A Reinforcement Learning Approach to Graph Edit Distance Estimation

Bachelor Thesis
Faculty of Science, University of Bern

submitted by

Gian Alexander Zwahlen

from Bern, Switzerland

Supervision:

PD Dr. Kaspar Riesen
Dr. Anthony Gillioz
Institute of Computer Science (INF)
University of Bern, Switzerland

Abstract

The continuous growth of large-scale networks in communication, transportation and data storage, among other areas, makes graph-based pattern recognition an increasingly vital tool for extracting insights and predicting changes in these complex systems. Particularly error-tolerant graph matching via the graph edit distance (GED) has proven useful in the domains of computer vision, bioinformatics and cybersecurity. In this thesis, a novel method for approximating the GED using reinforcement learning (RL) is explored. Unlike other methods, the GED is formulated as a sequential decision-making problem. An advantage actor-critic (A2C) agent is trained to select a sequence of graph edit operations, also called an edit path, to approximate the GED. A custom RL environment was designed and developed to handle the dynamics of sequential graph transformation. The environment leverages graph and recurrent neural networks to extract relevant information for the agent. The architecture's capabilities are validated through experimental evaluation on the graph atlas dataset. The challenges caused by sparse rewards and reward shaping are discussed and their solutions evaluated. Influential system parameters are analysed and their impact is contextualised qualitatively. Insights into the agent's behaviour are provided and patterns in its found edit paths are highlighted. While not aiming to outperform existing methods, it serves as proof of concept for applying RL to approximate the GED via an edit path. It demonstrates an interpretable alternative to current neural network-based approaches to GED approximation and provides the foundation for further research on larger, real-world graph data.

Acknowledgements

I would like to express my gratitude to my family and the Swiss education system, particularly PD Dr. Kaspar Riesen and Dr. Anthony Gillioz from the Pattern Recognition Group for their continuous guidance.

Contents

Chapter 1

Introduction

Graph-based representations are used in various domains such as computer graphics [1], computer vision [2], chemistry [3], bioinformatics [4], knowledge graphs [5] and many more. Graphs are flexible and expressive data structures, able to represent shapes, knowledge and relationships among other things. In pattern recognition however, commonly the goal is to compare, classify or aggregate observations based on extractable feature patterns. This constitutes the ability to measure some form of similarity between extracted feature patterns and given observation features. While well understood for vectorisable features in $\mathbb{R}^n$, this is not necessarily the case for observations in non euclidean spaces like the graph domain. Thus, various techniques for graph comparison i.e. graph matching have been developed such as graph kernels [6] or graph embedding methods [7], allowing distances between graphs to be measured.

1.1 Graph Edit Distance

One of the most flexible and interpretable error-tolerant graph matching techniques is the graph edit distance (GED), which formulates the similarity between two graphs g_1 and g_2 as a minimal sequence of elementary transformations to transform g_1 into g_2, with associated edit costs. This makes GED applicable to any graph pair, regardless of their respective node label or edge label feature space. Consequently, the drawback of exact GED algorithms [8] is its computational complexity as a NP-hard problem [9]. The complexity grows exponentially with the number of input nodes, making the computation of the exact GED prohibitively expensive for larger graphs, that is typically for graphs with more than 20 nodes. To still harness the benefits of GED, algorithms that find an approximate solution of the GED in polynomial time have been proposed [10][11]. More recent research leverages graph neural networks (GNNs), to approximate the GED [12][13] while also improving

accuracy or computational performance. In particular, GNN methods are performant, but they often lack an associated edit path since the GED is approximated in a latent space rather than in the graph domain. This makes the approximate GED more difficult to interpret and explain. One potential solution to remain in the graph domain is using reinforcement learning.

1.2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm where an agent selects an action to be performed on an environment in a specific state, thus transitioning into a new state with the agent receiving a reward. Ultimately the agent's aim is to maximize the return, commonly the sum of discounted cumulative rewards, by learning a policy to select the reward maximising actions. It has applications in robotics and automation [14], autonomous driving [15], games [16], and large language models [17]. Research has thus demonstrated the potential of RL to solve complex decision making problems, even in domains with high-dimensional or combinatorially large solution spaces. In addition, unlike supervised learning, RL does not require data with ground-truth labels for its learning procedure, which makes it attractive for various problems where acquiring such a dataset would be prohibitively expensive. These properties make it an attractive choice for graph-based problems in general [18] and GED in particular, since generating a dataset with GED ground-truth labels is impractical for larger graphs.

1.3 Approach

Utilizing RL to learn the GED represents an innovative and unexplored approach that differs from established GED approximation methods, lacking any prominent prior research. Therefore, the task is framed as a sequential decision making process, where an agent continuously selects graph edit operations to transform the given source graph into the target graph. This produces a sequentially generated edit path with its associated edit cost approximating the GED. This differs from using neural networks to directly learn an approximation of the GED metric, sharing more similarities with methods based on A^*-search [19]. For the specific RL algorithm, policy gradient variants are considered robust baseline options for a majority of tasks. They are well suited for rapid model iteration and have shown promising results on high complexity tasks.

1.4 Goals

Since this is — at the time of writing — the first attempt at utilizing RL to learn the GED, the focus is on ground work and experimentation rather than attempting to achieve comparable performance to established machine learning-based approaches. Therefore, the goals are set conservatively.

1. Develop a custom environment that handles graph edit operations and transitions, keeps record of the performed edit path, and validates if the source graph has been successfully transformed into the target graph.

2. Jointly implement a suitable RL architecture for selecting optimal edit operations based on the given state of the graph pair.

3. Experiment with various components and parameters to improve performance in selected metrics. This includes how the action space is defined, the reward signals, the network architecture, its hyperparameters and more.

4. Establish a proof of concept on the synthetic graph atlas [20] dataset, ideally validating the potential of RL to eventually approximate the GED for graphs of arbitrary size and feature space.

Considering a positive outcome of the initial goals, further goals can be defined and pursued, particularly expanding the environment and RL architecture to handle some of the common, larger datasets in the TUDataset benchmark suite [21].

The remainder of the thesis is structured as follows: Chapter 2 reviews in detail the theoretical foundation of RL, the GED and the applicability where they intersect. Following, Chapter 3 is dedicated to the general and specific implementation aspects while drawing both parallels and distinctions to existing RL methods. In Chapter 4 roadblocks and their solutions as well as primarily qualitative evaluations between explored avenues will be elaborated on. To conclude, in Chapter 5 the set goals are revisited, especially evaluating the suitability of RL, and paths for further research are outlined.

Chapter 2

Basic Concepts

This chapter will contain the necessary theoretical foundations and formal definitions from the realm of graph theory as well as the realm of RL. Firstly, the relevant basic graph terminology will be introduced, building up to the definition of GED used going forward. Secondly, RL will first be explained in general, then in specific to the application, culminating in an overview of current state-of-the-art methods in RL. Finally, both the definition of GED and the core of the RL paradigm are reframed to justify their joint applicability. Much of the material in this chapter is adapted from the introductory works by Riesen [22] and Sutton & Barto [23].

2.1 Graphs and Graph Edit Distance

Before discussing any distance measures, it is pertinent to introduce the elements over which we want to define said measures. In this case, the elements are graphs, entity-relationship data structures consisting of nodes and edges, and their particular node and edge feature space.

Definition 2.1 (Graph)**.** Let $\mathcal{L}_V$ and $\mathcal{L}_E$ be a finite set of node and edge labels, respectively. The *graph* is a four-tuple $g = (V, E, \ell_V, \ell_E)$ with,

- V being the finite set of *nodes*, sometimes called *vertices*.
- $E \subseteq V \times V$ being the finite set of *edges*.
- $\ell_V : V \to \mathcal{L}_V$ being the node labelling function.
- $\ell_E : E \to \mathcal{L}_E$ being the edge labelling function.

Further let $\mathcal{G}$ denote the set of all possible graphs over given label sets $\mathcal{L}_V$ and $\mathcal{L}_E$, called the *graph domain*. The *order* and *size* of a graph are denoted by $|V|$ and $|E|$, that is its number of nodes and edges, respectively. Edges are given by their

pair of nodes $\{u, v\}$, with $v, u \in V$ called the edge's *endpoints*. The edge is said to *join* node u and v and be *incident* on them, joined nodes are called *adjacent*. The *degree* of a node $v \in V$ is the number of edges incident to it i.e., the number of nodes adjacent to v. Consequently, the *neighbourhood* of a given node $v \in V$, denoted by $\mathcal{N}(v)$, is defined as the set of adjacent nodes $\{v_i\}$ to v. In cases where the neighbourhood references an *induced subgraph*[1] of g, induced by the adjacent nodes of v, denote it by $\mathcal{N}_g(v)$. In *directed* graphs edges are denoted by an ordered pair (u, v) with $u \in V$ being the source node, and $v \in V$ the target node. Any directed graph can model its undirected form by inserting a reverse edge (v, u) for every edge $(u, v) \in E$, with $\ell_E((u, v)) = \ell_E((v, u))$.
With this, graphs hold substantial expressive potency and flexibility. They offer themselves to depict complex data and their relations in a concise way. And while finding the shortest distance between nodes within a weighted graph is well understood, the same cannot be said about the shortest distance between graphs. Given their flexibility even defining *what* distance means is not trivial, so much so that it is its own problem formulation.

Definition 2.2 (Graph Comparison Problem)**.** Given g_1 and g_2 with $g_1, g_2 \in \mathcal{G}$, the graph comparison problem is finding a function

$$d : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$$

so that $d(g_1, g_2)$ quantifies the dissimilarity of g_1 and g_2

This problem formulation is frequently reframed as a *graph matching* task, where the goal is to find a substructure preserving correspondence between the nodes and edges of g_1 and g_2. Subsequently the similarity measure $d(g_1, g_2)$ is a by-product of quantifying the governing graph matching task.

In particular, *inexact graph matching* [24][25] (also called *error-tolerant graph matching*) is capable of generating a correspondence and thus a similarity score, even for entirely dissimilar graphs. Various approaches to error-tolerant graph matching have been introduced, such as graph embedding or graph kernel methods. Graph embedding methods explicitly map graphs into a latent vector space, enabling the use of distance measures defined on vector spaces [26]. Similarly, Graph kernels $k : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ implicitly map $g \in \mathcal{G}$ into a *Hilbert space* $\mathcal{H}$ [27] via $\phi(g)$, a metric space with respect to distance function induced by the inner product, on which a distance between graphs is easily calculated via $k(g_1, g_2) = \langle \phi(g_1), \phi(g_2) \rangle_{\mathcal{H}}$ [28].

[1] for a comprehensive definition of subgraphs consult the literature [22]

An error-tolerant graph matching method that has become increasingly popular is the *graph edit distance*, that formulates the graph comparison problem as finding an *edit path*, a set of elementary graph transformations with minimal cost to transform a source graph into the target graph, while staying in the graph domain.

Definition 2.3 (Graph Edit Distance)**.** Let $g_1 = (V_1, E_1, \ell_{V_1}, \ell_{E_1})$ be the source graph and $g_2 = (V_2, E_2, \ell_{V_2}, \ell_{E_2})$ the target graph, with $g_1, g_2 \in \mathcal{G}$. The graph edit distance $GED(g_1, g_2)$ between the two given graphs is defined as,

$$GED(g_1, g_2) = \min_{\{e_1,\ldots,e_k\} \in \mathcal{T}(g_1,g_2)} \sum_{i=1}^{k} \mathcal{C}(e_i)$$

where $\mathcal{T}(g_1, g_2)$ denotes the set of all complete edit paths that transform g_1 into g_2, with $\mathcal{C} : e_i \to c$ denoting the chosen edit cost function where $c \in \mathbb{R}$ quantifies the effort required for applying e_i to g_1. Note that a sequentially constructed edit path is no longer permutation invariant.

A substantial benefit of the GED, compared to implicit or explicit embedding methods, is its flexibility regarding node and edge labels, since by staying in the graph domain no information is lost. The trade-off is increased time complexity of GED techniques, scaling exponentially with the order of the input graphs. This, paired with it being a NP-hard problem, makes exact GED computation intractable for larger graphs and simultaneously attractive for approximate solutions. Several methods to approximate the GED have been introduced over time, capable of producing close GED estimates in polynomial time. However they usually introduce additional constraints to the problem, such as directly learning the GED metric without producing a viable edit path [12], reducing the task to a pure node level matching where edges are considered intrinsic to their incident nodes [11], or allowing GED estimates below the true GED [29].

Within recent years, substantial improvements in the field of geometric deep learning, particularly concerning GNNs [30], were brought forward. For example, the renowned Graph Convolutional Network (GCN) paper by Kipf & Welling [31] and other subsequent GNN architectures [32], a variety of GNN-based solutions to solve the problem of GED approximation have been proposed, all harnessing their potential to achieve substantial performance increase regarding either speed, accuracy or ideally both. However, these strides are mostly tied to supervised [33] or self-supervised [34] machine learning (ML) paradigms, usually with a well curated dataset suitable for the given task as a prerequisite. Other ML paradigms such as RL are comparatively underexplored, even though they have been applied to graph

problems of demonstrably similar complexity, for example the travelling salesman problem [35], resource management [36] or ML compiler graphs [37].

2.2 Reinforcement Learning

The principal idea of RL has its origins in early 20th century behavioural psychology, specifically in early operant conditioning by Edward Thorndike who empirically examined the *law of effect* and described it as "that what comes after a connection acts upon it to alter its strength" [38]. Thorndike thereby laid the foundation for the subsequent work of B.F Skinner [39] who established the standard framework of operant behaviourism revolving around "reinforcement" and "punishment".
In the latter half of the 20th century, Sutton & Barto then used the term "reinforcement learning" for the first time in a ML setting, where primitive neural networks were applied to optimal control problems such as vertically balancing a pole on a movable cart for as long as possible[2] [40].

This gives rise to the RL paradigm in its current form as depicted in Figure 2.1, revolving around an intelligent *agent* taking *actions* in a dynamic *environment*, receiving a *reward* that serves as the reinforcing component for its actions. Through continuously interacting and exploring the environment, the agent ideally learns a behaviour that maximizes the rewards it receives.

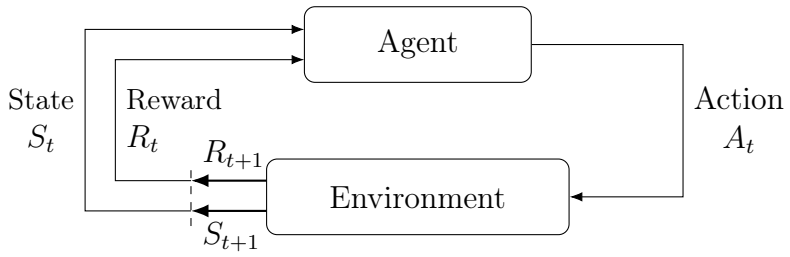


Figure 2.1: General form of an RL system and its feedback loop. Consisting of an agent selecting an action, and an environment returning a new state and reward.

The naming conventions in RL are inherited from the terminology used for Markov decision processes (MDPs), since early RL was primarily applied to solve MDPs of classic control theory, like the cart-pole problem. Particularly, consider an agent to interact in discrete time steps with the environment, selecting actions $A_t \in \mathcal{A}$ from the action space $\mathcal{A}$, receiving a new state $S_{t+1} \in \mathcal{S}$ from the state space $\mathcal{S}$ and a scalar reward $R_{t+1} \in \mathbb{R}$. A sequence of such states, actions and their associated rewards is called a *trajectory* $\tau = (S_0, A_0, A_1, S_1, \ldots)$. Trajectories can be either

[2] also known as the cart-pole problem

finite or infinite in nature. Consequently, one can define the *discounted return* of such a trajectory as $\mathcal{R}(\tau) = \sum_{t=0}^{T} \gamma^t R_{t+1}$ the sum of future discounted rewards, with $\gamma \in [0, 1)$ being the discount rate. Note that the special case of $\gamma = 1$ is equivalent to an undiscounted return, which for $T = \infty$ might no longer converge to a finite value. Within the scope of this thesis however, only finite trajectories with $T < \infty$ are of relevance.

The agent's component responsible for taking each action is its *policy* π, a function formally defined as $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ with $\pi(s, a) = Pr(A_t = a \mid S_t = s)$. A policy maps a state-action pair (s, a) to the probability of selecting the specific action a when the agent finds itself in state s. For ease of use, the short form $a \sim \pi(\cdot|s)$ describes the sampling i.e. selection of a, given the conditional probability distribution of π in state s. It should be noted that π can refer to any arbitrary policy used for action selection, thus giving no intrinsic guarantee about the quality of the actions selected by the agent. Thus, the concept of an *optimal policy* denoted by $\pi^* = arg\ \max_{\pi} \mathbb{E}_{\tau\sim\pi} [\mathcal{R}(\tau)]$, defined as the policy such that trajectories sampled from it provide the maximum expected return, is required.

Ultimately, the aim of any agent is to learn such an optimal policy through explorative interaction with its environment. *How* the agent does so is the RL problem to be solved. Following are some frequently used tools to guide the agent towards optimal behaviour.

Definition 2.4 (Value Function)**.** Let $\mathcal{R}(\tau)$ be the return for the sampled trajectory τ while following policy π, given the trajectory starts in state s. Call,

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi} [\mathcal{R}(\tau) \mid S_0 = s]$$

the *state value* of s as the expected return while following the policy π, with V^{π} the *state value function* under policy π, commonly shortened to *value function*.

This term serves as a quantifier for how 'good' it generally is to be in a specific state s while behaving according π. While reasonable as an estimator, the actual return $\mathcal{R}(\tau)$ might deviate substantially from the expected value, depending on the specific actions chosen in the trajectory. One can naturally expand the concept of the value function to specifically include the actions as well.

Definition 2.5 (Action Value Function)**.** Let $\mathcal{R}(\tau)$ and π be the same as in Definition 2.4 but require for τ to contain the starting action a in addition to the starting state s.

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau\sim\pi} [\mathcal{R}(\tau) \mid S_0 = s, A_0 = a]$$

Is then called the *state-action value* of the pair (s, a) and Q^π the *state-action value function* under policy π, shortened to *action value* and *action value function*, respectively. The function gives the expected return for starting in state s, taking initial action a, then continue acting according to π.

Thus, both how favourable it is to be in a specific state s as well as how favourable it is to take a specific action a in said state, can be quantified. While either function is potent enough on their own to have agents learn various tasks [41], they have particular advantages and disadvantages. For example, the increased granularity of the action value $Q^\pi(s, a)$ over the state value $V^\pi(s)$ comes at the cost of increased variance and subsequent instabilities in the agent's behaviour. Combining both provides the opportunity to attain the desired level of granularity of $Q^\pi(s, a)$, while also retaining the lower variance of $V^\pi(s)$.

Definition 2.6 (Advantage Function)**.** Let V^π and Q^π be as in Definition 2.4 and Definition 2.5. Denote the difference between the action value and state value,

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

as the *advantage* of (s, a), with A^π being the *advantage function* under policy π. The advantage directly quantifies the performance difference of the specific action a relative to the expected performance when sampling an action $a \sim \pi(\cdot|s)$, given state s.

In reality, the computational costs for Q^π in particular can become intractable for problems with sufficiently large $\mathcal{S}$ and $\mathcal{A}$. Thankfully, *bootstrapping* i.e. using current estimates as an approximation for future, unobserved estimates the action value function can be eliminated from the equation.

Definition 2.7 (Temporal Difference Error)**.** Let V^π be a value function, with γ the discount rate.

$$\delta_t = \underbrace{R_{t+1} + \gamma V^\pi(S_{t+1})}_{\approx\, Q^\pi(S_t, A_t)} - V^\pi(S_t)$$

Denotes the *temporal difference error* (TD error), with $R_{t+1} + \gamma V^\pi(S_{t+1})$ called the *temporal difference target* (TD target). The TD target is a bootstrapped approximation for $Q^\pi(S_t, A_t)$.

Thus, the TD error serves as an approximation $\delta_t \approx A^\pi(S_t, A_t)$ for the advantage. The aforementioned benefits of A^π are retained, albeit potentially introducing additional bias and compounding errors depending on the accuracy of V^π.

Note that all functions are dependent on π, thus the concept of optimality for a policy extends to these functions. Such functions are called optimal, when the underlying policy is itself optimal to begin with. In particular, $V^* = V^{\pi^*} = \max_{\pi} V^\pi$ denotes the *optimal* value function, with equivalent definitions for Q^* and A^* (not to be confused with A^*-search). This causes a circular problem, where the thing one wishes to find (π^*), and the thing to find it with (V^*), are mutually dependent on each other. An approach originating from policy iteration in dynamic programming (DP), splits this dependency into two interacting processes called *policy evaluation* and *policy improvement*. The former aligns the estimator function E with current policy π, the latter aligns π to behave greedy, i.e. return maximizing, with respect to E. This process is called *generalized policy iteration* (GPI) and is depicted in Figure 2.2. While the specific implementations for policy evaluation and policy improvement can vary, most RL methods align with the concept of GPI. Figure 2.4 provides an overview of deep RL methods. There is a class of problems with state and action spaces so large, dimensionality reduction techniques become mandatory. Luckily, one can use parametrisation so that $\pi_\theta \approx \pi$ and $V^\pi_\phi \approx V^\pi$, where θ and ϕ denote learnable neural network parameters. The cost of dimensionality reduction via parametrisation is loss of generality, since $\pi_\theta \approx \pi \implies \pi^*_\theta \approx \pi^*$. An optimal parametrised policy might only provide a local optimum, while π^* can guarantee a global optimum. However, finding exact — that is globally optimal — solutions for NP-Hard problems quickly becomes impossible, thus finding a local optimum via π^*_θ is a viable alternative.

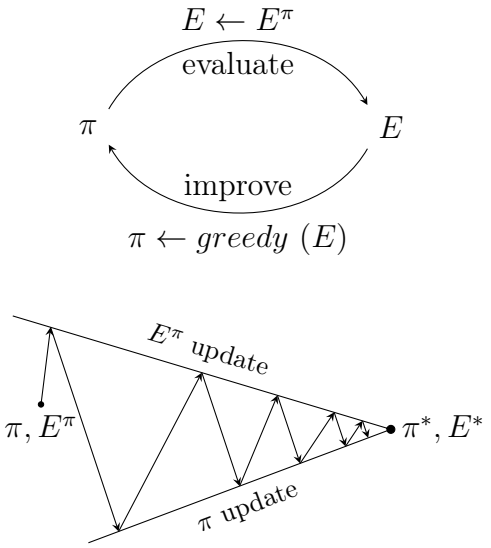


Figure 2.2: GPI process, where π is the policy and E the target estimator function (typically V, Q or A) [23].

Various such techniques for *policy optimisation* exist, for example evolution strategies (ES) [42] and genetic algorithms (GA) [43]. However, the most common technique utilizes gradient descent [44] adapted to policies, aptly called *policy gradient* algorithms. The *actor-critic* subcategory of policy gradient algorithms closely follows the GPI framework, with the policy π_θ as the *actor* and the estimator function E as the *critic*. More specifically the *advantage actor-critic* (A2C) algorithm depicted in Figure 2.3, which utilizes the bootstrapped advantage δ_t, will serve to illustrate policy gradient methods. In the context of policy gradient methods, the

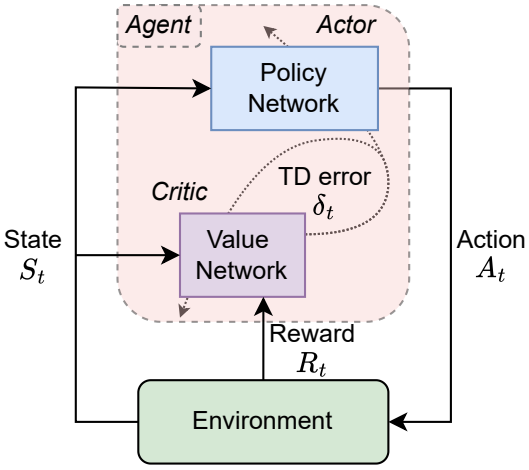


Figure 2.3: A2C-RL diagram showing the actor and critic components of the Agent and the use of δ_t to improve their respective policy and value networks.

expected return $\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\mathcal{R}(\tau)]$ of the parametrised policy π_θ is often denoted by $J(\theta)$, rarely by $J(\pi_\theta)$ in certain literature. Thus, the fundamental formula for gradient ascent on policies becomes,

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta_k)$$

with α denoting the learning rate and $\nabla_\theta J(\theta_k)$ the *policy gradient*, so that parameters θ converge to values that (locally) maximize $J(\theta)$.
Calculating $\nabla_\theta J(\theta)$ requires $\nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\mathcal{R}(\tau)]$ to be differentiable, which is not trivial. Omitting the rigorous mathematical argumentation, the solution for calculating this gradient is based on the "log-derivative trick". Though called a "trick", it is the chain rule for logarithms rearranged as $\nabla_\theta \pi_\theta(\cdot|s) = \pi_\theta(\cdot|s)\nabla_\theta \log \pi_\theta(\cdot|s)$. This allows the gradient of an expectation to be expressed as an expectation of the gradient[3].

Definition 2.8 (Policy Gradient)**.** Thus denote the general form of the policy gradient $J(\theta)$ as,

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t) \cdot \Psi_t\right]$$

with Ψ_t being a placeholder for any return such as $\mathcal{R}(\tau)$. For the A2C, Ψ_t normally refers to the Generalised Advantage Estimation (GAE) [45].

Other algorithms like proximal policy optimisation (PPO) and group relative policy optimisation (GRPO) adapt and expand upon this general form. They instead

[3] using the Leibniz integral rule

use a clipped surrogate advantage that measures the performance of π_θ relative to the old policy π_{θ_i} via $\frac{\pi_\theta(a,s)}{\pi_{\theta_i}(a,s)}$. This behaves equivalently to the policy gradient in Definition 2.8 for $\theta = \theta_i$.

Definition 2.9 (Generalized Advantage Estimation)**.** Let δ_t be the TD error introduced in Definition 2.7, with $\gamma \in [0, 1]$ the discount factor and λ the *trace decay*. The advantage estimate $A_t^{\mathrm{GAE}(\gamma,\lambda)}$ at time t is defined as an exponentially decaying sum of TD error terms from t until the end of the trajectory at time T:

$$A_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l},$$

The trace decay λ originated from the TD(λ) algorithm [46] and is used to interpolate between returns from Monte Carlo (MC) methods when $\lambda = 1$, and one-step TD methods when $\lambda = 0$. In practice, it is reasonable to treat λ as a balancing term between variance (MC methods) and bias (TD methods) when $\lambda \in (0, 1)$.

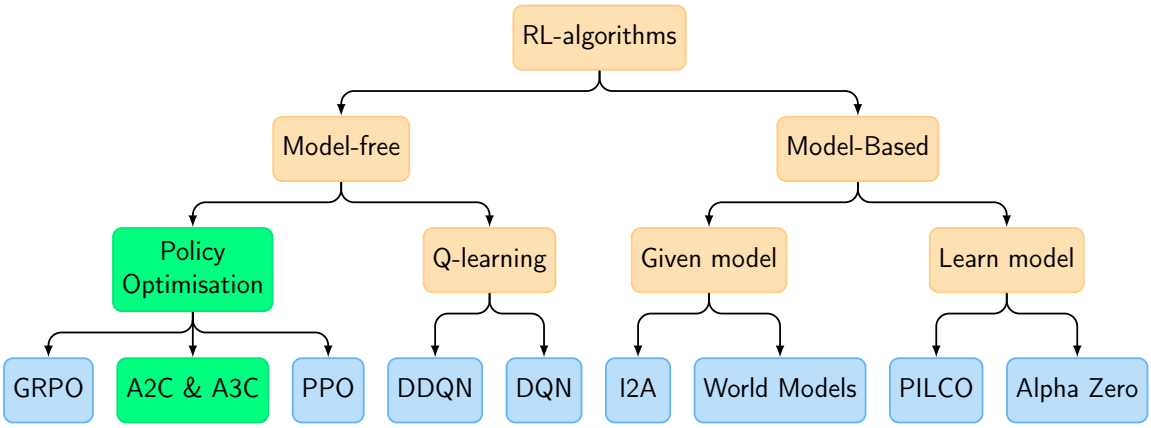


Figure 2.4: Taxonomy of current RL algorithms, with focus on policy optimisation methods.

2.3 Combining RL and GED

Having introduced both topics, the question arises *why* one should apply RL to a problem like approximating the GED. First, using RL for GED approximation is underexplored compared to other ML approaches. Second, the problem of exact GED computation being NP-hard encourages development of methods to find near optimal solutions. Reinforcement learning excels at such optimisation tasks and has been successfully applied to other NP-hard graph problems. Third, the core strength of RL is sequential decision problems with long-term dependencies. In seemingly unsolvable games like Go and Chess, humans were substantially outperformed by

comparatively unrefined RL models. There is precedent for treating the GED as a sequential decision problem in the case of applying A^*-search to a GED search tree. An example of such depicted in Figure 2.5. Components of the GED in Definition 2.3

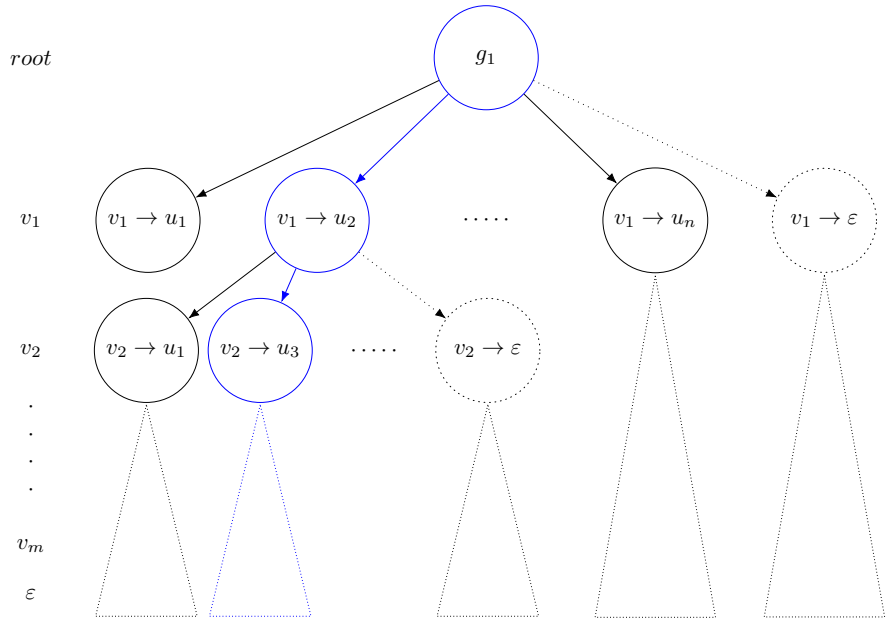


Figure 2.5: Example of a GED search tree for transforming g_1 into g_2, with nodes marked in blue depicting the selected edit path.

can be made to conform to RL terminology with some reformulation. Particularly the edit path, $\{e_1, \ldots, e_k\} \in \mathcal{T}(g_1, g_2)$ can be interpreted as a trajectory τ via $((S_0, A_0), (S_1, A_1), \ldots, (S_{k-1}, A_{k-1})) \equiv (e_1, \ldots, e_k)$. Enforcing the edit operations to be ordered, an approximately equivalent GED definition in RL would then be,

$$\min_{(e_1,\ldots,e_k)\in\mathcal{T}(g_1,g_2)} \sum_{i=1}^{k} \mathcal{C}(e_i) \iff \min_{\tau \sim \pi_\theta} \sum_{i=0}^{k-1} \mathcal{C}'(S_i, A_i)$$

with the optimal policy to be found defined as,

$$\pi_\theta^* = arg \min_{\pi_\theta} \mathbb{E}_{\tau\sim\pi_\theta} \left[\sum_{i=0}^{k-1} \mathcal{C}'(S_i, A_i) \right]$$

Note that the function $\mathcal{C}'$ can incorporate other parameters besides the edit cost if necessary. Finally, RL is a paradigm and therefore one is not confined to a specific architecture. This allows incorporating more established neural network solutions for the GED task, such as GNNs. In addition, the straightforward adaptation of the GED problem to a RL problem retains the major benefits of using the GED as a metric, particularly when concerned with interpretability and explainability of the found solutions as true edit paths.

Chapter 3

Methods

In the following chapter, the main focus will be on the detailed exposition of implemented components, allowing a holistic picture of their advantages and drawbacks. With no preceding reference literature, certain decisions will inevitably lack comprehensive justification. For such cases, a selection of alternatives will be outlined to enable an informed choice in case the implementation has to be adapted to respect additional constraints.
Structurally, the chapter will be comprised of three main sections, covering the essential components of Figure 2.3. Section 3.1 deals with the details of the environment, particularly the choice of graph datasets, components comprising the state and action space, and the intricacies of the reward signal. Section 3.2 deals with the agent, focusing on the chosen RL algorithm and its policy and value network components, as well as the loss function and reward shaping options. Section 3.3 shows how these components interface, focusing on the interacting components, and outlines the general training process used.

3.1 Environment

The environment is one of the two central pillars of RL. The selected tools influence design decisions of other RL components. For most RL tasks beyond the most basic, for which pre-made environments exist, a custom implementation of an environment specific to that problem is required. In order to prevent a multitude of incompatible custom environment implementations, OpenAI released OpenAI Gym [47], an open source toolkit for developing and evaluating RL environments, with maintenance continued by the Farama foundation on its replacement package 'gymnasium' [48]. This API is widely used and compatible with the majority of downstream libraries that implement RL-algorithms [49]. Consequently, for enhanced downstream compliance, it is reasonable to adhere to the gymnasium API when implementing the

```python
import gymnasium as gym
from gymnasium.spaces import Space

class CustomEnv(gym.Env):
    def __init__(self, *args, **kwargs):
        self.action_space = Space[ActType]
        self.observation_space = Space[ObsType]
        #...
    def step(self, action):
    def reset(self):
    def render():
    def close():
```

Code 3.1: Minimal requirements of the gymnasium API when defining a custom environment.

GED environment. Due to its API requirements, shown in Code 3.1 being minimalistic, the design freedom for developers is vast. The `__init__()` function can be expanded with various parameters that are passed to `gym.make()` during initialisation of the environment. For representing and manipulating graphs in Python, the NetworkX package [50] is a popular choice due to its flexibility, ease of use, and great integration with other Python libraries. Specifically the implementation of graph edit operations is considerably easier with NetworkX compared to other packages like igraph or graph-tool. These might be computationally faster, but the performance difference is likely negligible.
Another central task of the environment is managing the dataset. The TUDataset collection contains frequently used datasets for GED tasks. Since they are real-world datasets, the control over graph properties is limited, which is the model is designed and tested on a synthetic dataset first. Specifically on the graph atlas dataset provided by NetworkX via `nx.graph_atlas()`, which contains all graphs with up to seven nodes [20]. These graphs can then be given artificial node and edge labels, providing more control to gain insights into the agent's behaviour. Of course the environment would ideally be able to cope with any arbitrary dataset at some point. A prerequisite is a uniform representations of graph pairs as states.

3.1.1 Observation Space

First, a reminder of 2.2, where the concept of MDPs and the state space $\mathcal{S}$ were introduced. For MDPs, a generalisation called *partially observable Markov decision processes* (POMDPs) exists, with most modern RL utilisations falling into that category. IN POMDPs the agent is not privy to the entire state of the environment, for

example in cases where the environment is the real world. Thus, the agent receives an *observation* $O_t \in \mathcal{O}$ from the observation space $\mathcal{O}$ rather than the entire state S_t. In practice this translates into O_t being a digestible representation of S_t containing the necessary information for the agent to perform well. Thankfully, the field of graph theory provides reasonable graph representations, such as the adjacency or Laplacian matrix. These serve as a computationally inexpensive starting point and can be easily shaped into a neural network digestible format, however they are not well suited to encode labelled graphs. More sophisticated methods, like Node2vec or GNNs which are specifically designed to handle labelled graphs are therefore more appropriate, even if computationally more intense. The observation is not complete with just embedding the source and target graph. The applied graph edit operations are crucial information for the agent's decision making. A proven method for generating sequence embeddings are recurrent neural networks (RNNs). They provide a good compromise between more potent methods like transformers, and simple concatenation of edit operations. Naturally, one can consider utilizing transformers if the need arises.
The RNN and GNN embeddings form the core observation, encoding a reasonable amount of state information — namely the current position, a partial trajectory, and the target — for the agent. Gymnasium allows custom observation spaces by extending the `gymnasium.spaces.Space` class, but it is discouraged since certain utility functions provided by gymnasium might not handle custom spaces correctly. In addition, gymnasium already provides fundamental spaces like `spaces.Box()` and `spaces.Discrete()` as well as composite spaces like `spaces.Tuple()` and `spaces.Dict()`. These built-in spaces should cover most use cases and implement all useful utility functions. For example, the RNN and GNN embedding vectors can be represented using `spaces.Box()` as shown in Code 3.2, which represents a Cartesian product of n closed intervals in $\mathbb{R}^n$. These classes (and more) are also used for the action space of the environment.

```
self.observation_space = Box(low=-np.inf,
                             high=np.inf,
                             shape=(len(obs),),
                             dtype=np.float32,
                             )
```

Code 3.2: Box class representing the observation space as a vector in $\mathbb{R}^n$, with n the size of the observation `len(obs)`.

3.1.2 Action Space

To define the action space, the components of a single action need to be declared first. The actions are discrete graph edit operations, specifically insertion, deletion, and relabelling of nodes and edges. The action space becomes combinatorially large if the number of input nodes and edges becomes large enough. This is problematic for the agent downstream, as dealing with a discrete action space larger than a couple thousand actions becomes a detriment.
For example, for Starcraft 2 [51][52] tools like pointer networks and autoregressive policies have been used to tackle this issue. The simpler option, still potent enough to learn Dota 2 [53], is decomposing the action space and using a hierarchical approach to action selection. As depicted in Code 3.3, graph edit operations can be readily decomposed into individual components, since node and edge operations are mutually independent. This dimensionality reduction lets the action space grow linearly rather than combinatorially. Decomposition also makes dynamically generating the action space easier, since the number of possible node and edge labels can vary for each feature. This serves as the initial approach to ensure that the action structure remains interpretable and easy to implement. Alternatives for dimensionality reduction are action embeddings, for example via autoencoders, which allow generalisations over actions in the embedding space. The drawbacks of action embeddings are increased training complexity, lack of interpretability, potential instability, and increased sensitivity to hyperparameters. These are exacerbated when used in junction with stochastic policy gradient methods like A2C.

```
self.action_space = Tuple((Discrete(<edit operations>),
                           Discrete(<nodes>),
                           DIscrete(<edges>),
                           Discrete(<node labels>),
                           Discrete(<edge labels>),
                           #... Additional node or edge feature labels
                           ))
```

Code 3.3: General specification of the action space used, dynamically expandable with further node or edge features.

3.1.3 Reward Signal

While both observation and action space are crucial, the sole component responsible for guiding the agent's desired behaviour is the reward. Finding an edit path suffers from a sparse reward objective, where rewards for actions are zero until the relevant

target graph is reached. Only then is the action sequence positively determined as an edit path and a positive reward can be distributed. Luckily, one can consult other pathfinding problems with sparse rewards as reference to construct a goal-oriented reward. For example gridworld [54] and maze problems also exhibit sparse reward structures, learning to traverse a 2D space from point A to point B. The use of *reward shaping* — the introduction of smaller, more immediate auxiliary rewards that align with the true goal — provides a more dense reward structure i.e. feedback, which accelerates the agent's learning. For gridworlds, this culminates in small negative rewards (penalties) per step, with larger penalties for illegal actions or collisions, and positive rewards for reaching the goal. The drawback of reward shaping is introducing unintended behaviour, as additional rewards or penalties unrelated to the underlying task can derail the learning process. It should therefore be used sparingly and only if learning with sparse rewards is intractable.
Other options for reward shaping is the use of heuristics. In chess for example, a heuristic generated from Monte Carlo tree search (MCTS) is used to guide the agent. For even more complex games like Dota 2, both heuristics like gold per minute as well as subgoals like destroying enemy structures are applied. Finding a GED edit path lies somewhere between gridworld and Dota 2 in terms of complexity. A reward that incorporates a small step cost, a larger cost for illegal actions and a distance heuristic $R_t = -(c_{step} + c_{illegal} + h_{GED}) + r_{goal}$ is therefore a reasonable concept. The term h_{GED}[1] is similar to $h(x)$ in the $f(x) = g(x) + h(x)$ formulation of A^*-search.

3.2 Agent

The agent is the other central pillar of RL. Given an adequate environment, its job is to explore said environment and learn to solve the given task. As discussed in Section 2.2 policy gradient algorithms are simple, yet powerful and can be applied to a wide range of tasks. The combination of simplicity and performance make them a solid initial choice. As for the environment, there exist libraries with efficient implementations for most state-of-the-art RL aglorithms. The upfront benefit of pre-implementation comes with the drawback of limited customisation and slower iteration speed when adapting them to a specific task. Thus, given the novelty of using RL to find an graph edit path, it is reasonable to implement from scratch. For the algorithm, A2C as a PPO predecessor is a reasonable choice due to its simplicity. One can still upgrade to PPO later on if the need arises.
PyTorch and PyTorch Geometric (PyG) are used for the custom implementation

[1] consult Table A.1 for more details

as they provide maximum flexibility, good debugging tools and fast iteration speed due to its dynamic computation graph, making it the preferred libraries in research applications. For the actor-critic architecture shown in Figure 2.3, the actor and critic — that is the value and policy network — will be implemented as two separate networks allowing for better control. A joint network can be an alternative if training speed and computational overhead poses an issue.

3.2.1 Value Network

Various value functions have been discussed in Chapter 2, particularly the bootstrapped advantage Definition 2.7 and the GAE are of relevance. The policy network, also called the critic, guides the actor towards selecting reward maximizing actions. For simple problems like cart-pole or gridworld, even shallow networks of a few fully connected layers suffice to solve them. Naturally, more complex tasks prefer deeper networks and might require additional layer types for better normalisation, regularisation and feature extraction.
The architecture consists of a deep, fully connected network depicted in Figure A.8, returning a scalar $V^{\pi_\theta}(o)$ with the observation as input. Finding the optimal network configuration of hidden size and activation functions are subject to experimentation. The reduction from `len(observation)` to a scalar, suggests gradually reducing the hidden layer sizes similar to the encoder component of an autoencoder. For activation functions, the rectified linear unit (ReLU) and its variants are the standard. Since the predicted state value can be negative, leaky ReLU or parametrised ReLU (PReLU) are superior choices. These adapt the standard $max(0, x)$ ReLU to $max(\alpha x, x)$ with α being a specifiable hyperparameter for LeakyReLUs and a learnable one for PReLUs. A possible network adaptation is introducing additional value heads, so that $V^{\pi_\theta}(o_{edit})$ represents the value of a specific edit operation rather than the overall expected value of $V^{\pi_\theta}(o)$.

3.2.2 Policy Network

In comparison, the policy network tends to be significantly more complex than the value network, as the centrepiece of most RL architectures. Since its output is not a simple scalar but an action. The general form of an action was illustrated in Code 3.3. To mirror this action structure in the policy network, a separate action head for each action component is used. The output of each action head are logits of the same shape as required by the environment's action space. These logits are transformed into discrete distributions from which the action components are sampled and the log probability required for the policy gradient is calculated. The

```
#... for each action head
action_dist = torch.distributions.Categorical(action_head_logits)
action = action_dist.sample()
action_log_prob = action_dist.log_prob(action)
#...
```

Code 3.4: Process of sampling a partial action from each action head output.

dependency of some action components on the output of upstream action components can be problematic. For Example, the chosen node or edge will likely differ depending on which graph edit operation is selected. Similarly, certain components like the chosen edge has no impact if the graph edit operation is performed on nodes, and vice versa. These dependencies can be modelled using a hierarchical architecture. This hierarchy encompasses three distinguishable levels. First the observation level, that requires only the observation as input, such as the edit operation. Second is the graph level, that require both the observation and the graph edit operation as input, such as the node and edge action heads. Third is the feature level, containing components that require the observation and selected graph component as input, as is the case for all the node and edge features. For instance, a conditional probability $P(node \mid obs,\ edit\ op)$ is reflected in code as `node_head(torch.cat([obs, edit_head_output]))`. Note that within the network, both input and output are logits.

While not perfectly representing the real conditional probabilities, this approach allows actions to be sampled as depicted in Code 3.4, letting the actor implicitly learn the conditional relationships. This culminates in the architecture shown in Figure 3.1. Which can be expanded with additional feature heads if required. The edge selection can be split into two sequential heads, where one head selects the source node v_s and the other the target node v_t. This might be of interest when working with directed graphs, while for undirected graphs v_s and v_t are sampled simultaneously. The node and edge embeddings act as separate layers for extracting features from the observation for their respective downstream action heads.

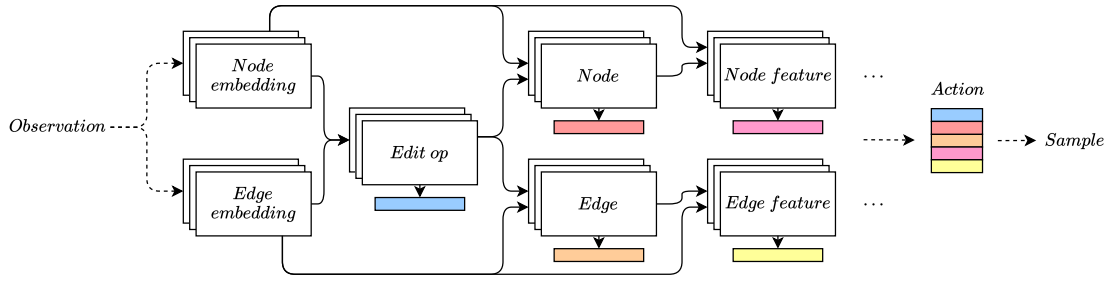


Figure 3.1: Diagram outlining the structure of the agent and its action composition.

3.3 Combined Architecture

With a thorough understanding of the agent and environment in isolation, their interaction can be assessed. An element that shapes this interaction is the RNN and GNN. Ideally, their network parameters would be updated in an end-to-end fashion, together with the parameters of the policy and value network. This is somewhat tricky, since we have no separate loss to calculate for the RNN and GNN. The loss to train the RNN and GNN needs to express the positive or negative deviation of the agent's behaviour with respect to the observation, which is difficult to measure directly. It is considerably simpler to train the RNN and GNN based on the loss for the agent. Their parameters should therefore change to implicitly align with the objective of maximizing the expected return.

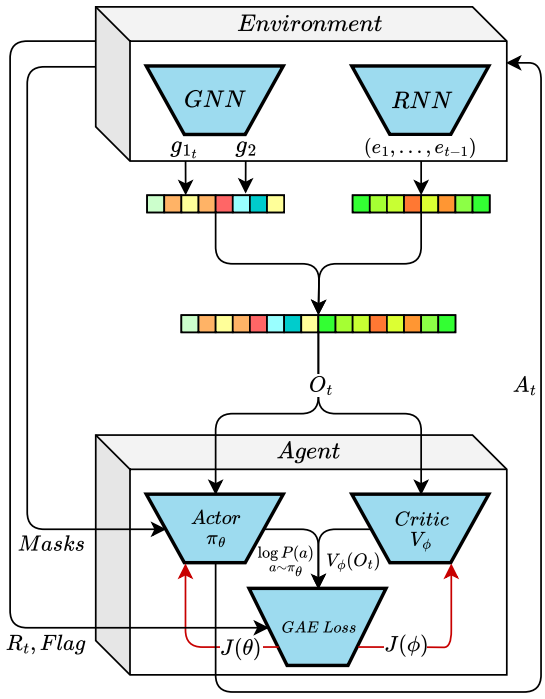


Figure 3.2: The combined model architecture, displaying the interacting components of agent and environment in detail.

Other elements that are sent to the agent by the environment are depicted in Figure 3.2. The `truncated` flag is set if the trajectory terminates successfully, while `terminated` is set if the trajectory terminates successfully. Note that the environment will not set these flags automatically, but rather are responsibility of the developer. Also useful is the ability to pass along auxiliary information together with O_t via `env.step()` and `env.reset()`. This is done using a `dict()` object named info, which contains additional relevant data for the agent, concerning the internal state of the environment, for example the GNN and RNN parameters. This info can also be passed on in each `env.step()`, such as action masks. While there is an argument for letting the agent learn to avoid actions rather than preempting their selection, if the error is trivial, it arguably contains no valuable information for the agent to learn from. There is definitely a limit to how much the agent's freedom should be restricted, as this can also influence its exploratory behaviour. As an extreme example, given a pair of graphs, one could only allow actions if they are part of a known, viable edit path. This would undermine the role of the agent, rendering the use of RL pointless.

The combination of all these components, provides the training cycle depicted in Algorithm 3.1

Training Loop

Initialize environment: Atlas dataset, network parameters β_{GNN}, β_{RNN}, state S_0 and state embedding O_0
Initialize agent: Actor $\pi_{\boldsymbol{\theta}}(\cdot|o)$ and critic $V_{\boldsymbol{\phi}}(o)$ with parameters $\boldsymbol{\theta}$, $\boldsymbol{\phi}$, with β_{GNN}, $\beta_{RNN} \subset \boldsymbol{\phi}$
Initialize: Cache to track data of training performance
Parameters: Learning rates: $\alpha^{\pi_\theta}, \alpha^{V_\phi}$, trace decay: λ, discount rate: γ, entropy coefficient: ε
loop for each episode:
    Initialize: Storage tensors for current episode
    while not (terminated **or** truncated) **do**
        $a \sim \pi_{\boldsymbol{\theta}}(\cdot|o), V_{\boldsymbol{\phi}}(o), \ldots \leftarrow$ agent selects action
        $r, s', o' \ldots \leftarrow$ environment applies action
        $T_r, T_a, T_{V_\phi(o)}, \ldots \Leftarrow r, a, V_\phi(o), \ldots \leftarrow$ store outputs in tensors
        $s \Leftarrow s'$
        $o \Leftarrow o'$
    End of trajectory
    $J(\boldsymbol{\theta}), J(\boldsymbol{\phi}) \leftarrow$ loss calculation for actor and critic
    $\boldsymbol{\theta_{k+1}} \Leftarrow \boldsymbol{\theta_k} + \alpha^{\pi_\theta} \nabla_\theta J(\boldsymbol{\theta}) \leftarrow$ actor update
    $\boldsymbol{\phi_{k+1}} \Leftarrow \boldsymbol{\phi_k} + \alpha^{V_\phi} \nabla_\phi J(\boldsymbol{\phi}) \leftarrow$ critic update
    store trajectory rewards, actor loss, critic loss etc. in cache
analyse and plot collected data for evaluation
...

Algorithm 3.1: General structure of the training loop.

Chapter 4

Experimental Evaluation

The following chapter dedicates itself to quantitative and qualitative analysis of the proposed system and its components. The vast number of adjustable variables ultimately renders extensive testing of every potential combination unfeasible. In Section 4.1 the occurrence and proposed solution for the circular relabelling problem is discussed. Then, Section 4.2 covers a selection of parameter combinations, their influence on approximating the GED, and examines the differences between specific RNN and GNN combinations for embedding the state. Finally, Section 4.3 highlights some graph pairs for which an edit path has been successfully found, and potential patterns within the set of solved pairs.

4.1 Circular Relabelling

As mentioned in Section 3.1.3, approximating the GED via an edit path suffers from sparse rewards. The agent receives no viable feedback if a trajectory does not produce an edit path. The proposed reward shaping mitigates that issue, providing the agent with dense rewards to learn from. Given the simplicity of the shaped reward structure, it proved susceptible to reward hacking. Reward hacking or specification gaming refers to an RL agent directly optimising the objective function created by the shaped rewards, entirely avoiding the intended task in the process. This occurs if the agent manages to exploit specific loopholes and shortcuts that are present in the environment. For example, a robot tasked to grab an object while observed by a camera would learn to position its hand in such a way as to pretend it was grabbing the object relative to the observer [55]. Similarly an agent tasked to learn CoastRunners — a boat racing game where the player gets points by hitting targets along the course — would indefinitely move in circles to collect these targets, rather than finishing the race [56].
Particularly the latter resembles *circular relabelling*, the problem created due to

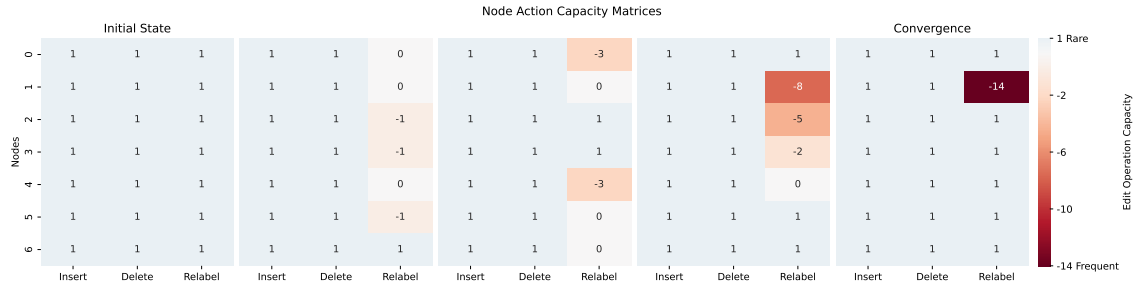
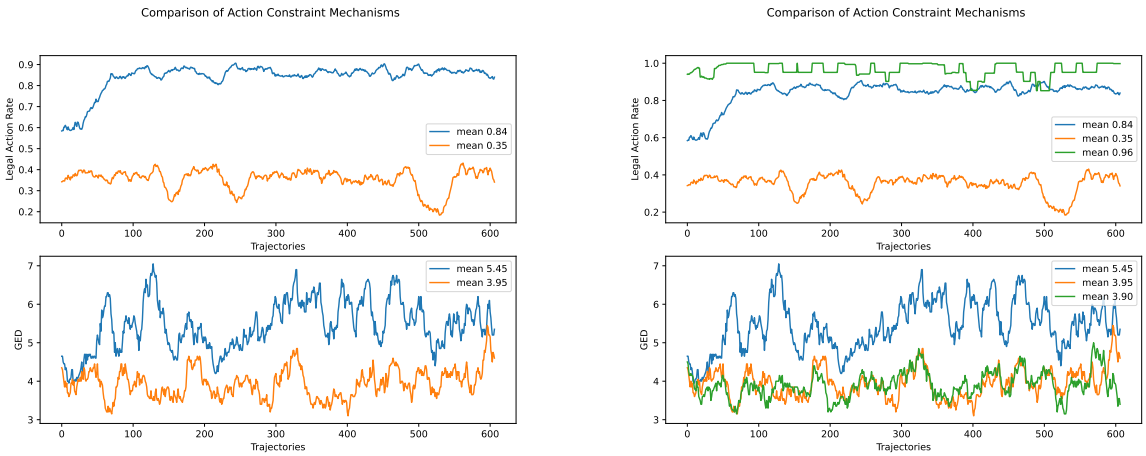


Figure 4.1: Progression of circular relabelling throughout a sample run, reflected in the convergent action capacity matrix.

shaping the reward structure. Circular relabelling refers to the agent's process of maximizing its reward, by finding a node present across all graphs and continuously relabelling that node. These graph edit operations do not contribute to reducing the GED between a given graph pair. However, when reminded of the reward structure $R_t = -(c_{step} + c_{illegal} + h_{GED}) + r_{goal}$ the reward hacking becomes evident. The agent's behaviour reduces it to $R_t = -c_{step} + r_{goal}$, since node relabelling is not an illegal action and does not negatively influence the heuristic term h_{GED}.

Figure 4.1 implicitly shows the progression of the agent learning this circular relabelling behaviour during a single sample run over the entire atlas dataset (626 graph pairings). The shared y-axis depicts the node, and the x-axis the edit operation selected by the agent. The leftmost heatmap illustrates the initial state of the action capacity matrix during each trajectory. The four remaining heatmaps, from left to right, depict the action capacity matrix at the end of the 10th, 30th, 100th and 300th trajectory, respectively. Whenever a specific edit operation on a node is performed, the corresponding matrix entry is decremented by one. As illustrated in the second heatmap from the left, the agent does not exhibit a pronounced preference for any particular node, with the node relabelling operations distributed relatively uniformly across all nodes. In the third heatmap, the agent already shows a preference for node 0 and 4. The fourth heatmap displays an almost convergent state, with the agent preferably relabelling node 1 and 2. The last heatmap exhibits the agent exclusively relabelling node 1 for the entire trajectory. This undesired behaviour of the agent is reflected in Figure A.1, illustrating a clear lack of any edge edit operations.

To regulate the agent, require the action capacity to remain greater than or equal to zero, since negative values in the capacity matrix quantify the superfluous amount of performed edit operations. Given the $\mathcal{C}$ used for the GED upholds the triangle inequation, for any edit path $\tau \in \mathcal{T}(g_1, g_2)$ that contains relabelling a specific node more than once, there exists $\tau' \in \mathcal{T}(g_1, g_2)$ without the superfluous relabelling operations so that $\mathcal{C}(\tau') \leq \mathcal{C}(\tau)$. This turns the action capacity matrix into a restriction



(a) restricted and unrestricted.

(b) with masking.

Figure 4.2: The measured legal rate and mean GED of action restraint mechanisms. With the blue, orange and green curves displaying the unrestricted, restricted and masked mechanisms, respectively.

matrix, such that any node relabelling beyond the first is declared as an illegal action, incurring the illegal cost. Equivalent restrictions exist for to node insertions and node deletions as well as all edge edit operations.

Figure 4.2a shows the restriction effects on the agent's performance. The y-axes of the upper and lower plot show the legal action rate and GED, respectively. The legal action rate, or simply legal rate, refers to the percentage of legal actions taken within a trajectory. Since a trajectory can be terminated via `truncated` even when no edit path is found, a low legal rate indicates an issue. The x-axes for both show the trajectories of a single sample run over the entire atlas dataset. notably with the same seeded initialisation for all runs. Thus, for example trajectory 128 is the same graph pair in both runs. The blue curve shows the unrestricted baseline with circular relabelling present. Particularly the legal rate rises steadily during the first 100 trajectories to above 0.8 where it remains, which is consistent with the insights of Figure 4.1. The orange curve shows the performance after restricting the node capacity to be non negative. In contrast to the blue curve, the restricted legal rate deteriorates, remaining at roughly 0.4 with occasional dips, presumably when encountering particularly difficult graph pairings. While the legal rate deteriorates compared to the unrestricted case, the mean GED improves by almost 30%. The restriction matrix thus causes a trade-off between legal rate and GED, likely because the agent starts selecting edit operations on edges as seen in Figure A.2.

This indicates the agent at least partially pursues the intended goal of minimising the GED. However, it appears the agent does so by any means possible, with over half of the selected actions marked illegal. An edit path by definition has a legal

rate of one. Therefore, a method that enhances the legal rate while maintaining the GED at its current level or lower is required. As mentioned in Section 3.3, action masking can be employed to filter unreachable or unproductive actions from the action space, depending on the current state. Table A.2 lists examples of the masking rules. The green curve in Figure 4.2b shows the agent's performance when combining the restriction matrix with masking. This almost achieves the target legal rate of one, with the mean legal rate at 0.96. A substantial improvement over the 0.35 mean legal rate of only applying the restriction matrix. Masking also slightly reduces the mean GED in the process.
This combination inspires confidence to assume the agent now pursues the intended goal. Still there is likely room for further improvement by fine-tuning specific hyperparameters.

4.2 Parameter Tuning

The amount of tunable parameters in the system is immense, and covering all possible configurations is vastly out of scope. Instead, a select few parameters and their influence are looked at in depth, to either validate pre-existing knowledge or get new insights. Beginning with two common RL parameters used in Definition 2.9, λ and γ. As a reminder, λ serves as bias-variance trade-off term, while γ governs the temporal component of rewards to regulate the agent's planning horizon. In

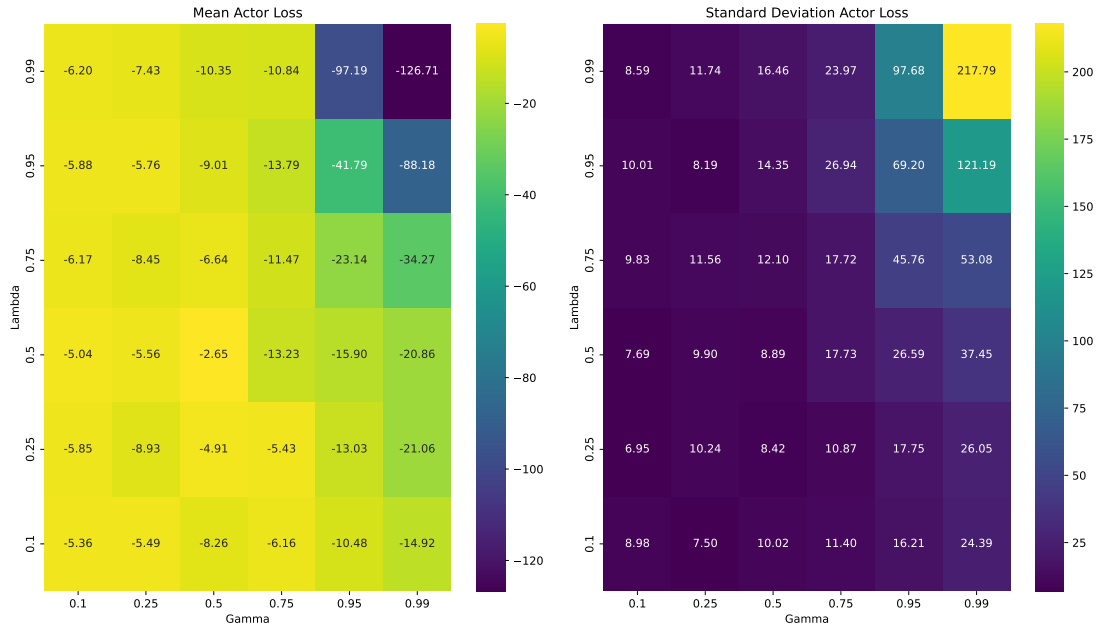


Figure 4.3: Heatmap of the mean actor loss and corresponding SD, depending on the GAE parameters λ and γ.

Figure 4.3, the mean actor loss and its standard deviation (SD) are depicted with respect to λ on the y-axis and γ on the x-axis. The data for each of the parameter combination is collected in a single sample run over the entire atlas dataset with all other parameters held constant, simply called "identical sample runs" from here on. With the variance equal to the SD squared, the variance is highest with large λ and γ values, as expected.
As depicted in the right heatmap by the columns $\gamma \geq 0.95$ and rows $\lambda \geq 0.95$, it can be deduced that reducing the λ term exerts less influence on controlling the variance than reducing γ. For example, the SD at $\lambda = 0.75, \gamma = 0.99$ is over 2.2 times the value of its inverse pair $\lambda = 0.99, \gamma = 0.75$. This difference extends to a varying degree to all entries in the lower right triangle and their anti-diagonal transpose. An ideal parameter combination keeping both λ and γ as small as necessary and as large as possible. Such a combination maximally reduces the variance, retains a maximal temporal horizon, and introduces as little bias as possible. Thus, optimal ranges for lambda and gamma are $\lambda \in [0.95, 0.99]$ and $\gamma \in [0.75, 0.95]$, with values below that range yielding a diminishing reduction of variance. The intuitive decision is to choose conservative values with $\lambda = 0.95$ and $\gamma = 0.75$.

Another aspect that can influence performance, as demonstrated in Section 4.1, is the reward structure. Particularly, the choice of step cost c_{step} and illegal cost $c_{illegal}$. With other RL pathfinding problems as reference, convention seems to be a small step cost $c_{step} \in (0, 1]$, with larger negative penalties $c_{illegal} \in [10, 100]$ for discouraged behaviour and positive rewards $r_{goal} > 0$ for encouraged behaviour. Figure 4.4 illustrates the effects the choice of c_{step} and $c_{illegal}$ have on reducing the GED. Rather than selecting fixed values for $c_{illegal}$, they are a scaled in relation to the base cost c_{step}. The tested base costs are shown on the x-axis, the $c_{illegal}$ scaling factors on the y-axis, with the value for each combination the result of an identical sample run. Noticeable is the $(0.01, 5)$ pair in both plots, displaying the highest mean GED as well as the highest SD. This result is likely an outlier, as the majority of mean GED values are approximately 10 percent lower, with their respective SD being approximately 25 percent lower. Excluding that outlier, the distribution of the remaining values is quite narrow. With the difference of 0.34 between the maximum and minimum mean GED still within the minimum measured SD of 0.35. This observation suggests that the specific values chosen for c_{step} and $c_{illegal}$ are less influential, and that the difference can be explained by inherent randomness of the sample runs. The choice of c_{step} and $c_{illegal}$ is presumably also overshadowed by the significant impact that masking has on preventing illegal actions in the first place. This suggests that the h_{GED} term is probably the main factor influencing the agent's learning. Still, small step costs $c_{step} \leq 0.01$ paired with small scaling

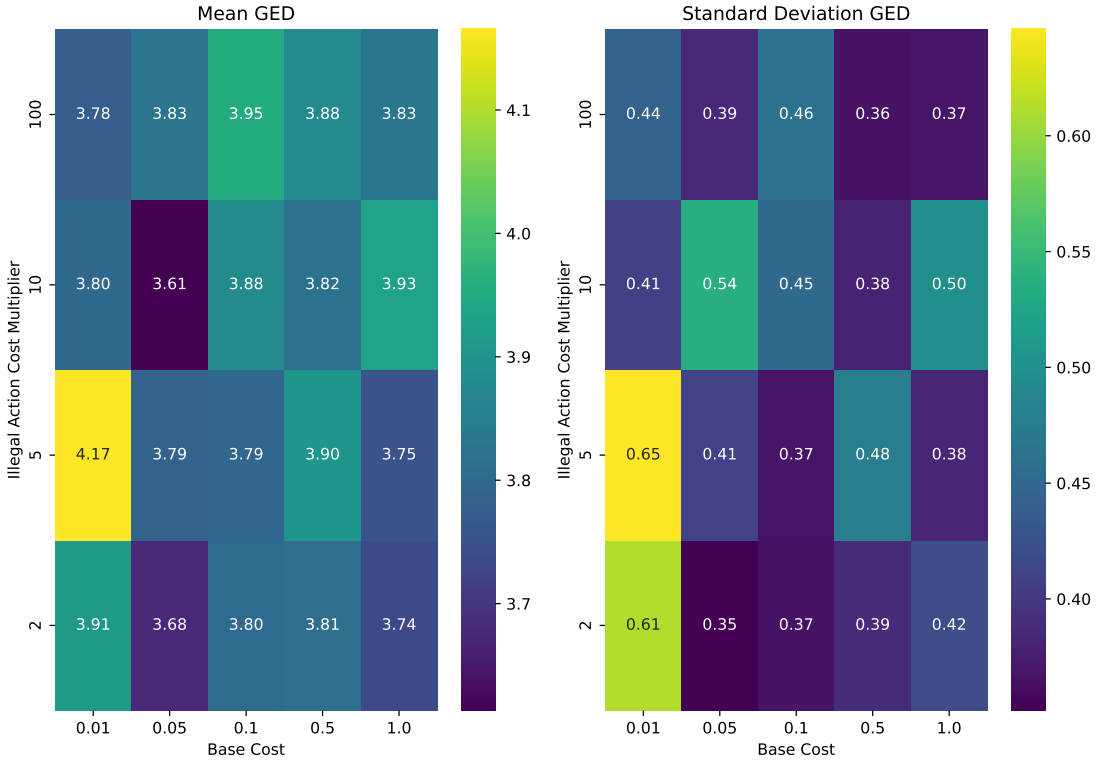


Figure 4.4: Heatmap of the mean GED and its SD, depicting the influence of the chosen step cost and illegal cost on the performance.

factors $c_{illegal} \leq 5 \cdot c_{step}$ should best be avoided.

Finally, a comparison of frequently used GNN and RNN architectures for graph and sequence tasks as part of the environment in Figure 3.2. Some combinations are likely more suited for generating the state embedding than others. The detailed GNN and RNN implementations are listed in the appendix in Code A.1 and Code A.2. Figure 4.5 depicts the mean GED, mean heuristic h_{GED}, and the minimum h_{GED} as heatmaps with respect to the combination of RNN type on the x-axis and GNN type on the y-axis. The values for each combination were collected in a single identical sample run. The minimum heuristic distance heatmap, specifically entries where $h_{GED} = 0$, shows that at least one valid edit path was found in the sample run. Conversely, values greater than zero indicate that no graph pair was solved. However, as all values are still close to zero, it is not possible to definitively exclude a specific pairing.

Looking at the mean GED and the mean h_{GED}, it is clear that no pairing is definitely outperforming the others in either category. Neither are there any clear outliers, with seemingly no identifiable trends between the mean GED and mean h_{GED}. This lack of a trend is supported by the supplementary data presented in Figure A.5, which shows that the correlation between the GED and the h_{GED} tends

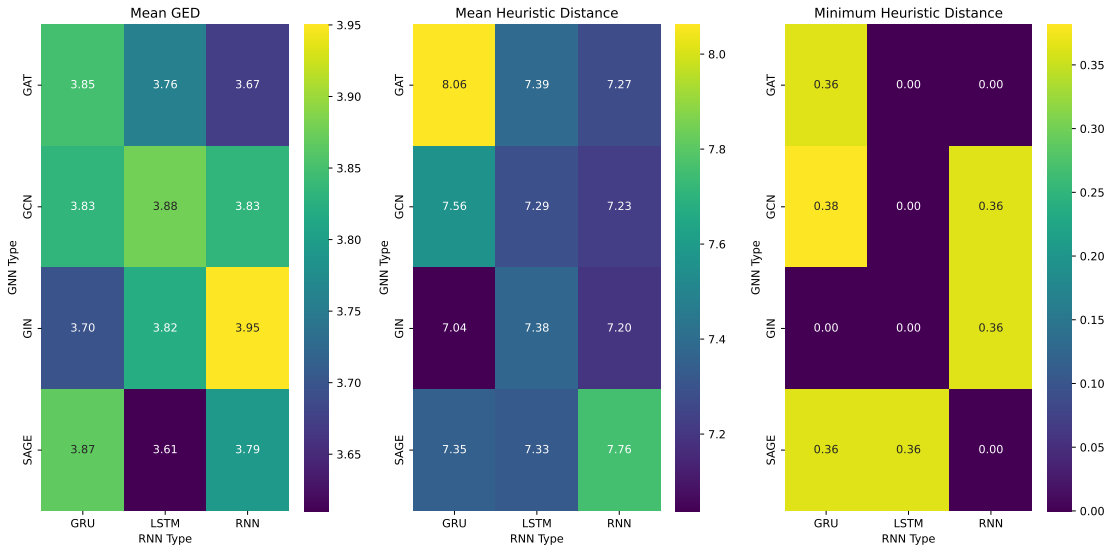
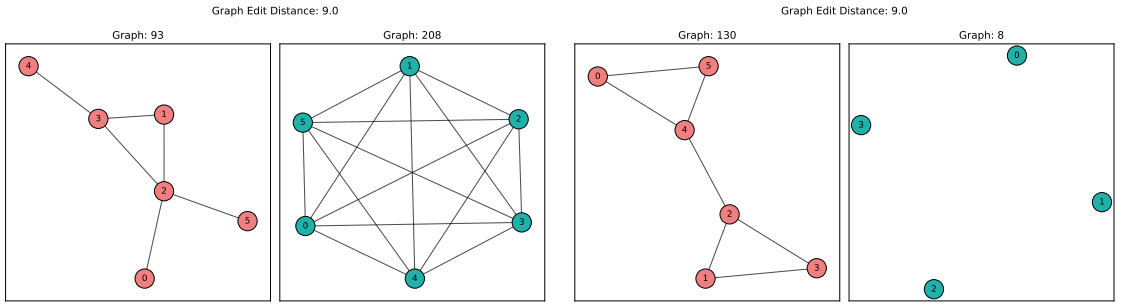


Figure 4.5: Heatmaps displaying the mean GED, mean h_{GED} and minimum h_{GED}, with respect to the RNN and GNN used for embedding the state.

to be around to 0.5. Figure A.5 also suggests that the sample run for the (GAT, GRU) pairing is likely an outlier, displaying an abnormally high GED standard deviation, mean h_{GED}, and correlation.
Regardless, with a correlation of $r = 0.5$ the coefficient of determination is $r^2 = 0.25$, which suggests that h_{GED} insufficiently determines the true GED. This is problematic, because h_{GED} guides the agent's learning. With the data inconclusive as to which GNN and RNN combination is best, other factors such as computational cost and how up-to-date the architecture is become relevant. The Graph Isomorphism Network (GIN) [57] and Gated Recurrent Unit (GRU) [58] are the most recent GNN and RNN architectures, respectively. The GIN is specifically developed for distinguishing graphs and graph structures, while the GRU is considered a direct improvement over LSTMs. The data in Figure 4.5 for the (GIN, GRU) pairing also suggests the combination is performant, with the lowest mean h_{GED} and third lowest mean GED. Therefore, it seems logical to use them to generate state embeddings.

4.3 Solved Pairs

In addition to analysing the model's numerical performance, investigating which graph pairs the fine-tuned agent can solve can provide insight into its strengths and weaknesses. Particularly, since certain pairings are inherently less complex than others. For example, if the source or target graph is the null graph K_0, which has no nodes or edges, then only trivial edit paths involving the insertion or deletion of the entire target or source graph are possible. The data for the subsequent figures



(a) Target graph: Complete graph with six nodes K_6.

(b) Target graph: Edgeless graph with four nodes $\overline{K_4}$.

Figure 4.6: Examples of graph pairs the agent is consistently able to find an edit path for, due to their structural properties.

was collected across 10 identical sample runs, each run passing over the atlas dataset twice. Figure 4.6 highlights two frequently solved graph pairs in the atlas dataset. The graph number is their index in the atlas dataset, with the associated GED calculated by `nx.graph_edit_distance(g1,g2)`. NetworkX uses an edit cost of one for insertions and deletions, with zero cost for relabelling. The left graph of any pair is the source graph and the right is the target graph. The numbers on the nodes are their labels.

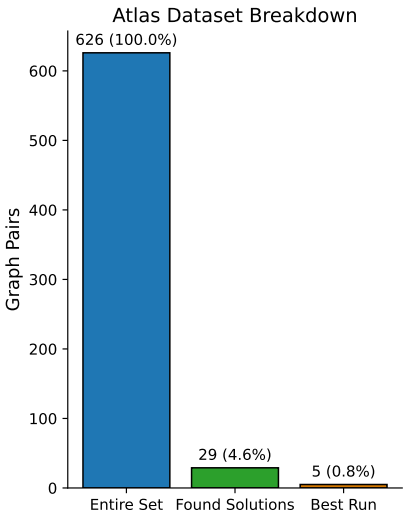


Figure 4.7: The amount of solved pairs across all runs and within a single run, compared to all graph pairs in the atlas dataset.

These two examples illustrate a trend that is also present, albeit to a lesser extent, in other solved graph pairs.Figure 4.6a depicts a pair where the target graph belongs to the class of complete graphs K_n. This significantly reduces the complexity for the agent, as the only available legal actions are node relabelling and edge insertion, both of which have a non-negative impact on reducing the GED. Similarly, the target graph shown in Figure 4.6b belongs to the complement class $\overline{K_n}$. Although it is a slightly more complex pairing because the graphs are of a different order, the available legal actions still naturally drive the source graph towards the target graph, as is the case in Figure 4.6a. This suggests that the agent can learn to select the appropriate actions to a certain degree, such as deleting the correct nodes in the source graph of Figure 4.6b. However, it also indicates that for pairs where

the available legal actions are not strictly neutral or positive in their impact on the GED, the agent fails to make optimal selections. Figure 4.7 shows the number of graph pairs in the atlas dataset for which the agent has found an edit path, relative to the total amount of graph pairs. The y-axis shows the number of graph pairs and the x-axis shows the categories. From left to right, the number of pairs in the entire set, the number of pairs that were solved at least once, and the maximum number of pairs solved in an individual sample run are depicted. The count is also expressed as a ratio relative to the number of pairs in the entire dataset. Notably, the success rate is exceptionally low, with the vast majority of pairs remaining unsolved. In addition, even the pairs for which the agent has found an edit path do not have a guarantee of optimality. Thus, there is still considerable room for improvement with regard to the agent handling less trivial graph pairings. For example, Figure 4.8 depicts a more complex graph pair, where simply inserting or deleting random edges will not yield an improvement like it would for Figure 4.6a. Immediately noticeable for humans but not for the agent, the target graph is a subgraph of the source graph. Its chosen edit path is illustrated in Figure A.6, showing a preference for insertion and deletion operations and omitting any relabelling operations. It suggests that the state embedding captures structural information about the graphs, while also indicating shortcomings in the action selection process or the actor architecture as a whole. This is because edit paths consisting primarily of insertions and deletions are inherently more costly than one with relabelling operations. This also means that the architecture's performance will deteriorate further when used on larger graphs.

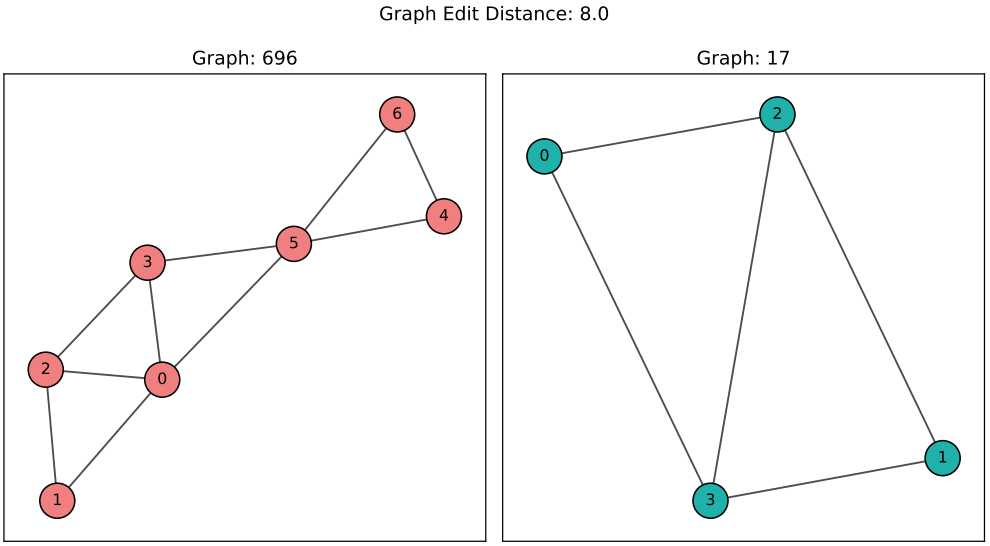


Figure 4.8: Example of less trivial graph pair, providing multiple viable edit paths for the agent.

Chapter 5

Conclusions and Future Work

In this chapter, Section 5.1 revisits the goals set in Section 1.4 and summarises the components developed to achieve them. In addition, the results are discussed and put into context of alternative methodologies for GED approximation. Section 5.2 then highlights remaining unanswered questions, elaborates on avenues for improvement, and defines further goals.

5.1 Conclusions

The overall goal of this thesis was to demonstrate the effectiveness of RL as the primary tool for estimating the GED of a given graph pair. To this end, a novel RL environment and accompanying RL architecture were designed, capable of dynamically exploring edit paths. The proposed custom environment can manage graph datasets such as the atlas dataset or specific TUDatasets. It is also responsible for applying edit operations, generating a state embedding, and returning a reward. The proposed RL architecture follows the actor-critic paradigm. A sequential actor network selects an edit operation, while a critic network predicts the state value which is then used to bootstrap the advantage.
The architecture was continuously improved through accelerated testing and iteration, perpetually expanding the functionality of the environment and the agent. Simultaneously, issues that were adversely affecting the system's performance were identified and resolved. These include the problem of circular relabelling as described in Section 4.1, as well as other, more minor problems.
The final architecture was tested on the atlas dataset, demonstrating its potential to approximate the GED when formulated as a procedural pathfinding problem. However, given its novelty, in terms of accuracy or speed it cannot compete with any of the state-of-the-art methods for GED approximation. In turn, the developed method provides greater interpretability and behavioural insights, thereby reducing

the 'black box issue' present with the current GNN-based methods.
In conclusion, this thesis explored a novel alternative to approximating the GED and demonstrated its feasibility. While not yet competitive, it bridges the gap between previous pathfinding-based approaches that calculate the exact GED and the metric learning-based approaches that directly learn the GED as a metric.

5.2 Future Work

Reinforcement Learning for approximating the GED is a new paradigm. Thus, the presented architecture offers plenty of unexplored avenues for future research. The following section highlights primary areas of unrealised potential that are likely to yield notable improvements with further exploration.
The first major area concerns the heuristic used to guide the agent. While the current heuristic is computationally advantageous, it is also problematic in terms of accurately predicting the remaining GED. Potential alternatives include heuristics based on bipartite matching, graph kernels, and ML. Since the heuristic is calculated after every edit operation, both graph kernels, such as the Wasserstein Weisfeiler-Lehman kernel (WWL) [59], as well as bipartite matching are likely too costly to use. This leaves learned heuristics that leverage neural networks such as GNNs as the most promising option.
The second major area concerns the design of the policy network. The current architecture's strength lies in its ability to generate a usable graph edit operation as output by decomposing its elements into a hierarchical neural network structure. However, the individual neural networks that select the specific edit type (node or edge) and their respective labels consist of basic multilayer perceptrons (MLPs), comprised of linear layers. They provide a good baseline and allow for quick prototyping and modification but are unlikely to be optimal for such a task. Introducing other types of layers, such as attention layers, residual layers and pooling layers could substantially improve the quality of selected actions. Further, transformers or residual networks (ResNets) can be leveraged to generate more nuanced intermediate embeddings. These embeddings would be combined with the output of the upstream MLPs before being fed as input into the downstream MLPs. These are just a few examples of potential adaptations to the policy network architecture. Similar considerations can be made for adapting the value network architecture.
The third major area concerns the environment and its components, with focus on the action space. Defining the action space as individual graph edit operations is straightforward and convenient, since actions then conform to the structure required by NetworkX. The drawback of this action definition is poor scalability to larger

graphs, as well as additional constraints introduced by the graph edit operations being applied sequentially rather than simultaneously. The latter is particularly problematic, since finding a specific sequence of graph edit operations rather than any of its permutations is factorially harder in the worst case. One possible change would be to define the action space more similarly to $\mathcal{T}(g_1, g_2)$, thus actions would represent selecting an entire edit path at once rather than individual graph edit operations. This would allow the GPU to be better utilized via batching and parallelisation, since moving data from the CPU to the GPU multiple times per trajectory is highly inefficient.
Finally, additional backlog items are listed below in no particular order.

Backlog

- ▷ Improve or adapt the masking system. Particularly for filtering relabelling operations.
- ▷ Make the environment capable of handling arbitrary graph datasets.
- ▷ Vectorize the environment for parallelisation.
- ▷ Implement PPO or GRPO as alternatives to A2C.
- ▷ Reduce runtime and memory overhead.
- ▷ Improve data collection and data visualisation for easier analysis.
- ▷ Investigate the potential of MCTS adjacent methods for heuristics.

Appendix A

Appendix

Heuristic Component h_{GED}	Formula
Node Difference	$\|\|V_1\| - \|V_2\|\|$
Edge Difference	$\|\|E_1\| - \|E_2\|\|$
Degree Sequence Difference	$Deg(g_1)\ \Delta_m\ Deg(g_2)$
Node Label Neighbourhood Difference	$\sum \left[\mathcal{N}^{\ell}_{g_1}(u)\ \Delta_m\ \mathcal{N}^{\ell}_{g_2}(v)\right] : \ell_{V_1}(u) = \ell_{V_2}(v)$

Table A.1: Components of the heuristic h_{GED}. Δ_m is the symmetric difference of multisets and $\mathcal{N}^{\ell}_{g}$ defined as: $\mathcal{N}^{\ell}_{g}(u) := \{\ell_{V_g}(w) : w \in \mathcal{N}_g(u)\}$ the label neighbourhood. The heuristic is the sum of these individual components, with $h_{GED} = 0$ declaring the graphs g_1 and g_2 as equal.

Graph Edit Operation	Requirement
Node Insertion	$\|V_1\| < \|V_2\| \wedge u \notin V_1$
Node Deletion	$\|V_1\| > \|V_2\| \wedge u \in V_1$
Node Substitution	$\|V_1\| = \|V_2\| \wedge \forall u \in V_1,\ \exists l' \in \mathcal{L}_V : (\ell_{V_1}(u) \nrightarrow l') \Leftrightarrow \ell_{V_1}(u) = l'$
Edge Insertion	$(u, v) \notin E_1 : u \neq v \wedge u, v \in V_1$
Edge Deletion	$(u, v) \in E_1$
Conditionals	...

Table A.2: An excerpt of requirements to prevent trivial illegal edit operations via masking. Depending on the preceding graph edit operations, further conditional restrictions may be applicable.

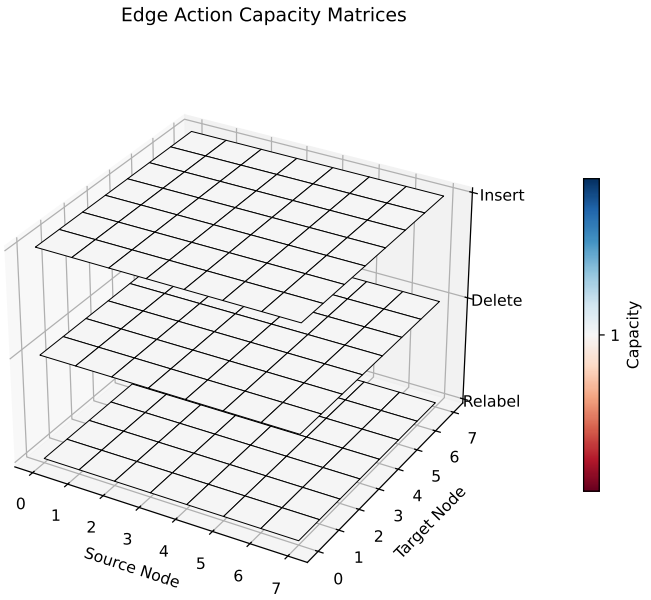
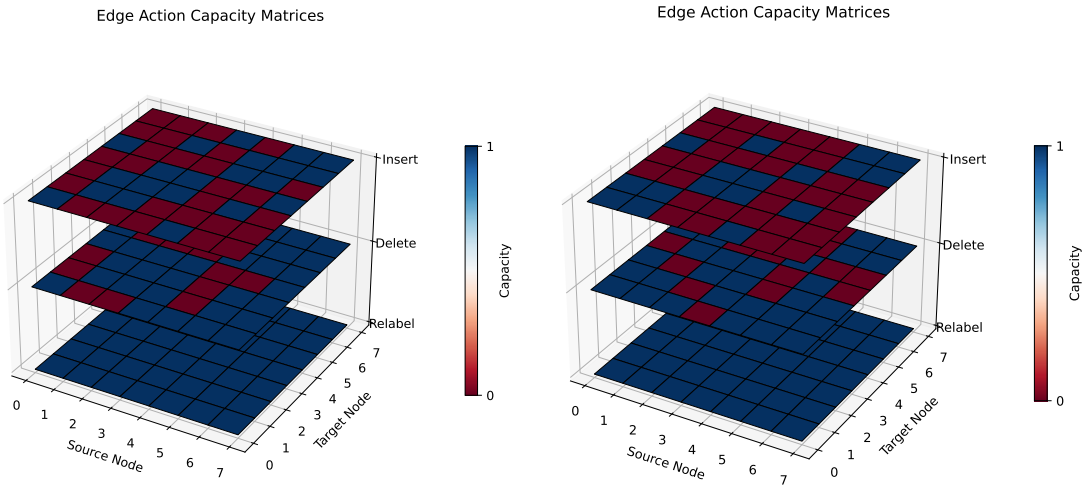


Figure A.1: Edge capacity matrices for each edge edit operation as heatmaps, depicting a consequence of circular relabelling. Edit operations on edges are never selected.



(a) Edge capacity matrices after preventing circular node relabelling.

(b) Edge capacity matrices with masking introduced.

Figure A.2: Edge capacity matrices as heatmaps per edge edit operation type show the agent also selects edge edit operations after restricting the edit capacities, compared to Figure A.1.

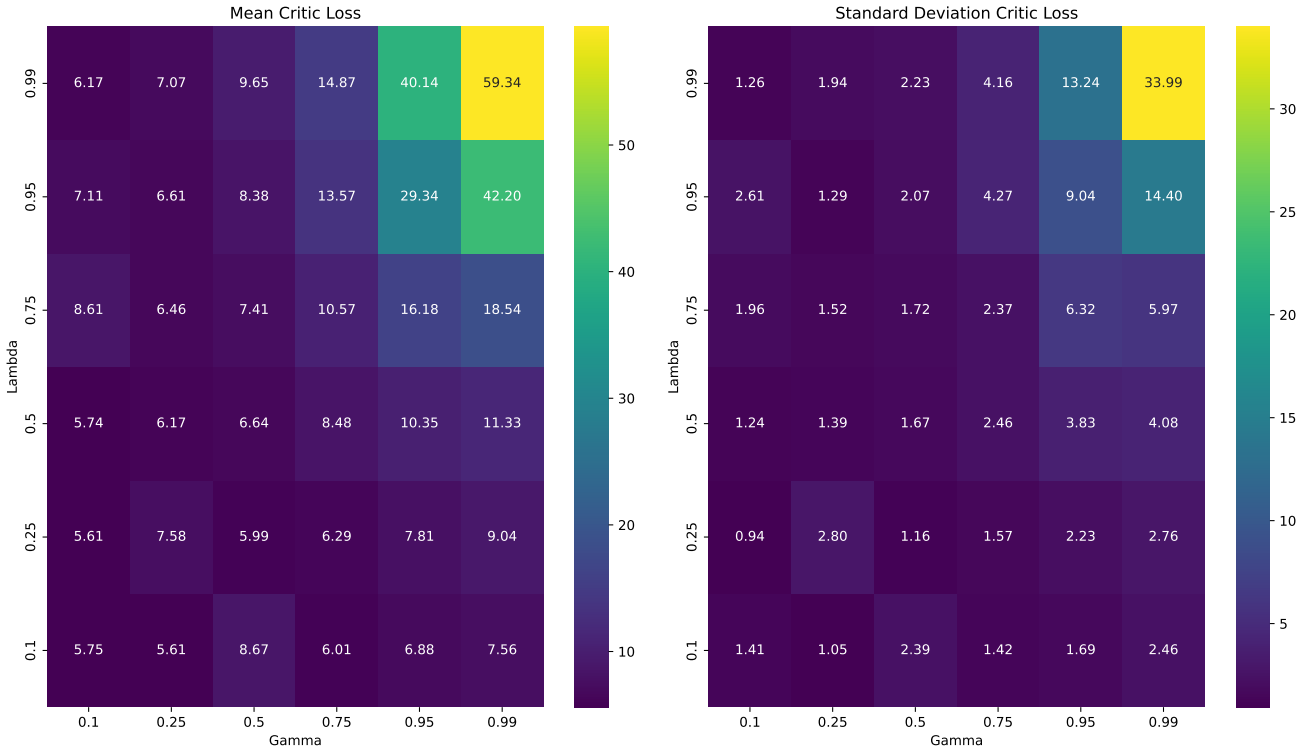


Figure A.3: Heatmap displaying the mean critic loss and its SD with respect to the GAE parameters λ and γ of a single pass over the atlas dataset on the left and right respectively.

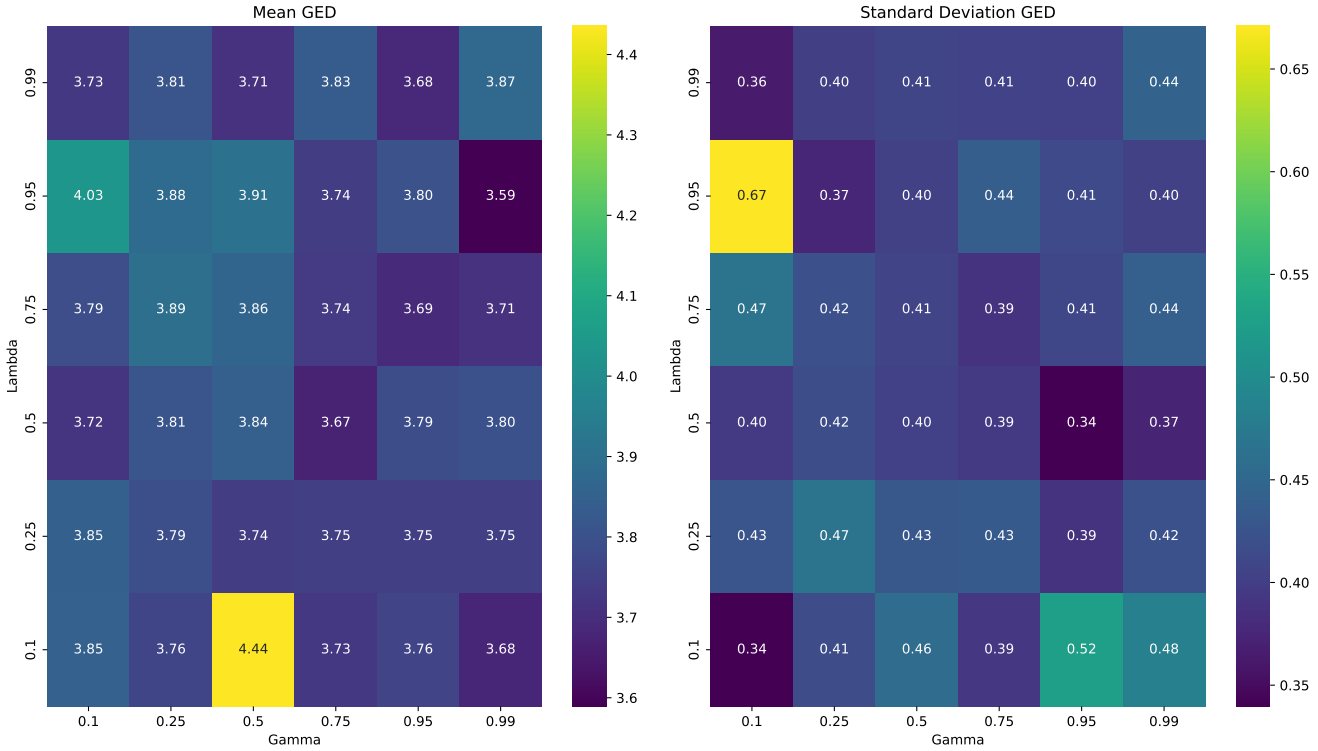


Figure A.4: Heatmap displaying the mean GED and its SD with respect to λ and γ, generated in the same run as Figure A.3 and Figure 4.3.

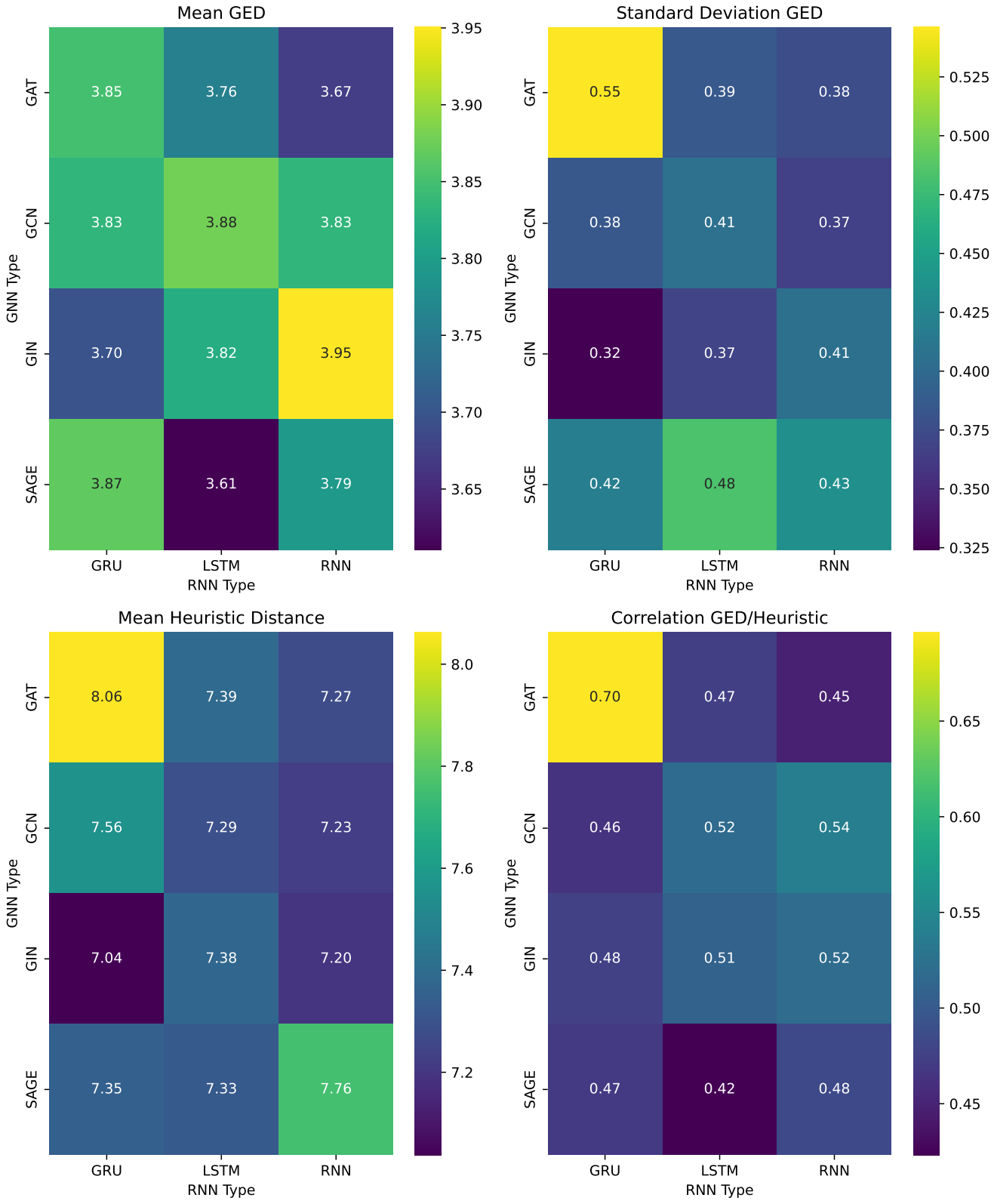


Figure A.5: Auxiliary data showing the mean GED, its SD, mean h_{GED}, and the correlation between GED and h_{GED} with respect to the GNN and RNN type.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GINConv, GCNConv, GATConv, SAGEConv,
from torch_geometric.nn import global_mean_pool

class GNNModule(nn.Module):
    def __init__(self, input_dim):
        super(GNNModule, self).__init__()
        self.pool = global_mean_pool
        # Sample and Aggregate (SAGE)
        self.conv1 = SAGEConv(input_dim, 32)
        self.conv2 = SAGEConv(32, 64)
        self.conv3 = SAGEConv(64, 128)
        # Graph Attention Network (GAT)
        self.conv1 = GATConv(input_dim, 8), heads=4)
        self.conv2 = GATConv(8, 16), heads=4)
        self.conv3 = GATConv(16, 32, heads=4)
        # Graph Convolution Network (GCN)
        self.conv1 = GCNConv(input_dim, 32)
        self.conv2 = GCNConv(32, 64)
        self.conv3 = GCNConv(64, 128)
        # Graph Isomorphism Network (GIN)
        self.conv1 = GINConv(nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.PReLU(),
            nn.Linear(32, 32),
        ), )
        self.conv2 = GINConv(nn.Sequential(
            nn.Linear(32, 64),
            nn.PReLU(),
            nn.Linear(64, 64),
        ), )
        self.conv3 = GINConv(nn.Sequential(
            nn.Linear(64, 128),
            nn.PReLU(),
            nn.Linear(128, 128),
        ), )

    def forward(self, features, edge_index):
        features = F.relu(self.conv1(features, edge_index))
        features = F.relu(self.conv2(features, edge_index))
        features = F.relu(self.conv3(features, edge_index))
        features = self.pool(features)
        return features
```

Code A.1: Detailed view of the GNN model implementations used to generate the graph embeddings.

```python
# imports ...
class SequenceModule(nn.Module):
    def __init__(self, input_size):
        super(SequenceModule, self).__init__()
        # Long Short-Term Memory (LSTM)
        self.model = nn.LSTM(input_size, 128, n_layers=1)
        # Recurrent Neural Network (RNN)
        self.model = nn.RNN(input_size, 128, n_layers=1)
        # Gated Recurrent Unit (GRU)
        self.model = nn.GRU(input_size, 128, n_layers=1)

    def forward(self, features):
        outputs, h_n = self.model(features)
        return h_n[-1, :]
```

Code A.2: Detailed view of the RNN model implementations used to generate the edit sequence embeddings.

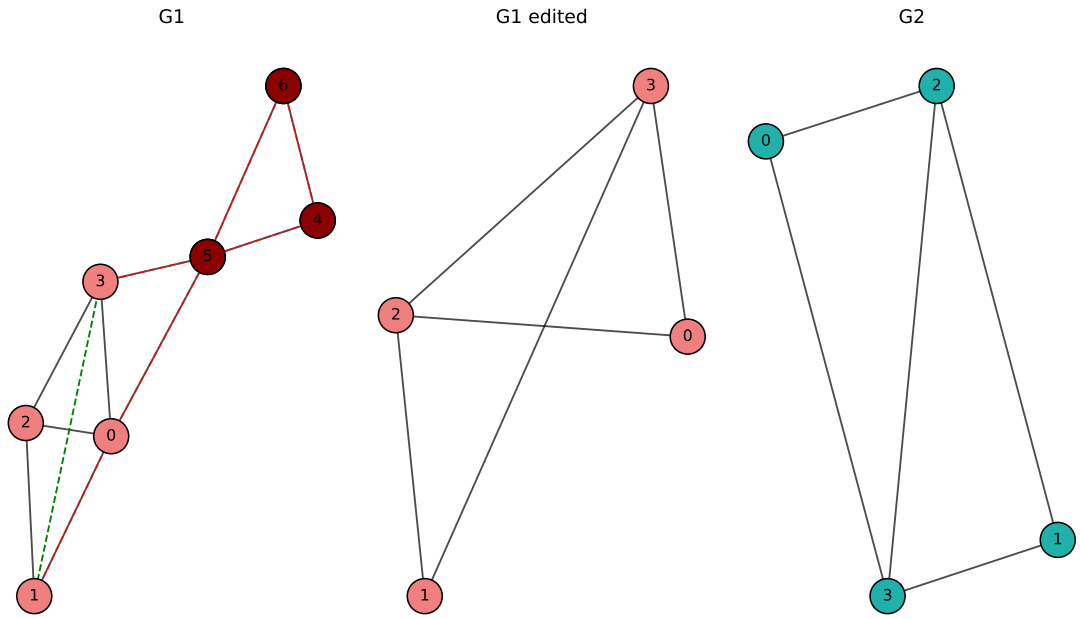


Figure A.6: Example of a suboptimal edit path transforming G1 into G2. Edge deletions are dashed red, edge insertions dashed green. Deleted nodes are dark red.

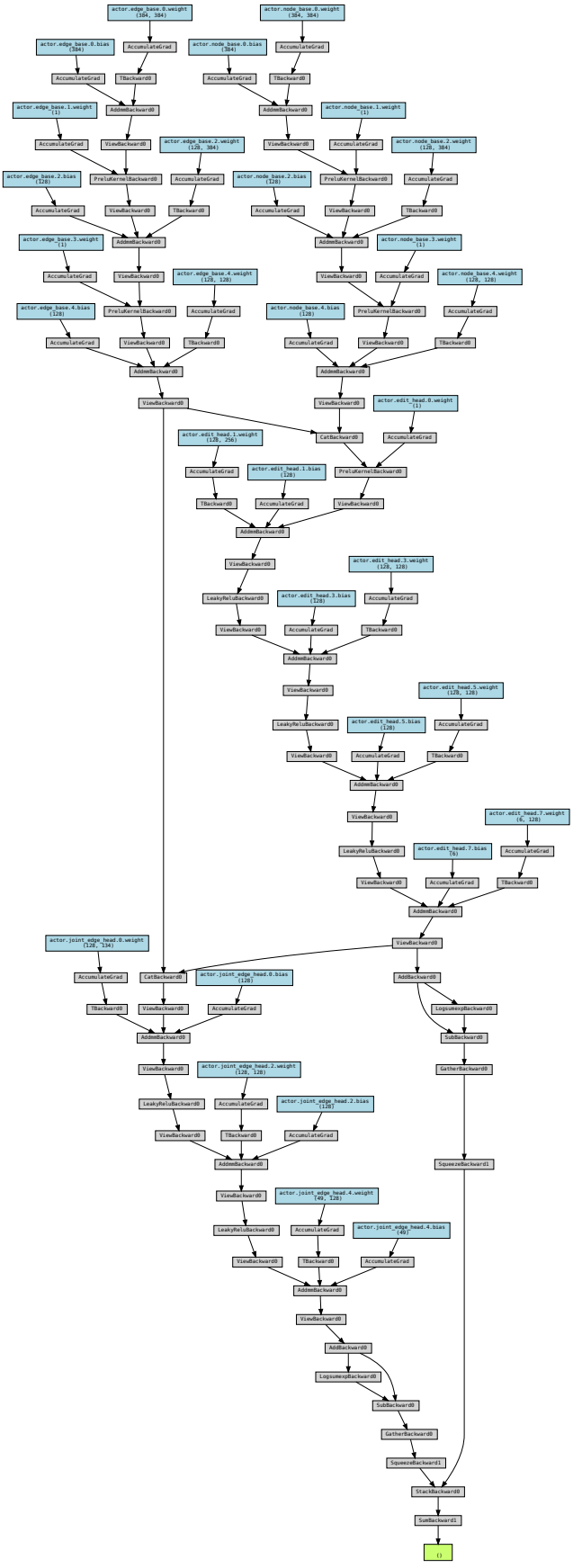


Figure A.7: Exemplary actor network architecture, depicting the entire edge selection process.

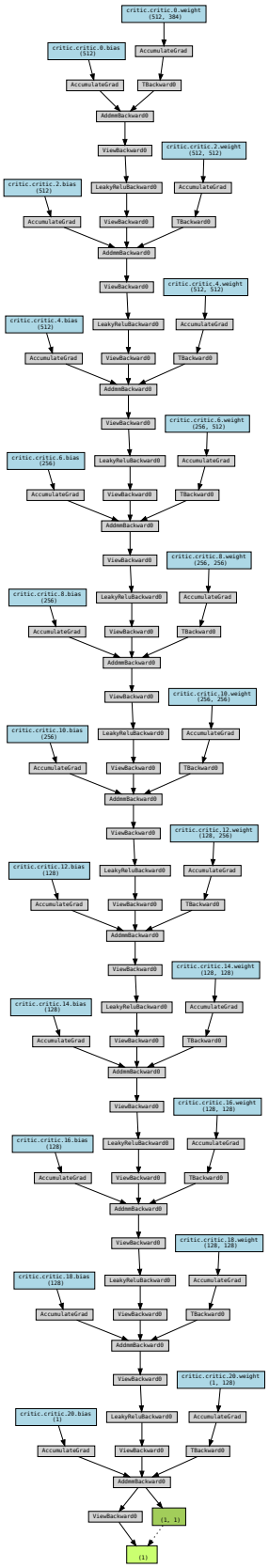


Figure A.8: Entire architecture of the critic network.

Bibliography

[1] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy J. Mitra, and Leonidas J. Guibas. Structurenet: hierarchical graph networks for 3d shape generation. *ACM Trans. Graph.*, 38(6):242:1–242:19, 2019.

[2] Laurenz Wiskott, Jean-Marc Fellous, Norbert Krüger, and Christoph von der Malsburg. Face recognition by elastic bunch graph matching. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(7):775–779, 1997.

[3] Alexandru T. Balaban. Applications of graph theory in chemistry. *J. Chem. Inf. Comput. Sci.*, 25(3):334–343, 1985.

[4] Wolfgang Huber, Vincent J. Carey, Li Long, Seth Falcon, and Robert Gentleman. Graphs in molecular biology. *BMC Bioinform.*, 8(S-6), 2007.

[5] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proc. IEEE*, 104(1):11–33, 2016.

[6] Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris. A survey on graph kernels. *Appl. Netw. Sci.*, 5(1):6, 2020.

[7] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 855–864. ACM, 2016.

[8] David B. Blumenthal and Johann Gamper. On the exact computation of the graph edit distance. *Pattern Recognit. Lett.*, 134:46–57, 2020.

[9] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *Proc. VLDB Endow.*, 2(1):25–36, 2009.

[10] Kaspar Riesen, Miquel Ferrer, and Horst Bunke. Approximate graph edit distance in quadratic time. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 17(2):483–494, 2020.

[11] Stefan Fankhauser, Kaspar Riesen, and Horst Bunke. Speeding up graph edit distance computation through fast bipartite matching. In Xiaoyi Jiang, Miquel Ferrer, and Andrea Torsello, editors, *Graph-Based Representations in Pattern Recognition - 8th IAPR-TC-15 International Workshop, GbRPR 2011, Münster, Germany, May 18-20, 2011. Proceedings*, volume 6658 of *Lecture Notes in Computer Science*, pages 102–111. Springer, 2011.

[12] Pau Riba, Andreas Fischer, Josep Lladós, and Alicia Fornés. Learning graph edit distance by graph neural networks. *Pattern Recognit.*, 120:108132, 2021.

[13] Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. Computing graph edit distance via neural graph matching. *Proc. VLDB Endow.*, 16(8):1817–1829, 2023.

[14] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *Int. J. Robotics Res.*, 32(11):1238–1274, 2013.

[15] B. Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Kumar Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Trans. Intell. Transp. Syst.*, 23(6):4909–4926, 2022.

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

[17] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[18] Mingshuo Nie, Dongming Chen, and Dongqi Wang. Reinforcement learning on graphs: A survey. *IEEE Trans. Emerg. Top. Comput. Intell.*, 7(4):1065–1082, 2023.

[19] Lei Yang and Lei Zou. Noah: Neural-optimized a* search algorithm for graph edit distance computation. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 576–587. IEEE, 2021.

[20] Ronald C Read and Robin J Wilson. Graphs. In *An Atlas Of Graphs*. Oxford University Press, 11 1998.

[21] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. *CoRR*, abs/2007.08663, 2020.

[22] Kaspar Riesen. *Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications*. Advances in Computer Vision and Pattern Recognition. Springer, 2015.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction, 2nd Edition*. MIT Press, 2018.

[24] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *Int. J. Pattern Recognit. Artif. Intell.*, 18(3):265–298, 2004.

[25] Pasquale Foggia, Gennaro Percannella, and Mario Vento. Graph matching and learning in pattern recognition in the last 10 years. *Int. J. Pattern Recognit. Artif. Intell.*, 28(1), 2014.

[26] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Trans. Knowl. Data Eng.*, 30(9):1616–1637, 2018.

[27] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. Graph kernels. *J. Mach. Learn. Res.*, 11:1201–1242, 2010.

[28] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, 2011.

[29] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. In J. Shane Culpepper, Alistair Moffat, Paul N. Bennett, and Kristina Lerman, editors, *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, WSDM 2019, Melbourne, VIC, Australia, February 11-15, 2019*, pages 384–392. ACM, 2019.

[30] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

[31] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[32] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[33] Jiyang Bai and Peixiang Zhao. Tagsim: Type-aware graph similarity learning and computation. *Proc. VLDB Endow.*, 15(2):335–347, 2021.

[34] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[35] Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6348–6358, 2017.

[36] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In Bryan Ford, Alex C.

Snoeren, and Ellen W. Zegura, editors, *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016*, pages 50–56. ACM, 2016.

[37] Yanqi Zhou, Sudip Roy, AmirAli Abdolrashidi, Daniel Wong, Peter C. Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phitchaya Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. Transferable graph optimizers for ML compilers. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[38] Edward L. Thorndike. The law of effect. *The American Journal of Psychology*, 39(1/4):212–222, 1927.

[39] B. F. Skinner. *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century-Crofts, New York, NY, USA, 1938.

[40] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Syst. Man Cybern.*, 13(5):834–846, 1983.

[41] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Mach. Learn.*, 8:279–292, 1992.

[42] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017.

[43] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, abs/1712.06567, 2017.

[44] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[45] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[46] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[47] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[48] Mark Towers, Ariel Kwiatkowski, Jordan K. Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments. *CoRR*, abs/2407.17032, 2024.

[49] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *J. Mach. Learn. Res.*, 22:268:1–268:8, 2021.

[50] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[51] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John P. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[52] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

[53] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[54] Mariya Tsvarkaleva and Louise A. Dennis. No free lunch: Overcoming reward gaming in AI safety gridworlds. In Ibrahim Habli, Mark Sujan, Simos Gerasimou, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security. SAFECOMP 2021 Workshops - DECSoS, MAPSOD, DepDevOps, USDAI, and WAISE, York, UK, September 7, 2021, Proceedings*, volume 12853 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2021.

[55] Joel Lehman, Jeff Clune, Dusan Misevic, Christoph Adami, Lee Altenberg, Julie Beaulieu, Peter J. Bentley, Samuel Bernard, Guillaume Beslon, David M. Bryson, Nick Cheney, Patryk Chrabaszcz, Antoine Cully, Stéphane Doncieux, Fred C. Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Frénoy, Christian Gagné, Léni K. Le Goff, Laura M. Grabowski, Babak Hodjat, Frank Hutter, Laurent Keller, Carole Knibbe, Peter Krcah, Richard E. Lenski, Hod Lipson, Robert MacCurdy, Carlos Maestre, Risto Miikkulainen, Sara Mitri, David E. Moriarty, Jean-Baptiste Mouret, Anh Nguyen, Charles Ofria, Marc Parizeau, David P. Parsons, Robert T. Pennock, William F. Punch, Thomas S. Ray, Marc Schoenauer, Eric Schulte, Karl Sims, Kenneth O. Stanley, François Taddei, Danesh Tarapore, Simon Thibault, Richard A. Watson, Westley Weimer, and Jason Yosinski. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *Artif. Life*, 26(2):274–306, 2020.

[56] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J. Russell, and Anca D. Dragan. Inverse reward design. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6765–6774, 2017.

[57] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[58] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL, 2014.

[59] Matteo Togninalli, M. Elisabetta Ghisu, Felipe Llinares-López, Bastian Rieck, and Karsten M. Borgwardt. Wasserstein weisfeiler-lehman graph kernels. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 6436–6446, 2019.

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname: Zwahlen Gian Alexander

Matrikelnummer: 19-932-979

Studiengang: BSc Informatik

Bachelor ☑ Master ☐ Dissertation ☐

Titel der Arbeit: A Reinforcement Learning Approach to Graph Edit Distance Estimation

LeiterIn der Arbeit: PD Dr. Kaspar Riesen
Dr. Anthony Gillioz

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern 31.07.2025
Ort/Datum

Unterschrift