DEEP REINFORCEMENT LEARNING FOR ADVERSARIAL GAMES ON GRAPHS

A Thesis

by

HARISH KUMAR

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Anxiao Jiang
Committee Members,	Dileep Kalathil
	Guni Sharon
	Theodora Chaspari
Head of Department,	Scott Schaefer

May 2020

Major Subject: Computer Science

ABSTRACT

The game of cops and robbers is a multi-agent adversarial game played on graphs. Previous research on agent strategies for this game has focused on designing heuristics for minimax strategies and these often impose strict restrictions on the graph structure. This research develops a methodology that instead uses Deep Reinforcement Learning and Graph Convolutional Networks by training a cop and robber iteratively against each other.

Naïve implementations of such iterative training suffer from instability and during training and can result in one-sided agent performance. This work overcomes this issue through a few simple modifications. Instead of training an agent against the most recent version of its opponent, several versions of the opponent are preserved and used to train an agent. In addition, the number of steps for which each agent is trained is set depending on its most recent performance evaluation.

To increase the operating range of the agents, based on recent work in hierarchical graph pooling, an efficient Vertex Pooling technique is introduced that allows the basic approach to be scaled to large graphs with only a sub-linear increase in the neural network depth. By aggregating local graph information from increasingly larger receptive fields, this approach achieves a scaling in which the number of GCN layers in the network needs to increase only logarithmically with the diameter of the graph.

The overall method is evaluated by measuring its performance in competition with clairvoyant opponents. Upon evalution, it is seen that these techniques together lead to agents that perform near-optimally on graphs that were never used during training. The approach proposed in this research is also compared with two traditional algorithms: Alpha-Beta pruning, and UCT search.

By creating neural architectures and training methods that allow Reinforcement Learning to be applied successfully to decision-making problems on graphs, this research results in techniques that can be applied to a range of RL problems involving decision-making by multiple agents on graphs.

DEDICATION

To my mother, my father, and my little brother.

ACKNOWLEDGMENTS

I would first like to express my heartfelt gratitude to my advisor, Prof. Anxiao Jiang for his time, guidance and patience. His expertise brought knowledge, stability and direction to our work on this topic. I must especially thank him for the support and freedom he offered during our research and for supporting me in exploring areas that I found interesting. I'm grateful to my committee members Prof. Dileep Kalathil, Prof. Guni Sharon and Prof. Theodora Chaspari for their advice during the course of this research. I thank all the instructors whose courses I have attended at TAMU - I thoroughly enjoyed every one of these courses and learned wonderful concepts some of whose very existence I was previously unaware of.

I thank the Computer Science Department of Texas A&M for funding me as a Teaching Assistant all throughout my Masters program. I thank Prof. Scott Schaefer, Prof. Eduardo Nakamura, Prof. Michael Moore and Prof. Robert Lightfoot for the pleasant work environment that they kept during the tenures when I worked with them. I thank my former students whose constructive feedback was essential in my pedagogical growth.

I thank my undergraduate advisor Prof. Balaraman Ravindran for granting me the experience of undertaking an enjoyable thesis in computer science, for continuing to offer me his guidance and good wishes to this day, and for being a great teacher to learn from. I also thank Prof. Neelima Gupte, my undergraduate advisor, for mentoring and advising me during my undergraduate degree. From Solid State Physics to Information Theory, the courses that I studied during this time were formative and fundamental to my career. I thank all the Professors and instructors of IIT Madras who have taught me, and my peers there for gifting me with an environment where I had something new to learn from everyone.

I wish to thank Julius Kusuma, Sourav Chatterjee, Paul Varkey, Vishvas Suryakumar, Ted Woodward and Martijn de Jongh for their warm and friendly mentorship during my summer internship. Their steady guidance and their unwavering belief in me ensured that I had a successful internship, and it is thanks to them that upon returning to college, I could focus without diversions

on this thesis.

Several friends have been an integral part of why I have enjoyed life for two and a half decades. I wish to thank Aditya Gurunathan, Ajay Krishna, Bharath Sivaram, T.R. Sriram, Sumit Kumar, Vijayaraghavan and Rudra Prasath for putting up with my antics and eccentricities all these years. These people have been my partners-in-crime for many years and I enjoyed the times I spent in their company.

In Lakshmi and Kumar, I have been blessed with the most supportive, encouraging, kind and loving parents anybody could ever ask for. Much of my success in life can be attributed to their love and support through decades of occasionally difficult, but always joyful times. My younger brother Nithish, now a graduate student himself, is in many ways the same delightful kid that I grew up with. I thank him for being a steady source of wisdom, wit and unintended comic relief through thick and thin. I thank all my relatives for their kindness, affection and encouragement. My grandparents, Ananthanarayanan and Ananthalakshmi, Yegna Ramanathan and Rajalakshmi are not with me today, but their inspiring lives and the kindness and warmth they showed to both friends and strangers have ensured that they will always be remembered, and their memories cherished.

Lastly, I do not think it out of place to express my admiration and indebtedness to the artists of all the music that I listened to over the past two years and more: Pandit Ravishankar, Ustad Zakir Hussain, Pandit Hariprasad Chaurasia, Rakesh Chaurasia, Pandit Ajoy Chakraborty, Kaushiki Chakraborty, Mandolin U. Srinivas, John McLaughlin, Jayanthi Kumaresh, Bombay Jayashree, Ranjani & Gayathri, A.R. Rahman and many other brilliant musicians. I also bow in gratitude to the great Carnatic and Hindustani composers from centuries ago whose wisdom and skill continues to influence and inspire the music of today. These compositions were not merely my antidote in tough times, but also founts of boundless joy.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professors Anxiao Jiang and Guni Sharon and Theodora Chaspari of the Department of Computer Science, and Professor Dileep Kalathil of the Department of Electrical and Computer Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by a series of Graduate Teaching Assistantship grants from Texas A&M University.

NOMENCLATURE

RL	Reinforcement Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
GCN	Graph Convolutional Network
MDP	Markov Decision Process
MMDP	Multi-agent Markov Decision Process
POMDP	Partially Observable Markov Decision Process
MOMDP	Mixed-Observability Markov Decision Process
ReLU	Rectified Linear Unit
UCT	Upper Confidence Trees
MCTS	Monte Carlo Tree Search
IAT	Iterative Adversarial Training
MAB	Multi-Armed Bandit
PRA*	Partial Refinement A*

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Graphs can represent a wide range of problems involving relational structures, local and global transportation networks, communication networks, the web and other abstractions of real-world spatial domains. Decision-making problems on graphs are often too complex for traditional approaches to solve without significant expert-driven design tailored specifically to the problem. One such problem is the Game of Cops and Robbers ([1]): this is a multi-agent adversarial game in which two teams of agents, cops and robbers, play a pursuit-and-evasion game on a connected graph. The cops attempt to arrest the robbers one by one through occupying the same vertices as the robbers, while the robbers attempt to avoid such a situation. This problem has applications in physical security, browser prefetching and disease prevention and control ([2]).

The problem of identifying strategies for the game of Cops and Robbers has been shown to be PSPACE-Hard in [3]. Even finding out the minimum number of cops required to guarantee a cop victory on a given graph is NP-Hard. A standard approach to solving zero-sum games like this is to use Negamax, Alpha-Beta pruning or Monte Carlo Tree Search along with well-designed heuristics. The research community has thus focused on heuristic approaches to build strategies for cops and robbers ([4], [5], [6]). An issue with many of these heuristics is that they are manually crafted and are tailored to the specific variant of the problem. In addition, their good performance is often contingent upon the graph following restrictive structures such as grids and octiles.

Reinforcement Learning (RL) deals with training intelligent agents to take optimal decisions in a given environment so that the cumulative reward resulting from the series of decisions is maximized. While RL techniques can use simple methods such as tables, linear models and other techniques to represent the agent's learning, training RL agents for complicated problems is a rather difficult task with these limited tools ([7]). The advent and rise of Deep Neural Networks resulted in the application of the strong generalization and representation capacity of DNNs to be applied to Reinforcement Learning tasks. Hence, a prime motivation behind using Deep Reinforcement Learning on the cops and robbers problem is the hope that there may exist an RL paradigm that does

not require a significant amount of manual intervention for each variant, and instead be general enough to be directly trained on each new variant with minimal changes.

Deep RL methods rely on a neural function approximator to represent the learned policy of the agent. The architecture of this neural network depends upon the problem that we are solving and is often tailored specifically to the domain. For instance, deep learning tasks involving visual inputs are often solved with deep Convolutional Neural Networks ([8], [9]) while those involving temporal inputs are solved with deep Recurrent Neural Networks that use units such as LSTMs ([10]). This thesis leverages DNNs designed to solve graph problems called Graph Convolutional Networks ([11]) towards training RL agents to act optimally on the cops and robbers problem.

We iteratively trained the GCN-based cop and robber by playing them against each other and made them both learn from their opponent. Upon a basic substrate of such an iterative training strategy, a number of modifications are introduced which make the process stable and prevent a single agent from dominating the game at the cost of both agents' learning. One such modification was to maintain a buffer of different versions of past opponents and train randomly against them in each training step. This expands the diversity of the opponents that the learning agent faces and alleviates the effect of catastrophic forgetting. Another useful modification was to adaptively vary the number of steps each agent is allowed to learn for before the other agent begins learning. We created agents that achieve very high success rates on graphs on which optimal play by the agents guarantees victory.

Starting from recent work in hierarchical graph pooling, we implement a fast and efficient Vertex Pooling technique that exponentially enlarges the receptive field of each GCN layer. By aggregating local graph information from increasingly larger receptive fields, this approach achieves a range scaling in which to maintain good performance, the number of GCN layers in the network needs to increase only logarithmically with the required operational range.

The overall method is evaluated by measuring its performance in competition with clairvoyant opponents. The approach proposed in this thesis is also compared with two other algorithms: alpha-beta pruning with a distance heuristic, and upper confidence trees. Our techniques together

result in agents that perform near-optimally on graphs that were never used during training.

Following are our contributions through this project:

1. A GCN-based neural network architecture that is capable of learning generalized policies for instances of the cops and robbers problem.

2. The Iterative Adversarial Training algorithm that trains the cop and the robber by allowing them to learn from each other, along with modifications that increase stability and equitable learning.

3. A modification to the hierarchical vertex pooling methodology introduced in [12]. This focuses on graph navigability and enlarges the receptive field of GCN layers and hence the operating range of our agents.

4. A rigorous evaluation of the trained agent undertaken by allowing the agents to compete against a clairvoyant, optimal opponent.

1.1 Organization

Chapter 2 goes into detail on the required background knowledge in Deep Learning, Reinforcement Learning, Graphs and Games that is helpful to understand this thesis. Chapter 3 describes the present state-of-the-art in the area and other related research that may be of interest. Chapter 4 defines the problem, expresses it as a Markov Decision Process and proposes the in-memory representation used in this project. Chapter 5 provides a detailed discussion of the training methods and algorithmic techniques that we have used in this thesis. Chapter 6 deals with the software implementation of our solution, programming challenges that we faced, and how we solved these issues. Chapter 7 details how we evaluate the performance of cop and robber agents, compares it with other algorithmic techniques, illustrates our results, offers relevant plots and discusses their significance. Chapter 8 concludes the thesis and offers a discussion of potential directions in which our work may be built upon and improved.

2. CONCEPTS FROM DEEP LEARNING, REINFORCEMENT LEARNING, GRAPH THEORY AND GAME THEORY

This chapter introduces background knowledge on Deep Learning, Reinforcement Learning, basic elements of Graph Theory, and Game Theory with a focus on zero-sum games. The introductory material in this chapter is relevant to the work done in this thesis and illustrates fundamental concepts from these topics.

2.1 Deep Learning

Deep learning is a methodology to estimate the optimal parameters for extremely complex mathematical functions (called deep neural networks) that use a set of inputs to predict random variables. Here, the various elements of deep learning are introduced in a bottom-up fashion.

2.1.1 Neurons

A neuron is the fundamental functional unit used in deep learning. Several neurons, often thousands of them, together form a network that is called a Deep Neural Network (DNN). One simple variant of a neuron takes a set of inputs x_i and returns a scalar y as output.

$$y = g(\sum_i w_i x_i) \tag{2.1}$$

w_i are the internal parameters of the neuron, commonly called weights. Here, g is a non-linear function such as the sigmoid function:

$$\sigma(x) = \frac{e^x}{1 + e^x} \tag{2.2}$$

2.1.2 Activation Functions

The presence of the non-linear function, called an activation function, enhances the representational capacity of the neural network. Without such a non-linearity, the neural network degenerates to a linear function, thereby severely limiting the set of functions that it can approximate.

Other variants of neurons may use different non-linearities such as $\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, $tan^{-1}(x)$, $Softplus(x) = \ln 1 + e^x$ and the now-widely used Rectified Linear Unit (ReLU):

$$ReLU(x) = \max(0, x) \tag{2.3}$$

Despite its apparent simplicity, the ReLU activation function is greatly preferred over other non-linearities due to the following reasons:

1. It is computationally very efficient to calculate.
2. The gradient of ReLU does not vanish close to 0 when $x > 0$, thus enabling efficient use of gradient-based optimization techniques.
3. It enforces sparsity in the input when $x \leq 0$, thereby allowing for more regularized intermediate representations as well as faster computation using sparsity-conscious matrix multiplication techniques.

We use ReLU activations almost all throughout this project for the above reasons.

2.1.3 Neural Networks

A neural network is a connected network built out of several neurons. While neural networks in general can have widely varying structures, in practice, these networks are organized into several layers that do not have any intra-layer connections. Each layer calculates an output that depends on earlier layers and passes the result to the next layer.

The simplest example of a neural network, the fully connected neural network, has the following structure: There are K layers in the network and the k^{th} layer has N_k neurons. Given a set of i input variables represented as an i-length vector X, the output at each layer can be represented as

$$Y_1 = g_1(W_1 X) \tag{2.4}$$

$$Y_k = g_k(W_k Y_{k-1}), 1 < k \leq K \tag{2.5}$$

W_k are the weights for the k^{th} layer. N_K, i.e. the number of neurons in the final layer must be equal to the number of outputs that we want our function to represent. Note that we are able to gather the operation of all the neurons in a layer together as a single matrix multiplication (followed by a non-linear function) due to the absence of connections within layers, and the same activation function being used for all neurons in a single layer.

The Universal Approximation Theorem showed that such a neural network with a sufficient number of neurons and the right weights can approximate any continuous function with arbitrary accuracy even for $K = 2$.

2.1.4 Variants of Neural Networks

Whereas the universal approximation theorem states that just fully connected neural networks can represent all continuous functions, finding the right weights for such networks is a very difficult task that may often be infeasible. Specific problem domains often admit much simpler (in terms of computation) neural network architectures and neuron variants that vastly outperform the fully connected architecture.

For instance, the Convolutional Neural Network is a very popular architecture for image inputs and can be described as follows: Given a two-dimensional image X as input,

$$Y_{1,i} = g_1(X * W_{1,i}), 1 \leq i \leq N_1 \tag{2.6}$$

$$Y_{k,j} = g_k(\sum_i (Y_{k-1,i} * W_{k,j})), 1 < k \leq K \text{ and } 1 \leq j \leq N_k \tag{2.7}$$

Here, $*$ is the two-dimensional convolution operation. Thus, all these layers perform a convolution of the input images with a set of kernels W that form the set of trainable parameters. These convolutional layers are followed by a set of fully connected layers that again produce as many outputs as required from the approximator function.

2.1.5 Training Neural Networks

2.1.5.1 Gradient Descent

This section focuses almost exclusively on gradient-based methods for training neural networks. Given a function $Y = f_W(X)$ that is parametrized by W, fitting it to approximate a set of samples (X_i, Y_i) drawn from a dataset $\mathbf{D}$ will have a fitting error:

$$e = E(f_W(X_i), Y_i) \tag{2.8}$$

In the standard gradient descent technique, we iteratively move in the direction of the local gradient of the error function until convergence as follows:

$$W_{t+1} \leftarrow W_t - \eta \nabla_W E(f_W(X_i), Y_i) \tag{2.9}$$

Here, we simply move in the direction of the gradient by a length that is proportional to η, a hyperparameter called the learning rate.

There are a number of issues with this simple process: there is a strong possibility that such gradient descent enters a sub-optimal local minimum and is unable to escape it. In addition, the descent even towards a local minimum might often slow down near saddle points and require a large number of steps to achieve convergence.

Several incremental techniques such as momentum, Nesterov momentum, Adagrad, RMSProp and Adam were proposed over the last decade to overcome these issues and to speed up convergence. As an example, the momentum technique proposes the velocity of the gradient as:

$$V_t = \beta V_{t-1} + (1 - \beta)\nabla_W E(f_W(X_i), Y_i) \tag{2.10}$$

This velocity is subsequently used to perform the gradient descent step as :

$$W_{t+1} \leftarrow W_t - \eta V_t \tag{2.11}$$

The advantage of using momentum instead of standard gradient descent is that we have an exponentially decaying memory of past steps that influences the direction in which we make our next step. It has been shown empirically that using momentum increases the convergence rate in many problems. Full descriptions and analyses on other incremental techniques mentioned earlier can be found in [13].

2.1.5.2 Backpropagation

Calculating the gradient for a function that is as complex as a neural network is a non-trivial task. This is accomplished by a Dynamic Programming algorithm called Backpropagation that expresses the gradient at each layer as a function of the gradient at a later layer.

In practice, backpropagation is achieved through automatic differentiation in code through packages such as Tensorflow or Pytorch.

2.2 Reinforcement Learning

Reinforcement Learning is the study of how to write software agents that can take optimal decisions under uncertainty so as to maximize the cumulative reward. It deals with random processes where an agent that is operating within an environment has a number of actions that it can choose from in each state of the environment. By taking actions over time, the agent is able to exert influence on the environment to various degrees and perhaps reach states that favor its goals.

2.2.1 Markov Decision Processes

A fundamental probabilistic concept required to understand RL is the Markov Decision Process (MDP). It is a stochastic, stateful, time-invariant process that can be explained as follows:

- An agent takes actions in an environment over time.
- At a given timepoint t, the process is in state s_t.
- The agent chooses an action a_t. The set of choices A_{s_t} available to the agent may depend on s_t.
- The process proceeds to the next state s_{t+1} and the agent receives a reward r_t from the distribution $R(s_t, a_t)$.

The transition from s_t to s_{t+1} need not be deterministic even given the action a_t. These state transitions are governed by a probabilistic state transition matrix $P_a(s_t, s_{t+1})$. Given the present state s_t, no new useful conditional information about the state transitions or the reward distribution is obtained by further knowing $s_{t-1}, s_{t-2}...s_0$, $a_{t-1}, a_{t-2}, ...a_0$ or $r_{t-1}, r_{t-1}, ...r_0$, thus making the process Markovian.

The goal of the agent given all these is to maximize $\sum_{t=1}^{T} r_t$ for a finite horizon MDP and $\sum_{t=1}^{\infty} \gamma^t r_t$ for an infinite horizon MDP. Here, γ is called the discount factor and is used to differentially weight near-term rewards against long-term rewards so that the notion of cumulative reward that we use continues to be bounded.

A Markov Decision Process is defined using the following parameters:

- The state-space S.
- The set of actions available from each state, A_s.
- The probabilistic state transition matrix given that action a was taken, $P_a(s, s')$. Elements of this matrix give us the probability that given that we took action a from state s, we reach s' in the next step.
- The reward distributions for each state transition given that action a was taken, $R_a(s, s')$.

MDPs are not constrained to have a finite or even a discrete set of states and actions, i.e. the sets S or A_s are not constrained to have a finite number of elements or be countable. That said, this thesis only deals with MDPs that are finite state, finite action, discrete state, discrete action and finite horizon.

2.2.2 Policies

A policy for a Markov Decision process is defined as conditional probability distribution over the action space, given the state. Alternately, it can be defined as a function from state-action pairs to the closed interval of $[0, 1]$. Good policies perform this mapping such that the expected cumulative reward is high, while an optimal policy must achieve the maximum expected cumulative reward from any initial state.

Formally, a policy can be stated as

$$\pi : S \times A \to [0, 1] \tag{2.12}$$

$$\pi(a, s) = Pr(a_t = a | s_t = s) \tag{2.13}$$

2.2.3 Discount Factor

The discount factor is a exponential multiplicative constant used to assign lower importance to rewards obtainable in the far future compared to rewards that can be obtained in the near future. Using a discount factor encodes the intuition of favoring near-term rewards over longer-term

rewards of equivalent magnitude. In addition, it bounds the cumulative reward in case of infinite horizon MDPs. With the discount factor, the cumulative reward becomes

$$R_t = \sum_{t=1}^{T} \gamma^i r_t \tag{2.14}$$

Some formulations include the discount factor as part of the MDP itself while others allow it to be a tunable parameter that does not have to be fixed first to fully determine the MDP, but they both lead to identical outcomes with regards to optimal policies given the same value for the discount factor.

2.2.4 Value Function

The value function $V_\pi(s)$ is a function that maps states to real numbers. The value function is parametrized by a policy π and is equal to the expected cumulative reward that can be obtained by starting from a state s and faithfully following the policy π. Formally,

$$V_\pi(s) = E[R|s_0 = s, \pi] \tag{2.15}$$

$$= E[\sum_{t=1}^{T} \gamma^i r_t | s_0 = s, \pi] \tag{2.16}$$

The optimal value function is defined as the function $V^*(s)$ whose value for each s is the maximum of the expected cumulative reward obtainable from that state across all policies.

$$V^*(s) = \max_\pi V_\pi(s) \tag{2.17}$$

The policy that achieves this maximum is called the optimal policy and is denoted by π^*.

$$V_{\pi^*}(s) = \max_\pi V_\pi(s) \tag{2.18}$$

2.2.5 Q-Values

The value function denotes the expected cumulative reward obtainable when beginning in a state s and following a policy π. On the other hand, the Q-Values under a policy π are defined as the expected cumulative reward obtainable when beginning in a state s, taking an action a, and then following a policy π afterward. Formally,

$$Q_\pi(s,a) = E[R|s_0 = s, a_0 = a, \pi] \tag{2.19}$$

$$= E[\sum_{t=1}^{T} \gamma^i r_t | s_0 = s, a_0 = a, \pi] \tag{2.20}$$

Naturally, the value function and Q-Values can be related by marginalizing over the policy's action distribution for the initial state as follows:

$$V_\pi(s) = E_{a\sim\pi(s,a)}[Q_\pi(s,a)] \tag{2.21}$$

$$= \sum_{a\in A} \pi(s,a) Q_\pi(s,a) \tag{2.22}$$

Similar to how we defined the optimal value function, the optimal Q-Values are defined as the maximum possible expected cumulative rewards achievable from a state s by taking an action a, over all possible policies.

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \tag{2.23}$$

The policy that achieves this maximum is called the optimal policy and is denoted by π^*. It must be noted that a policy that maximizes the Q-Values will also maximize the Value function.

$$Q_{\pi^*}(s,a) = \max_\pi Q_\pi(s,a) \tag{2.24}$$

Given the optimal Q-Values it is straightforward to construct a policy from them: the agent should just choose the action with the maximum optimal Q-Value for that state.

$$\pi^*(s,a) = \mathbb{1}(Q^*(s,a) = \max_{i \in A} Q^*(s,i)) \tag{2.25}$$

2.2.6 Function Approximators for RL

An optimal policy for an MDP can be created if we can accurately estimate $Q^*(s,a)$, and traditional RL algorithms attempt to do exactly that by using sampled trajectories from an MDP to estimate $Q^*(s,a)$ for all $s \in S$ and $a \in A$. Such methods are called tabular methods since their operation is akin to store all the Q-Values in a table and looking them up to create a policy. However, these tabular methods have intractable memory requirements when faced with MDPs that have large state spaces. In addition, it is impossible to encounter all state-action pairs in case of large problem domains, thus the convergence guarantees for these methods do not apply.

To solve this issue, instead of storing the Q-Values for all the state-action pairs, one variant of the function approximation RL paradigm seeks to fit a parametrized function to the estimates of the Q-Values. The goal here is to find a function $f_p(s,a)$ such that

$$f_p(s,a) = \min_{p'} (Q^*(s,a) - f_{p'}(s,a))^2 \tag{2.26}$$

Firstly, if the set of parameters p is much smaller than the total number of state-action pairs, then this function solves the memory conundrum. Secondly, if the function approximator has any generalization capacity, then that can cause the RL agent to predict the Q-Values accurately for at least some state-action pairs that were never encountered during the training phase.

2.2.7 Deep Reinforcement Learning

If a neural network is used as the function approximator in the approach described in 2.2.6, it is called Deep Q-Learning. Another approach, called policy gradients, uses a neural function approximator to take a state as input, and produce a probability distribution over the actions as the

output, thus directly providing a policy that can be used to take actions.

Research over the past several years has led to other neural function approximation approaches for RL of increasing complexity and performance. These include Deep Deterministic Policy Gradients (for MDPs that have continuous action spaces), Proximal Policy Optimization, Soft Actor-Critic, etc.

2.3 Elements of Graph Theory

A graph is defined as a structure that consists of a set of objects some of which have a pairwise relation between them. The objects are called vertices while the pairwise relations are called edges. A graph $\mathcal{G}$ is mathematically described as $\mathcal{G}(V, E)$ where V is the set of vertices, and E is the set of edges between the vertices in V.

Graphs have wide-ranging applications and can describe communication networks, transportation networks, inter-personal relationships, structural and functional dependencies, etc. Below are some mathematical concepts that are commonly used in topics related to graphs and graph theory.

2.3.1 Vertex

A vertex in a graph is an object that may be related to other objects in the graph. The set of all vertices in a graph is commonly denoted by V. As an example, in a graph describing a transportation network between cities, the cities themselves are the vertices. Irrespective of what named entities constitute vertices, they can all be assigned natural numbers in the range $1, 2, ...|V|$ to identify them individually. The ordering of vertices based on these assigned indices has no relevance unless stated otherwise.

2.3.2 Edge

An edge is a pair of vertices that are related within the graph. This pair can be ordered, in which case the graph is said to be “directed". Formally, an edge is represented as (u, v) where u and v are the vertices that are related. An edge may also have an additional real number w attached to it, called the weight of the edge. Often, the weight of an edge is used to denote how strongly the relation between the two endpoints is.

The vertices are also known as the endpoints of the edge. While multiple edges can exist between a single pair of vertices, we restrict ourselves to undirected graphs in which only one edge can exist between a pair of vertices. In addition, we rule out the existence of edges from a vertex to itself (called a self-loop). For a given vertex v, all the vertices with which v has an edge are called its neighbors.

As an example, in our transportation network example, the roads between any pair of cities constitutes an edge, which is undirected in this case. The weight of this edge could denote quantities such as the daily traffic that flows between these cities, the width of the road between these cities, the average time it takes to travel between these cities, the length of the road, etc.

2.3.3 Degree of a Vertex

The degree of a vertex is defined as the number of edges in the graph for which that vertex is an endpoint. For an undirected graph without self-loops,

$$d(v) = |\{(u, v) : (u, v) \in E\}| \tag{2.27}$$

The minimum and maximum of the degree of all vertices in a graph are called the minimum degree and maximum degree of a graph. A graph all of whose vertices have the same degree is called a regular graph. The degree of vertices in any graph (without self-loops or multiple edges between the same pair of vertices) can range only between 0 and $|V| - 1$. A graph where all vertices have a degree of $|V| - 1$ is called a complete graph. It is commonly denoted as K_n where n is the number of vertices in the graph.

2.3.4 Path

A path in a graph is a sequence of vertices $v_0, v_1, ...v_L$ such that $\{(v_0, v_1), (v_1, v_2), ...(v_{L-1}, v_L)\} \subseteq E$. In other words, a path is a sequence of vertices such that it is possible to "travel" from the first vertex in the sequence to the last vertex through all the intermediate vertices in the sequence while moving only to neighboring vertices at all steps. It must be noted here that the elements of a path sequence need not be unique. In case all the elements of the sequence are indeed unique, it is called

a simple path.

2.3.5 Cycle

A cycle in a graph is a simple path with at least three vertices such that the first vertex in the path is a neighbor of the last vertex. A graph is said to be acyclic if it contains no cycles. A graph is said to be a cycle graph if all the vertices in the graph are part of the same cycle, and every vertex has only two edges. It is denoted using C_n where n is the number of vertices in the graph. C_3 is commonly called a triangle.

2.3.6 Tree

A tree is a connected graph where there is only one simple path between any two vertices in the graph. The name comes from the fact that a tree graph can be drawn to resemble a real-life tree that has a root that splits into branches that split into more branches and so on.

2.3.7 Connectedness

Two vertices are said to be connected if there exists at least one path between them. Formally, u and v are said to be connected if $\exists(u, v_0, v_1, ...v_L, v)$ such that $\{(u, v_0), (v_0, v_1), ...(v_{L-1}, v_L), (v_L, v)\} \subseteq E$ where $L \geq 0$.

A graph as a whole is called a connected graph if all vertices in the graph are connected to all other vertices. Connected graphs are the interest space in this thesis as vertices that cannot be reached by any agent in a multi-agent problem are irrelevant to the scope of the problem.

2.3.8 Adjacency Matrix

It was noted earlier that the set of vertices V in a graph can be assigned consecutive natural numbers in the range $1, 2, ...|V|$. The adjacency matrix $\mathbf{A}$ is a matrix that is constructed using the rule $\mathbf{A}_{ij} = \mathbb{1}((i, j) \in E)$ where $\mathbb{1}(x)$ is the indicator function that has a value of 1 if x is a true proposition and 0 otherwise. An element in the adjacency matrix is 1 if and only if there exists an edge between the vertices corresponding to the row index and the column index of the element. In case the graph contains weighted edges, the element of $\mathbf{A}$ at the corresponding position is set to

the weight of the edge, i.e. $\mathbf{A}_{ij} = w \; \forall (i, j, w) \in E$.

All information about a graph can be derived from the adjacency matrix.

2.3.9 Adjacency List

An adjacency list is an alternate method of representing all information about a graph. For every vertex u in the graph, the adjacency list contains a set of vertices that u has an edge to, along with any weight for the edge.

An adjacency list uses far less memory than an adjacency matrix when the number of edges in the graph is much smaller than the number of possible edges.

2.3.10 Embedding of a graph

An embedding of a graph onto a surface is a representation of the graph such that

- All the vertices of the graph are mapped to unique points on the surface.
- All the edge of the graph are mapped to unique arcs on the surface.
- The endpoints of each arc maps to the points that correspond to the endpoints of the corresponding edges.
- No two arcs intersect except at a point that corresponds to a vertex on which both edges are incident upon.

2.3.11 Planar Graph

A graph is said to be planar if it has an embedding onto the Euclidean space. A consequence of a graph being planar is that a visual representation of the graph can be drawn onto a plane surface such that no two edges intersect.

2.4 Concepts from Game Theory

A game is a mathematical representation of a situation where rational actors must make decisions as they attempt to maximize the payoff that they receive at the end of the game. An example of a game is the situation of two vehicles that have the choice of either proceeding or waiting at a

traffic intersection. In case both vehicles simultaneously choose to move forward, they will have an accident. In case only one vehicle chooses the move forward and the other chooses to wait, they are both safe, but the waiting vehicle expends time. In case both vehicles choose to wait, they both lose time and need to play the game again.

2.4.1 Players

A player is a rational agent that is allowed to take decisions in a game. All players have the objective of maximizing their own payoff in case of deterministic games, and maximizing their expected payoff in case of games with stochastic elements.

2.4.2 Payoffs

A payoff can be thought of as the total reward that is granted to a player by the end of a game. The payoff that a player receives arises from the actions of the particular player, as well as the actions of other players.

2.4.3 Strategies in Games

A strategy is an option/action that a player can choose when facing a particular situation in game. Good strategies take into account the actions that other players can take and the corresponding consequences in the context of the action that the player is choosing.

A set of strategies that assigns only one strategy to all players involved in the game is called a strategy profile.

Following are definitions that are commonly used in the context of strategies for games:

- **Pure Strategy**: A pure strategy is a function that maps every situation that a player may face in a game to an action that the player would take in that situation. Naturally, a pure strategy is deterministic - in case a player has a pure strategy, knowing the player's strategy determines the action they will take in all situations.

- **Mixed Strategy**: A mixed strategy is a probability distribution over the set of possible actions that a player has. The player picks an action by randomly sampling an action from

this probability distribution. This means that the action that a player will take is not determinable by any agent before the action is taken. Formally, the mixed strategy for player i is a probability distribution P on Σ^i. There are some game formulations where upon playing many games, an intelligent player would always converge to a mixed strategy - rock, paper, scissors is an example. Here, in case one player chooses a deterministic strategy, the other player will quickly observe the pattern and choose a counter-strategy that dominates it.

- **Dominated Strategy**: A strategy P_i for player i is said to be dominated by another strategy S_i if over all strategies that opponents follow, S_i always leads to a better outcome than P_i.

2.4.4 Normal-form representation

In the normal form representation, we use a matrix called the Payoff Matrix to represent the payoffs of each agent arising from each decision that each player can take. For each set of choices assigned to the players, the corresponding element in the matrix contains a sequence of payoffs that depicts what each player will receive.

2.4.4.1 Example

The prisoner's dilemma is a game set in a situation where two suspects from a criminal gang have been arrested. The prosecutors do not have enough evidence to convict either prisoner on the principal charge. However, there is enough evidence to convict them on a minor charge. The prosecutors give both prisoners the following choice: In return for turning on their partner and giving sufficient evidence to convict them on the principal charge, the defector will be released without any charges, causing the silent partner to be jailed for 5 years. In case both suspects turn on each other, they both spend 3 years in jail. In case both are silent, they only serve 1 year under the minor charge.

The decisions and payoffs for this game can be represented in the normal-form in the following manner:

Prisoner B / Prisoner A	Remain Silent	Defect
Remain Silent	(-1, -1)	(-5, 0)
Defect	(0, -5)	(-3, -3)

Table 2.1: Payoffs for the prisoner's dilemma represented in Normal-Form

2.4.5 Extensive-form representation

The extensive-form representation is commonly used to represent sequential games as they require decisions to be made by different players at different timesteps, and these decisions themselves alter the set of available actions to each player. In this representation, the game's possible paths of evolution over time are depicted as a decision tree. Every level in the tree corresponds to a decision that is made by a single player. From any node in the tree, the branches that lead to the lower level correspond to the different decisions available to that player from this state of the game.

2.4.6 Types of Games

Games can be labeled and differentiated on the basis of several attributes that describe the nature of the game.

- **Perfect Information**: Games where all players know every move that has been made by every player.
- **Complete Information**: A game where all the available strategies and payoffs available to every player is known to every player in the game.
- **Zero-Sum**: Whatever strategy the various players of the game choose, the sum of the payoffs received by all the agents is equal to zero. This means that one agent receives a positive payoff only at the cost of other players.

- **Simultaneous/Sequential**: A game is said to be simultaneous if agents playing the game move at the same time, thus not being aware of the actions of other players. It is sequential if players make moves in their turn, thus allowing for each player to know the action that the earlier player took.

- **Symmetric/Asymmetric**: A game where interchanging the identity of players does not change the payoffs they receive for the same strategy. All other games are said to be asymmetric.

- **Discrete/Continuous**: A game where the number of players, actions, situations and outcomes are finite and thus can be modeled in a discrete sense.

2.4.7 Zero-sum Games

A zero-sum game is one where the gain or loss in utility of one player is exactly balanced by the loss or the gain that all other players have in total. In other words, adding up the utility received by all players will result in a sum of zero.

An example of a zero-sum game is chess where victory for one player automatically leads to defeat for the other and vice versa (We assume that draws are of no utility to either player). Trade is often thought of as a non-zero-sum game since two entities with a surplus of a different goods are able to exchange them with each other for mutual utility. Here, there is a strategy profile that leads to a gain for all players.

In the normal form representation for a zero-sum game, the sequence within every cell in the payoff matrix must sum to zero. In an extensive-form representation, the payoff sequence at all leaf nodes must sum to zero.

Formally, if $S_1, S_2, ...S_n$ are strategies for players $1, 2, ...n$ and π_i represents the payoff that the i^{th} agent receives at the end of the game, then

$$\sum_{i=1}^{n} \pi_i(S_1, S_2, ...S_n) = 0 \tag{2.28}$$

Two player zero-sum games can be formulated differently: The outcome of the game is described just in terms of the payoff that a designated first player receives. The goal of the first player is then to maximize the outcome of the game while the goal of the other player is to minimize it. This formulation lends itself to easier representation for algorithmic analysis although the dynamics of the game and the optimal strategies remains the same.

2.4.8 Minimax and Maximin

Consider the alternative formulation of two player zero-sum games proposed previously: Instead of describing the payoffs every agent receives, we designate the payoff received by one specific player as the outcome of the game. Thus, if the outcome of the game is a positive quantity, that means that the first player has benefited from the game at the cost of the second player, while if the outcome of the game is a negative quantity, the second player has benefited from the game at the cost of the first player. In this formulation, the first player is called the *maximizing* player while the second player is called the *minimizing* player.

In such a game, two quantities are interesting: the maximin, defined as

$$\underline{v_1} = \max_{a_1} \min_{a_2} v_i(a_1, a_2) \tag{2.29}$$

and the minimax, defined as

$$\overline{v_1} = \min_{a_2} \max_{a_1} v_i(a_1, a_2) \tag{2.30}$$

Here,

- a_1 denotes the action of the maximizing player.
- a_2 denotes the action of the minimizing player.
- v_1 is the value arising for the maximizing player from this combination of actions.

The maximin quantity is the highest game outcome that can be achieved when the minimizing

player knows the action that the maximizing player is going to take. Alternately, it can be stated to be equal to the best case (maximum) game outcome that the maximizing player can guarantee when they do not have any knowledge of what action the minimizing player is going to take.

The minimax quantity is the lowest game outcome that can be achieved by the minimizing player when they do not know anything about the maximizing player's strategy. Alternately, it is equal to the best case (minimum) game outcome that the minimizing player can achieve when their strategy is known to the maximizing player.

These statements are general and apply both to simultaneous and sequential games.

3. RELATED WORK

This chapter summarizes pertinent prior research work on the cops and robbers problem, and other work in reinforcement learning on graph domains.

3.1 Vertex-to-vertex pursuit in a graph

The cops and robbers problem was first studied in [1]. This paper introduces the problem, provides a set of general theorems that allow characterization of graphs into cop-win and robber-win, and uses these rules to distinguish a few classes of graphs as cop-win and robber-win.

The following concepts are useful in understanding the results in this paper:

- The *Retract* of a graph G is a mapping from G onto H such that the edge connectivity of G is preserved in H, allowing some vertices from G to be mapped to the same vertex in H. In a retract, vertices can be merged together while preserving their edge connectivity.

- A *Graph Product* is a binary operator that takes two graphs G_1 and G_2, and produces a graph H. H's vertices are formed out of the cartesian product of the vertices in G_1 and G_2, i.e. for every pair of vertices v_1 and v_2 s.t. $v_1 \in G_1.V$ and $v_2 \in G_2.V$, there is a vertex (v_1, v_2) in H. In addition, a rule dictates whether a given pair of vertices in H, (u_1, u_2) and (v_1, v_2) have an edge. This rule's result depends on the edges in G_1 and G_2.

- A vertex v is said to dominate a vertex u if the neighborhood of u, $\mathcal{N}(u)$ is a proper subset of the neighborhood of v, $\mathcal{N}(v)$.

- A vertex v is *irreducible* if there exists some other vertex u that is dominated by v.

- A graph G is dismantlable if and only if there exists an ordering $\{v_1, v_2, ...v_n\}$ for all its vertices such that in the ordering, for each $i < n$, v_i is irreducible in the induced subgraph formed out of the vertices $v_i, v_{i+1}, ...v_n$. Alternately, it can be stated that there is a sequence of vertices $v_1, v_2, ...v_n$ such that the vertices can be sequentially be folded into a neighboring vertex that dominates it.

Using these concepts, the authors show that given a pair of cop-win graphs G_1 and G_2, all the graphs that can be produced out of retracting any finite product of G_1 and G_2, are also cop-win.

In addition, they showed that for any graph, if there is a retract that has a cycle of more than length 4, then the graph is robber-win. Further, they show that graphs that are a finite product of two graphs that are simple paths themselves is a cop-win graph.

The authors also produce an alternate formulation of the above statements in which it is stated that a graph is cop-win if and only if it is dismantlable. The ordering associated with the dismantling of a graph is called the cop-win ordering.

Finally, the authors show that every regular graph that is not complete is robber-win.

3.2 State abstraction for real-time moving target pursuit: A pilot study

To allow scalability for minimax-like approaches, [6] attempts to produce good strategies for the target agent (robber) and proposes the concept of *abstract graphs* that are produced by combining pairs of vertices in the original graph to form super-vertices. They exploit the fact that if a robber can evade the cop at a higher-level of abstraction by making a move, then it can evade the cop in all lower levels as well by moving to some vertex in that super-vertex.

Their approach, called Dynamic Abstract Minimax (DAM) proceeds as follows:

1. At a given level of abstraction perform a minimax search on the graph to find out whether the robber can evade the cop.

2. If yes, the super-vertex that allows the target to evade the pursuer is the one to which it should travel.

   (a) Choose a random vertex from the set of vertices corresponding to the super-vertex.

   (b) Use Partial Refinement A* to plan a path to this vertex.

   (c) Choose the action recommended by PRA*

3. If no, then go to the immediate lower level of abstraction and repeat steps 1 to 3.

Their agent thus searches for the existence of winning strategies using minimax on the abstract graphs, and successively move to lower levels of abstraction only if the robber cannot evade the cop on that level.

3.3 Evaluating strategies for running from the cops

[4] proposes the Trailmax algorithm for robbers that performs a Minimax over a quantity called *Trail* which relies on path counts to make decisions. They begin by assuming that the robber's strategy is known to the cops and optimize the robber's actions under that constraint.

The trail $T(p_c, p_r)$ for a pair of paths p_c and p_r for the cop and robber respectively is defined as the total number of turns taken by both agents until capture occurs. Since the robber's goal is to find a path that maximizes this quantity while the cop's goal is to minimize it, the heuristic lends itself naturally to a minimax formulation. However, this heuristic is guaranteed to return the optimal value of the game only for octile graphs, i.e., graphs that are superimposable on a grid structure and have edges from any vertex only to its nearest eight neighbors on the grid.

The authors implement the heuristic search using the abstract graphs technique discussed in 3.2. They get a robber evasion rate of 96.7% with a search depth of 20 steps, as measured on octile graphs.

3.4 A cover-based approach to multi-agent moving target pursuit

To allow several cops to coordinate their pursuit of a robber, [5] uses a *cover* heuristic: it is the number of vertices that are reachable by some cop before the robber. They then use this heuristic with algorithms such as Alpha-Beta pruning, Greedy and Abstraction to choose the best action to take to catch the robber.

The authors do not use an optimal robber opponent. Instead, they use a robber motivated by simple strategies such as heading to a beacon vertex that is farthest away from cops, Dynamic Abstract Minimax or a Greedy Strategy where the robber simply runs away from the pursuers. They evaluate their algorithm on grid worlds by comparing the number of steps required by a team of optimal cops to catch a this robber, against the number of steps their algorithm requires. They

note that the optimal cops always catches their robber, and show a success rate of 84.3%.

3.5 POMCoP: Belief Space Planning for Sidekicks in Cooperative Games

[14] uses the cops and robbers game to study collaboration between human and artificial agents. They first model human actions into several archetypes of humans from instances of collaboration, then build an optimal policy for each archetype through a UCT search.

They propose the POMCoP algorithm which alters the transition probabilities for the human based on which human archetype the agent believes is playing with it. Given this new transition probabilities, the agent uses a UCT search to choose the action that should be taken for catching the robber in collaboration with the human.

3.6 Deeppath: A reinforcement learning method for knowledge graph reasoning

[15] uses a Policy Network to train a single agent to walk a relational knowledge graph. Their goal here is to identify the “best" path from one vertex that represents a particular logical entity to another such vertex. Global accuracy, path efficiency (length) and path diversity metrics are considered while deciding what a reasonably good relational path is.

3.7 Graph convolutional reinforcement learning for multi-agent cooperation

[16] models the interactivity between neighboring agents using a graph, then uses Graph Convolutional Networks to train these agents to cooperate. They encode the intuition that only agents in a k-hop neighborhood, where k is a small integer, are relevant when we consider cooperative actions with other agents. They demonstrate their results on a number of simple games that are all played on grid environments.

4. FORMAL STATEMENT OF THE COPS AND ROBBERS PROBLEM, MDP FORMULATION AND IN-MEMORY REPRESENTATION

This chapter describes the cops and robbers problem that is being solved in this thesis, its properties and applications, followed by definitions of how it is represented mathematically and in memory.

4.1 Problem Description

The cops and robbers problem is a turn-based pursuit and evasion game played on a graph by two teams of agents, the cop team and the robber team. In one variant of the problem, the n agents are placed randomly onto vertices in a connected graph $G(V, E)$. All agents make moves during their turn by moving from their current vertex v to any neighboring vertex, i.e., any element of $\mathcal{N}(v)$. The order of turns is such that all cops act one after another sequentially before the robbers begin acting. Whenever a cop is present on the same vertex as a robber, the robber is "arrested" and removed from the game immediately. The goal of the cop team is to cause all the robbers to be arrested, while the goal of the robber team is to make as many robbers as possible evade arrest indefinitely. To keep the problem solvable in practice, we limit "indefinitely" to a predefined number of turns that is reasonably larger than the diameter of the graph that the agents play on. We will also limit ourselves to the case where there is only one robber on the graph as the cops can use an optimal strategy to capture all the robbers one by one in case there were multiple robbers.

The most common variant of the game has the following attributes:

- Multi-player: The game has several agents all of which need to make their own decisions.
- Complete information: All agents have complete visibility of the graph as well as the positions of all agents, and thus can acquire knowledge of the strategies and moves available to all agents.
- Perfect information: Agents observe all moves of all agents over the game's history.

- Markov state: Given the network structure and the positions of all agents, no other historical information is required for agents to play optimally.
- Zero-sum: The cops and robbers have rewards of equal magnitude and opposite signs when the game ends. There is no outcome to the game in which the cops and the robbers both get rewards that do not sum to zero.
- Asymmetric: Players in the game have different roles and are not equal.
- Discrete: No element of the game is continuous.
- No-chance: While random variants of cops and robbers do exist, we primarily perform research upon the variant where actions result in deterministic state transitions. When an agent chooses to move to a particular vertex, it always succeeds.
- Extensive-form: The game proceeds for many turns or rounds until it ends. This is as opposed to a normal-form game where agents take only a single decision.
- Sequential: Agents act after the previously acting agent has completed its action and not simultaneously.
- Combinatorial: The set of moves available to each agent changes depending on the state.

4.2 Environment Categories

Different combinations of graph structures and start positions lead to different results for the game if one of teams plays optimally. There is a set of graphs where given a particular number of cops N_{cops}, an optimal strategy for the cop team that results in the (single) robber's arrest always exists. These configurations are called N_{cops}-cop-win graphs. In all other graphs, the robber has a strategy to evade the cops indefinitely. These graphs are called N_c-robber-win graphs.

Note that on cop-win graphs, the robber's strategy is irrelevant if the cop's strategy is optimal as the cop team is guaranteed to arrest the robber. Similarly, on robber-win graphs, the optimal robber is guaranteed to evade the cops indefinitely irrespective of cop strategy. In addition, for

nomenclatural simplicity, 1-cop-win and 1-robber-win graphs are commonly called cop-win and robber-win graphs respectively.

Some results that have been demonstrated from previous algorithmic work are:

- Trees are cop-win. The cop should attempt to force a situation where it resides on an ancestor vertex of the robber's vertex first, then traverse the only path that exists to the robber's vertex.
- A graph that does not contain any cycle with more than 3 vertices is cop-win.
- Cycles with more than three vertices, denoted by C_n where $n \geq 4$ are robber-win. The robber's optimal strategy is to just travel along the same direction on the cycle that the cop moved in the previous turn. Thus, all grid graphs are robber-win.
- Cycles are 2-cop-win. This is since one cop can maintain its current position while the other can circle around the graph and corner the robber.
- Planar graphs are 3-cop-win.

Such categorization is useful to us as we can evaluate the goodness of different trained agents on these graphs. For instance, our cop agent's strength can be measured by evaluating how well it performs on cop-win graphs against a strong opponent. Getting good opponents to evaluate against is itself not a trivial problem and required additional thought during the course of this project.

4.3 Variations

Upon the basic cops and robbers problem described in the previous section, existing literature describes the following variations that can extend to specific applications:

4.3.1 Quarantine and Cure

The robber here is a replicating, infectious virus that in every turn can randomly and independently spread from each of its current vertices to neighboring vertices with a probability p. Whenever a cop enters an infected vertex, it is disinfected. The virus cannot spread to vertices that contain at least one cop. However, once the cop leaves the vertex, the virus may spread to

that vertex if a neighbor has the virus. In this environment, a limited number of cop agents must disinfect the virus from the entire graph by ensuring that no vertex in the graph is infected.

The virus here replicates according to a simple, uniform strategy, and the focus is on training cop agents for the problem. However, the question of which vertices an intelligent virus must spread to from its current positions is also an interesting problem to solve.

4.3.2 Containment

The robber in this variant of the problem is a randomly moving internet user who is browsing the web. In each robber turn, the user moves to a neighboring vertex akin to clicking hyperlinks on the web. Once in a while, the user teleports to a random vertex on the graph by typing something into the search bar. A small team of cops have the ability to teleport to any vertex on the graph during their turn. They play the role of a browser prefetcher here. The goal of the cops is to ensure that the user almost always steps into a vertex that is currently occupied by a cop.

This problem again focuses on training cops as the robber does not play the role of a reactive adversary. This is a useful application that hopes to train agents that can efficiently predict and prefetch a small number of webpages while correctly anticipating which ones the user will request next.

4.4 Cops and Robbers as a Markov Decision Process

We return to our basic instance and show our formulation as a Markov Decision Process. For further background on MDPs, its components and mathematical formulation, please refer to **Sec. 2.2.1**. Recall that to describe an MDP, we need to decide the State Space S, the action space A_s, the reward distribution $R_a(s, s')$ and the transition matrix $P_a(s, s')$.

4.4.1 States

The state s_t at any timestep t is the structure of the graph $\mathcal{G}$ and the positions p_i of all agents that are still playing the game. Due to the perfect information nature of the game, the state information fully observable and is identical for all agents and requires no further demarcation at the level of agents. We enforce an arbitrary ordering on all the vertices of the graph in our state representation

that we consistently follow all across our methods. This is done by arbitrarily labeling the $|V|$ vertices in the graph as $1, 2, 3, ...|V|$. We include the graph's structure as part of the state since we want our agents to generalize to graphs that they have never seen during training. Naturally, transitions between states that correspond to different graphs are forbidden as are states where agents jump to vertices that are more than one vertex away. That the process is in a state s_t corresponding to a given network structure and agent positions can thus formally be expressed using the ordered tuple

$$s_t \equiv (\mathcal{G} = G(V, E), p_1 = P_1, p_2 = P_2, ...p_n = P_n) \tag{4.1}$$

or more concisely as

$$s_t \equiv (\mathcal{G} = G, \mathcal{P} = P) \tag{4.2}$$

This state s_t completely describes the present nature of the game and no other additional information is required to infer optimal strategies for the agents.

4.4.2 Actions

For the i^{th} agent when it is at vertex p_i on the graph $G(V, E)$, the sequence (since we order vertices in the lexicographical ordering used to represent the state) of available next vertices is the same as neighbors of p_i. We indicate this sequence by $\mathcal{N}(p_i) = [\mathcal{N}(p_i)]_1, [\mathcal{N}(p_i)]_2, ...[\mathcal{N}(p_i)]_d$, where d is the degree of vertex p_i. $A_s(i)$, the set of actions available to agent i is given by

$$A_s(i) = \{1, 2, ...\mathcal{N}(p_i), \mathcal{N}(p_i) + 1\} \tag{4.3}$$

where action a corresponds to the a^{th} element of the sequence $\mathcal{N}(p_i)$, i.e. $[\mathcal{N}(p_i)]_a$, and the last action corresponds to choosing to stay in the same vertex. There are as many actions available as one more than the number of neighbors for the agent's current vertex. It must be noted that this set $A_s(i)$ is independent of i given $p(i)$, i.e., the action set is the same for all agents given their

position. However, we retain the notation of $A_s(i)$ instead of simplifying it to $A_G(v)$ since we can directly extend it to problem instances that do not have identical action spaces for all agents. For instance, consider a case where one cop may teleport to any vertex in the graph while all other agents can traverse only one vertex per turn.

4.4.3 State Transitions

The state transitions for the instance of the problem that we are solving are all deterministic. Given that the current state is given by

$$s_t = (\mathcal{G} = G(V, E), p_1 = P_1, p_2 = P_2, ..., p_i = P_i, ...p_n = P_n) \tag{4.4}$$

and given that a particular agent i is at vertex p_i and chooses action a, the next state is given deterministically by

$$s_{t+1} = \begin{cases} (\mathcal{G} = G(V, E), p_1 = P_1, ..., p_i = [\mathcal{N}(P_i)]_a, ...p_n = P_n), & \text{if } a < |\mathcal{N}(p_i)| \\ s_t & \text{otherwise} \end{cases} \tag{4.5}$$

4.4.4 Rewards

We noted earlier that we bound the length of each episode to a fixed number of turns τ_{max}. Whether the team of cops gets a reward is determined by whether the robber is caught before all agents have taken this maximum number of turns. Consequently, whether the robber gets a reward is determined by whether it is able to evade arrest until the end of the game. Formally, given that agent r is the robber and all the other agents are cops,

$$R_t(C) = \min(1, \sum_{\substack{c=0 \\ c \neq r}}^{n} \mathbb{1}(p_r = p_c)) \tag{4.6}$$

$$R_t(r) = \mathbb{1}(t = \tau_{max})) \prod_{\substack{c=0 \\ c \neq r}}^{n} \mathbb{1}(p_r \neq p_c)) \tag{4.7}$$

In the case that an episode proceeds for $|\tau_e|$ turns, the episodic reward for both teams is then given by

$$R(C) = \sum_{t=1}^{\tau_e} \min(1, \sum_{\substack{c=0 \\ c \neq r}}^{n} \mathbb{1}(p_r = p_c)) \tag{4.8}$$

$$R(r) = \mathbb{1}(\tau_e = \tau_{max})) \prod_{\substack{c=0 \\ c \neq r}}^{n} \mathbb{1}(p_r \neq p_c)) \tag{4.9}$$

The rewards for the cop team and the robber team are both are scalars common to the whole team and do not need special methods to represent them.

4.5 In-memory Representation

Given the above Markov Decision Process, we need to establish a convention to express all the pertinent quantities in memory. Any proposed convention:

- Must express the entirety of the states, actions and transition probabilities without ambiguity.
- Must not occupy memory space that makes storing huge batches of observations infeasible.
- Must permit rapid calculation of possible actions, state transitions and termination conditions.

4.5.1 States

Recall that the state is expressed as

$$s_t \equiv (\mathcal{G} = G, \mathcal{P} = P) \tag{4.10}$$

We represent the graph structure using the Adjacency Matrix $\mathbf{A}$ of the graph. The adjacency matrix is a square matrix whose dimensions are $|V| \times |V|$ and is given by

$$[\mathbf{A}]_{ij} = \mathbb{1}((i, j) \in E) \tag{4.11}$$

The positions of all n agents are represented together using a $|V| \times n$ matrix called $\mathbf{P}$. Each column denotes the position of one agent such that only the row corresponding to the agent's position in the graph will have a 1, i.e.

$$[\mathbf{P}]_{ij} = \mathbb{1}(p_j = i) \tag{4.12}$$

Both the above matrices are stored as sparse matrices in Python and this means that the memory occupied by $\mathbf{A}$ is $O(V + E)$ and the memory occupied by $\mathbf{P}$ is $O(V + n)$.

4.5.2 Actions

Given that the available set of actions for the i^{th} agent when the process is in a state s is given by

$$A_s(i) = \{1, 2, ..., \mathcal{N}(p_i), \mathcal{N}(p_i) + 1\} \tag{4.13}$$

the action space is represented by a one-hot encoded vector with a length of $D + 1$, where D is the maximum vertex degree of all graphs over which we want the agents to operate. Each action corresponds to moving out of the current vertex through a particular edge to a neighboring vertex. The last action is the no-op action that leads to the agent maintaining its current position. It is favorable to us that the cops and robbers problem is more interesting in graphs that aren't very dense as such graphs have a large number of triangles and thus unfairly favor the cops. Thus, having an upper bound for the maximum vertex degree does not limit our ability to scale to most large graphs of interest.

In case a vertex's degree is less than D, we attach dummy actions to the remaining bits to retain the length of the vector. These dummy actions, if taken, result in the agent staying in its current vertex.

4.5.3 State Transitions

Since the transitions are deterministic, there is no need to store a state transition probability matrix. That said, even if the state transitions were probabilistic, we could functionally encode the transition matrix instead of directly storing it in memory. In other words, given the current state, we could calculate the probability of going to each of the next states by simply taking the set of allowed agent moves and using their probabilities to construct the set of potential next states along with the probability of transition. All other states have zero probability from the current state.

This concise functional encoding as opposed to representing the transition probability matrix explicitly is a commonly used practice in most modern RL environments as it saves significant amounts of memory.

4.5.4 Properties

The aforementioned representation meets our requirements since:

- All possible states and actions can be represented using the sparse matrix convention established previously, with the exception of graphs that contain vertices with very high degrees. Since such unnatural graphs are not in our interest domain, this is not a problem of significance.

- All state transition probabilities can be derived rapidly at runtime from the probabilities of fundamental events, i.e. conditional probabilities for agent movement. In our instance, these transitions are deterministic.

- The memory requirements have been shown to be linear in the size of the graph ($O(V+E)$).

- Given a state, the set of permitted actions and the resultant state transitions can be calculated directly from the adjacency matrix and the position of the agent. The termination condition can be evaluated immediately from the $\mathbf{P}$ matrix.

5. NEURAL NETWORK ARCHITECTURE DESIGN, ITERATIVE ADVERSARIAL TRAINING AND VERTEX POOLING

This chapter offers a detailed discussion of the initial investigation done during our neural network architecture design, the neural architecture we used subsequently, and two main algorithmic techniques that are used in this thesis: Iterative Adversarial Training and Vertex pooling for graphs.

5.1 Graph Convolutional Networks for Graph Navigation

The problem statement for this section is much simpler than our general goal. However, we use this as a testbed to develop a baseline neural network architecture that may perform well on the general problem. By vastly simplifying the problem, we have created a highly controlled test environment that must be solved by any network architecture that is capable of good performance on the general cops and robbers problem.

Consider a problem instance where there is only one cop and one robber, and the robber cannot make any moves. This degenerates into a path planning problem where the cop needs to find a sequence of actions that will result in its movement from its initial position to the position of the robber. We present an algorithm to solve this task as a series of matrix multiplication, matrix lookup and logical operations. We then show that this algorithm can be implemented using a GCN, then discuss the results obtained from such implementation.

5.1.1 Problem Statement

The cop's problem statement can be stated as follows:

On a graph $G(V, E)$, given that the cop starts in vertex v_c and the robber is present in vertex v_r that is d steps away from v_c, find a sequence of actions $a_1, a_2, ...a_N$ such that as the process

proceeds through the sequence of states $X_1, X_2, ...X_N$, the following properties are upheld:

$$p[s_{t+1} = (\mathcal{G} = G(V, E), p_1 = X_{t+1}, p_2 = v_r)|s_t = (\mathcal{G} = G(V, E), p_1 = X_t, p_2 = v_r), a_t] = 1 \tag{5.1}$$

$$X_N = v_r \tag{5.2}$$

$$N < \infty \tag{5.3}$$

In other words, we want the agent to identify an action sequence that causes it to enter the robber's vertex with probability 1 in a finite number of steps. Note that an agent following a policy that has a non-zero probability for every possible action from every state is a valid solution to this problem. This is since the the resultant Markov chain is irreducible and every state is positive recurrent - such an agent, wherever it starts, will eventually arrive at the destination vertex.

To make the problem more interesting, we complicate it by forcing $N = d$. In this case, the cop must find a sequence of actions $a_1, a_2, ...a_d$ such that given $X_0 = v_c$,

$$p[s_{t+1} = (\mathcal{G} = G(V, E), p_1 = X_{t+1}, p_2 = v_r)|s_t = (\mathcal{G} = G(V, E), p_1 = X_t, p_2 = v_r), a_t] = 1 \tag{5.4}$$

$$X_d = v_r \tag{5.5}$$

Alternately, we can say that the cop's goal is to find a sequence of actions whose resultant state transitions result in the cop moving on a path of length d from v_c to v_r.

5.1.2 Pathfinding Methodology using Matrix Operations

To the above problem, we now offer our solution that uses only matrix multiplication, logical and matrix lookup operations. The advantage of such a solution as we show later on is that we can implement it efficiently using a GCN.

5.1.2.1 Inputs

The following are the inputs to our algorithm:

1. **P**, a matrix describing the current position of every agent on the graph. The dimension of this input is $|V| \times 2$ as we have just two agents, and $\mathbf{P}_{j1} = 1$ if and only if vertex j has a cop, and $\mathbf{P}_{k2} = 1$ if and only if vertex k has a robber.

2. **C**, a set of one-hot encoded vectors describing the ordering of the available actions. Effectively, if A_s is the set of actions that are currently possible, this input describes to which vertex in the graph each action $a \in A_s$ will take the agent to. The dimensions of this input are $|V| \times D$ where D is the maximum vertex degree over all graphs of interest. $\mathbf{C}_{ji} = 1$ if and only if an agent can travel to vertex j by taking action a_i. This contextual input determines the ordering relationship between the actions and the neighboring vertices.

3. **A**, the adjacency matrix of the graph. This contains complete information about the structure of the graph. The dimension of this input is $|V| \times |V|$.

5.1.2.2 Solution

Consider a situation where agent j, the cop, is on the vertex v_j. The matrix **P** will then have 1 at $P_{v_j j}$ and 0 everywhere else. Then $(\mathbf{A} + \mathbf{I})^k\mathbf{P}$ gives us a matrix with the following property: Matrix element $[(\mathbf{A} + \mathbf{I})^k\mathbf{P}]_{ij}$ is the number of paths by which we can travel from vertex i to agent j. Alternately, it is the number of paths by which agent j can travel to vertex i, since this is an undirected graph.

Thus, if we have only two agents, a cop and a robber, element i of the first column of **(A + I)P** gives us the number of one-hop paths from vertex i to the cop. Element i in the first column of $(\mathbf{A} + \mathbf{I})^2\mathbf{P}$ gives us the number of two-hop paths from vertex i to the cop. Element i in the second column of $(\mathbf{A}+\mathbf{I})^3\mathbf{P}$ gives us the number of three-hop paths from vertex i to the robber. Thus, these matrices encode useful information that the cop or the robber can use to formulate their strategy.

Given these matrices and the input matrices, the cop, if it is at vertex i, has to perform the following operations to reach the robber's position:

1. Lookup the robber's current vertex using the matrix $\mathbf{P}$. The agent needs to find j s.t. $\mathbf{P}_{j,2} > 0$. This vertex is called T in subsequent discussions as it is the target vertex.

2. Find the cop's distance from the robber's vertex. This is done by looking up the values of elements $[(\mathbf{A}+\mathbf{I})^k\mathbf{P}]_{T,1}$ for different values of k. If $[(\mathbf{A}+\mathbf{I})^k\mathbf{P}]_{T,1} > 0$, the agent has a k-hop path from the current vertex to the target vertex T. To get the length of the shortest path, the agent should choose the least k for which this is true.

3. Find a neighbor vertex n such that the distance from n to T is $k-1$. This can be done by looking up the values of elements $[(\mathbf{A}+\mathbf{I})^{k-1}\mathbf{P}]_{n,2}$ over all n. If for some vertex n, $[(\mathbf{A}+\mathbf{I})^{k-1}\mathbf{P}]_{n,2} > 0$, then the agent has a $(k-1)$-hop path from vertex n to vertex T. Thus, it is a good candidate to move to for the cop's next action. However, this is possible only if vertex n is adjacent to the cop's current vertex.

4. This can be identified by checking whether $[\mathbf{C}]_{n,a} > 0$ for some a. The a for which this is true is the action that must be taken.

In the set of steps described above, the distance from the cop to the robber decreases by 1 after every action. If this initial distance between the cop and the robber was d, a cop following the above approach will take only d steps to reach the robber, which is the optimal number of steps.

Thus, from the input matrices $\mathbf{A}, \mathbf{P}and\mathbf{C}$ to the final action a, there are a series of look-up operations and matrix multiplications that produce the optimal strategy for the path planning problem.

5.1.3 Emulating Pathfinding using GCNs

A standard GCN layer with ReLU activation can be expressed as:

$$\mathbf{H}^{(out)}(\mathbf{W}, \mathbf{A}, \mathbf{H}^{(in)}) = \text{ReLU}((\mathbf{A}+\mathbf{I})\mathbf{H}^{(in)}\mathbf{W}) \tag{5.6}$$

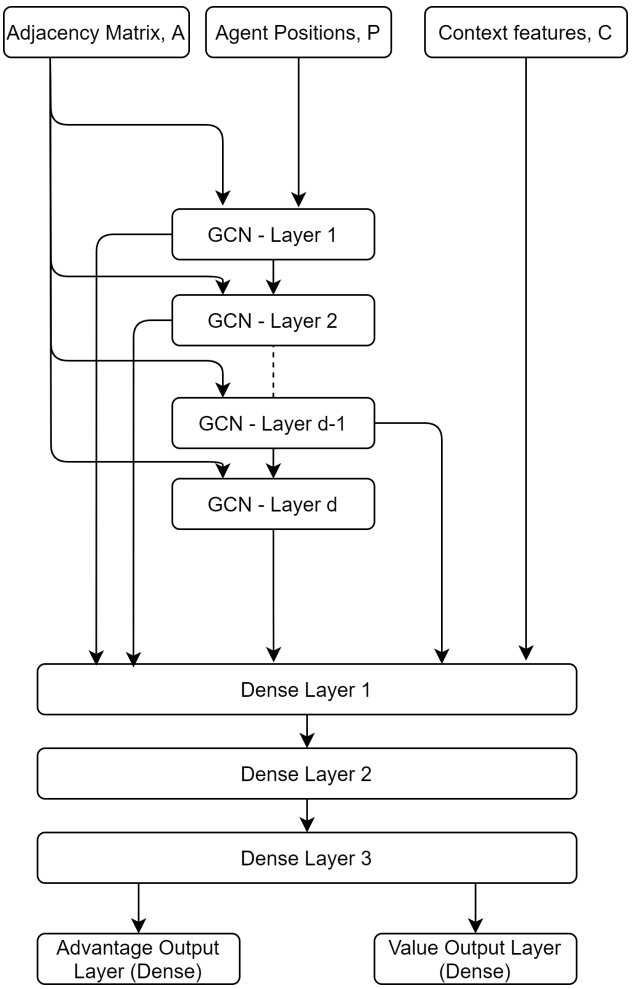

Figure 5.1: Network Architecture used for solving the path planning problem

Consider the architecture depicted in **Fig. 5.1**. The first GCN layer takes **P** as the input. If $W = I_{n\times n}$, the output $h^{(1)}$ is exactly $(\mathbf{A}+\mathbf{I})\mathbf{P}$. This $h^{(1)}$ is fed to the next GCN layer. The outputs of subsequent layers are hence $(\mathbf{A}+\mathbf{I})^2\mathbf{P}$, $(\mathbf{A}+\mathbf{I})^3\mathbf{P}$, ..., $(\mathbf{A}+\mathbf{I})^d\mathbf{P}$. All these matrices are fed as inputs to the dense layer along with C. The only remaining set of tasks is to perform the lookup and logical operations that are required to complete the algorithm. These are handled by the three dense layers that we have.

A natural question that arises is why the dense layers themselves cannot calculate the $(\mathbf{A}+\mathbf{I})^k\mathbf{P}$ matrices and undertake the lookup and logical operations. While the Universal Approximation Theorem does guarantee the existence of a neural network with just a single hidden layer that can

solve this task, we neither know the appropriate width of the hidden layer, nor do we have a method guaranteed to find the model parameters that such a dense network should have. Moving much of the logic for the graphical analysis sub-task to the GCN layers results in the following advantages:

- The GCN layer's parameters are shared across all vertices in the graph, thereby eliminating the need to learn an extremely large number of parameters corresponding to every vertex, while still performing the useful calculation that we need. This is due to the specially tailored nature of GCNs that knows to only focus on local graph structure.
- The hypothesis class $\mathcal{H}$ of possible neural networks is much smaller due to the greatly smaller number of parameters, thereby greatly reducing the chance of overfitting.

5.1.4 Results

We implemented the architecture described in **Fig. 5.1** using Tensorflow and trained the agent using the Proximal Policy Optimization algorithm (Ray RLLib implementation). We limited the maximum episode length to be twice the initial distance $d = 4$.

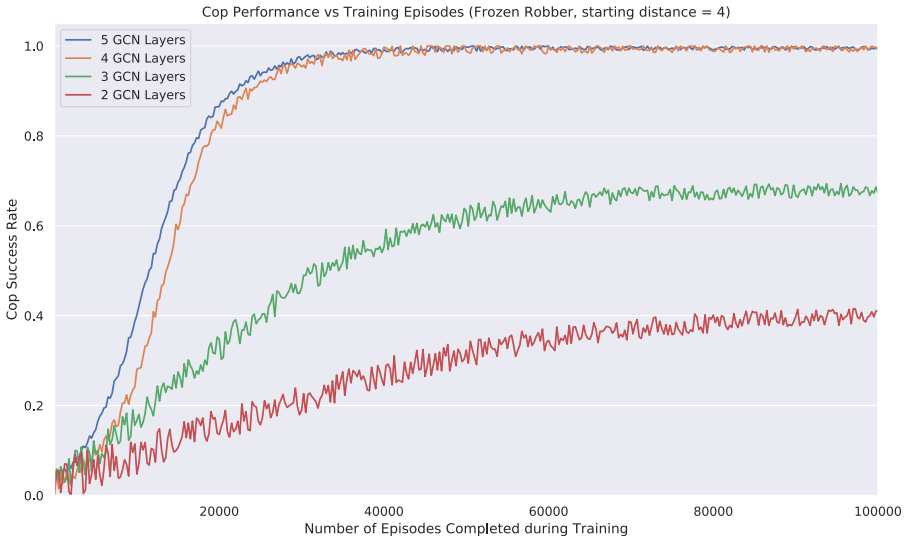

Figure 5.2: Cop's success rate for different GCN depths on the frozen robber problem

As seen in **Fig. 5.2**, when the initial distance between the cop and robber was less than the number of layers in the network, the agent's performance during training converged to a 100% success rate.

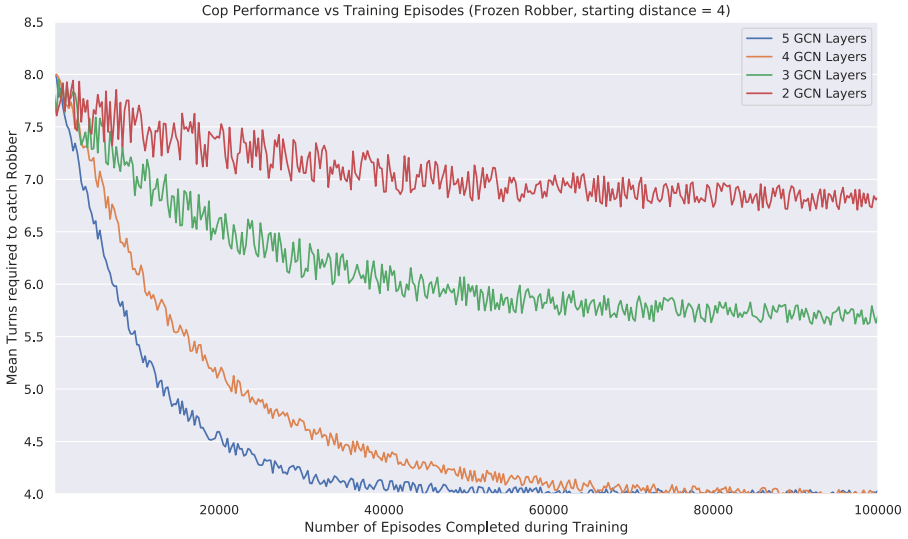

Figure 5.3: Capture Time for different GCN depths on the frozen robber problem

Fig. 5.3 shows that the number of steps or turns required to complete the path also converged to $d = 4$ over the training process. The conclusion obtained is that the chosen architecture is capable of performing efficient path planning from a source vertex to a destination vertex.

5.1.4.1 Limitation on Agent Range

It was observed that for a given initial distance, the performance of the cop depended noticeably upon the number of graph convolutional layers in the neural network. In the plot shown in **Fig. 5.2**, the success rate touches 1.0 when the number of GCN layers ≥ 4. However, when it is less than 4, the success rate dropped sharply. This was expected from the analysis in **Section 5.1.2**: to make inferences about vertices that are a distance of k away, the cop needs to know the matrix $\mathbf{A}^k\mathbf{P}$ and having less than k layers makes this nearly impossible due to reasons discussed in **Sec. 5.1.2.2**.

Such a limitation is not tolerable if we wish to scale our agent to larger graphs as we simply

cannot have as many GCN layers as the diameter of the space of graphs that we run our agent upon. The primary bottleneck in this naïve approach is neither the memory required to implement the large number of GCN layers, nor the performance cost of calculating the outputs of all the GCN layers, but the massive number of weights required to consume all the outputs of the different GCN layers at the dense layer. We implemented a simple, but efficient vertex pooling method to overcome this issue and describe it in **Sec. 5.3**.

5.2 Iterative Adversarial Training

Modern deep RL techniques make it straightforward to train a single agent to act optimally in a given environment. However, they do not directly extend to multi-agent environments. In our case, we need to train both the cop and the robber to make intelligent decisions during their adversarial game against each other. We do have the option of building a good opponent using other algorithms such as Alpha-Beta pruning or UCT search and playing against that, but it must be noted that even a 100% success rate against such opponents does not offer much evidence of how good our agent is against a truly good opponent. In addition, our agent's strategy in these cases would be limited by what situations it has encountered during its training against these algorithmic approaches. There is a need to have an adaptive opponent that can keep forcing the learning agent into new situations so that it can learn to counter them. We use a methodology that we call Iterative Adversarial Training to train both the cop and the robber by playing them against each other.

5.2.1 Pitfalls

While it might appear straightforward to train all the agents involved in a multi-agent game by playing them against each other in successive iterations, a poorly designed approach might result in one of the following negative scenarios:

- The cop could severely outperform the robber and make it improbable for the robber to explore trajectories where it manages to escape from the cop. The cop needn't achieve true competence to force such a situation as it is possible that an otherwise poor strategy is strong against an untrained robber.

- The robber could severely outperform the cop and make it improbable for the cop to explore trajectories that result in the robber's capture. Again, the robber needn't be significantly competent to cause this result.

- Both agents could fail to achieve any useful learning. In this case, the robber would have a much higher success rate just due to luck.

Our objective however is that both agents achieve useful learning and converge to a policy that allows them to win on most graphs where they can win with an optimal policy.

5.2.2 Algorithm

In the most basic version of Iterative Adversarial Training, we instantiate both agents to arbitrary, random policies. In each iteration,

1. Freeze the robber's policy to its current policy.

2. Using Proximal Policy Optimization, train the cop to play against this robber for 1000 episodes.

3. Freeze the cop's policy to its current policy.

4. Train the robber against this cop for 1000 episodes using Proximal Policy Optimization.

In this initial version of the Iterative Adversarial Training algorithm, we noticed that we were indeed suffering from some of the pitfalls described in **Sec. 5.2.1**. Our final version of the algorithm contains numerous tweaks that we implemented to alleviate these issues which we further describe in **Sec. 5.2.3**.

We begin by instantiating a robber agent `Robber_Agent` and a cop agent `Cop_Agent` and initialize these to follow arbitrary, random policies. We maintain two agent buffers implemented as queues: `Buffer_Robber` and `Buffer_Cop` of length L_{buffer}. These buffers store several recent versions of the respective agents in a First In, First Out (FIFO) fashion. The algorithm then proceeds as a series of *turns*, each of which consists of a number of *steps*. Each *turn* proceeds as follows:

1. A cops and robbers environment is instantiated where the robber is a randomly chosen version from `Buffer_Robber`. The cop is the learning agent in this environment.

2. We undertake `N_cop_steps` *steps*. In each step:

   (a) We perform several (200) rollouts of this environment under the chosen cop and robber policy in the present environment.

   (b) We use the trajectories gained from the rollouts to repeatedly update the cop's neural network weights using proximal policy optimization.

   (c) We randomly sample a robber version from `Buffer_Robber` and set this as the robber in our environment.

3. We push this version of the cop into the tail of `Buffer_Cop`, thereby causing the oldest version of the cop in the buffer to be popped out.

4. We evaluate the most recent version of the cop against all the robbers in the buffer by performing `N_eval_rollouts` rollouts for each version of the robber. We do not allow exploration and make the agents choose the action that is predicted by the respective networks to have the highest advantage $A(s, a)$.

5. We use the measured performance to set `N_cop_steps = Step-Count(performance)`. This function sets a higher step count for lower performance, thus giving the poorly performing agent more trajectories to learn from.

6. A cops and robbers environment is instantiated where the cop is a randomly chosen version from `Buffer_Cop`. The robber is the learning agent in this environment.

7. We undertake `N_robber_steps` *steps*. In each step:

   (a) We perform several (200) rollouts of this environment under the chosen cop and robber policy in the present environment.

(b) We use the trajectories gained from the rollouts to repeatedly update the robber's neural network weights using proximal policy optimization.

(c) We randomly sample a cop version from `Buffer_Cop` and set this as the cop in our environment.

8. We push this version of the robber into the tail of `Buffer_Robber`, thereby causing the oldest version of the robber in the buffer to be popped out.

9. We evaluate the most recent version of the robber against all the cops in the buffer by performing `N_eval_rollouts` rollouts for each version of the cop. Again, we do not allow exploration and simply choose the action that is predicted by the agents to have the highest advantage $A(s, a)$.

10. We use the measured performance to set `N_robber_steps = Step-Count(performance)`.

All the steps described above constitute a single turn. The IAT algorithm requires that these turns are run repeatedly until convergence.

Equivalent high-level pseudocode for the above algorithm is given in **Alg. 1**. Further elucidation about the helper methods used for the top-level procedure is given in **Alg. 2**.

5.2.3 Rationale

This section explains why we transformed the Iterative Adversarial Training algorithm from its simple beginnings to the vastly more complicated final version.

5.2.3.1 Agent Buffers

It is often noted while training neural networks that when there is significant temporal deviation in the nature of the input batches during training, the network loses its ability to perform well on samples that it was trained on long ago and fits its weights to just perform well on recent input samples. This effect, called *catastrophic forgetting*, is not as much of a problem in traditional deep learning as we can just randomly shuffle the input samples to eliminate the temporal deviations. However, input samples in reinforcement learning are highly correlated and it is not trivial to

Algorithm 1 Top-level procedure of the iterative adversarial training algorithm

procedure ITERATIVE-ADVERSARIAL-TRAINING ▷ This procedure trains a cop agent and a robber agent for the cops and robbers game by playing them against each other.

```
Cop_Agent ← Initialize with arbitrary neural network weights.
Robber_Agent ← Initialize with arbitrary neural network weights.
Buffer_Robber ← FIFO Queue of length L_buffer.
Buffer_Cop ← FIFO Queue of length L_buffer.
N_cop_steps ← Step-Count(1.0)
N_robber_steps ← Step-Count(1.0)

for i in 0 to L_buffer do
   Buffer_Cop.push(Cop_Agent)
   Buffer_Robber.push(Robber_Agent)
end for

for i in 0 to N_turns do
   for j in 0 to N_cop_steps do
      random_opponent ← Random-Element(Buffer_Robber)
      Cop-Learner-Env ← new Cop-Env(random_opponent)
      trajectories ← Perform-Rollouts(Cop-Learner-Env,
Cop_Agent, False)
      Cop_Agent.PPOUpdate(trajectories)
   end for

   Buffer_Cop.push(Cop_Agent)
   mean_perf ← Eval-Agent-Perf(Cop_Agent, Buffer_Robber,
Cop-Env)
   N_cop_steps ← Step-Count(mean_perf)

   for j in 0 to N_cop_steps do
      random_opponent ← Random-Element(Buffer_Cop)
      Robber-Learner-Env ← new Robber-Env(random_opponent)
      trajectories ← Perform-Rollouts(Robber-Learner-Env,
Robber_Agent, False)
      Robber_Agent.PPOUpdate(trajectories)
   end for

   Buffer_Robber.push(Robber_Agent)
   mean_perf ← Eval-Agent-Perf(Robber_Agent, Buffer_Cop,
Robber-Env)
   N_robber_steps ← Step-Count(mean_perf)

end for
```

end procedure

Algorithm 2 Utility methods for the iterative adversarial training algorithm

procedure EVAL-AGENT-PERF(Agent, Opponents, Env-Type) ▷ Procedure that evaluates a given agent against a set of opponents.

```
mean_perf ← 0
for j in 0 to N_eval_steps do
    random_opponent ← Random-Element(Opponents)
    Eval-Env ← new Env-Type(random_opponent)
    trajectories ← Perform-Rollouts(Eval-Env, Agent, True)
    mean_perf ← mean_perf + Sum(trajectories.rewards)
end for
mean_perf ← Step-Count(mean_perf/N_eval_steps)
return mean_perf
```

end procedure

procedure PERFORM-ROLLOUTS(Env, Agent, shouldExplore) ▷ Procedure to perform a fixed number of rollouts in an environment using an agent. Returns sample paths and rewards.

```
trajectories ← Empty set that can hold trajectory objects.
for j in 0 to N_rollouts do
    trajectory ← Empty trajectory
    state ← Env.reset()
    done ← False
    while not done do
        action_probs ← Agent.compute_action_probs(state)
        if shouldExplore then
            action ← Random-Sample(action_probs)
        else
            action ← Argmax(action_probs)
        end if
        next_state, reward, done ← Env.step(action)
        trajectory.append_info(state, action, next_state,
reward, done)
    end while
    trajectories.append(trajectory)
end for
return trajectories
```

end procedure

eliminate this dependency. The commonly used technique is experience replay where a long replay buffer of state transitions, actions and rewards is maintained. The neural network is trained on samples randomly chosen from this buffer instead of on the most recent samples.

The same problem surfaces in our case, albeit at a higher level: Catastrophic forgetting causes the agent to forget how to play well against older versions of the opponent agent, and instead overfit itself to exploit the peculiar weaknesses of its current opponent. We borrow a page from experience replay and create *agent buffers* that hold several opponents all of which an agent can learn from. Over time, we hope that the members of the buffer become dominantly superior to those members that are popped out, thus invalidating the need for what we expect to be obsolete agents.

5.2.3.2 Adaptive turn lengths

We observed that depending upon the input graph space that we used, there were numerous training runs where either the cop or the robber ended up dominating the training process and blocked the other agent's ability to learn by severely curbing meaningful exploration. Imbalance in the success rates of the cop and robber is acceptable and could be a natural result of the biased nature of the sample space. While imbalance can still produce competent agents, extreme skews in the success ratio in favor of one agent will result in an extremely small number of positive examples for the opponent.

Our solution to this issue was to adaptively vary the number of steps each turn consisted of for the agents. Thus, the agent that loses more games would get more training bandwidth to refine itself and compensate for its poor performance.

Specifically, if α is the success rate of an agent based on the most recent evaluation, we set the number of steps per turn for that agent as

$$\texttt{Step-Count}(\alpha) = \frac{C}{\max(0.5, \alpha)} \tag{5.7}$$

Effectively, the step count for an agent varies from C to $2C$ depending on how poor the agent's performance was during the most recent evaluation.

5.3 Vertex Pooling

It was noted in **Sec. 5.1.4.1** that a traditional GCN needs to have as many layers as the diameter of the graphs in the operating space to solve even a simple path planning task. On the cops and robbers problem, we observed poor performance for both agent classes on graphs with a diameter larger than the depth of the GCN they used. As mentioned earlier, it is not scalable to continually adding GCN layers to increase the operational range. This is since the first dense layer that takes the outputs of all these additional GCN layers would have an extremely large number of weights and thus defeat the purpose of using GCNs to abstract network information into a much smaller number of outputs.

This issue has parallels with the case of Convolutional Neural Networks: the input and output sizes for a CNN layer are nearly the same (unless we use $stride > 1$, which is not applicable to graphs) and there is no spatial aggregation of local information. The solution in case of CNNs was to introduce a Max-Pooling layer that aggregates information from neighboring pixels in the outputs of the CNN layers, thus enlarging the receptive field of each neuron. In the same vein, previous research has shown the efficiency of Graph Pooling layers that compress features from a large number of graph vertices into the outputs of a much smaller number of neurons. While a hierarchical vertex pooling method proposed in [12] forms the base of our vertex pooling method, our stringent computational requirements necessitate that we introduce a few optimizations before we can make use of this approach.

Starting from a graph and a set of associated feature vectors $F(v)$ of length f for each vertex v in the graph, our vertex pooling algorithm produces a series of hierarchical abstractions of the graph, each of which has close to half as many vertices as the lower level of abstraction. Given a graph $G(V, E)$ that has a particular level of abstraction, the next level is represented as a component graph $G'(V', E')$ each of whose vertices contains at least two vertices from the lower level. The abstraction process that we use in this thesis proceeds as:

1. Set all vertices in V to initially be *unmarked*.

2. Select an edge (u, v) that is incident on a pair of vertices u and v both of which are unmarked, such that the geometric mean degree $g = \sqrt{d_u d_v}$ is minimized.

3. Create a new component $v' \in V'$ and assign both u and v to that component.

4. Mark the unmarked vertices.

5. Repeat steps 2, 3 and 4 until at least one of the following stopping conditions is reached:

   (a) All vertices in V have been marked.

   (b) The sum of the number of components $|V'|$ and the number of unmarked vertices (henceforth referred to as the *gross component count*) is less than or equal to a threshold T.

6. If the gross component count is still greater than a threshold T after the loop in the above step terminates, more vertices from V have to be merged into the components that we have created. This is since the vertex pooling process must produce a resultant graph where the number of vertices is at most T.

7. Select an edge (u, v) that is incident on a pair of vertices u and v exactly one of which is unmarked, such that the geometric mean degree $g = \sqrt{d_u d_v}$ is minimized.

8. Assign the unmarked vertex to the component to which the marked vertex has been assigned.

9. Mark the unmarked vertex.

10. Repeat steps 7 and 8 until at least one of the following stopping conditions is reached:

    (a) All vertices in V have been marked.

    (b) The gross component count is less than or equal to the threshold T.

At the end of this process, the gross component count will either be T or $T-1$. We take the following steps to build the graph G':

1. Add any unmarked vertices that remain in G to V'.
2. For every edge $(u, v) \in E$ s.t. $u \neq v$, given that $M(u)$ and $M(v)$ are the components in which they are present, if $(M(u), M(v)) \notin E'$, add such an edge.

In this project, T was set to $\frac{|V|}{2}$ so that the next level of abstraction always has close to one-half the number of vertices from the earlier level. Thus, the above process will result in a graph $G'(V', E')$ and a many-one function $M : V \to V'$ with the following properties:

1. The number of vertices is equal to $\lfloor \frac{|V|}{2} \rfloor$.
2. An edge $(u', v') \in E'$ exists if and only if there exist vertices u and v such that $(u, v) \in E$ and $M(u) = u'$ and $M(v) = v'$.

We then do the pooling operation by aggregating together features corresponding to the vertices attached to each component. Thus, in the case that the input graph and features are of dimensions $|V| \times |V|$ and $|V| \times f$ respectively, the vertex pooling layer outputs a matrix of dimensions $|V'| \times f$ where V' is the number of vertices in the abstract graph. While the choice of the aggregation function to be used depends on the application, we observed that the maximum pooling function achieves the best performance.

The outputs of the complete vertex pooling process are:

1. The pooled features F' that correspond to each of the vertices in G'.
2. The adjacency matrix of the abstract graph G'.
3. A matrix $\mathbf{M}$ of dimensions $|V'| \times |V|$ that describes the mapping from vertices in G to vertices in G'. $[\mathbf{M}]_{ij} = \mathbb{1}(i = M(j))$.

Algorithm 3 Top-level procedure of the vertex pooling approach

procedure VERTEX-POOLING(G, F, T) ▷ This procedure takes a graph G, associated vertex features F and returns vertex-pooled features for an abstract graph built from G.

```
for v in G.V do
    v.isMarked ← False
end for
numUnmarked ← 0
G' ← New graph initialized to have no vertices or edges.

while not Stopping-Condition(numUnmarked, G', T) do
    edge ← Get-Edge-With-minGMDegree-and-Markedness(G, 0)
    addedComponent ← G'.V.add()
    Assign-Components-And-Mark(edge, addedComponent)
    numUnmarked ← numUnmarked - 2
end while
while not Stopping-Condition(numUnmarked, G', T) do
    edge ← Get-Edge-With-minGMDegree-and-Markedness(G, 1)
    c ← edge.u.isMarked ?  edge.u.component :  edge.v.component
    Assign-Components-And-Mark(edge, c)
    numUnmarked ← numUnmarked - 1
end while

for u in G.V do
    if u.unmarked then
        addedComponent ← G'.V.add()
        u.component ← addedComponent
    end if
end for
for e in G.E do
    if (e.u.component, e.v.component) not in G'.E then
        G.E.add((e.u.component, e.v.component))
    end if
end for
M ← Get-Mapping-Matrix(G, G')
return Get-Pooled-Features(G, G', F), G', M
end procedure
```

Algorithm 4 Helper methods for the vertex pooling approach

```
procedure ASSIGN-COMPONENTS-AND-MARK(edge, component)  ▷ Procedure to mark and assign a component to vertices on the endpoints of an edge.
    edge.u.component ← component
    edge.v.component ← component
    edge.u.isMarked ← True
    edge.v.isMarked ← True
end procedure

procedure STOPPING-CONDITION(numUnmarked, G', T) ▷ Procedure that returns whether or not the stopping condition has been reached
    if numUnmarked == 0 then
        return True
    end if
    if numUnmarked + len(G'.v) <= T then
        return True
    end if
    return False
end procedure
```

Equivalent high-level pseudocode for the above algorithm is given in **Alg. 3**. Further elucidation about methods used for the top-level procedure is given in **Alg. 4** and **Alg. 5**.

The vertex pooling approach proposed here have similar foundations to the abstract versions of the base graph proposed in [6] and subsequently used in [4]. However, the goal in our case is to use the abstract graphs as inputs to a GCN layer and enlarge the receptive field of the GCN. In contrast, [6] and [4] use them to reduce the computational complexity of their heuristic search methods. Implementationally, our method uses a deterministic merge strategy that is applicable to all connected graphs while the abstraction strategy that was proposed in [6] was defined only for grids and octile graphs.

Algorithm 5 Sub-task procedures for the vertex pooling approach

procedure GET-MAPPING-MATRIX(G, G') ▷ Procedure to calculate a matrix that encodes how the vertices of G are mapped to G'

```
M ← Matrix of dimensions len(G'.V) x len(G.V)
for u in G.V do
    M[u][u.component] ← 1
end for
return M
```

end procedure

procedure GET-EDGE-WITH-MINGMDEGREE-AND-MARKEDNESS(G, markedness) ▷ Procedure that returns the edge minimizing the Geometric Mean of endpoint degrees, with a given number of endpoints that have been marked.

```
minGMDegree ← Infinity
minGMDegreeEdge ← NULL
for e in G.E do
    if e.u.isMarked + e.v.isMarked == markedness then
        currentGMDegree ← GM(e.u.degree, e.v.degree)
        if currentGMDegree < minGMDegree then
            minGMDegree ← currentGMDegree
            minGMDegreeEdge ← e
        end if
    end if
end for
return minGMDegreeEdge
```

end procedure

procedure GET-POOLED-FEATURES(G, G', F) ▷ Procedure to pool features with the abstract graph.

```
outFeatures ← Matrix of zeros with shape len(G'.V) x F.shape[1]
featureSet ← Array of Empty sets, length len(G'.V)
for u in G.V do
    featureSet[u.component].append(F[u])
end for
for u in G'.V do
    outFeatures[u] ← Pooling-Aggregation(featureSet[u])
end for
return outFeatures
```

end procedure

6. SOFTWARE IMPLEMENTATION, CHALLENGES AND SOLUTIONS

This chapter details how we implemented our ideas in software, the major challenges that we faced in this phase and our solutions for each of those issues.

6.1 High-Level Elements of the Software System

The entire system has been designed and built to adhere as much as possible to Object Oriented Programming Principles. The different components of the system in decreasing order of generality are:

1. **Orchestrators**: These are top-level modules that are responsible for a complete, end-to-end process. Orchestrators instantiate and initialize all required objects, set configurations and guide the general program flow. In general, an orchestrator is a standalone script that need not be called by a different module to be executed.

2. **Environments**: Classes that represent our MDPs' environment and provide methods to instantiate a given environment, reset this environment, and simulate the actions received from an *agent* in the environment. Some of these methods return the consequences of agent actions, e.g. the next state reached and the reward.

3. **Agents**: These entities are capable of storing and learning a policy by interacting with an *environment*. The training methods required to train these agents are self-contained. Agents internally use *models* to encode their current policy.

4. **Models**: A model in our case is a class that can encode a policy that an agent can use to sample actions. In general, a model consists of methods that can take a state and return an action to be taken from that state. We do not restrict the models to be internally memoryless, nor force them to use function approximation.

5. **Layers**: Layers are components that are relevant to *models* that use neural networks for

function approximation. They store the forward propagation and backpropagation logic for one layer of a neural network.

6. **Graph Generators**: These entities are used by *environments* to build the world in which the *agents* compete against each other. They provide various methods to sample random graphs that have specific properties.

7. **Utilities**: Generic methods that abstract out code logic to increase readability in all modules of the system.

The system was completely implemented in Python v3.7.

6.2 Sub-system Description and Implementation

This section describes in detail the various sub-systems that were listed in **Sec. 6.1**.

6.2.1 Orchestrators

As mentioned before, an orchestrator is a standalone program that completes a specific process on its own. This could be as simple as processing outputs and creating a plot, or as complicated as evaluating an entire history of agents against a fixed opponent. Since the orchestrators are top-level programs that do not encode complicated low-level logic, very few external libraries were directly used in their implementation. As such, much of the orchestrator code is native Python code, procedure calls and straightforward application of tools from `numpy` and `matplotlib`.

6.2.1.1 agent_trainer

This orchestrator is used to train a given agent in a given environment in a sequential training process using a training algorithm such as PPO. Specifically, `agent_trainer`:

1. Instantiates a particular agent.

2. Instantiates a particular environment.

3. Uses an RL algorithm to train the agent in this environment.

4. Writes periodically sampled performance metrics to file.

6.2.1.2 iterative_adversarial_trainer

This orchestrator competitively trains a pair of agents to operate well in an environment by playing them against different versions of their opponents. Most of **Alg. 1** is implemented in this orchestrator. Specifically, `iterative_adversarial_trainer`:

1. Instantiates a pair of agents.
2. Instantiates respective learner environments.
3. Creates agent buffers and controls the entry and exit of agents into the buffers.
4. Iteratively trains these agents by playing them against each other while closely adjusting the step count.
5. Writes periodically sampled performance metrics to file.

6.2.1.3 solo_agent_rollout

This orchestrator focuses on visualization and evaluation of the finally obtained agents *after* all training is done. Its main role is to support debugging and analysis of a single trained agent in a post-facto situation. Specifically, `solo_agent_rollout`:

1. Instantiates a particular agent.
2. Restores it to weights learned earlier.
3. Instantiates an environment for this agent to operate in.
4. Performs rollouts that make this agents act in the environment.
5. Evaluates the results of the rollouts in terms of number of steps and episodes, mean rewards and success rates.
6. Renders the rollouts graphically.

6.2.1.4 performance_plotter

This orchestrator focuses on visualizing the progress of the agents *during* training. Specifically, `performance_plotter`.

1. Loads metrics gathered during training runs.
2. Processes and formats them.
3. Produces plots of the results.

6.2.2 Environments

An *environment* entity encodes all logic pertinent to the environment in an MDP. In our case, we want an environment to:

- Initialize required graph generators and opponent agents according to configurable parameters.
- Sample random graphs using an internal *graph generator* object.
- Provide a method to reset the environment to a starting position. This requires sampling a new graph and positioning the agents at a configurable distance of each other.
- Provide a method that allows the learning agent to take actions within the environment.
- Continually check for whether the game should be terminated under the current agent positions.
- Return the reward obtained by the agent after every action.
- Track the MDP from one state to another.
- Return logging information about the actions of the agents, the states, rewards, etc.
- Provide a method to render the state of the environment along with pertinent additional details.

- Contain attributes that describe the size of the state space and action space.

This project implements environment classes that enforces these properties by inheriting them from OpenAI's `gym.env` class. This abstract class throws a `NotImplementedError` if required methods such as `step`, `reset` and `render` have not been implemented, thus ensuring that our environment implementation has the minimum functionality required to emulate a Markov Decision Process.

The environments that this project uses are described further below.

6.2.2.1 MultiAgentIterativeAdversarialEnv

This environment is primarily what we used to train the cop and robber agents by playing them against each other. The environment's constructor takes in parameters that detail the following contextual information:

1. The learning agent for this instance of the environment. If the learning agent is a cop, a robber agent is automatically instantiated to be the learner's opponent. In case the learning agent is a robber, a cop opponent is automatically instantiated within the constructor.

2. Whether this environment is a "training" environment or an "evaluation" environment. In case that this is a training environment, the Watts-Strogatz graph generation algorithm is used. If it is an evaluation environment, then a cop-win graph generator or a robber-win graph generator is instantiated for cop learners and robber learners respectively. In addition, the environment returns dense rewards during the learning phase and sparse rewards (since we only care about success rates) during the evaluation phase.

A shortcoming of our implementation is that each instance of the environment (created during parallelized rollouts) must contain its own opponent instance, thereby increasing the amount of memory that each instance requires. We expended significant effort in attempting to overcome this issue by using a static variable for the opponent that is shared among all environment instances, but `Ray RLLib`'s agent instantiation methodology was incompatible with this approach. It is also

possible that even had we been successful, this would have caused a concurrency bottleneck since all the environment instances will request actions from the same opponent instance.

It must be noted that this limitation only reduces the parallelization capacity of our system while performing rollouts and not the performance of trained agents. Considering that `Ray RLLib` makes deep copies of all internal variables of an environment instance to perform parallel rollouts, this problem might just be an issue that will persist as long as we rely on Ray.

6.2.2.2 SingleAgentFixedStrategyEnv

This environment implements a simpler problem where the opponent of the learner agent uses a fixed, non-learning strategy such as a random policy or a greedy policy. The opponent agent is forced to be stateless here, thus allowing us to instantiate a large number of instances of this environment and perform several rollouts in parallel.

6.2.3 Agents

An *agent* is an entity that holds a policy and is capable of training the policy by performing rollouts on a pre-defined environment. As such, it contains both a model that encodes a policy through function approximation, and an RL algorithm such as PPO that can train this model. Our agent classes are of simple design and form a wrapper around the PPO class that `Ray RLLib` provides. The agent class has the following functionalities:

- Take an environment class along with required environment configuration and instantiate an agent that can act on the environment.
- Provide a method to perform rollouts within the environment and use the obtained trajectories to train the model using PPO.
- Compute the action that must be taken for a given state according to the learned policy.

`Ray RLLib`'s PPO class efficiently trains the agent by

1. Instantiating several “workers", each of which contains a copy of the environment and the model.

2. Making each worker perform rollouts in parallel by allocating the total available CPU time amongst all workers. Naturally, the total memory required by the workers must be less than the System RAM. If GPU training is enabled, the total GPU memory required by the workers must also be less than the available Graphics Memory.

3. Collecting the trajectories obtained by the workers and calculating the neural network targets.

4. Performing the neural network update using an algorithm such as Adam or RMSProp.

Evaluation is always performed with a single worker, thus requiring us to constrain the evaluation frequency to a small number.

6.2.4 Models

A model is an entity which contains a function approximator that can encode an agent policy. A model must:

- Instantiate itself to be compatible with a particular action space and state space. In our case, the action space and state space decide the number of output and input neurons respectively.

- Provide a method that can forward propagate a given state input and get action advantages for that state.

- Return the value function of the model as a callable method.

We enforce these requirements by inheriting our model classes from Ray's `TFModelV2` class. This mandates that the model implementation be capable of taking in inputs that follow Tensorflow's input specifications and provides outputs that have the exact type as a Tensorflow `Model` object.

In line with these restrictions placed on the model, we create its neural network object using Tensorflow's Functional API. We used the `Input` and `Dense` *layers* from the Tensorflow library along with our implementations of the GCN layer and the vertex pooling layer to build this neural

network. **Sec. 4.5** describes how many input descriptors of the state are represented as one-hot encoded vectors. To save memory, we store just the on-bit in the vector as an integer for as long as possible and convert that into a one-hot encoded vector at the very last moment, within the neural network. We use Tensorflow's `one_hot` layer to accomplish this.

Models, once declared need to be registered with Ray before they can be used within agents. We do this in the orchestrator since that is where agents will be initialized.

6.2.5 Layers

A *layer* is an entity that handles both the forward propagation and backpropagation logic for a particular type of neural network layer such as fully-connected, convolutional or LSTM. Any layer class must be capable of

- Instantiating a layer object given a set of input and output dimensions along with any required configurations.
- Storing its own weights and biases within and providing these when requested. This is necessary to support saving and loading weights, making copies of neural networks, etc.
- Performing forward propagation given a set of inputs and return the output of that layer.
- Perform backpropagation of gradients from the next layer to the previous layer.

We achieve some of these objectives by inheriting our layer classes from Tensorflow's `layers.Layer` abstract class. This class puts implementation requirements on our child classes, as well as provides additional functionality common to all layers such as providing a backpropagation method based on the forward propagation logic using automatic differentiation.

The layers that we have implemented in this project are described further below.

6.2.5.1 GCN Layer

The GCN layer takes in an Adjacency Matrix $\mathbf{A}$ and a matrix of input vertex features $\mathbf{F}$ from all vertices in a graph as input, and calculates the output as

$$\mathbf{o_W}(\mathbf{A}, \mathbf{F}) = ReLU((\mathbf{A} + \mathbf{I})\mathbf{F}\mathbf{W}) \tag{6.1}$$

Here, $\mathbf{W}$ is the weight matrix for this GCN layer. Given that the graph has V vertices and the input features for each vertex are of length f, the dimensions of the various tensors involved are:

- $\mathbf{A}$: Tensor of dimensions Batch size $\times\ V \times V$
- $\mathbf{F}$: Tensor of dimensions Batch size $\times\ V \times f$
- $\mathbf{W}$: Tensor of dimensions Output Feature Length $\times\ f$

Note that the feature matrix $\mathbf{F}$ need not be the original vertex features provided to the neural network as input - they can be transformed features from earlier GCN layers as well. In addition, the adjacency matrix does not have to correspond to that of the original graph. Vertex pooling results in abstract graphs, and the adjacency matrices corresponding to these graphs can be fed into the GCN layers as well in place of the adjacency matrix fed to the network as an input.

6.2.5.2 Vertex Pooling Layer

The algorithmic workings of the vertex pooling layer are described in detail in **Sec. 5.3**. The vertex pooling process creates an abstract version of the input graph by deterministically merging together neighboring vertices while preserving their connectivity information. The features of the vertices in the abstract graph are obtained by aggregating the features of vertices from the original graph.

The result of the vertex pooling process is twofold:

1. The graph structure of the abstract graph: This is represented using the adjacency matrix of the graph that was produced at the end of the abstraction.

2. The pooled features: A new set of features is assigned to each vertex in the abstract graph, and has a feature length equal to the length of the input features.

As in the case of the GCN layer, the Vertex Pooling layer inherits from Tensorflow's `layers.Layer` class. We take advantage of the fact that the graph abstraction process itself is deterministic and depends only on the graph. We externally build the abstract graph in the neural network's preprocessor and feed the vertex mapping to a layer that simply aggregates the vertex features according to the mapping. While this approach reduces code modularity, it greatly simplifies the implementation as we will have no incompatibility with Tensorflow's eager execution mechanism.

As described in **Sec. 5.3**, the mapping from vertices in the input graph to vertices in the abstract graph is represented with a matrix $\mathbf{M}$ that has a 1 at i, j if and only if vertex j in the original graph corresponds to component vertex i in the abstract graph.

We undertake the following steps in the preprocessor:

1. Take as input the adjacency matrix of the graph.
2. Use the vertex pooling algorithm described in **Alg. 3** to produce an abstract graph that contains a reduced number of vertices compared to the input graph.
3. Build the adjacency matrix $\mathbf{A}'$ for this abstract graph.
4. Build the mapping matrix $\mathbf{M}$.
5. Return $\mathbf{A}'$ and $\mathbf{M}$.

Within the neural network model, both $\mathbf{A}'$ and $\mathbf{M}$ are available. The vertex pooling layer's calculation can be expressed as

$$\mathbf{F}' = PoolingAggregation(\mathbf{M}, \mathbf{F}) \tag{6.2}$$

In the simple case where the pooling aggregation is done through additive pooling, the above equation reduces to a matrix multiplication.

$$\mathbf{F}' = \mathbf{M} \times \mathbf{F} \tag{6.3}$$

When the vertex pooling layer is used as part of a neural network, the $\mathbf{A}'$ is usually fed as the adjacency matrix to the next GCN layer along with $\mathbf{F}'$ as the vertex features.

6.2.6 Graph Generators

A *graph generator*'s responsibility is to sample graphs using a fixed random graph generation technique such as the Watts-Strogatz Algorithm. Clients may request specific constraints to be placed on the generated graph, such as restricting the maximum vertex degree or the diameter of the graph. In such cases the graph generator must keep sampling new graphs until it gets a graph that qualifies all such constraints.

A graph generator performs the following steps:

1. Instantiates itself with a set of configurable parameters that control the nature of the graphs generated.
2. Periodically generates a new graph every time one is requested.
3. Validates the generated graph and re-samples the graph if it fails any of the predefined constraints.

In this project, an abstract class called `GraphGenerator` implements the logic to check an undefined set of constraints and regenerate graphs until the constraints are met. All graph generators inherit from this parent class and define the graph validation constraints to integrate with this mechanism.

The graph generators that were used in this project are:

1. Small-world graph generator: This uses the Watts-Strogatz algorithm to generate connected graphs that have a relatively small diameter despite having a large number of vertices.

2. Grid graph generator: This graph generator creates a two-dimensional lattice and is deterministic. As expected, this graph generator cannot have constraints.

3. Tree generator: This generates random connected graphs where only one path exists between any two vertices.

All graph generators use the `networkx` library to sample graphs.

6.3 System Implementation Outcomes

Some positive outcomes arising from the programming practices that we followed during this project are:

- Fundamental entities such as graph generators and models have abstract classes defined, and all realizations of these entities inherit from these classes and override them. This helps enforce conventions as well as encourages code reuse.
- Configurable `Logger` classes allow uniform, systematic logging and metrics aggregation.
- Strong integration of existing libraries and open-source tools reduces the number of cases where we expend effort reinventing the wheel.
- Use of modern Python (>= v3.7) techniques such as checking argument type and return type statically reduces runtime errors and increases code reliability.

7. COP AND ROBBER EVALUATION METHODS, COMPARISONS WITH TRADITIONAL ALGORITHMS AND RESULTS

This chapter details the methodology that we use to evaluate the performance of the trained cop and robber agents. It then moves on to describe the alternative approaches that we compare our agents against. The chapter concludes with a presentation and analysis of the evaluation and comparison results thus obtained, illustrating the findings with detailed plots.

7.1 Evaluating Trained Cops and Robbers

7.1.1 Requirements and Restrictions on an Evaluation Methodology

The task of evaluating a trained cop or a robber has a number of requirements that should be met:

1. Evaluation should be against an opponent that is optimal or near-optimal.
2. Evaluation should be statistical and should result in a numerical measure of performance, not just a qualitative one.
3. The performance metric should be set in a range where the lowest value implies complete failure and the highest value implies complete success. Naturally, intermediate values should imply varying degrees of success on the problem.
4. The performance measurement process should be reproducible, given the same initial conditions.

These restrictions mean that firstly, using opponent agents trained using the approach suggested in this thesis, or against agents that use the heuristic approaches defined in prior work is ruled out. This is since **Condition 1** requires that we cannot evaluate a trained cop or robber by just letting it competing against an opponent that uses an algorithm that is not guaranteed to be optimal.

Secondly, **Condition 2** necessitates that for reliable performance measurement, the evaluation process must be carried out over a large number of graphs and not just a few hand-crafted graph worlds that are thought to be interesting or balanced.

Thirdly, evaluating the trained cop or robber on randomly sampled graphs without any distinction is ruled out. On randomly sampled graphs, without knowing the distribution of cop-win and robber-win graphs, there is no knowledge of the maximum attainable cop or robber performance when playing against an optimal agent. This prevents us from comparing the obtained performance against an ideal upper limit. Thus, **Condition 3** will not be met on randomly sampled graphs that do not have further qualifiers (cop-win and robber-win) attached to them.

Finally, to qualify **Condition 4**, the set of graphs and starting positions used to evaluate the trained agents must be parametrizable (e.g. using a random seed) to make the results reproducible.

7.1.2 The Clairvoyant Negamax Algorithm

We propose the Clairvoyant Negamax Algorithm, which is an evaluation methodology that measures the performance of a trained agent against an adversary that has full knowledge of its strategy. It arises from a simple tweak that we make to the standard Negamax Algorithm: The strategy of the agent being evaluated is used to fix the opponent actions in a Negamax algorithm run (instead of searching over all possible opponent actions), thus allowing the Negamax opponent to take the action that best exploits the weaknesses of the agent being evaluated. If an agent achieves high performance against a Clairvoyant Negamax opponent, it means that the agent's moves, even when known to an optimally playing opponent, do not lead to its defeat.

To illustrate the Clairvoyant Negamax algorithm, consider a case where the cop is the agent to be evaluated. Thus, on a cop-win graph, the trained cop will play against a robber that uses a strategy derived from the Clairvoyant Negamax algorithm. Say that the cop gets a reward of 1 if it catches the robber before the end of the game, and -1 otherwise, and say that the robber receives a reward of 1 if it remains uncaught, and -1 otherwise. Thus, the goals of both the cop and the robber are to get a reward of 1 before the game ends. The algorithm proceeds in a depth-first fashion to explore the search tree, with each level in the tree alternating between the

cop's turn and the robber's turn. The Clairvoyant Negamax algorithm does not use a heuristic and instead explores for as many levels as the maximum number of plies allowed in the game. This is primarily enabled by eliminating the computation required to explore over all possible actions of the agent being evaluated, and instead just using the action it will actually take.

The root node of the search tree corresponds to a state where the robber is about to make its move. At each level,

1. If the cop has already caught the robber, return -1.

2. If the robber is still uncaught and it is the robber's turn at this level in the tree, recursively calculate the node values for all children of the current node. From the results of all recursive calls made here, return the best (maximum) result as the node's value.

3. If the robber is still uncaught and it is the cop's turn at this level of the search tree, get the cop's action from its strategy and explore only that branch. The current's node's value is simply set to the value of the child node corresponding to the action that the cop is known to take from this state.

When the search is complete and the values of all child nodes of the root node are known, the node with the highest value is chosen. Ties are broken arbitrarily, with the action with the lowest index in the current arrangement being chosen over other actions.

Thus, if the trained cop's strategy is represented by $\sigma_{\mathbf{c}}$ and the robber's optimal actions when it has full knowledge of $\sigma_{\mathbf{c}}$ is $\sigma_{\mathbf{r}}$,

$$\sigma_{\mathbf{r}} = \arg\max_{\sigma} \pi_r(\sigma_{\mathbf{c}}, \sigma) \tag{7.1}$$

Here, $\pi_r(\sigma_{\mathbf{1}}, \sigma_{\mathbf{2}})$ is the payoff that the robber receives when the strategies of the cop and robber are $\sigma_{\mathbf{1}}$ and $\sigma_{\mathbf{2}}$ respectively. The cop's payoff v_c in such a case is equal to

$$v_c = \min_{\sigma} \pi_c(\sigma_{\mathbf{c}}, \sigma) \tag{7.2}$$

and is upper bounded by

$$\underline{v_c} = \max_{\sigma_{\mathbf{c}}} \min_{\sigma} \pi_c(\sigma_{\mathbf{c}}, \sigma) \tag{7.3}$$

It may be recalled from **Sec. 2.4.8** that $\underline{v_c}$, the maximin quantity, is the worst reward that all the adversaries of an optimal agent may jointly force the agent to get if they know its strategy. The closer the performance of a cop agent against the clairvoyant negamax robber is to $\underline{v_c}$, the closer its performance is to what an optimal cop strategy can obtain. In fact, an optimally playing cop will always receive a reward of $\underline{v_c}$ against a clairvoyant robber.

Similarly, if the trained robber's strategy is represented by $\sigma_{\mathbf{r}}$ and the cop's optimal actions when it has full knowledge of $\sigma_{\mathbf{r}}$ is $\sigma_{\mathbf{c}}$,

$$\sigma_{\mathbf{c}} = \arg\max_{\sigma} \pi_c(\sigma, \sigma_{\mathbf{r}}) \tag{7.4}$$

Here, $\pi_c(\sigma_{\mathbf{1}}, \sigma_{\mathbf{2}})$ is the payoff that the cop receives when the strategies of the cop and robber are $\sigma_{\mathbf{1}}$ and $\sigma_{\mathbf{2}}$ respectively. The robber's payoff v_r in such a case is equal to

$$v_r = \min_{\sigma} \pi_r(\sigma, \sigma_{\mathbf{r}}) \tag{7.5}$$

and is upper bounded by

$$\underline{v_r} = \max_{\sigma_{\mathbf{r}}} \min_{\sigma} \pi_r(\sigma, \sigma_{\mathbf{r}}) \tag{7.6}$$

As before, the closer the performance of a robber agent against the clairvoyant negamax cop is to $\underline{v_r}$, the closer its performance is to what an optimal robber strategy can obtain. In fact, an optimally playing robber will always receive a reward of $\underline{v_r}$ against a clairvoyant cop.

By the definition of a cop-win graph, on a domain of only cop-win graphs, $\underline{v_c} = 1$. Similarly, on a domain of only robber-win graphs, $\underline{v_r} = 1$ by the definition of a robber-win graph.

7.1.3 Proposed Evaluation Methodology

7.1.3.1 Evaluating Cops

The evaluation methodology proposed and subsequently used in this thesis to evaluate trained cops is as follows:

1. Generate a random cop-win graph. We did this by generating random graphs from a few graph classes that have been proven in prior work to be cop-win. Examples are trees and graphs that do not have irreducible cycles of length more than 3.
2. Randomly choose a starting position with a fixed distance between the cop and the robber.
3. On this cop-win graph:
   (a) Allow the cop agent that is being evaluated to take an action.
   (b) From the resulting state, using Clairvoyant Negamax with a depth equal to the horizon of the game, identify the optimal action for the robber.
   (c) Allow the robber to take this optimal action.
   (d) Repeat steps 3a to 3c until the end of the game.
4. The game is decided in favor of the cop or the robber depending on which agent wins.

This process is repeated over several cop-win graphs and the average win-rate is identified. The result reveals how well the trained cop performs against a robber that is playing optimally.

7.1.3.2 Evaluating Robbers

The evaluation methodology proposed and subsequently used in this thesis to evaluate trained robbers is as follows:

1. Generate a random robber-win graph. We did this by generating random graphs from graph classes that have been proven in prior work to be robber-win. Examples are graphs that have

irreducible cycles of length more than 3 where the robber can reach the cycle before the cop, and grids.

2. Randomly choose a starting position with a fixed distance between the cop and the robber.

3. On this robber-win graph:

   (a) From the current state, using Clairvoyant Negamax with a depth equal to the horizon of the game, identify the optimal action for the cop.

   (b) Allow the cop to take this optimal action.

   (c) Allow the robber agent that is being evaluated to take an action.

   (d) Repeat steps 3a to 3c until the end of the game.

4. The game is decided in favor of the cop or the robber depending on which agent wins.

This process is repeated over several robber-win graphs and the average win-rate is noted. The result reveals how well the trained robber performs against a cop that is playing optimally.

7.2 Comparing with other algorithms

This section details the process that we use to compare against other methods to solve the cops and robbers problem. We first provide the graph on which the game will be played as well as the starting positions to each algorithm and allow them to precompute quantities that they need during the game.

Given two agent training methods M_1 and M_2, there are multiple ways in which they may be compared, each of which has a different meaning:

1. Use M_1 for the cop agent and M_2 for the robber agent on cop-win graphs: This metric depends on both how close an M_1-cop is to optimal performance on cop-win graphs, as well as how well an M_2-robber is able to take advantages of mistakes that the M_1-cop makes. If the M_1-cop does not make any mistakes, then irrespective of M_2's quality, M_1's win-rate will be 100% and M_2's win-rate will be 0%.

2. Use M_1 for the cop agent and M_2 for the robber agent on robber-win graphs: This metric depends on both how close an M_2-robber is to optimal performance on robber-win graphs, as well as how well an M_1-cop is able to take advantages of mistakes that the M_2-robber makes. If the M_2-robber does not make any mistakes, then irrespective of M_1's quality, M_1's win-rate will be 0% and M_2's win-rate will be 100%.

3. Use M_1 for the robber agent and M_2 for the cop agent on cop-win graphs: This metric depends on both how close an M_2-cop is to optimal performance on cop-win graphs, as well as how well an M_1-robber is able to take advantages of mistakes that the M_2-cop makes. If the M_2-cop does not make any mistakes, then irrespective of M_1's quality, M_1's win-rate will be 0% and M_2's win-rate will be 100%.

4. Use M_1 for the robber agent and M_2 for the cop agent on robber-win graphs: This metric depends on both how close an M_1-robber is to optimal performance on robber-win graphs, as well as how well an M_2-cop is able to take advantages of mistakes that the M_1-robber makes. If the M_1-robber does not make any mistakes, then irrespective of M_2's quality, M_1's win-rate will be 100% and M_2's win-rate will be 0%.

This thesis reports metrics measured through all four reporting techniques to get a full picture of each algorithm's performance.

7.2.1 Alpha-Beta Pruning

Alpha-Beta pruning is a technique that follows the same approach as the standard Minimax algorithm, but avoids searching along paths in the tree that will never be picked during optimal play. As in the case of the Minimax and Negamax algorithms, the depth of the search has a major influence on the reliability of the actions chosen. Since Alpha-Beta pruning is more efficient in its search tree exploration, the tree can be explored to a greater depth before the algorithm is forced to rely on the heuristic to terminate the search along a given path.

Alpha-Beta pruning can be illustrated with the following example: consider a case where the cop finds a move a_1 that results in a favorable position as observed by searching until the search

depth. Consider a different move a_2, which if taken, can lead to an even more favorable position with a particular sequence of actions from the cop and the robber. However, the robber agent discovers on further exploring the tree that a_2 also enables the robber to make a move that results in an extremely unfavorable value for the cop. This means that an optimally playing cop will never choose a_2, thus allowing us to avoid exploring different possibilities arising from the cop choosing a_2 any more.

Consider the Minimax formulation of a zero-sum game, where the game's outcome is defined as the cop's payoff. Here, the cop is trying to maximize the outcome of the game while the robber attempts to minimize it. The Alpha-Beta pruning algorithm keeps track of two quantities, α and β throughout the process. α is the worst (least) game score that the maximizing player will at least get. β is the worst (highest) game score that the minimizing player will at most get. It is a recursive algorithm in each step of which:

1. If the recursion has reached the search depth, the heuristic value of the current node is returned.

2. In case of the call being made for the maximizing player, for each child node, i.e. action that the maximizing player can take:

   (a) The worst (minimum) score that the maximizing player can receive after taking that action is calculated.

   (b) α is set to the better (greater) out of its current value and the result of the candidate action.

   (c) If $\alpha \geq \beta$, then the minimizing player would never allow the parent node to be reached, thus allowing us to avoid exploring it further.

3. Over all the minimum scores returned from all the children, the best (thus, a maximin) is returned.

4. In case of the call being made for the minimizing player, for each child node, i.e. action that the minimizing player can take:

   (a) The worst (maximum) score that the minimizing player can receive after taking that action is calculated.

   (b) β is set to the better (smaller) out of its current value and the result of the candidate action.

   (c) If $\alpha \geq \beta$, then the maximizing player would never allow the parent node to be reached, thus allowing us to avoid exploring it further.

5. Over all the maximum scores returned from the children, the best (thus, a minimax) is returned.

A pseudocode version of the above algorithm is given in **Alg. 6**.

7.2.2 Upper Confidence Tree Search

Upper Confidence Tree Search is an algorithm that brings together concepts from Monte Carlo Tree Search and Multi-Armed Bandits. The basic premise of the algorithm is to perform random rollouts of full games, and attribute the win/loss statistics from these rollouts to actions taken during these rollouts. Actions that have a high win-rate are deemed favorable and are recommended over actions with low win-rates. The algorithm borrows a page from the Upper Confidence Bound-1 (UCB-1) algorithm used to solve Multi-Armed Bandits: during the exploration phase, it chooses to take those actions that have a high Upper Confidence Bound instead of uniform randomly choosing one of the available actions. UCT search has a number of differences with Minimax, Negamax and Alpha-Beta pruning:

- The game is rolled out to the very end. Each rollout gives a win/loss datapoint that is used to update the favorability of all actions that were taken in the rollout from various states.
- No heuristic is necessary since the depth is not limited to a quantity less than the game's length.

Algorithm 6 The Alpha-Beta Pruning Algorithm

```
procedure ALPHA-BETA-PRUNE(treeNode, H, isMaximizingPlayer, alpha, beta)  ▷ Procedure that sets explored nodes to the rewards obtainable during optimal play using Alpha-Beta pruning
    if treeNode.leafNode || nodeHeight == 0 then
        return heuristic(treeNode)
    end if
    if isMaximizingPlayer then
        maxVal ← -Infinity
        for c in treeNode.children do
            cVal ← Alpha-Beta-Prune(c, H-1, False, alpha, beta)
            maxVal ← max(maxVal, cVal)
            alpha ← max(maxVal, alpha)
            if beta <= alpha then
                break
            end if
        end for
        return maxVal
    else
        minVal ← Infinity
        for c in treeNode.children do
            cVal ← Alpha-Beta-Prune(c, H-1, True, alpha, beta)
            minVal ← min(minVal, cVal)
            beta ← min(minVal, beta)
            if beta <= alpha then
                break
            end if
        end for
        return minVal
    end if
end procedure
```

- A new parameter, the number of rollouts performed during the estimation phase, has a major influence upon the reliability of the estimates that we get from UCT search.

Each node in the search tree records the following information:

- w, the number of wins achieved by playing this node and then rolling the game out further.
- n, the number of games that have been played from this node and then rolling the game out further.

UCT Search proceeds as a series of rollouts. Each rollout is a descent in a tree towards a terminal state, i.e. a leaf node in the tree. From every node, the agent has the choice of moving to any of its child nodes in the tree, and this constitutes an action.

1. Given that $1, 2, ...|a|$ are the available child nodes from the current node, for the i^{th} child node, calculate $f_i = \frac{w_i}{n_i} + c\sqrt{\frac{\log N_i}{n_i}}$.
2. Take the action a that has the maximum value for f_i, i.e. $a = \arg\max_i f_i$.
3. Repeat steps 1 and 2 until the game ends.
4. Increment by 1 the value of n for the tree nodes corresponding to all the actions the agent took during this rollout.
5. If this game ended in a win, increment by 1 the value of w for the tree nodes corresponding to all the actions the agent took during this rollout.

Here, w_i is the number of wins achieved so far after playing the i^{th} action from the current node, n_i is the number of games played so far after playing the i^{th} action from the current node, N_i is the total number of games played from the current node, and c is the exploration parameter that decides how much we wish to favor exploration over exploitation. Note here that in our problem, the state depends only on the sequence of actions that were taken from the beginning of the game, given the initial state.

7.3 Results

7.3.1 Evaluation Against the Clairvoyant Negamax Algorithm

As mentioned earlier, the Clairvoyant Negamax algorithm is a true yardstick that exploits to the best extent any and all mistakes that an agent makes. In the cops and robbers game, optimal play implies that a cop will win on cop-win graphs whatever the robber strategy is, and a robber will win on robber-win graphs whatever the cop strategy is. Thus, a cop agent's success rate on cop-win graphs against a clairvoyant robber tells us how close to optimal play the cop is. Similarly, a robber agent's success rate on robber-win graphs against a clairvoyant robber tells us how close to optimal play the robber is. That said, there is little utility in measuring a trained cop agent's success rate on robber-win graphs against a clairvoyant Negamax robber. This is since by definition, a clairvoyant Negamax robber will win on a robber-win graph. The same can be said of evaluating trained robber agents on cop-win graphs against a clairvoyant Negamax opponent.

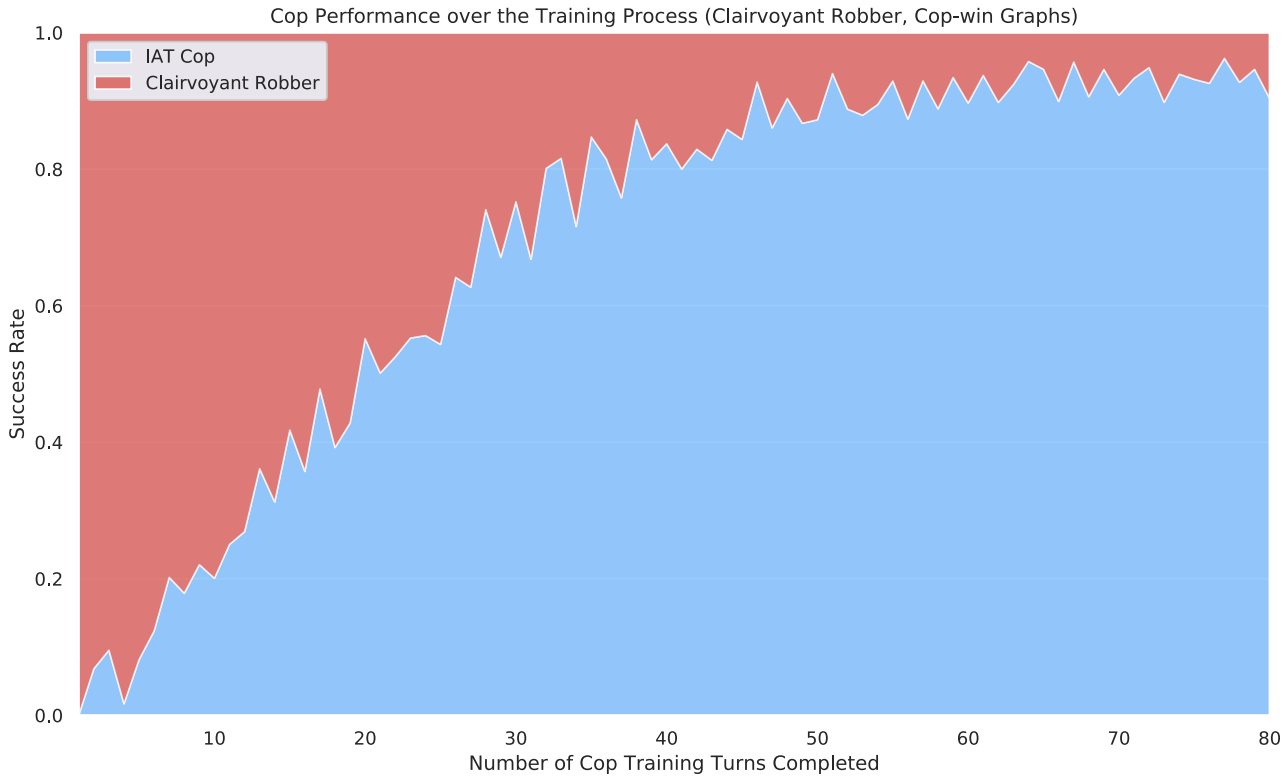

Figure 7.1: Plot depicting how a cop agent trained using the Iterative Adversarial Training algorithm performs against a Clairvoyant Negamax Robber when tested on cop-win graphs

The plot in **Fig. 7.1** was produced by allowing a cop trained using the IAT algorithm to compete against a clairvoyant Negamax robber on cop-win graphs. The initial distance between the cop and the robber was set to a random integral value in $[4, 10]$ during evaluation, and the maximum number of turns was set to 1.5 times the initial distance. While an optimally playing cop would achieve a success rate of 1.0, the success rate in this case increases from an initial value of 0.03 up to a saturation value of 0.93 over the course of training.

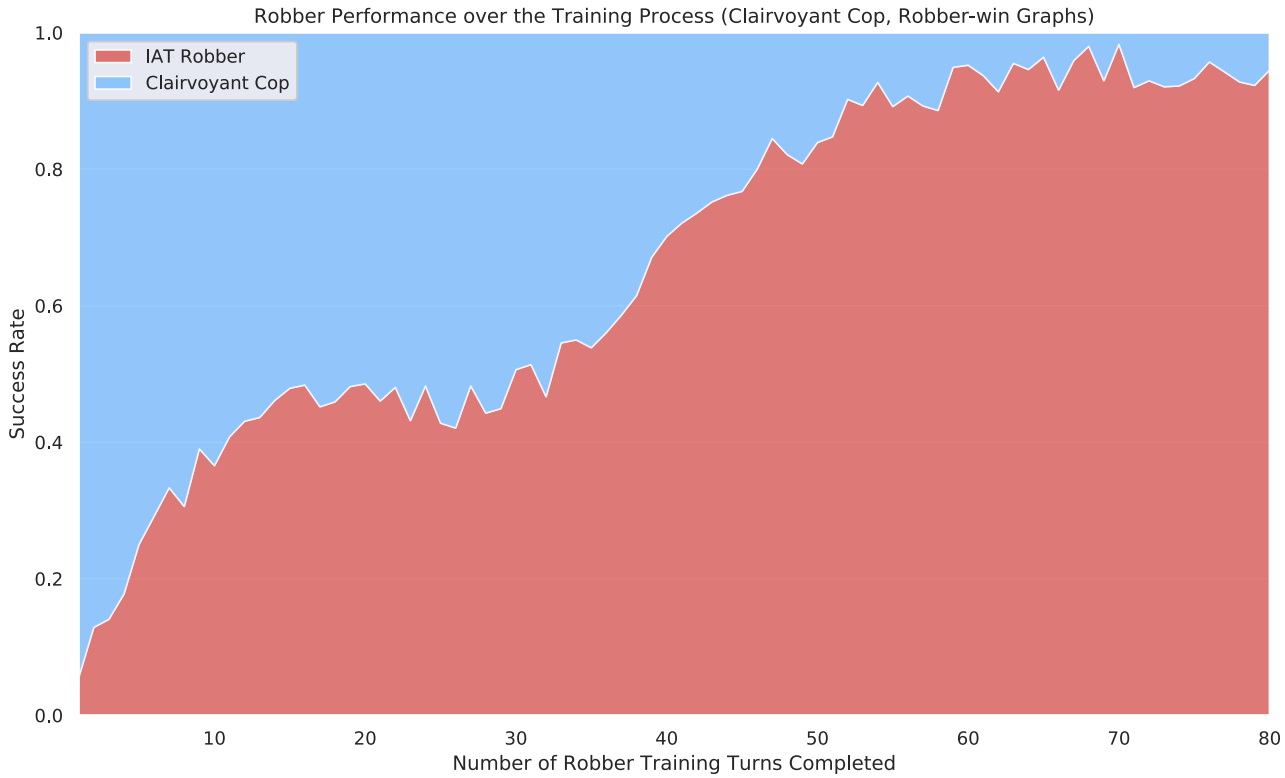

Figure 7.2: Plot depicting how a robber agent trained using the Iterative Adversarial Training algorithm performs against a Clairvoyant Negamax Cop when tested on robber-win graphs

The plot in **Fig. 7.2** was produced by allowing a robber trained using the IAT algorithm to compete against a clairvoyant Negamax cop on robber-win graphs. The initial distance between the cop and the robber was set to a random integral value in $[4, 10]$ during evaluation, and the maximum number of turns was set to 1.5 times the initial distance. While an optimally playing robber would achieve a success rate of 1.0, the success rate in this case increases from an initial value of 0.07 up to a saturation value of 0.96 over the course of training. An interesting element in this graph is the slowdown in training that happens from robber turn 20 to 35. This is the sequence of turns where the cop performance sees the sharpest increase as observed in **Fig. 7.1**. A possible effect from this rapid increase in the cop's performance is that the robber temporarily becomes unable to find rollouts where it manages to escape from its adversary. The presence of adaptive step counts (**Sec. 5.2.3.2**) alleviates this issue over time.

7.3.2 Effects of Vertex Pooling

This section describes the performance results that were obtained without vertex pooling (introduced in **Sec. 5.3**) for different problem sizes, and the improved results obtained by the addition of vertex pooling.

7.3.2.1 Without Vertex Pooling

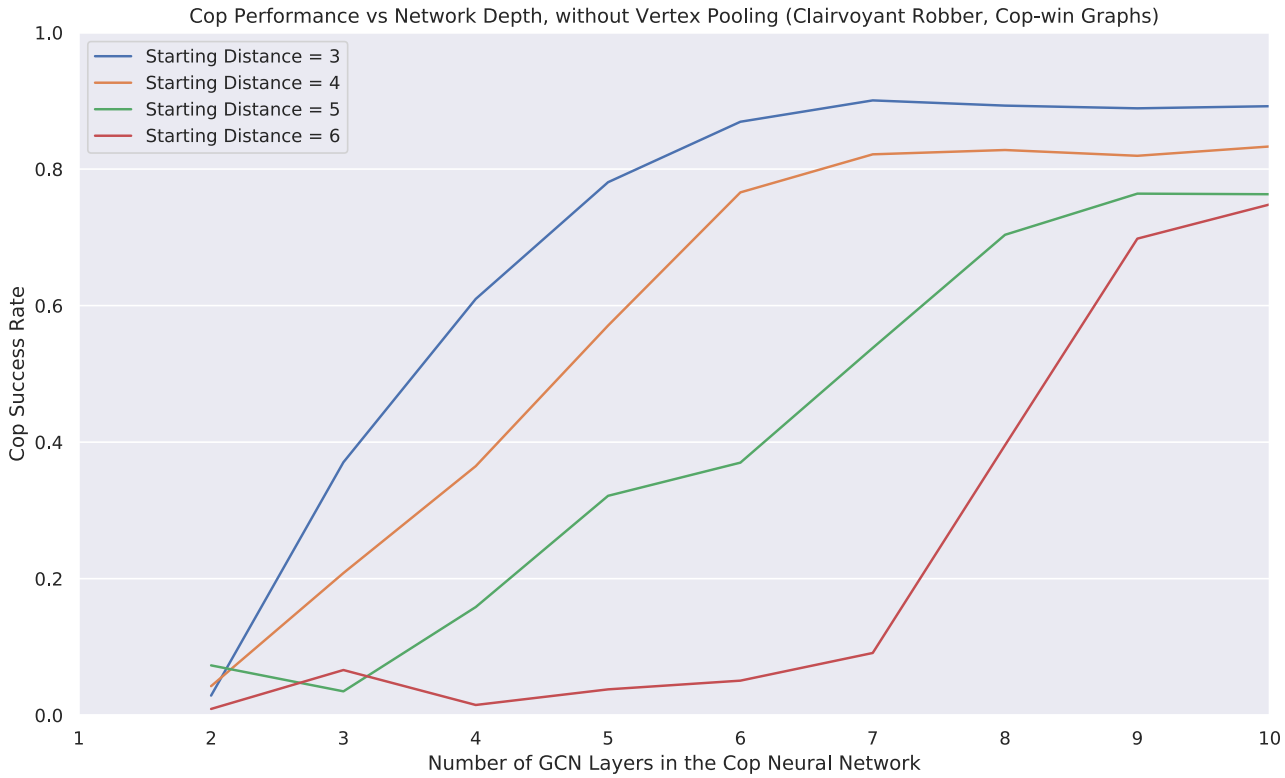

Figure 7.3: **Without Vertex Pooling**: Variation in Cop Performance with increasing number of GCN Layers for different starting distances between the cop and the robber. Measured on cop-win graphs.

The plot in **Fig. 7.3** was produced by allowing a cop trained using the IAT algorithm to compete against a clairvoyant Negamax robber on cop-win graphs. While an optimally playing cop would achieve a success rate of 1.0 irrespective of the starting distance, the maximum success rate (over

number of layers) when vertex pooling is not used decreases from 0.89 to 0.77 as the starting distance between the two agents increases from 3 to 6.

In addition, the scaling of the range of the agent with increasing number of GCN layers is sub-linear. For instance, for a 50% success rate with a starting distance of 3, only 4 GCN layers are required. This becomes 5 layers for a starting distance of 4, and nearly 9 for a starting distance of 6. Alternately stated, the number of GCN layers required to meet a particular accuracy threshold is super-linear in the required range. This is since the Dense layer that takes in the output of all the GCN layers receives a large number of inputs($N_{layers} \times |V| \times 2$), and extracting information out of the GCN-filtered output about the graph becomes harder and harder for the Dense layer with a much larger number of inputs.

This is aggravated by the need to have at least as many layers as the diameter of the graph for full operational range over the entire graph. For a small-world graph, note that the graph's diameter grows as the logarithm of the number of vertices in the graph. This implies

$$N_{layers} \geq c \log |V| \tag{7.7}$$

$$\text{Total number of outputs from GCN layers} \geq c|V| \times N_{layers} \times 2 \tag{7.8}$$

$$\text{Total number of outputs from GCN layers} \geq c|V| \log |V| \times 2 \tag{7.9}$$

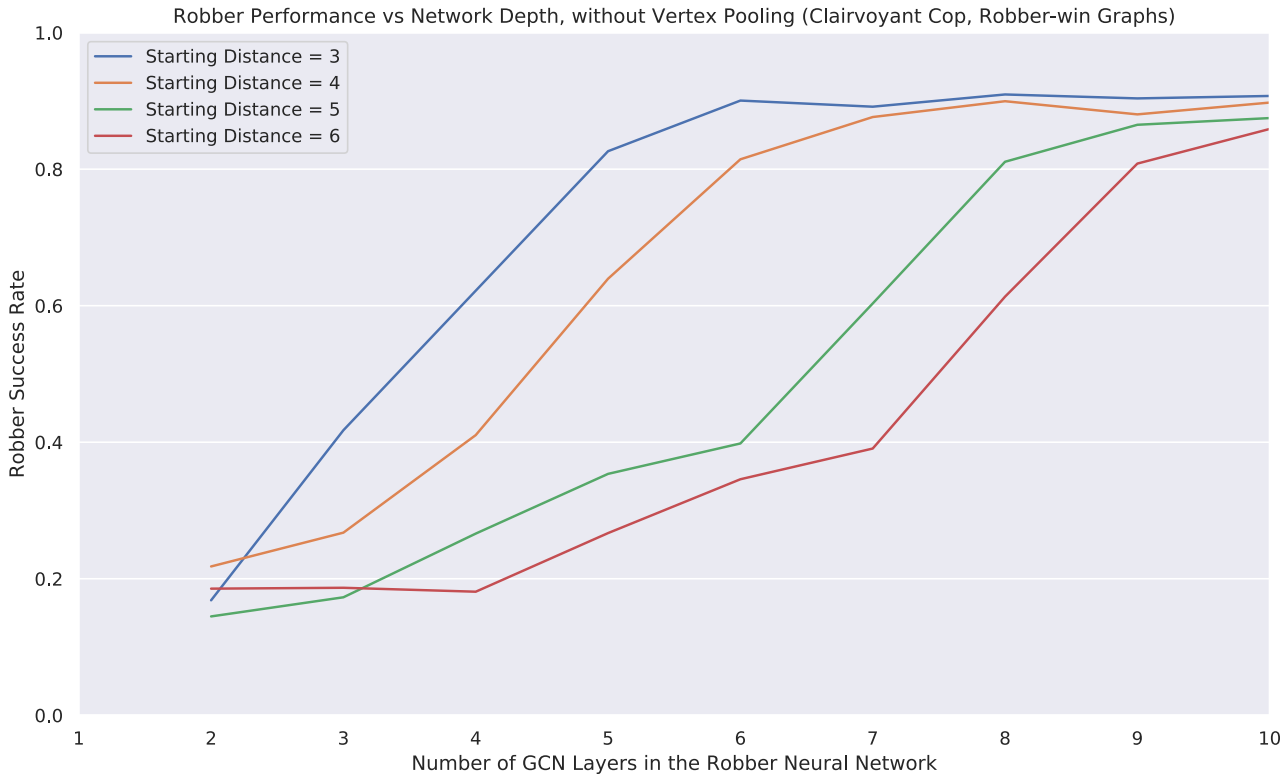

Figure 7.4: **Without Vertex Pooling**: Variation in Robber Performance with increasing number of GCN Layers for different starting distances between the cop and the robber. Measured on robber-win graphs.

The plot in **Fig. 7.4** was produced by allowing a robber trained using the IAT algorithm to compete against a clairvoyant Negamax cop on robber-win graphs. While an optimally playing robber would achieve a success rate of 1.0 irrespective of the starting distance, the maximum success rate (over number of layers) when vertex pooling is not used decreases from 0.91 to 0.86 as the starting distance between the two agents increases from 3 to 6.

The performance in case of the robber is somewhat better than that observed in case of the cop when vertex pooling is not used. This is expected since the robber-win graphs that we use are dominated by cycles of length greater than 4. The robber does not need significant foresight to maintain a constant distance from the cop on graphs with short cycles. However, as the diameter of the graph increases, the length of the cycles increases as well. To even identify that a section

of the graph forms a cycle, the agent needs many more GCN layers. In addition, the robber may also inadvertently lock itself into a “dead-end" in the graph when the cop is on the other side, thus resulting in the robber’s capture.

As in the case of the cop, the scaling of the range of the agent with increasing number of GCN layers is again sub-linear. For instance, for a 50% success rate with a starting distance of 3, only 4 GCN layers are required. This becomes 7 layers for a starting distance of 5, and 8 for a starting distance of 6.

7.3.2.2 With Vertex Pooling

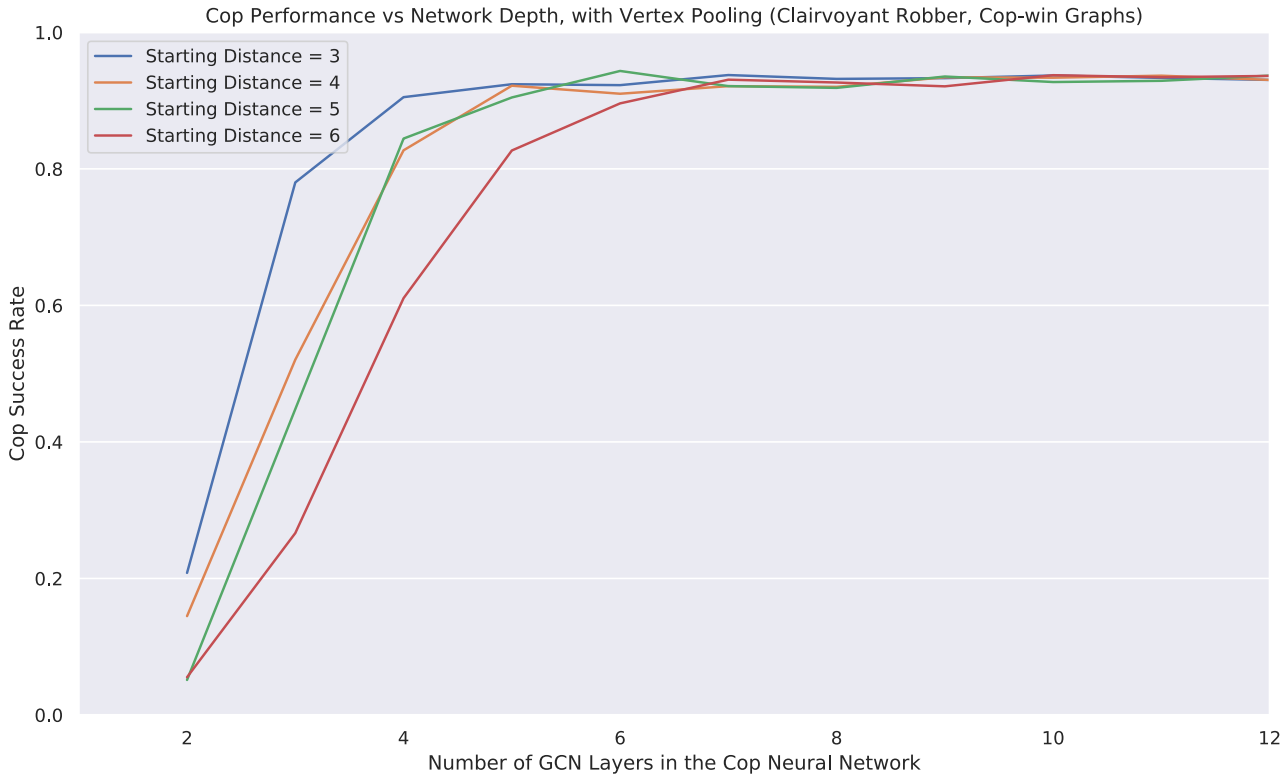

Figure 7.5: **With Vertex Pooling**: Variation in Cop Performance with increasing number of GCN Layers for different starting distances between the cop and the robber. Measured on cop-win graphs.

The plot in **Fig. 7.5** was produced by allowing a cop trained using the IAT algorithm to compete against a clairvoyant Negamax robber on cop-win graphs. While an optimally playing cop would achieve a success rate of 1.0 irrespective of the starting distance, the success rate when vertex pooling is used decreases from 0.93 to 0.92 as the starting distance between the two agents increases from 3 vertices to 6 vertices.

It is observed in this plot that there is a general increase in the range of the agent due to the introduction of vertex pooling. With only 4 GCN layers, the agent's success rate exceeds 60% whereas 8 layers were required in the case of no vertex pooling that was discussed in **Sec. 7.3.2.1**. In addition, the issue of a very large number of GCN-to-Dense connections that was bottle-necking the neural network is not as present here since the outputs of layer GCN layers is much smaller. Considering that each vertex pooling layer reduces the number of vertices in the graph to one-half its original value, the total number of outputs from d GCN layers is

$$\text{Total number of outputs from GCN Layers} = |V| \times 2 + \frac{|V|}{2} \times 2 + \frac{|V|}{2^2} \times 2 ... + \text{d terms} \quad (7.10)$$

$$= 2|V| \times \frac{(\frac{1}{2^d} - 1)}{\frac{-1}{2}} \quad (7.11)$$

$$= 4|V| \times (1 - \frac{1}{2^d}) \quad (7.12)$$

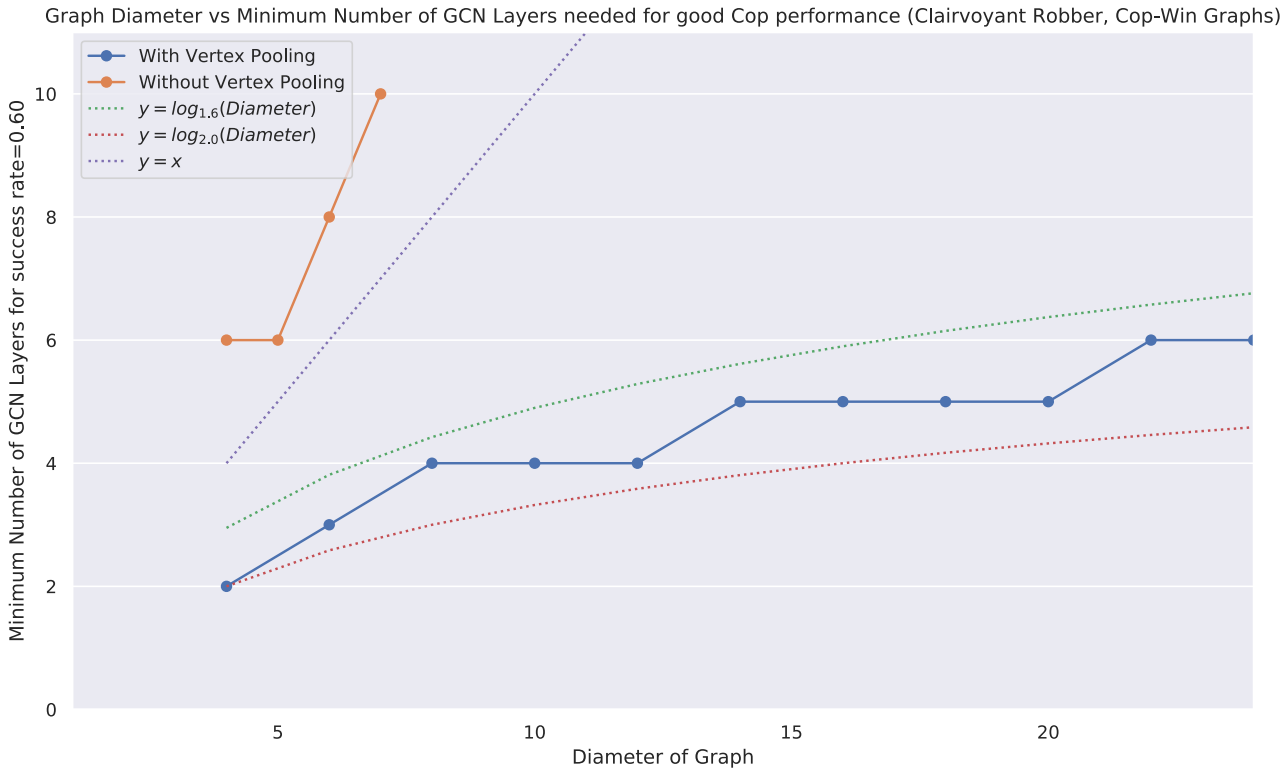

Figure 7.6: Number of GCN Layers required to achieve an acceptable cop success rate (0.60) against a clairvoyant robber, with and without vertex pooling. Measured on cop-win graphs of increasing diameters and start distances.

The plot in **Fig 7.6** was produced as follows:

1. Cops that use Vertex pooling are trained with IAT to play the game on graphs of a particular diameter.

2. The starting distance between the agents is set to half the diameter of the graphs, but to slightly lower values for larger graphs. The reduction for larger graphs is only to make the clairvoyant negamax algorithm tractable.

3. Upon training for a limited number of cop turns (35), training for that diameter setting is terminated and the performance is measured by evaluating against a clairvoyant negamax robber on cop-win graphs.

4. Steps 1 to 3 are repeated and the number of GCN Layers in the cop agent is increased until the success rate of the cop on that graph diameter is above 0.60.

5. The diameter of the graph is increased by 2 and the process is repeated.

The experiment was repeated for agents without vertex pooling as far as the available system memory allowed (10 GCN layers).

Within the domain of graph diameters that were evaluated using the above process, when vertex pooling was enabled, the number of GCN layers needed scaled with the diameter of the graph at a rate that is sandwiched by two logarithmic functions. In contrast, not using vertex pooling causes the scaling to be super-linear in the diameter of the graph.

7.3.3 Evaluation Against Alpha-Beta Pruning

Figure 7.7: Success rate of agents that use Alpha-Beta pruning, when competing against agents trained using the Iterative Adversarial Training Algorithm

The plot in **Fig 7.7** was produced by setting the initial distance between the two competing agents to 9. Following the approach in [4] and [5], the heuristic function that we use is the distance from the cop to the robber.

It can be observed that on unfavorable graphs, the IAT algorithm still manages to limit the Alpha-Beta pruning agent's performance to 0.81 for the cop and 0.89 for the robber. In contrast, the IAT algorithm obtained a mean success rate of 0.95 for the cop and 0.97 when playing on graphs that are favorable to it.

7.3.4 Evaluation Against Upper Confidence Tree Search

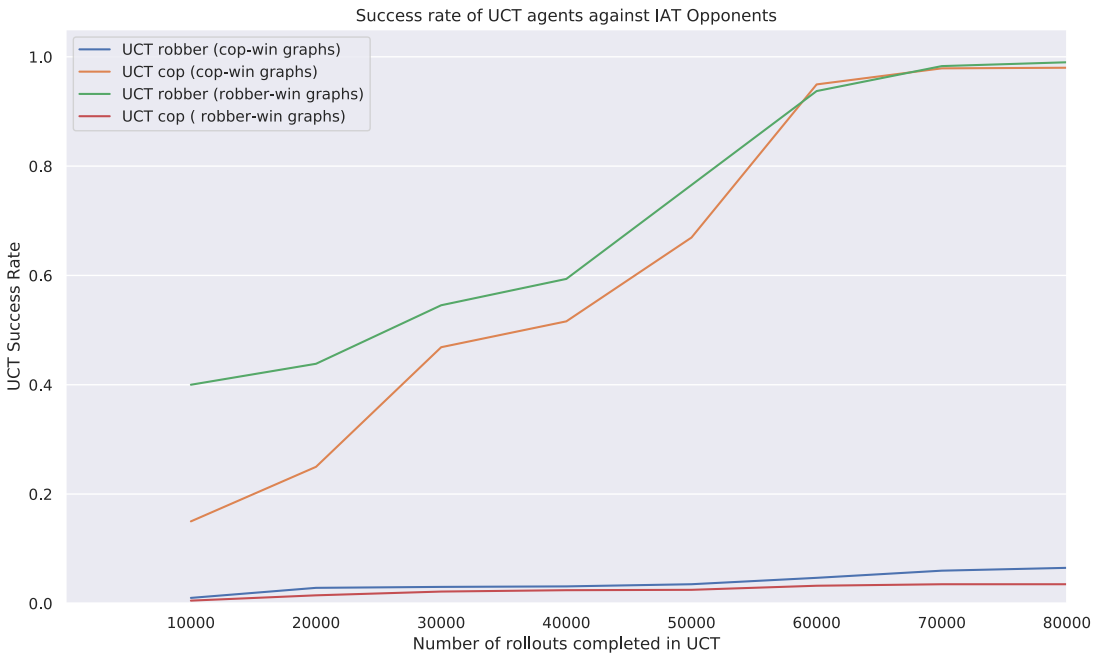

Figure 7.8: Success rate of agents that use Upper Confidence Trees, when competing against agents trained using the Iterative Adversarial Training Algorithm

The plot in **Fig 7.8** was produced by setting the initial distance between the two competing agents to 9. The number of rollouts was varied in increments of 10000, going up to 80000 rollouts per graph.

With 60000 rollouts for UCT, it can be observed that on unfavorable graphs, the IAT algorithm limits the Alpha-Beta pruning agent's performance to 0.949 for the cop and 0.937 for the robber. In contrast, the IAT algorithm obtained a mean success rate of 0.953 for the cop and 0.968 for the robber when playing on graphs that are favorable to it.

However, UCT's performance exceeds the IAT agents' performance when the number of rollouts exceeds 70000. At 80000 rollouts, when playing against an IAT opponent, the UCT algorithm achieves a performance of 0.981 for the cop and 0.989 for the robber for graphs favorable to the UCT agents. In contrast, the IAT algorithm only achieves a success rate of 0.935 for the cop and 0.965 for the robber.

It must be noted that the UCT algorithm had to perform a fresh set of rollouts for every graph that the game was played on, while the IAT agents required no such precomputation after the training phase is over.

8. CONCLUSIONS

This chapter concludes this thesis report, summarizes our findings and contributions, and suggests directions for future research.

8.1 Summary of Work

An important first step for this thesis was to produce a Markov Decision Process formulation of the cops and robbers problem. This was followed by an in-memory representation that allows this formulation to be usable for computation. Subsequently, the thesis looked at a simpler variant of the game where only the cop was allowed to make moves, and this problem was used to analytically find out what neural architectures could be viable. We then designed a neural network architecture for a deep reinforcement learning agent that showed near-perfect success rates on the path planning problem.

With this agent as the substrate for further work, the thesis then moved on to the problem of training both the cop and the robber to play optimally, and we decided to do this using an iterative process where both agents learn from each other in regular, alternating turns. Upon a basic version of the proposed Iterative Adversarial Training approach, a number of modifications were introduced that increase the stability and robustness of the process and allow for equitably training the two agents. In addition, our implementation of vertex pooling allowed the trained agents to require only a logarithmic increase in the number of GCN layers in order to extend their operational range. This is in contrast to an approach that does not use vertex pooling, which requires a linear (super-linear upon empirical observation) number of GCN layers to achieve a similar increase in their operational range.

It was observed that when competing against a clairvoyant, optimal opponent, the Iterative Adversarial Training approach achieved a success rate of 0.93 for cops and 0.96 for robbers when the testing was done on cop-win and robber-win graphs respectively. This is superior to Alpha-Beta pruning (with a distance heuristic), and to UCT search (up to 6×10^4 rollouts). Importantly,

our agents were able to achieve this without the requirement of precomputation for each graph.

8.2 Distinctions from Previous Approaches

Apart from the use of deep reinforcement learning to solve the cops and robbers problem, there are a number of important distinctions between our approach and previous methods:

- A focus on generalization to graphs that were not observed during training. Heuristic approaches aim to identify the optimal strategy for a single graph. They thus require a potentially expensive precomputation phase before they can start playing on each graph, or require a significant amount of computation before each step.

- The use of an opponent that learn over time. Previous approaches compared and evaluated against non-optimal opponents that do not evolve and apparently do not offer a worthy opponent for the agent to learn from.

- Performance evaluation when playing against an optimal opponent. Prior publications ([4], [6]) on this topic evaluate against an opponent that again runs using heuristic methods and let their agents compete against these. [5] does use an optimal agent as a benchmark to measure against, but not as a competitor. These comparisons does not give a clear picture of how close the agent is to optimal performance.

8.3 Potential Extensions and Alternate Methods

Future extensions of this work can include:

- **Using a team of multiple cops to capture a single robber**: This turns the problem into a collaborative as well as adversarial game. The question of whether all cops should follow the same policy, or whether different cops can assume specific roles in the game and have different strategies is also interesting.

- **Hybrid Approaches**: Instead of learning the cop and robber policies through RL, firstly, a hybrid approach can be followed where the optimal policies for a large number of graphs

are evaluated using an algorithmic approach. Subsequently, a GCN can be trained on these optimal policies and generalized to predict the policy for a new graph given the structure of the graph. This would just involve supervised learning and may offer faster convergence to a general solution.

- **Generalization to more variants of the cops and robbers problem**: There are other interesting variants that are different from this deterministic version of cops and robbers problem. For instance, the agents could have only partial visibility of the graph - this would force the cops to spread themselves across the graph to ensure that they have most of it covered, while the robber needs to balance between exploring the graph enough to know where the cops are, and avoiding harming itself by accidentally revealing its position to the cops.

- **Attention on Graphs**: Incorporating attention into various classes of deep neural networks has led to significant performance increases in recent research. The cops and robbers problem can benefit from the introduction of attention techniques.

REFERENCES

[1] R. Nowakowski and P. Winkler, "Vertex-to-vertex pursuit in a graph," *Discrete Mathematics*, vol. 43, no. 2-3, pp. 235–239, 1983.

[2] N. Nisse, *Cops and Robber games and applications*, 2013.

[3] M. Mamino, "On the computational complexity of a game of cops and robbers," *Theoretical Computer Science*, vol. 477, pp. 48–56, 2013.

[4] C. Moldenhauer and N. R. Sturtevant, "Evaluating strategies for running from the cops," in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[5] A. Isaza, J. Lu, V. Bulitko, and R. Greiner, "A cover-based approach to multi-agent moving target pursuit.," in *AIIDE*, 2008.

[6] V. Bulitko and N. Sturtevant, "State abstraction for real-time moving target pursuit: A pilot study," in *AAAI Workshop: Learning For Search*, pp. 72–79, 2006.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[12] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, pp. 3844–3852, 2016.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[14] O. Macindoe, L. P. Kaelbling, and T. Lozano-Pérez, "Pomcop: Belief space planning for sidekicks in cooperative games," in *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[15] W. Xiong, T. Hoang, and W. Y. Wang, "Deeppath: A reinforcement learning method for knowledge graph reasoning," *arXiv preprint arXiv:1707.06690*, 2017.

[16] J. Jiang, C. Dun, and Z. Lu, "Graph convolutional reinforcement learning for multi-agent cooperation," *arXiv preprint arXiv:1810.09202*, vol. 2, no. 3, 2018.