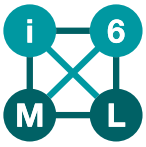


Chair of Machine
Learning and Reasoning
(Computer Science 6)

RWTH AACHEN
UNIVERSITY

This thesis was submitted to the Chair of Machine Learning and Reasoning.

Logical Distillation of General Policies From GNNs

Master Thesis

Presented by

Dominik Hoenzelaer
444524

Supervised by Niklas Jansen, M.Sc.

1st Examiner Prof. Hector Geffner, Ph.D.

2nd Examiner Prof. Dr. rer. nat. Christopher Morris

Aachen, March 31, 2026

I sincerely thank my parents for all their support throughout my studies,
especially during the final months of writing this thesis.

Abstract

In modern artificial intelligence (AI), creating goal-oriented agents that can generalize solutions across different tasks is a central challenge. Symbolic approaches to generalized planning offer interpretable solutions, but rely on handcrafted features or predefined feature pools. On the other hand, recent graph neural network (GNN)-based approaches have demonstrated that they can successfully learn general solutions in the form of policies and value functions without requiring predefined feature pools. However, the neural approaches lack the transparency offered by symbolic solutions.

This thesis addresses this gap between symbolic and neural methods by introducing relational IDTs (R-IDTs), which are designed to distill the knowledge of a trained GNN for generalized planning into symbolic formulas. R-IDTs are a direct generalization and extension of iterated decision trees (IDTs), adapted to handling the multi-relational structures of planning states. The proposed method reflects the layerwise neural calculations performed by a GNN in a symbolic basis by fitting decision trees on logical properties and embeddings provided by the GNN. To predict the optimal value function required for generalized planning, R-IDTs derive state-level features and learn a linear regression.

The method is experimentally evaluated on five generalized planning benchmark problems. The results demonstrate that R-IDTs can learn perfectly generalizing value functions for bounded problems in the *Gripper* and *Miconic* domains. For unbounded problems in the *Blocksworld* and *Visitall* domains, models fit the training data perfectly, but generalizing provides a challenge to the models. The experiments further reveal that the learned features are described by formulas that are too complex to be understood. Nonetheless, this work provides a starting point for future research to further bridge the gap between neural and symbolic methods for generalized planning.

Contents

1 Introduction

Creating goal-oriented agents that can generalize solutions and exploit similarities between tasks is arguably an important part of modern artificial intelligence (AI) [41]. The AI research field of planning [21, 41] is concerned with creating goal-oriented agents that solve tasks by generating a plan of actions such that following the plan sequentially achieves a set goal condition. While classical planning is concerned with solving individual problems by finding plans, generalized planning [2, 8, 12, 31, 43, 44] is concerned with finding solutions in a general form that solve a whole class of planning problems, for example, general policies or value functions.

Most approaches to solve classical and generalized planning problems are symbolic and result in human-readable solutions. However, these methods rely on handcrafted features that give relevant information about the agent's environment, or on large predefined feature pools from which an agent can select relevant features [11, 12, 19, 20]. In contrast, recent research has explored the possibility of leveraging deep learning with graph neural networks (GNNs) for generalized planning [45–47], aiming for better scalability. Such neural methods do not require handcrafted or predefined features as they are able to identify relevant information on their own. Specifically, relational GNNs (R-GNNs) have been applied successfully to learn optimal value functions and policies for generalized planning problems. R-GNNs are a special case of aggregate-combine GNNs (AC-GNNs), for which there exist crisp theoretical results on their capabilities. In particular, the logical expressiveness of AC-GNNs has been fully characterized in terms of a fragment of first-order logic [7, 24], which in turn can be used to express general solutions to many generalized planning problems [16, 29, 36, 45]. For many standard planning benchmark domains, the R-GNN-based approaches have been shown to yield promising results.

Unlike symbolic approaches, neural networks are inherently harder to understand and explain. For the R-GNNs used for generalized planning, it remains unclear what they learn about the planning problems, which information about the planning states they leverage, or how they generalize. Understanding AI systems is of crucial importance for many reasons, including trust in the system, and proving that a system is correct.

Recent progress in explainable AI and GNN research has shown promising results on explaining GNNs [34], motivating the application of such explanation methods on R-GNNs used for generalized planning. One method that seems particularly suited to explain the R-GNNs used for generalized planning was proposed recently by Pluska et al. [40]. They introduce a novel transparent and interpretable model called *iterated decision trees (IDTs)* as to capture the

calculations of AC-GNNs in a symbolic way. They call the process of finding matching symbolic formulas *logical distillation*. In their evaluation, they demonstrated that the method effectively explains models of several state-of-the-art AC-GNN architectures for graph classification.

A method that learns formulas from a given learned R-GNN would help to bridge the gap between symbolic and neural approaches for generalized planning, as it eliminates the need for predefined feature pools, but would be able to represent the learned solution transparently, effectively combining the advantages of both approaches. To learn IDTs based on AC-GNNs for generalized planning, several modifications and extensions to the original IDT model structure and learning algorithm are required, because the structures defined by planning states are more complex than the graphs that GNNs and IDT are usually applied to. The core contributions of this thesis are relational IDTs (R-IDTs), which provide a generalization and modification of the original IDTs and can be applied to R-GNNs. R-IDT are designed to leverage both the interpretability and transparency of symbolic methods and the independence of predefined features offered by R-GNN-based methods.

1.1 Thesis Outline

The remainder of this thesis is structured as follows: In Chapter 2, we review the necessary background of the approach. Afterward, Chapter 3 introduces related work, including IDTs. Chapter 4 presents the core contributions of this thesis, that is, R-IDTs. An experimental evaluation of R-IDTs is presented in Chapter 5. Chapter 6 concludes the thesis by providing a summary and directions for future research.

2 Background

This chapter introduces the background of our work. We begin with notational conventions in Section 2.1, followed by an overview of the relevant research fields. First, we define the planning setting that underlies our approach in Section 2.2. Section 2.3 then briefly introduces relational structures, followed by an overview of key deep learning concepts in Section 2.4. The subsequent sections detail GNNs, covering both their general formulation (Section 2.5) and their application to learning value functions for generalized planning (Section 2.6). We then establish the explicit connection between these models and logic in Section 2.7. Finally, we introduce decision trees (Section 2.8) and hierarchical clustering (Section 2.9).

2.1 Notation

First, we fix some notation and terms that are used throughout this work. Let $\mathbb{N}$ denote the set of non-negative integers $\{0, 1, ...\}$. Multisets, denoted by $\{\!\{...\}\!\}$, are analogous to sets, but allowing for multiple occurrences of elements. To simplify notation, a tuple of arbitrary length is written as $\bar{o}$ instead of $\langle o_1, \dots, o_n \rangle$. We denote the *Iverson bracket* by $\mathbb{I}[\cdot]$, which maps true statements to 1, and false statements to 0, e.g., $\mathbb{I}[2 < 3] = 1$, and $\mathbb{I}[1.5 \in \mathbb{N}] = 0$. For matrices or vectors $\mathbf{a}$ and $\mathbf{b}$ with an equal number of rows, $[\mathbf{a} \quad \mathbf{b}]$ denotes their concatenation, that is, $\mathbf{a}$ and $\mathbf{b}$ stacked horizontally. For a graph $\langle V, E \rangle$, the neighborhood of a node v is $N(v) \coloneqq \{w \mid \{v, w\} \in E\}$. True and false are denoted by $\bot$ and $\top$, respectively. For convenience, the set $\{0, 1\}$ of Boolean values is abbreviated by $\mathbb{B}$. For logical expressions φ and interpretations I of any kind, $I \vDash \varphi$ denotes that φ is satisfied by I.

2.2 Planning

Planning, which is the process of constructing a sequence of actions to achieve a goal, is a fundamental component of AI. There are many properties of environments that affect which kind of planning is required or applicable, such as which information is available to the agent, and to which degree the consequences of one's actions are predictable. Accordingly, many environment and agent types have emerged. For more information on planning environment and agent types, we refer to [21, 41]. The kind of planning this work focuses on is classical planning, which we will introduce in the following. Building upon classical planning, we also give an introduction to generalized planning.

2.2.1 Classical Planning

In classical planning, it is assumed that at every point in time, the relevant information about both the agent and its environment is fully available to the agent. Furthermore, the agent's action outcomes are deterministic and known, meaning that applying the same action in the same state will always lead to the same, computable successor state. In the following, the relevant details of classical planning as well as a more formal description will be provided.

A classical planning problem is given as the tuple $\langle \mathcal{D}, \mathcal{I} \rangle$, consisting of the domain $\mathcal{D}$ and the instance $\mathcal{I}$. The domain $\mathcal{D} = \langle P, A \rangle$ consists of a set of predicates and a set of action schemata, while the problem instance $\mathcal{I} = \langle O, \mathit{Init}, \mathit{Goal} \rangle$ consists of a set of objects O, a single initial state Init, and goal conditions Goal which are to be achieved. Predicates and objects are used to describe the state of both the agent and its environment. A predicate p that has k parameters is said to be of arity k, denoted by $\mathrm{ar}(p) = k$. The arguments that can be used are the objects O. Given k objects $o_1, \dots, o_k$, we call $q = p(o_1, \dots, o_k)$ a ground atom. States are sets of ground atoms, and an atom q is said to hold in a state s if and only if $q \in s$. The set Goal consists of ground atoms, and when in a state, all of these atoms hold, that state is a goal state. Each action schema specifies the atoms that need to hold in order to be able to execute the action, as well as how the action changes the state by adding and removing atoms. As a common standard, the Planning Domain Definition Language (PDDL) [22, 27] is used to encode such planning problems. The presented model induces a state model $\langle S, s_0, S_\mathrm{G}, \mathit{Act}, A, f \rangle$, where

- S is the state space, that is, the set of all sets of ground atoms,
- $s_0 = \mathit{Init}$ is the initial state,
- $S_\mathrm{G} = \{s \mid s \in S, \mathit{Goal} \subseteq s\}$ is the set of the goal states,
- Act is the set of all ground actions,
- $A(s) \in \mathit{Act}$ is the set of applicable actions for each state s, and
- $f(s, a)$ is the successor state obtained by applying a in s for every state s and action $a \in A(s)$.

Additionally, one can specify a cost function $c : S \times \mathit{Act} \to \mathbb{R}^{>0}$ that specifies the cost of executing a in s. Here, uniform costs are assumed, that is, for all $s \in S$ and $a \in \mathit{Act}$, $c(s, a) = 1$.

As an example, let us introduce the domain *Blocksworld*. In this domain, the objects are a number of blocks, which can be rearranged by a robot arm. The domain's predicates are

- $\texttt{on}(x, y)$, which means that block x is placed on top of block y,
- $\texttt{ontable}(x)$, which denotes that x is on the table,
- $\texttt{clear}(x)$, which says that there is no block above x and that it is not being held,
- `handempty`, which is true if no block is held, and
- $\texttt{holding}(x)$, which is true if x is the block being held.

For a full PDDL definition of the domain, see Listing A.1. In Listing 2.1, the PDDL definition of the action schema `unstack` of the Blocksworld domain is shown. It operates on two blocks x and y. For any two blocks, the ground action `unstack`(x, y) can only be applied if the preconditions are satisfied, that is, x is on y, x is clear, and the hand is empty. The effects of the action are that x will be held, y will be clear, x will be clear no longer, the hand will be empty no longer, and x will no longer be on y.

A Blocksworld problem instance defined in PDDL is shown in Listing 2.2. It defines the objects, the initially true atoms, and the goal. The initial state as well as two of its successor states are visualized in Figure 2.1.

Listing 2.1: Definition of the action `unstack` of the Blocksworld domain in PDDL.

```
(:action unstack
    :parameters (?x ?y)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect (and
        (holding ?x)
        (clear ?y)
        (not (clear ?x))
        (not (handempty))
        (not (on ?x ?y))
    )
)
```

Listing 2.2: Example of a Blocksworld problem instance definition in PDDL.

```
(define (problem blocks-4)
    (:domain blocks)
    (:objects a b c d)
    (:init
        (clear a) (on a b) (clear c) (clear d) (ontable b)
        (ontable c) (ontable d) (handempty)
    )
    (:goal (and (clear b)))
)
```

In classical planning, the task is to obtain a plan $\langle a_1, \dots, a_n \rangle$ of actions such that applying it sequentially leads from the initial state to a goal state, that is,

$$s_0 \xrightarrow{f(\cdot, a_1)} s_1 \xrightarrow{f(\cdot, a_2)} \dots \xrightarrow{f(\cdot, a_n)} s_n, \text{ with } s_n \in S_G.$$

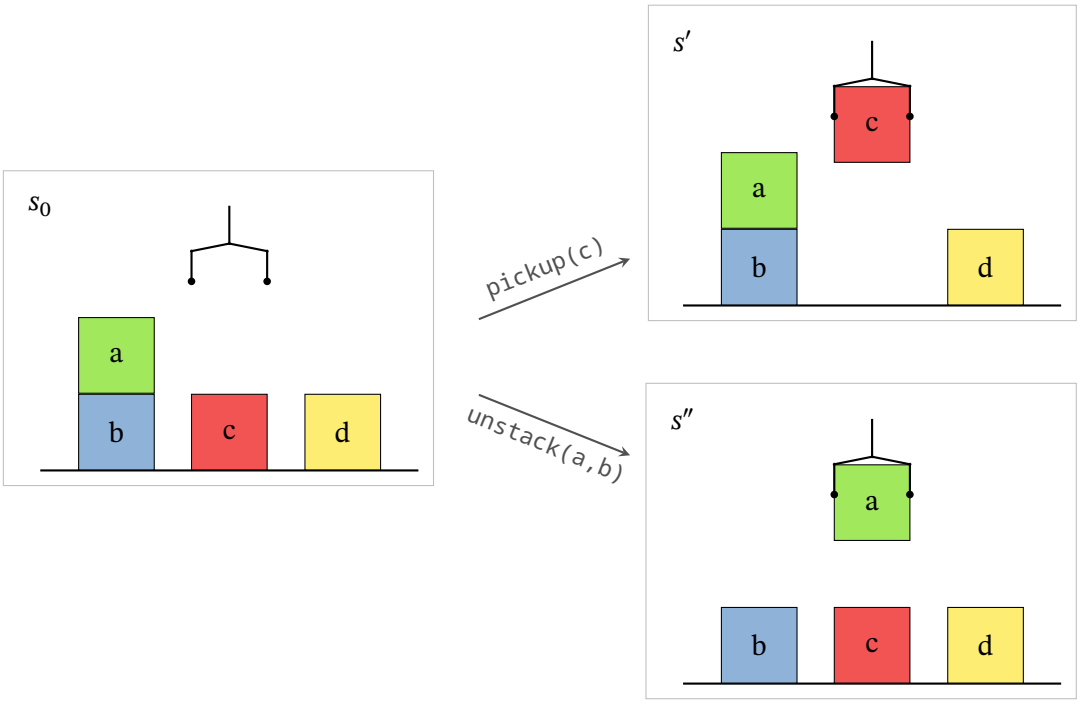


Figure 2.1: Visualization of three states of the problem instance shown in Listing 2.2. The left panel shows the initial state s_0. The top-right and bottom-right states are achieved by applying the actions `pickup(c)` and `unstack(a, b)` respectively: $s' := f(s_0, \texttt{pickup(c)})$, and $s'' := f(s_0, \texttt{unstack(a, b)})$.

When assuming uniform costs as we do, the cost of a plan equals its length. A plan is optimal if there is no plan with a lower cost. For a single problem instance, an (optimal) plan can be computed using a *planner*.

2.2.2 Generalized Planning

Following Bonet and Geffner [12], a generalized planning problem is defined as a collection of classical planning problems that share a common domain, but may differ in their objects, goals, and initial states. One such example is the problem class Q_{clear}, which denotes the problems in the Blocksworld domain where the goal is to clear a single specified block, independent of the number of blocks and their arrangement, that is, the initial state. To solve a generalized planning problem, the computation of a single plan is no longer sufficient. Rather, the computation of a general solution is required. Several forms of such general solutions and approaches to learn them have been explored, including synthesizing programs, controllers, and policies [2, 12, 14, 43, 44]. In our work, we are concerned with policies for generalized planning problems, called general policies.

General Policies and Value Functions

Policies are functions which take a state, and return an action to apply. When following a policy, that is, sequentially using the policy to get an action and applying the action, like

$$s_0 \xrightarrow{f(\cdot,\pi(s_0))} s_1 \xrightarrow{f(\cdot,\pi(s_1))} \dots \xrightarrow{f(\cdot,\pi(s_{n-1}))} s_n,$$

a path is generated. A policy is said to solve a problem (optimally) if the resulting path is a path (an optimal path) to a goal state. One can define the value function $V^\pi : S \to \mathbb{R}$ of a policy, which equals the number of actions required when following π until a goal state is reached. When given an arbitrary value function V, one can infer a policy that is greedy in V [48] by choosing an action that leads to a state with the lowest value among all successors, that is, $\pi(s) = a \in A(s)$ such that $V(f(s,a)) = \min_{a' \in A(s)} V(f(s,a'))$. The optimal value function $V^*(s)$ equals the number of actions required when following an optimal plan from s. Thus, when V^* is known, an optimal policy π^* can be obtained by constructing it greedily as described above.

As an example, consider the optimal values of the states depicted in Figure 2.1 where the goal is to clear block b. The optimal value of s'' is 0, because it is a goal state, that is, block b is clear. It follows that the optimal value of s_0 must be 1, because s'' can be reached with a single step, namely `pickup`(c). From s', at least two steps are required to reach a goal state: The hand has to be freed by dropping block c anywhere except on block a, and block a has to be unstacked from block b. Therefore, $V^*(s') = 2$.

Features as Abstractions

In the following, we introduce how one can define and use abstractions of planning states, known as *features*, that in turn can be used to define general policies. Features are abstractions in the sense that they provide information about a state, regardless of irrelevant details. Returning to the example of Figure 2.1, the number of blocks above the block that should be cleared is a feature. It abstracts away both the actual object names and the ordering of the blocks while still providing relevant information.

As explored by [11, 12, 19, 36], one can define unary properties known as *concepts* over states, that can express increasingly complex statements about the objects of the problem instance in a state. As an example for a concept in the Blocksworld domain, consider the formula $\varphi(x) \coloneqq \texttt{ontable}(x) \wedge \exists y\,(\texttt{on}(y,x) \wedge \texttt{clear}(y))$. This concept is true for an object x if there is a tower of two blocks such that x is on the table, and one block y that is stacked onto x and has no blocks stacked above it. Let ψ be such a concept and s be some state. We then distinguish two features based on ψ: The Boolean feature p_ψ, that evaluates to 0, if no objects satisfy ψ in s, and else to 1. Similarly, let n_ψ denote its numerical feature, that evaluates to the number of objects that satisfy ψ in s. The Boolean feature of a concept is therefore a more abstract

representation of the same information as the numerical feature, as

$$n_\psi = 0 \text{ iff } p_\psi = 0, \text{ but} \tag{2.1}$$

$$n_\psi \geq 1 \text{ iff } p_\psi = 1. \tag{2.2}$$

The necessary concepts underlying these features can be expressed by description logic formulas [36]. Their expressive power correspond to the fragment of first-order predicate logic called C_2, which are discussed in Section 2.7.

To define concepts that refer to the goal, one needs to be able to access the goal condition of the problem instance, as it is not assumed that the goal is shared between all instances. One way to provide information about the goal is to encode it in the states using special atoms. This can be done by adding the *goal predicate* p_G for each predicate p. In every state of a planning problem, $p_\mathrm{G}(\bar{o})$ is set to hold if and only if $p(\bar{o})$ is an atom that must be achieved as a goal [36]. For example, assuming that the goal of an instance is the single atom $\texttt{clear}(\mathrm{a})$, then *every* state of that instance gets the extra atom $\texttt{clear}_\mathrm{G}(\mathrm{a})$.

For many common benchmark domains, optimal value functions can be expressed based on features. More specifically, linear value functions of the form $V(s) = \sum_{f \in \mathcal{F}} w_f f(s)$ for a suitable set of features $\mathcal{F}$ and their respective weights $\langle w_f \rangle_{f \in \mathcal{F}}$ can often express the optimal value function [20, 45]. As an example, recall the problem class $\mathcal{Q}_\mathrm{clear}$. The goal is to clear some block x, which is encoded by the atom $\texttt{clear}_\mathrm{G}(x)$.

For any number of blocks $n \geq 1$, the optimal value function for this generalized planning problem can be written as

$$V^*(s) = (\alpha \wedge H)(s) + \sum_{k=1}^{n-1} (2k-1) B_k(s), \tag{2.3}$$

using the Boolean features

$$\alpha = \exists x \, (\texttt{clear}_\mathrm{G}(x) \wedge \neg \texttt{clear}(x)), \tag{2.4}$$

$$H = \exists x \, \texttt{holding}(x), \tag{2.5}$$

$$B_k = \exists x \, (\texttt{clear}_\mathrm{G}(x) \wedge \eta_k(x)), \tag{2.6}$$

and helper formulas

$$\eta_0(z) = \texttt{clear}(z), \tag{2.7}$$

$$\eta_k(z) = \exists y \, (\mathsf{on}(y, z) \wedge \eta_{k-1}(y)). \tag{2.8}$$

For any state s, the feature α holds if and only if s is not a goal state, H if and only if the gripper is occupied, and B_k if and only if there are exactly k blocks above the block to be cleared. To

express this, η is introduced. For any $k \in \mathbb{N}$ and block z, $\eta_k(z)$ holds if and only if there are exactly k blocks above z.

As explained by Ståhlberg et al. [46], if a value function can be expressed in such a linear form, then it can also be expressed as a function over objects directly, which can be written as $v = F(\phi(o_1), \dots, \phi(o_n))$. For this, a suitable function ϕ is needed, which maps objects to Boolean vectors, that is, $\phi : O \to \mathbb{B}^k$. This way, for each object, k Boolean features are computed. The other function, F, takes all the objects' feature vectors, and can then compute the values of state features from the values of the object features. Returning to the Blocksworld example, assume that for all objects o and some index i, $\phi(o)_i = 1$ if and only if object o is being held. Then, the corresponding Boolean state feature can be extracted by $p_\varphi = \max_{o \in O} \phi(o)_i$. Analogously, the numerical state feature can be extracted by $n_\varphi = \sum_{o \in O} \phi(o)_i$. This construction guarantees the existence of a suitable function F that can compute state values from features of objects.

2.3 Relational Structures

A *relational structure* $\langle D, R_1, \dots, R_n \rangle$ consists of a *domain of discourse* D (not to be confused with the domain of a planning problem) and a number of relations over D. A simple case of relational structures is the graph, where the domain of discourse is the set of vertices V, and the only relation is the edge relation $E \subseteq V^2$.

States of planning problems can also be seen as relational structures. Assume a planning problem with the set of objects O, and the set of predicates $P = \{p_1, \dots, p_n\}$. The set of objects is the domain of discourse, and for each predicate (including the added "goal predicates" p_G), there is a corresponding relation. The relation P of a predicate p is defined as $P \coloneqq \{\bar{o} \mid p(\bar{o}) \in s\}$, that is, the set of tuples of objects for which the predicate holds in s. As an example, consider the binary predicate on of the Blocksworld domain. In a state s, the induced binary relation $\mathrm{On} \subseteq O^2$ is then the set of all pairs of objects $\langle o_1, o_2 \rangle$ with $\mathsf{on}(o_1, o_2) \in s$. In conclusion, the corresponding relational structure of a state is $\langle O, P_1, \dots, P_n, P_1^\mathrm{G}, \dots, P_n^\mathrm{G} \rangle$. For convenience, we may refer to a state's relational structure as the state's graph, although strictly speaking, it is not a graph.

2.4 Deep Learning

In this section, we review the necessary background regarding deep learning. It is heavily based on the textbook by Goodfellow et al. [23], to which we refer for more information. Deep learning is part of the field of machine learning, and is particularly concerned with neural networks. Neural networks are parametrized functions which are used as learnable function approximations. It has been shown that, in a sense, they can universally approximate a wide class of functions [30].

2.4.1 Multilayer Perceptrons

Multilayer perceptrons (MLPs) are an essential type of neural network. They are composed of a number of functions which are applied sequentially, called layers. MLPs process data layer by layer, where each layer uses the information provided by the previous layer. When an MLP processes data, the first layer, known as the input layer, is applied to the input data. Next, the so-called hidden layers are applied. Finally, the network's output is computed by the output layer. Thus, an MLP is the result of chaining functions together. The function computed by an MLP can be written as

$$\mathsf{MLP}(\mathbf{x}) = f^{(n)}\big(\dots f^{(2)}\big(f^{(1)}(\mathbf{x})\big)\dots\big),$$

where $f^{(i)}$ is the i-th layer. To show this MLP's architecture in a compact way, we write

$$\mathbb{R}^{k_0} \xrightarrow{f^{(1)}} \mathbb{R}^{k_1} \xrightarrow{f^{(2)}} \mathbb{R}^{k_2} \dots \xrightarrow{f^{(n)}} \mathbb{R}^{k_n},$$

where k_0 is the size of the input, and k_i is the size of the vector that is computed by $f^{(i)}$.

Typically, the layers of an MLPs are constructed using linear transformations, followed by a non-linear activation function. A linear transformation uses a weight matrix $\mathbf{W} \in \mathbb{R}^{a\times b}$ and a bias vector $\mathbf{b} \in \mathbb{R}^a$ as learnable parameters. Its output is computed as

$$\mathrm{Linear}(\mathbf{x}) = \mathbf{W}\cdot\mathbf{x} + \mathbf{b}.$$

The size of the output of a layer is defined by the choice of a.

There are many valid choices for activation functions [23]. Below, we present some that are relevant for our work. There are both activation functions that are applied elementwise to each scalar value of a vector, and functions that take all the values of $\mathbf{x}$ into consideration for computing the result (such as softmax). When we write functions with respect to a scalar input like ReLU below, it is implied that this function is applied elementwise.

Examples for activation functions are the rectified linear unit (ReLU), defined as

$$\mathrm{ReLU}(x) \coloneqq \max\{0, x\},$$

and Mish [38], which is defined as

$$\mathrm{Mish}(x) \coloneqq x\tanh(\ln(1+\exp(x))).$$

While ReLU has been a popular choice for some time, Mish has gained increased interest recently.

For a visualization of the ReLU and Mish function, see Figure 2.2.

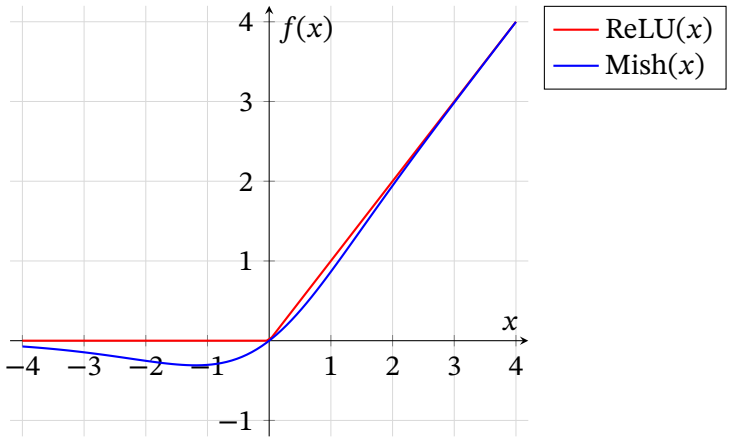


Figure 2.2: Plots of the ReLU and Mish activation functions.

There are many more common and historically influential activation functions like tanh and the sigmoid function, which we do not detail here. For more information, we refer to [10, 23].

An MLP's inner layer is then constructed by chaining an activation function h and a linear transformation, that is,

$$f(\mathbf{x}) = h(\mathbf{W} \cdot \mathbf{x} + \mathbf{b}).$$

When an MLP is used for regression, it is common that its last layer is a linear transformation without an activation function.

2.4.2 Training Neural Networks

To train a neural network, one defines a measure of loss L. The loss measures how bad the network's prediction of a single input is. For supervised learning, the loss is commonly based on comparing the prediction and the known true value. Training a network corresponds to minimizing the loss over the whole dataset by adjusting the learnable parameters θ. For a set of training data D, the empirical risk is defined as

$$J(\theta) \coloneqq \mathop{\mathbb{E}}_{\langle \mathbf{x}, y \rangle \sim D} L(f_\theta(\mathbf{x}), y) = \frac{1}{|D|} \sum_{\langle \mathbf{x}, y \rangle \in D} L(f_\theta(\mathbf{x}), y),$$

where f_θ denotes the function computed by the neural network using parameters θ. In practice, this loss is often approximated by sampling many minibatches, which are smaller subsets of the training data, and calculating the risk only with respect to one minibatch at a time.

For regression tasks, common loss functions include the absolute and squared error (also known as L1 and L2 loss), which are defined as

$$L_{\mathrm{AE}}(a, b) \coloneqq |a - b|, \text{ and} \tag{2.9}$$

$$L_{\mathrm{SE}}(a, b) \coloneqq (a - b)^2. \tag{2.10}$$

Based on these, the sum of absolute errors and sum of squared errors are defined as

$$\mathrm{SAE}(\theta) \coloneqq \sum_{\langle \mathbf{x}, y \rangle \in D} L_{\mathrm{AE}}(f_\theta(\mathbf{x}), y), \text{ and} \tag{2.11}$$

$$\mathrm{SSE}(\theta) \coloneqq \sum_{\langle \mathbf{x}, y \rangle \in D} L_{\mathrm{SE}}(f_\theta(\mathbf{x}), y). \tag{2.12}$$

Minimizing the sum of squared errors is especially suited under the assumption that the observed target y has additive normally distributed noise [10].

Most often, the empirical risk is minimized by using the gradient $\nabla_\theta J(\theta)$, which can intuitively be seen as a measure of the influence of each parameter on the risk. This gradient can be computed using the Backpropagation algorithm. As we do not go into details here, we again refer to [23] for more information.

Once the gradient has been computed, the parameters are updated, typically using some variant of gradient descent. In its most basic form, gradient descent iteratively computes the gradient $\nabla_\theta J(\theta)$ for the current parameter values θ and updates them as

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta),$$

where the hyperparameter η is called the learning rate. Again, it is common to use minibatches to estimate the gradient of the empirical risk. In this case, the update schema is called (minibatch) stochastic gradient descent (SGD) [23].

In practice, more involved optimizers may be used, which make use of the momentum of the gradient, second-order derivatives, adaptive learning rates, or other techniques. A popular example of an optimizer using momentum terms and adaptive learning rates is Adam [33].

2.5 GNNs for Graph Classification and Regression

GNNs [42] are neural networks that take graphs as inputs. They have become an important tool to analyze data with complex relations between objects, as well as data which is naturally represented as graphs, such as transportation systems, molecules, and many more [52]. Compared to MLPs, an important advantage of GNNs is that they are structurally designed to work on data of variable size, that is, graphs of a variable number of nodes and edges.

While there are also GNNs for tasks such as node prediction, edge classification, and other tasks, we focus on GNNs for graph classification and regression. Even for these use cases, there exist different types of GNNs, however, but the focus of this work is on the so-called message-passing or AC-GNNs.

An AC-GNN is specified by

- the number of layers $L \in \mathbb{N}$,
- the embedding size $k \in \mathbb{N}$,
- each layer's aggregation function $\langle \mathsf{agg}_i \rangle_{i=0}^{L-1}$,
- each layer's combination function $\langle \mathsf{comb}_i \rangle_{i=0}^{L-1}$, and
- the pooling function pool.

The number of layers L influences how far the messages generated by the GNN can travel between graph nodes. Specifically, the number of edges a message can pass is at most L. To ensure that a model is in principle able to learn features that require to access nodes that are a certain distance away, setting a proper value for L is of great importance. It also has a direct impact on the computational effort required by the GNN, as will become evident later. Similar to any neural network, the embedding size k influences how many different features about the data can be learned by the GNN. Thus, it also influences the number of model parameters, the model's size, and its computational requirements.

To illustrate how a GNN operates, let $G = \langle V, E \rangle$ denote an undirected graph. For each node $v \in V$, a GNN proceeds to initialize the node embeddings $\mathbf{x}_v^{(0)} \in \mathbb{R}^k$, which may include node features in some form. In each layer $i \in \{0, \dots, L-1\}$, every node simultaneously sends its embedding to each of its neighbors, which in turn will use them to update their own embedding. In layer i, each node v has the embedding $\mathbf{x}_v^{(i)}$. It aggregates all incoming messages into a single embedding, which is then combined with the node's old embedding into its new embedding, that is,

$$\mathbf{x}_v^{(i+1)} \coloneqq \mathsf{comb}_i\Big(\mathbf{x}_v^{(i)}, \mathsf{agg}_i\Big(\Big\{\!\!\Big\{ \mathbf{x}_w^{(i)} \;\Big|\; w \in N(v) \Big\}\!\!\Big\}\Big)\Big).$$

Finally, after L iterations, the resulting embeddings are pooled to yield a single output for the whole graph, that is,

$$\hat{y} \coloneqq \mathsf{pool}\Big(\Big\{\!\!\Big\{ \mathbf{x}_v^{(L)} \;\Big|\; v \in V \Big\}\!\!\Big\}\Big),$$

which can be used to achieve either graph level classification or regression.

Usually, the functions agg_i, comb_i, and pool make use of trainable neural networks. When every layer uses the same aggregation and combination functions, the GNN is called homogeneous. In this case, we simply write agg and comb instead of agg_i and comb_i, respectively.

2.6 GNNs for General Value Functions

Previously, GNNs for undirected graphs with a single edge relation E were introduced. GNNs can be modified to work on relational structures, as they are a generalization of graphs (see Section 2.3). Specifically, Ståhlberg et al. [45] introduced the R-GNN architecture, which has since been used to learn generalized value functions and policies for generalized planning problems using supervised and unsupervised learning as well as actor-critic reinforcement learning [3, 45–47]. This section introduces R-GNNs for learning the optimal value functions

Algorithm 1 R-GNN for optimal value functions over planning states.

procedure R-GNN(s, O)
  for all objects $o \in O$ **do**
    $\mathbf{x}_o^{(0)} \leftarrow \mathbf{0}^k$ *Initialize node embeddings*
  for $i = 0, \dots, L-1$ **do**
    for all atoms $q = p(o_1, \dots, o_n) \in s$ **do**
      $\mathbf{m}_q \leftarrow \mathsf{MLP}_p\left(\mathbf{x}_{o_1}^{(i)}, \dots, \mathbf{x}_{o_n}^{(i)}\right)$ *Atoms receive the embeddings of contained objects*
      for all $j = 1, \dots, n$ **do**
        $\mathbf{m}_{q,j} \leftarrow \mathbf{x}_{o_j}^{(i)} + \left(\mathbf{m}_q\right)_j$ *Atoms generate messages to be sent back to contained objects*
    for all objects $o \in O$ **do**
      $\mathbf{m}_o \leftarrow \left\{\!\!\left\{\mathbf{m}_{q,j} \mid q = p(\dots, o_j = o, \dots) \in s\right\}\!\!\right\}$ *Objects gather all incoming messages*
      $\mathbf{x}_o^{(i+1)} \leftarrow \mathbf{x}_o^{(i)} + \mathsf{MLP}_{\mathrm{U}}\left(\mathbf{x}_o^{(i)}, \mathsf{agg}(\mathbf{m}_o)\right)$ *Objects aggregate messages and combine with previous embeddings*
  $v \leftarrow \mathsf{MLP}_{\mathrm{R}}\left(\bigoplus_{o \in O} \mathbf{x}_o^{(L)}\right)$ *Apply pooling over all embeddings*
  return v

of generalized planning problems. Although the explained variant is based on the original formulation [45, 46], minor modifications are included. Algorithm 1 shows the neural network structure in an algorithmic form.

Before the first layer is applied, each object embedding is initialized with a vector of zeros. While embeddings can also be initialized randomly as in [45, 46], this option is neglected here. For each predicate p of the planning domain, there is an MLP $\mathsf{MLP}_p : \mathbb{R}^{k \cdot \mathrm{ar}(p)} \to \mathbb{R}^{k \cdot \mathrm{ar}(p)}$. For each parameter, it takes an object embedding as input and outputs a message, that is, it takes $\mathrm{ar}(p)$ object embeddings and outputs $\mathrm{ar}(p)$ messages of size k. After the initialization, each of the L iterations occurs as follows: Each object o receives one such message $\mathbf{m}_{q,j}$ by each atom $q = p(\dots, o_j = o, \dots)$ where it is used as an argument in the j-th position. The objects individually aggregate these messages by using the agg function.

For example, consider a Blocksworld state with the ground atom $q = \mathsf{on}(\mathrm{a}, \mathrm{b})$. The blocks a and b would send their embeddings $\mathbf{x}_\mathrm{a}$ and $\mathbf{x}_\mathrm{b}$ to the atom, which would then compute the messages

$$\left\langle \mathbf{m}_{\mathsf{on}(\mathrm{a},\mathrm{b}),1}, \mathbf{m}_{\mathsf{on}(\mathrm{a},\mathrm{b}),2} \right\rangle = \mathsf{MLP}_{\mathsf{on}}(\mathbf{x}_\mathrm{a}, \mathbf{x}_\mathrm{b})$$

for a and b, respectively. Both a and b would then aggregate this and all other messages they receive, that is, all messages $\mathbf{m}_{p(o_1,\dots,o_n),j}$ where $o_j = \mathrm{a}$ and $o_j = \mathrm{b}$, respectively.

The MLP for each predicate is of the form

$$\mathbb{R}^{k \cdot \mathrm{ar}(p)} \xrightarrow{\text{Linear}} \mathbb{R}^{k \cdot \mathrm{ar}(p)} \xrightarrow{\text{Mish}} \mathbb{R}^{k \cdot \mathrm{ar}(p)} \xrightarrow{\text{Linear}} \mathbb{R}^{k \cdot \mathrm{ar}(p)}.$$

Common choices for the aggregation function include summing up all messages, taking the maximum across all messages as

$$\max(\{\!\{\mathbf{x}_1, \dots, \mathbf{x}_n\}\!\}) \coloneqq \begin{pmatrix} \max_i x_{i,1} \\ \dots \\ \max_i x_{i,k} \end{pmatrix}, \tag{2.13}$$

or computing a smooth maximum with the logsumexp function

$$\mathrm{LSE}(\{\!\{\mathbf{x}_1, \dots, \mathbf{x}_n\}\!\}) \coloneqq \begin{pmatrix} \log(\exp(x_{1,1}) + \dots + \exp(x_{n,1})) \\ \dots \\ \log(\exp(x_{1,k}) + \dots + \exp(x_{n,k})) \end{pmatrix}. \tag{2.14}$$

For updating the object embeddings, a single shared MLP $\mathsf{MLP}_{\mathrm{U}}$ is employed in every layer. It takes both the previous object embedding and the aggregated messages as inputs and is of the form

$$\mathbb{R}^{2k} \xrightarrow{\text{Linear}} \mathbb{R}^{2k} \xrightarrow{\text{Mish}} \mathbb{R}^{2k} \xrightarrow{\text{Linear}} \mathbb{R}^{k}.$$

The object's new embedding is then computed by a residual update, that is, by summation of the old embedding and the output of $\mathsf{MLP}_{\mathrm{U}}$.

Finally, after L such iterations, a final graph-level aggregation $\bigoplus$ is applied to the object embeddings. As options for $\bigoplus$, we consider taking the component-wise maximum (as in Equation (2.13)) and summation. The R-GNN's single scalar output value v is computed by applying another MLP $\mathsf{MLP}_{\mathrm{R}}$ to the graph-level aggregation. The MLP is of the form

$$\mathbb{R}^{k} \xrightarrow{\text{Linear}} \mathbb{R}^{2k} \xrightarrow{\text{ReLU}} \mathbb{R}^{2k} \xrightarrow{\text{Linear}} \mathbb{R}^{1}.$$

Note that in the original formulation by Ståhlberg et al. [45, 46], the output is computed using two MLPs: $v \leftarrow \mathsf{MLP}_2\Big(\sum_{o \in O} \mathsf{MLP}_1\Big(\mathbf{x}_o^{(L)}\Big)\Big)$. When training the R-GNN with SGD, the learnable parameters are the parameters of the predicate MLPs, $\mathsf{MLP}_{\mathrm{U}}$, and $\mathsf{MLP}_{\mathrm{R}}$.

2.7 Graphs, GNNs, and Logic

The first-order logic fragment C_2 is closely connected to both the expressiveness of GNNs and description logics [6], which are able to express state concepts for planning. In the following, we give an introduction and highlight its importance for this work.

According to Grohe [24] and Lutz et al. [35], the formulas of C_2 are defined as the first-order formulas that

- do not use functions or constants symbols,
- only use the variables x and y, and

- only use relation symbols with an arity of one or two.

In addition to the usual first-order quantifiers $\exists$ and $\forall$, the counting quantifiers $\exists^{\geq n}$, $\exists^{\leq n}$, and $\exists^{=n}$ for $n \in \mathbb{N}$ can be used. Their interpretation is straightforward: Any C_2 formula $\exists^{\geq n} x\, \varphi(x)$ holds iff in the domain of discourse, there are at least n unique elements x that satisfy $\varphi(x)$. The other counting quantifiers are interpreted similarly.

As shown by Martín and Geffner [36], for many planning domains, general policies can be expressed through Boolean and numerical features induced by state concepts. These concepts can be expressed using description logics, which in turn are restricted by C_2 in their expressive power [6].

The logic fragment GC_2 (called guarded C_2) is obtained by restricting C_2 even further: Every formula must be guarded by a binary relation R, enforcing locality in a sense: For a formula $\varphi(x)$, that has the free variable x, every subformula using quantifiers over the other variable y is required to include $R(x, y)$ without negation as part of a conjunction. For example, assume some unary predicate U, then both $\psi_1(x) \coloneqq \exists^{\geq 8} y\, U(y)$ and $\psi_2(x) \coloneqq \exists^{\geq 8} y\, (\neg R(x, y) \wedge U(y))$ are not GC_2 formulas, but $\varphi(x) \coloneqq \exists^{\geq 8} y\, (R(x, y) \wedge U(y))$ is.

When considering graphs and their nodes, the relation symbols of arity one are the node features, and the only binary relation is the edge relation E. It follows that GC_2 formulas over graph nodes must be guarded by the edge relation, which is similar to each node only having information about itself and its neighbors.

Formulas with one free variable can be seen as node classifiers, in the sense that they are only satisfied by nodes of a certain class. Consider $\varphi(x)$ from before, where the relation R is the edge relation E: For any graph G and node v, $\langle G, v\rangle \vDash \varphi$ if and only if in G, v has at least eight neighbors which satisfy U. Further, formulas with no free variable can be seen as graph classifiers. For example, consider $\varphi \coloneqq \exists^{\leq 1} x\, \neg \exists y\, E(x, y)$, which expresses that there is at most one disconnected node in a graph.

As shown in [7, 24], the expressive power of AC-GNNs for node classification corresponds exactly to that of GC_2: Every GC_2 formula over graphs is equivalent to a function expressible by an AC-GNN (without a pool function), and vice versa. These connections are what motivated the application of GNNs to learn general optimal value functions in the first place (see [45]).

2.8 Decision Trees

Decision trees are prediction models that can be leveraged for classification and regression tasks. Here, we focus on decision tree regressors and numerical data features, as they are what we need for our work. We see Boolean data features as a special case of numerical data features, which can only take the values 0 and 1. The data decision trees work on can be divided into two

distinct parts: The input features based on which the prediction happens, and the dependent variable, which gets predicted based on the input features.

Decision trees are learned in a supervised way, that is, the training data is given as a collection of labeled samples. Each sample is a tuple $\langle \mathbf{x}, \mathbf{y} \rangle$, where the vector $\mathbf{x} \in \mathbb{R}^k$ contains the values of the input features, and $\mathbf{y} \in \mathbb{R}^l$ is the value of the dependent variable. In the following, we denote the training data by $D = \{\langle \mathbf{x}_i, \mathbf{y}_i \rangle\}_{i=1}^n$. In practice, it is common to give the training data in form of two matrices $\mathbf{X} \in \mathbb{R}^{n \times k}$ and $\mathbf{Y} \in \mathbb{R}^{n \times l}$, where each row contains all values regarding a data sample.

Training a decision tree works as follows: First a single root node is created, which contains all the training data samples D. Iteratively, at each node m, the algorithm computes the node's prediction value $\hat{\mathbf{y}}_m$, which is usually set to the mean

$$\hat{\mathbf{y}}_m = \frac{1}{|D_m|} \sum_{\langle \mathbf{x}, \mathbf{y} \rangle \in D_m} \mathbf{y}, \tag{2.15}$$

where $D_m \subseteq D$ are the data samples held by m. Then, the algorithm selects an input feature $f \in \{1, \dots, k\}$ and a threshold $t \in \mathbb{R}$, and splits the data into two disjunct parts $P_m^{\leq}(f, t)$ and $P_m^{>}(f, t)$. One part of the data only has values for the chosen feature that are less than or equal to the threshold, while the other part only has values that are greater than the threshold, that is,

$$P_m^{\leq}(f, t) \coloneqq \left\{ \langle \mathbf{x}, \mathbf{y} \rangle \mid \langle \mathbf{x}, \mathbf{y} \rangle \in D_m, x_f \leq t \right\}, \text{ and}$$
$$P_m^{>}(f, t) \coloneqq \left\{ \langle \mathbf{x}, \mathbf{y} \rangle \mid \langle \mathbf{x}, \mathbf{y} \rangle \in D_m, x_f > t \right\}.$$

These data parts are then used to create two new nodes m' and m'', each of which gets one of the partitions, that is, $D_{m'} \coloneqq P_m^{\leq}(f, t)$ and $D_{m''} \coloneqq P_m^{>}(f, t)$. A decision tree iteratively splits the data in this way. The *depth* of a node is the number of splitting decisions on its path to the root node. The depth of a whole decision tree is defined as the maximal depth of its leaf nodes. See Figure 2.3 for an illustration of a decision tree of depth two.

Splits are chosen such that they minimize some measure of the heterogeneity (or equivalently, maximize the homogeneity) of the dependent variable within each of the obtained partitions. For a specified heterogeneity measure h, the quality of a potential split $\langle f, t \rangle$ can be calculated by the weighted heterogeneity measure of the resulting partitions as

$$q_m(f, t) \coloneqq \frac{\left|P_m^{\leq}(f, t)\right|}{|D_m|} h\left(P_m^{\leq}(f, t)\right) + \frac{\left|P_m^{>}(f, t)\right|}{|D_m|} h\left(P_m^{>}(f, t)\right). \tag{2.16}$$

The best split is then chosen as $\langle f^*, t^* \rangle = \arg\min_{\langle f, t \rangle} q_m(f, t)$.

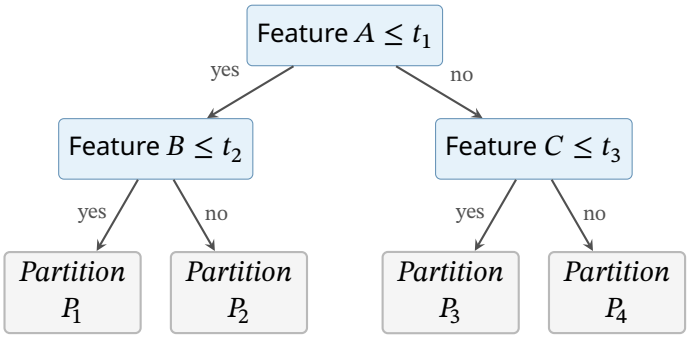


Figure 2.3: Illustration of a decision tree of depth two. It uses the features A, B, and C to split the data into four partitions.

There are many stopping criteria for determining when to stop splitting, for example, when a specified maximum depth is reached, when there are too few samples in a node to continue splitting, or when the heterogeneity does not improve by a set threshold.

When the model is used for regression, as in our case, a possible heterogeneity metric is the mean squared error, which is equivalent to the resulting in-partition variance. For a partition P, it is defined as

$$\mathrm{MSE}(P) := \frac{1}{|P|}\|\mathbf{y} - \hat{\mathbf{y}}_m\|^2. \tag{2.17}$$

In the case that some target values are strongly overrepresented in the training dataset, it can be useful to define a weight w for each data sample. When training decision trees for classification or regression on a discrete set of target values, a common choice is to weight each sample by its class's or target value's reciprocal frequency, For this, let p_v be the fraction of occurrences of the value v, that is,

$$p_v := \frac{|\{\langle \mathbf{x}, \mathbf{y}\rangle \mid \langle \mathbf{x}, \mathbf{y}\rangle \in D, \mathbf{y} = v\}|}{n}, \tag{2.18}$$

and define the weight of each sample $\langle \mathbf{x}, \mathbf{y}\rangle$ depending on its target value as

$$w_{\langle \mathbf{x},\mathbf{y}\rangle} := \frac{1}{p_{\mathbf{y}}}. \tag{2.19}$$

This weighting scheme ensures that the model does not become biased toward majority classes in imbalanced datasets.

Each leaf node corresponds to the conjunction of the splitting decisions on its path to the root of the tree. For example, partition 1 of Figure 2.3 corresponds to $A \leq t_1 \wedge B \leq t_2$. Further, a set of leaf nodes therefore corresponds to the disjunction of the conjunctions of the paths leading to each contained leaf node, for example $\{P_1, P_3\}$ corresponds to $A \leq t_1 \wedge B \leq t_2 \vee A > t_1 \wedge C \leq t_3$. A set of leaf nodes is called a *leaf set*.

When applying the trained model on data, the learned splitting decisions are sequentially executed on the input, and the predicted value of a sample is $\hat{\mathbf{y}}_m$, where m is the leaf node

it ends up in. In contrast to neural networks, decision trees are white box models, as their workings are transparent and can directly be extracted in the form of logical expressions. Decision trees are a much broader field of research than presented here. For more information, we refer to [28, 37].

2.9 Hierarchical Agglomerative Clustering

Hierarchical agglomerative clustering [1, 28] is used to create clusters of data based on a numerical feature. This kind of clustering works bottom-up, that is, at the beginning, each data point is its own cluster. Then, iteratively, two of clusters are merged together to create a larger cluster. Which clusters should be merged next is decided by computing the similarities between all pairs of current clusters, and choosing the pair which exhibits the highest similarity measure. We understand distances as the opposite of similarities. For more information on distances and similarities, we refer to [15]. The clustering process can be stopped after a specified condition, e.g., when the algorithm has computed a certain number of clusters. How the distances are computed is subject to configuration: Usually, one defines how the distances between any two data points is computed, as well as a linkage method, which determines the distance measure to minimize when choosing the clusters that should be merged next.

In the following, we present some similarity and distance measures and linkage methods commonly used with hierarchical agglomerative clustering. Let $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ denote the feature values of two data points, and $A, B \subseteq \mathbb{R}^n$ be two clusters of feature values.

Commonly used distance and similarity measures include the Manhattan distance

$$d_{\text{manhattan}}(\mathbf{a}, \mathbf{b}) := \sum_{i=1}^{n} |a_i - b_i|,$$

the Euclidean distance

$$d_{\text{euclidean}}(\mathbf{a}, \mathbf{b}) := \|\mathbf{a} - \mathbf{b}\| = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2},$$

as well as the cosine similarity

$$s_{\cos}(\mathbf{a}, \mathbf{b}) := \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|}.$$

From a given distance measure d, many linkage methods can be defined. In contrast to the distance and similarity measures shown before, linkage methods act on pairs of clusters of data points, rather than pairs of single data points. Each linkage method defines how the distance between two clusters is determined, which we denote by ℓ. Strategies commonly used with

hierarchical clustering include average linkage, where

$$\ell_{\text{average}}(A, B) \coloneqq \frac{1}{|A| \cdot |B|} \sum_{\mathbf{a} \in A} \sum_{\mathbf{b} \in B} d(\mathbf{a}, \mathbf{b}).$$

This strategy minimizes the average distance between any two data points in the new cluster. Maximum or complete-linkage minimizes the maximum distance between points in a cluster, formally defined by

$$\ell_{\text{complete}}(A, B) \coloneqq \max_{\mathbf{a} \in A, \mathbf{b} \in B} d(\mathbf{a}, \mathbf{b}),$$

and minimum or single-linkage minimizes the smallest distance in the new cluster, that is,

$$\ell_{\text{single}}(A, B) \coloneqq \min_{\mathbf{a} \in A, \mathbf{b} \in B} d(\mathbf{a}, \mathbf{b}).$$

Finally, Ward's clustering [50] minimizes the total in-cluster variance by choosing the clusters that minimize

$$\ell_{\text{ward}}(A, B) \coloneqq \frac{|A| \cdot |B|}{|A \cup B|} d_{\text{euclidean}}(\boldsymbol{\mu}_A, \boldsymbol{\mu}_B)^2, \tag{2.20}$$

where

$$\boldsymbol{\mu}_A = \frac{1}{|A|} \sum_{\mathbf{a} \in A} \mathbf{a} \quad \text{and} \quad \boldsymbol{\mu}_B = \frac{1}{|B|} \sum_{\mathbf{b} \in B} \mathbf{b}. \tag{2.21}$$

3 Related Work

This chapter gives an overview of work related to our approach. First, we present a method to distill AC-GNNs into a symbolic form [40]. As it forms the basis of our approach (see Chapter 4), it is discussed in more detail. There is also an approach by Ståhlberg et al. [45] to check if an R-GNN has learned a specified set of features, which is briefly presented afterward.

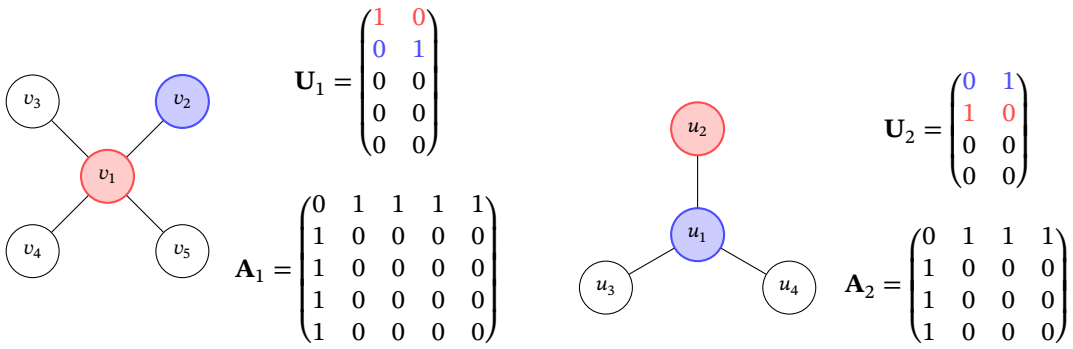
3.1 Logical Distillation of GNNs

A trained GNN is typically a black box to the user, that is, its learned function is neither transparent nor readable, leading to burgeoning interest in AI explainability. Due to the proven correspondence between GNNs and logic, there is significant motivation to find a logic formula equivalent to a given GNN, as this would provide a full and interpretable description of the model. However, there is no known, scalable, straightforward way to do so.

Recently, Pluska et al. [40] proposed iterated decision trees (IDTs) as a novel model structure and an algorithm to learn them from the embeddings of fitted AC-GNNs. In contrast to many neural network explainability methods, which have the goal of making a model's predictions understandable, the main purpose of IDTs is to learn symbolic prediction functions guided by neural networks, and then to be used as prediction models on their own.

The intuition behind IDTs is that an AC-GNN works in layers, and that a whole AC-GNN can be described by formulas. To reflect this, IDTs also consist of layers applied sequentially, each of which corresponds to extractable formulas, which could theoretically have been learned by the GNN. Each IDT layer learns a decision tree, whose splits correspond to logic formulas. As the decision trees receive data in a specific form, the splits they make can be expressed in the logic language $\mathcal{EMLC}$, which is proven to have the same expressiveness as C_2, and thus, as GNNs. By applying layers one after another, formulas of increasingly higher quantifier depth can be represented.

In this framework, Pluska et al. consider graphs consisting of a set V of n nodes, a single undirected edge relation E, and m Boolean node features. The features are represented by the feature matrix $\mathbf{U} \in \mathbb{B}^{n \times m}$, where $\mathbf{U}_{ij} = \mathbb{I}[\text{node } i \text{ has feature } j]$. For an example of such graphs, see Figure 3.1. Every graph has a class label, and the considered GNNs are trained to predict the label, that is, the task is graph classification. The learned IDTs therefore constitute graph classifiers.



(a) Graph G_1. (b) Graph G_2.

Figure 3.1: Example graphs G_1 and G_2 with their feature and adjacency matrices. The node features are $u_1 = \textit{red}$ and $u_2 = \textit{blue}$.

In the following, the logic $\mathcal{EMLC}$ is introduced first. Afterward, the definition of IDTs and the proposed learning method are introduced.

3.1.1 The Logic Language $\mathcal{EMLC}$

The logic language $\mathcal{EMLC}$ [7, 35] is central to their approach, as it is the language in which the learned formulas are expressed. It includes only formulas that have to be evaluated on a single element, comparable to first-order logic formulas with exactly one free variable. Indeed, as proven by Lutz et al. [35], the expressive power of $\mathcal{EMLC}$ formulas is equivalent to that of C_2 formulas with one free variable. Thus, the language can be used to express node classifiers (as introduced in Section 2.7). This language is especially suited to make statements about nodes of graphs, as the neighborhood of nodes is explicitly part of the language's grammar. In the following, we present a restricted definition for $\mathcal{EMLC}$: The *modal parameters* are defined as $\mathbf{I}$, $\mathbf{A}$, and $\mathbf{1}$. For a graph G and a node v, their *interpretations* ε are

$$\varepsilon_{\mathbf{I}}(v) \coloneqq \{v\}, \tag{3.1}$$

$$\varepsilon_{\mathbf{A}}(v) \coloneqq N(v), \text{ and} \tag{3.2}$$

$$\varepsilon_{\mathbf{1}}(v) \coloneqq V. \tag{3.3}$$

An $\mathcal{EMLC}$ formula is defined by the grammar

$$\varphi ::= u_j \mid \top \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid S\varphi > t, \tag{3.4}$$

where u_j is a node feature, S is one of the modal parameters, and $t \in \mathbb{N}$. The semantics are defined as follows:

$$\langle G, v\rangle \vDash u_j \qquad \text{iff node } v \text{ has feature } u_j \tag{3.5}$$

$$\langle G, v\rangle \vDash \top \qquad \forall v \tag{3.6}$$

$$\langle G, v\rangle \vDash \varphi \wedge \psi \qquad \text{iff } \langle G, v\rangle \vDash \varphi \text{ and } \langle G, v\rangle \vDash \psi \tag{3.7}$$

$$\langle G, v\rangle \vDash \varphi \vee \psi \qquad \text{iff } \langle G, v\rangle \vDash \varphi \text{ or } \langle G, v\rangle \vDash \psi \tag{3.8}$$

$$\langle G, v\rangle \vDash \neg\varphi \qquad \text{iff } \langle G, v\rangle \nvDash \varphi \tag{3.9}$$

$$\langle G, v\rangle \vDash S\varphi > t \qquad \text{iff } |\{w \mid w \in \varepsilon_S(v), \langle G, w\rangle \vDash \varphi\}| > t \tag{3.10}$$

To evaluate such formulas on a graph of n nodes, it is helpful to identify $\mathbf{I}$ with the identity matrix of size $n \times n$, $\mathbf{A}$ with the graph's adjacency matrix, and $\mathbf{1}$ with the all-ones-matrix of size $n \times n$. Also, each feature u_j is identified with the j-th column of the feature matrix $\mathbf{U}$, true with 1, and false with 0. Evaluating formulas for all nodes can then be vectorized: For any formula φ, let φ^G denote the Boolean vector which consists of the truth values of φ on every node of G. More specifically, for all indices $i \in \{1, \dots, n\}$, $\varphi_i^G = \langle G, v_i\rangle \vDash \varphi$. Then, the usual logic symbols are applied over all the vector's values, for example $(\varphi \wedge \psi)_i^G = \varphi_i^G \wedge \psi_i^G$, and $(\neg\varphi)_i^G = \left(\neg\varphi^G\right)_i$. The modal parameters can be evaluated using matrix-vector multiplication: For any modal parameter S and Boolean vector φ^G of size n, it holds that

$$\left(S \cdot \varphi^G\right)_i = |\{w \mid w \in \varepsilon_S(v_i), \langle G, w\rangle \vDash \varphi\}|. \tag{3.11}$$

For example, the product $\mathbf{A}\varphi$ gives for each node the number of neighbors that satisfy φ. Let $\psi \coloneqq S\varphi > t$, then $\psi_i^G = \left(S \cdot \varphi^G\right)_i > t$. The calculation of Equation (3.11) can be extended to a matrix of features $\mathbf{U}$, e.g.,

$$\left(\mathbf{A} \cdot \mathbf{U}\right)_{ij} = \left|\left\{w \mid w \in N(v_i), \langle G, w\rangle \vDash u_j\right\}\right|. \tag{3.12}$$

Note that every formula φ is equivalent to $\mathbf{I}\varphi > 0$, a fact that will be used later.

As an example, consider the $\mathcal{EMLC}$ formulas and equivalent C_2 formulas in Table 3.1, which are defined over graphs. The formula φ_1 is made true if and only if the node has more than three neighbors, and φ_2 is satisfied if and only if the node is blue and has at least one neighbor, that is not red. The more complex formula φ_3 is satisfied if and only if the node has at least one neighbor, that does not have any blue neighbors. Their evaluation on the graphs of Figure 3.1 is shown in Table 3.2.

The introduced part of $\mathcal{EMLC}$ is restricted in the sense that the full definition allows for five more modal parameters. These modal parameters allow making statements about non-

neighbor nodes like **1**, and are required for $\mathcal{EMLC}$ to have the exact same expressive power as C_2. For our purposes, however, the introduced modal parameters are sufficient.

Table 3.1: Example $\mathcal{EMLC}$ formulas and equivalent C_2 formulas over graphs.

$\mathcal{EMLC}$	C_2
$\varphi_1 \coloneqq \mathbf{A}\top > 3$	$\varphi_1(x) \coloneqq \exists^{\geq 4} y\, E(x,y)$
$\varphi_2 \coloneqq \mathit{blue} \wedge (\mathbf{A}\neg \mathit{red}) > 1$	$\varphi_2(x) \coloneqq \mathit{blue}(x) \wedge \exists^{\geq 2} y\, (E(x,y) \wedge \neg \mathit{red}(y))$
$\varphi_3 \coloneqq \mathbf{A}\neg(\mathbf{A}\mathit{blue} > 0) > 0$	$\varphi_3(x) \coloneqq \exists y\, (E(x,y) \wedge \neg \exists x\, (E(x,y) \wedge \mathit{blue}(x)))$

Table 3.2: Evaluation of the formulas φ_1, φ_2, and φ_3 on the graphs G_1 and G_2.

	G_1	G_2
φ_1	$\mathbf{A}\top > 3$	$\mathbf{A}\top > 3$
	$\equiv (4,1,1,1,1)^{\mathrm{T}} > 3$	$\equiv (3,1,1,1)^{\mathrm{T}} > 3$
	$\equiv (1,0,0,0,0)^{\mathrm{T}}$	$\equiv (0,0,0,0)^{\mathrm{T}}$
	$\equiv$ node v_1	$\equiv$ no node
φ_2	$\mathit{blue} \wedge (\mathbf{A}\neg \mathit{red} > 1)$	$\mathit{blue} \wedge (\mathbf{A}\neg \mathit{red} > 1)$
	$\equiv (0,1,0,0,0)^{\mathrm{T}} \wedge \left(\mathbf{A}\neg(1,0,0,0,0)^{\mathrm{T}} > 1\right)$	$\equiv (1,0,0,0)^{\mathrm{T}} \wedge \left(\mathbf{A}\neg(0,1,0,0)^{\mathrm{T}} > 1\right)$
	$\equiv (0,1,0,0,0)^{\mathrm{T}} \wedge \left(\mathbf{A}(0,1,1,1,1)^{\mathrm{T}} > 1\right)$	$\equiv (1,0,0,0)^{\mathrm{T}} \wedge \left(\mathbf{A}(1,0,1,1)^{\mathrm{T}} > 1\right)$
	$\equiv (0,1,0,0,0)^{\mathrm{T}} \wedge \left((4,0,0,0,0)^{\mathrm{T}} > 1\right)$	$\equiv (1,0,0,0)^{\mathrm{T}} \wedge \left((2,1,1,1)^{\mathrm{T}} > 1\right)$
	$\equiv (0,1,0,0,0)^{\mathrm{T}} \wedge (1,0,0,0,0)^{\mathrm{T}}$	$\equiv (1,0,0,0)^{\mathrm{T}} \wedge (1,0,0,0)^{\mathrm{T}}$
	$\equiv (0,0,0,0,0)^{\mathrm{T}}$	$\equiv (1,0,0,0)^{\mathrm{T}}$
	$\equiv$ no node	$\equiv$ node u_1
φ_3	$\mathbf{A}(\neg(\mathbf{A}\mathit{blue} > 0)) > 0$	$\mathbf{A}(\neg(\mathbf{A}\mathit{blue} > 0)) > 0$
	$\equiv \mathbf{A}\neg\left(\mathbf{A}(0,1,0,0,0)^{\mathrm{T}} > 0\right) > 0$	$\equiv \mathbf{A}\neg\left(\mathbf{A}(1,0,0,0)^{\mathrm{T}} > 0\right) > 0$
	$\equiv \mathbf{A}\neg\left((1,0,0,0,0)^{\mathrm{T}} > 0\right) > 0$	$\equiv \mathbf{A}\neg\left((0,1,1,1)^{\mathrm{T}} > 0\right) > 0$
	$\equiv \mathbf{A}\neg(1,0,0,0,0)^{\mathrm{T}} > 0$	$\equiv \mathbf{A}\neg(0,1,1,1)^{\mathrm{T}} > 0$
	$\equiv \mathbf{A}(0,1,1,1,1)^{\mathrm{T}} > 0$	$\equiv \mathbf{A}(1,0,0,0)^{\mathrm{T}} > 0$
	$\equiv (4,0,0,0,0)^{\mathrm{T}} > 0$	$\equiv (0,1,1,1)^{\mathrm{T}} > 0$
	$\equiv (1,0,0,0,0)^{\mathrm{T}}$	$\equiv (0,1,1,1)^{\mathrm{T}}$
	$\equiv$ node v_1	$\equiv$ nodes u_2, u_3, u_4

3.1.2 Iterated Decision Trees

In the following, IDTs are introduced. This section is divided into the formal definition, and the proposed learning algorithm. Recall that for GNNs, there are strictly proven results on their expressiveness, but there are no general guarantees on the models that are learned. For example, one could train a GNN to classify graphs that can be distinguished by C_2 formulas, but it may

still be that the GNN will not be perfectly accurate. This is because the training procedure, which is usually some form of gradient descent (see Section 2.4.2), does not guarantee that the trained model will perform well. Thus, it makes sense to make a distinction between the model's definition and its learning method. The case of IDTs is similar, as there is a formal definition and proven results on its expressive power, but the proposed learning method works heuristically and does not offer guarantees. In the following, we therefore distinguish between the IDT model definition and learning method.

Formal Definition and Expressiveness

An IDT is a sequence of layers, each of which corresponds to a set of $\mathcal{EMLC}$ formulas. When another layer is appended, it works on the formulas of the previous layers and can combine these using Boolean operations and modal parameters, which results in new formulas, that is, a set of disjunctions D. Thus, with an increasing number of layers, formulas with a higher number of nested modal parameters can be represented.

Formally, Pluska et al. [40] define an IDT layer to be composed of

- a decision tree that splits on the features $S\varphi > t$ for a modal parameter S and $t \in \mathbb{N}$, and
- a set D of leaf sets the layer's tree, which thus corresponds to a set of disjunctions of splitting paths (see Section 2.8).

As the modal parameters of splitting features can also be $\mathbf{I}$, a layer does not necessarily nest another modal parameter, but can also use existing features from its input.

From this, an IDT is defined as a sequence of L IDT layers. For every layer depth $i = 0, \dots, L-1$, every split $S\varphi > t$ of layer i either has to be

- on a formula φ of depth 0, or
- on a formula φ that is equivalent to a disjunction $d \in D^{(l)}$ of any previous layer $l < i$.

Similar to AC-GNNs, there is a crisp result on the expressiveness of IDTs: For every $\mathcal{EMLC}$ formula ψ with at most L nested modal parameters, an L layer IDT exists such that for some $d \in D^{(L-1)}$, $\psi \equiv d$. On the other hand, given an IDT of L layers, for each formula $d \in D^{(L-1)}$, there is an $\mathcal{EMLC}$ formula ψ with at most L nested modal parameters such that $\psi \equiv d$.

Learning Algorithm

To learn an IDT from an AC-GNN, a learning algorithm is proposed. It is explained in the following, and Algorithm 2 displays its pseudocode. Like GNNs, it requires a set of graphs as training data. To understand the algorithm, it is helpful to see a set of graphs as a single joined graph, where the individual graphs are disjoint. Following this view, one can define the adjacency and feature matrix of a set of graphs. For fitting an IDT, besides the graphs $\mathcal{G}$, the embeddings of the nodes of $\mathcal{G}$ after each layer of the GNN are required.

Algorithm 2 IDT Learning Algorithm.

Hyperparameters: width w, sample size s
procedure LEARNIDT($\mathcal{G}, \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(L)}, \mathbf{y}_{\text{true}}$)
  $layers \leftarrow$ EMPTYLIST()
  for $i = 1, \dots, L$ **do**
    $layers[i] \leftarrow$ EMPTYLIST()
    for $j = 1, \dots, w$ **do** *Learn w parallel layers at every depth*
      $\langle \widetilde{\mathcal{G}}, \widetilde{\mathbf{U}}, \tilde{\mathbf{x}} \rangle \leftarrow \text{SAMPLE}(s, \mathcal{G}, \mathbf{U}, \mathbf{x}^{(i)})$
      $\ell \leftarrow \text{LEARNINNERLAYER}(\widetilde{\mathcal{G}}, \widetilde{\mathbf{U}}, \tilde{\mathbf{x}})$ *Learn layer based on previous features*
      APPEND($layers[i], \ell$)
      $D_\ell = \{D_1, \dots, D_N\} \leftarrow \text{LEAFSETS}(\ell)$
      $\mathbf{U}_{\text{new}}^{(j)} \leftarrow [\text{FEATUREVECTOR}(\mathcal{G}, D_1) \quad \dots \quad \text{FEATUREVECTOR}(\mathcal{G}, D_N)]$ *Evaluate new features for all nodes*
    $\mathbf{U} \leftarrow \begin{bmatrix} \mathbf{U} & \mathbf{U}_{\text{new}}^{(1)} & \dots & \mathbf{U}_{\text{new}}^{(w)} \end{bmatrix}$ *Append the new features of this layer depth*
  $final \leftarrow \text{FITDECISIONTREE}(\mathbf{1} \cdot \mathbf{U}, \mathbf{y}_{\text{true}})$ *Learn to predict $\mathbf{y}_{\text{true}}$ based on $\mathbf{1} \cdot \mathbf{U}$*
  return $\langle layers, final \rangle$
procedure LEARNINNERLAYER($\mathcal{G}, \mathbf{U}, \mathbf{x}$)
  $\mathbf{A} \leftarrow$ adjacency matrix of $\mathcal{G}$
  $t \leftarrow \text{FITDECISIONTREE}([\mathbf{U} \quad \mathbf{A} \cdot \mathbf{U}], \mathbf{x})$ *Learn to predict $\mathbf{x}$ based on $\mathbf{U}$ and $\mathbf{A} \cdot \mathbf{U}$*
  $c \leftarrow \text{FITCLUSTERING}(t)$ *Learn clustering of leaf nodes based on their predictions*
  return $\langle t, c \rangle$

For each GNN layer, an IDT layer is created. The layer's decision tree is fitted to predict the nodes' embeddings $\mathbf{x}$ as computed by the GNN. The tree's predictions are learned based on the given graph features $\mathbf{U}$ and the features that were computed by any previous IDT layer. As in the formal definition of the layers, splits are learned in the form of $S\varphi > t$, where φ is one of the given features, that is, a column of $\mathbf{U}$. Analogously to the AC-GNN, which only has local information available at each node, S is restricted to $\mathbf{I}$ and $\mathbf{A}$. For each node and feature, the trees get two types of information, that is,

- whether the node has the feature. The feature matrix $\mathbf{U}$ contains this information for all nodes and features. The second type of information is
- the number of neighbors of the node that have the feature. For all nodes and features, this is given by the result of $\mathbf{A} \cdot \mathbf{U}$ (see Equation (3.12)).

When a tree makes a splitting decision, it effectively selects a combination of a modal parameter S and a feature φ and determines a splitting threshold t, resulting in a split on the derived $\mathcal{EMLC}$ feature $S\varphi < t$. Such a split results in two node partitions, where one only has $S\varphi < t$, and the other one only has $\neg(S\varphi < t) \equiv S\varphi \geq t$. Note that each such partition corresponds to a conjunction of $\mathcal{EMLC}$ formulas, and thus, corresponds to an $\mathcal{EMLC}$ formula itself according to the definition (Equation (3.4)).

After learning the decision tree, the set of leaf sets of the layer is computed. This is done by applying hierarchical agglomerative clustering (see Section 2.9) to the leaf nodes of the decision tree. The leaf nodes and their respective predictions $\hat{\mathbf{y}} \in \mathbb{R}^k$ of the embeddings are used as the input of the clustering, which determines hierarchically growing sets of the leaf nodes. When there are l leaf nodes, the clustering results in $2l - 1$ sets, as opposed to taking all combinations of the leaf nodes, which would result in 2^l sets. Since a set of leaf sets corresponds to a disjunction of $\mathcal{EMLC}$ formulas, it itself corresponds to an $\mathcal{EMLC}$ formula.

As an example, consider the IDT layer depicted in Figure 3.2. The equivalent formulas for each leaf set are shown in Table 3.3.

Table 3.3: Formulas corresponding to the leaf sets as determined by the IDT layer shown in Figure 3.2.

Leaf set	**Formula**
$\{P_1\}$	$blue < 1 \wedge \mathbf{A}red < 1 \equiv \neg blue \wedge \mathbf{A}red < 1$
$\{P_2\}$	$blue < 1 \wedge \neg(\mathbf{A}red < 1) \equiv \neg blue \wedge \mathbf{A}red \geq 1$
$\{P_3\}$	$\neg(blue < 1) \wedge \mathbf{A}\top < 2 \equiv blue \wedge \mathbf{A}\top < 2$
$\{P_4\}$	$\neg(blue < 1) \wedge \neg(\mathbf{A}\top < 2) \equiv blue \wedge \mathbf{A}\top \geq 3$
C_1	$\{P_1, P_2\} \equiv (\neg blue \wedge \mathbf{A}red < 1) \vee (\neg blue \wedge \mathbf{A}red \geq 1) \equiv \neg blue$
C_2	$C_1 \cup \{P_3\} \equiv (blue \wedge \mathbf{A}\top < 2) \vee \neg blue$
C_3	$C_2 \cup \{P_4\} \equiv (blue \wedge \mathbf{A}\top < 2) \vee (\neg blue) \vee (blue \wedge \mathbf{A}\top \geq 3) \equiv \top$

After a layer has been learned, its derived features in the form of leaf sets are explicitly evaluated on all training sample nodes. In the pseudocode, this is denoted by FeatureVector($\mathcal{G}, D$) for a leaf set D. For example, consider the leaf set $\{P_1\}$, its corresponding formula, and the graphs from Figure 3.1. Of G_1, only node v_1 satisfies the feature, and thus,

$$\text{FeatureVector}(G_1, \{P_1\}) = (1, 0, 0, 0, 0)^{\mathrm{T}}.$$

Regarding G_2, all nodes except u_1 satisfy the feature, therefore

$$\text{FeatureVector}(G_2, \{P_1\}) = (0, 1, 1, 1)^{\mathrm{T}}.$$

As the setting in which IDTs are used is graph classification, a special final layer is appended after the L-th layer. It uses a decision tree classifier instead of a regressor, and does not cluster leaf nodes. This layer is fitted to predict the graphs' true labels based on $\mathbf{1}\varphi$. The use of modal parameter $\mathbf{1}$ results in every node of each graph having the same valuation for any feature, that is, at this point, only graph-level but no node-level distinctions can be made. Thus, the final layer learns to classify graphs.

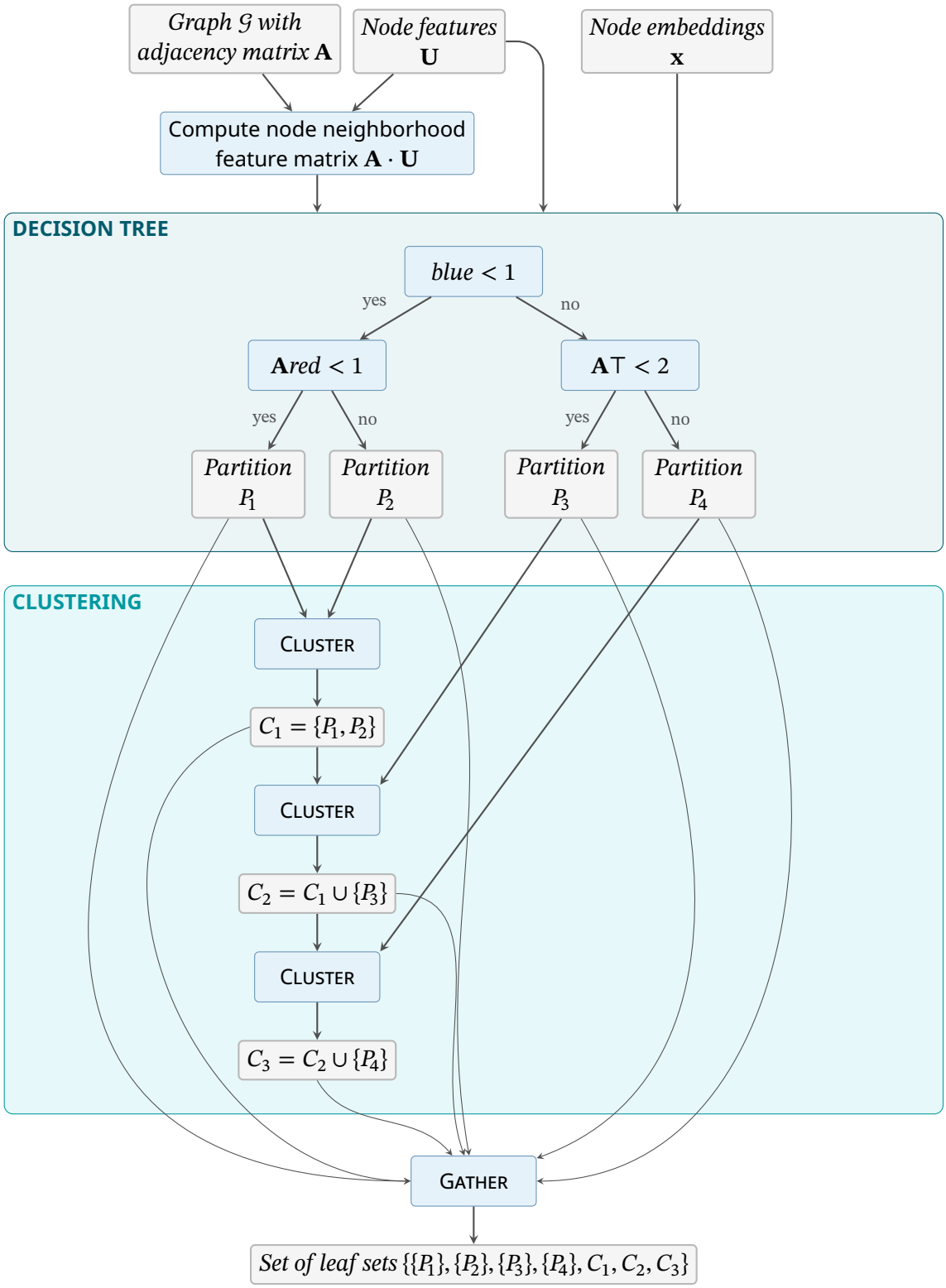


Figure 3.2: Schematic figure of a single IDT layer using the features *red* and *blue*. The upper block shows a decision tree with four leaf nodes, and the lower block shows a clustering of these leaf nodes. The layer results in a set of seven sets of leaf nodes.

Both the decision tree and clustering learning are heuristic approaches, which have an intuitive backing. It is a reasonable assumption that nodes with similar logical properties with respect to the task learned by the GNN will be embedded similarly. Learning decision trees and clustering on embeddings is where properties expressed in an interpretable way, that is, $\mathcal{EMLC}$ formulas, are matched with what the GNN has learned in an incomprehensible form.

The authors observe empirically that it is beneficial to create multiple layers in parallel for each GNN layer. The number of such parallel layers is called the width w. Each parallel layer is trained on a randomly sampled subset of s nodes of $\mathcal{G}$. The decision tree regressors use the mean squared error (Equation (2.17)) as the heterogeneity measure, and use Ward's linkage strategy (Equation (2.20)) for the hierarchical clustering. To increase the models' interpretability, the maximal depth of the decision trees is limited to two.

3.2 Explicit Feature Testing

Ståhlberg et al. [45] present handcrafted features and linear value functions for some of the covered general planning problem classes. Their approach to understanding a learned R-GNN is to check whether the pooled values of the R-GNN contain a linear transformation of the valuations of their handcrafted features. This method requires that a hypothesis about the learned features is stated explicitly.

Specifically, their models perform pooling as

$$v \leftarrow \mathsf{MLP}_2\Big(\sum_{o \in O} \mathsf{MLP}_1\Big(\mathbf{x}_o^{(L)}\Big)\Big),$$

where $\mathsf{MLP}_2(\mathbf{x}) \coloneqq \mathrm{Linear}(\mathrm{ReLU}(\mathbf{x}))$. Ståhlberg et al. collect the valuations of the pooling at each layer of MLP_2, that is, they define the layerwise pool vector $\mathbf{p}$ as the concatenation of the values

$$\begin{aligned}
\mathbf{p}' &\coloneqq \sum_{o \in O} \mathsf{MLP}_1\Big(\mathbf{x}_o^{(L)}\Big), \\
\mathbf{p}'' &\coloneqq \mathrm{ReLU}(\mathbf{p}'), \text{and} \\
\mathbf{p}''' &\coloneqq \mathrm{Linear}(\mathbf{p}'').
\end{aligned}$$

Further, let $\mathbf{y} \in \mathbb{R}^n$ be the vector with the valuations of the n handcrafted features. To check whether $\mathbf{p} \in \mathbb{R}^j$ is a linear transformation of the handcrafted features, they fit a regression, optimizing the weights $\mathbf{A} \in \mathbb{R}^{n \times j}$ and $\mathbf{b} \in \mathbb{R}^n$ to minimize the overall loss $\sum_{i=1}^{n} \left|(\mathbf{Ap} + \mathbf{b} - \mathbf{y})_i\right|$. This loss is the sum of absolute differences between the handcrafted features' valuations and the linear transformation of the values the R-GNN computes during its pooling operation.

When the loss is close to 0, it is likely that the R-GNN has indeed learned features equivalent to the handcrafted ones.

Using this method, Ståhlberg et al. conclude that some of their R-GNNs can be interpreted in terms of the handcrafted features. However, for some problems, the features their models have learned remain unknown.

4 Methods

In this chapter, we introduce our approach to apply logical distillation to the R-GNNs employed for learning optimal value functions of generalized planning problems. The basis for our approach is the IDT model and learning algorithm by Pluska et al. [40], see Section 3.1. While IDTs operate on graphs, nodes, node features, and edges, the corresponding terms for our method are state graphs, objects, object features, and edges of different relations. There are several key aspects in which our framework differs from theirs, and thus, several adaptations are required. We call the core of the adapted method *relational IDT (R-IDT)*. Additionally to R-IDTs, our setting also requires fitting a regression. The modifications and additions of the approach are summarized in the following.

1. In contrast to the framework of IDTs, our input is not directly given in the form of graphs, but planning states. Therefore, we first need to encode each state as a graph. In Section 4.1, we detail how the graphs are created.
2. The state graphs that we create have multiple relations, as opposed to the single edge relation E that IDTs can handle. Our modifications are both theoretical, as the logic $\mathcal{EMLC}$ as used by Pluska et al. [40] is no longer sufficient to express the required features, and practical, since computational concerns arise when considering multiple relations. Section 4.2 covers our additions to support multiple relations.
3. Another computational issue is that equivalent and thus, unnecessary features are generated. In Section 4.3, our observations and additions are discussed briefly.
4. Unlike the original IDT framework, we are concerned with graph-level regression instead of classification. Different types of state features require learning formulas at multiple levels, which IDTs alone cannot do. Section 4.4 further clarifies the issue, and presents an approach to overcome it. This modification concludes the part on R-IDTs, before the regression is discussed.
5. To finally fit the regression on the optimal value function, there are multiple viable approaches, which are discussed in Section 4.5.

Since several hyperparameters are introduced throughout this chapter, Section 4.6 provides an overview of the parameters and their possible values.

4.1 State Encodings

Planning states have to be encoded as graphs with multiple relations in order to use R-IDTs on them. In Algorithm 3, our encoding is defined. It assumes that the goal is already included in

the state using goal predicates as introduced in Section 2.2.2. In summary, the nodes of the graph are the objects of the instance. The edges of the state graph are defined by the binary atoms, and the features of the objects by the unary atoms. To illustrate the encoding, Figure 4.1 shows an example of Blocksworld states encoded as graphs.

Algorithm 3 Encoder mapping planning states to graphs.

procedure ENCODE(s, O, P)
  $\langle o_1, \dots, o_n \rangle \leftarrow \text{SORT}(O)$ — *Fix ordering of objects*
  $\langle u_1, \dots, u_l \rangle \leftarrow \text{SORT}(\{p \mid p \in P, \text{ar}(p) = 1\})$ — *Fix ordering of unary predicates*
  $B \leftarrow \{p \mid p \in P, \text{ar}(p) = 2\}$ — *Get the set of binary predicates*
  $\mathbf{U} \leftarrow \mathbf{0}^{n \times m}$ — *Initialize empty feature matrix*
  for all unary atoms $u(o) \in s$ **do**
    Let i, j be the indices such that $o_i = o$ and $u_j = u$
    $\mathbf{U}_{ij} \leftarrow 1$ — *Set Boolean entry of predicate-object combination to true*
  for all binary predicates $b \in B$ **do**
    $E^b \leftarrow \{\langle o_1, o_2 \rangle \in O^2 \mid b(o_1, o_2) \in s\}$ — *Construct the edge set of binary predicate b*
  $G \leftarrow \left\langle O, \langle E^b \rangle_{b \in B}, \mathbf{U} \right\rangle$
  return G

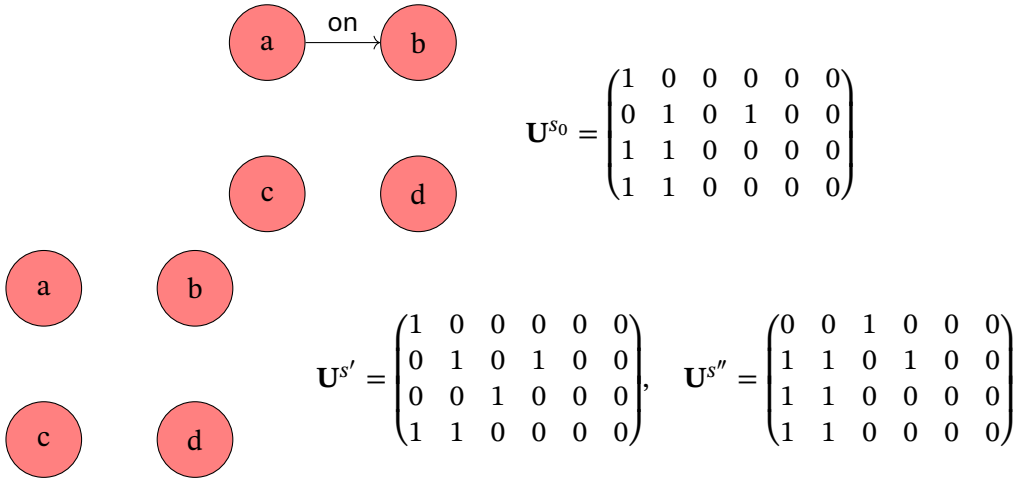


Figure 4.1: The states shown in Figure 2.1, encoded as graphs. The upper part shows the graph of s_0, while the lower part shows those of s' and s''. For the feature matrices, assume the object ordering $\langle o_1, \dots, o_4 \rangle = \langle \text{a}, \text{b}, \text{c}, \text{d} \rangle$, and the unary predicate ordering $\langle u_1, \dots, u_6 \rangle = \langle \texttt{clear}, \texttt{ontable}, \texttt{holding}, \texttt{clear}_\text{G}, \texttt{ontable}_\text{G}, \texttt{holding}_\text{G} \rangle$. The latter states' graphs only differ in their feature matrices $\mathbf{U}^{s'}$ and $\mathbf{U}^{s''}$.

For an overview of different methods to encode planning states as graphs and the impact of the choice of encoding, we refer to [29]. We chose the presented encoding method because it is similar to that used in [45–47] and to that used by the R-GNN implementation by Aichmüller and Krude [4] leveraged for our experiments. The state encoding method that we use to generate the input for the R-GNNs and the presented method for the R-IDTs are equivalent in

their expressiveness and thus, their differences are not expected to have any influence on our results.

Without loss of generality, it is sufficient to assume that only unary and binary predicates are given. The reason for this is that we are attempting to distill GNNs whose expressive power is bounded by C_2, and that every first-order logic formula expressible in C_2, but making use of more than two variables, can equivalently be written as a formula making use of only two variables (see the definition of C_2 in Section 2.7). When there are only two variables, it is sufficient to only consider predicates with an arity of at most two.

4.2 Supporting Multiple Relations

In contrast to the framework IDTs were originally considered for, we do not only have a single symmetric binary relation E that connects nodes, but every binary predicate induces a possibly non-symmetric binary relation between objects. To adapt to this relational context, we propose R-IDTs, which consider not only one relation for splitting decisions, but can choose to split on any of the available relations.

Recall that the logic $\mathcal{EMLC}$ considers only one relation, and thus, the single modal parameter $\mathbf{A}$ is sufficient to express statements about the neighbors of a node (see Section 3.1). In the relational framework, it is necessary to distinguish the relations, and the logic language needs to be appropriately extended. Therefore, for each relation R, $\mathcal{EMLC}$ is extended by the modal parameters $\mathbf{A}_R$ and $\mathbf{A}_R^{\mathrm{T}}$. Their interpretations are defined as

$$\varepsilon_{\mathbf{A}_R}(v) \coloneqq \{w \mid R(v,w)\}, \text{ and} \tag{4.1}$$

$$\varepsilon_{\mathbf{A}_R^{\mathrm{T}}}(v) \coloneqq \{w \mid R(w,v)\}, \tag{4.2}$$

and thus are generalizations of $\mathbf{A}$ (see Equation (3.2)). This extension is easily integrated, as the grammar of $\mathcal{EMLC}$ (Equation (3.4)) and the semantics (Equations (3.5) to (3.10)) do not change. In our running example Blocksworld, the binary predicates are on and on_{G}, and the corresponding modal parameters are $\mathbf{A}_{\mathsf{on}}, \mathbf{A}_{\mathsf{on}}^{\mathrm{T}}, \mathbf{A}_{\mathsf{on}_{\mathrm{G}}}$, and $\mathbf{A}_{\mathsf{on}_{\mathrm{G}}}^{\mathrm{T}}$.

Analogously to $\mathcal{EMLC}$, each $\mathbf{A}_R$ is identified with the adjacency matrix with respect to R, and $\mathbf{A}_R^{\mathrm{T}}$ with the transposed adjacency matrix. Since the adjacency matrix $\mathbf{A}_R$ encodes binary relations such that $(\mathbf{A}_R)_{ij} = \mathbb{I}[s \vDash R(o_i, o_j)]$, the transpose encodes the inverse of the relation, as $\left(\mathbf{A}_R^{\mathrm{T}}\right)_{ij} = \mathbb{I}[s \vDash R(o_j, o_i)]$. To calculate the number of "neighboring" objects with respect to a relation that satisfy a feature, the respective adjacency matrix is multiplied with the feature

matrix. Equation (3.12) generalizes to

$$(\mathbf{A}_R \cdot \mathbf{U})_{ij} = \left|\left\{o' \mid G \vDash R(o_i, o'), \langle G, o'\rangle \vDash u_j\right\}\right|, \text{and} \tag{4.3}$$

$$\left(\mathbf{A}_R^{\mathrm{T}} \cdot \mathbf{U}\right)_{ij} = \left|\left\{o' \mid G \vDash R(o', o_i), \langle G, o'\rangle \vDash u_j\right\}\right|. \tag{4.4}$$

To illustrate this on the Blocksworld example class $\mathcal{Q}_{\text{clear}}$, let φ_k be true for an object if and only if it is the block that should be cleared and has k blocks above it. In $\mathcal{EMLC}$, it can be expressed as

$$\varphi_k = \texttt{clear}_{\mathrm{G}} \wedge \eta_k, \tag{4.5}$$

$$\eta_0 = \texttt{clear}, \tag{4.6}$$

$$\eta_k = \mathbf{A}_{\mathsf{on}}^{\mathrm{T}} \eta_{k-1} > 0. \tag{4.7}$$

Recall that $\mathsf{on}(x, y)$ denotes that x is on y. With only $\mathbf{A}_{\mathsf{on}}$ but not $\mathbf{A}_{\mathsf{on}}^{\mathrm{T}}$ available, we would not be able to express η_k, and in consequence, φ_k. This is because with $\mathbf{A}_{\mathsf{on}}$, we can only access blocks *below* a given block, but not *above*. Note the resemblance to the definition of the *state feature* B_k as defined in Equation (2.6). However, also note that B_k was defined as a feature over states, while φ_k is a feature over objects. It can be used to express B_k as

$$B_k \coloneqq \mathbf{1}\varphi_k > 0, \tag{4.8}$$

as the modal parameter $\mathbf{1}$ is defined to range over all objects of a state (see Equation (3.3)).

To accommodate for the more general present case of multiple relations, an R-IDT layer needs to multiply each adjacency matrix with the feature matrix. Because of this, the required computational effort depends on the number of relations. More formally, given a set of adjacency matrices $\mathcal{A} \subseteq \mathbb{B}^{n \times n}$ and a feature matrix $\mathbf{U} \in \mathbb{B}^{n \times l}$ of l features, each adjacency matrix is multiplied with $\mathbf{U}$. Therefore, $|\mathcal{A}|$ matrix-matrix multiplications are performed, resulting in $|\mathcal{A}|$ matrices of size $n \times l$. The number of features a single decision tree needs to consider is

$$(|\mathcal{A}| + 1) \cdot l \in \mathcal{O}(|\mathcal{A}| \cdot l), \tag{4.9}$$

corresponding to the set of modal parameters $\mathcal{A} \cup \{\mathbf{I}\}$. As the matrices have potentially many entries, saving unnecessary calculations can have a considerable impact on computation times. Therefore, before fitting the R-IDT layers, an R-IDT pre-prunes empty relations such as On^{G} for $\mathcal{Q}_{\text{clear}}$. It also checks whether each relation R is symmetric, that is, whether $\mathbf{A}_R = \mathbf{A}_R^{\mathrm{T}}$, in which case it does not keep the inverse relation. Which relations are kept or discarded is decided once on the training data, and then applied when an R-IDT is applied as a predictive model. Note that this does not add any new assumptions or constraints, as we expect the relations that are useful on the training data to be exactly those that are useful on any other data:

- A split determined by a decision tree (see Section 2.8) in an R-IDT layer would never include an empty relation, as for any decision tree node m and dataset $D_m \subseteq D$, the resulting partitions would strictly be D_m and $\varnothing$, respectively. It follows from Equation (2.16) that $q_m(f,t) = h(D_m)$, that is, the split would not decrease heterogeneity measure at all.
- When a relation R is symmetric in the training data, a split on $\mathbf{A}_R$ is as good as a split on $\mathbf{A}_R^{\mathrm{T}}$ and thus, there is no reason to expect that choosing the inverse relation over the given relation is beneficial. Therefore, removing the inverse relation is not expected to limit the model in any way.

4.3 Filtering Unique Features

We note that when training R-IDTs, many equivalent features are generated. This happens both in parallel layers and in subsequent layers. We find that filtering out duplicate features before each layer markedly reduces the overall train time, as the number of calculations at each layer (as shown in Equation (4.9)) scales with the number of features. The influence of filtering features only grows as more and more R-IDT layers are learned sequentially. Specifically, we filter the unique features such that given a feature matrix, only the first occurrence of each feature is kept, that is, subsequent duplicate columns are removed.

Figure 4.2 provides an overview of an R-IDT's parallel layer, that is, what is learned for each layer depth. Learning a single R-IDT layer, denoted by LAYER, is essentially the same as an IDT layer as shown in Figure 3.2, with the exception, that more than one adjacency matrix is used. Further, note the similarities to the IDT learning algorithm shown in Algorithm 2.

4.4 From Object Features to State Features

Unlike Pluska et al. [40], our goal is to do regression on state features, not classification. While the classifier needs to be replaced by some kind of regression, one also needs to consider how to derive the features needed for the regression. Deriving the features is the topic of this section, while the next section discusses the regression.

Recall that the original IDT learning method (see Section 3.1.2) fits a final layer after the L inner layers. This final layer classifies graphs instead of nodes based on the features learned so far, again by using a decision tree. Although the decision tree classifier could simply be replaced by a decision tree regressor and fitted to predict V^*, this would predictably fail to generalize for state values that are not in the training data. This is because decision trees can only predict piecewise constant functions [28], as a tree's predictions are constant values that are grouped nodewise (see Equation (2.15)), and each node covers some partition of the data.

Alternatively, we consider representing V^* in a linear form as introduced in Section 2.2.2. As an R-GNN pools the object embeddings after its last layer into its scalar output, and the last

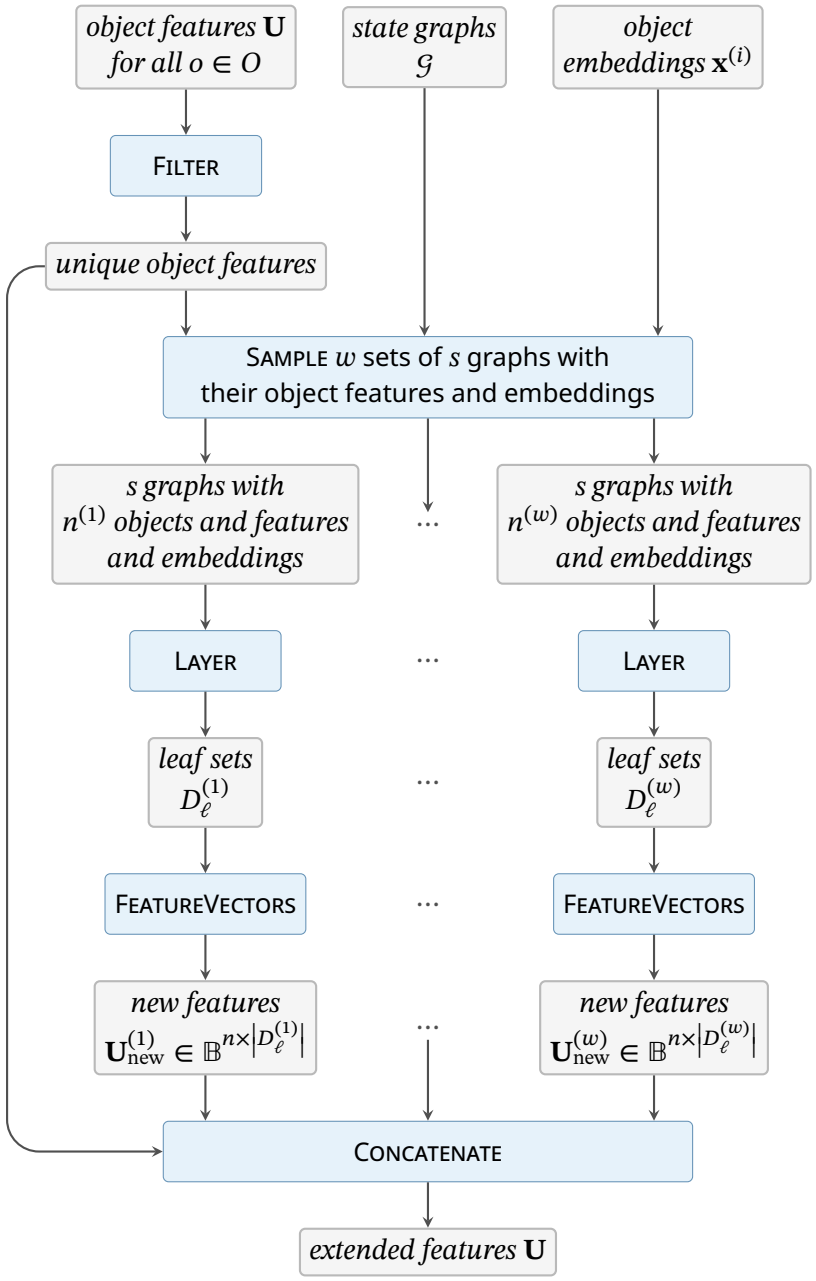


Figure 4.2: Schematic overview of a single parallel R-IDT layer. The input is a set of state graphs $\mathcal{G}$ along with its object features and object embeddings. By FILTER, the unique feature filtering (Section 4.3) is denoted, and LAYER denotes a single R-IDT layer.

layer of an R-IDT is fitted to reflect the GNN's last layer, one might assume that an R-IDT's last layer is also able to learn the features that are needed to compose the value function linearly. However, it turns out that not all required features can be learned by an R-IDT's last layer, but that in principle, the learned features can be used to compose the needed ones. The following sections explain how the features are derived.

4.4.1 Learning Boolean and Numerical State Features

First, we explain our approach to learning Boolean and numerical state features. When an R-IDT layer ℓ is fitted, it learns to divide the objects into a set of leaf sets D_ℓ, depending on logical properties. We can interpret the defining properties of each leaf set as state concepts, as they are formulas that are evaluated on objects (see Section 2.2.2). Since each R-IDT layer clusters the leaf nodes of its tree into sets, D_ℓ technically is a set of sets of leaf nodes, and each set of leaf nodes corresponds to a disjunction of formulas (see Section 2.8). Let all concepts that are learned at any layer of the R-IDT be gathered in $\mathcal{C}$, that is,

$$\mathcal{C} \coloneqq \bigcup_{\ell \in layers} D_\ell = \{c_1, \ldots\}.$$

From the concepts, the numerical state features f_j^{n} and the *simple* Boolean state features f_j^{b} are derived. We call these Boolean state features "simple" to distinguish them from other features introduced later on. For a state s, let $\mathbf{U}(s) \in \mathbb{B}^{n \times |\mathcal{C}|}$ denote the object feature matrix that holds the values of all features derived by an R-IDT for every object of that state. Each numerical feature's value is computed as

$$f_j^{\mathrm{n}}(s) \coloneqq \left|\{o \mid o \in O, s \vDash c_j(o)\}\right| = \sum_{i=1}^{n} \mathbf{U}(s)_{ij}, \tag{4.10}$$

and each simple Boolean feature's value is derived as

$$f_j^{\mathrm{b}}(s) \coloneqq \mathbb{I}\left[f_j^{\mathrm{n}}(s) > 0\right] = \max_{i \in \{1,\ldots,n\}} \mathbf{U}(s)_{ij}, \tag{4.11}$$

that is, each numerical feature evaluates to the number of objects satisfying a concept, and each simple Boolean feature evaluates to whether there are any objects satisfying a concept. Coming back to the example of φ_k of Equation (4.5), the state feature B_k can be expressed as the simple Boolean feature of φ_k. Clearly, the simple Boolean features α and H can also be expressed based on $\mathcal{EMLC}$ formulas. In the following, let the set of numerical state features be denoted by $F^{\mathrm{n}} \coloneqq \left\{f_j^{\mathrm{n}} \mid j \in \{1, \ldots, |\mathcal{C}|\}\right\}$, and the set of simple Boolean state features by $F^{\mathrm{b}} \coloneqq \left\{f_j^{\mathrm{b}} \mid j \in \{1, \ldots, |\mathcal{C}|\}\right\}$.

Note that computing a numerical feature's value also corresponds to applying the modal parameter **1** (Equation (3.3)). Computing a Boolean feature's value is similar, but provides an additional abstraction (Equation (2.1)). This is comparable to sum and max pooling at the end of GNNs. When considering pooling only for Boolean values, the pooling and feature value calculations are in fact the same.

4.4.2 Learning Combinations of State Features

While the previous section detailed how numerical and simple Boolean features state are computed, these types of features are not sufficient to compose optimal linear value functions for any of the problems we consider (see Section 5.1). Optimal linear value functions often involve conjunctions of state features [45], such as $\alpha \wedge H$ in our example for Q_{clear} (Equation (2.3)). However, these are *state* features derived from *object* features. This marks a transition from Boolean combinations expressible in GC_2 to those requiring C_2 formulas, or from the perspective of $\mathcal{EMLC}$, combining formulas that use the modal parameter $\mathbf{1}$ instead of $\mathbf{A}$. While an R-IDT's inner layers learn to combine object features, they cannot learn to combine state features. Simply taking the features based on the concepts $\mathcal{C}$ as defined above also is not sufficient, as there is no pair of weights $w_1, w_2 \in \mathbb{R}$ such that $w_1 b_1 + w_2 b_2 = b_1 \wedge b_2$ or $w_1 b_1 + w_2 b_2 = b_1 \vee b_2$ for all value combinations of the Boolean variables $b_1, b_2 \in \mathbb{B}$. Therefore, a function that is linear in α and H alone cannot truly equal the optimal value function. As we want to learn a linear regression on V^*, it is guaranteed that it will not be able to learn the function based on the types of state features derived so far. To overcome this limitation, such combined features need to be made explicitly available to the regression.

Recall that R-GNNs (Algorithm 1) pool the object embeddings by applying an MLP on a final state-level aggregation, namely

$$v \leftarrow \mathsf{MLP}_{\mathrm{R}}\Bigg(\underbrace{\bigoplus_{o \in O} \mathbf{x}_o^{(L)}}_{\mathbf{e}_s}\Bigg), \tag{4.12}$$

where the considered options for $\bigoplus$ are summation and taking the component-wise maximum. The inner part of the formula can thus be interpreted as a state level embedding $\mathbf{e}_s$.

To learn the combination of features that are needed to compose the value function, an R-IDT learns a final layer, again consisting of decision tree regressor and clustering. This layer is fitted on state-level data to predict the state embeddings $\mathbf{e}_s$. As the input data, it uses either the computed numerical state features F^{n} or the simple Boolean state features F^{b}, resembling sum and max pooling, respectively. This option is included as a hyperparameter. Like the inner R-IDT layers, it clusters the resulting decision tree leaf nodes with numerically close prediction values to create meaningful disjunctions of splitting decisions in a heuristic manner. The splitting decisions that the final layer's tree learns are of the form $f_j^{\mathrm{n/b}}(s) < t$, that is, comparing numerical or simple Boolean features to thresholds. Then, a leaf node corresponds to a conjunction of such splits. Finally, the clustering learns to build disjunctions of leaf nodes, resulting again in a set of disjunctions of conjunctions.

The resulting clusters of leaf nodes can directly be interpreted as Boolean state features, as unlike $\mathcal{C}$, they are not statements about single objects, that is, concepts, but about whole states.

As such, Boolean combinations like $\alpha \wedge H$ can be learned. We call features of this type *combined* (Boolean) state features as to distinguish them from the *simple* Boolean state features. In the following, let F^{c} denote the set of learned combined state features.

Similar to the inner layers, of which there are multiple parallel layers at each layer depth, we find that learning multiple parallel final layers is very helpful for learning the relevant combined state features. One reason for this could be that for some problems like Q_{clear}, many individual combined features can be required, for example the B_ks (see Equation (2.6)). When using only a single final layer, it may be less likely, or in some cases, even impossible, to learn all the required features. When parallel final layers are learned on random subsets of the training state graphs instead, there is a higher chance that more of these features are learned.

In summary, an R-IDT learns three types of state features:

- Simple Boolean features F^{b}, that give information about whether any object satisfies a property in the state,
- numerical features F^{n}, that evaluate to the exact number of objects that satisfy a property in the state, and
- combined features F^{c}, that consist of Boolean combinations based on numerical or simple Boolean state features.

To refer to all learned state features, let $\mathcal{F} := F^{\mathrm{n}} \cup F^{\mathrm{b}} \cup F^{\mathrm{c}}$.

At this point, the core of the approach, that is, R-IDTs, have been introduced. An overview of its composition is displayed in Figure 4.3. The following section introduces the regression component of our approach.

4.5 Learning the Regression

So far, it has been discussed how state features are derived directly based on the instance's objects as well as Boolean combinations of these state features. To finally predict the optimal value function in a linear form based on these features, a linear regression is fitted. On the training set of states $\mathcal{S}$, the regression learns a weight w_f for each feature f and a bias w_b such that for all $s \in \mathcal{S}$,

$$V^*(s) \approx V_{\mathbf{w}}(s) := \sum_{f \in \mathcal{F}} w_f f(s) + w_b. \tag{4.13}$$

Including the bias is optional, and the choice is left as a hyperparameter. Instead of using all features, the option to restrict the features to certain types only is available as a hyperparameter. To understand why this option is useful, consider the following cases:

- For some problems, the optimal value function can be written without using numerical features, but in the training set, some numerical features only take the values 0 or 1. Then, these numerical features equal their respective Boolean features in the training

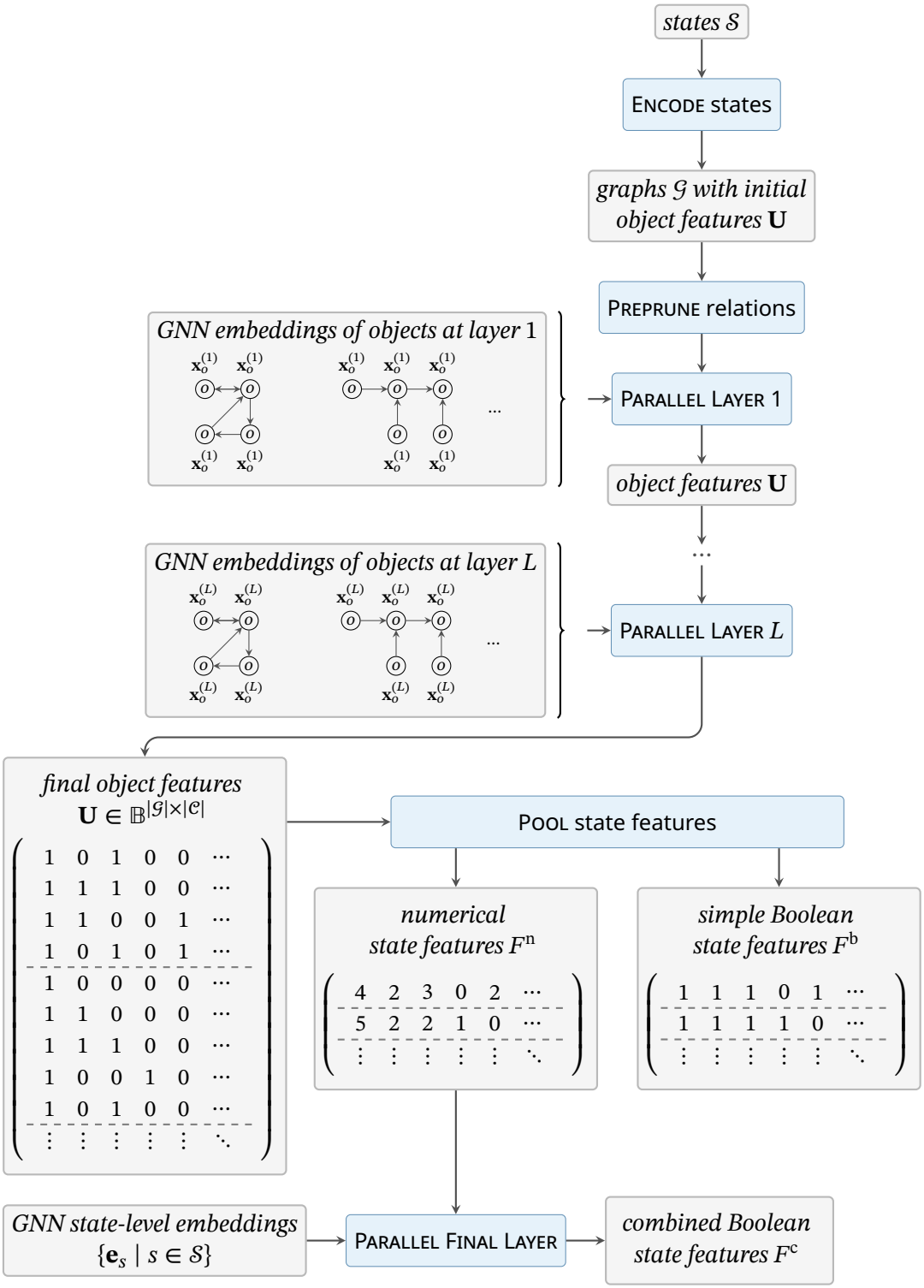


Figure 4.3: Schematic overview of the whole R-IDT architecture to compute state features. PREPRUNE relations denotes the prepruning step as described in Section 4.2, and PARALLEL LAYER is a single parallel layer as shown in Figure 4.2. The PARALLELFINAL LAYER is learned as explained in Section 4.4.2. The displayed feature matrices show illustrative values for two states with four and five objects, respectively.

data, and the regression learning algorithm cannot distinguish them. When the learned regression is applied to data outside the training dataset, where these numerical features take values greater than 1, the learned value function would not generalize well.
- The optimal value function for some problems, for example Q_{clear}, can be written without numerical features, but with a variable number of Boolean features, such as the B_ks (see Equation (2.3)). When an R-IDT is learned, there is no reason for it to learn such Boolean features for any k that is larger than what is required for the training data. In contrast, a numerical feature would be able to scale up to higher numbers of objects. Thus, it may be beneficial to not use (simple) Boolean features in the regression.

One method to fit linear regressions is ordinary least squares (OLS) [10], which chooses weights that minimize the sum of squared errors SSE($\mathbf{w}$), as defined in Equation (2.12). A desirable property of OLS is that it provably computes an optimal solution $\mathbf{w}^*$ with respect to the error on the training data, that is, there is no solution $\mathbf{w}'$ with SSE($\mathbf{w}'$) < SSE($\mathbf{w}^*$).

Minimizing the sum of squared errors is especially suited under the assumption that the observed target has additive normally distributed noise. However, this is not the case for V^*, as it is not a measurement, but an exactly computed value itself. Therefore, it is also considered choosing weights that minimize the sum of absolute errors SAE($\mathbf{w}$) (see Equation (2.11)). Further, this is the loss which the base GNNs were trained to minimize. To learn weights that minimize the sum of absolute errors, SGD can be used.

As $\mathcal{F}$ may contain unnecessary features or features that cancel each other out, we want their learned weights to be (close to) 0. To encourage sparse weights, we also consider the option of constraining the weights to non-negative values only. As many optimal value functions can be written using positive weights only [45], we expect this to not limit the prediction quality. An exception can be made for the bias w_b, as there may be cases where the optimal value function can be written with positive feature weights, but needs a negative offset. Solving for non-negative weights that minimize the sum of squared errors can also be done optimally, see for example [49].

The regression type and options are subject to configuration, and can be set as hyperparameters.

4.6 Hyperparameter Overview

Throughout this chapter, some hyperparameters of R-IDTs and the linear regression have been introduced. This section completes the list by introducing the remaining, not yet mentioned hyperparameters. Table 4.1 provides an overview of all hyperparameters and their value ranges.

The original IDT experiments were performed with the depth of the decision trees bounded by two as to enhance the readability of the resulting formulas [40]. In our experiments, we observe that for the logical distillation of R-GNNs, higher depths are required (see Section 5.5).

Therefore, the depth of all intermediate layers and the last layer are included as hyperparameters.

Further, the IDT's clustering of leaf nodes was implemented using Ward's method (see Equation (2.20)). Recall that the leaf nodes are trained to predict the GNN embeddings, which are generally not interpretable, and there may be better ways to cluster the leaf nodes than Ward's method. Therefore, we also consider other distance measures and linkage strategies that were introduced in Section 2.9.

Table 4.1: The available hyperparameters with their respective value range.

Hyperparameter	Value Range
R-IDT	
Maximal intermediate layer depth	positive integers
Maximal final layer depth	positive integers
Inner and final layer width	positive integers
Inner and final layer sample size	positive integers
Pooling type	max, sum
Clustering distance or similarity	Euclidean, Manhattan, cosine
Clustering linkage	Ward, average, complete, single
Feature types	subsets of {Boolean, numerical, combined}
Regression	
Regression type	OLS (squared error), SGD (absolute error)
Fit intercept	yes, no
Non-negative weights	yes, no

5 Experimental Evaluation

This chapter presents the experimental evaluation of the proposed method. Through our experiments, we seek to answer the following main research questions.

Q1 Can R-IDTs learn the respective optimal value function correctly? Does the learned function generalize to instances that are larger than the ones included in the train dataset?

Q2 What do R-IDTs learn? Are the learned value functions interpretable, that is, can the learned combinations of weights and features be represented in an understandable and readable way?

The experimental evaluation is conducted on five generalized planning problems, which will be introduced shortly in Section 5.1. Afterward, Section 5.2 provides information on the implementation of the method and the experiments. In Section 5.3 and Section 5.4, we detail how R-GNN and R-IDT were trained for the experiments, respectively. Section 5.5 presents the results of the experiments, which are then discussed in Section 5.6.

5.1 Generalized Planning Benchmark Problems

As benchmark problems, we consider five generalized planning problems where each optimal value function can be expressed by features in C_2 [16, 29, 45]. All the problem classes we consider are atomic in the sense that the goal is defined by a single atom. A problem class is called bounded, if for all states, the optimal value function is bounded by a constant independent of the problem instance. We test R-IDTs both on bounded and unbounded problem classes.

In the following, each problem class is introduced. For each, we show how the respective optimal value function can be composed linearly using state features. The respective features are displayed both in C_2 and in $\mathcal{EMLC}$ so that it becomes evident that R-IDTs have in fact the theoretical capability of learning the required features. Afterward, we provide details about the data we used to fit the R-GNN and R-IDT models in Section 5.1.7.

5.1.1 Gripper

In the planning domain *Gripper*, there are rooms in which balls may be located. Further, there is a robot with two grippers, with which it can take and drop balls. The robot can move in between the rooms. The number of both rooms and balls is not fixed and is determined by the problem instances. In the generalized problem class Q_{gripper} we consider, the goal is to transport

a single ball b to a certain room r, that is, the goal is $\texttt{at}(b, r)$. For convenience, we refer to these as the relevant ball and room, respectively. This problem is bounded, as the number of steps under an optimal policy is at most 5 for any state. The optimal value function can be composed as

$$V^*(s) = (\alpha \wedge E)(s) + 4(\alpha \wedge O)(s) + 3(\alpha \wedge P)(s) + 2D(s) + F(s), \tag{5.1}$$

using the following Boolean features:

$$\alpha \coloneqq \exists x, y\, (\texttt{at}_\text{G}(x, y) \wedge \neg \texttt{at}(x, y)) \tag{5.2}$$
$$E \coloneqq \exists x, y\, (\texttt{at}_\text{G}(x, y) \wedge \neg \exists y\, \texttt{free}(y) \wedge \neg \exists y\, \texttt{carry}(x, y)) \tag{5.3}$$
$$O \coloneqq \exists x, y\, (\texttt{at}_\text{G}(x, y) \wedge \exists y\, (\texttt{at}(x, y) \wedge \neg \texttt{at-robby}(y))) \tag{5.4}$$
$$P \coloneqq \exists x, y\, (\texttt{at}_\text{G}(x, y) \wedge \exists y\, (\texttt{at}(x, y) \wedge \texttt{at-robby}(y))) \tag{5.5}$$
$$D \coloneqq \exists x, y\, (\texttt{at}_\text{G}(x, y) \wedge \neg \texttt{at-robby}(y) \wedge \exists y\, \texttt{carry}(x, y)) \tag{5.6}$$
$$F \coloneqq \exists x, y\, (\texttt{at}_\text{G}(x, y) \wedge \texttt{at-robby}(y) \wedge \exists y\, \texttt{carry}(x, y)) \tag{5.7}$$

The feature α is true if the goal is not reached yet, E is true, if a ball needs to be dropped to free a gripper, O is true, if the robot needs to move to the relevant ball's location, P if it is at the relevant ball's location and the ball needs to be picked up, D if the robot has the relevant ball and needs to go to the target location, and F is true if the relevant ball only needs to be dropped. These $\mathcal{EMLC}$ formulas are equivalent to the C_2 formulas, where b and r are helper formulas that are only satisfied by the relevant ball and room, respectively.

$$b \coloneqq \mathbf{A}_{\texttt{at}_\text{G}} \top > 0 \tag{5.8}$$
$$r \coloneqq \mathbf{A}^\text{T}_{\texttt{at}_\text{G}} \top > 0 \tag{5.9}$$
$$\alpha \coloneqq \neg(\mathbf{1}(b \wedge \mathbf{A}_{\texttt{at}} r > 0) > 0) \tag{5.10}$$
$$E \coloneqq \neg(\mathbf{1}\texttt{free} > 0) \wedge \neg\big(\mathbf{1}\big(\mathbf{A}^\text{T}_{\texttt{carry}} b > 0\big) > 0\big) \tag{5.11}$$
$$O \coloneqq \mathbf{1}(b \wedge \mathbf{A}_{\texttt{at}}(\neg\texttt{at-robby}) > 0) > 0 \tag{5.12}$$
$$P \coloneqq \mathbf{1}(b \wedge \mathbf{A}_{\texttt{at}}\texttt{at-robby} > 0) > 0 \tag{5.13}$$
$$D \coloneqq \mathbf{1}\big(b \wedge \neg\big(\mathbf{A}_{\texttt{at}_\text{G}}\texttt{at-robby} > 0\big) \wedge \mathbf{A}^\text{T}_{\texttt{carry}} \top > 0\big) > 0 \tag{5.14}$$
$$F \coloneqq \mathbf{1}\big(b \wedge \mathbf{A}_{\texttt{at}_\text{G}}\texttt{at-robby} > 0 \wedge \mathbf{A}^\text{T}_{\texttt{carry}} \top > 0\big) > 0 \tag{5.15}$$

5.1.2 Miconic

In the *Miconic* domain, there is a multi-story building with an escalator. At each floor, there may be people, each with their own destination floor. The escalator can move freely in between floors, and manages when people enter and leave the escalator. The number of people and floors is defined by the instance. When a person p has reached its goal, the atom $\texttt{served}(p)$ holds. In the class $\mathcal{Q}_\text{miconic}$, the goal is to move a single specified person to its destination, that

is, the goal is an atom $\texttt{served}(p)$. Let this person be called the relevant person. Every problem is bounded by requiring at most 4 steps under an optimal policy. The optimal value function can be written as

$$V^*(s) = 4(\alpha \wedge \neg O \wedge \neg B)(s) + 3(\alpha \wedge O \wedge \neg B)(s) + (\alpha \wedge B)(s) + (\alpha \wedge B \wedge \neg D)(s). \tag{5.16}$$

The Boolean features are defined in C_2 and $\mathcal{EMLC}$, respectively, as shown below.

$$\alpha \coloneqq \exists x\,(\texttt{served}_{\mathrm{G}}(x) \wedge \neg\texttt{served}(x)) \tag{5.17}$$

$$B \coloneqq \exists x\,(\texttt{served}_{\mathrm{G}}(x) \wedge \texttt{boarded}(x)) \tag{5.18}$$

$$D \coloneqq \exists x\,(\texttt{served}_{\mathrm{G}}(x) \wedge \exists y\,(\texttt{destin}(x,y) \wedge \texttt{lift-at}(y))) \tag{5.19}$$

$$O \coloneqq \exists x\,(\texttt{served}_{\mathrm{G}}(x) \wedge \exists y\,(\texttt{origin}(x,y) \wedge \texttt{lift-at}(y))) \tag{5.20}$$

Here, α is true if the relevant person is not served yet, b is true if the relevant person is in the lift, d is true if the lift is at the relevant person's destination, and o if the lift is at the relevant person's destination. Equivalently, these can be written in $\mathcal{EMLC}$ as

$$\alpha \coloneqq \mathbf{1}(\texttt{served}_{\mathrm{G}} \wedge \neg\texttt{served}) > 0 \tag{5.21}$$

$$B \coloneqq \mathbf{1}(\texttt{served}_{\mathrm{G}} \wedge \texttt{boarded}) > 0 \tag{5.22}$$

$$D \coloneqq \mathbf{1}(\texttt{served}_{\mathrm{G}} \wedge \mathbf{A}_{\texttt{destin}}(\texttt{lift-at}) > 0) > 0 \tag{5.23}$$

$$O \coloneqq \mathbf{1}\big(\texttt{served}_{\mathrm{G}} \wedge \mathbf{A}_{\texttt{origin}}(\texttt{lift-at}) > 0\big) > 0. \tag{5.24}$$

5.1.3 Blocks-Clear

This problem class was already introduced earlier in this work as a running example: The Blocksworld domain was introduced in Section 2.2.1, and the problem class $\mathcal{Q}_{\text{clear}}$ of the Blocksworld domain was presented in Section 2.2.2, with its optimal value function and necessary features in C_2 shown by Equations (2.3) to (2.8). This problem class is not bounded, as with more blocks, one can always create states with a higher optimal value. Equivalently, these features presented in C_2 before can be expressed in $\mathcal{EMLC}$ as shown below.

$$\alpha \coloneqq \mathbf{1}(\texttt{clear}_{\mathrm{G}} \wedge \neg\texttt{clear}) > 0 \tag{5.25}$$

$$H \coloneqq \mathbf{1}\texttt{holding} > 0 \tag{5.26}$$

$$B_k \coloneqq \mathbf{1}(\texttt{clear}_{\mathrm{G}} \wedge \eta_k) > 0 \tag{5.27}$$

$$\eta_0 \coloneqq \texttt{clear} \tag{5.28}$$

$$\eta_k \coloneqq \mathbf{1}\big(\mathbf{A}^{\mathrm{T}}_{\mathrm{on}}\eta_{k-1} > 0\big) > 0 \tag{5.29}$$

5.1.4 Blocks-On

This problem class is also set in the Blocksworld domain. The goal is to stack two specified blocks x, y onto each other, that is, to achieve a state where $\mathsf{on}(x, y)$ holds. For convenience, let these blocks be called the first and second goal block, respectively. Like $\mathcal{Q}_{\text{clear}}$, the problem class $\mathcal{Q}_{\text{on}}$ is not bounded, as adding more blocks can increase the length of optimal plans indefinitely. The value function and features presented below are adapted from Ståhlberg et al. [45]. For any number of blocks $n \geq 1$, they can be written as

$$V^*(s) = (\alpha \wedge H)(s) + 2(\alpha \wedge L)(s) + 2 \sum_{k=1}^{n-1} kX_k(s) + kY_k(s), \tag{5.30}$$

where the features are defined by the following formulas, using the same η and H as $\mathcal{Q}_{\text{clear}}$.

$$\alpha \coloneqq \exists x, y\, (\mathsf{on}_{\mathrm{G}}(x, y) \wedge \neg \mathsf{on}(x, y)) \tag{5.31}$$

$$L \coloneqq \exists x, y\, (\mathsf{on}_{\mathrm{G}}(x, y) \wedge (\neg \mathtt{clear}(y) \vee \neg \mathtt{holding}(x))) \tag{5.32}$$

$$X_k \coloneqq \exists x, y \left(\mathsf{on}_{\mathrm{G}}(x, y) \wedge \eta_k(x) \wedge \neg \bigvee_{i=1}^{n-1} F_i(x) \right) \tag{5.33}$$

$$F_1(x) \coloneqq \exists y\, (\mathsf{on}(x, y) \wedge \exists x\, \mathsf{on}_{\mathrm{G}}(x, y)) \tag{5.34}$$

$$F_i(x) \coloneqq \exists y\, (\mathsf{on}(x, y) \wedge F_{i-1}(y)) \tag{5.35}$$

$$Y_k \coloneqq \exists x, y \left(\mathsf{on}_{\mathrm{G}}(x, y) \wedge \eta_k(y) \wedge \neg \bigvee_{i=1}^{n-1} S_i(y) \right) \tag{5.36}$$

$$S_1(x) \coloneqq \exists y\, (\mathsf{on}(x, y) \wedge \exists x\, \mathsf{on}_{\mathrm{G}}(y, x)) \tag{5.37}$$

$$S_i(x) \coloneqq \exists y\, (\mathsf{on}(x, y) \wedge S_{i-1}(y)) \tag{5.38}$$

Again, α is true if the goal is not reached yet, and L if the first goal block cannot be stacked onto the second goal block next. For all $k \leq n - 1$, X_k is true if there are k blocks above the first goal block, and it is not above the second goal block. This is expressed using the formulas F_i, which are satisfied by a block if there is a path of length i from the block to the second goal block. Combining them to $\neg \bigvee_{i=1}^{n-1} F_i(x)$ expresses that there is no such path of any length. For all $k \leq n - 1$, Y_k is similar to X_k, but with the roles of the first and second goal block switched, that is, it is true if there are k blocks above the second goal block, and it is not above the first goal block. Again, helper formulas S_i similar to F_i are used. The following $\mathcal{EMLC}$ formulas are equivalent, with the addition of G_f and G_s, which are satisfied only by the first and second

goal block, respectively.

$$G_f \coloneqq \mathbf{A}^{\mathrm{T}}_{\mathsf{on}_\mathrm{G}} \top > 0 \tag{5.39}$$

$$G_s \coloneqq \mathbf{A}_{\mathsf{on}_\mathrm{G}} \top > 0 \tag{5.40}$$

$$\alpha \coloneqq \mathbf{1}\big(G_f \wedge \neg(\mathbf{A}_{\mathsf{on}} G_s > 0)\big) > 0 \tag{5.41}$$

$$L \coloneqq \neg\big((\mathbf{1}(G_s \wedge \mathtt{clear}) > 0) \wedge \big(\mathbf{1}\big(G_f \wedge \mathtt{holding}\big) > 0\big)\big) \tag{5.42}$$

$$X_k \coloneqq \mathbf{1}\left(G_f \wedge \eta_k \wedge \neg \bigvee_{i=1}^{n-1} F_i\right) > 0 \tag{5.43}$$

$$F_1 \coloneqq \mathbf{A}_{\mathsf{on}} G_s > 0 \tag{5.44}$$

$$F_i \coloneqq \mathbf{A}_{\mathsf{on}} F_{i-1} > 0 \tag{5.45}$$

$$Y_k \coloneqq \mathbf{1}\left(G_s \wedge \eta_k \wedge \neg \bigvee_{i=1}^{n-1} S_i\right) > 0 \tag{5.46}$$

$$S_1 \coloneqq \mathbf{A}_{\mathsf{on}} G_f > 0 \tag{5.47}$$

$$S_i \coloneqq \mathbf{A}_{\mathsf{on}} S_{i-1} > 0 \tag{5.48}$$

5.1.5 Visitall

In the *Visitall* domain, the environment is a graph of connected nodes. Here, the setting is simplified to a discrete square grid of $n = c \times c$ cells, where each cell is connected only to its direct neighbors. In this domain, the agent's usual goal is to visit all cells. Here, we only consider the simpler goal of visiting a single specified goal cell, that is, $\mathtt{visited}(x)$. Because of the atomic goal, the problem effectively reduces to a simple grid navigation task. As the grid size n is unbounded, so is the optimal value function

$$V^*(s) = \sum_{k=1}^{2(c-1)} k(\alpha \wedge S_k)(s), \tag{5.49}$$

adapted from [45]. It uses the following features and helper formulas.

$$\alpha \coloneqq \exists x\, (\mathtt{visited}_\mathrm{G}(x) \wedge \neg \mathtt{visited}(x)) \tag{5.50}$$

$$S_k \coloneqq \exists x \left(\mathtt{at\text{-}robot}(x) \wedge P_k(x) \wedge \neg \bigvee_{i=1}^{k-1} P_i(x)\right) \tag{5.51}$$

$$P_1(x) \coloneqq \exists y\, (\mathtt{connected}(x, y) \wedge \mathtt{visited}_\mathrm{G}(y)) \tag{5.52}$$

$$P_k(x) \coloneqq \exists y\, (\mathtt{connected}(x, y) \wedge P_{k-1}(y)) \tag{5.53}$$

Here, α is true if the goal is not reached yet, and for every distance $k \geq 1$, S_k is true if the shortest path to the goal cell has length k. Each object feature P_k for $k \geq 1$ is satisfied by a cell if a path of length k from that cell to the goal cell exists. Corresponding $\mathcal{EMLC}$ formulas are

displayed below.

$$\alpha \coloneqq \mathbf{1}(\texttt{visited}_{\mathrm{G}} \wedge \neg \texttt{visited}) > 0 \tag{5.54}$$

$$S_k \coloneqq \mathbf{1}\left(\texttt{at-robot} \wedge P_k \wedge \neg \bigvee_{i=1}^{k-1} P_i\right) \tag{5.55}$$

$$P_1 \coloneqq \mathbf{A}_{\texttt{connected}}\texttt{visited}_{\mathrm{G}} > 0 \tag{5.56}$$

$$P_k \coloneqq \mathbf{A}_{\texttt{connected}} P_{k-1} > 0 \tag{5.57}$$

5.1.6 Concluding Remarks

For all the considered problems, the respective optimal value functions were described using Boolean features only. In case of the unbounded problems, that is, Blocks-Clear, Blocks-On, and Visitall, the number of required features is not fixed, but depends on the number of objects n. The reason for this is that the features B_k, X_k, Y_k, and S_k need to be defined for each k, and that the highest required value of k is dependent on n. Introducing numerical features can mitigate the need for a variable number of features. Consider Q_{clear}, where one can define the object feature A to be satisfied by all blocks that are above the block to be cleared. Let n_A and p_A denote the numerical and Boolean state feature of A, respectively. Then, the optimal value function can be expressed without B_k by only four features as

$$V^*(s) = (\alpha \wedge H)(s) + 2n_A(s) - p_A(s). \tag{5.58}$$

However, for n_A to scale up to any number of blocks, A would need to be defined on the *transitive closure* of the On relation, which is defined as

$$\mathrm{On}^+(x,y) \equiv \exists m \geq 1\ \mathrm{On}^m(x,y), \text{where} \tag{5.59}$$

$$\mathrm{On}^1(x,y) \equiv \mathrm{On}(x,y), \text{and} \tag{5.60}$$

$$\mathrm{On}^{i+1}(x,y) \equiv \exists z \left(\mathrm{On}(x,z) \wedge \mathrm{On}^i(z,y)\right). \tag{5.61}$$

Then, one could define $A(x) \coloneqq \exists y\,(\texttt{clear}_{\mathrm{G}}(y) \wedge \texttt{on}^+(x,y))$. As the R-GNNs do not have access to transitive closures, but can only compute one ”step” along a relation in each layer, their ability to learn such closures is limited by the number of layers L. The same applies to R-IDTs, as they can only learn to nest one more modal at each layer, e.g., $\mathbf{A}_{\texttt{on}}$. In this regard, these approaches differ from previous symbolic approaches [11, 12, 19, 20, 36], which make transitive closures explicitly available when creating formulas.

5.1.7 States Data

The data we use, that is, planning states labeled with their optimal value, is taken from Ståhlberg et al. [45]. It is available online[1] along with the PDDL files of the domains and instances. The states data was generated by performing a random walk of a given length on each initial state of the problem instances. From each state which was visited, an optimal plan was computed using the A* algorithm with the $h_{\max}$ heuristic [13]. Each state s that was visited when following the plan was labeled by the remaining plan length from s and added as data.

Following best practices, the data is split up into train, validation, and test data [23]. Similar to previous work [45], we train the models explicitly for generalization, that is, the train data consists of states with a lower number of objects than the validation data, and vice versa for the validation and test data. Further, if the problem class is unbounded, the maximal observed value of V^* also grows with the number of objects. For every problem, the R-GNNs and R-IDTs are trained on the same data splits to allow for direct comparability. Table 5.1 shows the number of objects and the maximal value of the optimal value function of the used states data.

Table 5.1: For each problem class and data split, the number of states (#S), the number of objects per instance (#O), and the maximal optimal value of all states (max V^*) are shown.

Problem	**Train Data**			**Validation Data**			**Test Data**		
	#S	#O	max V^*	#S	#O	max V^*	#S	#O	max V^*
Gripper	46 714	8 – 16	5	37 655	18 – 20	5	200 292	22 – 46	5
Miconic	42 599	3 – 24	4	30 730	27 – 33	4	103 834	36 – 90	4
Blocks-Clear	108 503	2 – 9	15	47 541	10 – 11	15	40 930	12 – 17	21
Blocks-On	122 464	2 – 9	18	41 176	10 – 18	22	22 290	12 – 17	20
Visitall	62 488	25 – 49	14	41 737	64	18	88 350	81 – 100	23

5.2 Implementation Details

We implemented our experiments in Python and make use of several libraries and frameworks. As states data, we use that provided by Ståhlberg et al. [45] and read it using their parser. To work with and create graph data from states data, we use the library NetworkX [25]. To convert graphs to inputs for the R-GNNs as well as working with graph datasets, we leverage PyTorch Geometric [18]. As basis for our R-GNN implementation, we rely on Plangolin [4], which implements the network architecture using PyTorch [5] and PyTorch Geometric. To simplify model training, we make use of the PyTorch Lightning library [17]. The implementations of fitting decision trees, hierarchical agglomerative clustering, ordinary least squares as well as non-negative least squares are provided by Scikit-learn [39]. Various mathematical operations

[1] `https://doi.org/10.5281/zenodo.6353140` (visited on March 3, 2026)

are performed with NumPy [26]. Plots are created with Matplotlib [32] and seaborn [51]. To log and organize experiments for GNNs and R-IDTs, Weights & Biases [9] was used.

The datasets of encoded states for R-GNNs and R-IDTs are created only once, and then saved so that they can be reused. Also, when an R-IDT is learned based on an R-GNN, the object embeddings $\mathbf{x}$ are only computed once and then saved. This allows training many R-IDTs for the same GNN without repeatedly requiring GPUs for a faster computation of the embeddings.

5.3 Training of R-GNN Models

Following the approach of Ståhlberg et al. [45], the R-GNN models are trained using supervised learning, where data is given as pairs $\langle G(s), V^*(s) \rangle$ of state graphs and each state's optimal values. For technical reasons, the state encoding method for the GNN input $G(\cdot)$ provided by Aichmüller and Krude [4] differs from ours (see Section 4.1), but not in a substantial way. The states are sampled randomly, where the probability of each state s being sampled is proportional to the inverse frequency of its value, as defined in Equation (2.19). The loss is measured by the absolute error (see Equation (2.11)). As training optimizer, Adam [33] is used with a learning rate of 10^{-3}, and a weight decay weighting of $5 \cdot 10^{-4}$. The batch size is set to 64 samples, and each epoch, 300 minibatches are sampled. Each GNN is trained for up to 22 hours or 1000 epochs. However, early stopping is employed to stop the training process if the loss on the validation set has not improved for 20 epochs.

For each of the selected problems (see Section 5.1), multiple R-GNNs are trained. As choices for the aggregation function, we test summation, taking the maximum (Equation (2.13)), and a taking a smooth maximum (Equation (2.14)). For the pooling function, summation and taking the maximum are tested. The choice for the number of layers L is dictated by the lowest number such that for all graphs in the train, validation, or test set, a message by any object can reach any other object over the state graph's edges (in case the graph is connected) regardless of the respective relation. For example, in the Blocksworld domain with instances that have up to ten blocks, the maximal distance a message may travel is nine objects, and thus, at least $L \geq 9$ is required. It is in our interest to keep L as small as possible, as the time and memory required to fit an R-IDT is highly dependent on the number of layers. The embedding size k is mostly fixed to 32, as experiments by Ståhlberg et al. [45] indicate that this is sufficient for the considered problems, and we obtain satisfying results with this option.

After the training of a model has completed, it is evaluated on each data split. As models to train R-IDTs on, we consider those that have successfully learned to predict the optimal values on the training set. Among all such models, we prefer those that show good generalization on the test data.

5.4 Training of R-IDT Models

From an R-GNN, many R-IDTs are distilled for our experiments using different hyperparameter configurations (see Section 4.6). Further, once an R-IDT is learned, multiple regressions using different hyperparameters can be learned on the features. For convenience, in the following we often refer to an R-IDT with a regression simply as an R-IDT. The reason for this is that for predicting the value function, such pairs are always required.

Additionally to the hyperparameter configurations, the R-IDTs are also given a seed to initialize the random components, which are twofold: One is the state graph sampling step for each inner and final layer (see Sections 3.1.2 and 4.4.2). The second is how each decision tree selects splits. There may be multiple splits that yield different partitions with the same weighted heterogeneity measure q_m (Equation (2.16)), but only one can be chosen. In the implementation used in our experiments [39], the split is selected randomly. For each selected hyperparameter configuration, four R-IDTs with different seeds are learned.

Recall that when fitting decision trees, it is common to weight samples to counter unbalanced datasets, for example by using the inverse frequency of the target value as defined in Equation (2.19). The decision tree of an R-IDT's inner layer works do not work on states, but on objects from a number of states. Each object is weighted dependent on the state it belongs to, that is, if a state has the weight $w_{\langle s,V^*(s)\rangle}$ dependent on $V^*(s)$, every object of s gets the state's weight.

Similar to how the R-GNNs are evaluated, each R-IDT is evaluated on each data split of the respective problem. Note that the respective validation sets are not used in any way during R-IDT model training, as the models are not fitted and validated incrementally like neural networks, but fitted in one consecutive procedure.

5.5 Experimental Results

This section presents the results of applying R-IDTs to learn general optimal value functions guided by R-GNNs, and provides answers to the overarching research questions posed at the beginning of Chapter 5.

Question **Q1**, that is, whether R-IDTs learn the value function well and whether they generalize, can be answered in a quantitative way. We measured the mean absolute error (MAE) of the trained models, and visualized the respective predictions. As the respective validation and test datasets contain planning states with more objects than the training dataset, and in the unbounded cases, also higher optimal values, they were used to testing for generalization.

Question **Q2**, that is, whether the resulting models are interpretable, and what they learn, can only be answered qualitatively. To answer it, it was required to extract the learned weights and

features and try to bring them into a readable format, similar to the readable value functions presented shortly in Section 5.1.

In the following, the results are presented grouped by problem class. The R-IDT models that are presented in the respective sections are chosen as example models to elucidate our findings. To visualize the findings, we often use box plots, which show the *distributions* of the predicted optimal state values grouped by the true optimal state values. This is analogous to confusion matrices for classification models. Boxes represent the range between the first and third quartiles, and the median is shown by a horizontal line within the boxes. Outlier values are marked by circles. When only a horizontal line is visible, it means that all values are extremely close to each other and thus, the distribution collapses to a single point.

5.5.1 Gripper

As basis for Gripper, we take an R-GNN that has 5 layers, an embedding size of 16, uses logsumexp for aggregation, and summation for pooling. The 5 layers are sufficient so that a message from any object in a state graph can reach any other object over the relational edges. From this model, we distill many R-IDTs, using different hyperparameter configurations (see Section 4.6).

Two of the distilled R-IDT models are presented as an example, called ”good” and ”bad” in the following. Table 5.2 displays the error values of the three R-GNN and R-IDT models. For a visual representation of the models' predictions, see Figure 5.2. The good R-IDT learns to predict the optimal value function *perfectly* for all data splits. On the other hand, the bad R-IDT does not perform well on any data split. In fact, both of these R-IDTs used *the same hyperparameter configuration*, and their configurations differed only in their random seed. As seen during the experiments' analysis, this is not an isolated incident, but the random seed has a high influence. This indicates that R-IDTs are highly susceptible to changes in the train data distribution, and that multiple models should be fitted before drawing conclusions.

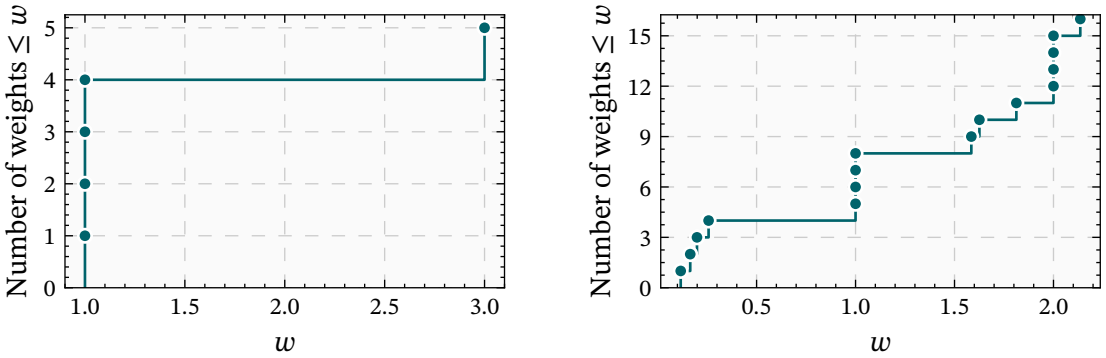
In the predictions' visualization, it is clearly visible that the good R-IDT generalizes perfectly, while the R-GNN seems to become less accurate when more irrelevant objects, that is, balls, are present. The bad R-IDT seemingly hardly learns anything about the problem. While it predicts the values one and two perfectly, it fails to correctly predict any other values.

For Gripper, there are many distilled models that are perfect on the train set. We find that every such model is also perfect on the validation and test sets, that is, the models do not seem to overfit in any way, and always generalize well.

The substantial impact of the random seed makes it hard to determine which hyperparameters are good, and which are not suitable to learn the required features and weights. Indeed, as we will see in the experiments for the other problems, this is not an isolated incident, but happens regularly. However, a conclusion that can be drawn from extensive testing is that the lowest

Table 5.2: Comparison of the MAE of the three models on the three data splits. The zero-error R-IDT performs best.

Model	Train Data	Validation Data	Test Data
Base R-GNN	0.0073	0.0069	0.0754
Good R-IDT	**0.0000**	**0.0000**	**0.0000**
Bad R-IDT	1.4020	1.3297	1.2855



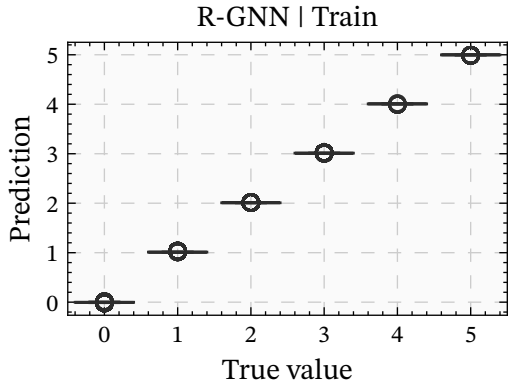
(a) The five non-zero weights of the good R-IDT.

(b) The 16 non-zero weights of the bad R-IDT.

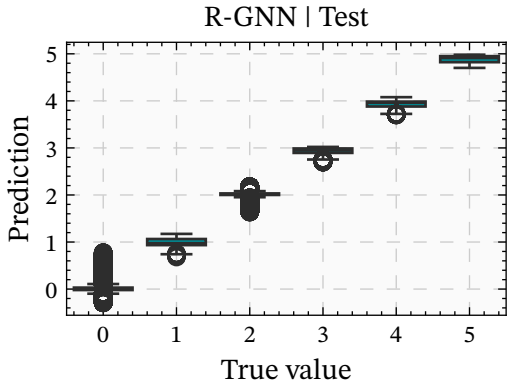
Figure 5.1: The distributions of weights learned by the good R-IDT (left) and the bad R-IDT (right).

values for the hyperparameters ”maximal intermediate layer depth” and ”maximal final layer depth” such that perfect R-IDTs can be learned to be four and three, respectively. This contrasts the insights from Section 5.1.5, which shows that there are features that could be learned by decision trees of lower depth, but can still express the optimal value function. When R-IDTs were fitted with higher layer depths, comparably more models that are perfect emerged. This can likely be attributed to the fact that with higher depths, more features are learned and thus, it becomes more likely that good features are learned. Also, recall that the original IDT method successfully learned logical classifiers with the depth bounded by two (see Section 3.1.2). It is unclear why our setting requires so much higher depths, especially when considering that the number of features that are generated by a tree grows exponentially with the depth.

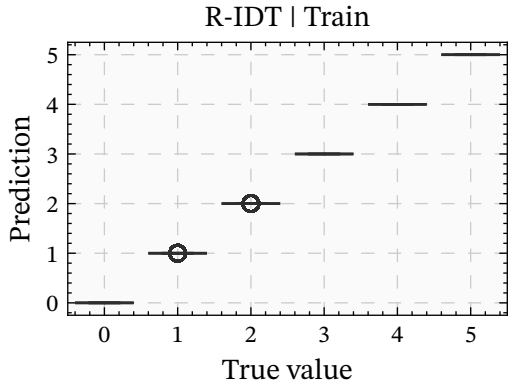
For Gripper, perfect models were learned with widths of $w = 5, 10$, and sample sizes for each layer $s = 10\,000, 20\,000$. The type of pooling by which the R-IDTs derive state features does not show any impact, meaning that the exact number of objects having a certain feature is not important, but only whether there are any objects having the feature. This fits well to the observation, that all the optimal value function can be written without counting quantifiers, or modal parameter comparisons with thresholds larger than zero (see Section 5.1.6). Further, the cosine clustering similarity with either complete, single, or average linking resulted in the best R-IDTs. There are also perfect models that use the Manhattan distance and single linkage. Additionally, the regression worked best using OLS with non-negative weights, and



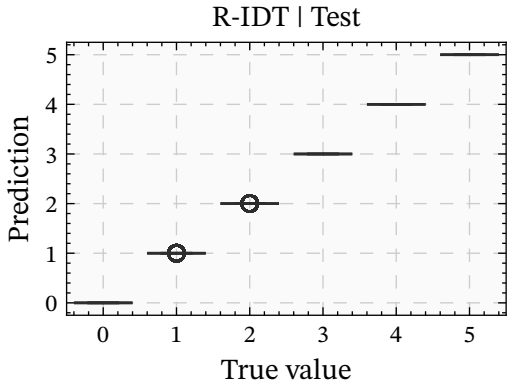
(a) The base R-GNN on train data.



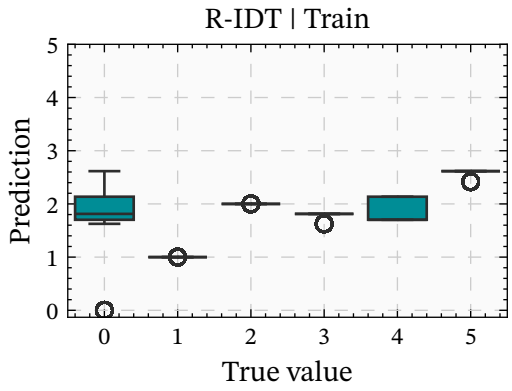
(b) The base R-GNN on test data.



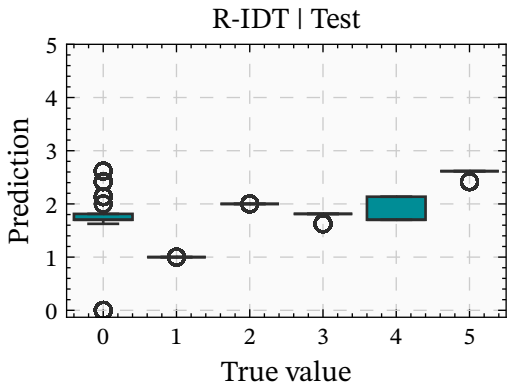
(c) The good R-IDT on train data.



(d) The good R-IDT on test data.



(e) The bad R-IDT on train data.



(f) The bad R-IDT on test data.

Figure 5.2: Box plots of the optimal state values for Gripper as predicted by the trained models, grouped by the true value. The predictions are shown for the training (left) and test (right) data sets. The top row shows the base R-GNN, the middle shows the good R-IDT, and the bottom row shows the bad R-IDT.

only requires combined state features to learn perfect models. A single final R-IDT layer proved to be sufficient.

Analyzing the R-IDTs' formulas turns out to be challenging, as they are too complex and too deeply nested to be understandable. However, the weights learned by the regression, shown in Figure 5.1, can be analyzed to some degree. In total, the good R-IDT provided 184 state features to the regression. However, only five of those are actually used by the regression, that is, they have a non-zero weight. These five features are all combined Boolean features, and the regression has learned integer weights only. Their weights are 1, 1, 1, 1, and 3, respectively. This is a strong indication that the learned features could indeed be expressed by simpler, readable formulas. In contrast, the bad R-IDT regression uses 16 features, with their weights being less meaningful.

5.5.2 Miconic

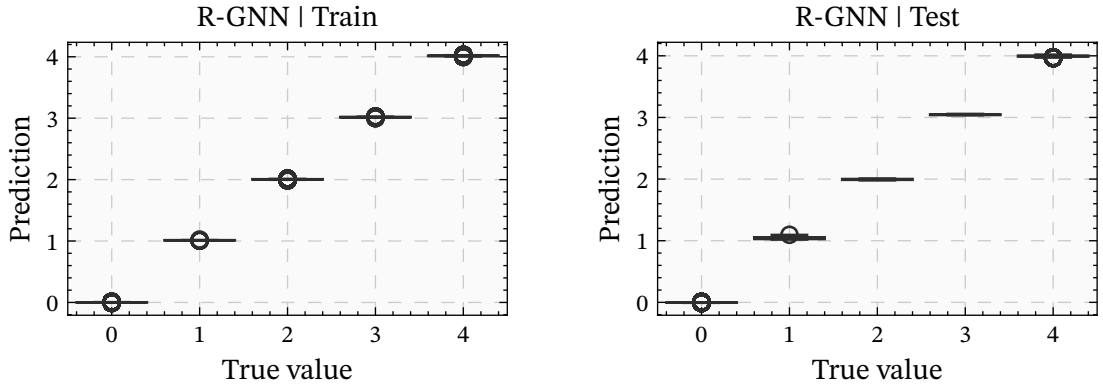
The R-GNNs for Miconic were created with $L = 5$ layers and an embedding size of $k = 32$. As for Gripper, this number of layers is sufficient to operate on any state graph of the problem class. For this problem, we present two R-GNNs, for reasons that will become clear later. Both models use max pooling. Let the first model be called model A. As aggregation function, it uses logsumexp. The other model, which we will call model B, uses max aggregation. Both R-GNN models show good error values, as displayed in Table 5.3. Figure 5.3 provides a visualization of the predicted values. It is evident that both models have learned the optimal value function very well.

Table 5.3: Comparison of the MAE of the four models on the data splits. The zero-error R-IDT A performed best on all splits, and R-IDT B outperformed both R-GNNs.

Model	Train Data	Validation Data	Test Data
R-GNN A	0.0045	0.0031	0.0063
R-GNN B	0.0024	0.0018	0.0043
R-IDT A	**0.0000**	**0.0000**	**0.0000**
R-IDT B	**0.0000**	0.0006	0.0053

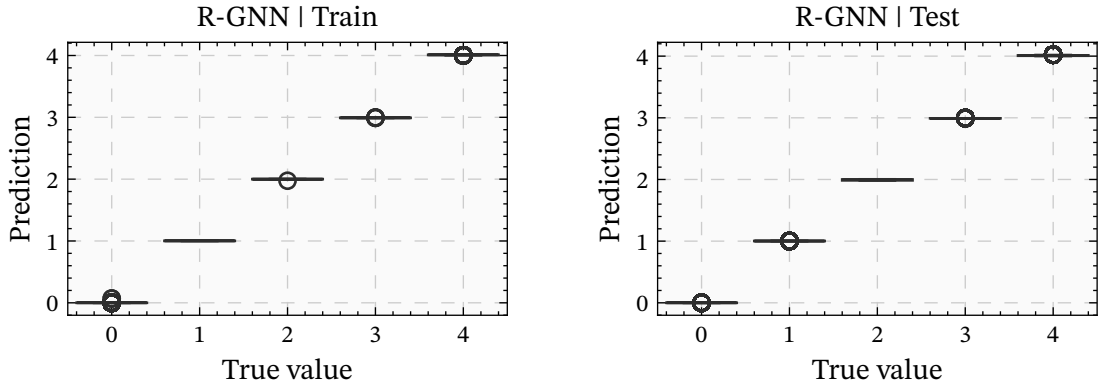
When learning IDTs on the models, an interesting phenomenon can be observed. On model A, multiple R-IDT that are *perfect* on all data splits were learned. On the other hand, we could not learn a single R-IDT that generalizes perfectly on model B. However, on model B, many R-IDTs that are perfect only on the training set were learned. Specifically, *all* of these R-IDTs struggle to predict the value three correctly, and confuse it with the value four. In the Miconic problem class (see Section 5.1.2), states with an optimal value of four are those, where the elevator must still go to the origin floor of the relevant person, while in a state with value three, the lift is at the origin floor, and must now pick up the person. Distinguishing such states seems to pose a challenge to R-IDTs distilled from model B, but it is not clear why.

For illustration, let us pick two good R-IDTs A and B which are learned on model A and B, respectively. Figure 5.4 displays the predictions of both R-IDTs, and their error values are



(a) R-GNN A on train data.

(b) R-GNN A on test data.



(c) R-GNN B on train data.

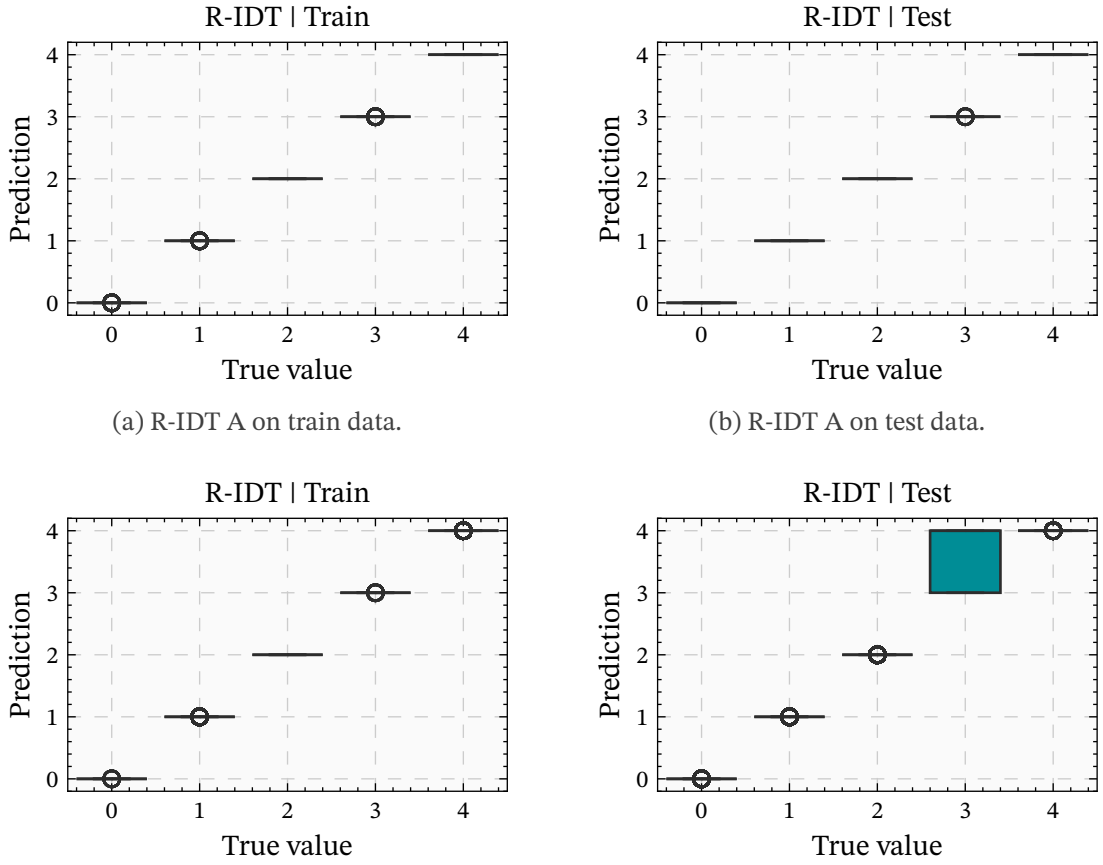
(d) R-GNN B on test data.

Figure 5.3: Box plots of the predictions for Miconic by the trained R-GNNs. The predictions are shown for the training (left) and test (right) data sets. The top row shows R-GNN A, and the bottom row shows R-GNN B.

reported in Table 5.3. Interestingly, although R-IDT B has lower error values than R-GNN A on every data split, R-GNN A is better at distinguishing the states. From the prediction visualization, it is evident that the policy that is greedy in the function of the GNN is optimal (on all tested states). In contrast, the greedy policy defined on the function of the R-IDT may not be optimal, as states with values three and four may be confused.

As for Gripper, the R-IDTs require high layer depths. The experiments show that the lowest value for the depth of the inner R-IDT layers and for the final state level layer such that perfect models can be learned is five. Unfortunately, the learned features are again not understandable due to their complexity. Similar to Gripper, the perfect models learned to assign non-zero weights to combined features only. For instance, the weights of both R-IDTs are displayed in Figure 5.5. For R-IDT A, we can write the learned value function in the form

$$V^*(s) = F_1(s) + F_2(s) + 3F_3(s) + 3F_4(s) + 3F_5(s) + 4F_6(s) \tag{5.62}$$



(a) R-IDT A on train data.

(b) R-IDT A on test data.

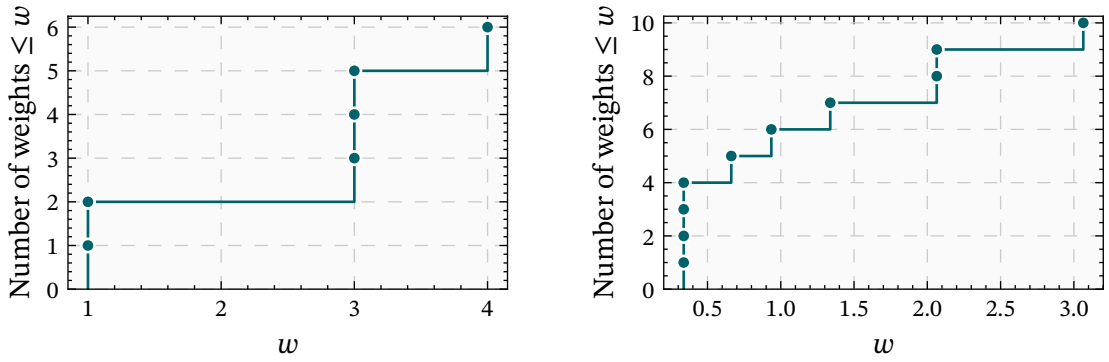
(c) R-IDT B on train data.

(d) R-IDT B on test data.

Figure 5.4: Box plots of the predictions by the R-IDTs A and B for Miconic. The predictions are shown for the training (left) and test (right) data sets. The top row shows R-IDT A, and the bottom row shows R-IDT B.

for combined state features F_i. Further, we can deduce that in all states of value one, always exactly one of the features F_1, F_2 must evaluate to 1. For all states with value 2, both F_1 and F_2 must evaluate to one, as there is no feature with weight 2. In states with a value of 3, exactly one of the features F_3, F_4, F_5 is true, and in states with value 4, it is most likely that only F_6 is true. Analyzing what R-IDT B has learned is harder, as it has learned ten noninteger weights.

Further insights about the influence of the hyperparameters are limited, but one can conclude that again, the pooling type does not matter. The models were configured to use widths of $w = 10, 20$, and an inner layer sample size of $s = 10\,000$. Here, any clustering configuration that used the Euclidean distance or the cosine similarity, together with any linkage strategy, worked best. Again, OLS with non-negative weights yielded the best regressions. Similar to Gripper, a single final R-IDT layer was sufficient to learn good models. While there are no examples where the random seed has an impact comparable to the presented models from Gripper, it is noteworthy that some seeds consistently result in worse models than other seeds.



(a) The six non-zero weights of R-IDT A.

(b) The ten non-zero weights of R-IDT B.

Figure 5.5: The distributions of weights learned on Miconic by the R-IDTs A (left) and B (right).

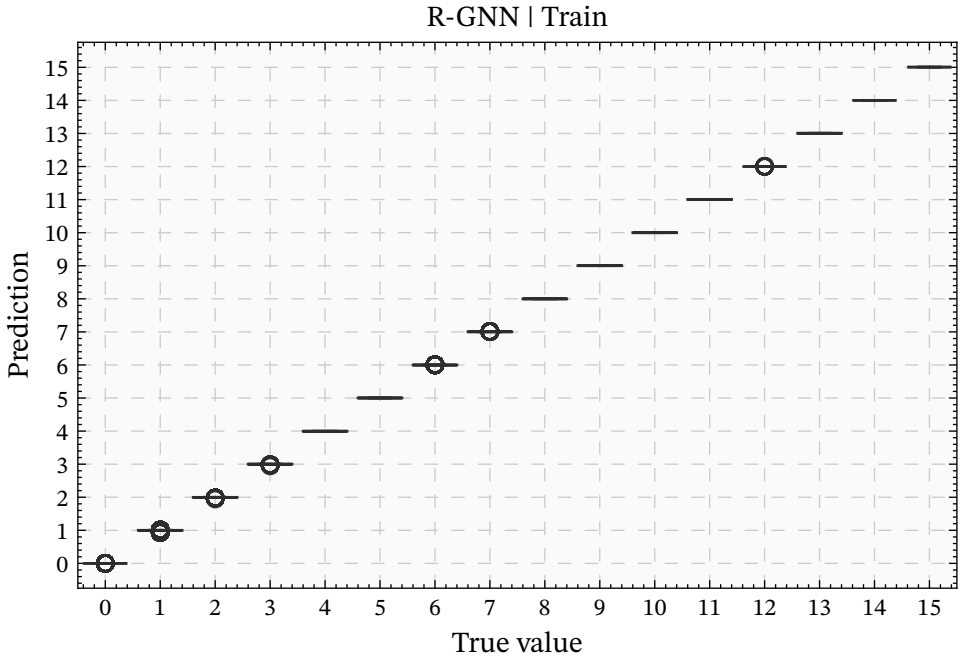
5.5.3 Blocks-Clear

The problem class Blocks-Clear is the first unbounded class for which results are presented. The R-GNNs we consider for Blocks-Clear have $L = 17$ layers and an embedding size of $k = 32$. This number of layers is enough for our instances, as they have up to 17 objects (see Table 5.1), and in any Blocksworld state graph with n blocks, the distance between any two connected blocks is at most $n - 1$. One model we train uses logsumexp aggregation and pools by summation. Let this model be called the base model from now on, as the presented R-IDTs were learned on this model. The error values of the R-GNN and the R-IDTs that will be introduced are shown in Table 5.4. From the visualization in Figure 5.6, it becomes clear that it has learned the optimal value function well for the train set. However, when blocks are added that do not influence a state's value, it predicts some wrong values, which seem to be still well-ordered. As these wrong predictions are rare outliers, they do not have a large negative impact on the model's error value. Interestingly, it does generalize very well for all states with a value of at least 6.

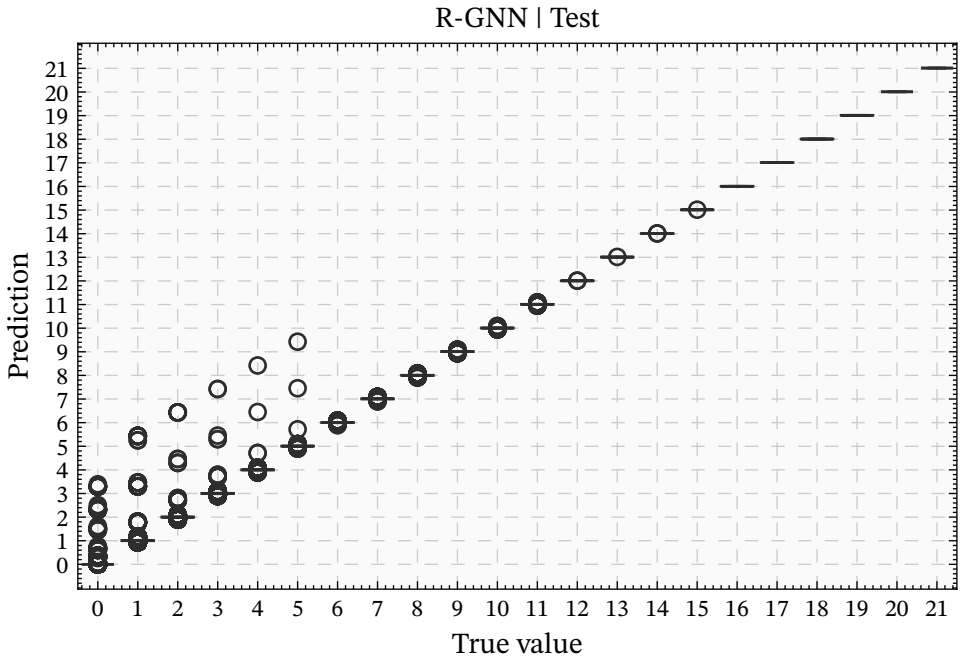
Table 5.4: Comparison of the MAE values of the three models on the data splits for Blocks-Clear.

Model	Train Data	Validation Data	Test Data
Base R-GNN	0.0020	**0.0029**	**0.0170**
R-IDT A	**0.0000**	0.0344	0.1754
R-IDT B	**0.0000**	0.0213	0.1312

Learning good R-IDTs for Blocks-Clear turned out to be difficult. We find that R-IDTs that fit the training data very well can be learned when using numerical and combined features and allowing non-negative feature weights only, but an unbounded bias. In the following, let R-IDT A be such a model. On the training data, it predicts the true values perfectly, as seen in Table 5.4 and Figure 5.7. Its prediction on the validation (not shown here) and test sets show that interestingly, high values are predicted more accurately than low values, although states



(a) Predictions of the base R-GNN on the test set.



(b) Predictions of the base R-GNN on the test set.

Figure 5.6: Predictions of the base R-GNN for Blocks-Clear on the train set (first figure) and test set (second figure). Note the different axis scales.

with such low values are more present in the train data. In this regard, the R-IDT is similar to the base R-GNN. It seems that for the lower values, it overfitted to instances with fewer blocks of the train data, while for higher values, it has indeed learned to scale with the number of blocks. Unfortunately, the learned features and weights do not allow any insights into how the R-IDT does so. Again, the features are too complex to understand, and the weights are not sparse and not integer, as shown in Figure 5.9. Specifically, this R-IDT uses 210 weights, and learns the value -7.1126 for the bias. It is not clear, what has been learned.

On the other hand, let us consider an R-IDT that can use numerical and combined features with non-negative weights only and no bias. Further, it uses ten final layers in parallel. R-IDT B is such a model, with its MAE values displayed in Table 5.4. Like R-IDT A, it fits the training data perfectly. On the test data, it shows a similar behavior, that is, higher values are predicted better than low values. However, the weights and features learned by these two models differ significantly. Figure 5.9 provides a visualization. There are only eleven non-zero weights. The ten features with a weight of 1 are all combined Boolean features, and the feature with weight 2 is a numerical feature. This numerical feature allows the model's predictions to scale up to high values. However, it seems that it has partially overfitted to the train states, as many predictions for low values are outliers. For this problem class, we observe that using parallel final layers helps to learn sparse weights, as this model shows exemplary.

Similar to the experiments on other problems, we observe that the type of the R-IDTs' pooling is not important when deriving combined features. Further, regressions learned with OLS generally were better than regressions with SGD. The models were created with layer depths of 5 for both the inner and final layers. The values for the width was set to $w = 5$, and the number of sample state graphs each inner layer was fitted on is $s = 40\,000$.

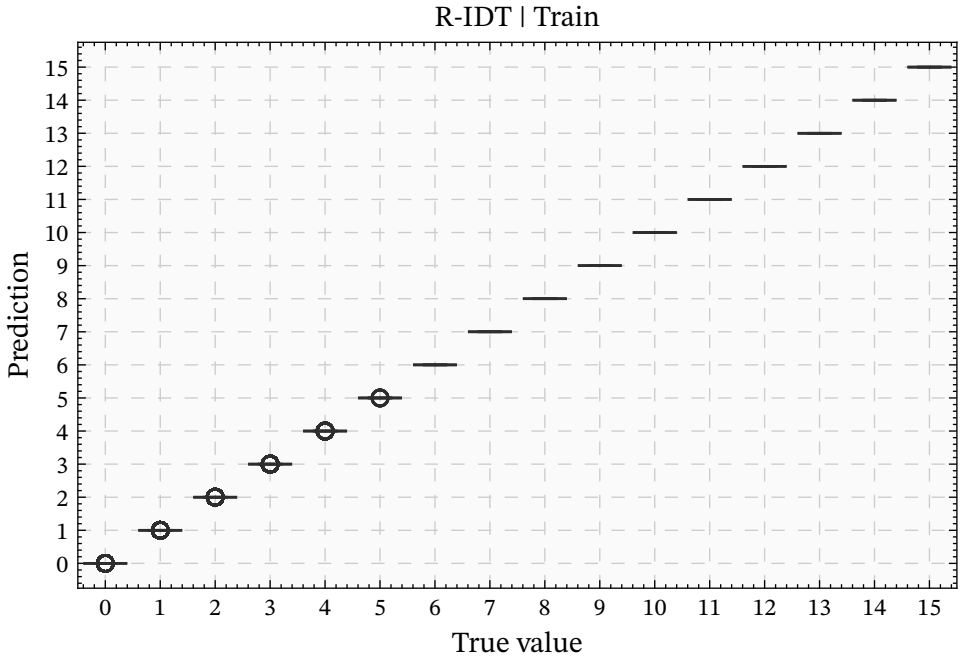
5.5.4 Blocks-On

Similar to Blocks-Clear, this problem class is unbounded, and the number of objects in the used states is at most 17. Therefore, the considered R-GNNs were created with $L = 17$ layers and used an embedding size of $k = 32$. The base model detailed in the following uses max aggregation and pools by summation. Its error values are depicted in Table 5.5.

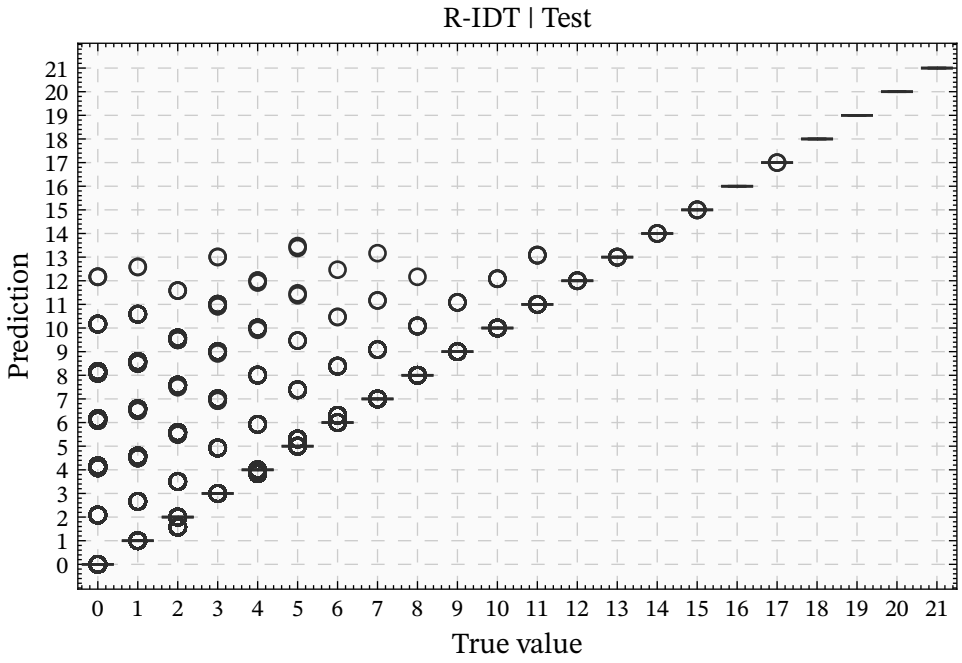
Table 5.5: Comparison of the MAE values of the presented models on the data splits for Blocks-On.

Model	Train Data	Validation Data	Test Data
Base R-GNN	0.0053	**0.0063**	**0.0196**
R-IDT	**0.0000**	0.0221	0.2314

On this model, many R-IDTs with different hyperparameters and seeds are fitted. The R-IDT model we consider in the following works with a layer width of $w = 10$ and a sample size of

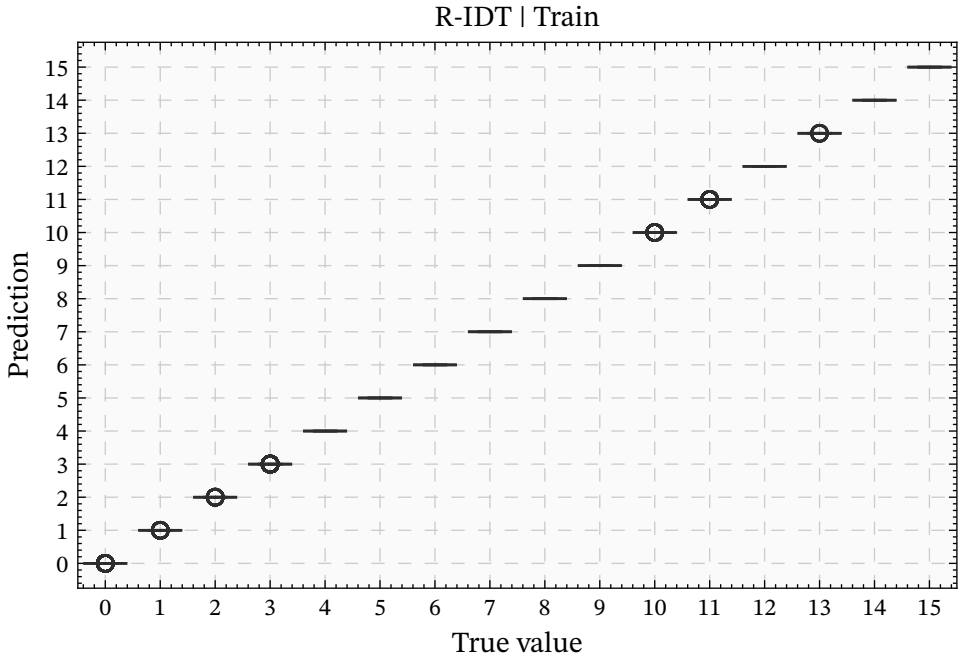


(a) Predictions of R-IDT A on the train set.

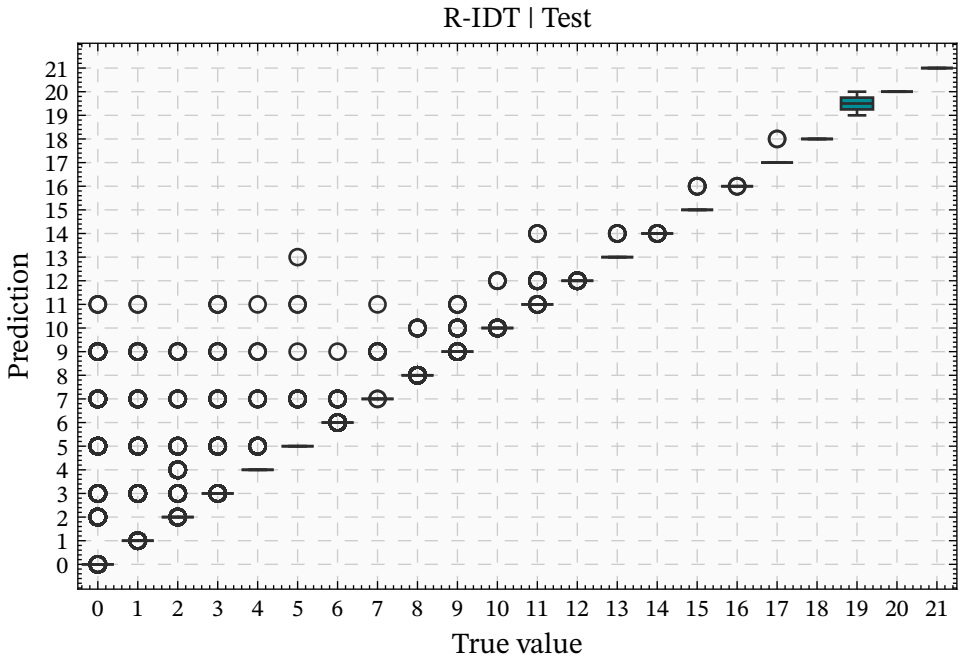


(b) Predictions of R-IDT A on the test set.

Figure 5.7: Predictions of R-IDT A for Blocks-Clear on the train set (first figure) and test set (second figure). Note the different axis scales.

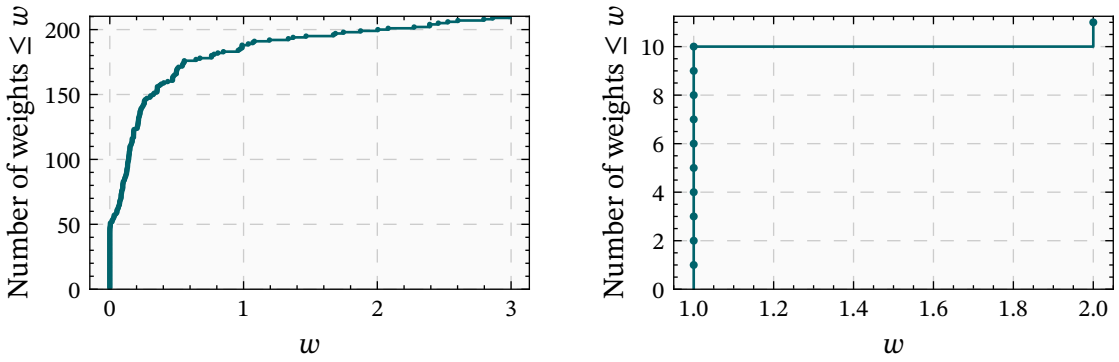


(a) Predictions of R-IDT B on the train set.



(b) Predictions of R-IDT B on the test set.

Figure 5.8: Predictions of R-IDT B for Blocks-Clear on the train set (first figure) and test set (second figure). Note the different axis scales.



(a) The 210 non-zero weights of R-IDT A.

(b) The 11 non-negative weights of R-IDT B.

Figure 5.9: The distributions of feature weights learned by the R-IDTs A (left) and B (right).

$s = 10\,000$ for each layer. Its feature types are set to combined and numerical state features. The error values of one such model are shown in Table 5.5.

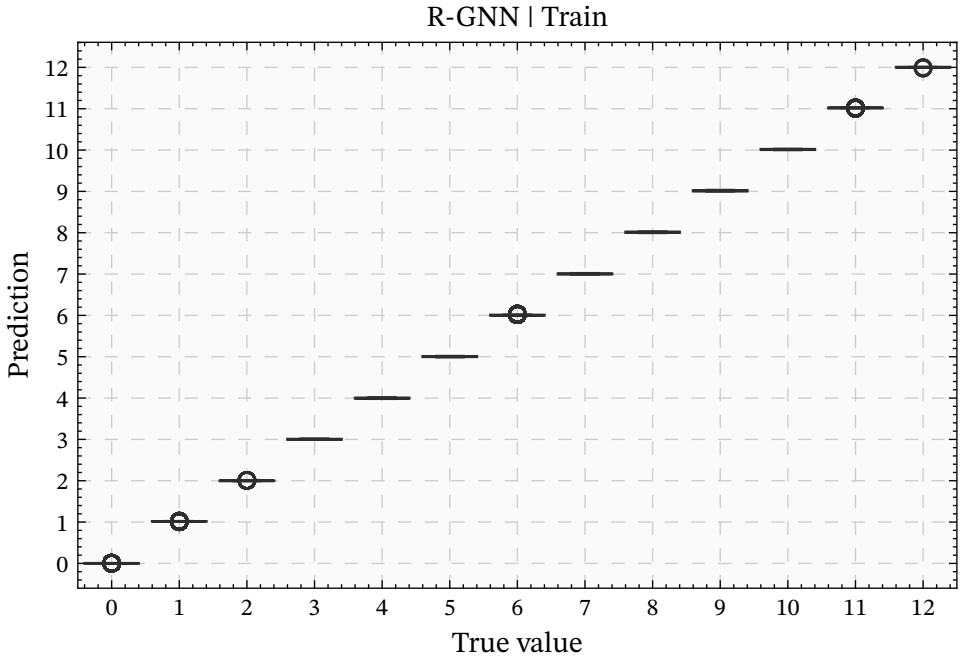
For brevity, plots about prediction details are omitted. To at least fit the training data perfectly, we found that layer depths of at least seven were required. Further, it can be observed that complete, single, and Ward's clustering work best with the Euclidean distance or the cosine similarity measures. The random seed again shows a high impact on the resulting models. Similar to Blocks-Clear, the use of parallel final layers seems to have a positive influence on the prediction quality, and mitigates the influence of the random seed to some degree.

For this problem, no single regression with sparse (integer) weights was learned, that is, all regressions learned to use hundreds of features, mostly with weights with absolute values less than 1. Thus, no learned model can be considered as interpretable.

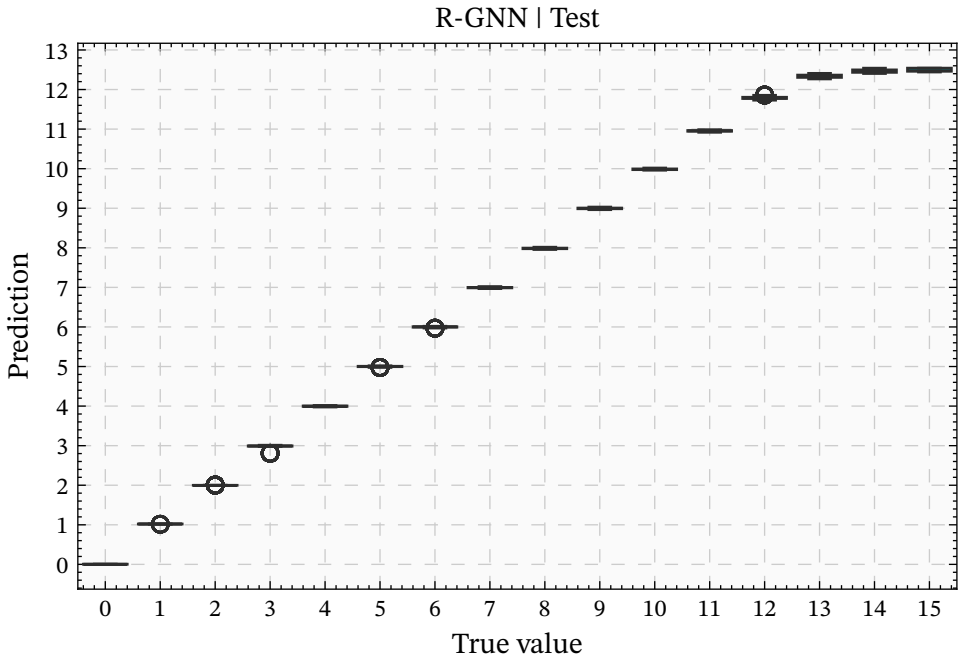
5.5.5 Visitall

The R-GNNs for Visitall have $L = 20$ layers and an embedding size of $k = 32$. The considered problem instances of Visitall have discrete square grids of up to $n = c^2 = 100$ cells, where each cell is connected to its direct neighbors. Thus, there is a path with a length of at most $2(c - 1)$ from any object to any other object in every state graph. Therefore, 20 layers are sufficient for objects to send messages to any other objects. The base R-GNN we detail in the following takes the maximum both for aggregation and pooling. Its predictions are shown in Figure 5.10. While it learned the optimal value function very well on the training states, its generalization on the test states is limited. In the predictions' visualization, its visible that the model works well for values up to 12, that is, on states which have shortest paths of length that are also present in the train states. Therefore, it has learned to ignore irrelevant cells. For the values 13 – 15, the predictions flatten out.

Many R-IDTs are distilled on the introduced base R-GNN (and other models not shown here). We found that the required depths for the inner and final R-IDT layers to obtain the presented



(a) Predictions of the base R-GNN on the train set.



(b) Predictions of the base R-GNN on the test set.

Figure 5.10: Predictions of the base R-GNN for Visitall on the train set (first figure) and test set (second figure). Note the different axis scales.

Table 5.6: Comparison of the MAE values of models presented for Visitall.

Model	Train Data	Validation Data	Test Data
Base R-GNN	0.0068	**0.0232**	**0.0390**
R-IDT A	**0.0000**	0.3168	0.5757
R-IDT B	**0.0000**	0.3540	0.7657

results is at least five. The experiments were conducted with the inner layer's sample size set to $s = 10\,000$, and the width to $w = 10$. Since the Visitall problem class is unbounded, one could expect models using numerical features to generalize better. However, this does not turn out to be the case.

While some models that fit the train data perfectly have been distilled, they do not generalize well. Consider R-IDT A, which uses combined features only, and R-IDT B, which uses combined and numerical features. Their MAEs are shown in Table 5.6, and their predictions are displayed in Figures 5.11 and 5.12, respectively. Both models fail to generalize to states with a higher value than present in the train states. Further, some of their predictions for values that are included in the train states, that is values up to 12, are wrong.

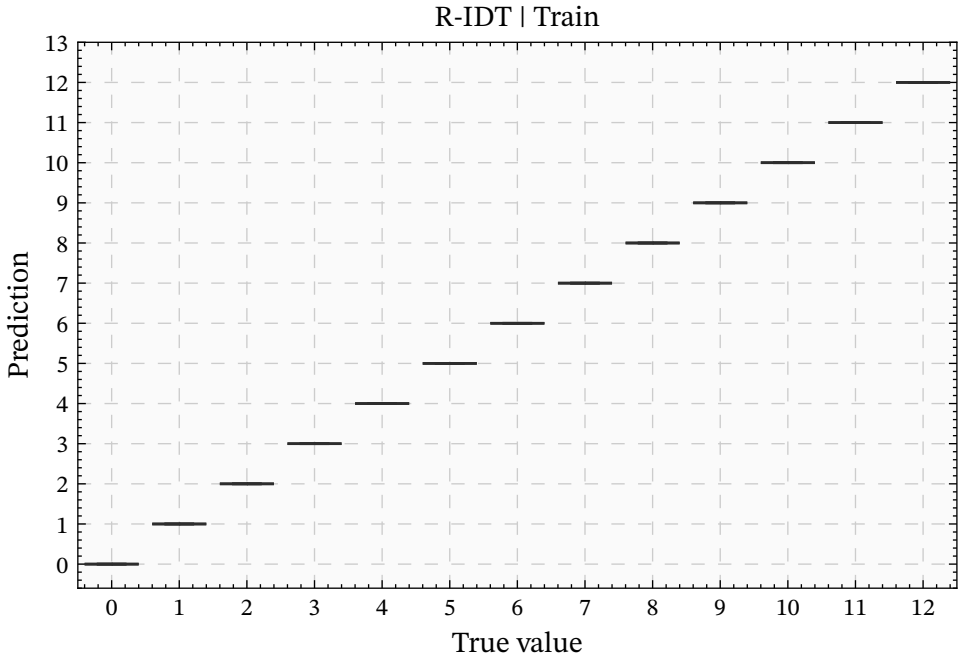
As with any R-IDT presented so far, the learned features are not understandable due to their complexity. As the number of layers of the base R-GNN is 20, nested formulas of up to a depth of 20 may be created. While R-IDT A has learned twelve non-zero integer weights in the range of 1 – 6, their meaning is unclear.

When analyzing hyperparameter relevance, we can draw only limited conclusions. We find that the cosine similarity and the Euclidean distance worked better for this problem. The type of pooling of the R-IDTs does not seem to be relevant. However, the random seed is of importance. This is highlighted especially by the fact that similar to experiments on other problem classes, fitting two models with the same hyperparameter configuration, but a different seed, can result in very different models. Interestingly, using parallel final layers did not seem to have an impact on this, as opposed to the other unbounded problems Blocks-Clear and Blocks-On.

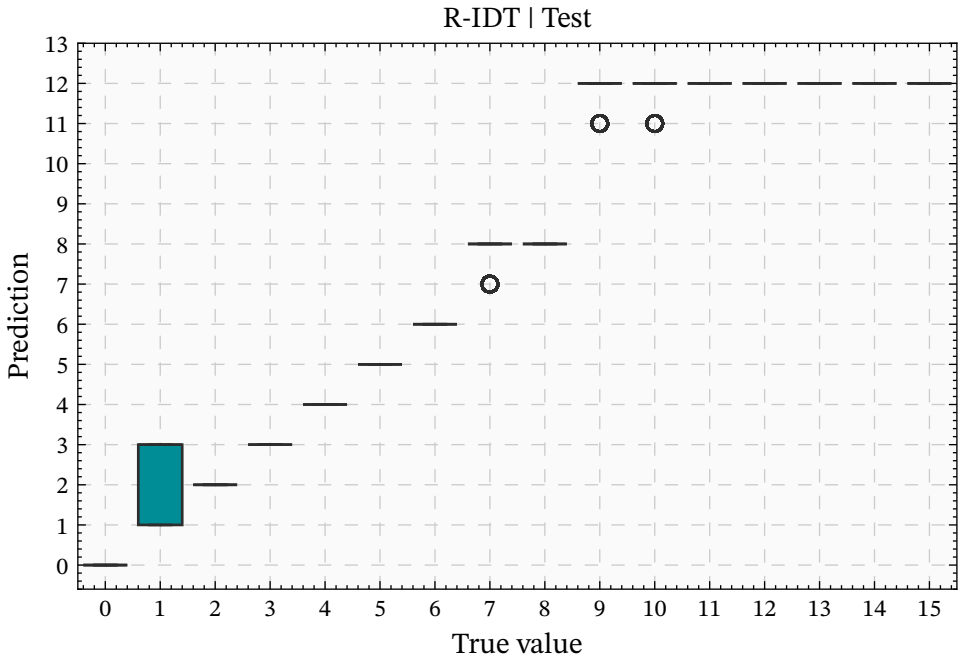
5.6 Discussion

In this section, we summarize the findings from the experimental evaluation, highlight good results, and address the apparent limitations of the current approach. Further, we introduce some ideas that may help to mitigate the limitations in future work.

The experiments showed that for the bounded problems Gripper and Miconic, R-IDTs are indeed able to learn linear optimal value functions that generalize to larger instances. The resulting value functions can be written in a compact form, using only a few combined Boolean features and integer weights. However, the learned features are not readable and thus, not

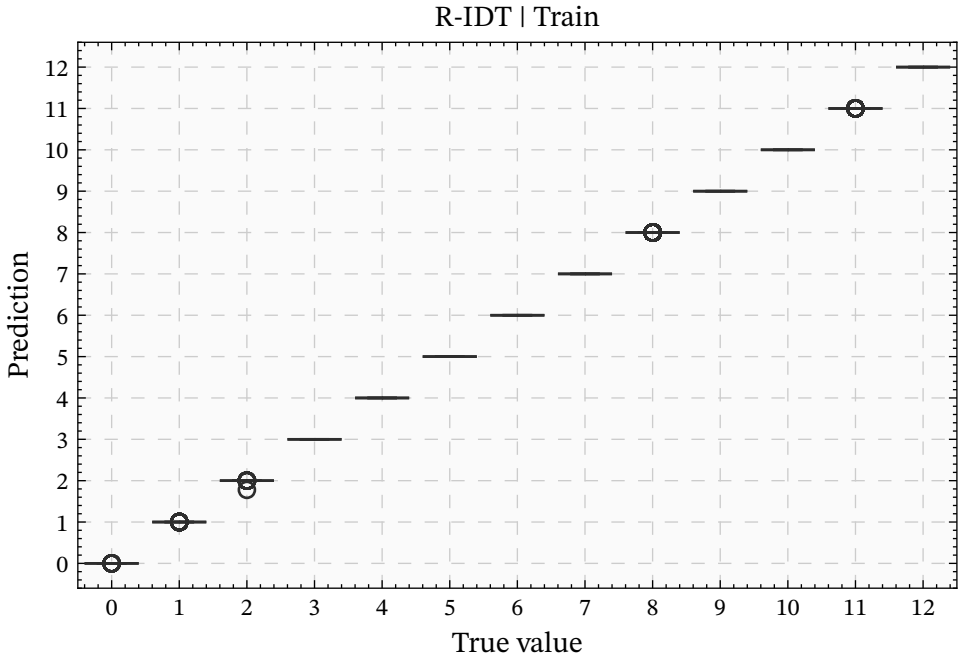


(a) Predictions of R-IDT A on the train set.

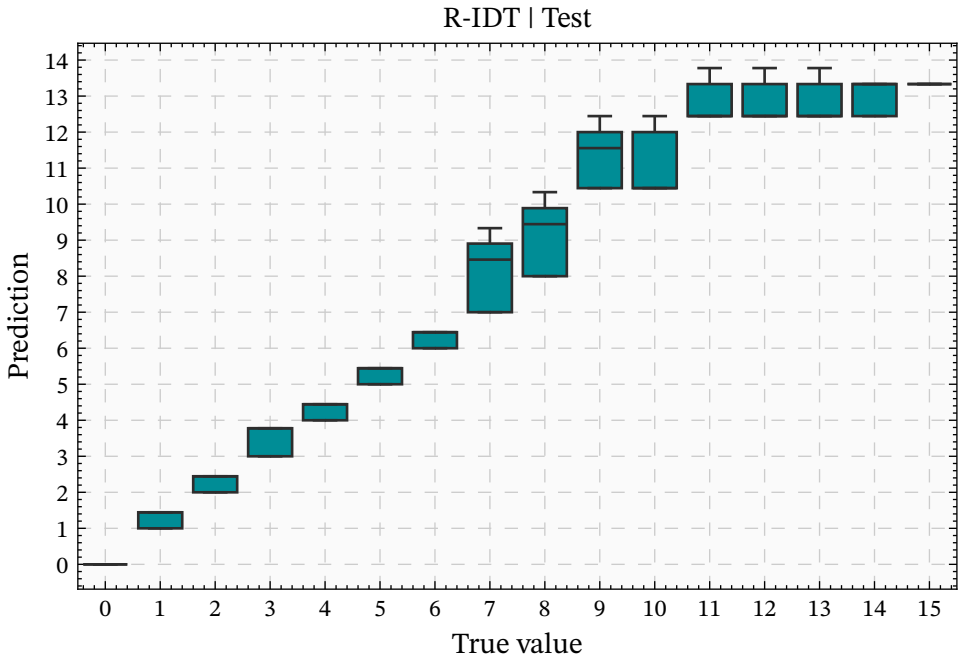


(b) Predictions of R-IDT A on the test set.

Figure 5.11: Predictions of R-IDT A for Visitall on the train set (first figure) and test set (second figure). Note the different axis scales.



(a) Predictions of R-IDT B on the train set.



(b) Predictions of R-IDT B on the test set.

Figure 5.12: Predictions of R-IDT B for Visitall on the train set (first figure) and test set (second figure). Note the different axis scales.

interpretable. This is because of the high layer depths (four and five) required to successfully learn models, combined with the R-GNNs using $L = 5$ layers. The resulting state-level features are defined by formulas of disjunctions of conjunctions, where each conjunction combines up to four or five object features, respectively. Each object feature itself is equally complex and further, is built up from formulas with a depth of up to L, that is, up to 5 modal parameters may be nested. This results in a massive amount of formulas, which is simply too large to comprehend.

For the unbounded problems Blocks-Clear, Blocks-On, and Visitall, R-IDTs can learn value functions that predict the optimal value function perfectly on the train states, but fail to generalize to larger instances. The failure to generalize is twofold, as for some problems, higher values are not predicted correctly, see for example the presented models for Visitall. On the other hand, for Blocks-Clear and Blocks-On, the predictions can scale up well to higher values when including numerical features, but show errors on low values. Model interpretability is even more limited for the unbounded problems, as in some cases, the regressions learn many noninteger weights, which prohibits interpreting the learned value function regardless of the learned features' interpretability. However, as for these problems, even higher layer depths and R-GNNs with more layers were required, understanding the learned features is even harder as for the bounded problems.

While many R-IDT model details can be configured using hyperparameters, our experiments only allow limited conclusions about good configurations and hyperparameter influences. We conclude that for all considered problems, max pooling, that is, learning the combined features based on Boolean object features, is sufficient. The layer depths have a huge influence on the resulting models, as across all problems, we find that the models require a certain layer depth to learn useful features. Why the respective layer depths are required remains unknown, especially as Section 5.1 shows that required features can also be written with smaller conjunctions. Mostly, the best results were achieved when using the Euclidean distance or cosine similarity as clustering metrics. The clustering linkage did not seem to have a huge influence. Across all problems, OLS provided solutions with better MAE values than SGD, and sometimes learned sparse integer weights where SGD did not.

Lastly, the random seed had a much larger influence than anticipated. This can likely be attributed to the decision trees being sensitive to the subsets of objects they are fitted on, which is sampled using the seed.

5.6.1 Directions for Future Work

In the following, we present some untested ideas that may be able to overcome the present limitations. However, implementing and testing them is beyond the scope of this thesis. They could serve as directions for future work.

To achieve better generalization for unbounded problems, it may be beneficial to detect patterns in the learned state features. For example, consider Q_{clear}, where a variable number of features B_k can be used to compose the optimal value function (Equation (2.6)). When an R-IDT has learned some of these features and their weights $\{\langle w_1, B_1\rangle, \dots, \langle w_n, B_n\rangle\}$, it is a reasonable assumption that for all $i > n$, the pair $\langle w_i, B_i\rangle$ is required for the model to generalize. However, how such patterns may be detected automatically and how weights and features for $i > n$ may be predicted falls outside the scope of this thesis. Further, the impact of this extension is hard to judge, as it is unclear, whether many such sequences are learnt at all.

Another idea to help with generalization issues is to make the transitive closure of relations explicitly available to R-IDTs. As shown in Section 5.1.6, this allows expressing generalizing numerical features easier. However, R-GNN cannot directly access transitive closures, but have to compute them layer by layer, as is the case with the currently implemented R-IDTs. Whether it is beneficial to give an R-IDT information that the distilled R-GNN does not have, is currently unclear.

As explained above, the formulas that R-IDTs learn are complex. Simplifying them automatically could make them interpretable, but whether or how they can be simplified automatically is currently not clear.

The following suggestion extends beyond the R-IDT approach, as it is also concerned with the types of R-GNN that get distilled. To obtain an optimal policy, it is not strictly necessary to predict the exact value of a state, but predicting whether a successor state has a lower value then the current state, that is, predicting whether $V^*(s') < V^*(s)$, is sufficient. Pairs of states s, s', where s' is a successor of s, can be combined into a single state graph $G(s, s')$. Such combined graphs can both capture the state graphs individually and connect each object to its counterpart in the other state, allowing the R-GNNs and R-IDTs to work on *relative changes* of object properties instead of absolute properties. Further, the prediction problem reduces from an unbounded regression to a binary classification. This may allow R-GNNs to generalize even better, and simplifies the task of distilling them enormously, as R-IDTs would no longer need to fit a regression.

Finally, changing how the random subset of training data is sampled for each R-IDT layer could mitigate the high influence of the random seed. Instead of sampling data uniformly at random, it could be sampled in a way such that each layer gets states that allow for meaningful splits to be found. How to exactly define meaningful splits and an appropriate sampling mechanism, however, is not clear and remains outside the scope of this thesis.

6 Conclusion

This thesis has addressed the gap between symbolic and GNN-based methods for learning optimal policies for generalized planning problems. While existing symbolic approaches for generalized planning make use of large given feature pools, R-GNN-based approaches do not need feature pools, but work directly on state graphs. However, the resulting neural models are not interpretable.

To bridge the gap between the symbolic and neural approaches, this thesis introduced the R-IDT as a predictive model for learning symbolic formulas independently a predefined feature pool, instead leveraging the embeddings computed by a fitted R-GNN. R-IDTs are both a generalization and modification of the original IDTs model proposed by Pluska et al. [40], adapted to the framework of generalized planning.

To apply R-IDTs to planning states, the states are first encoded as multi-relational graphs with Boolean object features. To fit a model guided by an R-GNN, a set of state graphs and the embeddings provided by the R-GNN are required. The model learning process is designed to reflect the layerwise neural computations of R-GNNs, and is guided by the respective embeddings. To learn the required state-level features, R-IDTs derive them from the learned object-level features. Additionally, Boolean combinations of state-level features are learned explicitly, leveraging the state-level embeddings computed by the R-GNN. The derived state-level features are used to learn a linear regression, that predicts the respective optimal value function.

R-IDTs are experimentally evaluated on five generalized planning problems. The experiments revealed that R-IDTs are indeed able to learn features and weights that can compose linear value functions. For bounded problems, R-IDTs were able to generalize perfectly to bigger instances, while generalizing in unbounded problems remains a challenge. Regarding the interpretability of the learned value functions, the experiments showed that the formulas that describe the learned features are too complex to be understandable. While in some cases, the learned weights are sparse and allow expressing the learned value function in a compact way, this is not always the case, posing an additional challenge to interpreting the learned function.

The previous section addresses the revealed limitations of interpretability and generalization. It proposes several modifications and additions to the current state of implementation, providing directions for future work to extend the proposed R-IDT method. Further, an altered framework to apply R-IDT to generalized planning is proposed.

A Appendix

Listing A.1: Complete PDDL definition of the Blocksworld domain.

```
(define (domain BLOCKS)
    (:requirements :strips)
    (:predicates (on ?x ?y) (ontable ?x)
        (clear ?x) (handempty) (holding ?x))

    (:action pickup
        :parameters (?x)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect (and
            (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)
        )
    )
    (:action putdown
        :parameters (?x)
        :precondition (holding ?x)
        :effect (and
            (not (holding ?x))
            (clear ?x)
            (handempty)
            (ontable ?x)
        )
    )
    (:action stack
        :parameters (?x ?y)
        :precondition (and (holding ?x) (clear ?y))
        :effect (and
            (not (holding ?x))
            (not (clear ?y))
            (clear ?x)
            (handempty)
            (on ?x ?y)
        )
    )
    (:action unstack
        :parameters (?x ?y)
        :precondition (and (on ?x ?y) (clear ?x) (handempty))
```

```
40          :effect (and
41              (holding ?x)
42              (clear ?y)
43              (not (clear ?x))
44              (not (handempty))
45              (not (on ?x ?y))
46          )
47      )
48  )
```

List of Acronyms

AC-GNN	Aggregate-Combine Graph Neural Network
AI	Artificial Intelligence
GNN	Graph Neural Network
IDT	Iterated Decision Tree
MAE	Mean Absolute Error
MLP	Multilayer Perceptron
OLS	Ordinary Least Squares
PDDL	Planning Domain Definition Language
R-GNN	Relational Graph Neural Network
R-IDT	Relational Iterated Decision Tree
SGD	Stochastic Gradient Descent

List of Symbols

General

x	(scalar) variable x
$\mathbf{x}$	vector variable $\mathbf{x}$
$\mathbf{X}$	matrix variable $\mathbf{X}$
$[\bullet \quad \bullet]$	horizontal concatenation of vectors or matrices
$\|\bullet\|$	Euclidean norm
$\bullet^{\mathrm{T}}$	transpose of a matrix
$\mathbf{0}$	zero matrix of implicit size
$\mathbf{1}$	all-ones matrix of implicit size
$\mathbf{I}$	identity matrix of implicit size
$\coloneqq$	defined as
$\approx$	approximately equal
$\langle \cdot, \cdot, \cdot \rangle$	tuple of multiple values
$\bar{o}$	tuple of implicit size
$\mathbb{N}$	set of non-negative integers
$\mathbb{R}$	set of real numbers
$\mathbb{B}$	set of the numbers 0 and 1
$\{\!\{\ldots\}\!\}$	multiset
$\mathbb{I}[\cdot]$	Iverson bracket
$\mathcal{O}$	big O notation for the upper bound of growth

Graph Neural Network

L	number of layers
k	embedding size
$\mathbf{x}_v^{(i)}$	embedding of node v at layer i
agg	aggregation function
comb	combination function
pool	pooling function
MLP	multilayer perceptron function

Graph

V	set of vertices
E	set of edges
$N(\cdot)$	neighborhood of a node

$\mathbf{A}$	adjacency matrix
$\mathbf{U}$	feature matrix

Logic

$\bot$	false
$\top$	true
$\models$	satisfies
$\equiv$	equivalent
C_2	first-order logic fragment with two variables and counting quantifiers
GC_2	formulas of C_2 which are guarded by a relation

Planning

$\mathcal{D}$	domain
$\mathcal{I}$	instance
P	set of predicates
A	set of action schemata
O	set of objects
$Init$	initial state
$Goal$	goal conditions
$\mathrm{ar}(\cdot)$	arity of a predicate
S	state space
s	state
s_0	initial state
S_G	set of goal states
Act	set of ground actions
$A(s)$	set of actions applicable in s
$f(s, a)$	successor state of applying a in s
V^*	optimal value function

List of Figures

List of Tables

List of Algorithms

List of Listings

List of References

[1] C. Aggarwal and C. Reddy, Eds., *Data Clustering: Algorithms and Applications*. Chapman and Hall/CRC, 2014. DOI: 10.1201/9781315373515.

[2] J. S. Aguas, S. J. Celorrio, and A. Jonsson, "Generalized planning with procedural domain control knowledge," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 26, 2016, pp. 285–293.

[3] M. Aichmüller and H. Geffner, *Sketch decompositions for classical planning via deep reinforcement learning*, 2025. arXiv: 2412.08574 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2412.08574.

[4] M. Aichmüller and J. Krude, *Plangolin*, 2025. [Online]. Available: https://github.com/maichmueller/plangolin.

[5] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. [Online]. Available: https://docs.pytorch.org/assets/pytorch2-2.pdf.

[6] F. Baader, I. Horrocks, C. Lutz, and U. Sattler, *An Introduction to Description Logic*. Cambridge University Press, 2017.

[7] P. Barceló, E. V. Kostylev, M. Monet, J. Pérez, J. Reutter, and J.-P. Silva, "The Logical Expressiveness of Graph Neural Networks," in *8th International Conference on Learning Representations (ICLR 2020)*, Virtual conference, Ethiopia, Apr. 2020. [Online]. Available: https://hal.science/hal-03356968.

[8] V. Belle and H. J. Levesque, "Foundations for generalized planning in unbounded stochastic domains.," in *KR*, 2016, pp. 380–389.

[9] L. Biewald, *Experiment tracking with weights and biases*, Software available from wandb.com, 2020. [Online]. Available: https://www.wandb.com/.

[10] C. Bishop, *Pattern Recognition and Machine Learning*. Springer, Jan. 2006. [Online]. Available: https://www.microsoft.com/en-us/research/publication/pattern-recognition-machine-learning/.

[11] B. Bonet, G. Francès, and H. Geffner, "Learning features and abstract actions for computing generalized plans," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 2703–2710, Jul. 2019. DOI: 10.1609/aaai.v33i01.33012703.

[12] B. Bonet and H. Geffner, "Features, projections, and representation change for generalized planning," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2018, pp. 4667–4673. DOI: 10.24963/ijcai.2018/649.

[13] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1, pp. 5–33, 2001, ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00108-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001084.

[14] B. Bonet, H. Palacios, and H. Geffner, "Automatic derivation of memoryless policies and finite-state controllers using classical planners," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 19, no. 1, pp. 34–41, Oct. 2009. DOI: 10.1609/icaps.v19i1.13379. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/13379.

[15] M. M. Deza and E. Deza, *Encyclopedia of Distances*, 4th ed. Springer Berlin, Heidelberg, Aug. 2016. DOI: 10.1007/978-3-662-52844-0.

[16] D. Drexler, S. Ståhlberg, B. Bonet, and H. Geffner, "Symmetries and expressive requirements for learning general policies," in *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, vol. 21, Aug. 2024, pp. 845–855. DOI: 10.24963/kr.2024/79.

[17] W. Falcon and The PyTorch Lightning team, *PyTorch Lightning*, version 1.4, Mar. 2019. DOI: 10.5281/zenodo.3828935. [Online]. Available: https://github.com/Lightning-AI/lightning.

[18] M. Fey, J. Sunil, A. Nitta, R. Puri, M. Shah, B. z. Stojanovi c, R. Bendias, A. Barghi, V. Kocijan, Z. Zhang, X. He, J. E. Lenssen, and J. Leskovec, "PyG 2.0: Scalable Learning on Real World Graphs," *Temporal Graph Learning Workshop @ KDD*, 2025.

[19] G. Frances, B. Bonet, and H. Geffner, "Learning general planning policies from small examples without supervision," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 11 801–11 808.

[20] G. Francès, A. B. Corrêa, C. Geissmann, and F. Pommerening, "Generalized potential heuristics for classical planning," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2019, pp. 5554–5561. DOI: 10.24963/ijcai.2019/771.

[21] H. Geffner and B. Bonet, *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers, 2013.

[22] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld, "Pddl - the planning domain definition language," Aug. 1998.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[24] M. Grohe, "The logic of graph neural networks," in *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '21, Rome, Italy: Association for Computing Machinery, 2021, ISBN: 9781665448956. DOI: 10.1109/LICS52264.2021.9470677.

[25] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[26] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2.

[27] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise, *An Introduction to the Planning Domain Definition Language / by Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, Christian Muise* (Synthesis Lectures on Artificial Intelligence and Machine Learning), eng. Cham: Springer International Publishing, 2019, ISBN: 9783031015847.

[28] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning, Data Mining, Inference, and Prediction, Second Edition* (Springer Series in Statistics), 2nd ed. Springer New York, NY, 2009. DOI: 10.1007/978-0-387-84858-7. [Online]. Available: https://hastie.su.domains/ElemStatLearn/printings/ESLII_print12_toc.pdf.download.html.

[29] R. Horčík and G. Šír, "Expressiveness of graph neural networks in planning domains," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 34, no. 1, pp. 281–289, May 2024. DOI: 10.1609/icaps.v34i1.31486.

[30] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989, ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[31] Y. Hu and G. De Giacomo, "Generalized planning: Synthesizing plans that work for multiple environments," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, 2011.

[32] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[33] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1412.6980.

[34] A. Longa, S. Azzolin, G. Santin, G. Cencetti, P. Liò, B. Lepri, and A. Passerini, "Explaining the explainers in graph neural networks: A comparative study," *ACM Computing Surveys*, vol. 57, no. 5, pp. 1–37, 2025.

[35] C. Lutz, U. Sattler, and F. Wolter, "Modal logic and the two-variable fragment," in *Computer Science Logic*, L. Fribourg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 247–261, ISBN: 978-3-540-44802-0. DOI: 10.1007/3-540-44802-0_18.

[36] M. Martín and H. Geffner, "Learning generalized policies from planning examples using concept languages," *Applied Intelligence*, vol. 20, no. 1, pp. 9–19, 2004.

[37] I. D. Mienye and N. Jere, "A survey of decision trees: Concepts, algorithms, and applications," *IEEE Access*, vol. 12, pp. 86 716–86 727, 2024. DOI: 10.1109/ACCESS.2024.3416838.

[38] D. Misra, *Mish: A self regularized non-monotonic activation function*, 2020. arXiv: 1908.08681 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1908.08681.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[40] A. Pluska, P. Welke, T. Gärtner, and S. Malhotra, "Logical Distillation of Graph Neural Networks," in *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, Aug. 2024, pp. 920–930. DOI: 10.24963/kr.2024/86.

[41] S. Russell and P. Norvig, *Artificial Intelligence A Modern Approach, Global Edition*. Pearson Deutschland, 2016, ISBN: 9781292153964. [Online]. Available: https://elibrary.pearson.de/book/99.150005/9781292153971.

[42] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009. DOI: 10.1109/TNN.2008.2005605.

[43] S. Srivastava, N. Immerman, and S. Zilberstein, "Learning generalized plans using abstract counting.," in *AAAI*, vol. 8, 2008, pp. 991–997.

[44] S. Srivastava, N. Immerman, and S. Zilberstein, "A new representation and associated algorithms for generalized planning," *Artificial Intelligence*, vol. 175, no. 2, pp. 615–647, 2011. DOI: 10.1016/j.artint.2010.10.006.

[45] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, no. 1, pp. 629–637, Jun. 2022. DOI: 10.1609/icaps.v32i1.19851.

[46] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning generalized policies without supervision using gnns," in *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning : Special Session on KR and Machine Learning*, ser. Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), IJACAI Organization, 2022, pp. 474–483, ISBN: 9781956792010. DOI: 10.24963/kr.2022/49.

[47] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning General Policies with Policy Gradient Methods," in *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, Aug. 2023, pp. 647–657. DOI: 10.24963/kr.2023/63.

[48] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, second edition. MIT Press, Nov. 13, 2018, ISBN: 978-0-262-03924-6. [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf.

[49] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[50] J. H. Ward, "Hierarchical grouping to optimize an objective function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, Mar. 1963, ISSN: 01621459, 1537274X. Accessed: Feb. 19, 2026. [Online]. Available: http://www.jstor.org/stable/2282967.

[51] M. L. Waskom, "Seaborn: Statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, 2021. DOI: 10.21105/joss.03021.

[52] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021. DOI: 10.1109/TNNLS.2020.2978386.