Towards a Framework for Learning Algorithms: The Case of Learned Comparison Sorting

Philipp Kunz, **Ilche Georgievski** and **Marco Aiello**
Service Computing Department, Institute of Architecture of Application Systems
University of Stuttgart, Germany
philipp_kunz@outlook.com, ilche.georgievski@iaas.uni-stuttgart.de, marco.aiello@iaas.uni-stuttgart.de

Abstract

Designing algorithms is cumbersome and error-prone. This, among other things, has increasingly led to efforts to extend or even replace designing algorithms with machine learning models. While previous research has demonstrated that some machine learning models possess Turing-completeness, the findings are largely theoretical, and solutions for specific algorithmic tasks remain unclear. With this in mind, we investigate the feasibility of learning representations of classical algorithms from data on their execution, enabling their application to different inputs. We propose a novel and general framework for algorithm learning consisting of a model of computation that facilitates algorithm analysis across various levels of abstraction. We formalize the problem of learning an algorithm using an algebraic approach for graph traversal. We apply this framework to comparison sorts and evaluate the inferred machine learning models' performance, demonstrating the applicability of the approach in terms of accuracy and sensitivity.

1 Introduction

Reverse engineering in the context of software is the process of creating a representation of a software system at a higher level of abstraction by analyzing its components and their interrelationships [Chikofsky and Cross, 1990]. Usually, this process starts with binary code, from which a more abstract and concise representation is derived. This situation is characterized by the presence of a complete specification of the system's behavior throughout the whole process. In contrast, we assume the absence of such a specification and focus on learning an algorithm from data that reflects its manipulation of some data structure. Our ultimate goal is to derive a representation of the algorithm that allows for its application to inputs sampled from different distributions.

A precise understanding of algorithms is crucial for the ensuing discussion. Accepting the Church-Turing thesis [Turing, 1936], we can regard Turing machines and algorithms as equivalent, which allows for expressing algorithms through systemic manipulations on a tape. In practice, however, algorithms are often studied and implemented at a more abstract level, utilizing a more sophisticated data structure that provides the basic operations. Heap sort [Forsythe, 1964], for instance, is a data-structure-driven algorithm that heavily relies on the ability to determine minima using a min-heap. Since algorithms are usually formulated in terms of some underlying operations, such as those of a min-heap, our proposal must be adaptable to factor that in.

In 1995, Siegelmann et al. [1995] proved recurrent neural networks (RNNs) to be Turing-complete. Thus, an equivalent recurrent neural network exists for any algorithm. However, practical experiments have highlighted challenges in training RNNs to perform algorithmic tasks. Particularly, high error rates occurred even in the case of simple tasks, such as copying, due to memory limitations [Graves *et al.*, 2014]. In response, Graves et al. [2014] proposed the neural Turing machine (NTM) architecture, which outperforms classical RNNs on simple tasks like sorting or copying inputs. Despite these advances, both NTMs and RNNs have limitations in our context. Particularly, their training aims only at generating correct outputs for specific inputs without revealing the underlying solution method - the algorithm. This characteristic makes NTMs and RNNs less suitable for our objective of understanding and extracting algorithms themselves.

To overcome such limitations, we propose a formal and general framework for learning algorithms from data. The framework features computation graphs over homogeneous algebras as our computational model. Within this framework, we detail how a computation graph induces what we name *transition algebra*. Utilizing this framework, we define the problem of learning an algorithm from data as a classification task. This approach not only provides a formalism for learning algorithms but also offers a generalized mechanism applicable across various algorithmic contexts. Additionally, we evaluate our approach empirically using the widespread class of comparison sorts as a case study. The aim is to address the questions: (1) How does a learned model's accuracy and computational performance compare to its classical counterpart? and (2) How does the learned model's accuracy evolve as the distribution from which inputs are sampled changes?

The rest of the paper is organized as follows. Section 2 offers an introduction to the learning framework. Section 3 details the case study and evaluation. Section 4 contains a discussion, Section 5 overviews related work, and Section 6 concludes the paper.

2 Framework

We define algorithm learning as a classification task using computation graphs over homogeneous and transition algebras. Computation graphs provide operational terminology to represent computation, while transition algebras facilitate a compact and functional representation of computations. This approach results in reducing the number of classes.

2.1 Computation Graphs

A Turing machine is a kind of finite state machine augmented with an infinite tape and a read-write head. Considering this conceptualization, the level of abstraction for expressing an algorithm is determined by the tape operations, which are linked to an underlying data structure. To overcome the limitations imposed by traditional data structures, we expand the class of underlying data structures while keeping the finite state machine concept. This expansion leads to the domain of computation graphs, a computational model loosely inspired by control-flow graphs [Allen, 1970]. Computation graphs offer a structured and adaptable way to represent computational processes.

Homogeneous Algebras

Considering the limitation of traditional data structures and the interpretation of such data structures as algebras [Ehrich, 1989], we propose using arbitrary homogeneous algebras as data structures. This approach enables a more diverse and comprehensive exploration of algorithms to be learned.

Definition 1. *A* homogeneous algebra *is a tuple $(M, (f_i)_{i\in I})$, where M is a set and $(f_i)_{i\in I}$ an indexed family of functions of finite arity on M with I being a set of indexes.*

A *configuration* is an element $m \in M$. By $M_f := \bigcup_{i\in I} \{f_i\}$ we denote the set of operations on M_f.

Properties and Property Spaces

In the course of a single step governed by the transition function $\delta : \gamma \times Z \to \gamma \times Z \times \{R, L, N\}$, a Turing-machine machine evaluates only a single cell of its infinite tape. We want to transfer this concept of localized evaluation to computation graphs, guiding us to consider properties and property spaces within computation graphs.

Definition 2. *A property π is a mapping $\pi : M \to W$, where $(M, (f_i)_{i\in I})$ is a homogeneous algebra and W is a non-empty set.*

A property π extracts some information from a configuration $m \in M$ and represents it as an element of W. Usually, there are multiple properties of interest, $\pi_1, ..., \pi_n$. By imposing an order on them and applying them accordingly, we obtain an ordered tuple of measurements, $(\pi_1(m), ..., \pi_n(m))$. If we repeat this for all $m \in M$, we obtain a property space, which is, vividly speaking, the set of all observations for all configurations.

Definition 3. *Let $(M, (f_i)_{i\in I})$ be an an algebra and $(\pi_1, ..., \pi_n)$ a tuple of properties.*

The mapping $\Pi_{(\pi_1,...,\pi_n)}$ is defined as follows:

$$\Pi_{(\pi_1,...,\pi_n)} : M \to W_1 \times ... \times W_n, m \mapsto (\pi_1(m), ..., \pi_n(m)), \quad (1)$$

Definition 4. *A property space $P^M_{(\pi_1,...,\pi_n)}$ is the image of M under $\Pi_{(\pi_1,...,\pi_n)}$, that is, $P^M_{(\pi_1,...,\pi_n)} := \Pi_{(\pi_1,...,\pi_n)}(M)$.*

Computation Graphs Definition

Having all the necessary building blocks, we can now define computation graphs. In our computation graph, each node is associated with a function from the homogeneous algebra. Under such a setting, a computation corresponds to traversing the graph and systematically applying the function associated with each node. The order in which nodes are traversed and their functions are applied is governed by directed edges.

To eliminate ambiguity in the valid paths through the graph, each edge is assigned a predicate. These predicates play an important role in determining the traversal path. For every non-terminal node, there is precisely one outgoing edge whose assigned predicate evaluates to true at a given time. This mechanism guarantees a clear and deterministic progression through the graph, allowing for precise computation.

Definition 5. *Let $(M, (f_i)_{i\in I})$ be an algebra, $(\pi_1, ..., \pi_n)$ a tuple of properties, and Q_M a non-empty set of predicates, where f_i is of arity one $\forall i \in I$. Then a* computation graph *G is a 7-tuple $G = (V, E, M, \phi, \gamma, s, t)$, where*

1. *V is a finite set of nodes with $s \in V$ being the initial node and $t \in V \setminus \{s\}$ being the terminal node,*
2. *$E \subset V^2$ is a finite set of directed edges such that $\forall v \in V : (v, s) \notin V$ and $(t, v) \notin V$, and $\forall v_1 \in V \setminus \{t\} \exists v_2 \in V : (v_1, v_2) \in E$,*
3. *$\phi : V \to M_f$ assigns each node a function such that $\phi(t) = identity$,*
4. *$\gamma : E \to Q_M$ assigns each edge a predicate such that:*
   - $\forall p \in P^M_{(\pi_1,...,\pi_n)} \forall v_1 \in V \setminus \{t\} \exists v_2 \in V : (v_1, v_2) \in E \wedge \gamma(v_1, v_2)(p) = 1$
   - $\forall p \in P^M_{(\pi_1,...,\pi_n)} \forall (v_1, v_2), (v_1, v_3) \in E : (v_1, v_2) \neq (v_1, v_3) \wedge \gamma(v_1, v_2)(p) = 1 \implies \gamma(v_1, v_3)(p) = 0$

By $identity$ we refer to the identity $M \to M$. Therefore, there must be an $i \in I$ for which $f_i = identity$ holds.

Given a computation graph $G = (V, E, M, \phi, \gamma, s, t)$, we refer by G' to the directed graph $G' = (V, E)$.

Computation

In the present treatment a *computation* is a sequence of states $(v, m) \in V \times M$, where each transition from a state (v_i, m_i) to the subsequent state (v_{i+1}, m_{i+1}) goes in line with an underlying computation graph G. To articulate the legitimacy of state transitions, we employ a transition relation. This relation establishes the conditions that must be met for a transition between states to be considered valid.

Definition 6. *Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph and $(v_i, m_i), (v_{i+1}, m_{i+1}) \in V \times M$ be states. A transition relation $\vdash_G$ between (v_1, m_1) and (v_2, m_2), denoted as $(v_i, m_i) \vdash_G (v_{i+1}, m_{i+1})$, is* valid *iff $(v_i, v_{i+1}) \in E \wedge \gamma(v_i, v_{i+1})(\Pi_{(\pi_1,...,\pi_n)}(m_1)) = 1 \wedge m_2 = \phi(v_i)(m_1)$.*

Considering the definition of the transition relation, every non-terminal state in the computation graph has exactly one successor.

Theorem 1. *Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph. Then, $\forall(v_1, m_1) \in (V \backslash \{t\} \times M)\ \exists!(v_2, m_2) \in V \times M$ such that $(v_1, m_1) \vdash_G (v_2, m_2)$ holds.*

The proof of Theorem 1, along with the proofs of all subsequent theorems, can be found in the supplementary material. We release the supplementary material as well as the code of our experiments at *https://github.com/pkunz96/algorithm-learning*.

Theorem 2. *Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph. Then, $\forall(v_1, m_1) \in V \times M : (v_2, m_2) \in \{t\} \times M \implies (v_2, m_2) \nvdash_G (v_1, m_1)$.*

Combining $\vdash_G$ with the uniqueness of a state successor as established by Theorems 1 and 2, we can formalize the notion of a computation.

Definition 7. *Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph and $m_1 \in M$ an initial configuration. A* computation *is a sequence $(s_1, m_1), (s_2, m_2), ... \in V \times M$ such that $(s_1, m_1) \vdash_G (s_2, m_2) \vdash_G \dots$ holds.*

A computation with initial tuple (s, m_1) is considered *terminating* if there exists (t, m_n) such that $(s, m_1) \vdash_G ... \vdash_G (t, m_n)$ without a successor under $\vdash_G$.

Corollary 3. *Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph, $m_1 \in M$ an initial configuration, and $\vdash_G^+$ a transitive hull of $\vdash_G$. Then, $(s_1, m_1) \vdash_G (s_2, m_2) \vdash_G ...is finite \iff \exists m_2 \in M : (s, m_1) \vdash_G^+ (t, m_2)$.*

The question of which algorithms can be formulated as computation graphs is crucial in determining the suitability of computation graphs as models of computation. Since we accept the Church-Turing thesis [Turing, 1936], it suffices to establish the existence of a suitable algebra and confirm that the computation graph derived from this algebra has the property of Turing completeness. This ensures that the computation graph is capable of simulating any algorithm that a Turing machine can, thereby validating the computation graph as a computational model.

Theorem 4. *For a given deterministic Turing machine $T := (Z, \Gamma, \Sigma, \delta, z_0, b, t_T)$, there exist a homogeneous algebra $(M, (f_i)_{i \in I})$ such that a corresponding equivalent computation graph $G = (V, E, M, \phi, \gamma, s, t_G)$ can be constructed, where for any initial tape configuration $\alpha z_0 \beta$, there exist an initial configuration $m_1 \in M$ such that $\alpha z_0 \beta \vdash_T^+ \alpha' t_T \beta' \iff (s, m_1) \vdash_G^+ (t_G, m_2)$.*

2.2 Transition Algebra

A simplistic approach towards learning an algorithm from data would be training a classifier to predict the correct successor (z_{i+1}, m_{i+1}) for a given state (z_i, m_i) with respect to $\vdash_G$. However, this approach is unfeasible as the number of classes is potentially infinite, making standard classification algorithms ineffective. Instead, we can predict functions $a_i : V \times M \to V \times M$. To define such functions effectively, we require a compact functional representation of graph traversals, such as $v_2 = f_n \circ f_{n-1} \circ \dots \circ f_1(v_1)$.

To minimize the number of functions a_i and, therefore, the number of classes a classifier must be able to distinguish, it is essential to limit and minimize the number of functions $f_1, \dots, f_n$ required to represent all possible graph traversals. Considering these requirements and leveraging the ordering imposed by the topology of the transition graph, we are led to the concept of transition algebra, a framework that allows for efficient and structured learning of algorithms. Considering a directed graph, fundamental to the concept of the transition algebra are the $suc(v_i)$ and V_{deg}, where $suc(v_i)$ is the number of successors of the node v_i and V_{deg} is the set of nodes that have a predecessor with multiple successors.

Definition 8. *Let $G = (V, E)$ be a directed graph. Then $suc : V \to N$ is defined as follows*

$$suc(v_i) := |\{v_j \in V | (v_i, v_j) \in E\}| \tag{2}$$

and V_{deg} is defined as:

$$V_{deg} = \{v_2 \in V | \exists v_1 \in V : (v_1, v_2) \in E \wedge suc(v_1) > 1\} \tag{3}$$

Consider a node $v_i \in V_{deg}^{\complement}$. By definition, v_i has no predecessor, or it is the single successor of its predecessor v_j. In such cases, a single function, $transition_\epsilon$, suffices to describe the transition from v_j to its successor v_i. On the other hand, when $v_i \in V_{deg}$, v_i has a predecessor v_j with multiple successors, such as $v_i, v_{i+1}, \dots, v_{t+k}$. In this situation, explicitly defining the transition target from v_j is necessary. This requirement can be encapsulated through a family of functions: $transition_{v_i}(v_j) = v_i, \dots, transition_{v_{t+k}}(v_j) = v_{t+k}$.

Definition 9. *Let $G = (V, E)$ be a directed graph. A* transition algebra T_G induced by G *is a tuple $T_G = (V, (transition_i)_{i \in I})$, where for all $i \in I := V_{deg} \cup \{\epsilon\}$:*

$$transition_\epsilon(v_i) = \begin{cases} v_j & \text{if } suc(v_i) = 1 \wedge (v_i, v_j) \in E \\ v_i & \text{else} \end{cases}$$

$$transition_{v_j}(v_i) = \begin{cases} v_j & \text{if } (v_i, v_j) \in E \\ v_i & \text{else} \end{cases}$$

Following this definition, we say G induces the transition algebra T_G. We denote by $T_{transition}^G$ the set of functions on T_G, $T_{transition}^G := \bigcup_{i \in I} \{transition_i\}$.

In our pursuit of representing any graph traversal as a composition of functions $f_i : V \to V$, it is essential to ensure that every composition corresponds to a graph traversal and vice versa. For technical clarity, we first introduce a predicate $connected_G^n(v_i, v_j)$, which determines whether a path of length n exists connecting node v_i with node v_j in the graph G. This predicate enables establishing the following theorems, which collectively substantiate the desired correspondence property.

Definition 10. $connected_G^n(v_i, v_j) :\iff$
$\exists(v_i, ..., v_j) \in V^n \forall 1 \leq i \leq n-1 : (v_i, v_{i+1}) \in E$.

Theorem 5. *Let $T_G = (V, (transition_i)_{i \in I})$ be a transition algebra induced by a directed graph G. Then, for t_1, ..., $t_{n-1} \in T_{transition}^G$, it holds that $\forall n \geq 2 : \forall(v_1, ..., v_n) \in V^n : connected_G^n(v_1, v_n) \implies v_n = t_{n-1} \circ ... \circ t_1(v_1)$.*

Theorem 6. *Let $G = (V, E)$ be a directed graph and $T_G = (V, (transition_i)_{i \in I})$ an induced transition algebra. Then, for all $v_i, v_j \in V$, $v_i \neq v_j$ it holds that* $\forall n \geq 0 \forall (v_1, ..., v_{n+2}) \in \{v_i\} \times V^n \times \{v_j\}$ $\forall k \geq 1 : (t_1, ..., t_k) \in T_G^k \implies v_j \neq t_k \circ ... \circ t_1(v_i)$.

2.3 Learning Problem

We are ready to formalize the task of learning an algorithm within the supervised learning framework, specifically as a classification task. This is grounded on the abstract computation graph $G = (V, E, M, \phi, \gamma, s, t)$ with the assumption that both T_G and M_f are known entities.

For a given arbitrary transition $(t(v_i), f(m_i)) = (v_{i+1}, m_{i+1})$, we can show the existence of $(t, f) \in T_G \times M_f$ for which $(t(v_i), f(m_i)) = (v_{i+1}, m_{i+1})$ holds. Our aim is to represent computations as chains of function applications, starting with an argument $(v, m) \in V \times M$. To achieve this, we combine each pair $(t, f) \in T_G \times M_f$ into a single function, which we refer to as an *actuator*.

Definition 11. *Let $f_t \in T^G_{transition}$ and $f_m \in M_f$. An* actuator *is a function $a_{f_t f_m} : V \times M \to V \times M$ such that $(v, m) \mapsto (f_t(v), f_m(m))$.*

For a given $T^G_{transition}$ and M_f, the set of all actuators A_G is defined as $A_G := \{a_{f_t f_m} | (f_t, f_m) \in T_G \times M_f\}$. An actuator $a_{f_t f_m}$ is fully characterized by the corresponding functions $f_t \in T^G_{transition}$ and $f_m \in M_f$. We can, therefore, predict tuples $(f_t, f_m) \in T_G \times M_f$ when actually an $a_{f_t f_m} \in A_G$ is required.

With regard to our goal of representing computations as a concatenation of actuators, it remains to be clarified how a sequence (v_1, m_1), ..., (v_n, m_n) can be mapped to an equivalent actuator sequence a_1, ..., a_{n-1} such that $(v_n, m_n) = a_n \circ ... \circ a_1(v_1, m_1)$.

With this in mind, we introduce the mapping γ_G, which assigns the correct actuator to a state (v, s).

Definition 12. *An* actuator-state mapping *γ_G is a function $\gamma_G : V \backslash \{t\} \times M \to M_f \times T_G$ such that $(v, m) \mapsto (f_t, \phi(v))$, where $f_t \in A := \{t \in T_G | \phi(v, t(v))(m) = 1\}$ and $|A| \leq 1$ holds.*

By applying γ_G successively to the first $n - 1$ states of a sequence of transitions, we obtain the desired sequence of actuators.

Supervised Learning Task

Earlier, we established the existence of countably many infinite algebras for which the resulting computation graph is Turing-complete. As a result of this proof, we can focus our further discussion on countably infinite carrier sets M, thereby eliminating the need for measure-theoretic considerations in our discussion. Let $(V \times M, \mathcal{P})$ be a discrete probability space. We identify a random variable X with an identity function, i.e., $X : V \times M \to V \times M$ such that $(v, m) \mapsto (v, m)$; and Y with γ_G, i.e., $Y : V \times M \to A_G$ such that $(v, m) \mapsto \gamma_G(v, m)$. In this context, Y represents a categorical response, as the image set is finite. We choose $L(X, Y)$ as the zero-one loss function, penalizing each wrong classification with uniform cost [Hastie *et al.*, 2009]. The corresponding search space F is thus defined as $F := \{f | f : V \times M \to A_G\}$.

Insofar the corresponding supervised learning problem boils down to finding the function in F that minimizes the expected prediction error:

$$argmin_{f \in F} EPE(f) := E(L(f(X), Y) \tag{4}$$

Given our choice of loss function, the following holds true for an optimal $f \in F$ according to [Hastie *et al.*, 2009]:

$$f(v, m) = argmax_{a \in A_G} P(Y = a | X = (v, m)) \tag{5}$$

The process of generating training data basically involves drawing samples $(v_1, m_1), \ldots, (v_n, m_n)$ from a common distribution and then applying the function γ_G. The resulting dataset, denoted as D, is thus defined as $D := \{(v_1, m_1, \gamma_G(v_1, m_1)), \ldots, (v_n, m_n, \gamma_G(v_n, m_n))\}$.

Learning the Transition Algebra

To address the question of how T_G can be derived from training data, we assume n computations as input, that is, $(v_{11}, m_{11}) \vdash_G ... \vdash_G (v_{nm_1}, m_{nm_1}) \ldots (v_{n1}, m_{n1}) \vdash_G ... \vdash_G (v_{nm_n}, m_{nm_n})$. Based on Definition 6, we have:

$$(v_i, m_i) \vdash_G (v_{i+1}, m_{i+1}) \overset{6}{\implies} (v_i, v_{i+1}) \in E \tag{6}$$

These settings allow the reconstruction of an edge set $E' \subset E$ by successively applying the implication to all transitions and adding the resulting tuple to the set E'. Once we have established E', we recover the corresponding set of nodes V' using the following union of projections:

$$V' = \bigcup_{e \in E'} \{\pi_0(e), \pi_1(e)\} \tag{7}$$

where π_i is a projection extracting the i-th component from a tuple. Since the computation graph is generally unknown, we cannot directly verify whether the conditions $V' = V$ and $E' = E$ are satisfied, and we must proceed under the assumption that these conditions hold.

Classification Algorithm

The basic supervised learning problem is to predict an actuator $a \in A_G$ for a given configuration $(v, m) \in (V \backslash \{t\}) \times M$ such that $(v, m) \vdash_G a(v, m)$. As a consequence of Definition 11, determining such an actuator reduces to identifying the corresponding $t \in T_G$ and $f \in M_f$. This process enables us to distinguish between predictions that are dependent on the distribution of the input and those that are independent of it.

The function ϕ can be directly inferred from the training data. We proceed under the assumption that a training data set exists defined as $D \subset (V \backslash \{t\}) \times M \times T_G \times M_f$. Given this dataset, and in line with Definition 5, it follows that for all $(v_i, m_i, t_i, f_i) \in D$: $f_i = \phi(v_i)$. Consequently, if the training data set D contains a tuple for each $v \in V$, then ϕ can be inferred. This inference can be systematically achieved using Algorithm 1.

For a given tuple (v, m), we can reduce the problem of predicting the correct actuator to determining the correct $f_t \in T^G_{transition}$ and $f_m \in M_f$. The function f_m can be easily inferred using ϕ, which can be derived from the training data as previously discussed, $f_f = \phi(v)$. In the case of

Algorithm 1

Input: $D := [(v_1, m_1, t_1, f_1), \ldots, (v_n, m_n, t_n, f_n)]$ - an array of training data

```
φ ← []
i ← 0
while i ≤ D.length do
    φ[D[i][0]] ← D[i][3]
    i ← i + 1
end while
```

determining $f_t \in T^G_{transition}$, we have to differentiate based on G:

1. If $suc_G(v) \leq 1$, $f_t = suc$.
2. If $suc_G(v) > 1$, $f_t \in \{t \in T_G | \phi(v, t(v)) = 1\}$.

Regarding the functions of the data structure $f_m \in M_f$, it is clear that their selection solely depends on $v \in V$ and is, therefore, independent of the distribution of input data. Conversely, for functions of the transition algebra T_G, the decision about which edge is passable for the given $m \in M$ is determined by a predicate $q : P^M_{(\pi_1,\ldots,\pi_n)} \rightarrow \{0, 1\}$. This predicate, therefore, dictates the correct $t \in T_G$.

In general, we cannot directly infer these predicates from the data. Still, we can employ traditional machine learning algorithms to train a classifier, referred to as *classify*, to evaluate them. If a computation based on a state graph $G = (V, E)$ arrives at a configuration $(v, m) \in V \times M$, we have to distinguish two cases: given that the corresponding state v has at most one successor, so that $suc_G(v) \leq 1$ holds, the successor state t can be directly inferred from G. Otherwise, we can employ $classify$ and set $t \leftarrow classify(v, m)$. Using the function $\phi : V \rightarrow M_f$, we can determine the succeeding configuration by evaluating $(t, \phi(v)(m))$. Following this approach, we can integrate input-independent knowledge with the evaluation of input-dependent predicates.

3 Case Study

We apply the framework to classic examples of comparison sorts to evaluate the achievable performance in terms of accuracy. For this, we train respective machine learning models, which implement $classify$.

3.1 Algorithm Taxonomy

Comparison sorts form a class of sorting algorithms that determine the sorted order of input elements by performing their pairwise comparison based on an order relation, such as $\leq$ [Cormen *et al.*, 2009]. Our analysis of the algorithms listed in Table 1 leads to the inference of a sorting algebra, as illustrated in Figure 1. Building on this foundation, we develop a taxonomy of comparison sorts, taking into consideration factors such as Lines of Code (LoC), Cyclomatic Complexity (CC), and the Dimensionality of Training data (DoT). A training data point includes register and stack cells, referenced array cells, and the integer-encoded current state. Therefore the dimensionality equals the number of cells plus one. To identify similarities among these algorithms, we applied Lloyd's algorithm [Lloyd, 1982], which leads to the four clusters presented in Table 1.

	LoC	CC	DoT
Cluster 1			
Comparison Counting	88	4	8
Straight Insertion Sort	82	4	9
Straight Selection Sort	79	4	8
Shell Sort	133	5	11
Bubble Sort	63	4	7
Cluster 2			
Quick Sort	184	5	19
Binary Insertion Sort	199	6	19
Cluster 3			
Tree Selection	392	8	25
Tree Selection By Iverson	338	8	21
Top-Down-Merge-Sort	391	12	25
Bottom-Up-Merge-Sort	309	11	23
Cluster 4			
Heap Sort	328	15	21
Natural-Two-Way-Merge-Sort	371	17	22

Table 1: Algorithm taxonomy in terms of sorting algebra functions.

3.2 Experimental Setup

We perform two sets of experiments. First, we train machine learning models to implement $classify$. Second, we evaluate the impact of changing the distributions inputs are sampled from. Experiments were conducted on a 2019 MacBook Pro 16" equipped with an Intel Core i7 processor and 16GB of RAM. The software environment included MacOS v11.7.6, Python v3.8.1, TensorFlow 2.12.0, NumPy 1.23.5, and Matplotlib 2.7.1.

Experiment: Classifier Training

We train feed-forward neural networks to predict the actuators for various sorting algorithms, namely heap sort, insertion sort, quick sort, and top-down merge sort, each representing a cluster in our taxonomy. The training data for these algorithms is generated by drawing samples from $(I, \mathcal{P}_\mathcal{I})$, where $I := \bigcup_{i \geq 0} Z^i$.

Definition 13. *Let $X_0 \sim N(\mu_0, \sigma_0)$ and $X_1, X_2, \cdots \sim N(\mu_1, \sigma_1)$ be a sequence of independent and equally distributed random variables, then $P_I : \mathcal{P}(I) \rightarrow [0, 1]$ is defined as $P_I(A) := \sum_{(a_1,\ldots a_k) \in A} P(\lceil |X_0| \rceil = k, ..., \lceil X_k \rceil = a_k)$.*

For this experiment, we set the parameters as follows: $\mu_0 = 20, \sigma_0 = 5, \mu_1 = 0, \sigma_1 = 100$. We leverage our knowledge of the structure of sorting algebra for feature extraction. In particular, we consider that the chosen comparison sorts typically involve the element at the top of the stack, the register file, the register file at the top of the stack register, and elements in the array referenced by indices. Therefore, only these parts of the data structure are incorporated into each training data point during feature extraction.

In our approach to hyperparameter optimization, we conduct a grid search with an emphasis on simplicity in network architecture. Thus, we consider neural networks with at least one to a maximum of three hidden layers. For activation functions, we allow 'relu' [Hahnloser *et al.*, 2000] and 'sigmoid' [Goodfellow *et al.*, 2016] in all layers except the

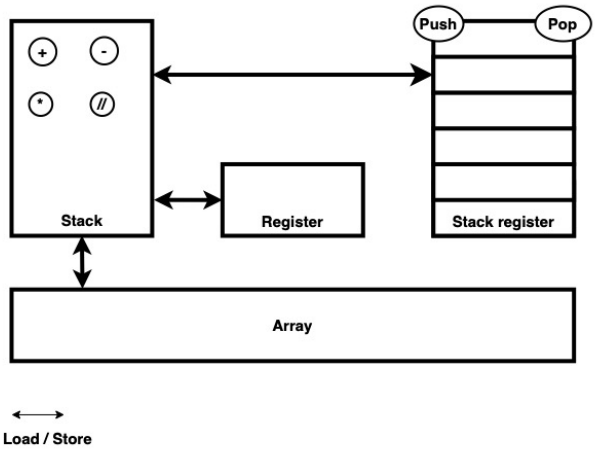

Figure 1: A visualization of the structure of sorting algebra, consisting of a stack, a register characterized by a fixed-size linear address space, a stack of registers, and an array of variable length. The bidirectional arrows indicate the flow of data between these components. The instruction set, M_f, includes functions to push constants ($push\_constant_i$) or metadata ($push\_array\_length$) onto the stack. An operation $\circ \in \{+, -, *, //\}$ performs integer arithmetic operations using the top stack elements, whereas $load$ and $store$ are responsible for handling data flow.

output one, where softmax [Goodfellow *et al.*, 2016] is applied. From a theoretical point of view, we expect improved performance on validation data as the quantity of training data increases. This forces the model to generalize better while reducing variance. To this end, we trained models using training data sets of sizes 2^e, where $13 \leq e \leq 16$.

Regarding the selection of a learning rate, we choose $\alpha \in \{0.1, 0.01, 0.001\}$ as parameter space. Preliminary experiments consistently indicated a monotonically decreasing trend in validation losses. Based on these observations, we kept the number of training epochs constant at 1000.

Experiment: Changing the Distribution of Inputs

Recall that our goal is to derive a representation of an algorithm from data and apply this knowledge to inputs drawn from different distributions. To assess the impact of changing the input distribution, we undertake the following procedure for each algorithm and its corresponding best model trained using 16384 samples. We perform two series of trials, one with $\mu = 0$ and $\sigma = 150$ and another with $\mu = 100$ and $\sigma = 100$. This approach allows us to maintain one of the parameters constant relative to the previous experiment, providing a controlled variation in the input distribution. For each parameter configuration and selected model, we repeat the entire training process, including the generation of training data, five times. This repetition is important for obtaining more robust and reliable estimates of $E(A)$ and $E(A_i)$, computed as the arithmetic mean of the outcomes from these individual iterations.

Insertion Sort	Heap Sort	Quick Sort	Merge Sort
Count:	4096		
T_0	T_0	T_0	T_0
T_{71}	T_{153}	T_{102}	T_{169}
T_{18}	T_{154}	T_{21}	T_{127}
Count:	8096		
T_0	T_0	T_0	T_0
T_{71}	T_{153}	T_{102}	T_{169}
T_{18}	T_{154}	T_{21}	T_{127}
Count:	16384		
T_0	T_0	T_0	T_0
T_{71}	T_{153}	T_{102}	T_{169}
T_{18}	T_{154}	T_{21}	T_{127}

Table 2: Expected least frequent labels per algorithm and sample size.

Algorithm	**Layers**	$E_{(A_{\mu=100})}$	$E_{(A_{\sigma=150})}$
Insertion Sort	r,s,r,σ/512	0.155	0.578
Heap Sort	r, s, σ/512	0.155	0.578
Quick Sort	r, r, r, s σ/512	0.715	0.718
Merge Sort	r, r, σ/256	0.600	0.650

Table 3: Achieved accuracy as the μ the σ as parameters of the distribution of inputs are varied. $E(A_{\mu=100})$ refers to the accuracy achieved for $\mu = \sigma = 100$ and $E(A_{\sigma=150})$ the respective quantity for $\mu = 0$ and $\sigma = 150$.

3.3 Results

Experiment: Classifier Training

For the model evaluation, we calculate the achieved accuracy using separate test sets. Given the high imbalance in the training data, we estimate the probability A_i of a correct prediction for individual classes $C_1 \ldots C_n$, where C_i is a true class. We summarize the top five hyperparameter configurations for each sorting algorithm and dataset size in the supplementary material. These configurations are ranged based on two criteria: the estimated value of the minimum expected conditional accuracy, determined by $argmin_{i \in \{1,\ldots,n\}} E(A_i)$, and the estimated value of the overall expected accuracy, $E(A)$. In the case of heap sort and top-down merge sort, we observe a significant difference between $E(A)$ and $E(A_i)$. In reference to the class C_i with minimal $E(A_i)$, we observe that in all four cases, this class is the one also listed in Table 2, indicating that this is a class that was represented in the training data to a comparatively small extent.

Experiment: Changing of the Distribution of Inputs

Table 3 shows the results of changing the input distribution. We can observe a significant drop in accuracy for insertion sort and heap sort due to the altered expectation value μ. In contrast, this phenomenon appears more moderate in the cases of quick sort and merge sort. Additionally, it is noticeable that changes in the variance σ lead to a decrease in accuracy, but this impact varies in magnitude, being significantly less pronounced in certain instances.

4 Discussion

The proposed framework is a general framework for learning representations of standard algorithms. The framework's foundation is computation graphs over homogeneous algebras, providing a computation model for analyzing algorithms at various levels of abstraction. This model, due to its Turing completeness, is capable of any arbitrary computation performed by a Turing machine, making our framework a general tool for learning algorithmic representations.

The introduction of transition algebra in the framework is crucial as it allows us to conceptualize algorithmic computations as a process of chaining actuators, enabling us to frame the task of learning an algorithm as a supervised learning problem. We demonstrate that a significant importance of transition algebra is its role in reducing the number of classes within the context of the classification task, potentially leading to a reduced quantity of required training data.

Properties are either manually inferred or must be automatically derived by the respective machine learning algorithm in the course of training. A central requirement unaffected by that is the need for training data, which allows for the distinction between payload and state, where the payload corresponds to the elements of the underlying algebra and the state to a node of the computation graph.

Focusing on applying the framework on comparison sorts as a representative of standard algorithms, we investigate which machine learning approaches are suitable for these algorithms and what the performance characteristics are. Our findings indicate that feedforward networks with relatively simple architectures, consisting of at most two hidden layers and a maximum of 512 neurons per layer, are promising. However, the practical adequacy of achieving an overall accuracy of 95% or higher, as achieved for insertion sort, quick sort, and heap sort, requires further investigation. For example, given the high volume of predictions for even small input sequences, an error rate of 0.05 could result in a significant number of incorrect predictions. Also, our results show an increase in accuracy with the increase in training data size.

Our first experiment revealed that classes with low presence in the training data have lower sensitivity. Moreover, the achieved accuracy declines with the increase in the cyclomatic complexity of the algorithms and the dimensionality of the training data. Our second experiment highlighted that the changes in the expected values of the input distributions particularly impact the accuracy. In contrast, the change of variance seems to have a less significant effect. This finding suggests that our initial goal to derive an algorithm representation applicable to different input distributions has not been fully realized. The challenges observed in adapting to varying input distributions point towards the need for further research in this area.

5 Related Work

Previous research in machine learning has primarily focused on enhancing deterministic algorithms by integrating machine-learning methods. One can distinguish several cases of integration. One case involves using the empirical distribution function (CDF), which is inferred as an approximation of the actual commutative distribution function. Subsequently, this knowledge is used to directly compute the final or an intermediate solution [Kristo *et al.*, 2020; Kristo *et al.*, 2021; Carvalho and Lawrence, 2023]. Another strategy uses CDF to accelerate lookups in ordered data structures [Amato *et al.*, 2021]. Furthermore, Axtmann et al. [2021] use decision trees trained on common inputs to partition an input in a deterministic manner. In contrast to the previous approaches, the work on SageDB focuses on replacing entire database subsystems with machine learning models, such as a differentiable and trainable cost model for query optimization [Kraska *et al.*, 2019]. In all these cases, the application of machine-learning techniques is embedded in a deterministic algorithmic context. In contrast, our work focuses on deriving machine-learning models that represent deterministic algorithms as whole entities. Our overall goal is to transfer this knowledge to inputs sampled from varying distributions, whereas the works above assume the distribution to be unchanged. Finally, Petry and Biermann [1976] reconstruct control flow graphs from sequences of instructions and memory traces. In contrast to our framework, there is no mechanism to derive predicates governing branching at runtime.

6 Conclusion

We proposed a novel general framework for learning algorithms. The framework uses computation graphs over homogeneous algebras to analyze algorithms at multiple abstraction levels and transition algebras to simplify the learning process. Our investigation into machine learning approaches for learning comparison sorts highlighted feed-forward networks with simple architectures as effective. However, the practicality of their achieved accuracy is debatable due to potential errors in large-scale predictions. Our experiments indicate that low presence in training data correlates with low sensitivity and that algorithm accuracy decreases with increasing algorithm complexity and training data dimensionality. Furthermore, we observed that accuracy is more affected by changes in the expected value of input distributions than by changes in variance, suggesting challenges in achieving our goal of applying algorithm representations to varying inputs.

Further efforts should be made to place the evaluation of the framework on a broader basis. In particular, we plan to apply the learned sorting algorithms to concrete inputs to evaluate the practical implications of the achieved sensitivity values. Regarding the work of Petry and Biermann [1976], it remains to be evaluated whether the state graph can be learned even if the training data no longer comprises state information but only a sequence of instructions. In the future, our framework may advance the field of explainable AI by allowing for the inference of comprehensible and deterministic algorithms from data. Additionally, one can explore the framework's potential for the development of approaches for domain-specific optimizations in applications where available data might allow for optimizations that classical or generic algorithms cannot achieve. Such type of data has consistent and/or predictable patterns. For instance, a sorting algorithm could optimize sorting and recommendations based on data on user behavior in the context of e-commerce.

References

[Allen, 1970] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[Amato *et al.*, 2021] Domenico Amato, Raffaele Giancarlo, and Giosuè Lo Bosco. Learned sorted table search and static indexes in small space: Methodological and practical insights via an experimental study. *CoRR*, abs/2107.09480, 2021.

[Axtmann *et al.*, 2021] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms, 2021.

[Carvalho and Lawrence, 2023] Ivan Carvalho and Ramon Lawrence. Learnedsort as a learning-augmented samplesort: Analysis and parallelization. In *35th International Conference on Scientific and Statistical Database Management*, SSDBM 2023. ACM, July 2023.

[Chikofsky and Cross, 1990] Elliot Chikofsky and James H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[Cormen *et al.*, 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Ehrich, 1989] Gogolla Lipeck Ehrich. *Algebraische Spezifikation abstrakter Datentypen*. B.G. Teubner Stuttgart, Stuttgart, 1989.

[Forsythe, 1964] George E. Forsythe. Algorithms. *Commun. ACM*, 7(6):347–349, 1964.

[Goodfellow *et al.*, 2016] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016.

[Graves *et al.*, 2014] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *CoRR*, abs/1410.5401, 2014.

[Hahnloser *et al.*, 2000] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and Hyunjune Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 408(6815), 2000.

[Hastie *et al.*, 2009] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[Kraska *et al.*, 2019] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *Biennial Conference on Innovative Data Systems Research*, 2019.

[Kristo *et al.*, 2020] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The Case for a Learned Sorting Algorithm. In *ACM SIGMOD International Conference on Management of Data*, pages 1001–1016, 2020.

[Kristo *et al.*, 2021] Ani Kristo, Kapil Vaidya, and Tim Kraska. Defeating duplicates: A re-design of the LearnedSort algorithm. *CoRR*, abs/2107.03290, 2021.

[Lloyd, 1982] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[Petry and Biermann, 1976] Frederick E. Petry and Alan W. Biermann. Reconstruction of Algorithms from Memory Snapshots of Their Execution. In *ACM Annual Conference*, pages 530–534, 1976.

[Siegelmann and Sontag, 1995] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

[Turing, 1936] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.