Neural Network for Bisimulation

Chao Xiang

April 2019

Abstract

Neural Network is widely used in many fields. And it has shown the ability to solve complex problems. In addition, bisimulation is a very important logical concept. This project is aiming to explore the capability of the neural network to understand this concept. The project is divided into three parts. In the first part, standard bisimulation algorithm was developed. And one test case generator that can generate random graphs pairs with bisimulation flag is also developed base on the standard bisimulation. In the second part, a simple neural network was developed to recognise bisimulation. In the third part, a wide experiment and a deep experiment were done. The wide experiment tests the performance of the neural network on the graphs of different density. The result shows that the network performance well and show similarity in most of the cases. It indicates the representativeness of the test cases produced by the developed random test case generator. The deep experiment explores the limitation of the network by training the network in the large scale graphs pairs. The result shows a good ability to understanding the bisimulation. However, for some larger scale graphs pairs, the network shows an unusually good performance. It is speculated coursed by the small size of data set compare with the massive amount of possible graph pairs. And more experiments still needed.

Contents

1 Introduction

Artificial Neural Networks (ANNs) has been widely used in multiple areas. It has been proved that ANN is Turing complete [1]. It indicates that understanding the logic (i.e. symbolic computation) is possible theoretically, gives broad application prospects. There are ANNs [2, 3] which are able to mimic some first-order logic program. However, these solutions were based on the Recurrent Neural Network (RNN) and tried to simulate the logic program rather than catch the logic feature. Trying to explore the limitation of ANNs for understanding logical concepts, will help the research of neuro-symbolic integration.

Thus, here we are concerned with the capability of different Multilayer Perceptrons (MLPs) to understand the logic concept (i.e. bisimulation of logic structures). And MLPs here is introduced as the representative of ANNs, as it is recognised as the original and most basic model of ANNs. Datasets used to train the MLPs is consisted of groups of generated graphs with flags (i.e. a number that indicates if two graphs are bisimulation). Then test if MLPs are able to learn the ability to distinguish the bisimulation equivalence. The report is structured as follows. First, in Section 2, relevant research of bisimulation and relation between neural network and logic will be given. The specification of the problem and the approach used to explore the limitation will be explained in Section 3. Then in Section 4, the process of realisation including component test and the process of the project will be stated. Section 5 will discuss the entire process of the experiments including the setting up, impelementation and analysation of result. And Section 6, the summary and the evaluation from the aspect of engineer and research will be explained. Knowledge and skills gained will be talked in Section 7. The professional issue will be discussed in Section 8. At last the full code of the project, part of the data, screenshots of sample runs, user manual, progress log and Gantt chart will be appended in the Section A.

2 Background

This section recalls some notions used in the project with relevant researches. The notion of *Bisimulation* is introduced as the target logic feature that needs to be learned. The notion of *Artificial Neural Networks* is introduced as the object that studied in the project.

2.1 Bisimulation

Bisimulation is an important concept. It has been introduced in many different fields and plays a very important role. In modal logic, it was discovered when researches started to study the expressiveness for the logic, in 1970 [4]. Then that was developed by Van Benthem [5] who came up with *p-relation* that define a symmetric directional relationship between models, i.e. bisimulation. In computer science, the concept was introduced by Milner while studying concurrency theory [6]. The notation was developed to test the equivalence of processes in Calculus of Communication System [7]. In the set Theory, it was used to determine if the equal objects can be identified with certain model [4]. Also, in formal verification, while describing a system, bisimulation was used to minimise the space of states [8].

Some of the past solutions for the bisimulation problem is from the perspective of the set theory. One approach to this problem was studied by Kanellakis and Smolka while studying the equivalence testing of the calculus of communicating systems expression [9]. Originally they designed the algorithm to test the equivalence called congruence of finite state processes in CCS [10]. The algorithm was given with $O(mn)$ in time and $O(m+n)$ in space. Another approach was given by Paige and Tarjan [10]. They converted the problem as relational coarsest partition problem. By refining with respect to old blocks rather than current blocks, they reduce the running time into $O(m \log n)$. Both of them did not solve the problem directly. By using a group of reasoning on different perspectives, they convert

the problem into a clear and easier problem. In this project, the learning algorithm will base on the work of [10].

For the first part of the project, the baseline algorithm will use the algorithm based on the set theory. In the algorithm, one logical is regarded as a *non-well-founded set*. "Such a set has an infinite descending membership sequence" i.e. the elements of the set recursively consisting itself [11]. Mathematically, in this part the multi-directed graph (logic structure) $G = \langle N, E\rangle$ is regarded as a group of sets where each set represents a *accessible pointed graphs*. Edges resents membership, that is edge $\langle m, n\rangle$ (also $m \to n$) means that node m has node n as a element (also $n \in m$). Then naturally, these sets will be *non-well-founded sets*. Thus extensionality axiom (two objects are equal if and only if they contain the same elements) cannot be used to define the bisimulation, since these sets may lead to cyclic arguments. In this project, the baseline algorithm will base on the definition from [12].

Definition 2.1. *Given two graphs $G_1 = \langle N_1, E_1\rangle$ and $G_2 = \langle N_2, E_2\rangle$, a bisimulation between G_1 and G_2 is a relation $b \subseteq N_1 \times N_2$ such that*

1) $u_1 b u_2 \wedge \langle u_1, v_1\rangle \in E_1 \Rightarrow \exists v_2 \in N_2(v_1 b v_2 \wedge \langle u_2, v_2\rangle \in E_2)$

2) $u_1 b u_2 \wedge \langle u_2, v_2\rangle \in E_2 \Rightarrow \exists v_2 \in N_2(v_1 b v_2 \wedge \langle u_1, v_1\rangle \in E_1)$

Also, for understanding purpose, a more common definition from the perspective of Kripke model [13] is given. Here the multi-directed graph (logic structure) are labelled transition system (LTS), denoted as $(S, \to, \models)$ with the set of actions A and the set of predicates P. S stands for a class of states, $\to$ stands for a collection of binary relations $\xrightarrow{a} \subseteq S \times S$ where $a \in A$. And $\models \subseteq S \times P$, where $s \models p$ indicate that predicate $p \in P$ holds in state $s \in S$ [13]. Definition in this prospect is given below [13].

Definition 2.2. *Let $(S, \to, \models)$ be an LTS over A and P. A bisimulation is a binary relation $R \subseteq S \times S$, satisfying:*

1) if sRt, then $s \models p \leftrightarrow t \models p$ for all $p \in P$.

2) if sRt and $s \xrightarrow{a} s'$ with $a \in A$, then there exists a t' with $t \xrightarrow{a} t'$ and $s'Rt'$

3) if sRt and $t \xrightarrow{a} t'$ with $a \in A$, then there exists an s' with $s \xrightarrow{a} s'$ and $s'Rt'$

2.2 Artificial Neural Networks (ANNs)

For the second part of the project, the capability of the MLPs for catching the bisimulation equivalence is explored. Here MLPs is chosen as the representative of ANNs as a kind for machine learning methods. Generally, machine learning can be defined as an experience-base computational method [14]. Mohri pointed out that machine learning is mainly used in the classification, regression, ranking, clustering and dimensionality reduction [14]. Among all the machine learning methods ANN is a unique architecture which is good at the pattern related problems [15]. This architecture is inspired by the neural cell in the brain. By simulate neural cell, a neural network is prospected to learn from experience also understand the logic concepts. In 1943, McCulloch and Pitts carried out the first neural network, namely the McCulloch-Pitts Neuron [15]. It is a binary discrete-time element with excitatory, that comes to the perception which can learn the patterns through supervised learning. And for multilayer perceptron (MLP), it is a feed forward artificial neural network. A MLP has an input layer, an output layer and at least one hidden layer [16] (See Fiture 2.1). More importantly, most ANNs are developed base on the MLP.

While research on logic and ANN has been studied. Logic or symbolic system is more expressive and more readable for human. Learning base on a symbolic system has less dependence on data. While commonly, the neural network is known as a less formal system short at human reasoning. Thus, neuro-symbolic integration is growing rapidly. It has

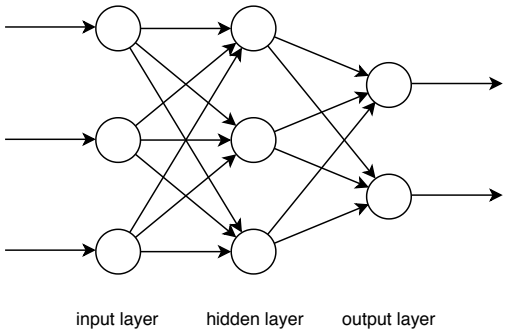


Figure 2.1: Typical MLP

been shown that feedforward connectionist networks are able to compute certain semantics operators in positional logic programs and approximate them in first-order logic [17]. Also, there are scholars constructed recurrent connectionist networks that can approximate logic program and its semantics [2, 3]. Further, for the logic relation, a system PAN who accept symbolic input as the condition of the logical rule is developed [18]. There are also research investigates the neural network approach for the first-order abductive inference[19].

However, this project will focus on MLP and bisimulation only. Because MLP is the most typical neural network and is part of many neural networks. By study, the capability, the compute ability of MLPs with certain depth will be evaluated.

3 Problem and Approach

Here we will give a specification of the problem, and a detailed approach (i.e. the design of solution). In order to simplify the problem and keep MLPs as simple as possible, one MLP will be used to check only the graphs (models) with a certain number of nodes and certain types of edges. In other words, each type of graph will have its own MLP. Notice that all the MLPs will have the same overall architecture. The only difference will be the number of nodes at each layer. That is only the bisimulation between graphs with the same scale will be studied. The problem of the project become that if an MLP is possible to compute the bisimulation equivalence of two same scale models and how well it can do. The basic steps of approach are as follows.

- Generate a standard distinguishing algorithm.
- Based on the standard algorithm, develop a dataset generator.
- Construct MLPs that can accept graphs pairs and output judgement.
- Do experiments on the performance of the MLP with different training sets. [1]

3.1 Standard Bisimulation Algorithm

For the first step, we will directly use the algorithm that solves the *relational coarsest partition problem* [10], i.e. given the relation E and initial partition P over a set U find the partition P that "every other stable partition is a refinement of it ". From the perspective of set-theory, it is the same as solving bisimulation equivalence [12]. Namely, if each block of the relational coarsest partition of the union of two graphs has nodes from both multi-directed graphs, these two graphs are bisimilar (see the set-perspective definition in Section 2). Moreover, compared with the algorithm given by Kanellakis and Smolka [9], their algorithm

[1] N.B. this step will be stated independently in Section 5

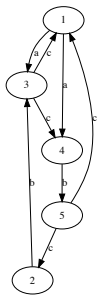
is more efficient in time (see Section 2.1). Yet, the algorithm given by Paige and Tarjan is not designed for multi-directed graphs. It only processes the directed graph with only one kind of edges. So by replacing every operation on one relation by a group of operations on each relation (see Step 4d in Algorithm 3.1), the algorithm can be used for multi-directed graphs. Here we give the description of the algorithm based on [10].

Algorithm 3.1. *Standard Bisimulation Algorithm*

1. *Get the union of two given graphs U.*
2. *Initialise the partition X and block set C, where $X = C = \{U\}$.*
3. *Get the initial partition Q, by refine the only block U in Q with respect to U itself. In other words, split U on the preimage of itself, i.e. $Q = \{E^{-1}(U), U - E^{-1}(U)\}$, where E is the any type of binary relation of nodes (also edges).*
4. *Loop until block set C is empty.*
   (a) *Pop the first block s from block set C*
   (b) *Check first two blocks of partition Q, that is contained in block s, make block b be the smaller one.*
   (c) *In partition X split the block s into block b and block $s' = s - b$. if s' is compound with respect to partition Q, add it back to block set C*
   (d) *For each type of relation E_i, i.e. $i = 1, i = 2$ if there are two types of edges. Refine every blocks in partition Q with respect to block b and block s'.*
   (e) *Add all blocks $x \in X$ that are compound with respect to partition Q to block set C*
5. *If there are nodes of both graphs in each block of partition Q return true, else return false.*

3.2 Test Case Generator

All the data used in this project is self-generated random and abstract without any human participation. Here the model (multi-directed graph) generated will be represented as an expend adjacency matrix (see Figure 3.1), where 1 stands for existence.



(a) Model

	1			2			3			4			5		
	a	b	c	a	b	c	a	b	c	a	b	c	a	b	c
1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
5	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0

(b) Adjacency Matrix

Figure 3.1: Example model represent

There are two generators implemented. However, only one is used in the final experiments (to be explained in Section 4) The first idea is to randomly generate graphs. Depends on the rate given, generate a bisimilar graph of another random non-bisimilar graph. Then record them into the appointed file with flags. Description of this algorithm is given as follow.

Algorithm 3.2. *Random Test Case Generator*

1. *Generate a random graph with certain scale, using the given density times a random number ($< 1, > 0$) as a probability of each possible edge.*
2. *According given rate of positive cases and negative cases, decide to generate a bisimilar graph (jump to Step 4) or non-bisimilar graph (jump to Step 3).*
3. *Keep generate random graph like Step 1 and check the bisimulation equivence, until they are not (then jump to Step 8).*
4. *Get the* relational coarsest partition P_{origin} *of given graph.*
5. *Randomly divides the nodes of the new graph into the same partition $P_{similar}$, i.e. the partition that has the same number of blocks (N.B. the length of each block not necessary to be same).*
6. *Assemble the minimum bisimilar graph, where each block of its partition contains only one node.*
7. *For each edge of the minimum graph, generate a random number (≥ 1) of edges between the random nodes from two corresponding blocks of $P_{similar}$ respectively.*
8. *Convert the new generated graph into adjacency matrix.*
9. *Write two graphs with flag into file (jump to Step 1 if there are not enough test cases).*

However, the test cases generated may not actually random, which may have some superficial features that are easy to be caught by the later MLP (to be described in Section 3.3). For example the heuristic algorithm for generate bisimilar graph (see Algorithm 3.2, Step 4 to Step 7) may have some kind of inclination. To get rid of these potential bias, a full case generator is designed. This generator can generate all possible graphs in given scale and record all possible combinations with flags. The basic idea of iteration is that since a graph can be seen as an adjacency matrix, it can be represented as a binary string. By counting in binary, all possible graphs on the same scale can be traversed. Here gives the description of this algorithm.

Algorithm 3.3. *Full Test Case Generator*

1. *For every graph, calculate its minimum bisimilar graph. Meanwhile, construct a dictionary where the key is the minimum bisimilar graph and the value is a set of all graphs that are bisimilar, i.e. have the same minimum bisimilar graph which is their key.*
2. *According to the dictionary, catalogue all the graph (graphs with the same key will have the same label).*
3. *Go through all graphs again, generate all combinations with flag base on their label.*

3.3 MLP Structure

After the test case generator produces data, the network should be trained. The structure is designed to be as simple as possible. Combine with the process step of the standard algorithm, the network is structured to meet the process (see Figure 3.2). The first layer is the input layer, which will accept fixed length binary (i.e. adjacency matrix). In order to push the network to get the abstraction, the second layer called re-represent layer has fewer neurons than the input layer. And the first two layers are divided into two independent parts that share parameters (see Part 1 and Part 2 in Figure 3.2). Each part will accept one graph. The distinguishing part will accept input from two graphs and get the conclusion

about bisimulation equivalence, i.e. output the probability of positive and negative. There are three reasons to design a network like that. First is by sharing the parameters, the overall training will be master. The second reason is that it solves the problem of sequence, i.e. the output will not be affected by the priority of two graphs. The third reason is that this kind of modular design will be much easier for reuse, e.g. used as part of the graph simplify the network.

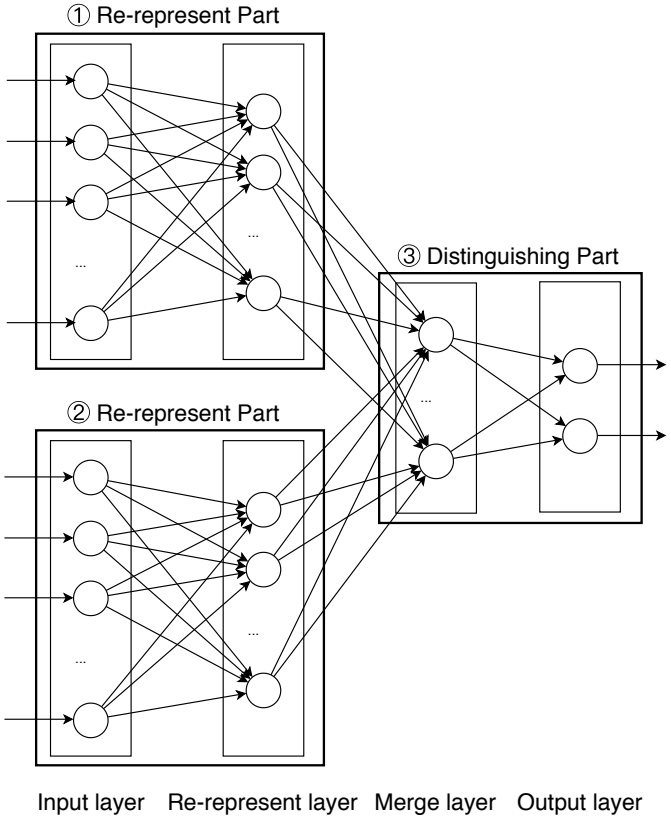


Figure 3.2: MLP model

3.4 General Structure

The design of the general structure is aimed at developing a reusable tool. Thus the structure is very simple and highly modularised (see Figure 3.3). Dataset generator and ML-algorithm is packed independently. And each module can be called in a python program or used as a command line tool directly by the user.

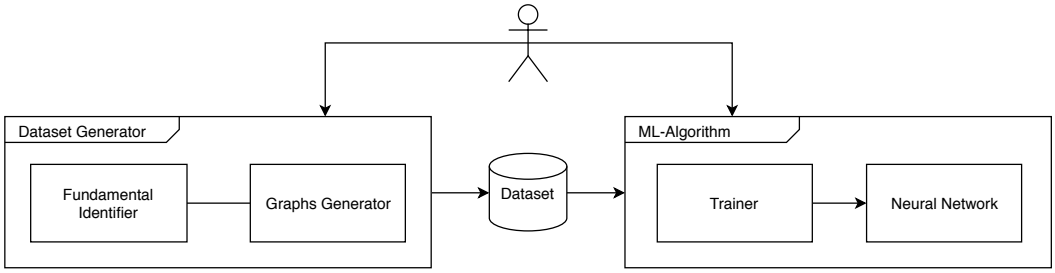


Figure 3.3: General structure

4 Realisation

Implementation is not always the same as the design. During the realisation of the standard bisimulation algorithm, a visualisation tool for graphs is needed for development. It gives the graph an intuitional presentation so that the standard algorithm can be verified much easier. And the same situation happens in the development of machine learning part. Follows are the actual structure implemented (see Figure 4.1).

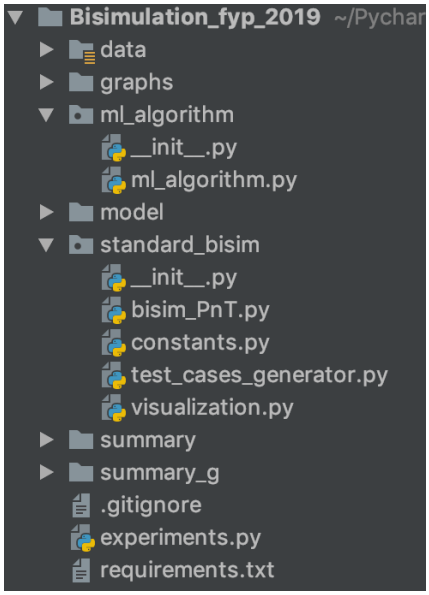


Figure 4.1: Actual structure

`/data/` It is used to store the generated test cases data. They are saved as `.csv` file and are the output of `test_cases_generator.py`, the input of `ml_algorithm.py`.

`/model/` This directory is used to store the parameters of trained/half-trained machine learning models. It allowed training resume form break point and also directly apply of trained models. Models are save as `.ckpt` files. And during the training, `ml_algorithm.py` will keep recording the current model.

`/graphs/` The graphs figure that been visualised is put in this directory.

`/summary/` It is used to store the summary file generate by `ml_algorithm.py`. These file record all the metric calculated during the training (i.e. loss, accuracy, precision and recall) for later analyse.

`/summary_g/` TensorFlow Graphs generated during the calculation of train are stored here. It can be visualised by the TensorBoard, which will do a great help in the implementation of the neural network.

`/requirement.txt` This file is a text file which records all the package needed. An environment can be established easily based on this file.

`/.gitignore` The project is managed by the distributed version control system `Git`. It is a configuration file that will make `Git` ignore assigned files.

`/experiments.py` This file contain the experiments of the proeject, and will described in Section 5.

`/standard_bisim/` Under this directory, that are `bisim_PnT.py`, `test_cases_generator.py` and `visualisation.py`. Detail will described below (i.e. Section 4.1).

`/ml_algorithm/` It is used to hold `ml_algorithm.py`. Detial of this part will be descussed below (i.e. Section 4.1).

4.1 Components Implementation

The detailed description of implementation is given in this part.

4.1.1 Implementation of Standard Bisimulation Algorithm

The first part of the project is to develop the standard bisimulation algorithm base on the algorithm given in [10]. This algorithm is implemented as a class `BisimPnT`. Following functions are included. Also the code listing of `coarsest_partition()` is given here (See Listing 4.1).

Signature	`__init__(self, labels, graph_g, graph_h=None)`
Description	Initiate the object. Union the given graphs and record the possible edges (`labels`)
Return	N/A

Signature	`preimage(self, block_s, label=None)`
Description	Calculate the preimage of block with respect to a given type of edge (also relation). If type asigned, it will compute the preimage without distinguishing the types of edge, i.e. all edges will be seen as same relation (`label`).
Return	$E^{-1}_{\text{label}}(\text{block\_s}) = \{x \mid \exists y \in \text{block\_b such that } xE_{\text{label}}y\}$

Signature	`split_block(self, block_b, block_s, label=None)`
Description	Calculate the result of split the `block_b` with the preimage of `block_s`. It `label` is not assigned, then block b will be split with respect to all relation.
Return	$\text{split}(\text{block\_b}, \text{block\_s}) = \{\text{block\_b} \cap E^{-1}_{\text{label}}(\text{block\_s}), \text{block\_b} - E^{-1}_{\text{label}}(\text{block\_s})\}$

Signature	`compound_blocks(self, partition_Q, partition_X)`
Description	Check if all the blocks in `parition_X`, if their are compound or not. Return a set of all compound blocks.
Return	a set of blocks in `partition_X` that is compound with respect to `partition_Q`

Signature	`is_compound_to(self, block_s, partition)`
Description	Check if `block_s` is compound with respect to `partition`, i.e. if `block_s` contain more then one block in `partition`.
Return	The smaller contained block in `partition` of `False` if `block_s` is simple

Signature	`coarsest_partition(self, plot=False)`
Description	Calculate the coarsest partition of the graph given when initialisation.
Return	The coarsest partition

Signature	`get_min_graph(self)`
Description	Assemble a min bisimulation graph.
Return	The minimum bisimulation graph

Signature	`is_bisimilar(self)`
Description	Check it two given graph is bisimulation by checking if there are nodes from both graph in each of the coarsest partition.
Return	`True` or `False`

Listing 4.1: Function calculate coarsest partition

```
def coarsest_partition(self, plot=False):

    # Step 1 Initial U
    block_u = set(self.full_graph_U.nodes())

    # Step 2
    partition_X = [block_u]
    block_set_C = [block_u]

    # Step 3
    partition_Q = self.split_block(block_u, block_u)

    # Step 4
    block_set_C = self.compound_blocks(partition_Q, partition_X)
    while len(block_set_C) != 0:
        # Step a:
        # select refining block_b
        # a compound block of partition_X
        block_s = block_set_C.pop()
        # a block of Q that contained in s
        block_b = self.is_compound_to(block_s, partition_Q)

        # Step c:
        # update X:
        partition_X.remove(block_s)
        partition_X.append(block_s - block_b)
        partition_X.append(block_b)  # split block S in X

        # if the rest is still not simple, put it back to C
        if self.is_compound_to(block_s - block_b, partition_Q) is not
        ↪  False:
            block_set_C.append(block_s)

```

```
        # Step d
        for label in self.labels:

            # compute preimage of B
            preimage_b = self.preimage(block_b, label)
            # compute preimage of S - B
            preimage_s_sub_b = self.preimage(block_s - block_b, label)

            # refine Q with respect to B and S-B
            new_partition_Q = []
            for block_d in partition_Q:
                block_d1 = block_d.intersection(preimage_b)
                block_d2 = block_d - block_d1
                block_d11 = block_d1.intersection(preimage_s_sub_b)
                block_d12 = block_d1 - preimage_s_sub_b

                if len(block_d2) and block_d2 not in new_partition_Q:
                    new_partition_Q.append(block_d2)
                if len(block_d11) and block_d11 not in new_partition_Q:
                    new_partition_Q.append(block_d11)
                if len(block_d12) and block_d12 not in new_partition_Q:
                    new_partition_Q.append(block_d12)
                partition_Q = new_partition_Q

        # Step 5
        block_set_C = self.compound_blocks(partition_Q, partition_X)
        # if needed visualise the result
        if plot:
            vi.plot_graph_with_partition(self.full_graph_U, partition_Q)

    self.co_partition = partition_Q
    return partition_Q
```

However, there is a visualisation tool that can plot the multi-directed graphs (e.g. Figure 3.1(a) is drawn by this tool) are developed based on `graphviz`. It can also colour the graph with a given partition.

With this tool, the development and component tests became much easier. There is one main function below.

Signature	`plot_graph_with_partition(graph, blocks=None, file_name="test")`
Description	Visualisation the graph with partition and save the image. If no `blocks` (i.e. partition) given it will output image without colour.
Return	Image file

4.1.2 Implementation of Test Case Generator

It was considered that there are two generators (see Section 3.2). However, during the primary test, the full test generator (see Algorithm 3.3) is too expensive both in computation and space. Specifically, for the graphs with n node and m types of edges. Then there are 2^{n*n*m} possible graphs. After combination, there will be $(2^{n^2m} * (2^{n^2m} - 1))/2 \approx 2^{2n^2m}$. If graphs have 3 nodes and 2 types of edges, there will be 2^{36} (around 60 billion) test cases.

Thus, it is too expensive for this project. Instead, we will repeat the experiments in different parameters to minimise the possibles.

Here we give the code about random test case generator described in Algorithm 3.2 and the heuristic algorithm for generate bisimilar graph (see Algorithm 3.2, Step 4 to Step 7). Also, the generator is packed as a command line tool for the convenience of further use (see Figure 4.2).

```
(venv) ➜  Bisimulation_fyp_2019 git:(master) ✗python standard_bisim/test_cases_generator.py -h
usage: test_cases_generator.py [-h] [-t {random,all}] [-n NUMBER]
                               [-f FILE_NAME] [-v NODE_NUMBER]
                               [-e EDGE_TYPE_NUMBER] [-r P_RATE]
                               [-p PROBABILITY]

optional arguments:
  -h, --help            show this help message and exit
  -t {random,all}, --type {random,all}
                        Type of data set
  -n NUMBER, --number NUMBER
                        The length of data set
  -f FILE_NAME, --file_name FILE_NAME
                        Name of the output file
  -v NODE_NUMBER, --node_number NODE_NUMBER
                        Number of the nodes of the graph in the data set
  -e EDGE_TYPE_NUMBER, --edge_type-number EDGE_TYPE_NUMBER
                        The total types of the edge in the graphs
  -r P_RATE, --p_rate P_RATE
                        Rate of the positive cases over all cases
  -p PROBABILITY, --probability PROBABILITY
                        The density of the random generate graphs
```

Figure 4.2: Arguments of test cases generator

Listing 4.2: Random test case generator

```
1  def test_cases_generator(c_type="random", number=100,
   ↪  file_name="test_cases", min_node_number=5, edge_type_number=3,
   ↪  probability=0.5, p_rate=0.5):
2      labels = [chr(97 + i) for i in xrange(edge_type_number)]
3      if not os.access('./data/', os.R_OK):
4          os.mkdir('./data/')
5      with open('./data/' + file_name + '.csv', 'w') as csvfile:
6          print('write in path: ' + './data/' + file_name + '.csv')
7          if c_type == 'random':
8              fieldnames = ['g1', 'g2', 'bis']
9              writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
10             writer.writeheader()
11
12             for _ in xrange(number):
13                 if _ % (number/10) == 0: print("generating # %d" % _)
14                 g1 = ''
15                 while not g1 or not nx.is_weakly_connected(g1):
16                     g1 = random_labeled_digraph(min_node_number,
                        ↪  edge_type_number, random.random() * probability)
17                 g1_bin = get_bin_array_from_graph(g1, min_node_number,
                    ↪  edge_type_number)
18                 bis = 1
19                 if random.random() > p_rate:
```

```
                    g2 = generate_random_similar(g1, edge_type_number)
                else:
                    g2 = random_labeled_digraph(min_node_number,
                     ↪  edge_type_number, random.random() * probability)
                    k = bi.BisimPnT(labels, g1, g2)
                    bis = int(k.is_bisimilar())
                g2_bin = get_bin_array_from_graph(g2, min_node_number,
                 ↪  edge_type_number)
                writer.writerow({fieldnames[0]: g1_bin, fieldnames[1]:
                 ↪  g2_bin, fieldnames[2]: bis})
```

Listing 4.3: Function of generating random bisimilar graph

```
def generate_random_similar(graph, edge_type_number):
    k = bi.BisimPnT([chr(97 + t) for t in range(edge_type_number)],
     ↪  graph)
    min_graph = k.get_min_graph()
    min_node_number = min_graph.order()
    partition = [{i} for i in range(min_node_number)]
    node_number = graph.order()
    if min_node_number == node_number:
        return random_relabel_nodes(graph)
    for i in xrange(min_node_number, node_number):
        partition[random.randint(0, min_node_number - 1)].add(i)
    result_graph = None
    # print min_node_number, node_number
    while not result_graph or result_graph.order() != graph.order():
        result_graph = nx.MultiDiGraph()
        for start_node_type in xrange(min_node_number):
            for end_node_type in min_graph.successors(start_node_type):
             ↪  # for each origin type pair
                for edge_label in {edge['label'] for edge in
                 ↪  min_graph.get_edge_data(start_node_type,
                 ↪  end_node_type).values()}:
                    # for each type edge
                    accept = False
                    while not accept:
                        start_nodes =
                         ↪  random.sample(partition[start_node_type],
                         ↪  random.randint(0,
                         ↪  len(partition[start_node_type])))
                        for start_node in start_nodes:
                            end_nodes =
                             ↪  random.sample(partition[end_node_type],
                             ↪  random.randint(0,
                             ↪  len(partition[end_node_type])))
                            for end_node in end_nodes:
                                exist_types = {exist_type_dic['label'] for
                                 ↪  exist_type_dic in
                                 ↪  result_graph.get_edge_data(
                                 ↪  start_node,end_node,
                                 ↪  default={0:{'label':'#'}}).values()}
                                if  edge_label not in exist_types:
```

```
                                    result_graph.add_edge(start_node,
                                    ↪  end_node, label=edge_label)
                        t1 = partition[start_node_type].copy()
                        t2 = partition[end_node_type].copy()
                        for start_node in partition[start_node_type]:
                            for end_node in partition[end_node_type]:
                                exist_types = {exist_type_dic['label'] for
                                ↪  exist_type_dic in
                                ↪  result_graph.get_edge_data(
                                ↪  start_node,end_node,
                                ↪  default={0:{'label':'#'}}).values()}
                                if edge_label in exist_types:
                                    t1.discard(start_node)
                                    t2.discard(end_node)
                        accept = not bool(t1) and not bool(t2)
    return result_graph
```

4.1.3 Implementation of Machine Learning

This neural network is developed on the `Tensorflow` along with the visualisation tool `TensorBoard`. TensorFlow Graph is a computation graph drew by `TensorBoard` that indicates the process of the computation. Figure 4.3 are the TensorFlow graph of the network model .

After all parameters are randomly initialised by `tf.random_normal_initialiser()`, the model start with a fixed length (mn^2) input layer, and accept the numeral input, i.e. the 1 or 0 in the adjacency matrix are accepted as a number rather than a catalogue. And the activate function used here is rectified linear unite (ReLU). Because it is computationally efficient and more plausible on biology which may improve the comprehension of the model. Also it promises fewer problems on vanishing gradient [20]. The adjacency matrixes become tensor after input. Then the tensor of each graph goes through the re-representing part (see g1_p and g2_p in Figure 4.3). After that, they are merged into one big tensor and go through the distinguishing part (see merge and logits in Figure 4.3). At the end of the distinguishing part, the tensor is output as logits, i.e. two number that indicated the possibility of bisimilar or not. Finally pass through a *softmax* layer the prediction will be given. Yet, during the training, there are metrics including loss, accuracy, precision, recall logged for further studying. The main code is shown in Listing 4.4. It is also packed as a commend line tools (see Figure 4.4).

Listing 4.4: ML algorithm and training

```
def fc(self, continue_train=False):

    kernel_initializer = tf.initializers.glorot_normal()
    tf.reset_default_graph()
    small_layer_number = int(math.log(self.tupe_length) * 5)
    # print small_layer_number, self.tupe_length
    with tf.name_scope('input'):
        g1 = tf.placeholder(tf.float32, [None, self.tupe_length])
        g2 = tf.placeholder(tf.float32, [None, self.tupe_length])
        y = tf.placeholder(tf.float32, [None, 2])

    with tf.name_scope('g1_p'):
        with tf.variable_scope('graph_pross'):
```

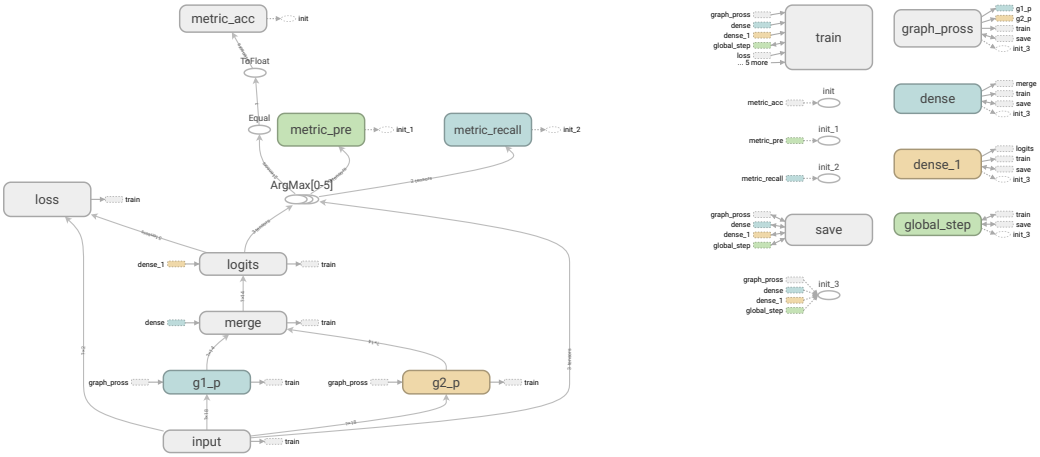


Figure 4.3: TensorFlow graph of MLP

```
(venv) ➜  Bisimulation_fyp_2019 git:(master) ✗  python ml_algorithm/ml_algorithm.py -h
usage: ml_algorithm.py [-h] [-e EPOCH] [-l LEARNING_RATE] [-b BATCH_SIZE]
                       [-p DATA_PATH] [-r TEST_TRAIN_RATE] [-c CONTINUE_TRAIN]
                       [-n MODEL_NAME]

optional arguments:
  -h, --help            show this help message and exit
  -e EPOCH, --epoch EPOCH
                        Number of training epochs
  -l LEARNING_RATE, --learning_rate LEARNING_RATE
                        Initial learning rate
  -b BATCH_SIZE, --batch_size BATCH_SIZE
                        Number of data for one batch
  -p DATA_PATH, --data_path DATA_PATH
                        Path to input data
  -r TEST_TRAIN_RATE, --test_train_rate TEST_TRAIN_RATE
                        The rate of test cases and train cases
  -c CONTINUE_TRAIN, --continue CONTINUE_TRAIN
                        Continue last training
  -n MODEL_NAME, --model_name MODEL_NAME
                        The name of the model
```

Figure 4.4: Arguments of machine learning algorithm

```
14              g1_dence1 = tf.layers.dense(g1, self.tupe_length,
                ↪  activation=tf.nn.relu,
                ↪  kernel_initializer=kernel_initializer,
                ↪  bias_initializer=tf.random_normal_initializer(),
                ↪  name='dence1')
15
16              g1_s_dence1 = tf.layers.dense(g1_dence1, small_layer_number,
                ↪  activation=tf.nn.relu,
                ↪  kernel_initializer=kernel_initializer,
                ↪  bias_initializer=tf.random_normal_initializer(),
                ↪  name='s_dence1')
17
18          with tf.name_scope('g2_p'):
19              with tf.variable_scope('graph_pross', reuse=True):
```

```
20              g2_dence1 = tf.layers.dense(g2, self.tupe_length,
                ↪  activation=tf.nn.relu, name='dence1', reuse=True)
21
22              g2_s_dence1 = tf.layers.dense(g2_dence1, small_layer_number,
                ↪  activation=tf.nn.relu, name='s_dence1', reuse=True)
23          with tf.name_scope('merge'):
24              two_graphs = tf.concat([g1_s_dence1, g2_s_dence1], 1)
25              merge_layer = tf.layers.dense(two_graphs, small_layer_number,
                ↪  activation=tf.nn.relu, kernel_initializer=kernel_initializer,
                ↪  bias_initializer=tf.random_normal_initializer())
26          with tf.name_scope('logits'):
27              logits = tf.layers.dense(merge_layer, 2, activation=tf.identity,
                ↪  kernel_initializer=kernel_initializer,
                ↪  bias_initializer=tf.random_normal_initializer())
28          with tf.name_scope('loss'):
29              loss = tf.losses.softmax_cross_entropy(y, logits)
30              tf.summary.scalar('loss', loss)
31
32          global_step = tf.Variable(0, trainable=False, name='global_step')
33
34          with tf.name_scope('train'):
35              train =
                ↪  tf.train.AdadeltaOptimizer(learning_rate=self.learning_rate)
                ↪  .minimize(loss=loss, global_step=global_step)
36
37          acc_metric, acc_metric_update = tf.metrics.accuracy(tf.argmax(logits,
            ↪  1), tf.argmax(y, 1), name='metric_acc')
38          pre_metric, pre_metric_update = tf.metrics.precision(tf.argmax(logits,
            ↪  1), tf.argmax(y, 1), name='metric_pre')
39          recall_metric, recall_metric_update =
            ↪  tf.metrics.recall(tf.argmax(logits, 1), tf.argmax(y, 1),
            ↪  name='metric_recall')
40          tf.summary.scalar('accuracy', acc_metric_update)
41          tf.summary.scalar('precision', pre_metric_update)
42          tf.summary.scalar('recall', recall_metric_update)
43
44          metric_acc_var = tf.get_collection(tf.GraphKeys.LOCAL_VARIABLES,
            ↪  scope="metric_acc")
45          acc_initializer = tf.variables_initializer(var_list=metric_acc_var)
46          metric_pre_var = tf.get_collection(tf.GraphKeys.LOCAL_VARIABLES,
            ↪  scope="metric_pre")
47          pre_initializer = tf.variables_initializer(var_list=metric_pre_var)
48          metric_recall_var = tf.get_collection(tf.GraphKeys.LOCAL_VARIABLES,
            ↪  scope="metric_recall")
49          recall_initializer =
            ↪  tf.variables_initializer(var_list=metric_recall_var)
50
51          merged_summary = tf.summary.merge_all()
52          saver = tf.train.Saver()
53
54          with tf.Session() as sess:
55              if continue_train:
```

```
56              saver.restore(sess, DIR_MODEL + '/' + self.model_name +
                ↪  '/model.ckpt')
57              print('continue training, model loaded')
58          else:
59              self.clean_old_record()
60              print('new training, old record cleaned')
61              # initial
62              sess.run(tf.global_variables_initializer())
63
64          train_writer = tf.summary.FileWriter(DIR_SUMMARY + '/' +
            ↪  self.model_name, sess.graph)
65          test_writer = tf.summary.FileWriter(DIR_SUMMARY + '/' +
            ↪  self.model_name + '_test')
66
67          for epoch in range(self.epochs):
68              sess.run([acc_initializer, pre_initializer,
                ↪  recall_initializer])
69              loss_p = None
70              g_step = None
71
72              summary_11, g_step_1= sess.run([merged_summary,global_step],
                ↪  feed_dict={g1: self.g1_dfs[1], g2: self.g2_dfs[1], y:
                ↪  self.label_dfs[1]})
73
74              test_writer.add_summary(summary_11, g_step_1)
75
76              for i in range(self.batch_number):
77                  _, loss_p, summary, g_step = sess.run([train, loss,
                    ↪  merged_summary, global_step], feed_dict={g1:
                    ↪  self.g1_dfs[0][i], g2: self.g2_dfs[0][i], y:
                    ↪  self.label_dfs[0][i]})
78                  train_writer.add_summary(summary, g_step)
79
80              if epoch % 10 == 0:
81                  # summary,acc = sess.run([merged_summary ,acc_metric])
82                  acc, pre, recall = sess.run([acc_metric, pre_metric,
                    ↪  recall_metric])
83                  log_str = "Epoch %d \t G_step %d \t Loss=%f \t
                    ↪  Accuracy=%f \t Precision=%f \t Recall=%f "
84                  print(log_str % (epoch, g_step, loss_p, acc, pre,
                    ↪  recall))
85                  saver.save(sess, DIR_MODEL + '/' + self.model_name +
                    ↪  '/model.ckpt')
86
87      train_writer.close()
88      test_writer.close()
```

4.2 Components Test

During the development, some tests are used for examining each component. Here some of the test cases and the test results will be given.

4.2.1 Test for Standard Bisimulation Algorithm

The test cases used here is from the textbook. And by manipulating these cases, this component can be tested.

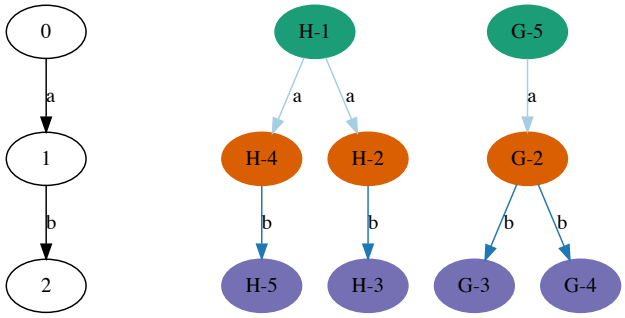
Test case 1:
This case is form [21]. By changing two egdes in this case (i.e. H-4 $\xrightarrow{b}$ H-5 $\Longrightarrow$ H-4 $\xrightarrow{c}$ H-5, G-2 $\xrightarrow{b}$ G-4 $\Longrightarrow$ G-2 $\xrightarrow{c}$ G-4), these two graph should lose bisimulation equivalence. The test code is shown in Listing 4.5. The output is shown in Figure 4.5-4.8. Notice that in a figure like Figure 4.6, edges that are one type has the same colour and nodes in the same partition (i.e. are bisimilar) will be painted in the same colour.

Listing 4.5: Test code for case 1

```
# ============== example 1 ==============
G = nx.MultiDiGraph()
G.add_edge(5, 2, label='a')
G.add_edge(2, 3, label='b')
G.add_edge(2, 4, label='b')
# G.add_edge(2, 4, label='c')

H = nx.MultiDiGraph()
H.add_edge(1, 2, label='a')
H.add_edge(1, 4, label='a')
H.add_edge(2, 3, label='b')
H.add_edge(4, 5, label='b')
# H.add_edge(4, 5, label='c')

labels = ['a', 'b']
# labels = ['a', 'b', 'c']

k = BisimPnT(labels, G, H)
par = k.coarsest_partition()
vi.plot_graph_with_partition(k.full_graph_U,par,'example_1')
vi.plot_graph(BisimPnT(labels,H).get_min_graph(), 'mini_g')
print("Example 1: ")
print(par)
print(k.is_bisimilar())
```

```
/Users/cx/PycharmProjects/Bisimulation_fyp_2019/venv/bin/python
  /Users/cx/PycharmProjects/Bisimulation_fyp_2019/standard_bisim/bisim_PnT.py
Example 1:
[set(['H-1', 'G-5']), set(['H-4', 'G-2', 'H-2']), set(['G-3', 'H-5', 'H-3', 'G-4'])]
True

Process finished with exit code 0
```

Figure 4.5: Output for test case 1 positive instance



(a) Mini graph (b) Coloured graphs

Figure 4.6: Test case 1: positive instance

```
/Users/cx/PycharmProjects/Bisimulation_fyp_2019/venv/bin/python
  /Users/cx/PycharmProjects/Bisimulation_fyp_2019/standard_bisim/bisim_PnT.py
Example 1:
[set(['H-1']), set(['G-5']), set(['H-2']), set(['G-2']), set(['H-4']), set(['G-3',
   'H-5', 'H-3', 'G-4'])]
False

Process finished with exit code 0
```

Figure 4.7: Output for test case 1 negative instance

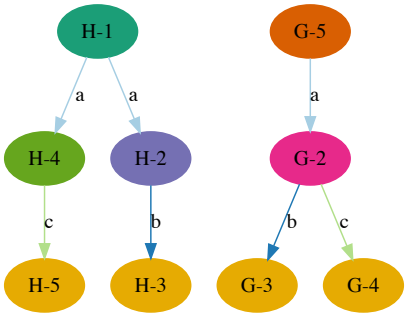


Figure 4.8: Test case 1: negative instance

Test case 2:
This case is form [21]. By changing two egdes in this case (i.e. G-3 $\xrightarrow{c}$ G-2 $\Longrightarrow$ G-3 $\xrightarrow{b}$ G-2 these two graph should lose bisimulation equivalence. The test code is shown in Listing 4.6. The output of the test result is shown in Figure 4.9-4.12.

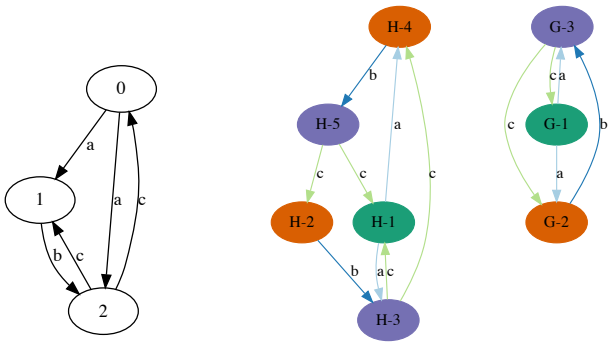
Listing 4.6: Test code for case 2

```
1 # =============== example 2 ===============
2 G = nx.MultiDiGraph()
```

```
G.add_edge(1, 3, label='a')
G.add_edge(1, 2, label='a')
G.add_edge(2, 3, label='b')
G.add_edge(3, 1, label='c')
G.add_edge(3, 2, label='c')
# G.add_edge(3, 2, label='b')

H = nx.MultiDiGraph()
H.add_edge(1, 3, label='a')
H.add_edge(1, 4, label='a')
H.add_edge(2, 3, label='b')
H.add_edge(3, 1, label='c')
H.add_edge(3, 4, label='c')
H.add_edge(4, 5, label='b')
H.add_edge(5, 1, label='c')
H.add_edge(5, 2, label='c')

labels = ['a', 'b', 'c']
k = BisimPnT(labels, G, H)
partition = k.coarsest_partition()
vi.plot_graph_with_partition(k.full_graph_U, partition, 'example_2')
vi.plot_graph(BisimPnT(labels, H).get_min_graph(), 'mini_g2')
print("Example 2: ")
print(partition)
print(k.is_bisimilar())
```

```
/Users/cx/PycharmProjects/Bisimulation_fyp_2019/venv/bin/python
  /Users/cx/PycharmProjects/Bisimulation_fyp_2019/standard_bisim/bisim_PnT.py
Example 2:
[set(['G-1', 'H-1']), set(['H-4', 'G-2', 'H-2']), set(['G-3', 'H-5', 'H-3'])]
True

Process finished with exit code 0
```

Figure 4.9: Output for test case 2 positive instance



(a) Mini graph (b) Coloured graphs

Figure 4.10: Test case 2: positive instance

```
/Users/cx/PycharmProjects/Bisimulation_fyp_2019/venv/bin/python
  /Users/cx/PycharmProjects/Bisimulation_fyp_2019/standard_bisim/bisim_PnT.py
Example 2:
[set(['H-1']), set(['G-1']), set(['H-4', 'H-2']), set(['G-2']), set(['G-3']),
  set(['H-5', 'H-3'])]
False

Process finished with exit code 0
```

Figure 4.11: Output for test case 1 negative instance

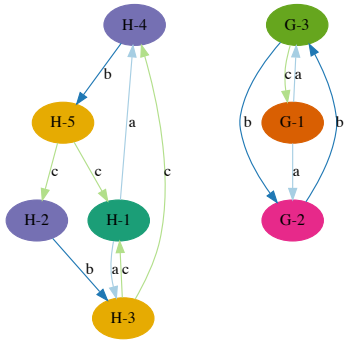


Figure 4.12: Test case 2: negative instance

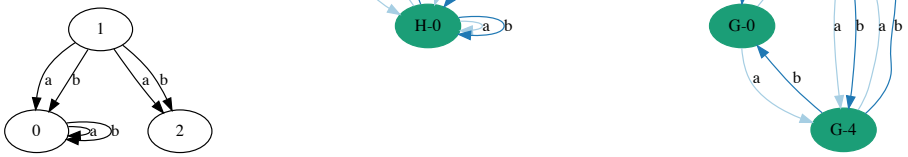
4.2.2 Test for test case generator

Function for Generating Random bisimilar Graph:
Firstly the function `generate_random_similar()` is tested. The idea of testing is to randomly generate a graph then call the function. Use the *standard bisimulation algorithm* to compute the bisimulation equivalence of the output (to be seen in Listing 4.7). Here, we also give some test generated results (see Figure 4.13-4.16). Notice that these results are selected examples for different types of result.

Listing 4.7: Test code for `generate_random_similar`

```
# =============== test for random_labeled_digraph() ===============
flag = True
i = 0

# if there are any error the loop will break
# until reach the test time
while flag :
    i = i+1
    a = random_labeled_digraph(5,2,0.5*random.random())
    b = generate_random_similar(a, 2)
    k = bi.BisimPnT(['a','b'],a,b)
    flag = k.is_bisimilar()
    print i
    print flag
    if i == 100:
        break
```

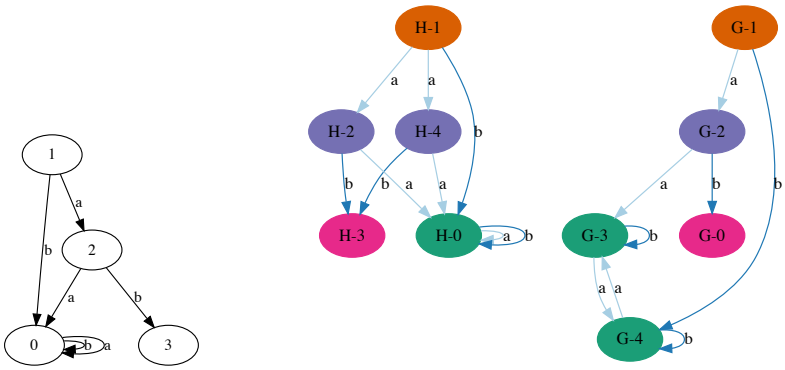
```
17
18 vi.plot_graph(a, 'test')
19 vi.plot_graph(bi.BisimPnT(['a','b'],a).get_min_graph(),"mini_a")
20 vi.plot_graph(b, 'test1')
21 vi.plot_graph(bi.BisimPnT(['a','b'],a).get_min_graph(),"mini_b")
```



(a) Mini graph

(b) Coloured graphs

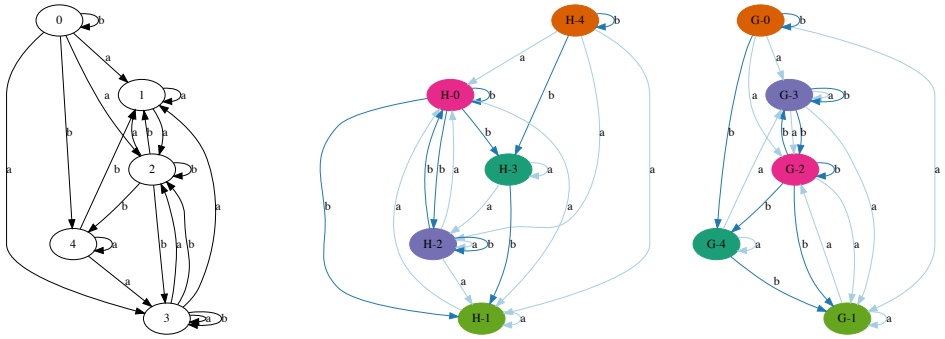
Figure 4.13: Test result 1



(a) Mini graph

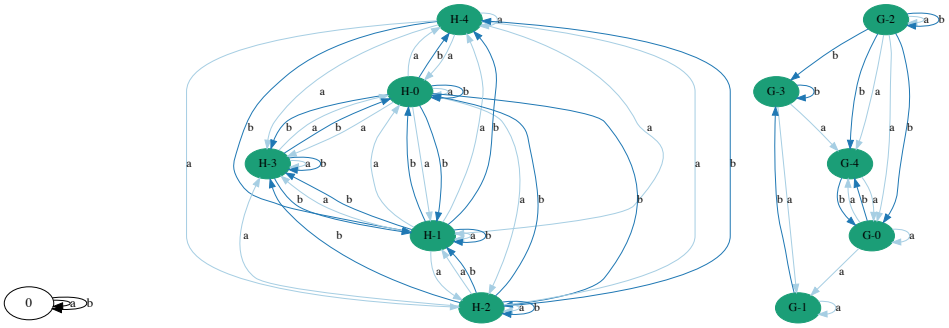
(b) Coloured graphs

Figure 4.14: Test result 2



(a) Mini graph (b) Coloured graphs

Figure 4.15: Test result 3



(a) Mini graph (b) Coloured graphs

Figure 4.16: Test result 4

Functions for generator: For the Generator, the test is for the I/O. Here part of the file generated with 3 nodes and 2 types of edge is given in Table 4.1. And the running time screenshot is shown in Figure 4.17.

```
(venv) ➜  Bisimulation_fyp_2019 git:(master) ✗  python standard_bisim/test_cases_generator.py
=============== strat generating data ===============
write in path: ./data/test_cases.csv
generating # 0
generating # 100
generating # 200
generating # 300
generating # 400
generating # 500
generating # 600
generating # 700
generating # 800
generating # 900
```

Figure 4.17: Screen shot of generator running

g1	g2	bis
1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1	0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0	0
0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0	0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1	1
1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0	0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0	1
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0	1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0	0
1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1	0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1	0
1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0	0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1	0
1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0	0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1	0
1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0	1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0	1
0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1	0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0	0
1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1	1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1	1
...	...	...

Table 4.1: Part of the generated file `test_cases.csv`

4.2.3 Test for ML Algorithm

Here we will use some simple generated test cases. And through the change of the metrics, the algorithm can be tested on some level. Here give graphs of first 1000 epochs training of the test sample where the node number is 3 and the type number of edges is 2 and 1000 entries (see Figure 4.19), along with the running time screenshot (see Figure 4.18). The orange and blue lines indicate the trend of metrics on train data and test data respectively. In Figure 4.19(a), along with the training the accuracy on training data and test data growth distinctively. And in Figure 4.19(b), the loss on both data sets decreases. Several tests show similar results. It denotes that the model is actually learning as expected.

```
(venv) ➜  Bisimulation_fyp_2019 git:(master) ✗ python ml_algorithm/ml_algorithm.py
============== star training ==============
loading file: ./data/test_cases.csv
new training, old record cleaned
Epoch 0         G_step 10       Loss=0.657330   Accuracy=0.531000       Precision=0.886497      Recall=0.524306
Epoch 10        G_step 110      Loss=0.634410   Accuracy=0.530000       Precision=0.661448      Recall=0.532283
Epoch 20        G_step 210      Loss=0.627578   Accuracy=0.532000       Precision=0.596869      Recall=0.537919
Epoch 30        G_step 310      Loss=0.621637   Accuracy=0.545000       Precision=0.590998      Recall=0.551095
Epoch 40        G_step 410      Loss=0.615556   Accuracy=0.553000       Precision=0.596869      Recall=0.558608
Epoch 50        G_step 510      Loss=0.609536   Accuracy=0.569000       Precision=0.606654      Recall=0.574074
Epoch 60        G_step 610      Loss=0.603661   Accuracy=0.576000       Precision=0.610568      Recall=0.581006
Epoch 70        G_step 710      Loss=0.597446   Accuracy=0.588000       Precision=0.622309      Recall=0.592179
Epoch 80        G_step 810      Loss=0.592332   Accuracy=0.602000       Precision=0.624266      Recall=0.607619
Epoch 90        G_step 910      Loss=0.587339   Accuracy=0.601000       Precision=0.616438      Recall=0.608108
```

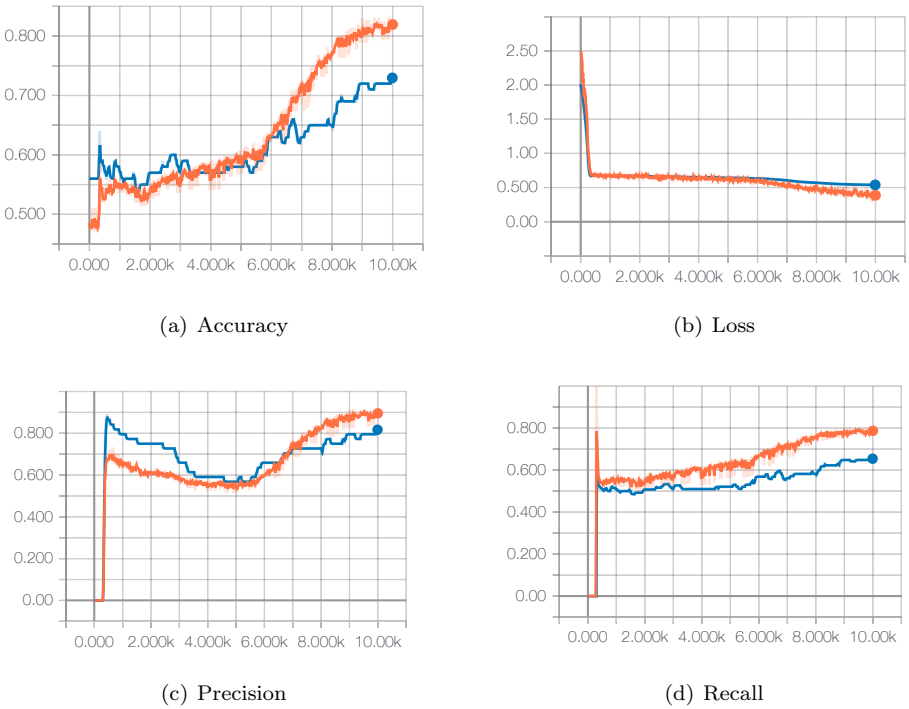
Figure 4.18: Screen shot of machine learning algorithm running

4.3 Project Progress

The whole project is mainly divided into three parts. Each phase took around 2 months. The first part is concentrated on the literature review and prepare the knowledge about machine learning (detail will be included in Section 7).

- Literature review about bisimulation
- Literature review about machine learning on logic
- Learn to use packages
- Plan and design the project

The second part is to develop a bisimulation algorithm and a test cases generator.



(a) Accuracy

(b) Loss

(c) Precision

(d) Recall

Figure 4.19: Metrics for test training sample

- Review relevant work on bisimulation
- Implement the standard algorithm
- Develop the visualisation tool
- Test and adjust the developed algorithm
- Develop the test case generator
- Test and adjust the generator

The last part is to construct the machine learning algorithm and experiment.

- Review relevant work on the logical learning in neural network
- Learn to use TensorFlow
- Implement the neural network
- Test the network and adjust the hyper-parameters
- Experiment on different cases (to be seen in Section 5)

For the timeline of the project, Gantt-chart is given below in Figure 4.20.

4.4 Environment

Hardware

2012 Mac mini computer, with processor Core i5 (i5-3210M), 8GB RAM is used during the developing of the project. When comes to the experiment, it is run on the Google Cloud Platform with 2 standard vCPUs (virtual CPU), 7.5 GB RAM and one NVIDIA Tesla K80 GPU.

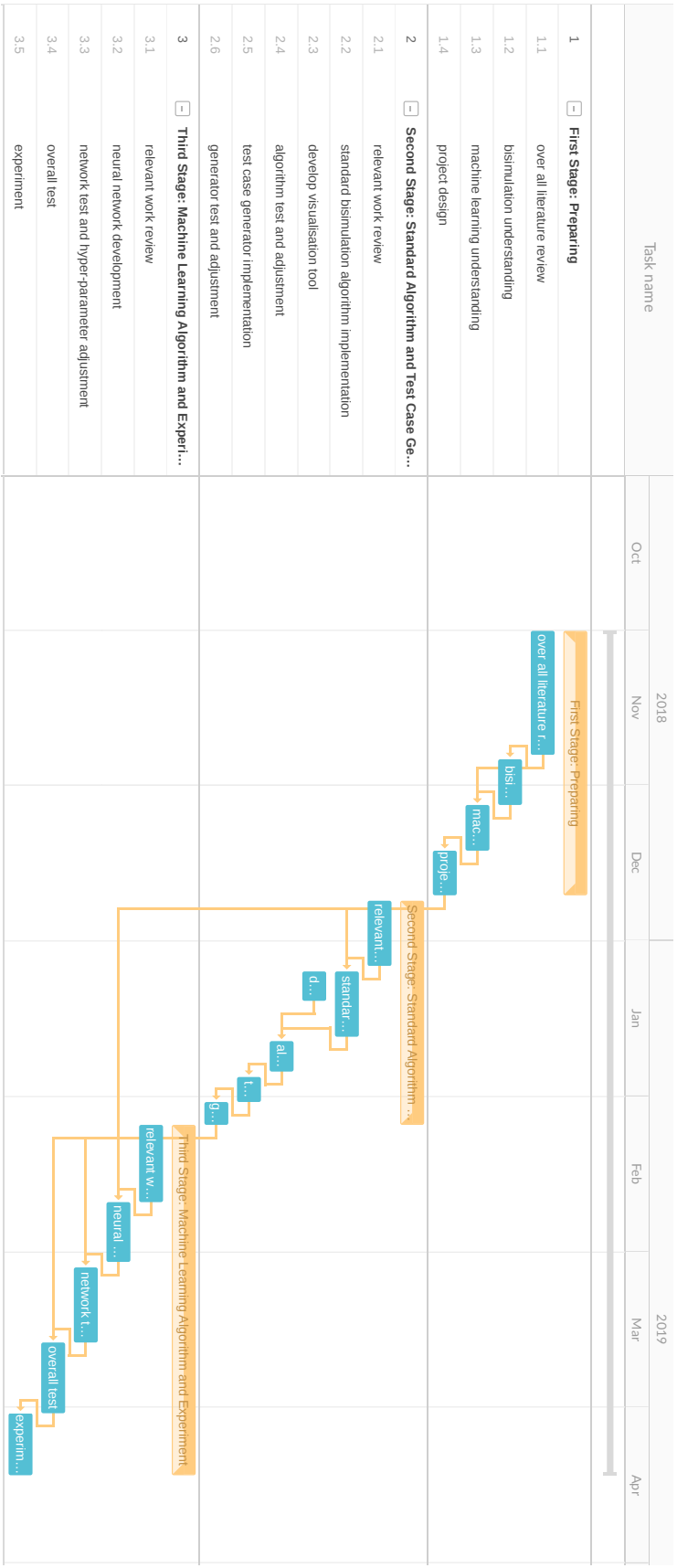


Figure 4.20: Gantt Chart

Software

This project is developed on `PyCharm 2018.3.6`, under `macOS Mojave 10.14`. For the main packages/library used in the project, `NetworkX 2.2` is used to manipulate the graph objects. The visualisation of the graphs is based on `Matplotlib 2.2.3`. Neural network and relevant training is based on the `TensorFlow 1.12.0` and `TensorFlow-GPU 1.12.0`. For the experiment the project is running under `Ubuntu 16.04.6 LTS`.

5 Experiment

In this section, the design, realization and the result of experiments will be stated. Due to the relatively large scale test is needed, experiments are deployed on Google Cloud Platform. In addition, the experiments run on `TensorFlow-GPU 1.12.0` rather than CPU version, which is used during the development. Consequently the result there may be slightly different.

5.1 Design

The aim of the experiments is to explore the capability of the network for understanding the logical feature. The experiment is designed into two parts. The first part is designed to test the stability of the training with respect to all the possible graphs with the same scale. So that the reliability of a single test can be estimated. The reason to estimate the reliability of a single test is that extensive and complete experiments are not feasible when the graph is big, due to the exponential growth of possible graph pairs. However, even for relatively small graph, generate all possible cases would be very time-consuming. And that is also the reason why generate-all-generator was abandoned (see Section 4.1.2). Instead, sampling is chosen as an alternative plan. Random test case generator can be seen as a sampling machine. By generate multiple groups of graphs with different density, the sampling is expected to cover most of the combinations. Specifically, the experiment parameters will be shown in Listing 5.1.

Listing 5.1: Wide experiments arguments

```
# ========== wide experiments ==========
# ---- ARGUMENTS ----
nodes_num = 5
edge_type_num = 3
c_type = random
case_number = 10000
graph_density = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
p_rate = 0.5
learning_rate = 0.1
epochs = 500
batch_size = 100
# --------------------
```

The second part of the experiments is aiming to find the limit of this network, namely the algorithm designed in this project (see Figure 3.2). In this part, the algorithm will be trained on graphs with different scale, i.e. bigger graphs. And by comparison, the limitation of the network can be estimated. Notice that for graphs with different scale, the network will be slightly different, i.e. only the length of the input layer and re-represent layer (see Figure 3.2) will be different. However, the overall structure will be the same. Here are the arguments used in this part (see Listing 5.2).

Listing 5.2: Deep experiments arguments

```
# ========== deep experiments ==========
# ---- ARGUMENTS ----
nodes_num = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
edge_type_num = 3
c_type = random
case_number = 10000
graph_density = 0.5
p_rate = 0.5
learning_rate = 0.1
epochs = 500
batch_size = 100
# --------------------
```

5.2 Results

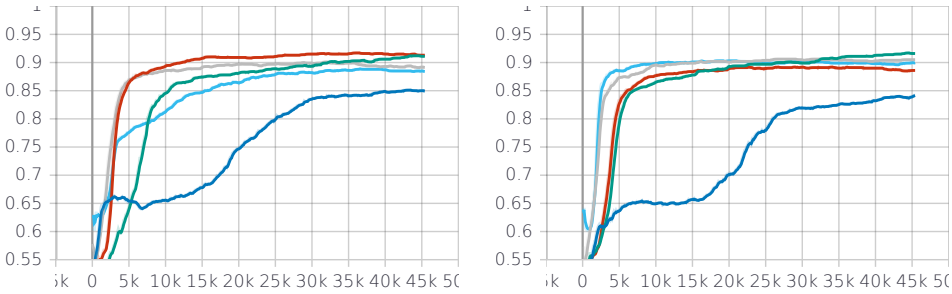
All the test will be run for multiple times to avoid potential coincidence. They will be uploaded including the full raw data of training, trained model and all the metrics logged during the training. Here we only give two results to make sure the characteristic can be identified easily.

Wide Experiment

For the wide experiment, the accuracy and loss of two duplicate experiments are given in Figure 5.1. Notice that in order to make graphs more clear, only the cases generated with `graph_density = 0.1, 0.3, 0.5, 0.7, 0.9` (see Listing 5.1) are shown in the graphs. Also, all the metrics here is base on the test set rather than the training set. Each line in the chart stands for one training. The legend shows the parameter of the train. For example, `random_n5_e3_10000_gd0.1_p0.5_test` means the corresponding training data is generated by the random generator. The graphs represented has 5 nodes, 3 types of edge with an average 0.1 graph density. And this accuracy is calculated base on the test data set. Notice that "graph density" used here is not a strict mathematical concept. It only stands for the degree of how dense the graph is (see Algorithm 3.2). The size of the test case is 10000. The rate between negative cases and positive cases is around 0.5.

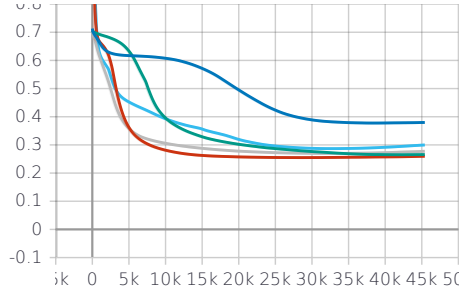
The first conclusion is that this kind of neural network can learn the logical concept like bisimulation. Most cases showed approximately 85% accuracy when 500 epochs end. This reveals the train is successful and the network actually catches the concepts of bisimulation. Compare all training, the graphs with 0.1 as density (i.e. most sparse) are harder for the network to be trained. Intuitively, the graphs with more link can express more complicated logic. And should be harder for the network to learning the feature. However, it may because the dense graph will show less complicate bisimulation propriety. In other words, when the graph becomes denser, the bisimulation pattern may go more simple. Concretely, for a dense graph, the mini bisimilar equivalent graph will most likely be either completely same (e.g. Figure 4.15) or be one node with self-loop of all possible link (e.g. Figure 4.16). For the first situation, two bisimilar graphs will be isomorphic (i.e. nodes of two graph are connected in the same way, only the sequence of the nodes is different). In this situation, the problem turns to distinguish if two graphs are isomorphic, which is much easier. For the second situation, the graph can be represented in a graph with one node. In this situation, the graphs are bisimilar may be diverse. But graphs that are not bisimilar may have a big difference which will make the task easier.

The wide experiment can guide the deep experiment. The result of wide experiment indicates that using random test case generator sample form the space of possible graphs is

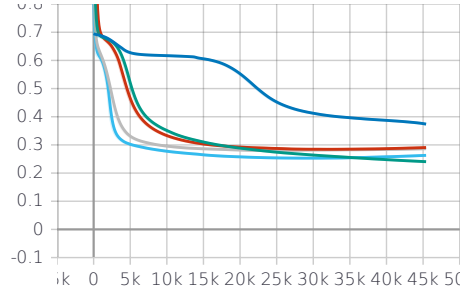


(a) Wide experiment 1: accuracy

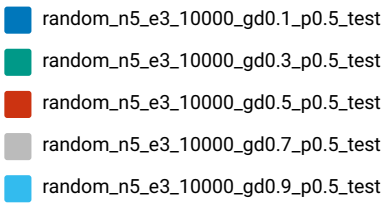
(b) Wide experiment 2: accuracy



(c) Wide experiment 1: loss



(d) Wide experiment 2: loss



(e) Legend

Figure 5.1: Result of wide experiment 1 and 2

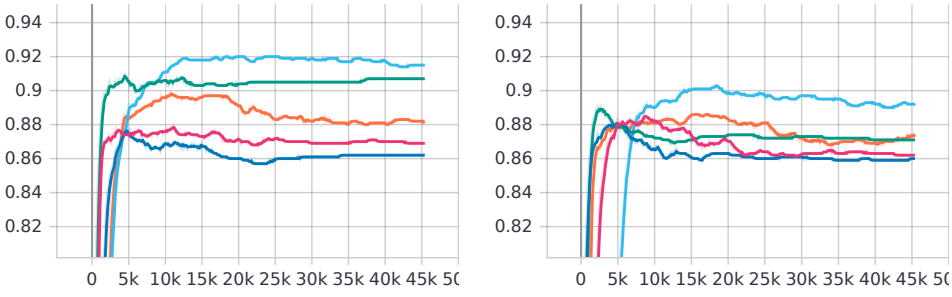
practicable (i.e. the performance of the algorithm when feeding dataset generated by one generator is representative). Because, except the one training (i.e. generator has 0.1 as density), most training show similar results. It reveals that the result of training for graphs with a certain scale can be represented by the training that bases on the data generated by a random test case generator.

Deep Experiment

The purpose of the deep experiment is to test the capability of the neural network. Thus in this experiment, graphs used here will be larger (i.e. has more nodes while maintaining the same number of edge type). According to the learning curve and where the accuracy converges, the capability of this neural network can be estimated.

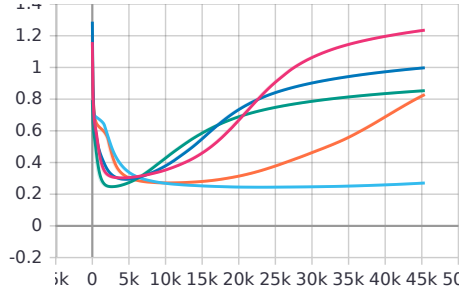
In this experiment, only the number of nodes for the generated graphs will change (see Listing 5.2). And the `graph_density` will be 0.5. Since in the wide experiment, the test cases generated with 0.5 graph density can better represent the overall leaning states (see Figure 5.1).

The result is shown in Figure 5.2. Same as the wide experiment, in order to make graphs more clear, only the cases generated with `nodes_num = 6, 8, 10, 12, 14` are shown in the graphs. The model show the sign of overfitting, i.e. the learner start to learn very specific

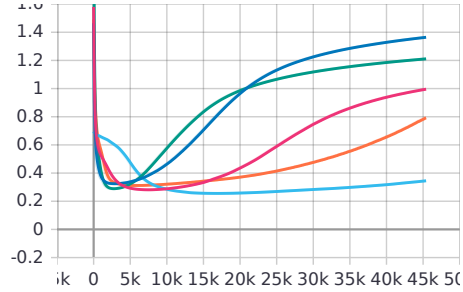


(a) Deep experiment 1: accuracy

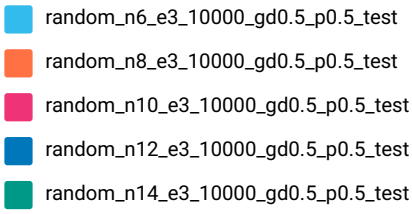
(b) Deep experiment 2: accuracy



(c) Deep experiment 1: loss



(d) Deep experiment 2: loss



(e) Legend

Figure 5.2: Result of deep experiment 1 and 2

random features rather than the target feature. Specifically, in the process of training, the loss of training set decreases while the loss on the test set is increasing. Similarly, the accuracy becomes better on the training set, but worse on the test set.

While it also shows some interesting result which is unexpected. For the aspect of training result, the value of accuracy will be compared. Considering the overfitting, the accuracy will drop when training performed too much. And for different training cases, the point of overfitting is different. To reduce the effect of the overfitting, the peak of each line will be taken for comparison. Form Table 5.1 and Figure 5.3, except for the case with 14 and 13 nodes, other cases show that accuracy is negatively correlated with the node number. In other words, when the graphs are larger, it is harder for the network to infer bisimulation. However, this correlation is not very significant. The differences between them are only around 10%. This may be caused by the small difference of the network. Because, when facing different size of graphs, the input layer will increase the size in order to handle it. It enhances the processing power of the network. Yet there is still abnormality when it comes to the case of 13, 14 nodes. This may be because of the quantity of the data. Because the number of the possible graphs pairs explode from $2^{1}50$ to $2^{1}176$. And the quantity of the data is still 10000 groups. Regarding the random generator as a sampling machine, for a much bigger space, the average difference of the two graphs took randomly will greater. But the bisimilar pairs generated will be much similar. Therefore, if the quantity of sampling is

greater, the result may be more "normal". Since there will be more misleading cases.

From the training process, the graph with fewer nodes is hard to be learned. The accuracy of the graph with least nodes (i.e. `random_n6_e3_10000_gd0.5_p0.5_test`) always converge slower than other cases. But the accuracy of graph with most nodes (i.e. `random_n14_e3_10000_gd0.5_p0.5_test`) always increase fast than other. Usually, the network with more nodes usually is considered more complicated. And will need more time/epochs to learn. However, in this case, the target learned is the bisimulation concept. Much the same as the reason explained above, deficient of data may cause a big difference between negative and positive cases, which will lead to higher learning efficiency.

Generally speaking, the wide experiment proves that the performance of learning algorithm feeding with dataset generated by one random generator can be representative respect to the performance of all possible data space with the same scale. In addition, the deep experiment did indicate that there may be a limitation of this neural network. However, there is still uncertainty exists. Once the scale of graphs become bigger enough (i.e. bigger than 12) the situation becomes "abnormal". It may be coursed by the insufficient quantity of data. And it still needs more experiments.

Nodes number	Accuracy experiment 1	Accuracy experiment 2
5	0.917	0.888
6	0.920	0.902
7	0.900	0.865
8	0.898	0.885
9	0.883	0.889
10	0.878	0.884
11	0.889	0.888
12	0.874	0.880
13	0.892	0.906
14	0.906	0.889

Table 5.1: Peak accuracy of each line

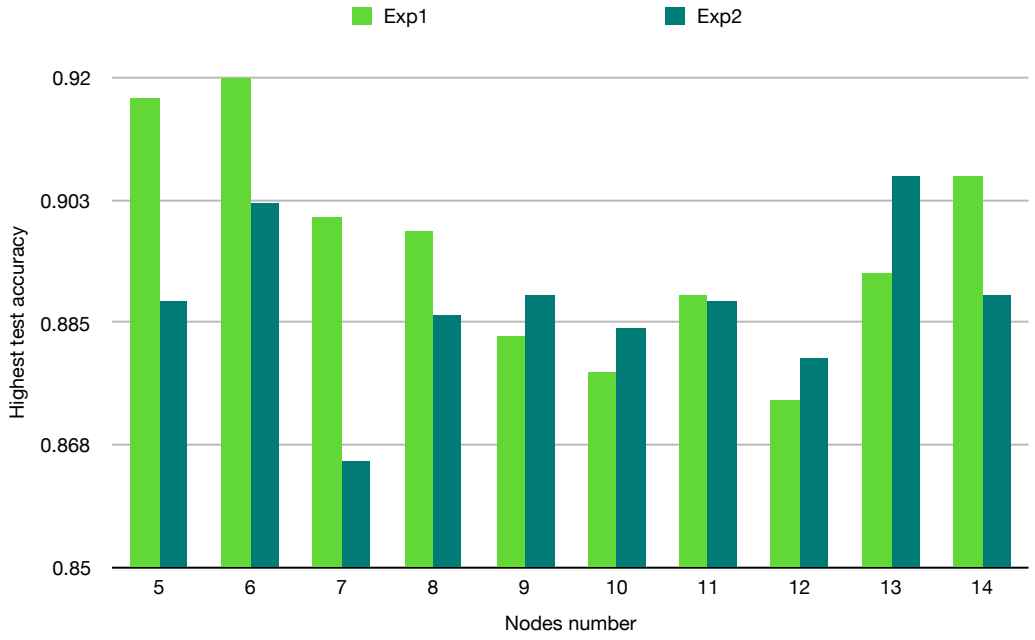


Figure 5.3: Chart of test accuracy

6 Evaluation and Summary

From the engineering aspect, the project is generally good and gives two command line application as research tools. One can generate same scale graphs pairs with the bisimulation flag that ready to be used as data set for machine learning. And the scale of the graph, number of pair, the rate of positive and negative, graph density can be customised. Other can use the data generated to training a preset neural network. In this tool, the number of training epoch, learning rate, batch size, the rate of test set and training set division can be customised. And the training can be continued after the program ends. The whole project was pushed as expected. All components were tested and work well. Nevertheless, the process of the project still needs improvement. The most important is the test of components. It was done by hand, which not reliable and inefficient. Most test data from the textbook is reliable but not enough in quantity.

On the other hand, form aspect of research, the goal is generally achieved. Whereas, it opens the next stage of research. There are two experiments done in this project. First is the wide experiment. It tests the performance of the learning algorithm when feeding the data (mainly) from different parts of space (i.e. all possible same scale graphs). The experiment shows that in most situation the performance of the algorithm is similar. Base on it, one can say that except the extreme situation (i.e. graph density equals 0.9) the performance of the algorithm when feeds a non-extreme value, is representative respect to the graphs pair with the same scale. Second is the deep experiment. It is aimed at testing the capability of the preset neural network. It will test the performance of the learning algorithm when feeding the data with different scale (i.e. node number) Base on the conclusion of the wide experiment, only the dataset generated with one density will be used to represent the result of the whole space in this very scale. And the result shows that when the graphs become bigger, the performance did become worse, which indicate the limitation of the neural network. However, when the scale keeps expanding to a certain level, the performance unexpectedly become better. It may be because of the insufficient quantity, but there still needs more experience.

Still, there are many things need to be improved as a research project. Firstly, the literature review needs to be more at the begin of the project. At the start of the project, most literature reviewed is about bisimulation. The main aim was overlooked during the development of the first part. It caused problems when the project stepped into the second part. Some components need to be rewritten in order to serve the second part. Second, the experiment design should be carefully considered. For the wide experiment, the dataset used can be better by mixing all graphs pairs generated on different density then sampling from the pool. For the deep experiment, the criteria of the deep experiment is quite blurred. And the result indicates that more experiments are needed.

Moreover, for further experiments a better test cases generator may need to be designed. The generator designed and implemented in the project aimed at “random”. In other words, the generator was designed to be as random as possible. However, in fact, true random generated graphs are mostly like the graphs shown in Figure 4.15 and Figure 4.16. They are either very easy to be denied or very simple to be recognised (see Section 5.2). Concretely, it is possible that 80% of graphs pairs belongs to the above two types. In that situation, the network will only need to learn to distinguish these two types of graphs, the performance can reach over 80% accuracy while lacking generalisation ability. Thu, the better idea may be a generator that can generate all kinds of pair while decreasing the incidence of these two types. And generative adversarial network (GAN) can be a good choice.

7 Learning Points

In this section all the materials and skill had been learning will be listed. During the development of this project, many things had been learned, including the theories, programming skills and research skills.

- **Theories**
  - Bisimulation
    - Online Course: System Validation: Automata & behavioural equivalences [22]
    - Thesis reading, about modal logic (e.g. [5, 23]), model checking (e.g. [8]), bisimulation (e.g [13, 21, 24]) and relevant algorithm (e.g. [10, 12])
    - Understand the bisimulation and the algorithm
  - Machine learning
    - Online Course: Machine Learning by Andrew Ng [25]
    - Thesis reading, about neural network (e.g. [15]), machine learning on logic (e.g. [2, 26])
- **Programming Skills**
  - programming in `Python` with `PyCharm`
  - Using `TensorFlow` to construct a neural network with visualisation, i.e. `TensorBoard`
  - Using GPU to train the neural network
  - Using package like `matplotlib, graphviz` to visualise the graphs
  - Using `Git` and `Github` to manage the project
  - Using remote computing server i.e. Google Cloud Platform
  - Deploying experiments on `Linux`
  - Using LaTeXto typeset the dissertation
- **Research Skills**
  - Searching relevant thesis
  - Reading, understanding and reproducing the algorithm of the thesis especially [10].
  - Scheduling the project with Gantt-chat
  - Analysis of information from different sources

Nevertheless, except the skills had been learned. There are also many skills that need to be improved or mastered. The most important one is the skill to find the information. At the first stage of the project, the research focus on the bisimulation. Meanwhile, the literature about the neural network on logical learning was not collected and reviewed sufficiently, which affected later study on machine learning. The literature review should cover the whole project before actually working on it, rather than reviewing partially.

8 Professional Issues

In this section the professional Issues base on the codes of practice and conduct issue by the British Computer Society will be discussed.

Technical Competence:

At very first, trying to maintain the technical competence, all course provided by the university is taking. Further some extra online courses about the project like Machine Learning (see Section 7) and etc. has been taken during the project. Also, before using the newly learned skill or some new design structure, it will be discussed during the regular meeting with the supervisor. For example, the design scheme of the test case generator and the structure of the neural network were all discussed before actual implementation.

Public Interest:

Data of this project did not involve any issue of public privacy, security and etc. directly. All the data used here is self-generated abstract mathematical concepts. Any information related to intellectual property is annotated and used legitimately. The softwares and libraries used either open source (e.g `TensorFlow`) or use free education version (e.g. `PyCharm`).

Duty to Relevant Authority:

In the process of the project, the Relevant Authority's instruction is and will be respected and acted. Section 9 of the Code of Practice on Assessment and University's Code of Practice is understood and acted honestly, ethically and professionally, during the whole process of the project. Also, there is not any misrepresent information on the performance of any products involved in this project.

Duty to the Profession:

Many pieces of literature relevant to the project had been read to maintain the specialism. After the evaluation of this project, it is planned that the code of this project will be open-source under permission. Thus, knowledge can be shared freely and support other researchers.

A Appendices

A.1 User Manuel

Installation

To run these tools, the depend packages need to be installed.

```
$ pip install -r requirements.txt
```

Notice that, it is recommend to installed in virtual environment.

Usage

These tools is developed on `Python 2.7`.

For quick test, please run the command directly without any parameters.

```
$ python ml_algorithm/ml_algorithm.py
$ python standard_bisim/test_cases_generator.py
```

To run the wide and deep experiments:

```
$ python experiments.py
```

Follows are specific direction:

```
$ python ml_algorithm/ml_algorithm.py -h

usage: ml_algorithm.py [-h] [-e EPOCH] [-l LEARNING_RATE] [-b BATCH_SIZE]
                       [-p DATA_PATH] [-r TEST_TRAIN_RATE]
                       [-c CONTINUE_TRAIN] [-n MODEL_NAME]

optional arguments:
  -h, --help            show this help message and exit
  -e EPOCH, --epoch EPOCH
                        Number of training epochs
  -l LEARNING_RATE, --learning_rate LEARNING_RATE
                        Initial learning rate
  -b BATCH_SIZE, --batch_size BATCH_SIZE
                        Number of data for one batch
  -p DATA_PATH, --data_path DATA_PATH
                        Path to input data
  -r TEST_TRAIN_RATE, --test_train_rate TEST_TRAIN_RATE
                        The rate of test cases and train cases
  -c CONTINUE_TRAIN, --continue CONTINUE_TRAIN
                        Continue last training
  -n MODEL_NAME, --model_name MODEL_NAME
                        The name of the model
```

```
$ python standard_bisim/test_cases_generator.py -h

usage: test_cases_generator.py [-h] [-t {random,all}] [-n NUMBER]
                               [-f FILE_NAME] [-v NODE_NUMBER]
                               [-e EDGE_TYPE_NUMBER] [-r P_RATE]
```

```
                              [-p PROBABILITY]

optional arguments:
  -h, --help            show this help message and exit
  -t {random,all}, --type {random,all}
                        Type of data set
  -n NUMBER, --number NUMBER
                        The length of data set
  -f FILE_NAME, --file_name FILE_NAME
                        Name of the output file
  -v NODE_NUMBER, --node_number NODE_NUMBER
                        Number of the nodes of the graph in the data set
  -e EDGE_TYPE_NUMBER, --edge_type-number EDGE_TYPE_NUMBER
                        The total types of the edge in the graphs
  -r P_RATE, --p_rate P_RATE
                        Rate of the positive cases over all cases
  -p PROBABILITY, --probability PROBABILITY
                        The density of the random generate graphs
```

Here is the directory tree of the whole project.

```
FYP
    Bisimulation_fyp_2019
        experiments.py
        ml_algorithm
        requirements.txt
        standard_bisim
    FYP_report
        img
        main.tex
        reference.bib
        tex
    experiment_data
        deep
        wide
```

To visulise the performance of machine learning please use `TensorBoard`

```
$ tensorboard --logdir <path-to-summary-folder> --host localhost
```

References

[1] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.

[2] S. Hölldobler, Y. Kalinke, and H.-P. Störr, "Approximating the semantics of logic programs by recurrent neural networks," *Applied Intelligence*, vol. 11, no. 1, pp. 45–58, 1999.

[3] S. Hölldobler, Y. Kalinke, F. W. Ki, F. Informatik, and T. Dresden, "Towards a new massively parallel computational model for logic programming," in *In ECAI'94 workshop on Combining Symbolic and Connectioninst Processing*, pp. 68–77, 1991.

[4] D. Sangiorgi, "On the origins of bisimulation and coinduction," *ACM Transactions on Programming Languages and Systems*, vol. 31, pp. 1–41, may 2009.

[5] J. van Benthem, *Modal Correspondence Theory*. Phd thesis, Universiteit van Amsterdam, 1976.

[6] R. Milner, "Operational and Algebraic Semantics of Concurrent Processes," jan 1990.

[7] D. Park, "Concurrency and automata on infinite sequences," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 104 LNCS, (Berlin/Heidelberg), pp. 167–183, Springer-Verlag, 1981.

[8] A. W. Roscoe, "Model-checking CSP," in *A classical mind : essays in honour of C.A.R. Hoare*, pp. 353–378, Prentice Hall, 1994.

[9] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980.

[10] R. Paige and R. E. Tarjan, "Three Partition Refinement Algorithms," *SIAM Journal on Computing*, vol. 16, no. 6, pp. 973–989, 1987.

[11] J. L. Bell, "Aczel Peter. Non-well-founded sets. With a foreword by Jon Barwise. CSLI lecture notes, no. 14. Center for the Study of Language and Information, Stanford 1988, also distributed by the University of Chicago Press, Chicago, xx + 131 pp.," *The Journal of Symbolic Logic*, vol. 54, pp. 1111–1114, sep 1989.

[12] A. Dovier, C. Piazza, and A. Policriti, "An efficient algorithm for computing bisimulation equivalence," *Theoretical Computer Science*, vol. 311, pp. 221–256, jan 2004.

[13] R. van Glabbeek, "Bisimulation," *Encyclopedia of Parallel Computing*, pp. 136–139, 2011.

[14] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*, vol. 53 of *Adaptive computation and machine learning series*. Cambridge, MA : MIT Press, 2012., 2013.

[15] J. E. Dayhoff, *Neural network architectures : an introduction*. New York: Van Nostrand Reinhold, 1990.

[16] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," tech. rep., Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[17] P. Hitzler, S. Hölldobler, and A. K. Seda, "Logic programs and connectionist networks," *Journal of Applied Logic*, vol. 2, no. 3, pp. 245 – 272, 2004. Neural-symbolic Systems.

[18] M. Guillame-Bert, K. Broda, and A. d'Avila Garcez, "First-order logic learning in artificial neural networks," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, July 2010.

[19] O. Ray and B. Golénia, "A neural network approach for first-order abductive inference."

[20] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, PMLR, 11–13 Apr 2011.

[21] C. Stirling, "Bisimulation and Logic," in *Advanced topics in Bisimulation and Coinduction* (D. Sangiorgi and J. Rutten, eds.), pp. 173–196, Cambridge: Cambridge University Press, 2011.

[22] J. F. Groote, "System Validation: Automata and behavioural equivalences — Coursera," 2016.

[23] P. Blackburn, M. de. Rijke, and Y. Venema, *Modal Logic: Graph. Darst.* Cambridge University Press, 2001.

[24] K. G. Larsen and A. Skou, "Bisimulation through probabilistic testing," Tech. Rep. 1, 1991.

[25] A. Ng, "Machine Learning — Coursera," 2015.

[26] M. Leshno, "On the Computational Power of Neural Networks and Neural Automata," in *International Neural Network Conference*, pp. 1013–1013, 2013.