SOLVING GRAPH COLORING PROBLEM BY USING MONTE CARLO TREE SEARCH AND ARTIFICIAL NEURAL NETWORKS

by

Buğra Altuğ

Engineering Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

2019

SOLVING GRAPH COLORING PROBLEM BY USING MONTE CARLO TREE SEARCH AND ARTIFICIAL NEURAL NETWORKS

APPROVED BY:

Prof.Dr. Emin Erkan Korkmaz (Supervisor)	……………………………
Assist.Prof.Dr. Onur Demir	………………………………
Assist.Prof.Dr. Dionysis Goularas	………………………………

DATE OF APPROVAL: /   /2019

ACKNOWLEDGEMENTS

First of all, I am especially thankful to my supervisor Prof. Dr. Emin Erkan Korkmaz and Çağrı Yeşil for their guidance and support through the creation of this project.

Also, I would like to thank my family for supporting my decisions and being with me when I needed them.

ABSTRACT

SOLVING GRAPH COLORING PROBLEM BY USING MONTE CARLO TREE SEARCH AND ARTIFICIAL NEURAL NETWORKS

Over the past few years, with the development of technology, machine learning techniques (especially artificial neural networks) proved their importance again. Upon this, recent achievement of AlphaGo Zero has proven that an agent that collects its own data and performs an efficient learning on this data could exceed superhuman performance. It has been seen that the agent can create its own model without knowing anything about the domain.

Self-learning agent could be used in complex areas where humans don’t have perfect knowledge. Solving a graph coloring problem by examining the graph’s characteristics is such a domain. It is possible to design an agent that plays graph coloring games to extract information and by learning that information, the agent can color the graph. This study provides a method where an agent utilizing Monte Carlo Tree Search and artificial neural networks can solve relatively simple graph coloring problems all by itself. This study presents in-depth information about this graph coloring agent.

ÖZET

MONTE CARLO ARAMA AĞACI VE YAPAY SİNİR AĞLARINI KULLANARAK GRAFİK BOYAMA PROBLEMLERİNİN ÇÖZÜLMESİ

Son birkaç yılda, teknolojinin gelişmesiyle birlikte, makina öğrenmesi teknikleri (özellikle yapay sinir ağları) yeniden önemli olduklarını kanıtladılar. Bunun üzerine AlphaGo Zero'nun son başarısı, alanı ile ilgili hiçbir bilgisi olmayan bir ajanın kendini eğiterek insanüstü performansı geçebileceği kanıtladı.

Kendi kendine öğrenebilen bir ajan, insanların çok bilgisinin bulunmadığı karmaşık alanlarda kullanılabilir. Örneğin bir graf boyama probleminin çözümüne ulaşmak adına (graf'ın tavrını inceleyerek) kendi yaklaşımını geliştirebilir. Bu tezde, oynadıkça bilgi çıkaran ve çıkan bilgiyi daha iyi oynamak için kullanan bir ajan tasarlanmış ve uygulanmıştır. Ajan, Monte Carlo arama ağacı ve yapay sinir ağı kullanarak nispeten basit graf boyama problemlerini kendi başına çözebilir. Bu proje, bahsi geçen ajan hakkında detaylı bilgi içermektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SYMBOLS / ABBREVIATIONS

MCTS	Monte Carlo Tree Search
GPU	Graphics Processing Unit
AI	Artificial Intelligence
ML	Machine Learning
UCB	Upper Confidence Bound
CUDA	Compute Unified Device Architecture (Nvidia)

LIST OF EQUATIONS

1. INTRODUCTION

Machine Learning (ML) techniques have shown how promising they are with today's technological advances. These techniques are handy on the complex problems whose solutions need a long list of rules. Problem with the ML techniques is that they depend on human knowledge in most cases. That's why ML techniques have never exceeded the superhuman capabilities.

The recent achievement of AlphaGo Zero shown a promising reinforcement method where a tree search algorithm (named Monte Carlo Tree Search) is used to extract information. Using this method, DeepMind's agent, starting from *tabula rasa*, exceeded superhuman capabilities without using a single human knowledge. This method could also be used to train an agent to solve problems where humans don't have sufficient knowledge

Project's objective is utilizing the mentioned reinforcement method for extracting information from a pseudo game which is a representation of a graph coloring problem. The aim is training an agent with this extracted information which will make the agent understand and ultimately solve this given pseudo-game.

Accordingly, this report is structured as follows: Chapter 2 discusses the generic information and background of graph theory, graph coloring, artificial intelligence, search algorithms of artificial intelligence, and Monte Carlo tree search. In chapter 3, used methods and created algorithms are explained. Chapter 4 contains information about usable hardware and software to implement a graph coloring agent. Moreover, chapter 4 contains information on the mandatory hyper-parameters and a class diagram of the graph coloring agent. In chapter 5, the agent is tested on some graph coloring problems and the acquired results are mentioned. The last chapter discusses the deficiencies and future work of the project.

2. BACKGROUND

2.1. Previous work

This section discusses the research on graph theory, general search tree algorithms that are used in the artificial intelligence combined with Monte Carlo tree search and some areas of machine learning. These are relevant to this thesis. First, it presents the related work on graph coloring and graph labeling in the discipline of graph theory which is a study of mathematical graphs. Second, it presents the related work in artificial intelligence types, why search algorithms are needed and give an overview of the Monte Carlo Search Tree. Finally, it presents artificial neural networks in machine learning, reinforcement learning and how to combine reinforcement learning with Monte Carlo Tree Search.

2.1.1. Graph Theory

The graph is defined by a set named vertices (*V)* and a set of pairs named edges (*E*). A vertex is often labeled by some unique alphanumeric characters (for example “v1”, “v2”, etc…). Similarly, the edge is labeled by alphanumeric too (for example *“e1”, “e2”, etc…*). Furthermore, every edge label also corresponds to a pair of vertex label. This pair is edge’s starting and ending vertices (for example: *“e1 = (v1, v2)”, “e2 = (v1, v3)”, etc…*) [1].

Example representation of graph would be “*G = (V, E)*” where vertices are “*V = {v1, v2, v3, v4, v5}”* and edges are “*E = {e1, e2, e3, e4, e5} = {(v1, v2), (v2, v5), (v5, v5), (v5, v4), (v5, v4)}”*. This representation defines the graph in the Figure 2.1.

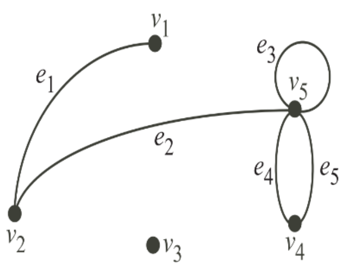


Figure 2.1 Representation of a Graph **[1]**

Although, there are mainly two graph types (Directed and Undirected). This project will deal only with the undirected graphs. While the edges of an undirected graph are bidirectional, directed graph' are unidirectional. This means that, for undirected graph, in an edge's corresponding label pair (*"e1 = (v1, v2)"*), swapping the starting and ending vertices would not result in a different graph. [2]. A visual difference could be observed in Figure 2.2 and Figure 2.3.

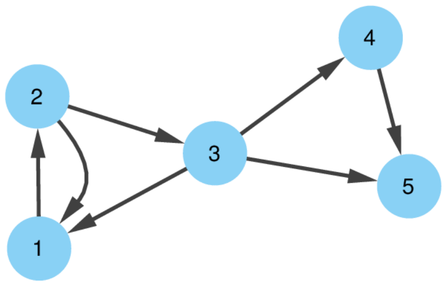


Figure 2.2 Directed Graph **[3]**

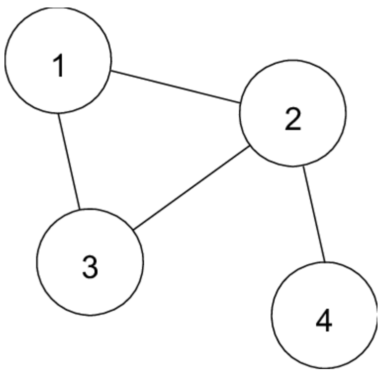


Figure 2.3 Undirected Graph **[4]**

Furthermore, there are some distinct labeling techniques. A graph is said to be labeled, without any qualification, means that each node labeled uniquely (but arbitrarily). So that all nodes have distinct enumeration [5]. But in this project, graph coloring will be used instead of the default one. The difference is that, in graph coloring, one assigns colors to the vertices of the graph so that no two adjacent vertices have the same color [6]. This could be seen in the graph that is labeled according to graph coloring in Figure 2.4.

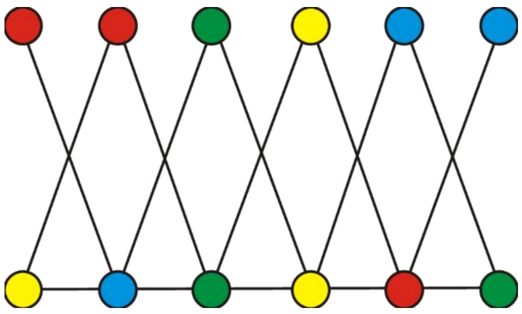


Figure 2.4 Graph Coloring. **[7]**

2.1.2. Artificial Intelligence, Search Algorithms, and Monte Carlo Tree Search

The definition of artificial intelligence could be organized into two main approaches where each approach contains two main categories. Human-centered approach defines artificial intelligence as a system that thinks like humans or acts like a human. Whereas the rationalist approach defines as a system that thinks rationally or acts rationally. The human-centered approach is an empirical science while the rationalist approach is a combination of mathematics and engineering. Acting rationally could be defined as a system acting according to its own beliefs. This project will focus mainly on the acting rationally (the rational agent approach) [8]. Among other things, the definition of agent is a system that can perceive and act in an environment. Simple agent functionality can be seen in Figure 2.5. There are some different kinds of agent types like reflex-agent (Figure 2.6), utility-based agent (Figure 2.7), and goal-based agent (Figure 2.8).

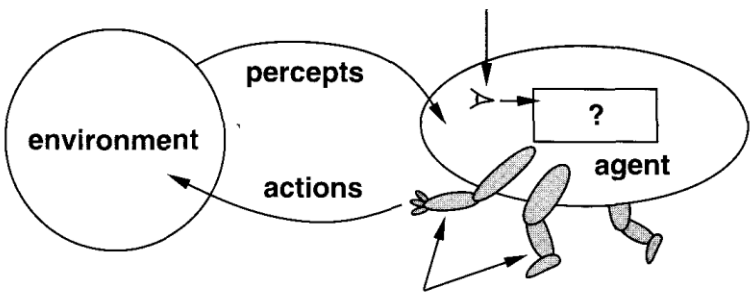


Figure 2.5 Generic Agent **[8]**

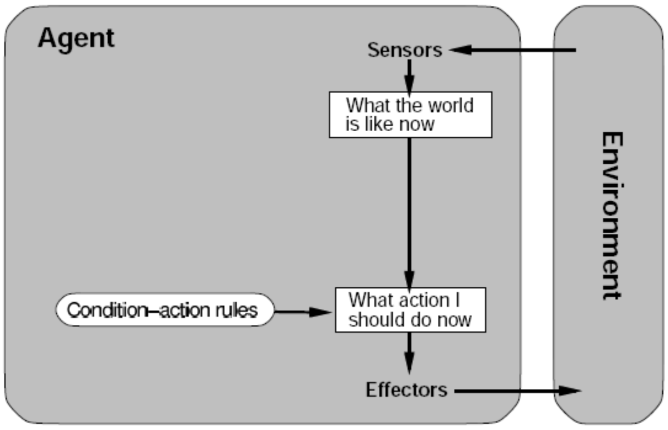


Figure 2.6 Reflex Agent **[8]**

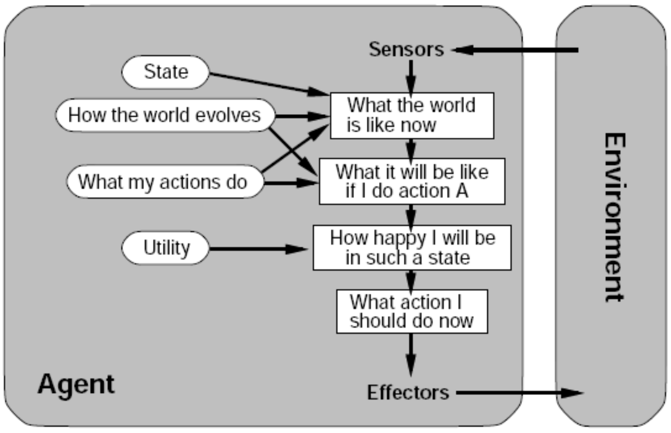

Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Effectors
Environment

Figure 2.7 Utility-Based Agent **[8]**

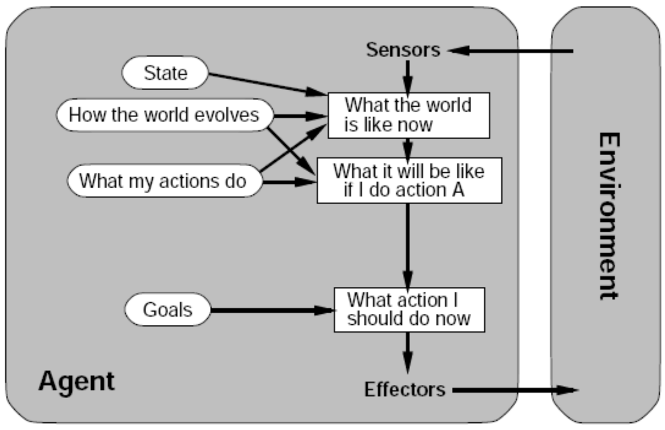

Sensors
State
How the world evolves
What my actions do
Goals
What the world is like now
What it will be like if I do action A
What action I should do now
Agent
Effectors
Environment

Figure 2.8 Goal-Based Agent **[8]**

Furthermore, there is a special goal-based agent called problem-solving agent which will be used in this project. Problem-Solving agent's working principle is that the agent selects an action which is in an action set. It's such a set that, if these actions are applied sequentially, it would result in the best state in the observable state space. The agent needs to search action space to determine which action set is the best one. Additionally, action space's creation process is named problem formulation. A simple problem-solving agent's pseudocode is in Algorithm 1**.**

Algorithm 1 Problem-Solving Agent **[8]**

```
function SIMPLE-PROBLEM-SOLVING-AGENT(perception) returns an action
        create sequence, state, goal, space, action
        state ← UPDATE-STATE(state)
        if sequence is empty then
             goal ← FORMULATE-GOAL(state)
             space ← FORMULATE-PROBLEM(state, goal)
             sequence ← SEARCH(space)
        action ← RECOMMENDATION(sequence, state)
        sequence ← REMAINDER(sequence, state)
        return action
```

Unfortunately, when dealing with the complex environments, best action set may not be found in the action space within an acceptable amount of time. The solution to this problem is letting agent act according to limited rationality and selecting an optimal action set. Some of the algorithms for finding theoretically best action are minimax and Monte Carlo tree search [8].

Both of these algorithms are mainly used for two-player zero-sum games. In minimax, the agent tries to maximize its own utility (score of the game state) while trying to decrease its opponents. When computational budget (or acceptable amount of time) is exceeded, returns the first action is its best action set. In Algorithm 2 and Algorithm 3, the computational budget is not presented but the main parts of the algorithm are described [8].

Algorithm 2 MINIMAX Decision **[8]**

```
function MINIMAX-DECISION(game) returns an operator
        for each op in OPERATORS[game] do
            VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
        end
        return op with the highest VALUE[op]
```

Algorithm 3 MINIMAX-VALUE **[8]**

```
function MINIMAX-VALUE(state, game) returns a utility value
        if TERMINAL-TEST[game](state) then
            return UTILITY[game](state)
        else if MAX is to move is state then
            return the highest MINIMAX-VALUE of SUCCESSORS(state)
        else
            return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

On the other hand, the Monte Carlo tree search is also trying to select the best action according to the utility. The distinguishing feature is that instead of calculating utility values of every resulting state for every action in the action space (until the computational budget is reached), it randomly explores the action space [9].

Its algorithm can be seen at Algorithm 4. Tree policy selects a leaf node, then creates a child node to it. Finally returns that child node. On the other hand, default policy plays out the domain from a given state (until the terminal state) to produce a utility value [9].

Algorithm 4 Monte Carlo Tree Search **[9]**

```
function MCTS(state) returns an action
        create vroot
        vroot ← root node with state
        while within computational budget do
             v ← TREEPOLICY(vroot)
             q ← DEFAULTPOLICY(v)
             BACKPROPAGATE(v, q)
        return ACTION(BESTCHILD(vroot))
```

2.1.3 Machine Learning, Reinforcement Learning, and Artificial Neural Networks

Generally, Machine learning is the science and art of programming agents that can learn from data. Some other definitions are:

"[Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed". —Arthur Samuel, 1959

"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E." —Tom Mitchell, 1997

Machine learning's main advantages are that they can dynamically adapt to new information, solve complex problems where traditional approaches are not enough or need a long list of rules. There are some fields of machine learning like supervised learning, unsupervised learning and reinforcement learning [10].

In Figure 2.9, the program uses a supervised machine learning algorithm. Where in its training set, every mail (instance) has a label whether corresponding mail is spam or not? In the end, this program will try to predict the new instance's possibility of being a spam [10].

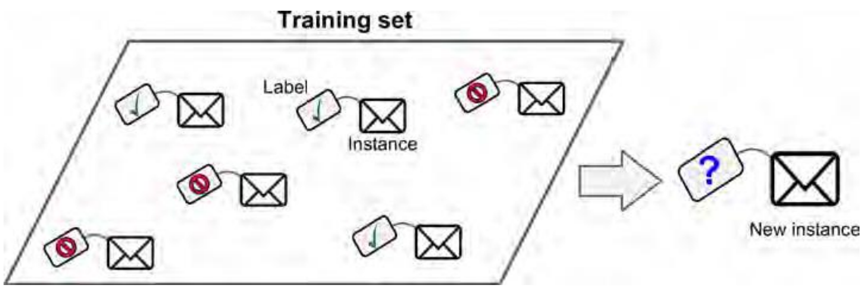


Figure 2.9 Supervised Learning **[10]**

Furthermore, in Figure 2.10, the program uses unsupervised learning. As one can see, instances in the training set are not labeled. The unsupervised algorithm will try to understand the relationship between these instances (for example it will try to cluster the instances) [10].

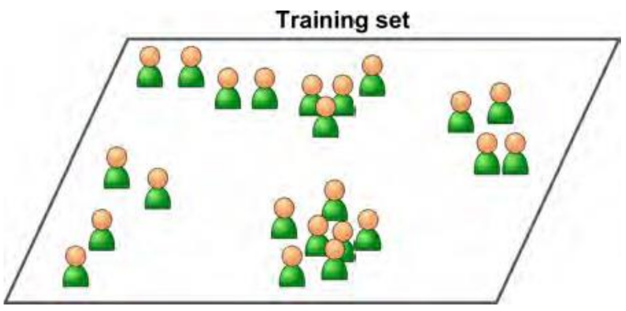


Figure 2.10 Unsupervised Learning **[10]**

This project will mainly focus on reinforcement learning instead of supervised and unsupervised techniques. In reinforcement learning, the agent can observe, interact with the environment and get rewarded by those interactions (can be seen in Figure 2.11). The agent tries to discover the best strategy, called *policy*, to get the most reward over time [10].

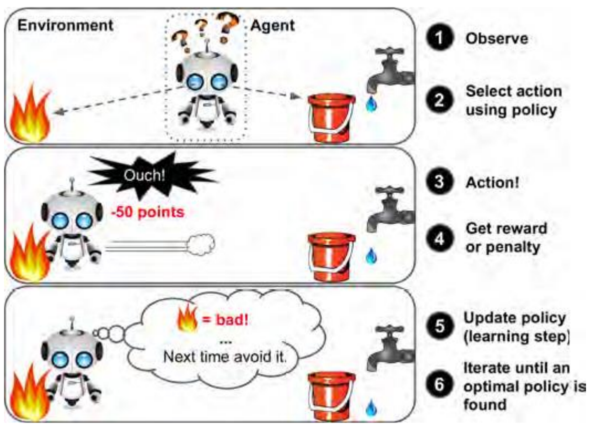

Environment
Agent
1 Observe
2 Select action using policy
Ouch!
-50 points
3 Action!
4 Get reward or penalty
= bad!
...
Next time avoid it.
5 Update policy (learning step)
6 Iterate until an optimal policy is found

Figure 2.11 Reinforcement Learning **[10]**

In the field of machine learning, there are many algorithms that one can use. But projects focus will be given to the neural network because of its recent achievement in the complex environments [10]. Before the discussion of the neural networks, biological neuron (Figure 2.12) must be quickly examined.

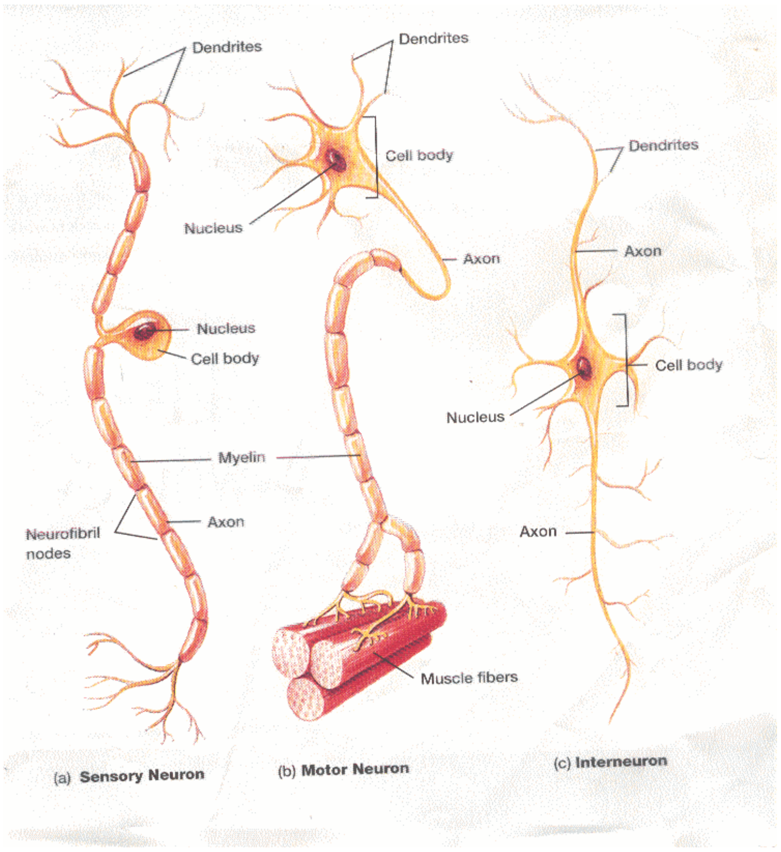


Figure 2.12 Biological Neuron **[11]**

A neuron is the fundamental unit of the nervous system. There approximately 100 billion neurons in the human body. Neurons tend to locate into compact groups or sheets. These groups usually run in a common direction and form a compact bundle like nerve, tract, peduncle, brachium, etc... Moreover, biological neurons are specialized to:

“1. To receive information from the internal and external environment;

2. To transmit signals to other neurons and to effector organ;

3. To process information (integration) and

4. To determine or modify the differentiation of sensory receptor cells and effector cells.” [11]

Motivation in the creation of artificial neural networks is their remarkable power to derive meaning in the complex and uncertain environments. They detect such patterns and trends that neither humans nor other computer programs can. After the detection of the patterns (train), the neural network can answer “what if” questions. Furthermore, there is an important difference between other machine learning techniques and neural networks are that the neural network creates its own representation and the organization of the information [11].

There are many neural network architectures but this project will use multilayer perceptron. Multilayer perceptron’s every unit produces an output by performing a biased weighted sum of its inputs and passing the result to an activation function (can be seen in Figure 2.13). Learning in this system involves adjustments to weights and biases. Moreover, units are arranged in a layered topology (Figure 2.14). Biological neuron’s cell body, dendrites, synapse, and axon corresponds to artificial neuron’s node, input, weight, and output respectively [11].

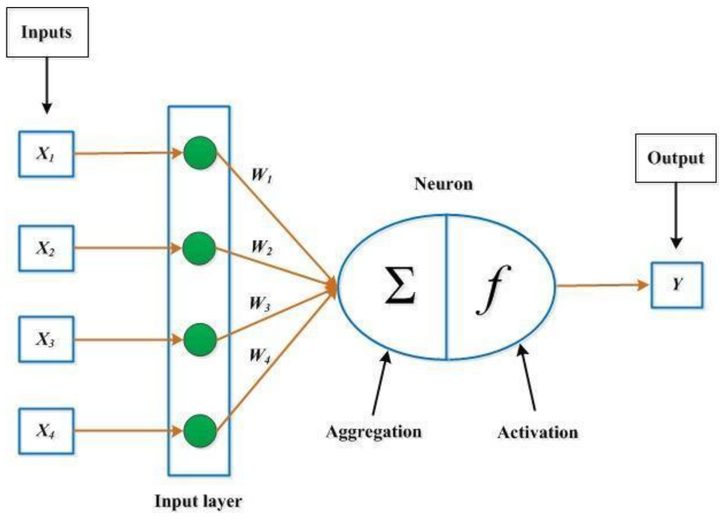


Figure 2.13 Single Artificial Neuron **[12]**

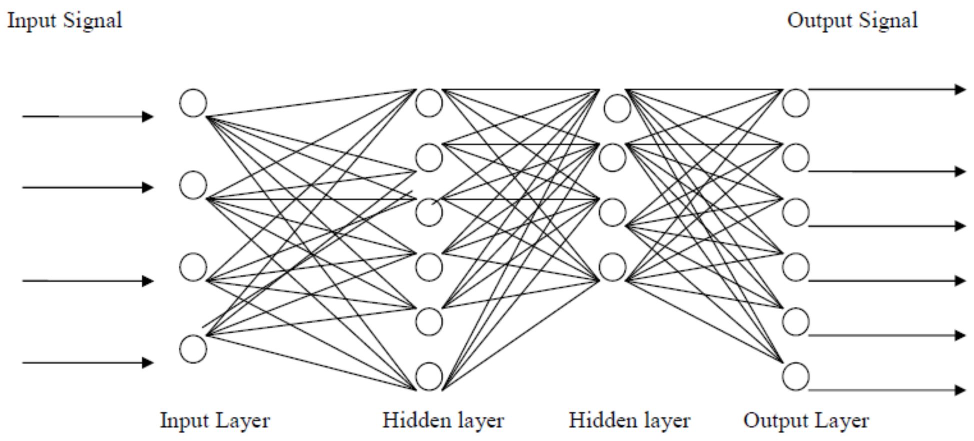


Figure 2.14 Multilayer Perceptron **[11]**

Additionally to the previously given information, now the combination of Monte Carlo Tree Search and neural network will be mentioned, with respect to “AlphaGo Zero”. In AlphaGo Zero, for each game state “*s*”, Monte Carlo tree search is executed where its *default policy* is done only by the neural network [13].

When MCTS simulations are ended for the turn, action will be selected according to the search probabilities computed by the MCTS and these probabilities are stored. Later, in the terminal position of the game, the game score is stored. The training part of the AlphaGo

Zero done by feeding the neural network the game state, its stored probability and end game score [13].

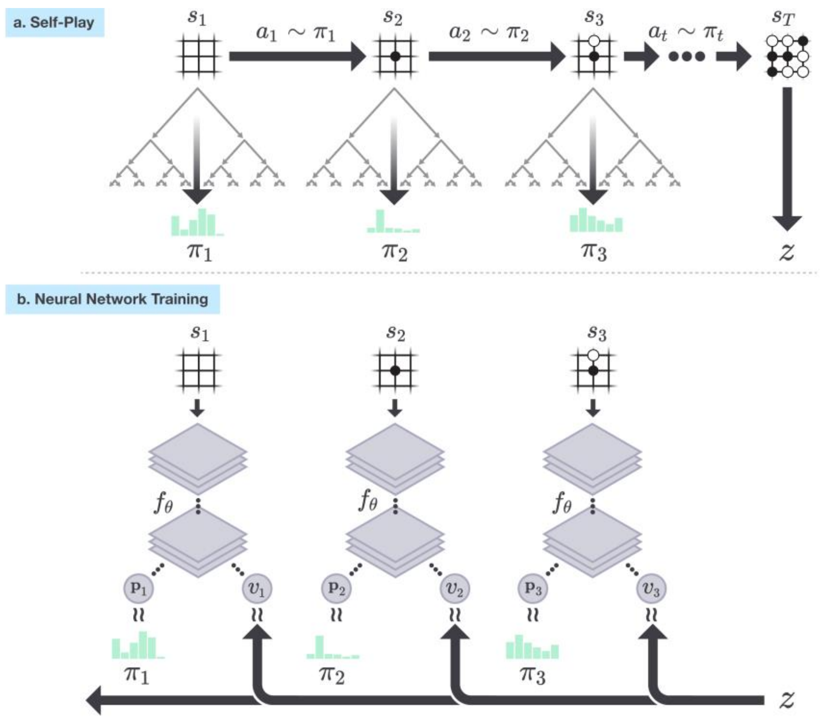


Figure 2.15 AlphaGo Zero **[13]**

3. ANALYSIS AND DESIGN

There have been researches about agents that try to solve graph coloring problems. Nevertheless, there is an improvable point. If dependence to domain knowledge is eliminated, then finding solutions might become easier. Especially in complex domains like graph coloring problems. This knowledge dependence could be removed by creating an agent with an MCTS and a reinforced neural network that is trying to solve a graph coloring problem. This will cause the agent to acquire its own knowledge about the given graph's characteristics and ultimately to create its own approach to solve the problem.

For this approach, graph coloring problems are needed to be transformed into a single player game first. In this kind of game, in each turn, the player should select a color to color the current vertex. Also, there should be a turn for every vertex and the order the vertices will be colored must be set with respect to the graph's vertex layout.

Player of this artificial *puzzle* needs an algorithm which applies the most promising action while trying to learn which action is really the best one for a given state. Due to this need, this project will use the tree search algorithm named MCTS which lets the agent balance its exploitation and exploration dilemma. Furthermore, extracted information about the explored actions is used to train a neural network which will be used afterward. These operations will cause the agent to create its own solution approach.

3.1. Graph Coloring Game

Before giving further information about the agent, deeper information about the graph coloring game will be given. A graph is represented by two distinct arrays, namely *edges* and *sequence*. *Edges* array is an array of arrays. Each index represents a unique vertex in the graph's vertex layout. Moreover, each element in any index's array represents the neighboring vertices of the corresponding vertex. For example, if the given graph is like the one in Figure 2.3, then "*edges = [[2, 3], [1, 3, 4], [1, 2], [2]]"*.

On the other hand, the *sequence* is just one-dimensional array where each index represents a unique vertex's current color. Furthermore, colors are represented as unique unsigned integers. Also, there is a no-color option which is "0". Initially, *sequence* array is filled with "0"s. *Edges* and *Sequence* together represents a graph coloring game's state.

Every graph coloring game has a number of turns which is equal to a number of vertices in the graph. That means in each turn, the player must paint the given vertex or pass by painting it to "0". As known, the difficulty of finding solutions to graph coloring problems lies in the rule which imposes that a vertex cannot be painted to the same color of its neighbors. This rule also applies for the graph coloring game which could be seen at Algorithm 5. Furthermore, a single episode (game) of the graph coloring game could be seen in Figure 3.1.

Algorithm 5 Graph Coloring Game Check Legal Move

```
function Legal(graph, color) returns an boolean
        if color = 0 do
             return True
        else do
             sequence ← GetSequence(graph)
             current ← GetCurrentVertex(graph)
             edges ← GetEdges(graph)
             adjacents ← GetIndexedList(edges, current)
             for vertex in adjacents do
                  if GetIndexedColor(sequence, vertex) =  color do
                       return False
             return True
```

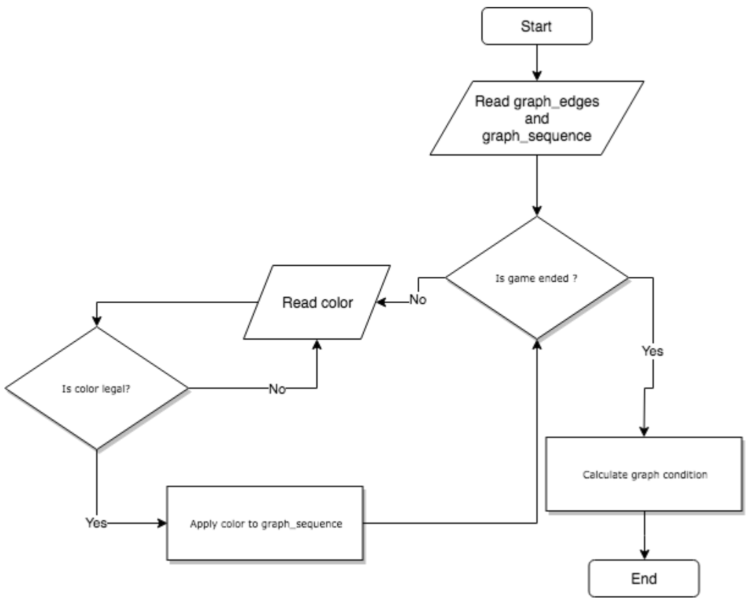

Start
Read graph_edges and graph_sequence
Is game ended ?
No
Read color
Is color legal?
No
Yes
Apply color to graph_sequence
Yes
Calculate graph condition
End

Figure 3.1 Graph Coloring Game Flowchart

3.2. Monte Carlo Method

Before explaining the further details of the agent, general information about the Monte Carlo method must be given. This method relies on executing many random walks (which do not need any domain knowledge) and aggregating the gathered results to find an approximated solution. These specifications make this method fast, convergent, and able to escape from any local optima. An example plot of some random walks can be seen in Figure 3.2. Monte Carlo method is used in the MCTS at simulating a sample game. [14]

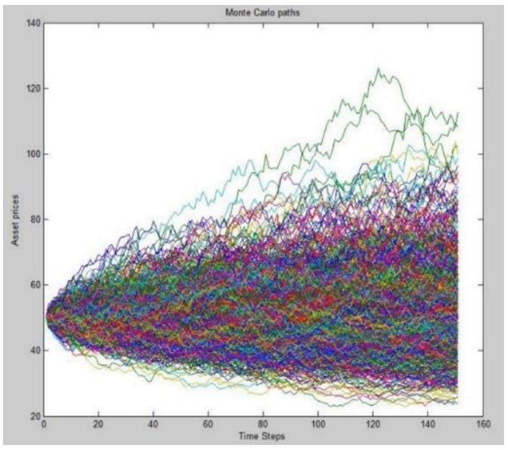


Figure 3.2 Monte Carlo Method **[15]**

3.3. Monte Carlo Tree Search

The simple definition of MCTS was given in section 2.1.2, more depth details will now be given. MCTS is a method that is used to find optimal actions by taking random walks in the action space and building a search tree according to it. Because of the random walks, the algorithm does not need any domain knowledge.

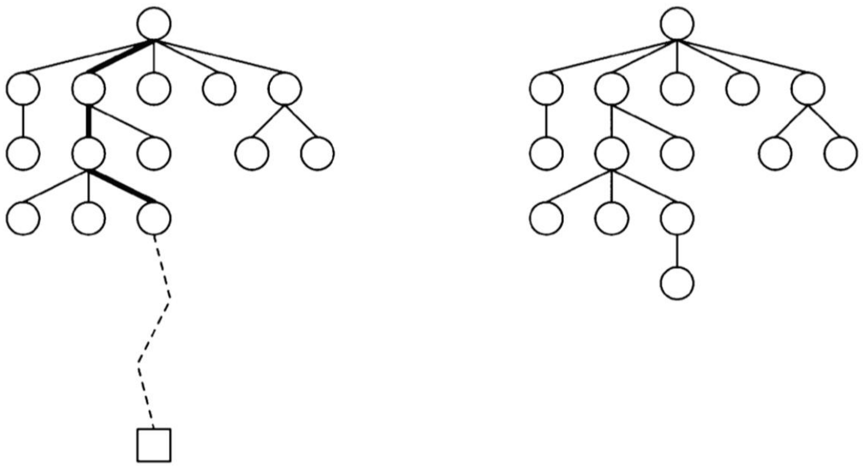


Figure 3.3 MCTS Search Tree **[9]**

As shown in Figure 3.3, a tree is built in incremental and asymmetric order. Each iteration of the MCTS contains:

- Tree Policy: Used to select the most urgent leaf node of the current tree. MCTS will select such a leaf node that tree will try to balance itself for exploration and exploitation considerations. Exploration consideration is looking at the areas that have not been well sampled yet. Exploitation consideration is looking at the areas which appear to be promising. Finally, created a new child node to the selected leaf node.
- Simulation: A simulation is then run, starting on the freshly created node until a terminal node is reached. Moves of the simulation are selected according to the default policy. Default policy, in the simplest case, is selecting random moves.
- Backpropagation: In this final part, updates the statistics of both the simulated node's and its ancestors according to the simulation.

MCTS contains at least some number of iterations. At each iteration, MCTS converges to the solution. [9]

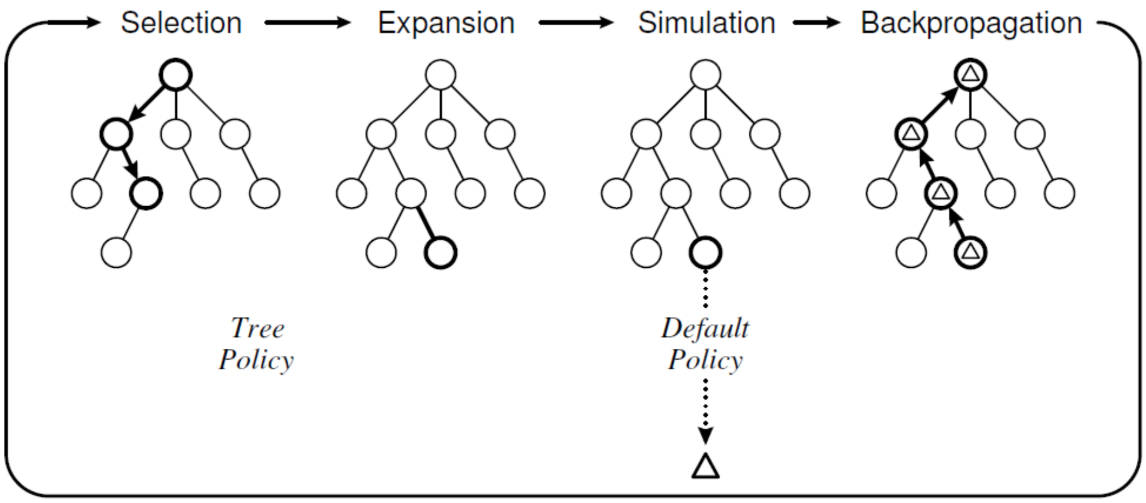


Figure 3.4 Iteration of MCTS **[9]**

3.4. AlphaGo Zero

Like MCTS, also a piece of simple information about AlphaGo Zero was given in section 2.1.3. Now more detailed information will be given. As being said, the most surprising thing about AlphaGo Zero is that it started from *tabula rasa* and acquired a superhuman proficiency in a challenging domain without human knowledge [13]. This was acquired with their reinforcement technique which is an MCTS with a default policy where a neural network tries to predict the given state's applicable actions probabilities (π) and ultimate utility value of the end game (z) rather than randomly walking until the terminal state. When the game terminates, for each state, AlphaGo Zero trains the neural network with the state's search tree's statistics, state's information, and terminal state's utility value. This can be seen at the Figure 2.15. At every game, AlphaGo Zero plays with itself and ultimately MCTS converges and neural network learns this convergence.

Each edge in the search tree stores a probability value, action value, and visit count. Probability value represents the probability of selecting the corresponding action. Action value represents the utility value of a pseudo terminal state. This pseudo terminal state is the best expected terminal state after the given action is applied. Together, probability and actions values are predictions of the neural network. On the other hand, visit count is the number of times the corresponding node is selected. Tree Policy of the neural network, starting from the root node, iteratively selects the moves that maximize upper confidence bound until a leaf node is encountered. As can be seen in the Equation 1. "***Q(s, a)***" is the action value of the given state and "***U(s, a)***" is Equation 2 where "***P(s, a)***" is the probability and "***N(s, a)***" is the visit count of the given state.

$$UCB = Q(s, a) + U(s, a)$$

Equation 1 Formula of Upper Confidence Bound

$$U(s, a) \propto P(s, a) / (1 + N(s, a))$$

Equation 2 Formula of function U

In one game (episode) there are some number of turns. Each turn contains numerous MCTS iterations. Moreover, when some number of episodes are ended, the algorithm compares two neural networks, the one with and the one without the knowledge gained in these episodes. This is called an iteration in AlphaGo Zero. [13]

3.5. The agent of Graph Coloring Game

After the principles of MCTS and AlphaGo Zero have been summarized, detailed information about graph coloring game's agent will be given. MCTS's fundamental component is a *node*. Starting from a root node, together they form the tree structure. *Node* structure contains *a parent node, children nodes*, *visit count*, *utility value* (which will be called *q value* now on), *color* and *probability value*.

The tree grows (like in Algorithm 6) iteratively, using outputs of the neural network. The most important thing about this tree structure is the *backup* algorithm (Algorithm 7). It will update statistical information about the nodes which will be used in the selection of the *best child* (Algorithm 8) and in the training of neural network.

Algorithm 6 Grow Node Tree

```
function Grow(node, probabilities, utilityValue)
        for probability in probabilities do
            color ← GetColor(probability)
            childNode ← CreateNode(node, color, probability)
            AddParent(childNode, node)
            AddChildren(node, childNode)
            Backup(childNode, utilityValue)
```

Algorithm 7 Backup

```
function Backup(node, q)
        tmpVisits ← GetVisits(node)
        SetVisits(node, tmpVisits + 1)
        q’ ←  ((tmpVisits * GetQ(node)) + q) / GetVisits(node)
        SetQ(q’)
        if node has parent do
            Backup(GetParent(node), q)
```

Algorithm 8 Best Child

```
function BestChild(node) returns a node
        create bestUcb, selected
        bestUcb ← -∞
        for child in GetChildren(node) do
             ucb ← CalculateUcb(node, child)
             if ucb > bestUcb do
                  bestUcb ← ucb
                  selected ← child
        return selected
```

As the basic structure of the MCTS's node is mentioned, now information about MCTS simulation will be given. In every simulation, the graph coloring game's current state is copied. This sample game is then played using the actions of the most promising nodes in the search tree. Promising node selection starts from the root node and continues until a new leaf node is discovered or a terminal state is encountered. These promising nodes are determined by computing Equation 1 using the information on each node. The important part is that, when a new leaf node is discovered, action of that leaf node is applied and the resulting state is then fed to the neural network. Flowchart of this simulation can be seen in Figure 3.5.

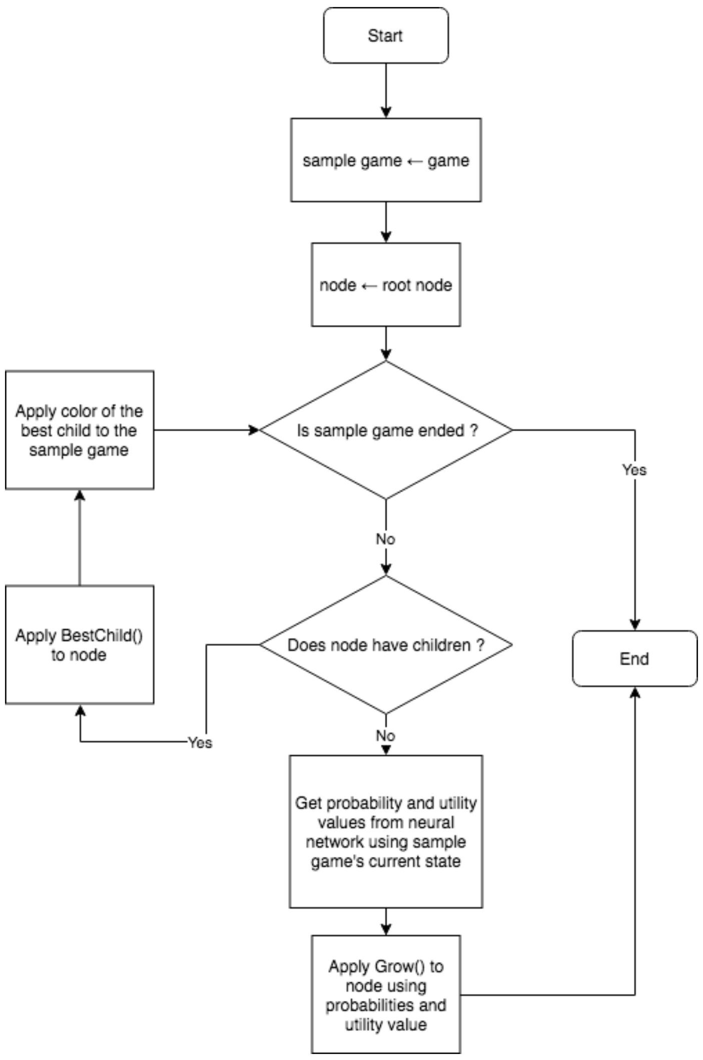

Start
sample game ← game
node ← root node
Is sample game ended ?
Yes
No
Apply color of the best child to the sample game
Does node have children ?
Yes
No
Apply BestChild() to node
End
Get probability and utility values from neural network using sample game's current state
Apply Grow() to node using probabilities and utility value

Figure 3.5 MCTS Simulation of Graph Coloring Game

In every turn, MCTS simulates some number of games. At the end of each simulation, sample game state resets to the real game state. When this desired number of simulations end (a turn ends), MCTS calculates π values. These π values and the applied color are added to the *ExpandableEpisodicDataSet*.

Value of π contains a list of each color's selection probabilities in the current game state, calculated by MCTS. Single child of the root node's probability calculation operation is done by dividing the child node's visit count by parent node's visit count. If a child node's color is "0" (no-color), then calculation operation will be passed on that node and all the values of π are rescaled to make their sums up to one.

Algorithm 9 Calculate Pi

```
function CalculatePi(root) returns a float array
    create $\pi$
    for child in GetChildren(root) do
        probability ← GetVisits(child) / GetVisits(root)
        SetChild ($\pi$, child, probability)
    remove GetChildWithColor($\pi$, 0)
    RescaleValues($\pi$)
    return $\pi$
```

Color is selected with a weighted distribution of random choice where weights are *π*. There is also a threshold *T* for turn count. After the threshold is reached, *color* is set to the biggest probability on *π*. This is giving us the ability to tune additional exploration chance. At the end of each turn, the root node is set to the selected color’s node. Flowchart of a turn could be seen at Figure 3.6.

Algorithm 10 Select Color

```
function SelectColor(turnCount, turn, treshold, $\pi$) returns an int
    if turn < turnCount * treshold  do
        color ← RandomDistributed($\pi$)
        return color
    color ← MaximumOf($\pi$)
    return color
```

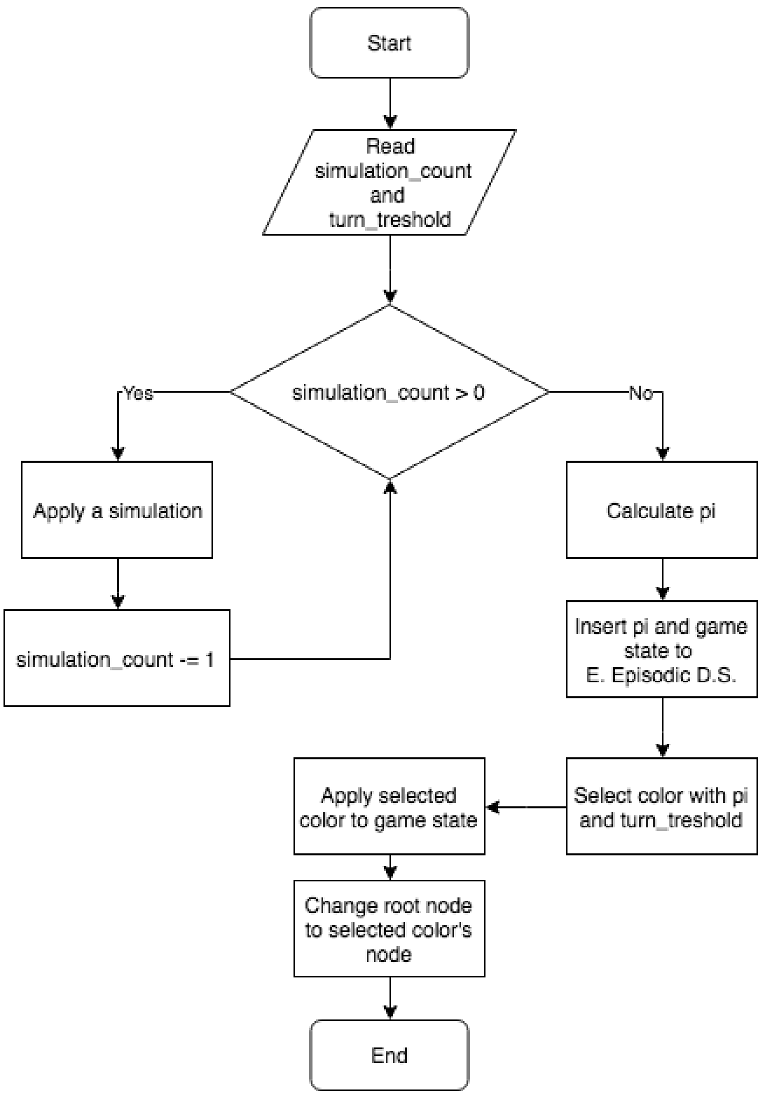

Start
Read simulation_count and turn_treshold
simulation_count > 0
Yes
No
Apply a simulation
simulation_count -= 1
Calculate pi
Insert pi and game state to E. Episodic D.S.
Select color with pi and turn_treshold
Apply selected color to game state
Change root node to selected color's node
End

Figure 3.6 Graph Coloring Agent's Turn

Each game, MCTS creates an *ExpandableEpisodicDataSet* which contains information about a *game state*, π value for each turn. When a turn ends, MCTS inserts the utility value of the terminal state to *ExpandableEpisodicDataSet*. If wanted, new information could be derived using the previous game's data. This derivation is done by changing the sequence of colors (Algorithm 11). These turns and derivation operation together creates a single game of graph coloring agent. Flowchart could be seen at Figure 3.7.

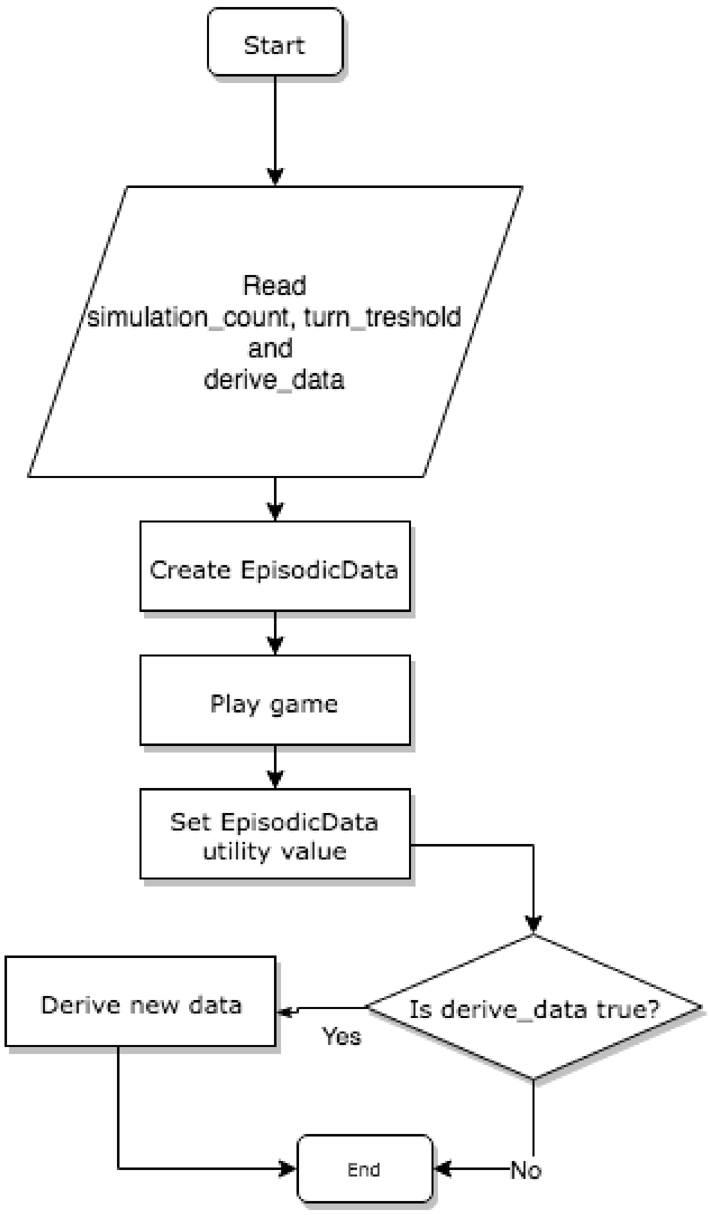

Start
Read simulation_count, turn_treshold and derive_data
Create EpisodicData
Play game
Set EpisodicData utility value
Is derive_data true?
Yes
Derive new data
No
End

Figure 3.7 Graph Coloring Agent’s Game

There is another game type called comparison. It is used to determine how good a neural network’s result will be with minimal exploration. For this minimal exploration, one needs to set the threshold in Algorithm 10 to “0”. Except for this threshold change, there is no difference between a normal game and a *comparison* game.

Algorithm 11 Derive Data

```
function Derive(dataset, colors) returns a data set
        create dataset'
        r ← GetUtility(data)
        for data in dataset do
             state ← GetState(data)
             $\pi$ ← GetPi(data)
             for color in colors do
                  create state', $\pi'$
                  for color' in GetColors(state) do
                     state', $\pi'$←SwapPositions(state,$\pi$,color,color')
                     AddData(dataset', state', $\pi'$, r)
        return dataset'
```

At the end of each game, *ExpandableEpisodicDataSet* is extracted to the main dataset. When some number of games are played by Graph Coloring Game's agent, if there are no solutions found, after copying the current neural network, main dataset is shuffled and ultimately fed to the current neural network.

For a reasonable improvement in the neural network, high-quality data must be used for training. In the reinforcement learning cases, especially in their complex situations, one can't be sure about the quality of data because of the exploration/exploitation dilemma. That's why the quality of the extracted information must be measured (even this procedure may let an ML model get stuck on some local optimum).

Extracted information's measurement is done by playing a *comparison* game with two neural networks, the one with and the one without the knowledge of last training. Winner of this one *comparison* game gets to be the default neural network.

A number of game plays, training, and the *comparison* are called one iteration of Graph Coloring Game's Agent and can be seen at Figure 3.8. Graph coloring agent plays some number of iterations to find the graph coloring solutions. A general state diagram of this agent could be seen at Figure 3.9. Formed states in the run, iteration, game, turn, and simulation has shown.

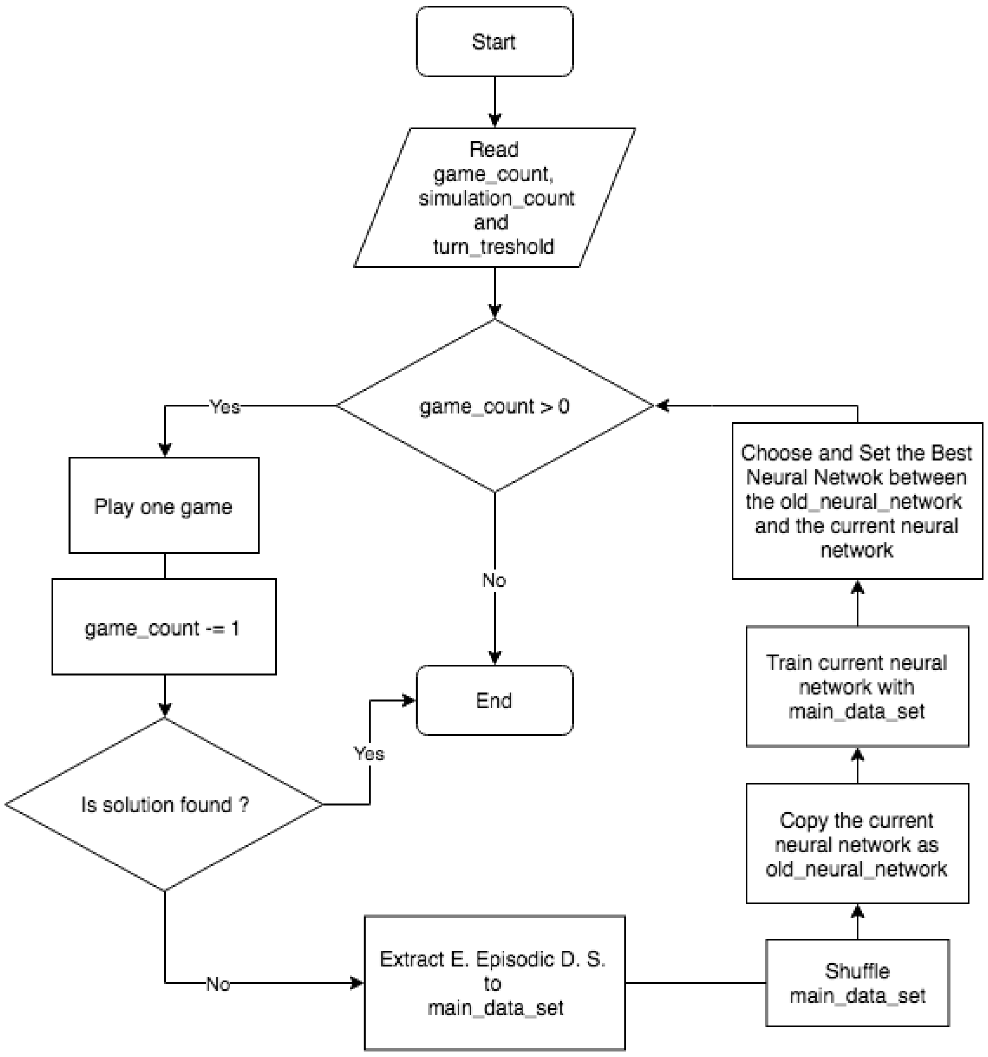

Start
Read game_count, simulation_count and turn_treshold
game_count > 0
Yes
Play one game
game_count -= 1
Is solution found ?
Yes
End
No
No
Extract E. Episodic D. S. to main_data_set
Shuffle main_data_set
Copy the current neural network as old_neural_network
Train current neural network with main_data_set
Choose and Set the Best Neural Netwok between the old_neural_network and the current neural network

Figure 3.8 Graph Coloring Agent's Iteration

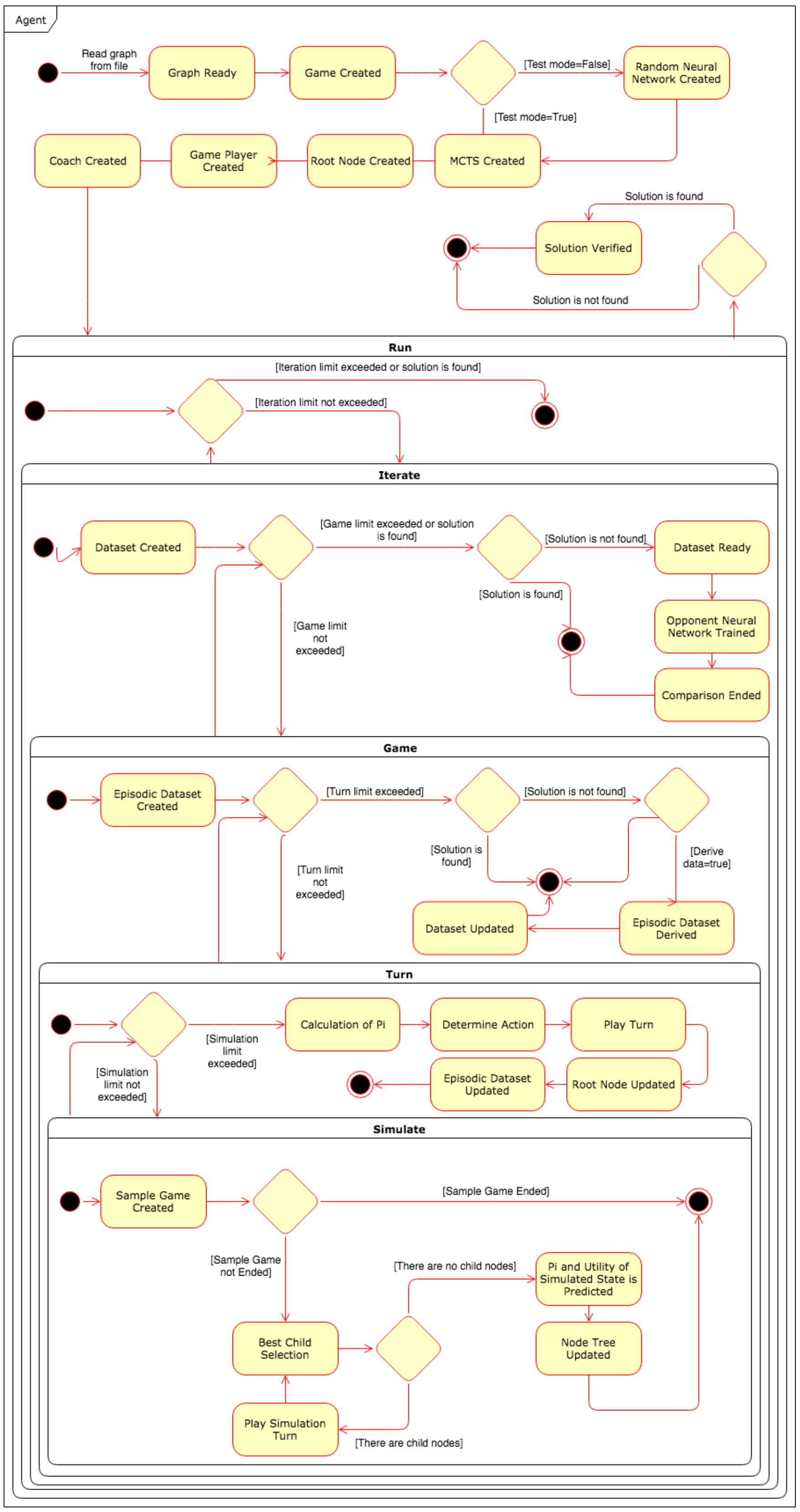
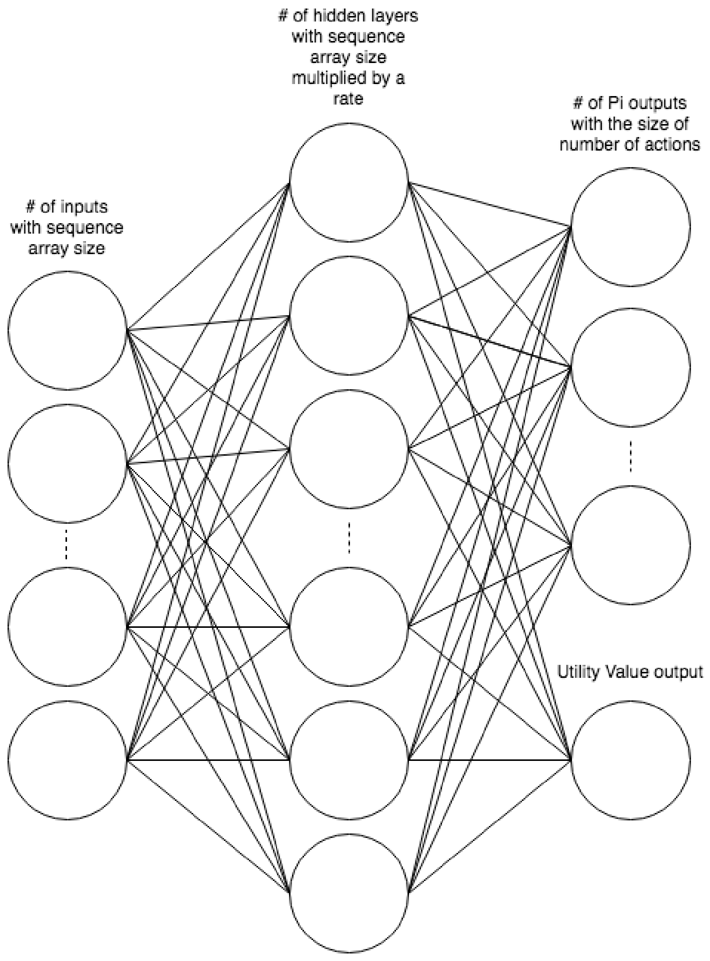


Figure 3.9 State Diagram of Graph Coloring Agent

In this last part, information about the neural network will be given. Agent's neural network has an input layer with the size of the game's state size which fully connects to a hidden layer with the size of the game's state size multiplied by a hidden layer rate. The activation function of these hidden layers is rectified linear unit. This hidden layer is fully connected to two output layers, the one which will predict the action probabilities and the other one which will predict the utility value. Used loss functions are respectively cross entropy and mean-squared error. Also, ridge regularization (L2 regularization) is utilized to prevent overfitting. Design of the neural network could be seen at Figure 3.10.



# of hidden layers with sequence array size multiplied by a rate
# of Pi outputs with the size of number of actions
# of inputs with sequence array size
Utility Value output

Figure 3.10 Design of the Neural Network

3.6. Example Run of Graph Coloring Agent

In this section an example run of the graph coloring agent will be discussed. Graph in the Figure 3.11 will be used. This graph has three vertices with two edges. First and second vertex is colored in blue and red respectively while and third one is uncolored. This run will contain one iteration, one game, one turn, and one simulation for the sake of simplicity. Agent will have two color possibilities. First one will be blue (represented as 1) and the second one will be red (represented as 2). Agent will also be able to choose not to paint (represented as 0).

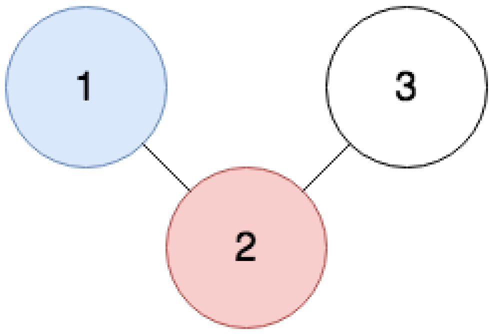


Figure 3.11 Example Graph

Initially *edges*, *sequence*, *next vertex*, *colored vertex count*, and *vertex count* of the graph structure will be set to: "*edges = [[2], [1, 3], [1, 2], [2]]*", "*sequence = [1, 2, 0]*", 2, *2*, *3* respectively. Agent will start the run by executing one iteration. In the iteration, an empty *main_data_set* will be created. Then *game_count* (initially 1) will be checked whether it is bigger than zero or not. In the example, it is bigger. So agent will continue by executing one game. In this one game, an empty *EpisodicData* will be created and turns will be played until the game ends. *Sequence* array has one uncolored vertex initially. Therefore one turn will be played to paint that vertex. This turn will contain one simulation.

In that simulation, an empty node tree will be created. Root node of that tree will contain null parent node, null child nodes, 0 visit count, 0 utility value, and null probability initially. Best child selection will be tried on that root node and will be failed because the root node do not have any children. Then this root node's corresponding state (which is "*sequence = [1, 2, 0]*") will be fed through the neural network. Using this state information, neural network will predict something like *0.7, 0.2, 0.1* for pi and 1 for *utility value* (best expected terminal state). Afterwards, root node will create *3* new children. Set their probability values according to the predicted pi. That means first child with the action 1 will have a probability of *0.7* and so on. Furthermore, for every children, *parent node* is set to *root* node, *child nodes* is set to *null*, and *visit count* is set to 1. *Utility value* for both root node and children are set to the predicted utility value. Finally, *visit count* of parent node is set to *3* and the simulation part ends.

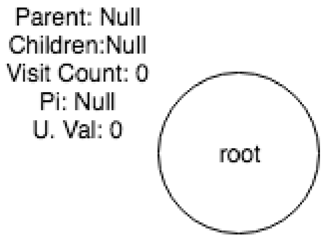


Figure 3.12 Example Node Tree Before Grow

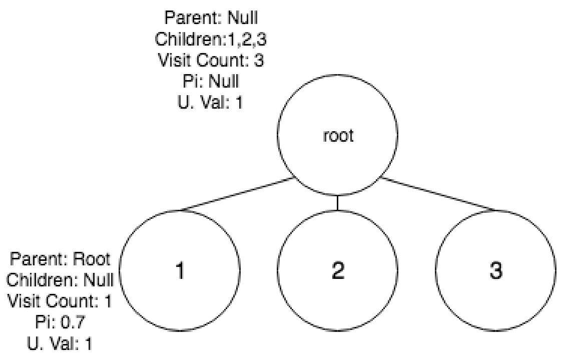


Figure 3.13 Example Node Tree After Grow

After the end of simulation, agent continues executing the turn part. It first tries to generate a pi value (using the root node's children's visit counts). In our example that pi value contains *0.33*, *0.33*, and *0.33* (for the colors 1, 2, and 3 respectively). It is computed by dividing the child's *visit count* with root node's *visit count (1/3 = 0.33)*. Then a weighted random selection (with the weights 0.33, 0.33, and 0.33) is done to the actions 1, 2, and 3.

Let’s assume color 1 is selected, *sequence* will then become *sequence = [1, 2, 1]*. Finally, calculated pi values (*0.33, 0.33*, and *0.33*) and state (*sequence = [1, 2, 1]*) are inserted to the *EpisodicData* and the turn ends.

After the end of turn, game part continues. Turn routine will not be executed anymore due to the lack of uncolored vertices in the *sequence* array. Agent will append *EpisodicData* with the terminal state’s utility value (which is *1*). *EpisodicData* now contains only one row with the values of [*0.33, 0.33, 0.33*], *[1, 2, 1],* and *1*. If data derivation is set to *true* then *EpisodicData* will contain two rows. The second row will have values [*0.33, 0.33, 0.33*], *[2, 1, 2],* and *1*. After this, run will end because the solution is found (*sequence = [1, 2, 1]*).

4. IMPLEMENTATION

In this section of the report, used libraries and methods will be examined. However, firstly the brief requirements are shown as follows.

1. Computer with a preferred operating system that supports Pytorch and has Python3 interpreter.
2. Pytorch module for the neural network operations.
3. Pandas, numpy, random, math, and time modules.
4. An undirected graph.
5. A Pytorch neural network model could be given to continuing its reinforcement learning while trying to solve the graph or a fresh model will be created automatically.
6. Tuning the hyper-parameters.

4.1. Hardware Environments

In this project, one can use a computer which runs an operating system that supports Pytorch and Python3 as a hardware environment. If CUDA toolkit is supported by computer’s GPU and operating system, then the neural network will run on the GPU.

Figure 4.1 Nvidia CUDA **[16]**

4.2. Software Environments

Development of this project is achieved in the Jupyter Notebook, which is an interactive editor and Anaconda which is an “AI/ML enablement platform” [17]. In this project, anaconda will contain python3 and Pytorch.

Figure 4.2 Jupyter Notebook **[18]**

4.2.1. Python3 and its Modules

Currently, the best language choice for programming AI/ML applications is Python3. The reason could be its simplicity or modules. Without further ado, this projects one of software requirement is Python3. Python could contain external modules. The important ones, for the machine learning, are pandas and numpy. These modules make manipulation and analysis of data easier with their pre-defined methods.

Moreover, this project also requires some of the python’s internal modules. One of them is the *Random* module, which contains methods like shuffling an array randomly. The second one is the Math module for using square root, logarithm methods. Final internal module's name is Time and will be used for calculating time distributions between the neural network and MCTS.

4.2.2. Pytorch

Another mandatory software requirement is Pytorch. It is an open source deep learning platform which will help to develop a neural network. Furthermore, it is deeply integrated with python and numpy [19]. As being said, if CUDA is usable in the computer, then Pytorch's network will run on the GPU.

Figure 4.3 Pytorch **[19]**

4.3. Graph Coloring Agent's Hyper-Parameters

Graph coloring agent will contain some mandatory hyper-parameters to tune, this part will give a piece of brief information about what they are.

A number that represents how many colors that will be used to color the graph. Number of epoch, learning rate, and batch size will be used in the training the neural network. L2 Regularization coefficient will be used for tuning the Ridge Regression. Turn threshold will be used to balance additional exploration rate. Derive data will be used to create more variance with additional data. Iteration, episode, and simulation count will be used in comparison and normal games.

4.4. Graph Coloring Game’s Agent

In this part, information about the class diagram will be given. This class diagram could be seen in Figure 4.4.

Agent’s tree search structure contains *nodes.* One *node* has a parent node, children node, utility value, probability, current color, and information about its visit count. *MCTS* class controls *node* structure, *NeuralNetwork,* and a sample *ColoredGraph* for simulation purposes which are detailed at section 3.5.

GraphColoringGame is the real game. It contains a *ColoredGraph* structure for playing the game and a *GraphReader* for a reading graph from a file. *ColoredGraphGamePlayer* combines both *GraphColoringGame* and *MCTS* to make them interact with each other. *ColoredGraphGamePlayer* plays graph coloring games, extracts/derives data, and ultimately trains the neural network. *Coach* controls *ColoredGraphGamePlayer*, iteration cycles, and chooses the best (trained) neural network.

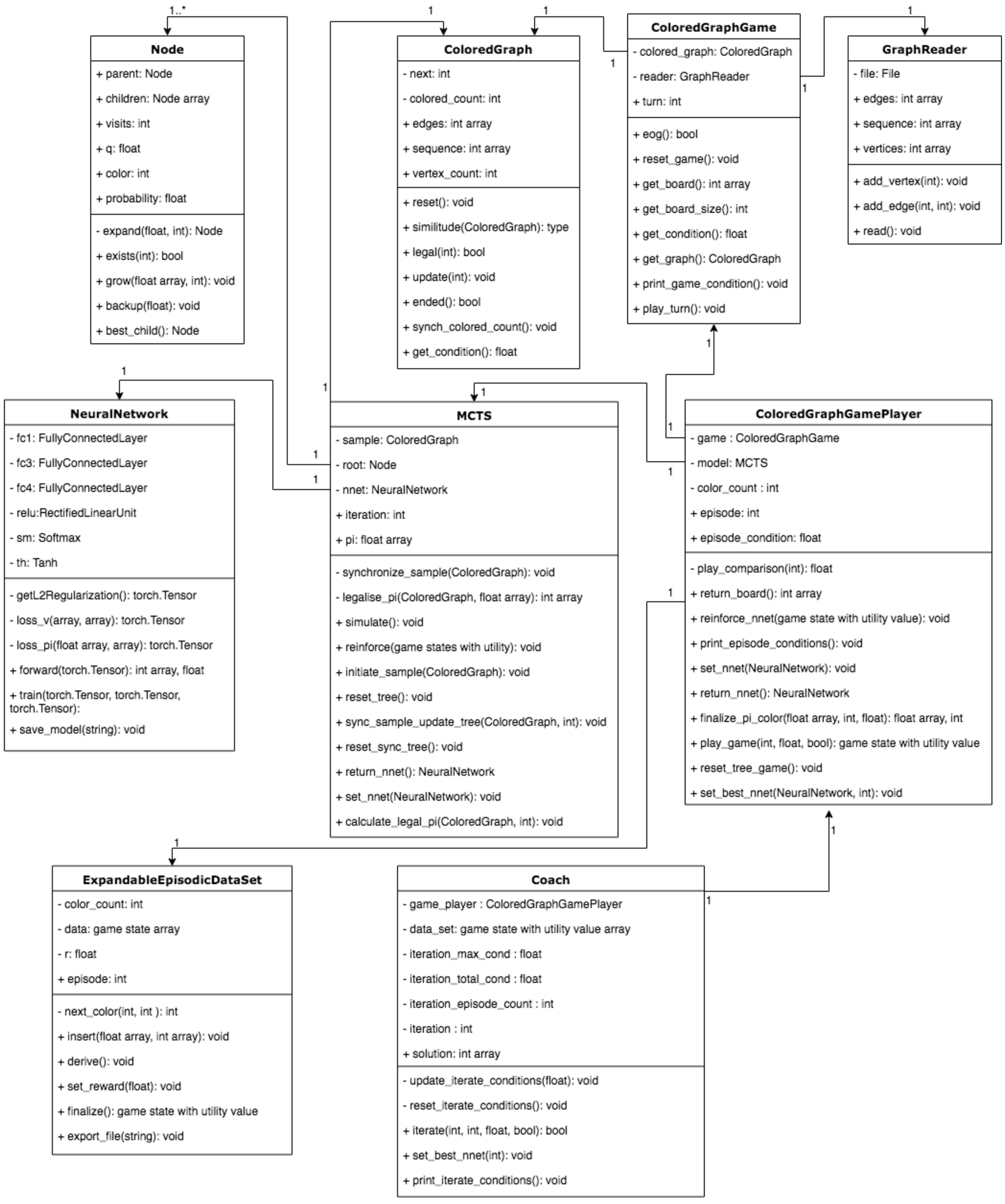

Node
+ parent: Node
+ children: Node array
+ visits: int
+ q: float
+ color: int
+ probability: float
- expand(float, int): Node
+ exists(int): bool
+ grow(float array, int): void
+ backup(float): void
+ best_child(): Node
ColoredGraph
- next: int
- colored_count: int
+ edges: int array
+ sequence: int array
+ vertex_count: int
+ reset(): void
+ similitude(ColoredGraph): type
+ legal(int): bool
+ update(int): void
+ ended(): bool
+ synch_colored_count(): void
+ get_condition(): float
ColoredGraphGame
- colored_graph: ColoredGraph
- reader: GraphReader
+ turn: int
+ eog(): bool
+ reset_game(): void
+ get_board(): int array
+ get_board_size(): int
+ get_condition(): float
+ get_graph(): ColoredGraph
+ print_game_condition(): void
+ play_turn(): void
GraphReader
- file: File
+ edges: int array
+ sequence: int array
+ vertices: int array
+ add_vertex(int): void
+ add_edge(int, int): void
+ read(): void
NeuralNetwork
- fc1: FullyConnectedLayer
- fc3: FullyConnectedLayer
- fc4: FullyConnectedLayer
- relu:RectifiedLinearUnit
- sm: Softmax
- th: Tanh
- getL2Regularization(): torch.Tensor
- loss_v(array, array): torch.Tensor
- loss_pi(float array, array): torch.Tensor
+ forward(torch.Tensor): int array, float
+ train(torch.Tensor, torch.Tensor, torch.Tensor):
+ save_model(string): void
MCTS
- sample: ColoredGraph
- root: Node
- nnet: NeuralNetwork
+ iteration: int
+ pi: float array
- synchronize_sample(ColoredGraph): void
- legalise_pi(ColoredGraph, float array): int array
+ simulate(): void
+ reinforce(game states with utility): void
+ initiate_sample(ColoredGraph): void
+ reset_tree(): void
+ sync_sample_update_tree(ColoredGraph, int): void
+ reset_sync_tree(): void
+ return_nnet(): NeuralNetwork
+ set_nnet(NeuralNetwork): void
+ calculate_legal_pi(ColoredGraph, int): void
ColoredGraphGamePlayer
- game : ColoredGraphGame
- model: MCTS
- color_count : int
+ episode: int
+ episode_condition: float
- play_comparison(int): float
+ return_board(): int array
+ reinforce_nnet(game state with utility value): void
+ print_episode_conditions(): void
+ set_nnet(NeuralNetwork): void
+ return_nnet(): NeuralNetwork
+ finalize_pi_color(float array, int, float): float array, int
+ play_game(int, float, bool): game state with utility value
+ reset_tree_game(): void
+ set_best_nnet(NeuralNetwork, int): void
ExpandableEpisodicDataSet
- color_count: int
- data: game state array
- r: float
+ episode: int
- next_color(int, int): int
+ insert(float array, int array): void
+ derive(): void
+ set_reward(float): void
+ finalize(): game state with utility value
+ export_file(string): void
Coach
- game_player : ColoredGraphGamePlayer
- data_set: game state with utility value array
- iteration_max_cond : float
- iteration_total_cond : float
- iteration_episode_count : int
- iteration : int
+ solution: int array
- update_iterate_conditions(float): void
- reset_iterate_conditions(): void
+ iterate(int, int, float, bool): bool
+ set_best_nnet(int): void
+ print_iterate_conditions(): void
1..*
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1

Figure 4.4 Class Diagram of Graph Coloring Game and Agent

5.TEST AND RESULTS

In this section, tests that have been carried out on different graphs and the results obtained are being analyzed. Also, the software and hardware specifications of the tested computer will be presented.

Specification of hardware is a custom PC with an Intel Core i7-4790K CPU @ 4.00 GHz x 8, 8 GB RAM and a GeForce GTX 970. Specification of software is Ubuntu 16.04 LTS 64-bit.

As being said, the tests are made on some selected graphs. These graphs are “queen6_6.col.b”, “queen7_7.col.b”. Note that, diamond symbols in the figures represents the selection of new neural network instead of the old one. In data loss graphs, y axis represents pi and v losses while x axis represents total epochs. Similarly, in colored count graphs, y axis represents average and maximum colored counts while x axis represents iteration.

5.1. Queen6_6 Graph with Derived Data

This graph has 36 vertices and 290 edges. The best-known color count is 7 [20]. When MCTS is used with only the initial neural network (random parameters, without any training), the problem is successfully solved in 11 out of 20 runs. On the other side, MCTS with a reinforcing neural network can solve the problem in 20 out of 20 runs. Detailed information about the test of this MCTS with a reinforcing neural network will be given in the following paragraphs.

Agent’s selected hyper-parameters are: “*Color count* = is 7”, “*Number of iterations in real game* = 100”, “*Number of episodes in real game* = 40”, “*Number of simulations in real game* = 40”, “*Number of simulations in comparison game* = 20”, “*L2 coefficient* = 0.5”, “*Number epochs* = 20”, “*Batch size* = 1”, “*Learning rate* = 0.001”, “*Turn threshold* = 0.7” and “*Derive data* = true”. Details of these hyper-parameters are given at the sections 3.5 and 4.3.

As being said, the graph coloring agent has run 20 times where each run contains 100 iterations. The total elapsed time of this test is nearly 1881 minutes where MCTS simulations took 63, neural network's training took 1811, and other operations (deriving data, taking the data from CPU to GPU and back, etc…) took 7 minutes.

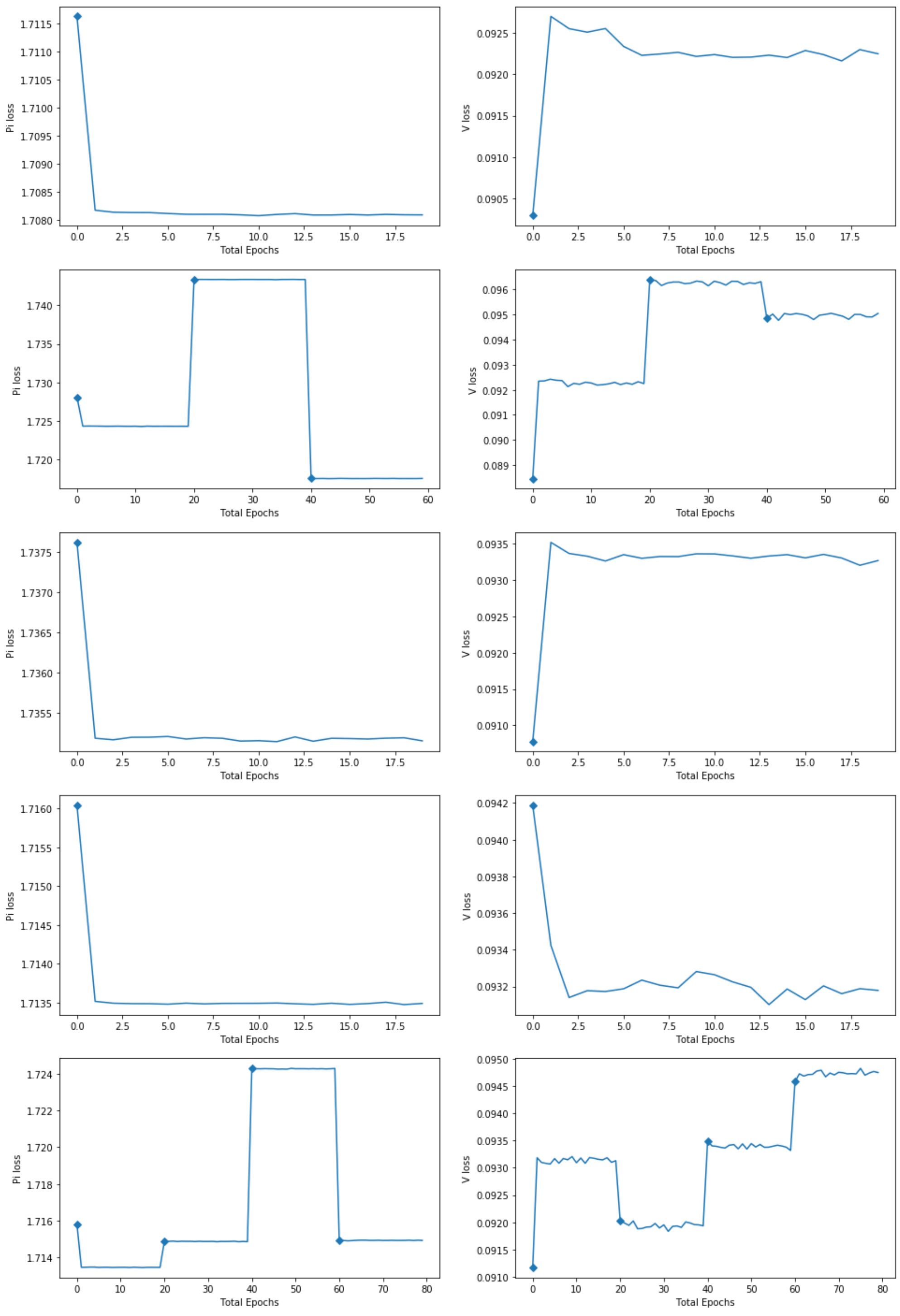


Figure 5.1 Queen 6 with Derived Data Losses – 1

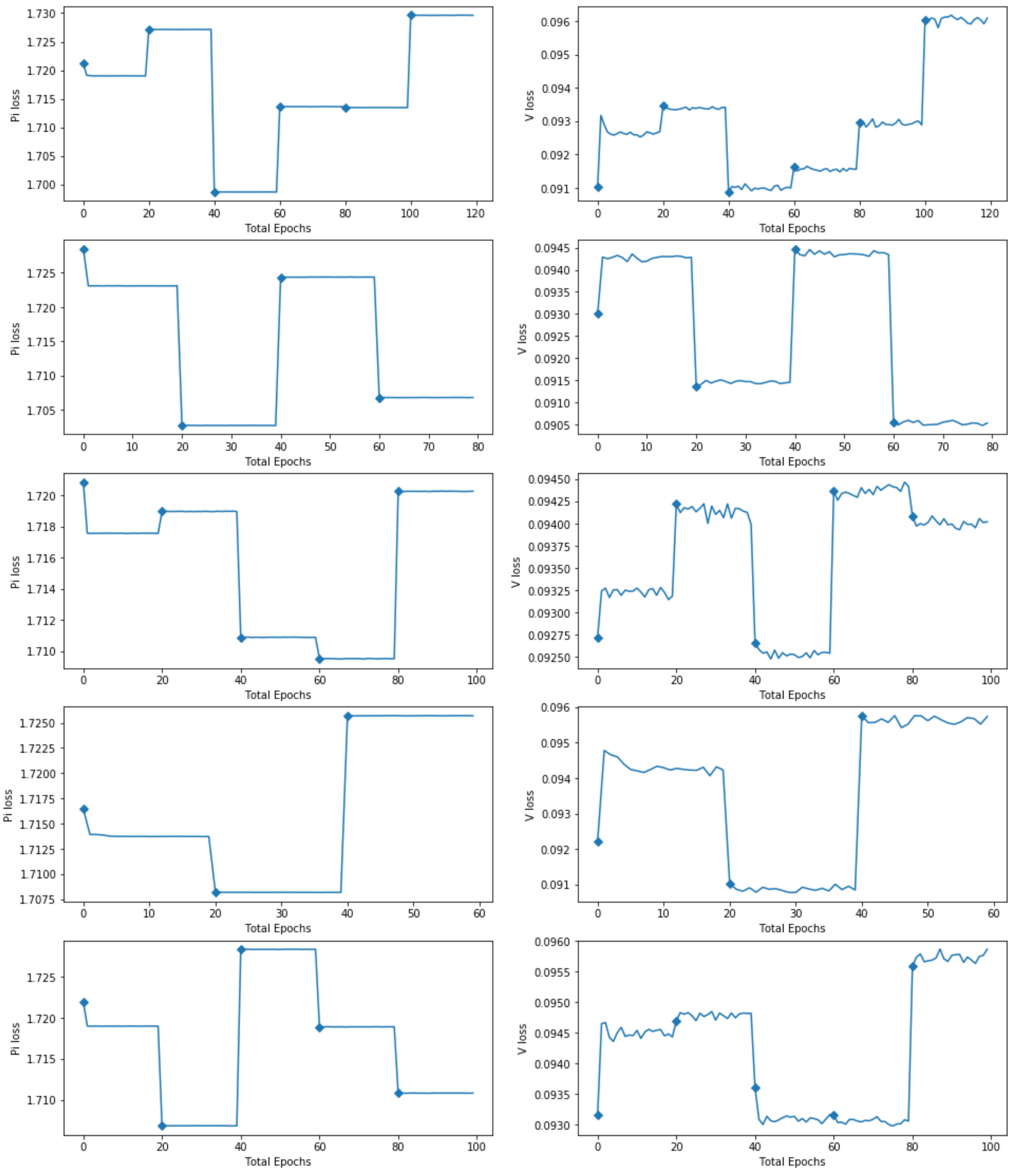


Figure 5.2 Queen 6 with Derived Data Losses – 2

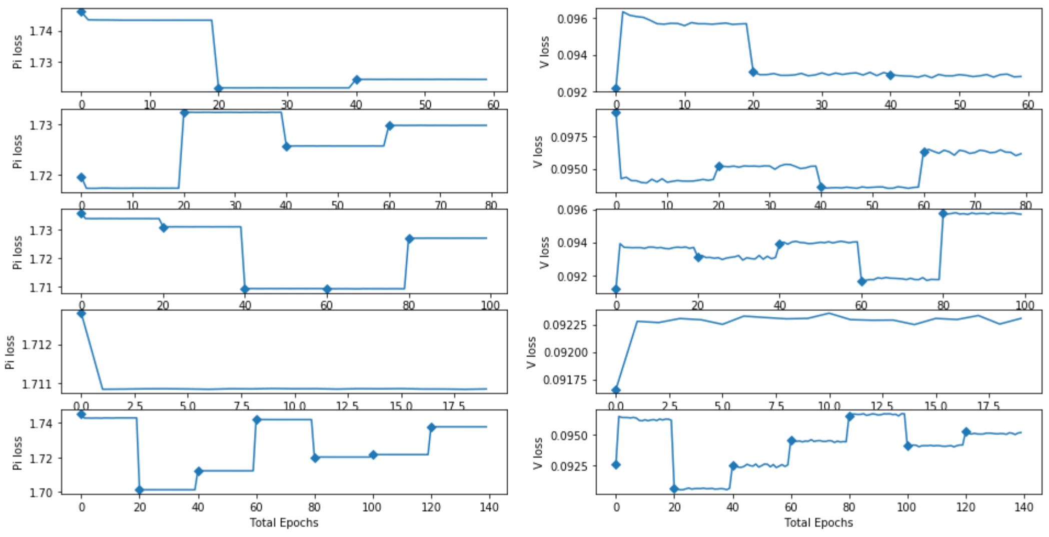


Figure 5.3 Queen 6 with Derived Data Losses – 3

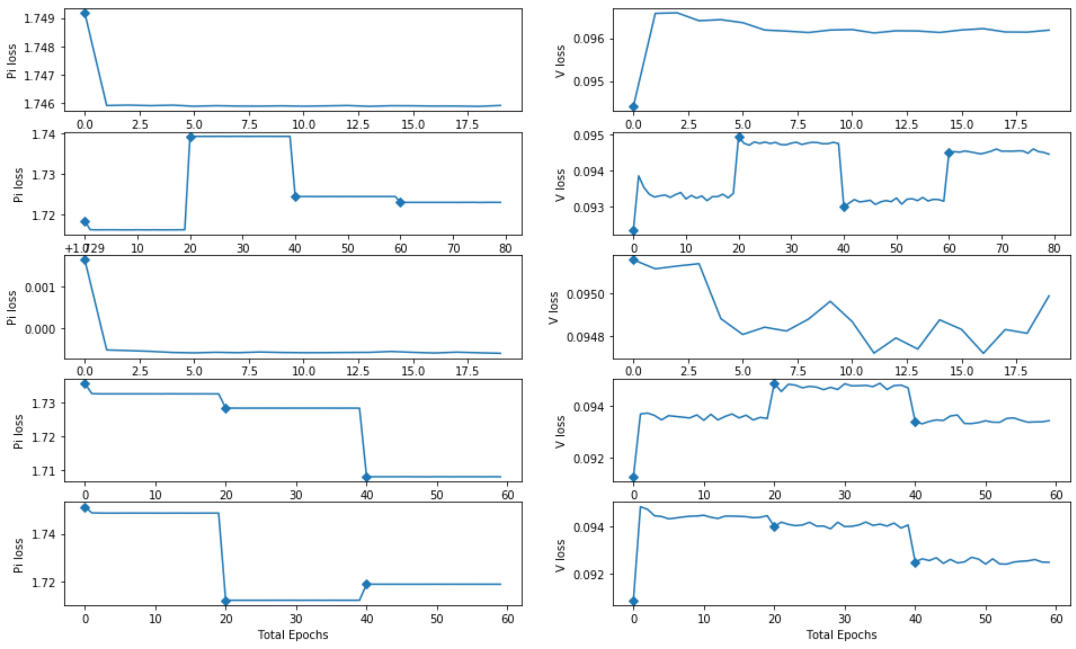


Figure 5.4 Queen 6 with Derived Data Losses – 4

In Figure 5.1, Figure 5.2, Figure 5.3, and Figure 5.4 π and utility value's loss results are shown. It should be noted that in each iteration, neural network trains with an epoch equal to 20. Also, on some iterations, trained neural networks are not selected and thus their losses are not shown for the sake of simplicity. As it can be seen in the figures, sometimes these loss values increase, that's because MCTS explores new possibilities and neural network tries to predict these unknown possibilities. Eventually, neural network learns these new possibilities.

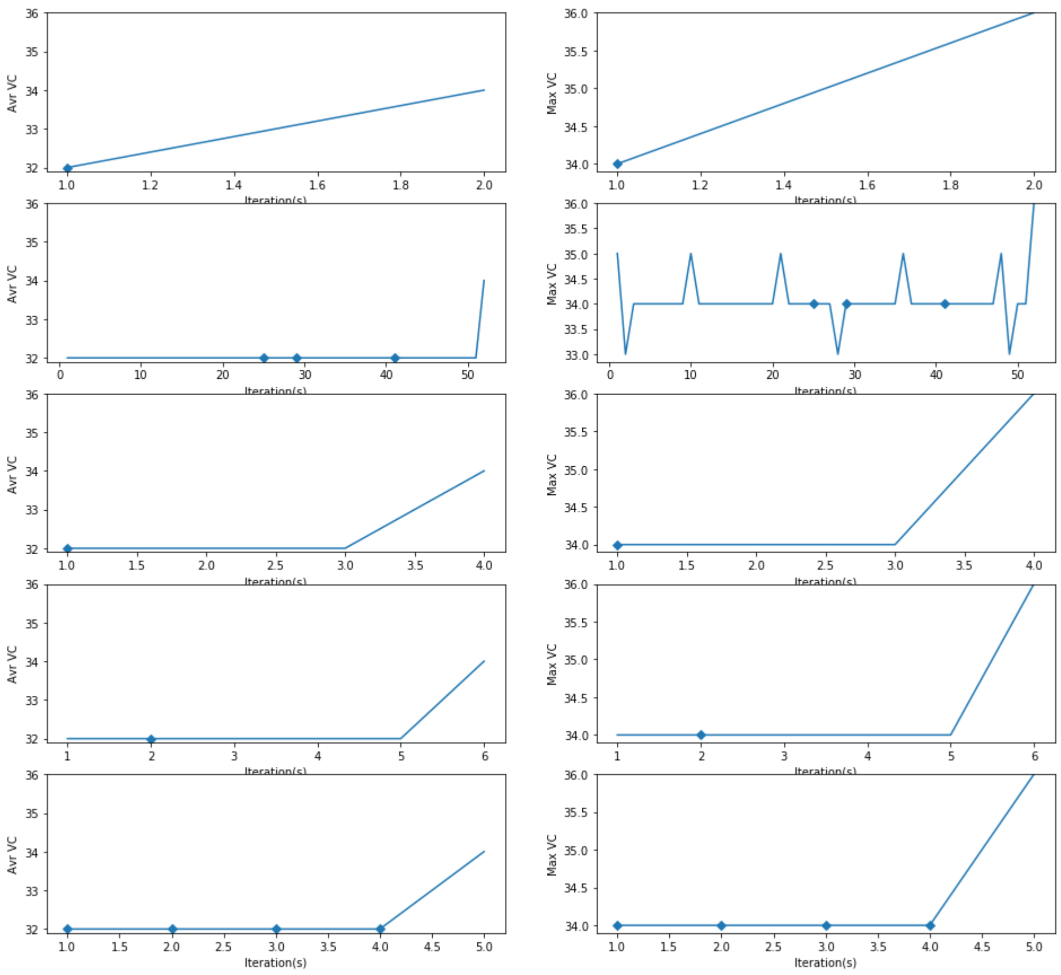


Figure 5.5 Queen 6 with Derived Data Colored Counts – 1

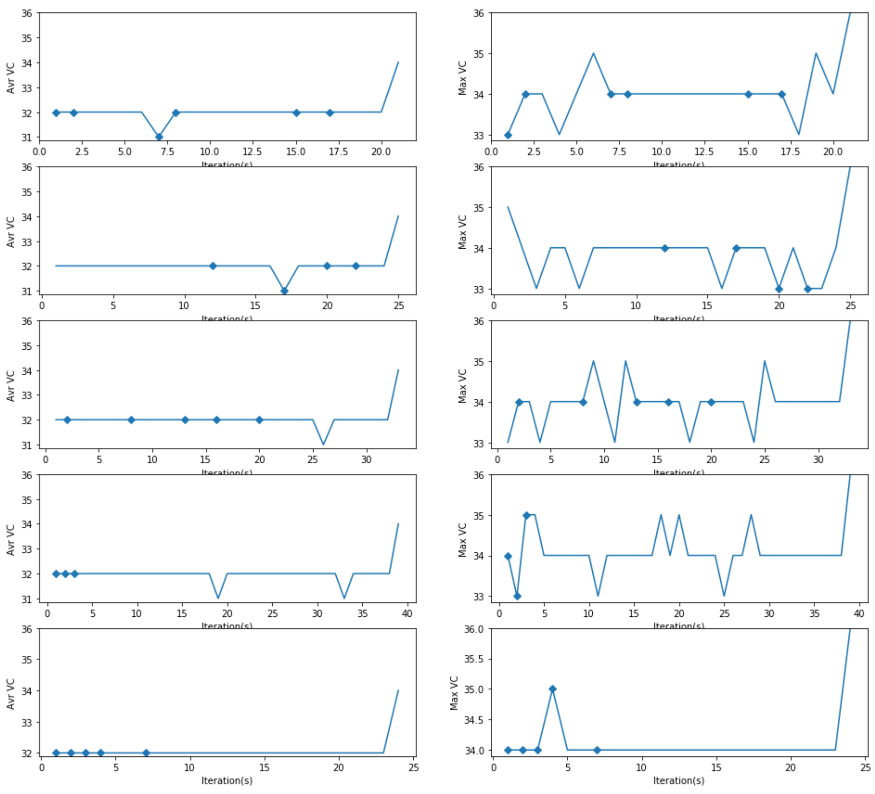


Figure 5.6 Queen 6 with Derived Data Colored Counts – 2

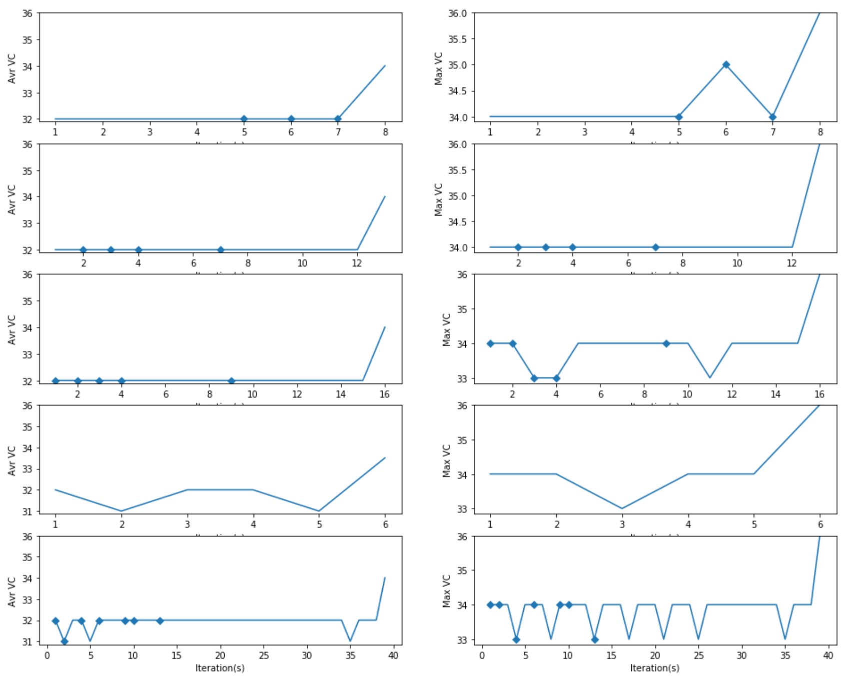


Figure 5.7 Queen 6 with Derived Data Colored Counts – 3

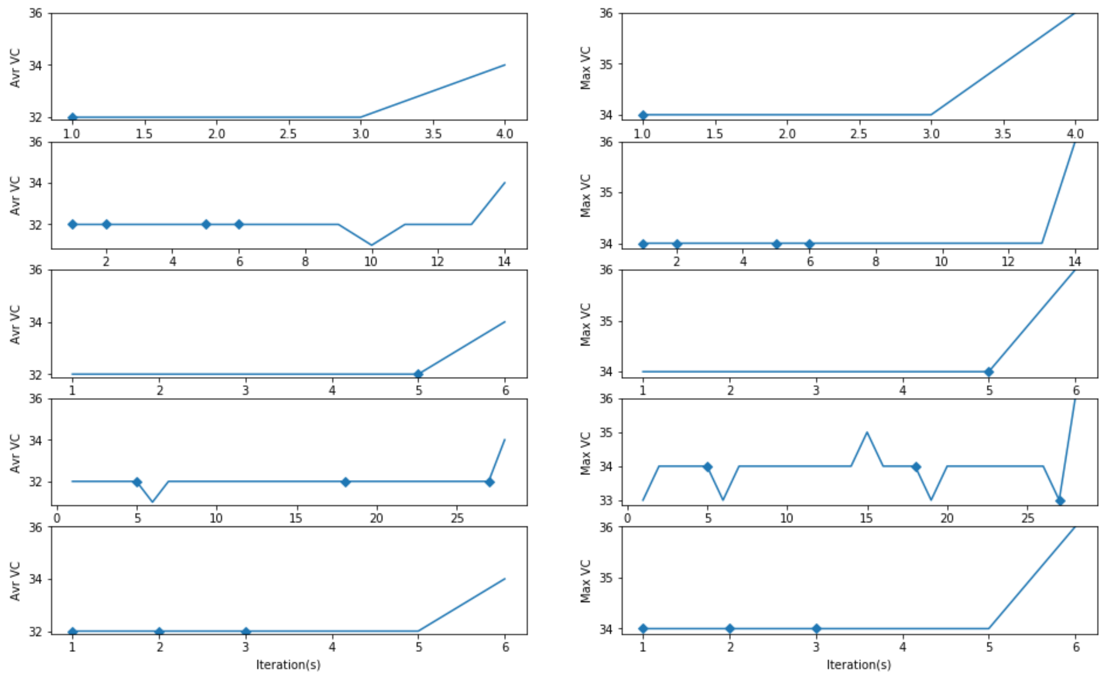


Figure 5.8 Queen 6 with Derived Data Colored Counts – 4

In Figure 5.5, Figure 5.6, Figure 5.7, and Figure 5.8 one can see that graph coloring agent is learning newly exploited data to find a result. Average iteration (of every run) takes for a solution to be found is “17.55”.

5.2. Queen6_6 Graph without Derived Data

Differences between this test’s hyper-parameters and the ones in section 5.1 are *derived data* is set to “false” and *Number epochs* is set to “5” thus this test has taken 139.7 minutes where neural network’s training took 78.6 minutes, MCTS simulations took 55.5 minutes, and the other operations took 5.6 minutes. Without a reinforcing neural network, 10/20 runs are successful. In reinforcing neural net, 20/20 runs are successful.

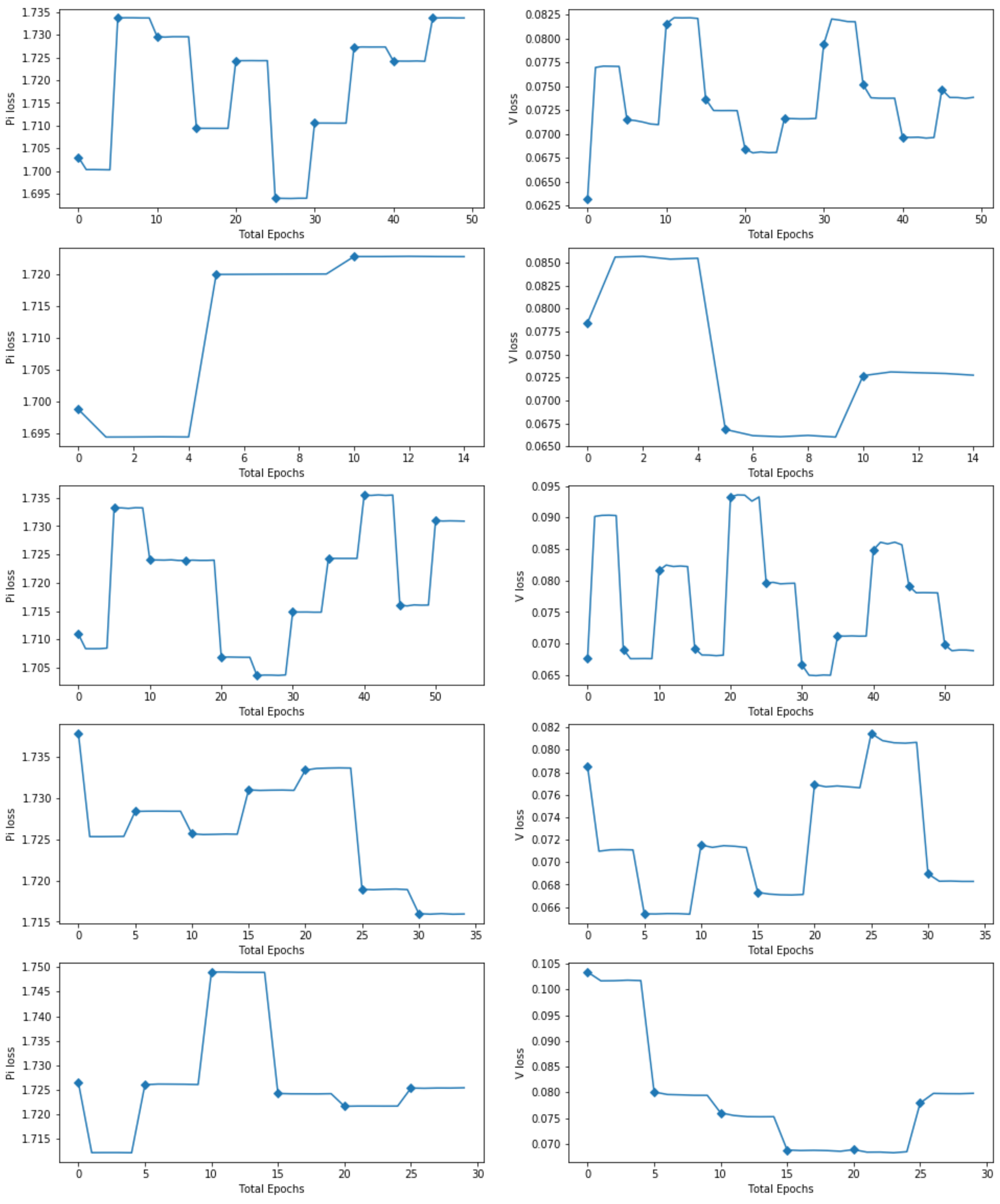


Figure 5.9 Queen 6 without Derived Data Losses – 1

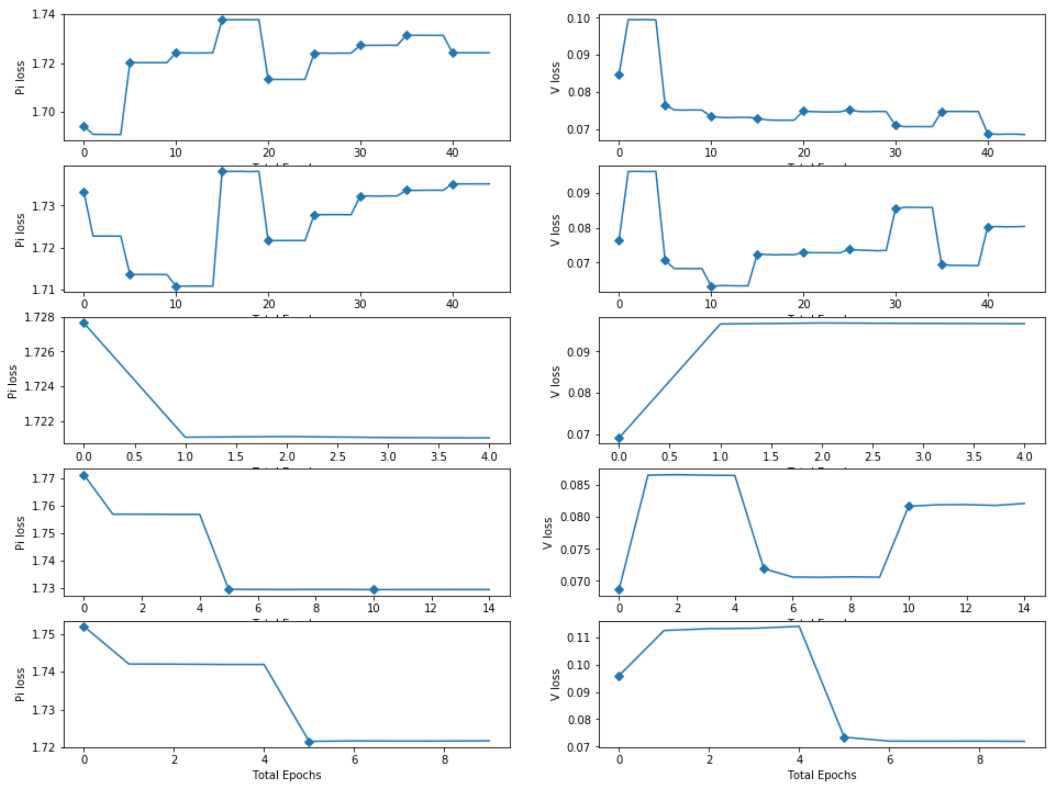


Figure 5.10 Queen 6 without Derived Data Losses – 2

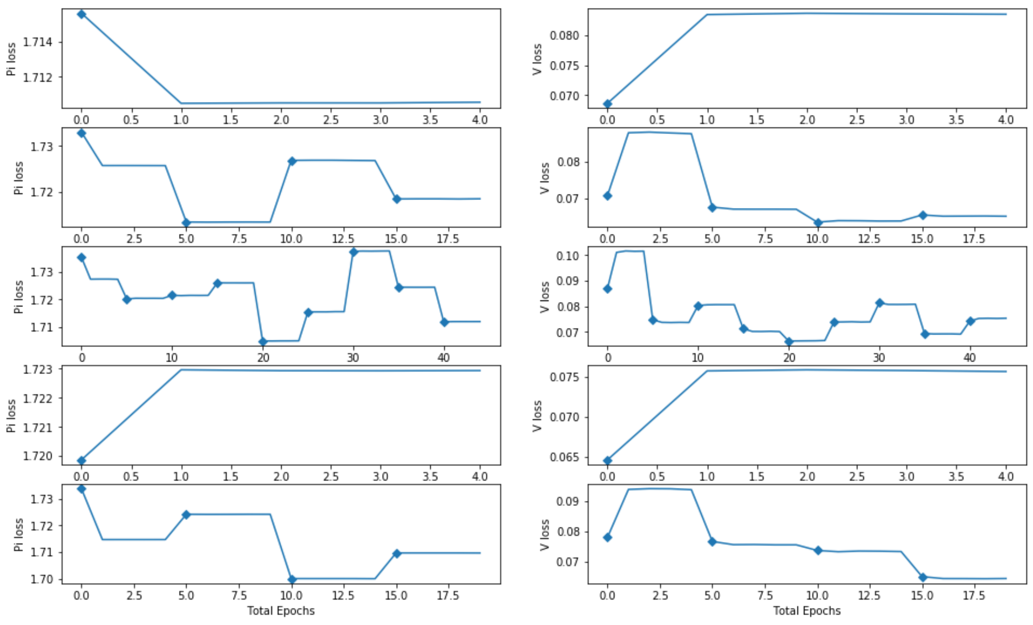


Figure 5.11 Queen 6 without Derived Data Losses – 3

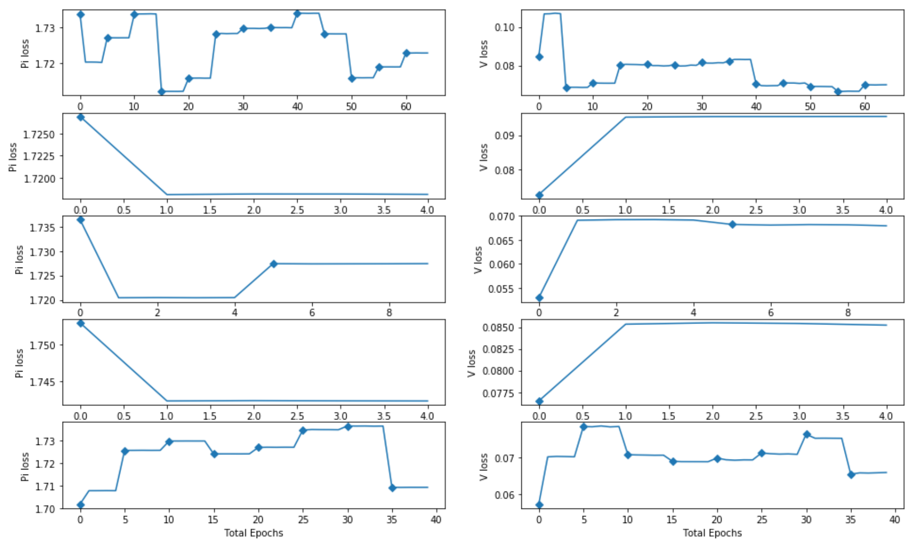


Figure 5.12 Queen 6 without Derived Data Losses – 4

As can be seen in Figure 5.9, Figure 5.10, Figure 5.11, and Figure 5.12 many new neural networks are created when a lower number of an epoch is selected. In larger epoch size (like the one in 5.1), trained neural network sometimes over-fitted and eventually lost the comparison games.

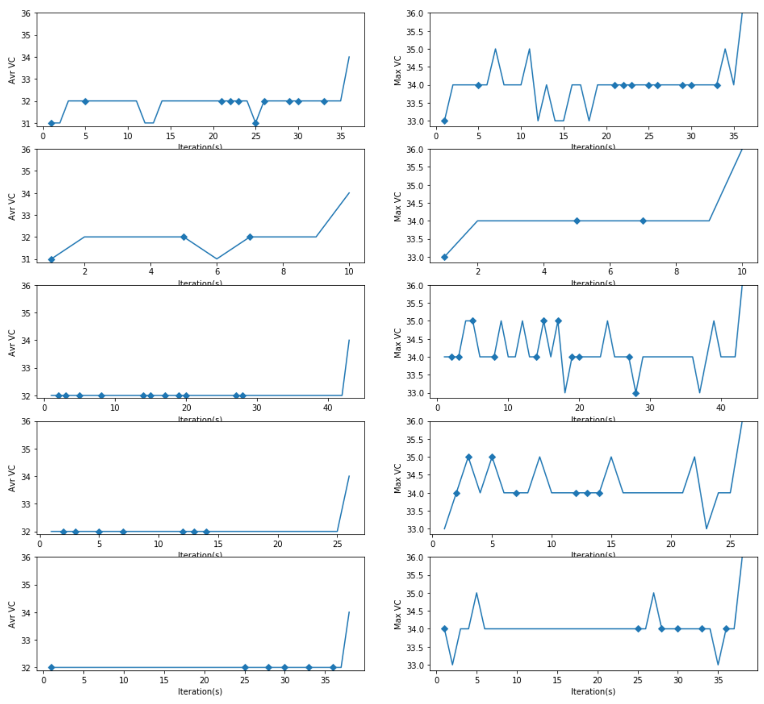


Figure 5.13 Queen 6 without Derived Data Colored Counts – 1

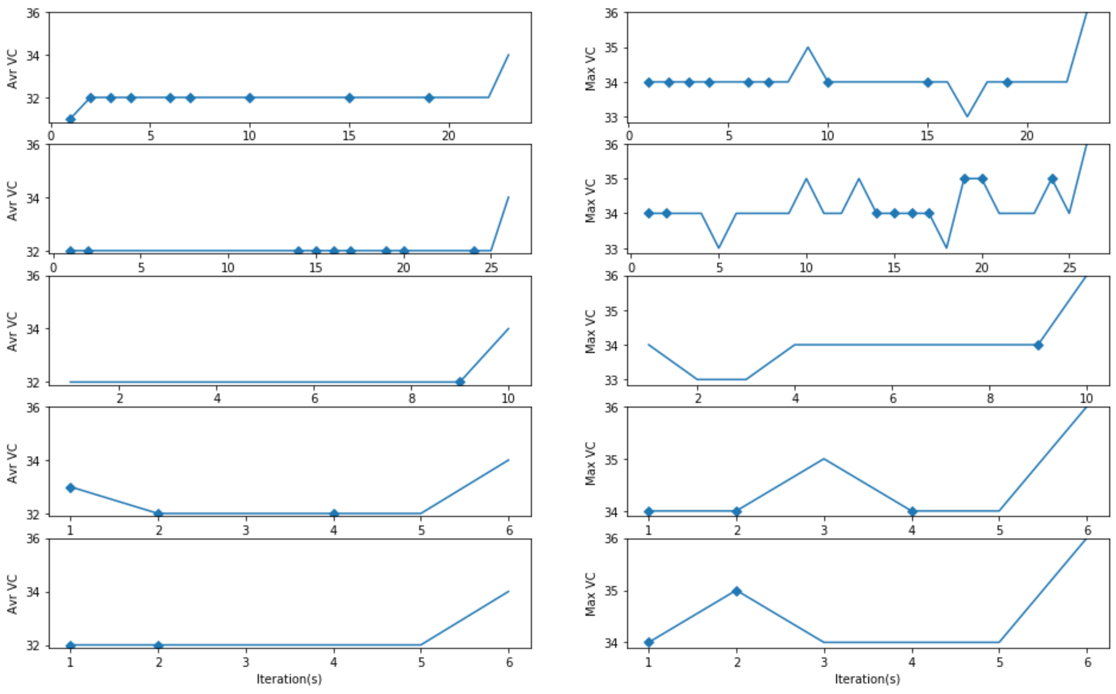


Figure 5.14 Queen 6 without Derived Data Colored Counts – 2

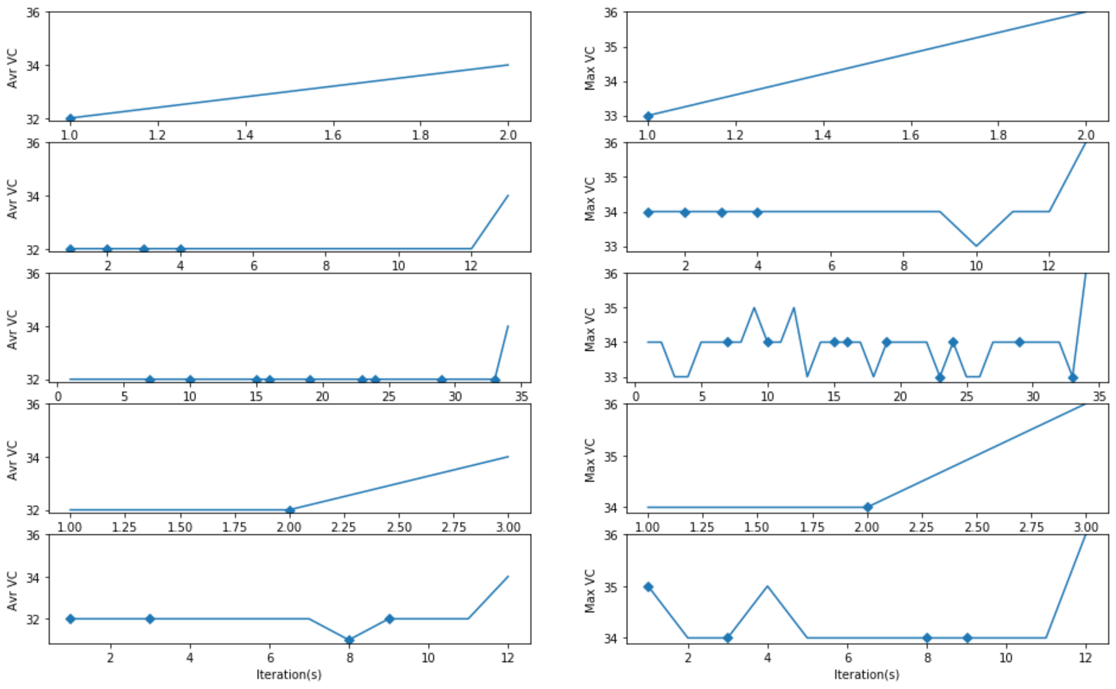


Figure 5.15 Queen 6 without Derived Data Colored Counts – 3

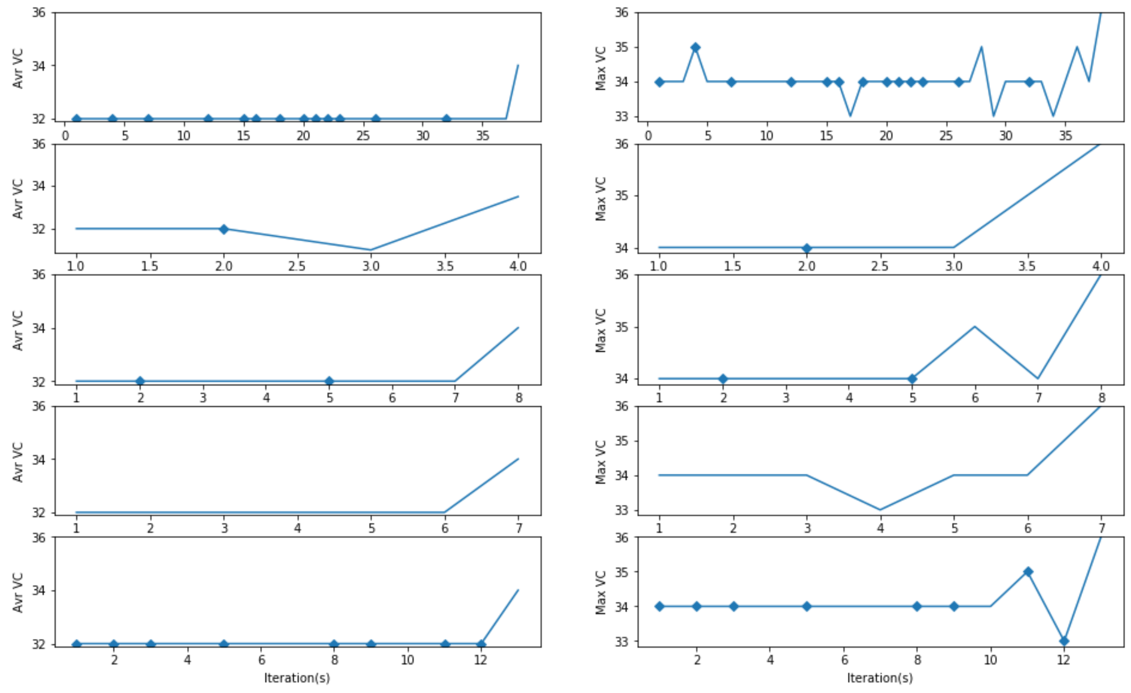


Figure 5.16 Queen 6 without Derived Data Colored Counts – 4

In Figure 5.13, Figure 5.14, Figure 5.15, and Figure 5.16 one can see that graph coloring agent can find solutions without deriving data. Furthermore, it takes a much lower run time. Average iteration (of every run) takes for a solution to be found is “17.9” which is higher than the section 5.1’s test. This is happened due to closing data derivation and lowering the epoch size.

5.3. Queen 7

This graph has 49 vertices and 476 edges. The best-known color count is 7 [20]. Again, MCTS using the only initial neural network, successfully solved the problem in 9 out of 20 runs. MCTS with a reinforcing neural network solved the problem in 20 out of 20 runs. Detailed information about the test of this MCTS with a reinforcing neural network will be given in the following paragraphs.

Agent’s selected hyper-parameters are: “*Color count* = is 7”, “*Number of iterations in real game* = 100”, “*Number of episodes in real game* = 40”, “*Number of simulations in real game* = 40”, “*Number of simulations in comparison game* = 40”, “*L2 coefficient* = 0.5”, “*Number epochs* = 20”, “*Batch size* = 1”, “*Learning rate* = 0.001”, “*Turn threshold* = 0.7” and “*Derive data* = false”. This test took a total of 112.62 minutes where neural network training took 59.97, MCTS simulations took 47.10 and other operations took 5.55 minutes.

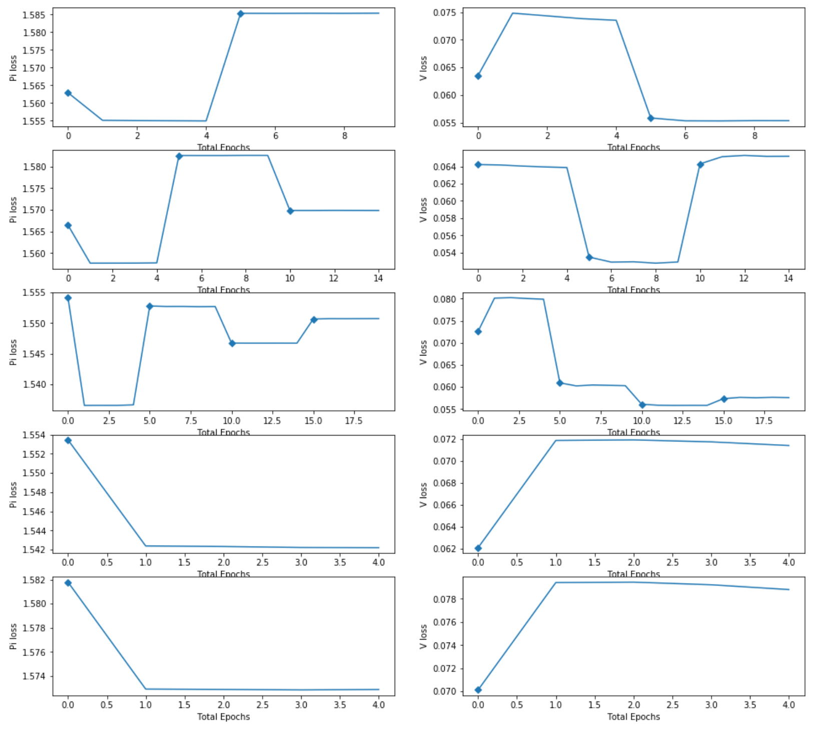


Figure 5.17 Queen 7 Losses - 1

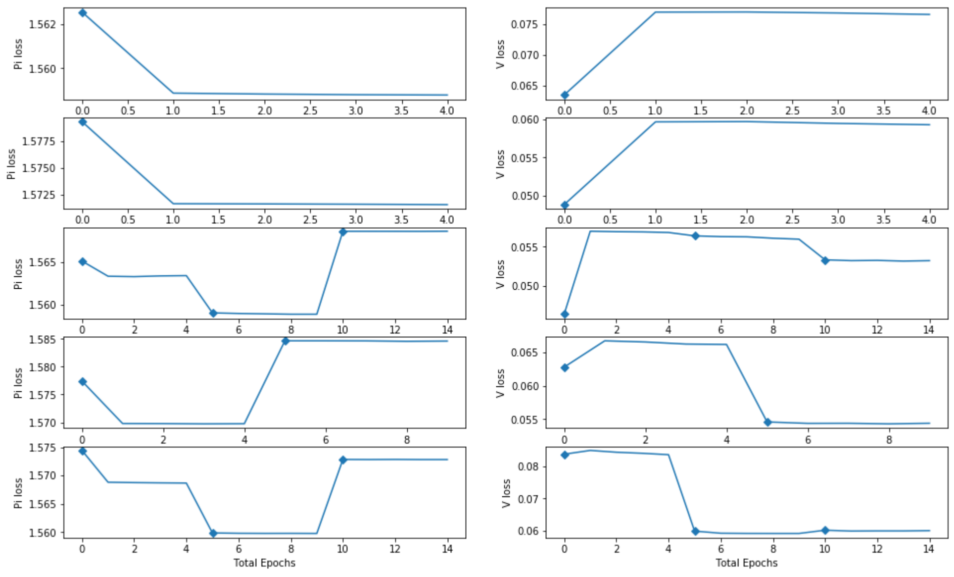


Figure 5.18 Queen 7 Losses – 2

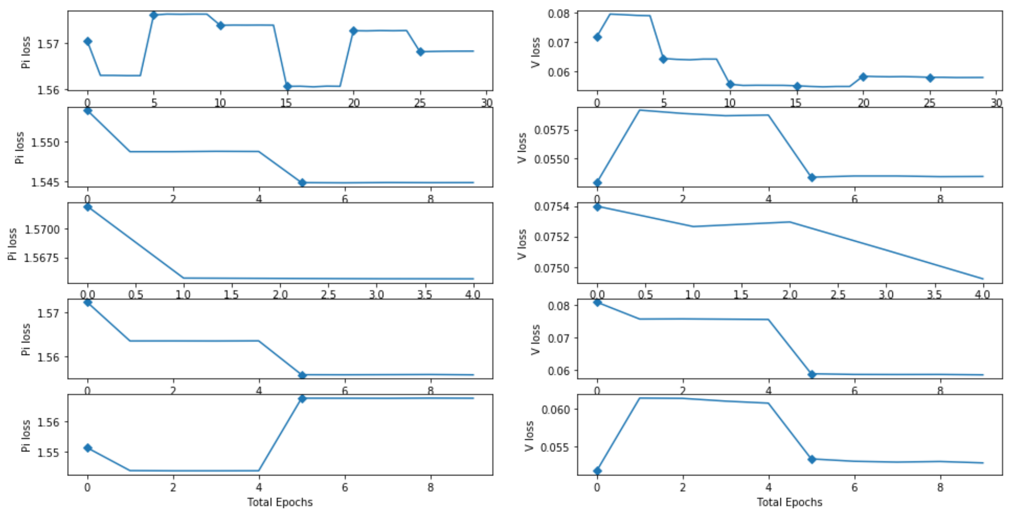


Figure 5.19 Queen 7 Losses – 3

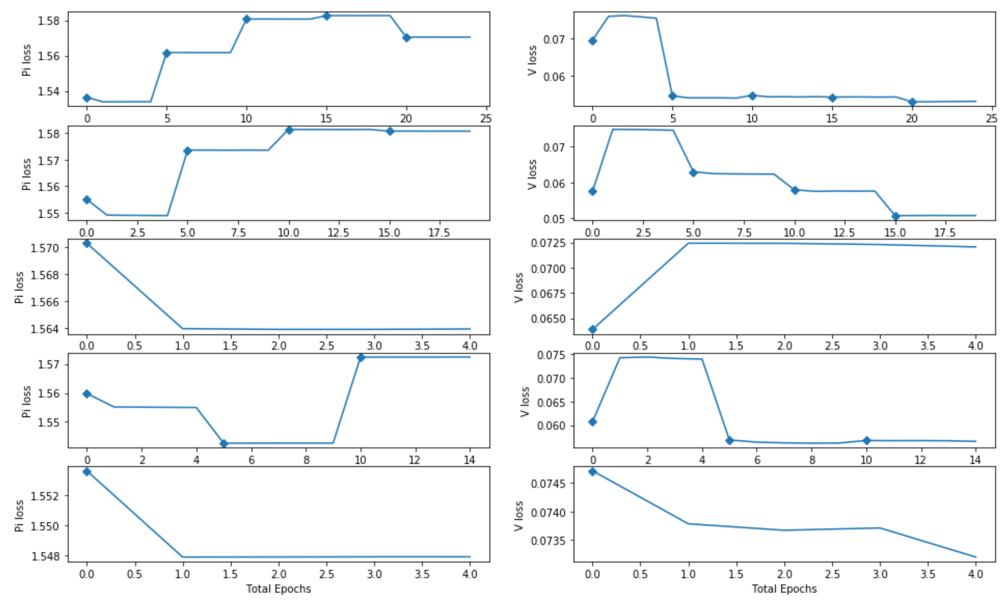


Figure 5.20 Queen 7 Losses – 4

In Figure 5.17, Figure 5.18, Figure 5.19, and Figure 5.20 it is clear that losses are lower than the relatively easier "*Graph 6*". Agent have a more descending learning curve (excluding newly explored possibilities). Furthermore, Figure 5.21, Figure 5.22, Figure 5.23, and Figure 5.24 show that agent has surpassed every challenge it saw and solved all of Graph Coloring 7 runs. Average iteration count for a solution to be found is "10.9". It is even lower than the respectively easier Graph 6's iteration count average. Reason it that the simulation count on comparison is the same as with simulation count at a real game. This affects the results considerably. Because the neural network learns the information of a specific simulation count. If the comparison game's simulation count is smaller, then it might lose unnecessarily.

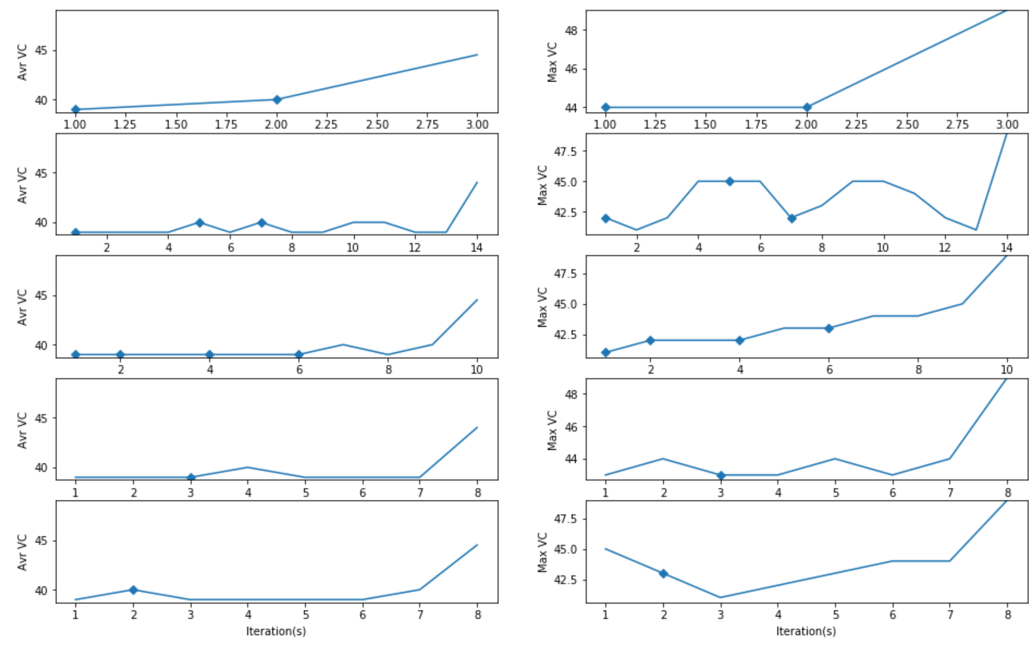


Figure 5.21 Queen 7 Color Counts - 1

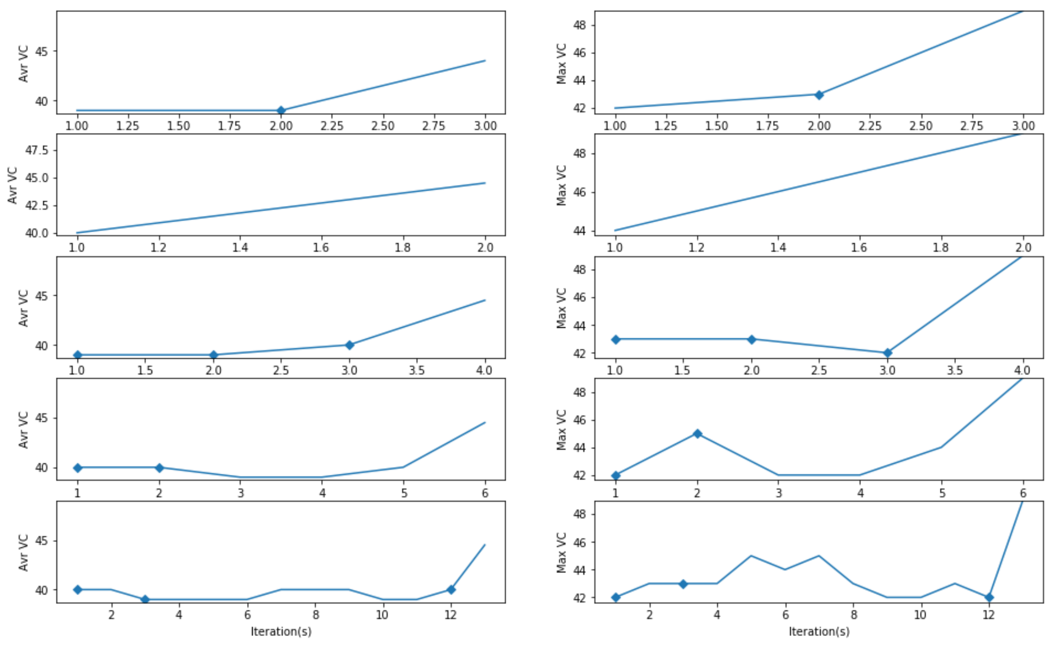


Figure 5.22 Queen 7 Color Counts - 2

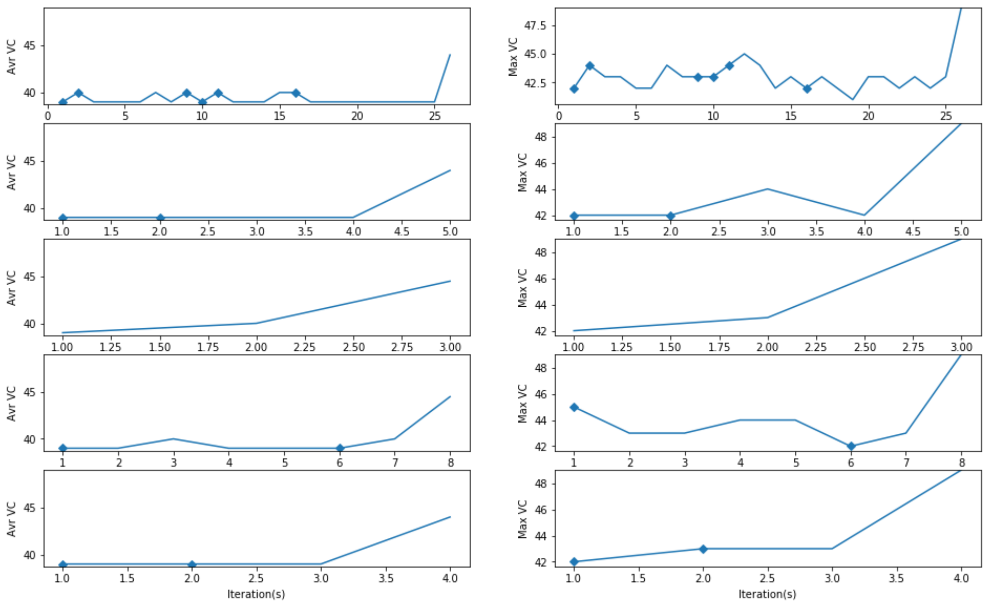


Figure 5.23 Queen 7 Color Counts – 3

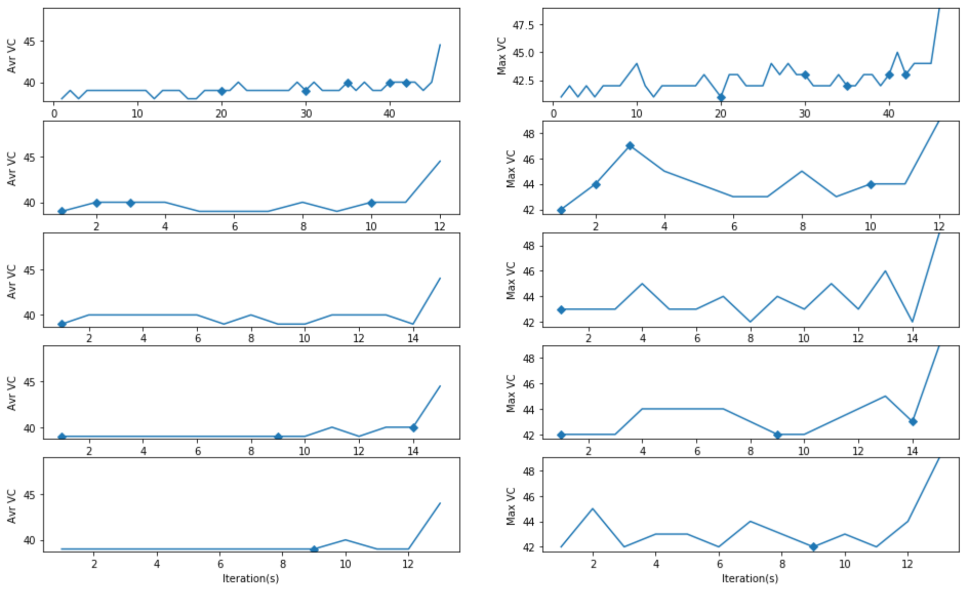


Figure 5.24 Queen 7 Color Counts – 4

In summary, agent accomplished self-learning task and successfully solved some graphs (queen6_6 and queen7_7). However, it spent longer time compared to traditional graph coloring algorithms. Some other tests are done for queen8_8 and more complex graphs. Their solutions were not found within a certain time constraint. Agent may be improved by increasing it’s time constraint or using multithreading.

6. CONCLUSION AND FUTURE WORK

The purpose of the project is creating a graph coloring agent that can solve graph coloring problems by creating its own experience from *tabula rasa*. However, within a given computational budget, the agent was not able to solve relatively complex graphs.

The foremost objective should be optimizing the agent to shorten its iteration time. Optimization could be acquired by creating multiple colored graph game players to play parallelized games and their experience can be used to train a single neural network. If parallelization is not sufficient, some hardware improvements could be made. Because without shortening the iteration time, tests about finding the optimal hyper-parameters and about the effects of any improvements could not be carried out within a reasonable time.

Comparison operation of the two neural networks could be done by playing multiple comparison games (instead of one). For each comparison game, simulation count could be increased. When these comparison games end, neural network with the best averaged results could be selected. This will show which neural network performs better when simulation count is increased.

When these objectives are completed, future work could split into two paths, trying to make this agent work on bigger graphs or letting the agent extract knowledge on different graphs to solve an unseen graph.

References

[1] K. Ruohonen, Graph Theory (English Edition), Keijo Ruohonen, 2013.

[2] D. Q. Nykamp, "Undirected graph definition," Math Insight, [Online]. Available: http://mathinsight.org/definition/undirected_graph.

[3] P. Cintia, M. Coscia and L. Pappalardo, "The Haka Network: Evaluating Rugby Team Performance with Dynamic Graph Analysis," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, San Francisco, 2016.

[4] S. Malerud, "Modelling human social behaviour in conflict environments using complex adaptive systems.," Norwegian Defence Research Establishment, 2008.

[5] S. P. N. J.A. Sloane, The Encyclopedia of Integer Sequences, 1995.

[6] D. Guichard, Combinatorics and Graph Theory, San Francisco.

[7] A. Parmar, "A Study of Graph Coloring," Gujarat, 2015.

[8] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Global Edition, Pearson, 2016.

[9] C. B. Browne, E. Powley, D. Whitehous E., S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 4, no. 1, pp. 1-43, 2012.

[10] A. Géron, Hands-On Machine Learning with Scikit-Learn and TensorFlow, O'Reilly Media, 2017.

[11] O. S. Eluyode and D. T. Akomolafe, "Comparative study of biological and artificial neural networks," *European Journal of Applied Engineering and Scientific Research,* vol. 2, no. 1, pp. 36-46, 2013.

[12] N. Kumar, G. Upadhyay and P. Kumar, "Comparative Performance of Multiple Linear Regression and Artificial Neural Network Based Models in Estimation of Evaporation," *Advances in Research,* vol. 11, no. 5, pp. 1-11, 2017.

[13] S. D., S. J., S. K., A. I., H. A., G. A., H. T., B. L., L. M., B. A. and C. Y., "Mastering the game of Go without human knowledge," *Nature,* vol. 550, no. 7676, p. 354–359, 2017.

[14] . N. Metropolis and S. Ulam, "The Monte Carlo Method," *Journal of the American Statistical Association,* vol. 44, no. 247, pp. 335-341, 1949.

[15] C. Pease, "An Overview of Monte Carlo Methods - Towards Data Science," Medium, 6 September 2018. [Online]. Available: https://towardsdatascience.com/an-overview-of-monte-carlo-methods-675384eb1694.

[16] "Scicomp Accelerates Market Leading Derivative Pricing Software With NVIDIA CUDA," Nvidia, 16 SEPTEMBER 2008. [Online]. Available: https://www.nvidia.com/object/io_1221568471314.html.

[17] "What is Anaconda," Anaconda, 2018 . [Online]. Available: https://www.anaconda.com/what-is-anaconda/.

[18] "Project Jupyter," Project Jupyter, 03 December 2018. [Online]. Available: https://jupyter.org.

[19] "PyTorch," 2018. [Online]. Available: https://pytorch.org.

[20] N. B. Thang, H. N. ThanhVu, M. P. Chirag and T. P. Kim-Anh, "An ant-based algorithm for coloring graphs," *Discrete Applied Mathematics*, vol. 156, no. 2, pp. 190-200, 2008.