which expand the e-graph indefinitely and prevent termination under all circumstances. To begin, consider the following set of rewrite rules:

$$\begin{aligned} &inc\colon h(g(x,Z)) \to h(g(f(x),Z)) \\ &dec\colon g(f(x),y) \to g(x,f(y)) \\ &clear\colon g(Z,y) \to Z, \end{aligned} \tag{3.1}$$

where Z is a terminal and x and y are variables. This set of rules resembles counting in Lambda calculus with Z representing zero and levels of applications of f upon it being similar to Church numerals such that $f(Z)$ can be interpreted as 1 and $f(f(Z))$ as 2.

Using this allegory, the first rule, *inc*, increments the first parameter x of g by one each time it is applied, with the system adding this higher increment term to the e-graph as a new e-class. Meanwhile, the second rule counts down by eliminating one application of f from x and incrementing the second parameter of g by one, thereby keeping the term in the e-graph. Finally, the last rule detects when the system has successfully counted the first argument of g to zero and reduces the entire term to the terminal Z. Therefore, in a standard term rewriting system in which *inc* has been applied i times, it takes i applications of *dec* followed by a single application of *clear* to terminate. For example, the following equation displays an input expression and series of rules applied to it which results in a terminal where no rules are applicable:

$$h(g(Z, Z)) \implies inc \to inc \to dec \to dec \to clear \implies Z \tag{3.2}$$

However, given this same input and rule application sequence, we can demonstrate non-termination in equality saturation as it is unexpectedly able to continue applying *inc* even once the terminal Z has been proved equivalent to the input expression, thereby infinitely expanding the e-graph and preventing saturation (and therefore termination) from being reached. This is because the original expression remains present within the e-graph (and therefore matchable for *inc*) even once it has been proven equivalent and merged into the same e-class as Z, even though we intuitively know that any additional applications of *inc* are ultimately equivalent to Z. As the saturation condition required for termination requires a complete application of all rules throughout which the e-graph remains static, the equality saturation algorithm will *never terminate* in this scenario even though a regular, destructive term rewriting system is guaranteed to. In other words, it can be said that the need to apply all rules in equality saturation makes termination impossible in this domain; for the algorithm to terminate, it would be necessary to *stop* applying *inc* altogether.

Summary. In this section wwe have illustrated a theoretical limitation of equality saturation by demonstrating that its use of the e-graph data structure results in non-termination even under rewrite sequences which terminates under traditional, destructive term rewriting. To solve this problem requires reorienting equality saturation around an objective other than saturation; this serves as a key justification for the design of our proposed RL solution, *Omelette*, presented later in this chapter.

3.2.2 Practical Limitations

While the previous section described scenarios in which equality saturation will never terminate, this section aims to demonstrate that rule ordering *does* have a great effect on the *efficiency* and *practicality* of equality saturation in real-world conditions under finite memory and time constraints. This is because, as we will show throughout this work, even simple tasks which are trivial through other optimization techniques have a tendency to exceed the limits of equality saturation by growing the e-graph rapidly until a node or time limit is reached before the best solution can be found. In this section, we will present a simple scenario in which the optimal solution to a task is unreachable by equality saturation as a memory limit is reached before it can be added to the e-graph. Following this, we demonstrate how a different ordering of rule applications can naturally solve this problem, highlighting the need for intelligent rewrite rule sequencing.

Intuitively, an optimal equality saturation algorithm would apply rewrite rules in the sequence which most quickly builds an e-graph which contains the optimal reduced expression. However, equality saturation as proposed by [40] simply applies every rewrite rule in a loop until saturation or a memory limit is reached, offering no direction to e-graph growth. While modern equality saturation solvers such as egg now offer exponential back-off on rules which expand the e-graph too quickly to improve fairness and enable other rules to have a chance to execute before memory or time limits are exceeded, the solution is not fully sufficient as demonstrated by the many subsequent applications of egg which have needed to devise domain-specific methods to address this problem independently [29, 45, 48].

Equality saturation is inherently computationally expensive as it must simultaneously evaluate every possible equivalent representation of a target program. While the use of the e-graph data structure drastically reduces the memory needed for this approach, certain rewrite rule sequences have the potential to *explode* the e-graph by adding exponential numbers of nodes, negating these benefits. In this scenario, *saturation* can never be reached, as it is not possible to simultaneously represent every possible program representation within finite memory-space.

To illustrate this, we have identified a simple propositional logic expression in which this scenario arises:

$$(x \wedge y) \vee (x \Rightarrow z) = (y \vee (x \Rightarrow z)) \tag{3.3}$$

Where the left-hand-side is input expression to be optimized. The right-hand-side is the optimal representation of the input expression, and is reachable through the use of a clever sequence of rewrite rule applications (see Table 4.1 for the complete list of propositional logic rewrite rules available).

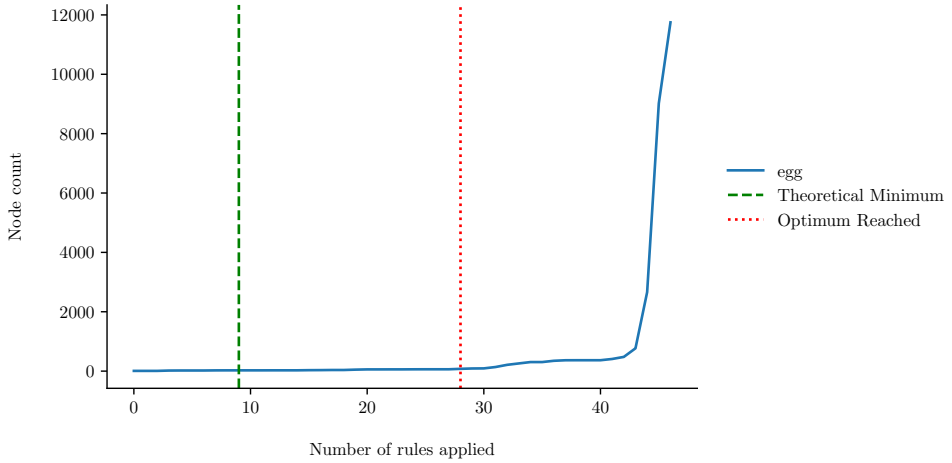


Figure 3.1: The size of the e-graph over time with a $10,000$ node limit applied. The Theoretical Minimum is the minimum number of rule applications necessary to reach the optimal cost. The second dotted line indicates the point at which egg can successfully extract the optimal cost solution; note that it will still continue growing the e-graph infinitely beyond this point as it has no means of stopping in this scenario until a timeout or node limit is reached.

This simple task causes egg to rapidly expand the e-graph due to unintuitive relationships between rewrite rules. Figure 3.1 displays the size of the e-graph over the course of egg's execution; interestingly, the e-graph remains at a reasonable size for the first 42 rule applications before growing exponentially until node limit is reached. Figure 3.2 denotes the number of applications of each rule, as well as the effect of each rule on the e-graph size. From the figure, it is immediately apparent that the `dist_or_and` rule, denoted as

$$\text{dist\_or\_and}\colon (a \vee (b \wedge c)) \to ((a \vee b) \wedge (a \vee c)) \tag{3.4}$$

is responsible for over $6,000$ nodes being added to the e-graph. However, this rule is applied four times before its explosive effects are felt; in fact, a causality analysis indicates that 9 other specific rewrite rules must be applied before `dist_or_and` can grow the e-graph exponentially. This scenario highlights the difficulty in predicting precisely when a given rewrite rule will explode the e-graph; it can be the case that only after a specific sequence of prior transformations can a rule become troublesome. This makes manually-defined heuristics to determine rule applications unfeasible, especially as a rule, such as `dist_or_and`, may be necessary to reach the optimal expression, and only becomes a problem when combined with a specific sequencing of other rule applications.

The first dotted line in Figure 3.1 indicates the theoretical minimum number of rewrite rule applications necessary to reach the target expression, whereas the second dotted line indicates the first point at which egg's solver *could* be terminated to return the target expression. As shown in the figure, theoretically only 9 rule applications are needed to obtain the optimal cost, while the e-graph built by egg takes 28 applications to reach

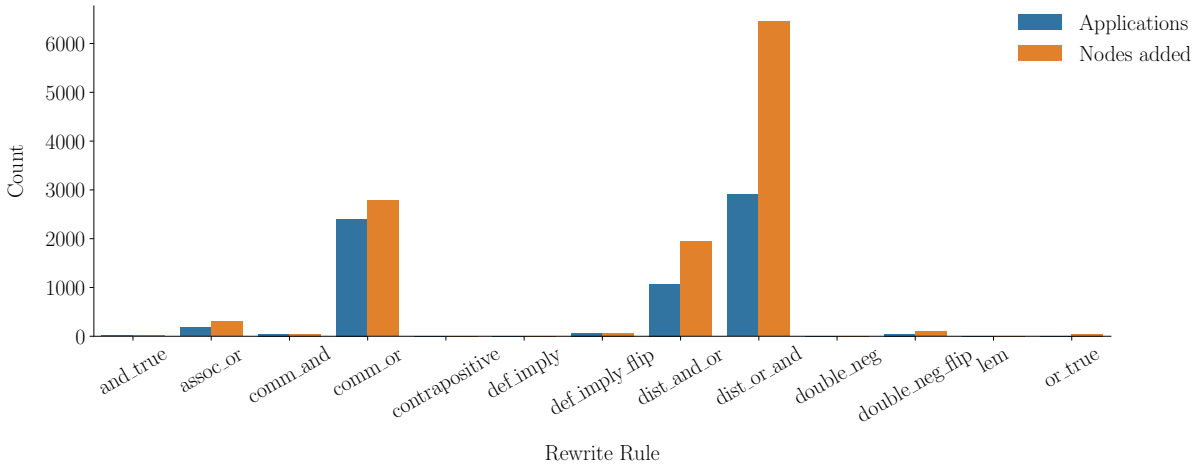


Figure 3.2: The number of applications of each rule and the total number of nodes added by all applications of each rule for Equation 3.3 with egg's default configuration.

a situation in which the optimal expression *could* be extracted; in other words, egg applies over $3x$ more rules than necessary to build an e-graph which contains the optimal expression. This problem arises even when e-graph explosion is not a concern, and is due to the fact that egg (and equality saturation approaches in general) do not *prioritize* rule applications; presumably, this design choice is made as in an optimal scenario equality saturation eliminates phase-ordering altogether by reaching saturation, and thus the sequence of rule applications does not matter. However, as shown here, phase ordering remains an efficiency problem in even simple scenarios, and should be an important consideration for performance by solver designers.

Additionally, in e-graph explosion scenarios such as this one, egg continues expanding the e-graph indefinitely past the point at which the optimal expression could be extracted until the node limit is reached, thereby unnecessarily wasting both computation and memory resources.

Summary. This section has raised three practical concerns with equality saturation. Firstly, the e-graph can expand in size unpredictably, such that it is unfeasible for manually-specified heuristics to detect when such a scenario could occur. Secondly, we note that egg has no method of knowing when to stop when saturation is impossible; it will continue to expand the e-graph until a node or time limit is reached, even if it could have extracted the optimal expression long ago. Thirdly, egg does not prioritize its rule applications, resulting in it taking significantly longer to reach a point where the optimal solution could be extracted from the e-graph; this holds true even in scenarios where e-graph explosion is not a concern. In combination, these three problems set the stage for our learning-based approach proposed in the next section.

3.3 Deep RL for Equality Saturation

This section presents *Omelette*, a Deep Reinforcement Learning variant of equality saturation which seeks to address the challenges laid out in the previous section. Recognizing that the saturation condition for termination is unlikely to be reached in any rewriting system of even moderate complexity, we instead design our agent with the objective of finding an *approximate* best solution *as quickly as possible* within the fixed memory and time constraints provided to the solver. To accomplish this, we give our agent the ability to choose both *which* rewrite rules are applied as well as their *order* in the hopes that it can learn the task of e-graph construction in order to minimize both the e-graph size and the number of rule applications necessary to produce solutions superior to those achievable by existing solvers. As the objective of our approach is no longer *equality saturation*, we propose the term *equality hydration* to more accurately reflect our method.

We hypothesize that Omelette should be able to find better solutions than egg on tasks where egg hits the e-graph node limit before the best solution can be discovered, as the RL agent should be able to intelligently learn which actions are needed to reach the solution while avoiding actions which increase the e-graph size unnecessarily. Additionally, by not running until the node limit is reached on tasks that don't require it, Omelette has the potential to reduce the computational and time requirements of equality saturation.

First in this section is a discussion on why we chose to build upon equality saturation in light of the limitations discussed previously, rather than searching destructive rewrite sequences directly upon the input expression. Following this, several experimental hypotheses are formulated to guide the evaluation of the RL agent. A high-level overview of the Markov Decision Process (MDP) of the RL agent is then provided, before diving into specific modifications and additions to the RL model designed to aid it in handling the equality saturation task. The section concludes with a description of the criteria by which the agent will be evaluated.

3.3.1 Why RL with Equality Saturation?

As we have sacrificed the goal of saturation, it may be questioned why we build upon equality saturation rather than training an RL agent to directly apply traditional, destructive rewrites upon the input expression. In response, we theorize that the unique properties of the e-graph data structure could enable an RL agent to learn much faster and more effectively.

This is because, while the e-graph cannot efficiently encode all possible term rewriting systems (as we have shown in the prior section), it can still reduce the search space by orders of magnitude, as many possible rewrite sequences can result in the same e-graph (See section 2.1.2). Therefore the agent only has to learn a policy over this smaller space of *possible e-graphs* rather than *possible rewrite sequences.*

Additionally, because rewrites are non-destructive in equality saturation, even if the agent chooses a sub-optimal rewrite, it is never prevented from later correcting itself; in other words, portions of the search space are never blocked off by a bad action choice (except in cases where a node limit is reached). This means that the RL agent's exploration has greater potential to lead to better states which can inform its policy function than if a destructive rewrite system were used. This should theoretically speed up training and enable better solutions to be found faster.

3.3.2 Hypotheses

Now that we have discussed the justification for combining RL with equality saturation to form our new method of *equality hydration*, we present a series of hypotheses regarding its abilities:

1. *The RL agent is capable of learning significantly shorter sequences of rewrite rule applications needed to obtain solutions for tasks across all domains when compared to egg, thus preventing e-graph explosion.*
2. *The agent can learn non-obvious action sequences in order to achieve a lower cost final expression than existing solvers within the node limit specified.*
3. *Omelette is capable of optimizing expressions from any language without modification to the agent, given only a list of rewrite rules and the set of operators available.*

3.3.3 MDP Definition

To summarize the design of the RL agent and environment, it is common to describe it through the lens of a Markov Decision Process (MDP) consisting of states, actions, and rewards. See Section 2.2 for background on MDPs and RL.

States

Each state s is the complete e-graph G at the current timestep. To construct a state, egg is queried to retrieve a list of e-classes and their constituent e-nodes. An encoding procedure builds a graph from this list, where each node has the attributes shown in Table 3.1. Each e-class node has an edge pointing to each of its member e-nodes, and each e-node has edges pointing to its children e-class nodes.

The reason e-classes are represented as nodes rather than as attributes of member e-nodes is because the number of e-classes can vary wildly throughout execution, as they are created and merged by the equality saturation algorithm. Therefore it is not possible to uniquely identify them within a fixed-size embedding.

Additionally, it should be noted that this encoding scheme does not encode the values of all scalar nodes present in the input expression. Instead, only *terminal* scalar values

Feature	Type	Applies to	Description
is_eclass	Boolean	e-classes	Whether the node represents an e-class.
is_enode	Boolean	e-nodes	Whether the node represents an e-node.
is_scalar	Boolean	e-nodes	Whether the node represents an unknown scalar (i.e. one not matched for in any rewrite rules).
is_terminal	Boolean	e-nodes	Whether the node represents a known terminal (i.e. one that is searched for by rewrite rules).
onehot_terminal	One-hot	e-nodes	One-hot encoding of the known terminal type.
onehot_operator	One-hot	e-nodes	One-hot encoding of the operator type.
onehot_avail_rules	One-hot	e-classes	One-hot encoding of the rules applicable to nodes within this e-class.

Table 3.1: The feature representation of each node within our E-Graph representation.

which are part of a left-hand-side pattern in the rewrite rule set are included through a one-hot encoding. For example, given a rule $(x * 0 \to 0)$ and an input expression $(27 * 0)$, the e-node representing 27 will not contain the numerical value 27, while the e-node representing 0 will, as it is used for matching. This is a crucial distinction, as it ensures that the agent does not over-fit on arbitrary specific values, but rather only is concerned with values which may be useful for rewrite matching. Additionally, because the set of rewrite rules, and therefore the set of terminals, are known in advance and static between problems within a domain, terminals can be automatically one-hot encoded.

It would be useful at this stage to mention that although many RL-based applications model their tasks as MDPs, their domains are often not truly Markovian. For example, the seminal work of Mnih et al. [26] uses a finite sequence of previous state-action pairs to decide upon a decision at the current state. Unlike such applications, the MDP as defined in this work obeys the first-order Markov property as all decisions are made entirely on the current e-graph structure.

Actions

The set of rewrite rules R in the provided language are one-hot encoded to form the action space of the agent. When an action a is chosen, the rewrite rule is applied at *every possible location* in the e-graph simultaneously. This design massively reduces the action space versus one in which the agent must also choose *where* to apply the action, but naturally comes with the limitation of potentially eliminating some solutions from being found if a rule application over all locations expands the e-graph beyond the node limit.

Additionally, two special actions are provided to the agent:

End. The *end* action terminates the current episode, and should only be applied when the agent believes there is no benefit to additional rewrite rule applications. This enables *early stopping* to prevent equality saturation from needlessly running until the node limit is reached, as shown in Section 3.2.2.

Rebase. The second special action, *rebase*, aims to enable recovery from e-graph explosion by extracting the best expression from the current e-graph, and creating a new e-graph from this expression to be used as the base for future actions. This effectively shrinks the e-graph back down to a manageable size so that optimization can continue further than otherwise possible. However, this action can potentially inhibit future optimization opportunities which are dependent on paths in the prior e-graph not included in the extracted result. However, since all rules are strictly additive, there is no reason the agent cannot restore these paths through further actions if necessary, at the cost of additional steps.

Rewards

Reward engineering is often considered the most important aspect of RL agent design [39]. This is because there are many ways for reward functions to unintentionally promote bad outcomes. For example, if the reward function is too sparse such that it fails to guide the agent's search, or is manipulable by the agent such that it can receive positive reward without accomplishing the intended goal, RL will fail for the given task.

In our domain, the reward of a $(state, action)$ pair is dependent on a number of factors beyond raw cost reduction to ensure these problems do not occur; this is necessary, as cost does not smoothly decrease with rule applications, thereby creating a sparsity problem. The reward calculation constructed for the various scenarios which arise during equality saturation are enumerated below. It should also be noted that all rewards are normalized to the range $[-1, 1]$ to stabilize training.

Saturated Rewrites. If the application of a chosen rewrite rule fails to modify the e-graph, it means that the rule has already been fully saturated into the current state, and further applications provide no benefit. In this case, the agent receives a reward of -0.5. This was chosen so that the agent does not waste time applying already-saturated rules in the future; rather, it should attempt to learn the *minimal* sequence of rules required to reduce expression cost.

Improved Cost. Cost reduction is the ultimate goal of term rewriting, and therefore the most important aspect of our reward function. A key constraint of this design is that the agent should be willing to suffer hundreds of negative reward intermediary states through which cost stays constant in order to obtain a small improvement later in execution. To achieve this, we first observe that cost is a positive and nonincreasing

function; because the E-Graph is purely additive (as described in Section 2.1.2), cost can only be decreased or stay the same throughout the equality saturation procedure. This means that the cost at the initial state is the maximum cost that will be observed through the episode. This enables us to normalize our reward by dividing the observed cost improvement by this maximum to obtain a reward in the range $[0, 1]$, thereby allowing reward design to be kept constant between domains which operate at wildly different cost scales (as some tasks equality saturation may be used for involve improving runtime by 0.01s, while others operate using integer cost metrics of "abstract syntax tree depth" which may have values in the hundreds). By doing so, we aim to satisfy Hypothesis 3 by having a singular agent design be flexible across domains.

E-Graph Explosion. If the application of a rewrite rule results in the pre-set node limit or time limit being reached, the agent's receives the lowest possible reward of -1 as punishment, and the current episode is terminated. This ensures our agent avoids unlimited e-graph expansion if at all possible, thereby aligning with Hypothesis 1.

Termination. If the agent chooses the *end* action when it has yet to improve the cost of the extracted e-graph expression, it receives the maximum negative reward of -1. In any other case, ending the episode provides 0 reward. The penalization of ending without any improvement is used to encourage exploration by preventing the agent from learning to greedily end the episode before it has discovered rewrites that deliver positive reward. In scenarios where no improvement is possible, a maximum step limit is configured so that the agent will eventually terminate regardless.

Rebasing. The reward obtained by the rebase action is *always negative*, and proportional to the reduction in the node count of the e-graph in the range $(-1, 0)$. In other words, a smaller decrease in e-graph size results in a stronger negative reward, as the goal of rebasing is to shrink the e-graph such that previously unreachable actions can be applied. The expense of rebasing is that it is a temporarily destructive action as described in Section 3.3.3, and therefore it should not be needlessly applied.

With the reward engineering of Omelette now fully discussed, we will proceed with describing the design of the agent's architecture.

3.3.4 RL Agent Architecture

While Figure 3.3 summarizes the high-level architecture of Omelette, this section will describes crucial implementation decisions necessary for the RL agent to function in the domain of equality saturation. Theoretical justification for the inclusion of specific features is provided, while their actual effectiveness will be tested in Chapter 4: Evaluation.

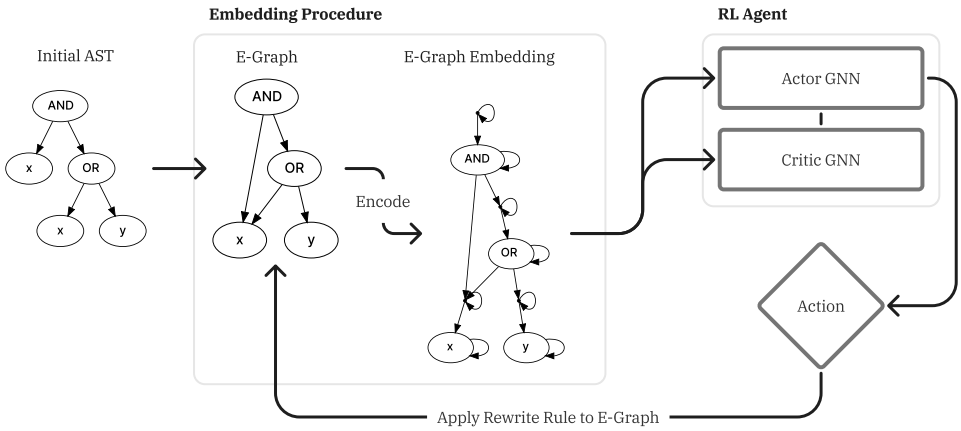


Figure 3.3: A high-level overview of Omelette's architecture.

Choice of RL Algorithm

Notable to our task is that the state space of all possible e-graphs is infinite, making tabular RL methods impossible as the memory required would be unbounded. Therefore a Deep Reinforcement Learning (DRL) technique must be used to approximate the value and policy functions. Proximal Policy Optimization (PPO) [36], a popular policy gradient method which strikes a balance between sample complexity, implementation simplicity, and wall-time, is used for the agent primarily due to the stability benefits described in Section 2.2.2.

While there exist many alternative possible RL algorithms, such as Deep Deterministic Policy Gradient (DDPG) [18], an additional constraint of our domain is that actions are *discrete* rather than *continuous*, thereby restricting the algorithms we can utilize.

While other architectures which support discrete action spaces, such as Deep Q-Networks (DQN), which features improved sample efficiency at the cost of additional training time and memory usage, were considered, PPO was chosen as sample efficiency is not of chief concern; collecting observations takes as little as single-digit milliseconds in our environment.

Additionally, a challenge in our work involved adapting the RL algorithm to utilize Graph Neural Networks; doing so required a custom ground-up implementation, as GNNs are incompatible with the parallelization and batching strategies used by off-the-shelf RL implementations. Because of this, the implementation complexity of the algorithm was a key concern in decision-making; as PPO's implementation has been comprehensively studied [1] with multiple reference implementations available, it serves as an ideal base for our experimental work.

GNN Architecture

As previously mentioned, unlike most existing work in RL literature, which operates on fixed-size observations such as an image of the current frame in a video game [25] or sensor data from a robot [12], our observations take the form of graphs with varying numbers of nodes and edges. Because of this, typically used value and policy network architectures such as MLPs and CNNs are not applicable; a Graph Neural Network is used in their place, bringing with it a number of additional challenges and considerations.

Graph Convolutional Networks (GCNs) [14], Graph Attentional Networks (GATs) [42], and Graph Isomorphism Networks (GINs) [47] were all tested for use as the actor and critic networks in the PPO agent. A 3-layer GAT with 64 neurons per layer was found to produce the best results, likely due to GAT's improved expressivity as described in Section 2.3.2. To address initial challenges with non-convergence, the GNN normalization method GraphNorm, presented by Cai et al. [5], has ben used to significantly stabilize training; it does so by normalizing hidden representations across nodes using a learnable shift to avoid degrading expressivity.

As both the actor and critic networks operate at the graph-level (with the critic returning an estimated value for a given e-graph, and the actor predicting the best action to take given the state), an aggregation layer must be used to convert the GNN's learned node-level representations into graph-level values. For this purpose, a sum pooling aggregation is used which adds the individual node features before passing them through a 2-layer MLP of 64 neurons per layer to produce the final predictions. The choice of pooling function is not arbitrary, as other forms of pooling, such as average or max pooling, sacrifice the ability to distinguish certain types of graphs. For example, average pooling would be unable to differentiate between two graphs with identical node representations where one had twice the nodes of the other.

OpenAI Gym [4] was used to create the RL environment such that other agent architectures can be easily tested upon it. As a contribution of this work, we have assisted in adding support for graph environments to Gym; as of writing, this has been merged into the library and is now generally available.

Action Masking

Training time for RL agents scale heavily with the number of actions available at each step. As our agent design must be flexible across domains which may feature dozens or hundreds of rewrite rules, action masking is used to prevent rules which are not applicable to the current e-graph from being selected. This is achieved by masking the actor network's logits such that invalid actions have a zero probability of being taken. While the best way to mask actions is still an active area of research [11], this method was found to be sufficient for our purposes.

3.4 Agent Evaluation Configuration

RL is notorious for its large number of hyperparameters and its high sensitivity to them, as even slight changes to its initial configuration can yield drastically different outcomes [39]. Because of this, we have taken efforts to ensure all experiments are reproducible, and provide all important configuration details used in our evaluation in this section. Due to computational cost, a full hyperparameter sweep to find optimal values was outside the scope of this work; nonetheless, a high-level grid search was performed and augmented with hand-tuning to produce the settings which will be discussed in this section.

Firstly, all experiments were conducted with the same random seed of 1 to ensure fair comparison between methods, as the PPO algorithm is stochastic. Note that all other optimization techniques (such as constant folding) have been disabled for all experiments so that equality saturation is considered in isolation. This significantly increases the difficulty of optimization tasks, and enables even simple expressions which are readily interpretable to humans to be challenging to solve.

Training. The agent is trained for $100,000$ steps (split between episodes consisting of up to 100 steps) for each task. While this may be longer than necessary for some problems and shorter than necessary to converge for others, it establishes a reasonable baseline for performance. To stabilize training, the agent is trained on 32 environments concurrently to provide it with a wider sample of observations to be used in each gradient update round; these occur every 128 steps (cumulative across all parallel environments) using a learning rate of $2.5e-4$. Additionally, while the agent is provided with a maximum episode length of 100 actions, in practice each episode's duration is far shorter; this is because the reward function described in Section 3.3.3 penalizes unnecessary steps, incentivizing the agent to find the minimal sequence of actions required for the task at hand.

Evaluation. As PPO produces a stochastic policy, a single rollout is not sufficient to determine its performance. Therefore for final metrics regarding agent performance, the learned policy is rolled out five times, with the best result reported in our evaluation. This is reasonable, as policy roll-outs take on the order of milliseconds in most cases.

3.4.1 Baselines

egg. The egg equality saturation solver is used as the primary baseline for comparison. For all experiments, egg is configured to use a maximum e-graph size of 500 nodes and a time limit of 5 minutes. Although in practice egg can be configured to use a larger node limit (such as $1,000$), the exact number is not important for a relative comparison between the RL agent and egg; however, a higher node limit would greatly increase the memory and compute requirements of these experiments, and

as such is left to future work.

Random Search. Random search is used as a secondary baseline to show that RL is truly effective for this task. This is surprisingly important, as random search has been shown to be competitive with RL across many domains [20]. Ideally, if there is truly information in the environment which can guide the search, Omelette should be capable of finding significantly better solutions than a random search ran for the same number of roll-outs.

3.4.2 Evaluation Criteria

The following metrics are used to evaluate Omelette's performance against that of egg:

Cost. The cost of the extracted expression, in terms of abstract syntax tree size, is the primary metric used for evaluation. A lower cost is superior, but note that there exists a theoretical minimum cost for every task; therefore it cannot be expected that Omelette achieves superior performance for every expression if egg is also able to find the optimal.

Rule Applications. It must also be noted that the number of rewrite rule *actions* is distinct from the number of rewrite rule *applications*. This is because, upon picking a single rewrite rule to run (an *action*), the rule is then *applied* to every possible location within the e-graph. Therefore a single action can result in potentially thousands of rule applications. As the number of rule applications is a better indicator of runtime duration than rule actions, it is the primary metric reported. A lower number of rule applications is superior, and indicates that a solution was able to be found faster.

E-Node Count. The size of the e-graph at the end of optimization, measured in terms of e-node count, is used to compare how efficiently each method was able to find its solution within the node limit specified. A lower e-graph size is preferable if the same or lower cost was able to be achieved within it.

3.4.3 System Configuration

All tests were conducted on an AWS EC2 instance with four vCPU cores, 32GB of RAM, and an Nvidia T4 GPU with 16GB of VRAM. The machine runs Ubuntu 18.04 and CUDA 10.2.

Omelette consists of approximately $5,000$ lines of Rust and Python code. The e-graph data structure from egg has been used in our work and bridged into Python such that the RL agent can interact with it. All code for the work in this thesis is open-source and available on GitHub [1].

[1] `https://github.com/ZakSingh/omelette`

Chapter 4

Evaluation

This chapter evaluates Omelette's performance against both egg and a random agent across a number of tasks in order to test the hypotheses presented in Section 3.3.2. The objective of this section is to ascertain the following:

1. Whether Omelette can *significantly* reduce the number of rule applications needed to obtain the same or better solution than egg, thus answering Hypothesis 1.
2. Whether Omelette is capable of finding *lower cost* solutions than egg in scenarios where the node limit is reached before saturation, thus answering Hypothesis 2.
3. Whether Omelette can learn across different languages without requiring modification of the agent, thus answering Hypothesis 3.

The section is structured such that first the two domains used for evaluation, consisting of propositional logic (`PROP`) and mathematical simplification (`MATH`), are presented. This is followed by a high-level overview of the general performance behaviors of the RL agent over datasets of expressions generated within these two domains. Subsequently, we analyze *when* and *how* Omelette is able to significantly outperform egg on certain tasks due to the unique advantages of RL and Omelette's design as discussed in Section 3.3.

4.1 Test Domains

Two distinct domains are used to evaluate Omelette; their specific constitutions will now be discussed. With each defined rewrite system we specify our method of generating valid sample expressions which will be used to test the agent. In doing so, we report the number of operators in each generated expression, the number of e-nodes in the initial e-graph encoding of each expression, and the initial cost found by egg given the initial e-graph; all of these metrics serve as rough proxies for expression complexity and therefore difficulty. However, as previously noted in Section 3.2.2, there exist diabolical simple expressions which can unpredictably lead to unbounded e-graph growth and non-termination.

Rule	LHS	RHS
def_imply	IM(a, b)	OR(NOT(a), b)
double_neg	NOT(NOT(a))	a
def_imply_flip	OR(NOT(a), b)	IM(a, b)
double_neg_flip	a	NOT(NOT(a))
assoc_or	OR(a, OR(b, c))	OR(OR(a, b), c)
dist_and_or	AND(a, OR(b, c))	OR(AND(a, b), AND(a, c))
dist_or_and	OR(a, AND(b, c))	AND(OR(a, b), OR(a, c))
comm_or	OR(a, b)	OR(b, a)
comm_and	AND(a, b)	AND(b, a)
lem	OR(a, NOT(a))	True
or_true	OR(a, True)	True
and_true	AND(a, True)	a
contrapositive	IM(a, b)	IM(NOT(b), NOT(a))

Table 4.1: The rewrite rules available in the PROP domain.

4.1.1 The PROP Domain

`PROP` is a simple propositional logic language proposed by egg [45]. It features 4 operators from which expressions can be built (AND, OR, $IMPLIES$, NOT), two terminal values ($True$ and $False$), and 13 rewrite rules listed in Table 4.1. The cost metric is the depth of the abstract syntax tree of the extracted expression, thereby incentivizing simplification as the ultimate goal.

To evaluate solver performance in this domain, 29 expressions have been generated with a maximum depth of six operators before leaf node terminals are reached; however, the generator has a 50% chance of choosing a terminal at each level, meaning that most expressions are not of this maximum size. A summary of expression complexity is provided in Figure 4.1, illustrating that the generated expressions have a median of 50 operators. Note that this translates to a slightly lower e-node count in the initial e-graph embedding as shared terms are de-duplicated. Also shown in the figure is a distribution of the best cost achievable by extraction from the e-graph without any rewrite rule applications: this serves as a tangible basis for the relative cost improvements which will be discussed in the evaluation. Additionally, a breakdown of the operator distributions of each generated expression is shown in Figure 4.2a, providing a high-level overview of the diversity of tasks presented to the solver.

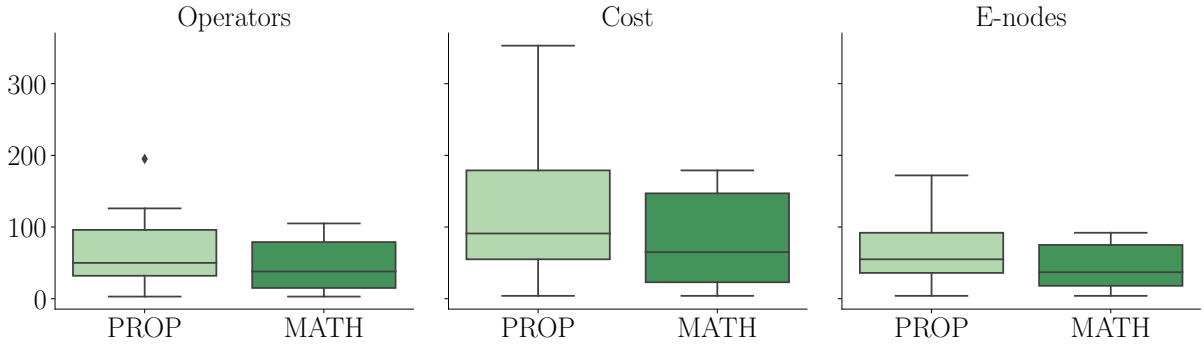
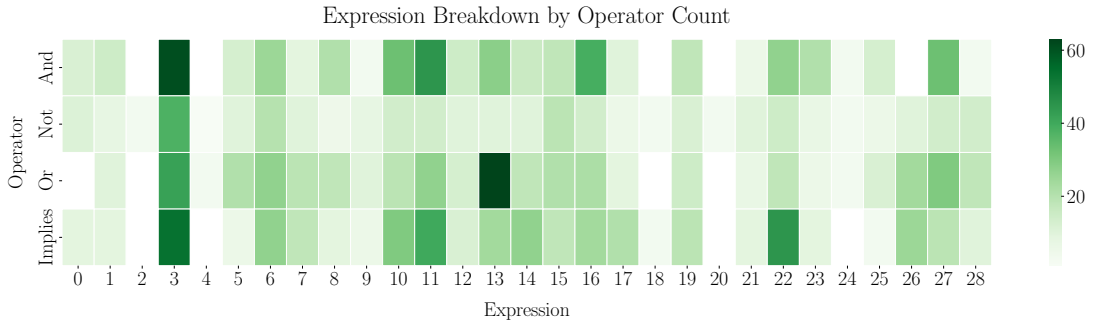
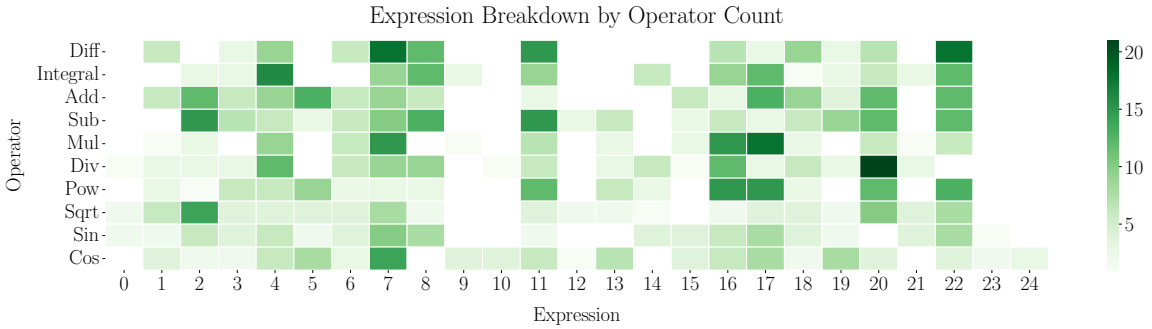


Figure 4.1: Distributions of the number of operators, initial cost, and number of E-Nodes present in the generated datasets. The box shows the quartiles of the distribution centered around the median while the whiskers extend to the maximum and minimum, with outliers denoted as floating diamonds. All three metrics serve as rough approximations of expression complexity.



(a) PROP Operator Distribution



(b) MATH Operator Distribution

Figure 4.2: The distribution of operators within the expressions generated for each domain. Note how different expressions have significantly different constitutions, thereby posing unique challenges to the solver.

Rule	LHS	RHS
comm_add	ADD(a, b)	ADD(b, a)
comm_mul	MUL(a, b)	MUL(b, a)
assoc_add	ADD(ADD(a, b), c)	ADD(a, ADD(b, c))
assoc_mul	MUL(MUL(a, b), c)	MUL(a, MUL(b, c))
sub_canon	SUB(a, b)	ADD(a, MUL(-1, b))
zero_add	ADD(a, 0)	a
zero_mul	MUL(a, 0)	0
one_mul	MUL(a, 1)	a
add_zero	a	ADD(a, 0)
mul_one	a	MUL(a, 1)
cancel_sub	SUB(a, a)	0
distribute	MUL(a, ADD(b, c))	ADD(MUL(a, b), MUL(a, c))
factor	ADD(MUL(a, b), MUL(a, c))	MUL(a, ADD(b, c))
pow_mul	MUL(POW(a, b), POW(a, c))	POW(a, ADD(b, c))
pow1	POW(x, 1)	x
pow2	POW(x, 2)	MUL(x, x)
d_add	DIFF(x, ADD(a, b))	ADD(DIFF(x, a), DIFF(x, b))
d_mul	DIFF(x, MUL(a, b))	ADD(MUL(a, DIFF(x, b)), MUL(b, DIFF(x, a)))
d_sin	DIFF(x, SIN(x))	COS(x)
d_cos	DIFF(x, COS(X))	MUL(-1, SIN(x))
i_one	INTE(1, x)	x
i_cos	INTE(COS(x), x)	SIN(x)
i_sin	INTE(SIN(X), x)	MUL(-1, COS(x))
i_sum	INTE(ADD(f, g), x)	ADD(INTE(f, x), INTE(g, x))
i_dif	INTE(SUB(f, g), x)	SUB(INTE(f, x), INTE(G, x))
i_parts	INTE(MUL(a, b), x)	SUB(MUL(a, INTE(b, x)), INTE(MUL(DIFF(x, a), INTE(b, x)), x))

Table 4.2: The 26 rewrite rules available in the MATH domain.

4.1.2 The MATH Domain

The `MATH` domain consists of 11 operators: differentiation, integration, addition, subtraction, multiplication, division, exponentiation, logarithms, square roots, sine, and cosine. Table 4.2 illustrates the 26 rewrite rules available. There are three terminal values used on the left-hand-side of the rewrite rules which are therefore encoded into the GNN embedding: 0, 1, and 2. With a significantly larger number of operators and rules than the `PROP` domain, the `MATH` domain is designed to be a more challenging task for the agent while still remaining easily interpretable to humans. Just as in `PROP`, the cost metric is defined as the depth of the abstract syntax tree, incentivizing the solver to simplify the expression to a minimum number of terms.

25 expressions were generated for evaluating solver performance within the `MATH` domain. It should be noted that, due to the larger number of rules and operators at play, memory constraints necessitated that expressions in this domain be reduced to a maximum depth of 5 (versus `PROP`'s 6), resulting in the slightly smaller operator and e-node counts visible in Figure 4.1. However, this presents an interesting opportunity to determine if the difficulty posed to the equality saturation solvers is correlated more to input expression complexity or the complexity of the rewrite rule set. Figure 4.2b displays a summary of the generated expressions by their operator counts, highlighting the variety of constructions being tested; note how some inputs feature a wide range of operators, such as expression 7, while others, such as 9, are sparse, thereby posing unique challenges to the solvers.

With the test domains now comprehensively described, the following section will discuss the results of running Omelette and egg upon them.

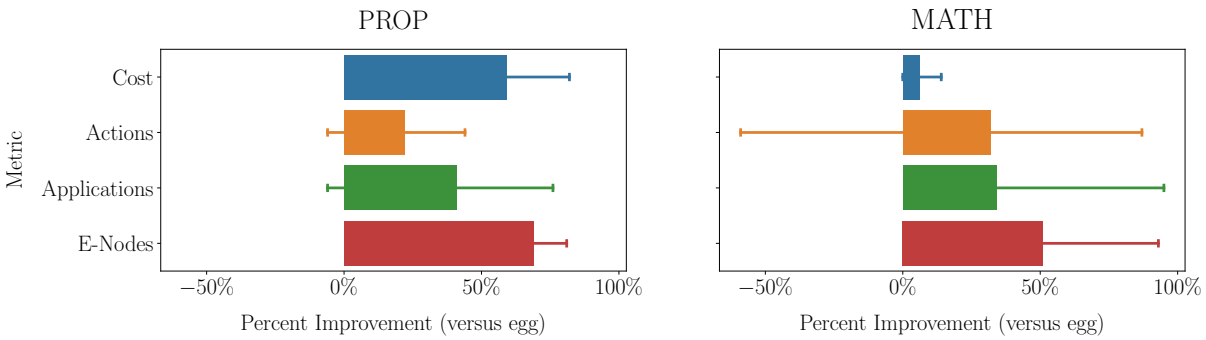


Figure 4.3: Summary of performance metrics for Omelette over the PROP and MATH datasets, shown as a percentage improvement over egg. Each bar displays the median, with minimums and maximums shown via whiskers.

4.2 Overview of Results

Figure 4.3 summarizes our results on the `PROP` and `MATH` datasets. From the graph, it is clear that Omelette greatly reduces the final expression cost, the number of actions taken during optimization, the number of individual applications of those actions, as well as the size of the e-graph in terms of e-node count in comparison with egg. Notably, we observe that Omelette is able to significantly reduce optimized expression cost over egg by an average of 58% in the `PROP` domain and 13% in the `MATH` domain. Furthermore, Omelette is able to accomplish this with, on average, 42% fewer rule applications across both domains. Additionally, the final e-graph size produced by Omelette is on average 64% smaller than that of egg. Overall, it is clear that Omelette's learned rewrite rule sequences are far superior to traditional equality saturation, validating Hypotheses 1 and 2 by finding solutions which are both better and shorter than egg's.

While the cost reduction achieved in the `MATH` domain by Omelette is smaller than that of `PROP`, this is likely due to the decrease in complexity of the expressions contained within `MATH` necessitated due to memory limits as described in Section 4.1.2. In other words, there is less opportunity for improvement over egg on tasks from this domain; nonetheless, Omelette still displays strictly superior performance.

Notable within 4.3 is the increased action counts used by Omelette, shown as negative improvement over egg. However, upon closer analysis, this behavior is required for discovering lower cost solutions; Omelette's clever avoidance of the node limit enables it to apply *more* actions in order to find previously unreachable solutions. Additionally, as visible within the `MATH` domain, the large increase in actions does not translate to an increase in applications: because the e-graph being operated upon stays small, actions are being matched to fewer e-classes, resulting in *fewer* applications for *more* actions. This ability to work around the node limit to find lower cost solutions will be discussed further in the subsequent sections.

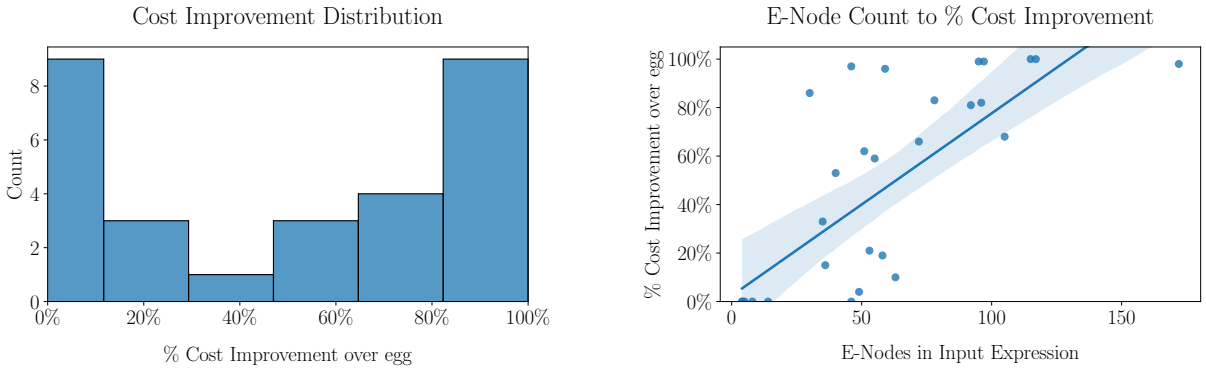


Figure 4.4: The distribution of cost improvements found by Omelette (left) and the relationship between the number of E-Nodes in the input expression and Omelette's cost improvement (right).

4.3 Finding Lower Cost Solutions

As stated in Hypothesis 2, a primary objective of Omelette is to find lower cost solutions than egg. Figure 4.4 displays a histogram of the cost improvements in solutions found by Omelette over egg, as well as a graph of the relationship between the number of e-nodes within the input expression and the improvement found. From these results, it is apparent that Omelette is *strictly superior* than egg in terms of cost, and its improvement over egg partially scales with the complexity of the expression to be optimized.

It is interesting to observe on the left plot in Figure 4.4 that cost improvement appears to be largely *all-or-nothing*: either the improvement over egg is insignificant (between 0 and 10%) or drastic (over 80%). This highlights the nonlinearity of performance improvements found through rewriting. Applying additional rules rarely increase performance; rather, it may be the case that 20 rules must be applied before a single additional rule drops the cost by 100% by enabling extraction along a path in the e-graph which depends on these previous rule applications. When egg reaches the node limit before that final, all-or-nothing rule application can be reached, it will have no improvement upon cost.

This is known as the *delayed reward* problem in RL literature, and is notoriously difficult for agents to solve [2]; nonetheless, within this domain, Omelette is able to converge to the optimal solution. Section 4.4.1, later in this chapter, will provide a case study of Omelette's training behavior which illustrates how it is able to achieve this.

Also of note is the presence of outliers within the figure. Notice how some expressions with fewer than 50 e-nodes achieve an up to 94% cost improvement with Omelette. This highlights how unpredictable equality saturation tasks may be; even simple expressions can hit the node limit with egg and therefore benefit greatly from Omelette's learning-based approach.

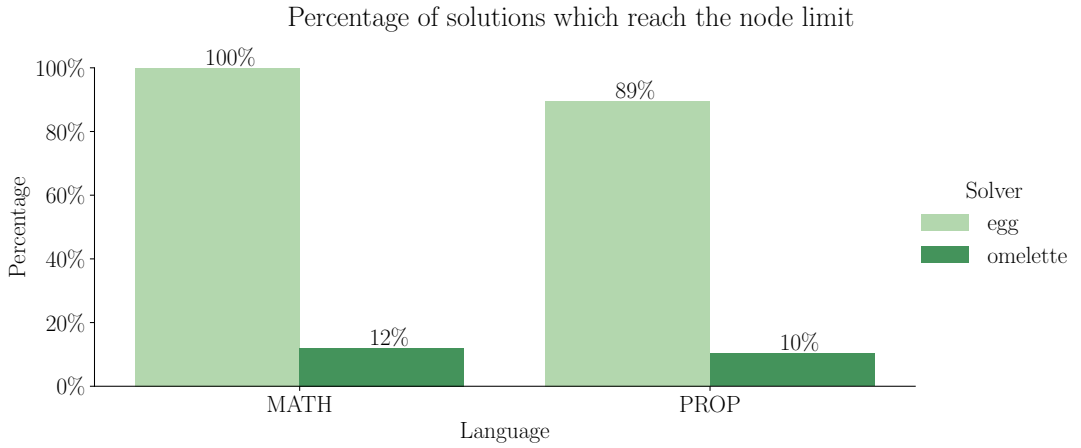
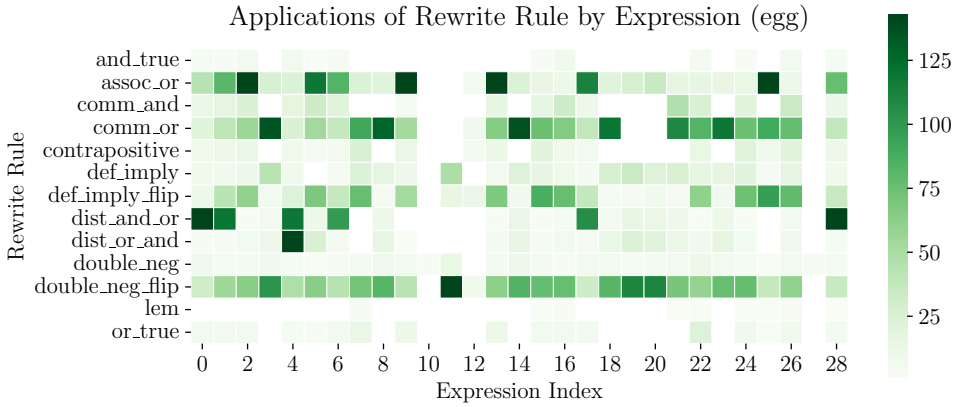


Figure 4.5: Percentage of solutions which hit the node limit. While Omelette is able to tremendously improve upon egg in this regard, it is not able to avoid reaching the node limit in *all* scenarios.

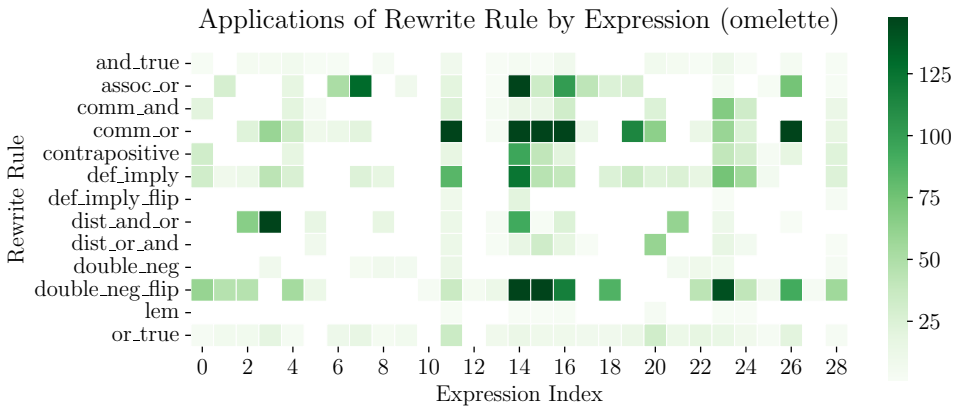
4.4 Avoiding Node Limits

Figure 4.5 shows the percentage of tasks in each domain which cause egg or Omelette to reach the node limit. From the graph, we can see that egg's solutions almost always terminate due to hitting the node limit, doing so on 100% of tasks from the complex `MATH` domain and on 89% of those in `PROP`. Meanwhile, Omelette fares much better, only being stopped by the node limit on 16% of expressions in `MATH` and 3% of those in `PROP`. These results highlight Omelette's ability to effectively avoid running into the node limit across a variety of tasks, while a comprehensive case study of agent performance on a task in which it is able to achieve a lower cost than egg by navigating around the node limit will be provided in Section 4.4.1. However, this plot also demonstrates that Omelette is not *always* successful in doing so. This raises a limitation in Omelette's design: by applying each chosen rule to *every* applicable location, there are situations in which it is not possible to reach the minimum cost without also hitting the node limit if a widely-applicable rule is the only method of doing so. Although a more granular action structure which allows the agent to choose *where* to apply each rewrite may address this, doing so would drastically increase the action space and therefore hinder training time.

To show how Omelette alters its action plan to avoid the node limit, Figure 4.6 displays a breakdown of the number of rule applications per input expression compared between egg and Omelette as a heatmap. Immediately apparent in the chart is the sparsity with which Omelette selects rules; it is clear that Omelette is capable of altogether ignoring rules which do not aid in finding a good result, while egg wastes applications (and therefore an increase in e-graph nodes) upon rules which may not be beneficial at all, thus validating Hypothesis 1.



(a) egg



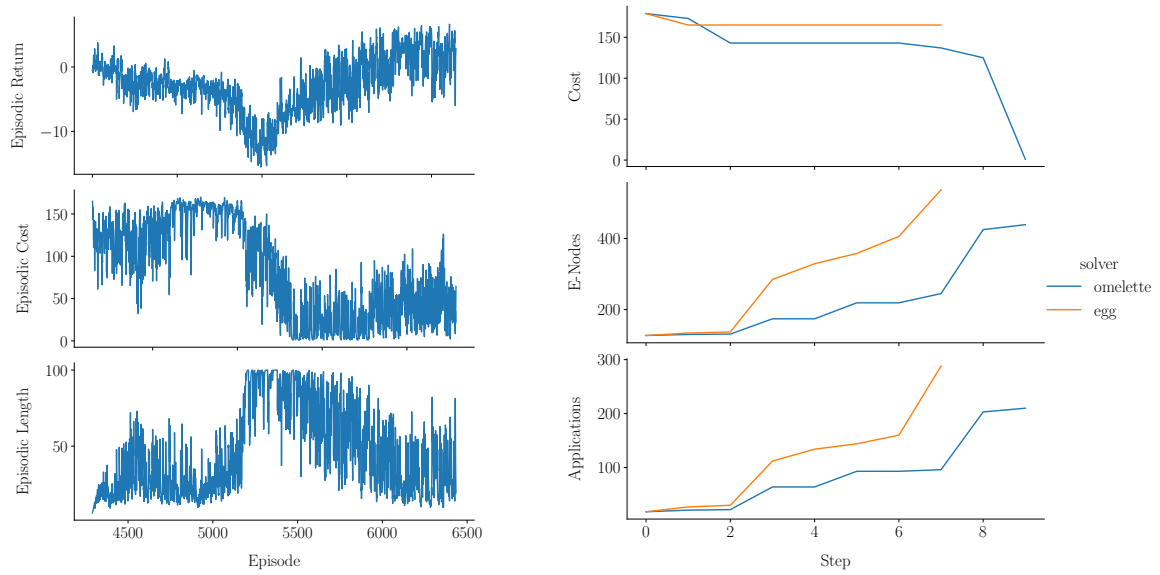
(b) Omelette

Figure 4.6: The number of rule applications of each rule by input expression in the PROP domain. White squares denote rules with zero applications. Note how Omelette learns to only apply rules which are useful for the input expression, and in some cases will apply them *more* than egg if needed to find lower cost solutions.

4.4.1 Case Study: Node Limits Prevent Optimal Solutions

This section illustrates Omelette's training process on a task in which egg is unable to find the optimal solution within the node limit specified. Hypothetically, Omelette's RL agent should be able to learn around the node limit constraint to identify lower cost solutions through the use of intelligent rewrite rule orderings and the rebase action proposed in Section 3.3.3.

Expression 8 from the `PROP` dataset has been chosen for this demonstration; it consists of 254 operators, making it one of the more complex tasks tested. While egg is only able to reduce the cost from its initial value of 179 to 165, Omelette is able to fully reduce it to the minimum possible cost of 1. Figure 4.7 displays both the training curves and learned policy of Omelette. Many interesting observations can be derived from these figures; of clear note is the high variance exhibited by the agent's episodic return, cost, and length curves over the course of training. In fact, it is apparent that the agent has not truly converged by the time we stop training. This large amount of noise is due to a number



(a) Agent training metrics over episodes.

(b) Learned policy roll-out of a single episode *after* training.

Figure 4.7: A visualization of Omelette's training process and a roll-out of its learned policy for a single episode compared to egg for Expression 8 from the PROP domain.

of factors: firstly, a single rule action can have a dramatic effect on cost; notice how the top-right plot in the figure shows how Omelette is able to go from 144 cost to 1 with a single action. This is because of the e-graphs shared encoding of alternative expression representations; a single rewrite can simultaneously open up hundreds of new possible extraction paths through the e-graph, one of which may result in a dramatic change in performance. Therefore noise is to be expected, especially in tasks such as this in which there are many possible rule choices which reduce cost by varying extents.

Additionally, from the figure we can observe how the agent was able to discover this massive cost reduction. Cost remains static while episodic return declines until around 5300 episodes, when the new promising rewrite sequence is first seen. Notice how, at that point, the episodic length jumps to its maximum of 100, then steadily declines. This trend shows how the agent first expanded its search through the rewrite space (at a significant cost to its received reward), taking advantage of the purely additive nature of the e-graph, before progressively shrinking down the episode length as it determines which actions were necessary to reach the new minimum cost. This reduction process is naturally noisy, as some modifications to the sequence will unintentionally drop pre-requisites for later rules which reduce cost, causing the return and cost to vary significantly.

Finally, it is clear from the roll-out of the learned policy compared to that of egg that Omelette is able to firmly stay within the node limit throughout execution, while egg reaches the limit by the 8th action step and terminates. This example provides clear validation for Hypothesis 2 as Omelette is readily able to find a lower cost solution than egg within the same node limit.

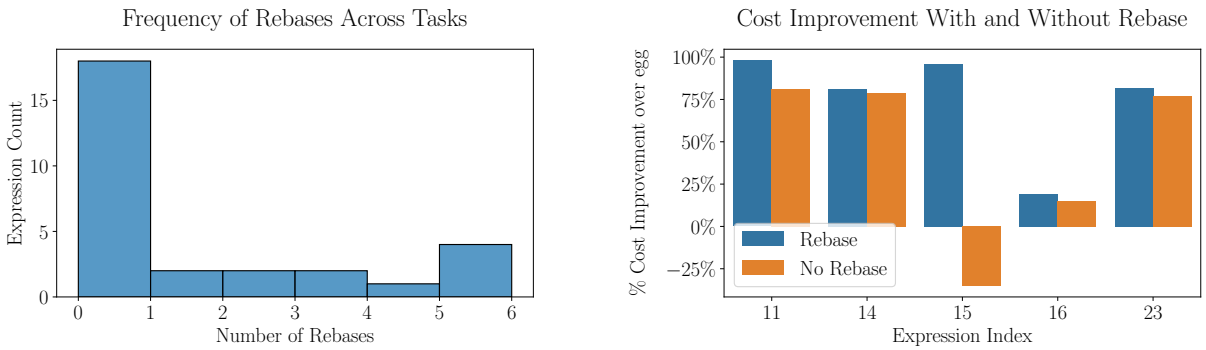


Figure 4.8: Frequency of rebase operations within Omelette's policy (left) and a comparison of the median cost reduction (averaged over 5 runs) achieved with and without the rebase action (right). While the rebase action is only needed for a small number of tasks, it can cause a sizable improvement in final cost in those cases.

4.5 Effectiveness of the Rebase Action

The rebase action, described in Section 3.3.3, extracts the best expression from the current e-graph and continues optimization upon it. We theorize that this action should enable the agent to reach better solutions within the node limit by shrinking the e-graph once it has grown too large before re-expanding it to access portions of the search space that were previously unreachable.

To determine the effectiveness of the rebase action, the agent was evaluated both with and without it available for use. Figure 4.8 displays how frequently rebase occurs in the learned policy of the agent across different tasks, as well as a performance comparison on tasks where disabling it affects the final cost. As visible in the figure, rebase is not always needed by the RL agent for optimization; rather, it is only used when required to find better solutions. In these scenarios, rebase can sometimes make a tremendous difference, going from a 28% cost *increase* over egg with it disabled to a 94% cost *decrease* with it enabled on Expression 15 from the `PROP` dataset. More generally, however, the cost decrease found via rebase is smaller, with between a 5 to 10% cost reduction being achieved. While this may not appear meaningful, an improvement of this scale can make a significant difference in the hot-path of compiled systems. Additionally, this small improvement indicates that the reward engineering of the agent described in Section 3.3.3 is successful in incentivizing cost reduction over all else, as the agent is willing to sacrifice reward to utilize the costly rebase action in order to later be rewarded for achieving a lower cost solution, no matter how small.

From these results, we can conclude that the rebase action is a valuable addition to the design of the RL agent and is necessary for achieving good results on some tasks. Additionally, we note that the rebase action does not require the use of RL; equality saturation solvers could readily implement it to improve performance. However, to do so would require the design of heuristics or other mechanisms to guide its usage.

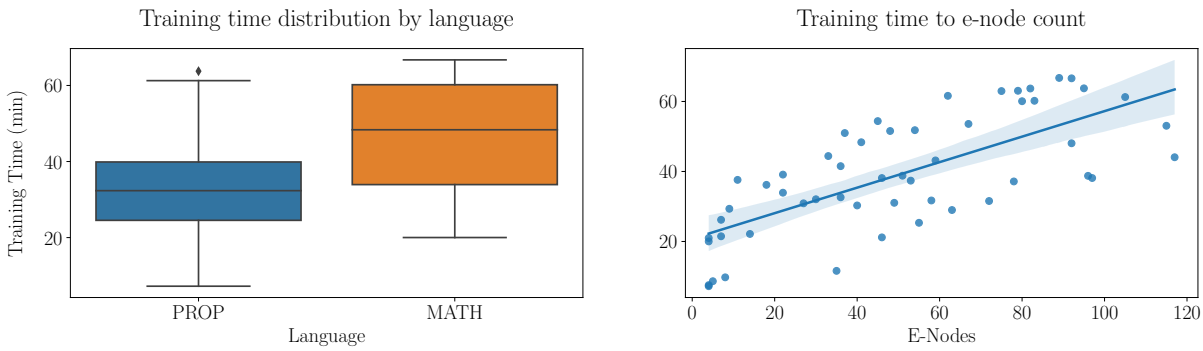


Figure 4.9: Training time distribution by language, as well as the relationship between node count and training time across both domains.

4.6 Training Time

As the RL agent must learn to solve each task in isolation, its training time is an important consideration for the practical feasibility of Omelette; this is especially a concern as RL is known for its tremendous computational costs [3]. As previously mentioned in Section 3.4, the RL agent is trained for $100,000$ steps on each task. However, this does not translate to a uniform wall-clock time; as displayed in Figure 4.9, training time varies significantly across tasks from 3 to 70 minutes and appears to be at least partially dependent on the size of the final e-graph. Additionally of note is that `MATH` tasks take considerably longer, even though they are constituted by fewer nodes; this could be due to the increased node feature sizes caused by the larger number of rules and operators available in the domain.

While egg benefits from not requiring training, and is therefore able to run within milliseconds to seconds, Omelette remains useful in situations where it is able to find lower cost solutions than egg. Omelette's training time may be further improved by introducing early-stopping to end training when a good solution is reached, or through performance improvements such as parallelization and supporting multi-GPU training. Nonetheless, training time remains a significant obstacle for the practical application of RL-based methods generally.

4.7 Comparison with a Random Agent

To determine if RL is truly effective for this task, we compare it to an agent which chooses a random action at each step. The percentage improvement in terms of cost over egg for Omelette and the Random search is provided in Figure 4.10. Clearly, Omelette is far superior to random search on this task, achieving an average 46% improvement over egg while random search attains 10%. This result indicates that the solutions found by Omelette are nontrivial, and that the reward function is successfully guiding it through the search space to find solutions that would be extremely difficult to reach through a random search. This results also highlights an interesting phenomenon: random search

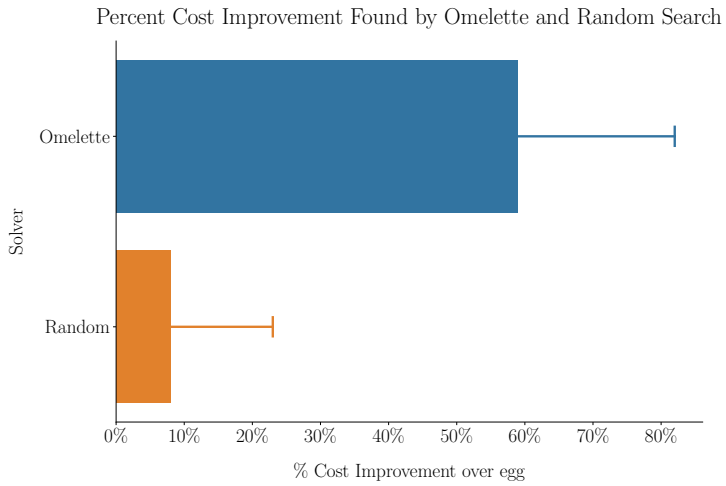


Figure 4.10: A comparison of the cost improvement over egg found by Omelette and a Random agent across both domains. The median is shown by the bar with the maximum displayed by the whisker. While Omelette performs significantly better, it is interesting to note that random search still always performs better than egg.

is superior to egg in the majority of cases. This aligns with the practical limitations identified in Section 3.2.2; because egg applies rules exhaustively in each iteration, it loses some optimization opportunities that trivially occur through random search (for example, by applying a rule twice in a row) when a node limit is reached before those paths can be added to the e-graph. Because of this, we suggest that future equality saturation solvers experiment with the use of random rewrite rule ordering as a configuration option.

4.8 Overview

In this chapter, we have evaluated Omelette against both egg and a random agent across tasks from the `PROP` and `MATH` datasets described in Section 4.1. In doing so, we have verified the hypotheses presented in Section 3.3.2 by demonstrating that Omelette is able to find lower cost solutions than egg in both the `PROP` and `MATH` domains while avoiding unnecessary growth of the e-graph. In some cases, Omelette is even able to achieve a 100% cost reduction over egg. Additionally, by finding new optimization paths in both symbolic and numerical domains, our results support Hypothesis 3's claim that the RL agent is flexible across rewrite systems without requiring architectural changes or hyperparameter tuning. However, to fully justify this hypothesis would require wider testing through the construction of additional domains outside of the scope of this work.

Chapter 5

Summary and Conclusions

In this thesis, the various challenges surrounding Equality Saturation have been identified. We have demonstrated that equality saturation cannot reliably achieve saturation even in simple domains as unbounded e-graph expansion can occur unepredictably within rule sets. We argue that equality saturation solvers must carefully consider the order in which rewrites are applied so as to not go down a path of infinite expansion.

We identify that this challenge is under-considered in existing equality saturation literature, and present a novel Reinforcement Learning solution, *Omelette*, to guide the ordering of rewrite rule applications to the e-graph in order to address this problem.

By evaluating Omelette across both propositional logic and mathematical simplification tasks, we have shown it to be capable of discovering superior solutions than egg within the e-graph memory limits specified, while reducing the number of rewrite rule applications necessary to reach such solutions across the board. Specifically, we have verified the following hypotheses:

1. *The RL agent is capable of learning significantly shorter sequences of rewrite rule applications needed to obtain solutions for tasks across all domains when compared to egg, thus preventing e-graph explosion.*

2. *The agent can learn non-obvious action sequences in order to achieve a lower cost final expression than existing solvers within the node limit specified.*

3. *Omelette is capable of optimizing expressions from any language without modification to the agent, given only a list of rewrite rules and the set of operators available.*

Additionally, we demonstrated that random search can often be superior to egg's exhaustive application, supporting our claim that rule ordering is critical for the success of e-graph construction. Furthermore, we have shown that Omelette significantly improves upon this random search benchmark by up to 50%, indicating that Deep Reinforcement Learning for Equality Saturation is effective and warrants further investigation.

Future Work

Generalization. While we have shown RL for equality saturation to be a promising new approach, challenges remain. Greatest of these is generalization: it is not always feasible to train an agent to optimize for each new expression independently. A generalizable agent, trained using behavioral cloning [41] or multi-task learning [49], would ideally be capable of solving unseen optimization tasks after having been trained on a corpus of known problems. Accomplishing this would enable RL to be widely used within real-world equality saturation solvers, as a policy could ideally be learned in advance then distributed for high-speed roll-outs by clients without additional training. While strides have recently been made in multi-task learning [33], it remains an under-discussed topic when combined with Graph Neural Networks. Additionally, a significant obstacle in accomplishing this goal is RL's high computation requirements; further work would need to be done to support multi-GPU training and other performance optimizations to make this feasible.

Additional Domains. While we have shown Omelette to be flexible in both numerical and purely symbolic domains, this merely scratches the surface of the possible problem spaces equality saturation could be used for. With this proof-of-concept established, future work may attempt applying Omelette to real-world tasks such as neural network optimization [13, 48] or automated theorem provers. This step is crucial in proving the wider effectiveness of RL as a means of term rewriting; however, scaling Omelette to these more complex languages will require larger compute resources as with generalization.

Model-Based RL. As discussed in Section 3.3.4, Proximal Policy Optimization (PPO), a stochastic on-policy algorithm, has been used to train Omelette's RL agent. While PPO was chosen due to its reliability (as described in Section 2.2.2), novel model-based RL methods may be suitable for application to Omelette now that the dynamics and limitations of E-Graph expansion are better understood. As this task is both fully observable and deterministic, it should theoretically be possible for the agent to *plan* its decisions into the future with algorithms such as DeepMind's AlphaGo and AlphaZero [37]. Exploring these alternative architectures, as well as finding means of incorporating Graph Neural Networks within them, represents a promising avenue of future research.

Bibliography

[1] Anonymous. The 37 implementation details of proximal policy optimization. In *Submitted to Blog Track at ICLR 2022*, 2022. URL https://openreview.net/forum?id=Hl6jCqIp2j. under review.

[2] Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards, 2018. URL https://arxiv.org/abs/1806.07857.

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, nov 2017. doi: 10.1109/msp.2017.2743240. URL https://doi.org/10.1109%2Fmsp.2017.2743240.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[5] Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-Yan Liu, and Liwei Wang. Graphnorm: A principled approach to accelerating graph neural network training, 2020. URL https://arxiv.org/abs/2009.03294.

[6] Nachum Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2):122–157, 1985. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(85)80003-6. URL https://www.sciencedirect.com/science/article/pii/S0019995885800036.

[7] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, 1990.

[8] Ameer Haj-Ali, Qijing Huang, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Compiler phase-ordering for high level synthesis with deep reinforcement learning, 2019. URL https://arxiv.org/abs/1901.04615.

[9] Matteo Hessel, Ivo Danihelka, Fabio Viola, Arthur Guez, Simon Schmitt, Laurent Sifre, Theophane Weber, David Silver, and Hado van Hasselt. Muesli: Combining improvements in policy optimization, 2021. URL https://arxiv.org/abs/2104.06159.

[10] Jieh Hsiang, Hélène Kirchner, Pierre Lescanne, and Michaël Rusinowitch. The term rewriting approach to automated theorem proving. *The Journal of Logic Programming*, 14(1):71–99, 1992. ISSN 0743-1066. doi: https://doi.org/10.1016/0743-1066(92)90047-7. URL https://www.sciencedirect.com/science/article/pii/0743106692900477.

[11] Shengyi Huang and Santiago Ontañ ón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, may 2022. doi: 10.32473/flairs.v35i.130584. URL https://doi.org/10.32473%2Fflairs.v35i.130584.

[12] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4-5):698–721, jan 2021. doi: 10.1177/0278364920987859. URL https://doi.org/10.1177%2F0278364920987859.

[13] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei A. Zaharia, and Alexander Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[14] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016. URL https://arxiv.org/abs/1609.02907.

[15] jan willem Klop. Term rewriting systems. 08 2000.

[16] Prasad Kulkarni and Gary Tyson. Evaluating heuristic optimization phase order search algorithms. pages 157–169, 03 2007. doi: 10.1109/CGO.2007.9.

[17] Daniel Kästner and Marc Langenbach. Code optimization by integer linear programming. pages 122–136, 03 1999. ISBN 978-3-540-65717-0. doi: 10.1007/978-3-540-49051-7_9.

[18] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015. URL https://arxiv.org/abs/1509.02971.

[19] Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJe1334YDH.

[20] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning, 2018. URL https://arxiv.org/abs/1803.07055.

[21] Nina Mazyavkina, Sergei Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *CoRR*, abs/2003.03600, 2020. URL https://arxiv.org/abs/2003.03600.

[22] José Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721–781, 2012. ISSN 1567-8326. doi: https://doi.org/10.1016/j.jlap.2012.06.003. URL https://www.sciencedirect.com/science/article/pii/S1567832612000707. Rewriting Logic and its Applications.

[23] Diego Mesquita, Amauri H. Souza, and Samuel Kaski. Rethinking pooling in graph neural networks, 2020. URL https://arxiv.org/abs/2010.11418.

[24] Nie Mingshuo, Chen Dongming, and Wang Dongqi. Reinforcement learning on graphs: A survey, 2022. URL https://arxiv.org/abs/2204.06127.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[27] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, jun 2020. doi: 10.1145/3385412.3386012. URL https://doi.org/10.1145%2F3385412.3386012.

[28] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, jun 2020. doi: 10.1145/3385412.3386012. URL https://doi.org/10.1145%2F3385412.3386012.

[29] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation, 2021. URL https://arxiv.org/abs/2108.10436.

[30] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, apr 1980. ISSN 0004-5411. doi: 10.1145/322186.322198. URL https://doi.org/10.1145/322186.322198.

[31] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIG-*

PLAN Conference on Programming Language Design and Implementation, PLDI '15, page 1–11, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737959. URL https://doi.org/10.1145/2737924.2737959.

[32] Laurent Perron and Vincent Furnon. Or-tools. URL https://developers.google.com/optimization/.

[33] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A generalist agent, 2022. URL https://arxiv.org/abs/2205.06175.

[34] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 5th edition, 2002. ISBN 0072424346.

[35] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015. URL https://arxiv.org/abs/1502.05477.

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[37] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL https://arxiv.org/abs/1712.01815.

[38] Bill Stoddart. The halting paradox, 2019. URL https://arxiv.org/abs/1906.05340.

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

[40] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), mar 2011. doi: 10.2168/lmcs-7(1:10)2011. URL https://doi.org/10.2168%2Flmcs-7%281%3A10%292011.

[41] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation, 2018. URL https://arxiv.org/abs/1805.01954.

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017. URL https://arxiv.org/abs/1710.10903.

[43] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. Spores: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(12):1919–1932, jul 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407799. URL https://doi.org/10.14778/3407790.3407799.

[44] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Why reasonable rules can create infinite loops · discussion #60 · egraphs-good/egg. URL https://github.com/egraphs-good/egg/discussions/60.

[45] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, jan 2021. doi: 10.1145/3434304. URL https://doi.org/10.1145%2F3434304.

[46] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. Carpentry compiler. *ACM Trans. Graph.*, 38(6), nov 2019. ISSN 0730-0301. doi: 10.1145/3355089.3356518. URL https://doi.org/10.1145/3355089.3356518.

[47] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2018. URL https://arxiv.org/abs/1810.00826.

[48] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization, 2021. URL https://arxiv.org/abs/2101.01332.

[49] Yu Zhang and Qiang Yang. A survey on multi-task learning, 2017. URL https://arxiv.org/abs/1707.08114.

[50] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2018. URL https://arxiv.org/abs/1812.08434.