Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning
Special Session on KR and Machine Learning

A Graph Neural Network Reasoner for Game Description Language

Alvaro Gunawan[1], **Ji Ruan**[1], **Xiaowei Huang**[2]
[1]Auckland University of Technology,
[2]University of Liverpool
{alvaro.gunawan, ji.ruan}@aut.ac.nz, xiaowei.huang@liverpool.ac.uk

Abstract

General Game Playing (GGP) aims to develop agents that are able to play any game with only rules given. The game rules are encoded in the Game Description Language (GDL). A GGP player processes the game rules to obtain game states and expand the game tree search for an optimal move. The recent accomplishments of AlphaGo and AlphaZero have triggered new work in extending the neural network approach to GGP. In these works, the neural networks are used only for optimal move selection, while the components dealing with GDL still use the logic-based methods. This motivates us to explore if a neural network based method would be able to approximate the logical inference in GDL with a high accuracy. The structured nature of logic tends to be a difficulty for neural networks, which rely heavily on statistical features. Inspired by the recent work on neural network learning for logical entailments, we propose a neural network based reasoner that is able to learn logical inferences for GDL. We present three key contributions: (i) a general, game-agnostic graph-based representation for game states described in GDL, (ii) methods for generating samples and datasets to frame the GDL inference task as a neural network based machine learning problem and (iii) a GNN based *neural reasoner* that is able to learn and infer various game states with high accuracy and has some capability of transfer learning across games.

1 Introduction

General Game Playing (GGP) (Love et al. 2008) aims to develop agents that are able to play any game with only the rules given. The game rules are encoded in Game Description Language (GDL), a logical language that is a variant of Datalog with function symbols and a few known keywords. In order to play games, the players in GGP parse the game rules using logic-based inferences to obtain game states and expand the game tree to search for an optimal move. These inferences are typically done through Prolog or PropNet (propositional networks) (Schkufza, Love, and Genesereth 2008) implementations. The recent accomplishments of AlphaGo (Silver et al. 2016) and AlphaGo Zero (Silver et al. 2017b) use Monte-Carlo Tree Search (MCTS), deep neural networks and learning through self-play to achieve superhuman performance in Go. The generality of this method is extended in AlphaZero(Silver et al. 2017a) to also allow the agent to play Chess and Shogi effectively. However, AlphaZero is limited in that it does not automatically extend to play other games without additional human information and the game types are always assumed to be two-player, turn-taking and zero-sum. In the domain of GGP, an agent must be able to account for games with any number of players, simultaneous action and non-zero-sum, without the aid of human information. GDL serves the purpose of representing such games in a flexible way.

In this paper, we propose methods for a neural network based reasoner that is able to learn logical inferences for GDL. As GDL allows flexible representation of various games, the reasoner has to handle game states with various sizes and features; e.g., a state in Tic-Tac-Toe is quite different to a state in Connect Four. Typical neural network architectures such as multilayer perceptrons and convolutional networks would not be able to capture the general nature of the different state features of various games without introducing unintended biases.

To overcome this, we propose graph-based representations of GDL rules and game states, using a Graph Neural Network (GNN) based architecture for inference. This combination provides an avenue for the development and implementation of GNN based reasoners that are able to learn to infer the rules of general games. We present three key contributions: (i) a general, game-agnostic graph-based representation for game states described in GDL, (ii) methods for generating samples and datasets to frame the GDL inference task as a neural network based machine learning problem and (iii) a GNN based *neural reasoner* that is able to learn and infer various game states with high accuracy and has some capability of transfer learning across games.

The rest of the paper is organised as follows: Section 2 gives an overview of the different components of our neural reasoner. Section 3 describes a general representation for game states in GGP, and in particular we propose *instantiated rule graphs*. Section 4 introduces our proposed GNN architecture that is able to handle the instantiated rule graphs and outputs the predicted legal actions and next fluents. Section 5 presents and discusses the results of our neural reasoner across a variety of games as well as a mixture of games. The neural reasoner is evaluated across a set of games to investigate its generalisation capability. In particular, we find that our reasoner can achieve 100% accuracy in several games and over 80% for the majority. The design of the GNN architecture also allows us to transfer learned

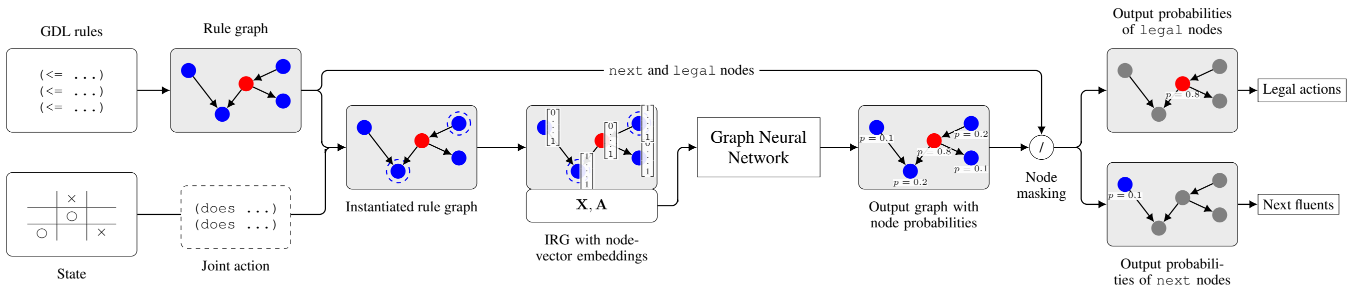


Figure 1: Overview of the Neural Reasoner

knowledge from one game to a different game which we demonstrate with experimental results showing the neural reasoner's transfer learning capability. We discuss in Section 6 related work in graph neural networks and GGP. Finally, we discuss the limitations of the approaches we have presented and conclude with possible future directions for research.

2 Overview of the Neural Reasoner

We first show the inference tasks in GGP that our neural reasoner learns to approximate and then present an overview of the different components in the neural reasoner. Games in the GGP setting are described using Game Description Language (GDL), a logical programming language based on Datalog that can describe game states and mechanics with logical rules. Figure 2 shows an example fragment of the description for game Tic-Tac-Toe. The left rule states that in the next state, `(cell 1 1 o)` is true if `oplayer` does action `(mark 1 1)` and `(cell 1 1 b)` is true in the current state. The right rule states that it is legal for `oplayer` to take the action `(mark 1 1)` if `(cell 1 1 b)` and `(control oplayer)` are true in the current state. A full GDL description of a game will consist of many other rules to define the full dynamics of the game.

```
(<= (next (cell 1 1 o))            (<= (legal oplayer (mark 1 1))
    (does oplayer (mark 1 1))          (true (cell 1 1 b))
    (true (cell 1 1 b)))               (true (control oplayer)))
```

Figure 2: Fragment of GDL description of Tic-Tac-Toe.

A game state described with GDL consists of a set of dynamic predicates called *fluents*, such as `(cell 1 1 o)`, `(cell 1 1 b)` `(control oplayer)`. A subsequent state (next-state) can be logically derived by the combination of rules, fluents present in the current state and the actions made by the players. Suppose we have a game G, defined as a set of rules in GDL, two key inference tasks are to find out *(1) what are the legal actions for the players at the current state and (2) what is the next state if the agents make a joint move*. Task (1) can be formalized as the following: to find out for each player i, its legal move m at the current state s, as in a logical representation $G \wedge s \models legal(i, m)$. There can be multiple legal actions in a state. Task (2) can be formalized as the following: to find out all the fluent f that holds in the next state, given each player $i \in [1, k]$ doing a move m_i at the current state s, as in a logical representation $G \wedge s \wedge does(1, m_1) \wedge ... \wedge does(k, m_k) \models next(f)$.

Figure 1 gives an overview of the different components in our neural reasoner for GGP. We start with a game described as a set of rules in GDL. It is converted into a rule graph, which is built upon the dependency of the fluents or other predicates in such rules. The rule graph is then combined with a game state into an instantiated rule graph (IRG). We apply a node-vector embedding to the nodes of the IRG to prepare it as input of the Graph Neural Network component. The output of the GNN component is a rule graph with node probabilities. Node masking is then applied based on node type to extract the probabilities of the `legal` and `next` nodes, using information from the input rule graph. During training, the output probabilities are compared to the ground truth training targets to calculate the error for backpropagation. Finally, the nodes with output probabilities greater than a threshold are selected as the legal actions and next fluents.

The training samples, generated by a logical reasoner, are in the form of the pairs $((G, s), S_{\text{legal}})$ for Task (1) and $((G, s, Does), S_{\text{next}})$ for Task (2), where G is a game, s the current state, S_{legal} the set of legal actions, $Does$ the joint action set, and S_{next} the set of fluents true in the next state. Once the graph neural network is trained, we then use it to predict the legal actions and next fluents. In the next two sections, we will provide more details on these components.

3 Game State Representations

A key element of applying neural networks to games is finding a state representation that allows for effective feature extraction, while also being computationally efficient. Typically, these representations consists of a vector, matrix or tensor based structure, depending on the type of neural network architecture used. For example, in Chess or Go, the game board can be directly translated into a matrix, with elements representing the presence of different pieces on the board. These matrices can then be used directly as input to a convolutional neural network, as the grid-structure of the matrix retains the positional information of the various pieces. Although they are effective and efficient, these vector or matrix based representations are "fixed" in size for each game and are not compatible across different games.

As different games described in GDL may have varying sets of possible fluents, a simple vector or matrix is not capable of representing the general features of various game states. Consider the games Tic-Tac-Toe and Connect-Four: one could represent various GDL states by constructing a one-hot vector representing all possible fluents that could be present in a state, such as `cell(1, 1, o)`, `cell(1, 2, x)` and so on. However, as Tic-Tac-Toe and Connect-Four have different numbers of fluents (28 and 127 respectively), these vectors would have different sizes. Likewise, if we were to instead represent a state by mapping its board positions to a matrix, these matrices would also have different sizes. A multilayer perceptron or convolutional neural network would have a fixed size for its input layer and would not be able to take both Tic-Tac-Toe and Connect-Four states as input without additional preprocessing or padding, which may introduce unintended biases. This means a different neural network needs to be trained for each game separately.

In this section, we propose a more general "game-agnostic" representation that suits the domain of GGP better. This representation will be able to fully capture the features present in states, while still allowing for states of various games to be valid inputs for the same neural network. The basis of this representation is a *rule graph*, a graph-based representation of game rules first introduced in (Kuhlmann and Stone 2007), initially used to identify similarity in game descriptions. The rule graph captures the relational inductive biases present in the game rules. We first give the details of rule graphs in section 3.1 and then propose to extend the rule graphs to be a general state representation in section 3.2.

3.1 Rule Graphs

Rule graphs are coloured, directed graphs consisting of four types of nodes: keyword nodes, predicate nodes, label nodes and label argument nodes. *Keyword nodes* represent logical and relational sentences consisting of a predefined GDL keyword such as `legal`, `next`, `does`, etc. *Predicate nodes* represent sentences consisting of non-keyword nodes, i.e., custom defined predicates declared specifically for the game. In the fragment in Figure 1, the predicates `mark` and `cell` are examples of non-keyword predicates. *Label nodes* are used to identify which predicates are the same throughout the description without keeping explicit or specific lexical labels. This is done by drawing an edge from label nodes to each predicate which is an instance of said label. Finally, *label argument nodes* are used to identify the position of arguments in a predicate, encoding the positional information of predicates into the rule graph. Similarly to label nodes, an edge is drawn between the label argument node to each instance of an argument given its position. Additionally, variable nodes can also be included, however for our purposes the game descriptions are initially grounded which instantiates all variables. Edges are drawn between predicates and their arguments, constructing a syntax tree.

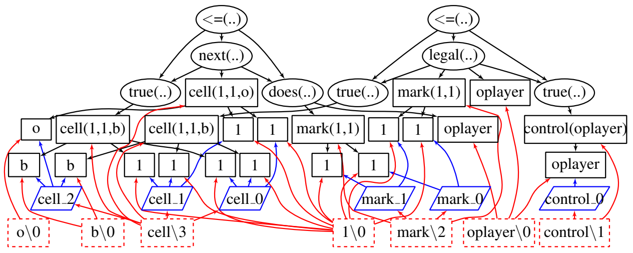


Figure 3: Rule graph of the fragment in Figure 2. Lexicographic information such as predicate names are not stored and only shown here to aid in visualisation.

Figure 3 shows the rule graph generated from the rule fragment presented in Figure 2. Oval nodes are keyword nodes. Rectangular nodes are predicate nodes. Red nodes are label nodes. Blue nodes are label argument nodes. Lexical labels are only for visualisation and are not retained in the actual rule graph. To generate a rule graph from a game description G, we follow the method in (Schiffel 2010):

- Ground all rules to eliminate all variables in G.
- For each logical sentence, relational sentence and predicate in G, create a node in the graph. E.g.,
  - The rule graph in Figure 3 consists of nodes for each keyword and predicate present in the fragment in Figure 2: the keyword `legal` is represented by an oval keyword node and the predicate `(control oplayer)` is represented by a rectangular predicate node.
- Add an edge between each term's node to all the term's argument nodes. For the backwards implication node, add edges from the head node to the body nodes. E.g.,
  - The node `legal(oplayer,mark(1,1))` has edges to its arguments, nodes `oplayer` and `mark(1,1)`. Aditionally, as it is the head argument of a backwards implication node, it has edges to the body nodes `true(control(oplayer))` and `true(cell(1,1,b))`.
- For each unique non-keyword predicate in the game description, create a label node and an edge to each node that represents an instance of that predicate being used. E.g.,
  - As the predicate `mark` is used in both rules, the red `mark\2` label node has edges to each instance of the predicate.
- For each label node's predicate with arity i, create i label argument nodes, representing each possible argument's position. Add an edge from the label node to each corresponding label argument node. Add an edge to each node that represents an argument with the corresponding label argument node according to its position. E.g.,
  - The blue `mark\2` node has two label argument nodes: `mark_0` and `mark_1`.

As the rule graphs only retain the relations and positions of the various predicates without storing actual predicate names, the representation is isomorphic to predicate scrambling. Two different descriptions of the same game that have their predicate names changed would have the same rule graph representation. Additionally, similar features defined in different games would have similar subgraphs within their rule graph representations.

3.2 Instantiated Rule Graphs

Although GDL provides a representation for rules that are general across different games, it does not provide a general *state* representation. Likewise, the rule graphs discussed in the previous section are still only a representation of game rules and are not a general representation of game states. To overcome this limitation, we present *instantiated rule graphs*, a general state representation generated by instantiating the rule graph of a game, localising it to a specific state by providing a unique node labelling for each state.

Definition 3.1. Instantiated Rule Graph. Given a rule graph $R = (V, E)$ and a state $S = \{f_1, f_2, \dots, f_n\}$ consisting of fluents f_i, an instantiated rule graph is a graph $I = (V, E, L)$ where V, E are the same as in rule graph R and $L_S : V \rightarrow \{false, true\}$ is a labelling function based on state S and the following requirements:

For node n in rule graph R and current state S:

- If node n is a predicate node that corresponds to a fluent in state S and parent node n_p is a `true` node with edge $(n_p, n) \in E$, then nodes $L_S(n) = true$, $L_S(n_p) = true$ and $L_S(n_c) = true$ for all children nodes given $(n, n_c) \in E$.
- For all other nodes, $L_S(n_o) = false$.

Figure 4 shows the rule graph from Figure 3 instantiated to a state S where fluent `cell(1,1,b)` $\in S$. Nodes labelled $true$ are shaded in blue. All other nodes are labelled $false$. A benefit of the instantiated rule graphs is that any modifications or improvements can be made to the labelling function without loss of generality across all games.

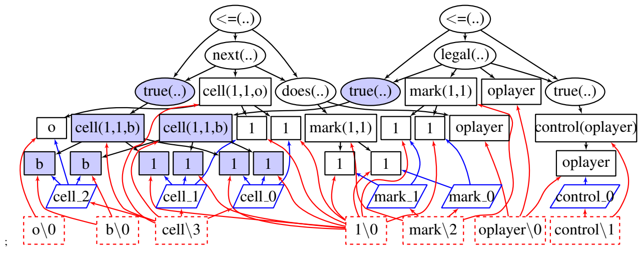


Figure 4: An example of an instantiated rule graph

4 Neural Network Architecture

This section outlines the neural network architecture used with instantiated rule graphs. Firstly we introduce our strategy of embedding the node features into a vector. Secondly, we provide details of the graph neural network based architecture. Thirdly, we present a general method of generating datasets for different games. Finally, we provide details on the training of the neural network itself.

4.1 Node-Vector Embeddings

To prepare the instantiated rule graph as input for a neural network, we convert the labelling and the node types into a vector embedding. For node $n \in V$ in instantiated rule graph $I = (V, E, L)$, we construct a vector embedding $\vec{v}^n \in \{0, 1\}^{19}$. This embedding provides a general fixed

Index	Node Embedding
1	True in state (label)
2	Constant symbol
3	Variable symbol
4	Predicate
5	Variable

(a) Label and non-keywords

Index	Node Embedding
6	`<=`
7	`not`
8	`or`
9	`distinct`
10	`does`
11	`goal`
12	`init`
13	`legal`
14	`next`
15	`role`
16	`terminal`
17	`true`
18	`input`
19	`base`

(b) Keywords

Table 1: Vector embedding values

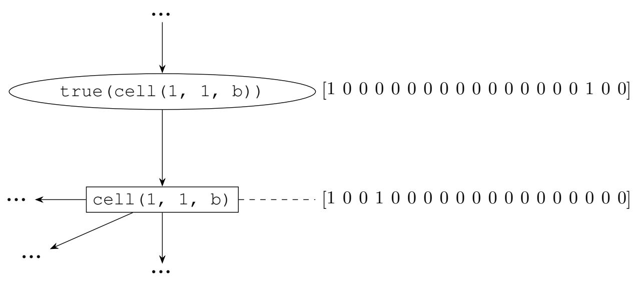


Figure 5: Example vector embedding of two nodes of the instantiated rule graph from Figure 4.

length vector that can represent without ambiguity the various types of nodes and labelling present in an instantiated rule graph. The first value of the vector directly corresponds to the labelling function, with $true$ and $false$ represented by values 1 and 0 directly. The rest of the vector acts as a one-hot encoding representing the type of node, with the first four corresponding to the non-keyword node types and the rest corresponding to the keywords present in GDL. Table 1 shows the node feature that each value $\vec{V}_{1\dots19}$ corresponds to.

Figure 5 shows an example of a vector embedding of the `cell(1, 1, b)` nodes in the instantiated rule graph from Figure 4. Note that as the node is a predicate node, the value of $\vec{v}^n_4 = 1$. The node `true(cell(1, 1, b))` is a `true` keyword node, resulting in the value of $\vec{v}^n_{17} = 1$. Both nodes are labelled $true$ by the labelling function, so both vectors will have the value $\vec{v}^n_1 = 1$.

The vector embeddings of all nodes in the graph are then stacked to create node feature matrix $\mathbf{X} \in \{0, 1\}^{|V| \times 19}$. Alongside the adjacency matrix of the graph $\mathbf{A}$, the node feature matrix $\mathbf{X}$ is used as the input for the graph neural network described in the next section.

4.2 Graph Neural Network Architecture

As the game state representation we have discussed in the previous section is defined as a graph, we use a graph neural network. The overall architecture of our GNN, shown

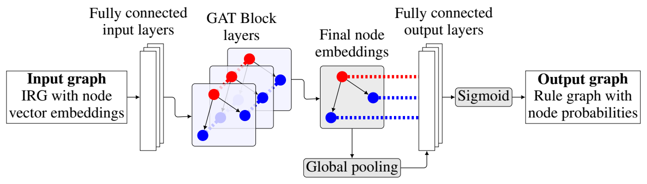

Fully connected input layers
GAT Block layers
Final node embeddings
Fully connected output layers
Input graph
IRG with node vector embeddings
Sigmoid
Output graph
Rule graph with node probabilities
Global pooling

Figure 6: Architecture of Our Graph Neural Network

in Figure 6, consists of three distinct layers: an initial set of fully connected input embedding layers, a set of 3 GAT Block layers and finally a set of fully connected output layers. The GAT Blocks consist of two pairs of Graph Attention Networks (GAT) (Veličković et al. 2018) acting as bi-directional edge layers, a skip connection and ReLU nonlinearity.

An input IRG with node vector embeddings is first passed through a set of fully connected input layers, attending only to individual node embeddings. Then, the graph is passed through a set of GAT Blocks, propogating messages to update the node embeddings. After the last GAT Block, final node embeddings are generated. Global soft attention (Li et al. 2016) is applied across the final node embeddings to pool into a graph-level embedding. This graph-level embedding is combined with each individual final node embedding and passed through the fully connected output layers. Lastly a sigmoid activation is used to provide the final output node probabilities.

We chose the Graph Attention Network architecture for the graph neural network layers as they provided the best balance between performance and memory usage. Experiments with other architectures such as Graph Convolutional Networks (Kipf and Welling 2017) exhibited worse performance, while more complex architectures such as Directed Acyclic Graph Neural Networks (Thost and Chen 2021) restricted the size of games that were usable while providing minimal improvement in performance.

4.3 Dataset Generation

To train the neural network, we require a dataset of training samples containing game states and the inferred legal actions and next fluents. Using the definition of IRGs and node-vector embeddings, we propose a general method of generating training datasets for a game described in GDL.

To generate a dataset, the GDL description of the selected game G is used to play out random games. As new states are visited throughout the game plays, we store the states as instantiated rule graphs and use a Prolog based reasoner to generate ground truth logical inferences for the states as the training target. While inferring the legal actions only requires the fluents of a state, inferring the fluents present in the next state additionally requires the actions selected. To account for this, we additionally store a randomly selected legal joint action in a given state. The training targets for each example is the truth valuations of the `legal` and `next` rules for a given state or state and joint action pair respectively. These correspond directly to the `legal` and `next` nodes present in the IRG. The task of the neural network is then to learn to approximate a function that converts each initial node-vector embedding into an output probability. As such, the dataset can be seen as a partial node classification task across multiple graphs.

Game	IRG size		Outdegree		Mean path
	Nodes	Edges	Max	Mean	length
blocker	5087	12201	559	2.40	4.23
connectfour (C4)	19672	47221	2249	2.40	4.33
connectfour3p (C4_{3p})	17076	42623	1731	2.50	4.25
hamilton	14963	31516	1601	2.11	5.27
hanoi6disks	28055	70237	4098	2.50	4.98
knightstour	12291	30154	1393	2.45	4.45
parallelbuttonsandlights	1602	3245	182	2.03	4.93
tictactoe (TTT)	4810	11031	591	2.29	4.19
doubletictactoe (TTT$_D$)	7379	16338	803	2.21	4.61
tictactoelarge (TTT$_L$)	3553	8185	323	2.30	4.69
connectfour-flat	21450	51477	2543	2.40	4.29
doubletictactoe-flat	27280	64342	2583	2.36	4.52
tictactoelarge-flat	4203	9735	423	2.32	4.60

Table 2: Graph metrics of game datasets

In our implementation, a game description G is used to generate dataset $\mathcal{D}_G$ consisting of training examples of the form $((\mathbf{X}, \mathbf{A}), \vec{l})$ for legal actions and $((\mathbf{X}_a, \mathbf{A}), \vec{n})$ for next fluents. For a given state S and joint action $a = does(1, m_1) \wedge \ldots \wedge does(k, m_k)$, $\mathbf{X}$ is the node feature matrix of IRG $I = (V, E, L)$ of S and $\mathbf{X}_a$ is the the node feature matrix of the IRG $I_a = (V, E, L_a)$ of S with `does` nodes labelled $true$ according to actions a in addition to the standard fluent labelling. $\mathbf{A}$ is the adjacency matrix of IRGs I and I_a. The training targets $\vec{l} = \{0, 1\}^{|V|}$ and $\vec{n} = \{0, 1\}^{|V|}$ are the target output probabilities of the nodes V that the network must predict. As $\mathbf{X}_i$ corresponds to node i, if node i is a `legal` node, $\vec{l_i} = 1$ if the valuation of the legal rule represented by node i is $true$ and $\vec{l_i} = 0$ otherwise. Likewise, for `next` node j, $\vec{n}_j = 1$ if the valuation of the next rule represented by node j is $true$ and $\vec{n}_j = 0$ otherwise.

The above method is general and can be applied to any game described using GDL. For the experiments in Section 5, we generate 12000 training examples for each game. Although the datasets are generated on a per-game basis for simplicity, multiple datasets can be combined for multi-game mixed training or sequential training.

Table 2 shows some metrics of the IRGs present in the game datasets generated. Although the IRGs are directed graphs, we calculate the average path length by treating them as undirected graphs as the graph neural network architecture is able to propagate messages along both directions separately. While the size of the IRG gives a sense of the complexity of the game description, it is distinct from the state complexity of the game itself. These metrics highlight the discrepancy between game state complexity and game description complexity. For example, while TTT$_L$ has a larger number of game states compared to standard TTT due to its larger board size, the IRG for TTT$_L$ is smaller in both the number of nodes and edges. These metrics correspond roughly to the complexity of the game description, as the IRG size corresponds directly to the number and length of rules present in the description, while the average path length corresponds to the average distance of logical dependencies present in the rules. In the end of the table, G-flat refers to an equivalent game of G, which we examine later in Section 5.4.

We have only generated the datasets for 10 different games which are typically used in GGP tournaments, but there are many other game descriptions that exist in various repositories such as the Stanford [1] and Dresden[2] GGP repositories. Explanations of game rules and example game plays can also be found on these repositories. Descriptions for more complex games like Chess and Go are also available, but due to hardware limitations we avoid using these games for our experiments. Additional game descriptions can be written for games that do not have a description already available. This presents an opportunity to provide an extensive benchmark for the inference tasks in GDL, which we encourage others to use in further graph neural network research.

4.4 Training Details

We train the neural network using the cross-entropy loss criterion for both legal action and next fluent prediction. Only `legal` and `next` node output probabilities are used in the calculations for the cross-entropy loss, with other nodes masked out. These other nodes contribute to the loss values implicitly through the message propagation mechanism in the GAT layers and the global pooling operator. The Adam optimiser (Kingma and Ba 2015) is used with a base learning rate of 0.0001 and dropout is applied to the fully connected layers during training. Additionally, we use Pairnorm (Zhao and Akoglu 2020) normalisation throughout the graph neural network layers.

The neural reasoner is trained on datasets generated from various games with 10% of each dataset reserved as a validation split. A batch size of 32 examples are used for most games, however for larger game descriptions a smaller batch size is used due to hardware limitations. Once the network training is complete, threshold values for `next` and `legal` node probabilities are selected. For each task, threshold values $0.1, 0.15, 0.2, \dots, 0.9$ are used to discretise the output probabilities, selecting the threshold value that provides the lowest mean squared error between the discretised output probabilities and the examples from the validation set. These selected threshold values are used for each game during evaluation.

5 Results and Discussion

We conduct a series of experiments to evaluate the performance of the neural reasoner on a collection of games and to investigate training behaviour in a variety of settings and learning modalities. Section 5.1 presents the performance of the neural reasoner trained on individual games. Section 5.2 investigates the neural reasoner's ability to transfer its learned knowledge across to different games as well as a mixed training scheme with multiple games being learned simultaneously. Section 5.3 compares the mixed training scheme with a sequential training scheme. Finally, Section 5.4 present a solution to the issues faced with unflattened rules that causes poorer performance in certain games.

All experiments are conducted on an Intel Xeon Silver 4210 CPU with 96 GB of memory running Ubuntu 20.04.2 LTS and 4 NVIDIA GeForce RTX2080 Ti. The implementation is written in Python using the PyTorch Geometric library (Fey and Lenssen 2019) and PySwip (Tekol and contributors 2020).

[1]http://ggp.stanford.edu

[2]http://general-game-playing.de

5.1 Individual Game Experiments

As the networks are trained to predict `legal` actions and `next` fluents, we measure the accuracy of the neural reasoner over 100 randomly played games. For a given input state, the predicted legal actions and next fluents are given by the neural reasoner's output node probabilities that are greater than the selected threshold. This is then compared to the ground truth which is inferred with a Prolog based reasoner.

Game	Next fluents (%)	Legal actions (%)
parallelbuttonsandlights	84.35%	100.00%
tictactoe (TTT)	100.00%	100.00%
tictactoelarge (TTT$_L$)	100.00%	63.69%
doubletictactoe (TTT$_D$)	100.00%	91.96%
connectfour (C4)	93.58%	100.00%
connectfour3p (C4$_{3p}$)	95.50%	91.75%
blocker	88.94%	100.00%
knightstour	100.00%	100.00%
hamilton	94.94%	100.00%
hanoi6disks	87.55%	72.66%

Table 3: Neural reasoner accuracy over 100 games.

Table 3 shows the accuracy of the neural reasoner on 10 different games, over 100 game plays on each. Across all games, the neural reasoner shows fairly high accuracy, achieving 100% at several games and at least 80% accuracy in most games. Notable exceptions are TTT$_L$ and hanoi6disks at legal prediction, which we investigate further in Section 5.4.

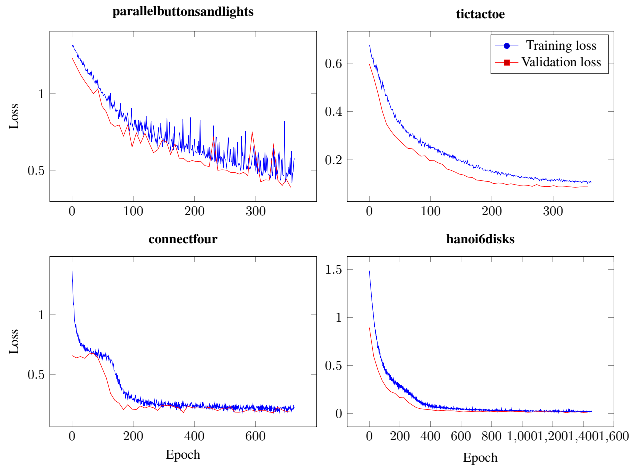


Figure 7: Training loss for parallelbuttonsandlights, tictactoe, connectfour and hanoi6disks.

Figure 7 shows that the training behaviour across four different games. The differences exhibited here also contribute to the varying levels of accuracy that the neural reasoner achieves for different games. Dropout is not applied when calculating the validation loss, leading to the the lower values. The differences in epochs correspond to smaller batch sizes due to hardware limitations as mentioned earlier in Section 4.4.

5.2 Transfer and Mixed Learning

To evaluate the neural reasoner's transfer learning capabilities, we run evaluations across the following games: Tic-Tac-Toe (TTT), Connect-Four (C4), Double Tic-Tac-Toe (TTT$_D$) and Tic-Tac-Toe Large (TTT$_L$). We evaluate the neural reasoner on games it has not learned. Additionally, we conduct mixed training, where the reasoner is trained on all four games simultaneously with a uniform random mixture of game states. All networks are initialised with random weights. Table 4 shows the results of the training conducted. Interestingly, the results have shown the potential to achieve "zero-step generalisation" as the network is able to achieve 100% accuracy on several games that it was not trained for, which ideally shows that the network is learning to reason GDL more generally. However we can see that in several cases this transfer does not occur, such as the network trained on TTT$_L$ which shows poor performance across all other games.

The network trained on all games jointly shows balanced performance across the board - notably it outperforms the C4 trained network on C4 itself on `next` prediction and achieves equal or comparable performance in other games. This shows that mixed training can be used to train a neural reasoner that can reason multiple games simultaneously with high accuracy.

5.3 Sequential vs Mixed Training

While the mixed-game training shows that the neural reasoner can achieve similar accuracy to its solely trained counterpart, the method in which the examples are mixed might not be practical or well-suited for all game-playing scenarios. In particular, we would like to see if a *sequential training* scheme is possible. Rather than mixing the examples from various games into a single dataset, the network is trained on the different game datasets sequentially. There are two questions this raises: Firstly, does the sequential training cause the network to "forget" the earlier trained games? Secondly, can the prior training on other games assist the later training, leading to faster convergence? To answer this, we train a neural reasoner on the game sequence TTT$\rightarrow$TTT$_D$ $\rightarrow$ TTT$_L$ $\rightarrow$ C4 and compare its accuracy to mixed training.

Table 5 shows the accuracy of the sequentially trained neural reasoner compared to the mixed training reasoner. The sequentially trained neural reasoner shows similar performance to the mixed trained neural reasoner. This shows that the network does not "forget" its learned reasoning on earlier games after training on the other games. Like with the mixed neural reasoner, the network shows balanced performance across the games it is trained on when compared to the individually trained networks as in Section 5.2.

However, as Figure 8 shows, the sequentially trained neural reasoner exhibits much more instability during training as can be seen in the fluctuating validation loss. The sequentially trained network is still able to achieve final loss values equivalent to the mixed training counterpart despite the instability. The random mixing of various game states likely acts as a type of regularisation, whereas the sequential training can potentially lead to overfitting on each individual game before moving on to the next. Regardless, sequential training presents an alternative training scheme when mixing is not possible or impractical.

Training method		TTT	TTT$_D$	TTT$_L$	C4
Mixed	Next	100.00%	100.00%	100.00%	98.00%
	Legal	100.00%	82.62%	55.80%	100.00%
Sequential	Next	100.00%	100.00%	92.52%	93.63%
	Legal	100.00%	82.35%	64.45%	100.00%

Table 5: Neural reasoner accuracy in mixed and sequential trainings.

Train dataset	TTT	TTT$_D$	TTT$_L$	C4
TTT	100.00%	100.00%	100.00%	93.45%
TTT$_D$	100.00%	100.00%	94.77%	93.53%
TTT$_L$	40.66%	41.06%	100.0%	40.92%
C4	100.00%	94.68%	92.49%	93.58%
Mixed	100.00%	100.00%	100.00%	98.00%

(a)

Train dataset	TTT	TTT$_D$	TTT$_L$	C4
TTT	100.00%	1.35%	56.02%	100.00%
TTT$_D$	91.36%	91.96%	64.66%	89.71%
TTT$_L$	59.02%	14.18%	63.69%	12.85%
C4	100.00%	1.39%	56.47%	100.00%
Mixed	100.00%	82.62%	55.80%	100.00%

(b)

Table 4: Neural reasoner accuracy over 100 games for (a) `next` and (b) `legal` prediction.

5.4 Rule Flattening

Of note in Section 5.1 was the relatively poor performance of the variants of TTT, namely TTT$_D$ and TTT$_L$, when compared to default TTT. We discovered that a potential cause

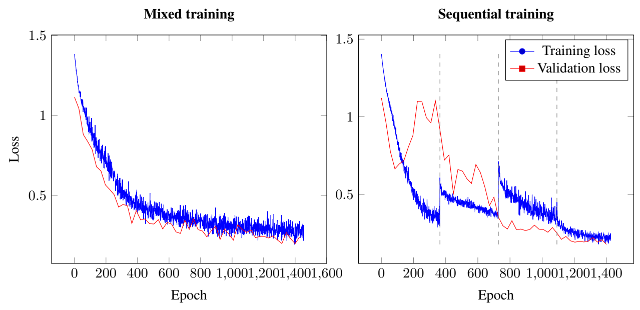


Figure 8: Comparison of mixed training and sequential training. Dashed lines show dataset transitions.

for this disparity is in the way some rules are defined. In TTT, all of the legal and next rules are defined in a *flat* manner, with the heads (legal or next predicates) directly depending on fluents that represent the basic features of the states. However, in other games such as TTT$_D$ and TTT$_L$, several of the legal and next predicates have a longer dependency on fluents via other intermediate predicates.

For example, in the rule fragments shown in Figure 9, the left two rules are from the standard description of TTT$_L$. The legal predicate `legal(.)` in the first rule depends on the predicate `emptycell(.)`, which is not a fluent and further depends on the fluent `cell(.)` in the second rule. While in the rule on the right, the legal predicate `legal(.)` directly depends on the fluent `cell(.)`. We give simple flattening as a process analogous to substitution, inserting the body of the second rule into the first rule. In particular, `emptycell(.)` is replaced with its corresponding body. This process shall retain the equivalence of the games.

```
(<= (legal xplayer (mark 1 1 x))        (<= (legal xplayer (mark 1 1 x))
    (true (control xplayer))                (true (control xplayer))
    (emptycell 1 1))                        (index 1)
                                            (index 1)
(<= (emptycell 1 1)                         (not (true (cell 1 1 x)))
    (index 1)                               (not (true (cell 1 1 o))))
    (index 1)
    (not (true (cell 1 1 x)))
    (not (true (cell 1 1 o))))
```

Figure 9: Left: Standard definition of rules in TTT$_L$. Right: Flattened version of the same rules

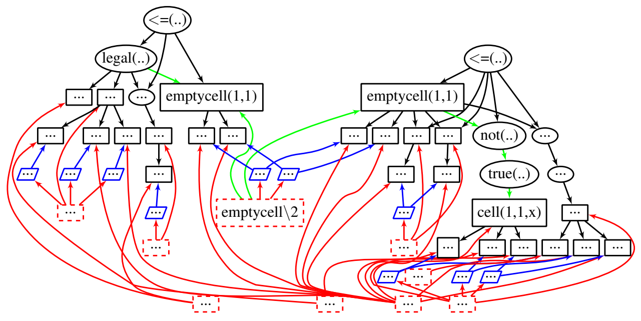
Figure 10 shows the rule graphs of the unflattened and flattened rules, highlighting the effect of this process: the dependency path from the `legal(..)` node to the fluent `cell(1,1,x)` node reduces from 6 in unflattened version to 3 in the flattened version. This shortened dependency between nodes can be seen when comparing the average path lengths of the unflattened and flattened versions of the same game in Table 2 (last three rows), with all the flattened versions showing shorter average paths lengths than their unflattened counterparts.

To test whether the poor performance seen in some games is caused by the behaviour seen in unflattened rules, we train and evaluate the neural reasoner on the flattened versions and compare the performance to the standard, unflattened versions. Table 6 shows the improved accuracy of the neural reasoner trained on flattened rules compared to the standard unflattened rules, on the three games.

These results suggest that the distance of the dependency of legal and next predicates to the fluents is an important factor in our representation and flattening the rules could reduce the distance and potentially increase overall performance of the neural reasoner. Some limitation of this approach is that a systematic automated flattening process can be difficult, e.g., with recursively defined rules and exponential growth of the number of the rules. Note that game hanoi6disks also exhibited a large number of unflattened legal rules, which could not be easily flattened.

Game	Next fluents accuracy	Legal actions
TTT-L (standard)	100.00%	63.69%
TTT-L (flat)	100.00%	100.00%
TTT-D (standard)	100.00%	91.96%
TTT-D (flat)	100.00%	100.00%
C4 (standard)	93.58%	100.00%
C4 (flat)	100.00%	100.00%

Table 6: Terminal prediction for 100 games of standard and flattened games



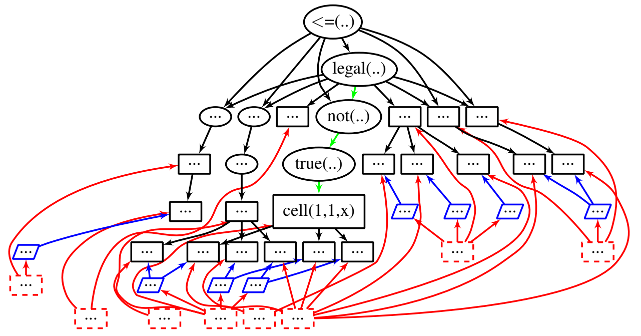


Figure 10: Rule graphs of unflattened (top) and flattened (bottom) fragment from Figure 9. Labels have been removed for clarity.

6 Related Work

Work in the 90s and early 2000s on the modelling of logic programming using neural networks (Garcez and Zaverucha 1999; Abdullah 1992; Garcez et al. 2002) laid the foundations of neuro-symbolic computing - the synthesis of logic and neural networks. Early work in the field rely on directly translating the logical statements to the neural network structure itself.

Recently, there has been a resurgence in applying neural network based methods to logical domains, following their successes in computer vision and natural language processing. This more recent work focuses on using the logical statements themselves as input to the neural networks, allowing for more general neural network architectures. However, the structured nature of logic tends to be a difficulty for neural networks, which rely heavily on statistical features. Evans et al. introduced a dataset of logical entailment examples to train and evaluate neural network architectures on (Evans et al. 2018), as well as their model PossibleWorldNet which was able to outperform several benchmarks. Rawson and Reger (Rawson and Reger 2020) were able to further extend this by taking advantage of the relational inductive bias present in the logical dataset. Their work presented a

GNN-based architecture and a graph based encoding for the logical statements, which combined was able to outperform PossibleWorldNet in several of the categories of the dataset.

Beyond the two mentioned earlier, many other researchers have applied graph neural networks to the logical domain (Thost and Chen 2021; Abdelaziz et al. 2021; Glorot et al. 2019; Paliwal et al. 2020; Crouse et al. 2019; Olšák, Kaliszyk, and Urban 2019). Most work in the logical domain focus on tasks such as entailment and automated theorem proving, which typically uses much smaller graphs as the logical formulae consist of fewer clauses than an entire GDL game description. Additionally, these tasks are typically formulated as a graph classification problem, which differs from our approach, which is more akin to a node classification problem across multiple graphs.

Some recent work such as (Silver et al. 2020; Lin et al. 2022) has applied graph neural networks to planning problems. However, most work in this field typically use graphical representations of the objects in the environment and their relationships instead of directly modelling a logical description. As such, the GNNs are tasked to learn the importance and probabilities of the objects themselves, rather than learning to infer legal actions or transition dynamics.

Some of the most notable recent work applying neural network based approaches to games is that of AlphaGo, AlphaGo Zero and AlphaZero (Silver et al. 2016; Silver et al. 2017b; Silver et al. 2017a), which were able to achieve super-human performance on Go, Chess and Shogi. The networks were used for policy and value estimation and trained through self-play. Although the architecture presented in AlphaZero was general across the three games, the networks had were specific for each game and had to be trained separately due to the varying state representations. Several recent work (Goldwaser and Thielscher 2020; Gunawan et al. 2020) have extended the methods in AlphaZero to the domain of GGP and shown their effectiveness. The neural networks are used similarly as in AlphaZero for optimal move selection and are specific for individual games too. There are two main differences in our work. Firstly our graph neural networks are able to handle the game states more generally. Secondly, we use neural networks in approximating the logical reasoning, instead of the move selection.

Alongside the initial introduction of rule graphs by Kuhlmann and Stone (Kuhlmann and Stone 2007), they have also investigated other applications of transfer learning in GGP. In their work rule graphs are used to identify similar games, to which they transfer known value mappings for Q-learning. Schiffel (Schiffel 2010) further extends this work and uses rule graphs for symmetry detection within games. As these make use of standard rule graphs, they do not apply their methods to individual game states. Additionally, they do not make use of graph neural networks, instead using more traditional approaches such as graph isomorphism algorithms.

7 Limitations

The current implementation of instantiating the rule graph as described can be improved both in expressiveness and efficiency. Firstly, the most effective labelling function for instantiating the rule graph is still not known and the current implementation can be improved for a more expressive representation. Secondly, the current instantiation method requires the game description to be grounded, which is a computationally expensive process that can lead to extremely large graphs. It is possible that only targeted rules need to be grounded instead, as the rule graph representation is capable of representing ungrounded variables.

A key limitation we have discovered is the poor performance of the neural reasoner on games that are described using unflattened rules. We found that by flattening these rules, the neural reasoner was able to improve its accuracy. However, as mentioned before, this flattening process is resource intensive. Furthermore, there are several games where this flattening is not possible without completely rewriting the rules. More generally, this issue is likely due to the graph neural network architecture being unable to make inferences across long-range dependencies. This also includes other relational structures such as negation, which can be seen as a two-hop dependency.

Currently, we have only trained the neural reasoner to predict legal actions and next fluents. To implement a complete GDL reasoner, we also need to infer whether a state is terminal and the goal values of each player when the state is terminal. We leave this as an open problem as there are issues that arise from the unbalanced nature of the dataset (there are more non-terminal states than terminal states) as well as the limitations faced due to unflattened rules (most terminal and goal rules are unflattened).

8 Conclusion and Further Work

In this paper we have presented a method to approach GDL reasoning with neural networks in a general manner. The translation of the GDL rules and game states to a graph-based representation allows for the application of graph neural networks that are able take as input various games without resorting to game-specific networks. We have implemented a neural reasoner that is able to learn to infer the legal actions and next fluents in various games, with high accuracy in most of them. Furthermore, we show that with the general instantiated rule graph representation and graph neural network architecture, we are able to train the neural reasoner on multiple games simultaneously as well as transfer across unlearned games. Sequential training is also possible: the neural reasoner is able to be trained on a sequence of various games without forgetting the earlier trained games. This is likely the most practical method of training a neural reasoner for use in a game player, as this allows the neural reasoner to learn new games as they are introduced to it. However, we have highlighted several limitations.

While the instantiated rule graph representation and the graph neural network architectures have been used to learn the reasoning task in this paper, further work could extend these methods to apply them to the task of game playing. Using a modified network architecture based on these approaches in an AlphaZero style self-learning agent would allow for the agent to learn and play multiple game simultaneously, in a truly general manner.

References

Abdelaziz, I.; Crouse, M.; Makni, B.; Austil, V.; Cornelio, C.; Ikbal, S.; Kapanipathi, P.; Makondo, N.; Srinivas, K.; Witbrock, M.; et al. 2021. Learning to guide a saturation-based theorem prover. *arXiv preprint arXiv:2106.03906*.

Abdullah, W. A. T. W. 1992. Logic programming on a neural network. *International journal of intelligent systems* 7(6):513–519.

Crouse, M.; Abdelaziz, I.; Cornelio, C.; Thost, V.; Wu, L.; Forbus, K.; and Fokoue, A. 2019. Improving graph neural network representations of logical formulae with subgraph pooling. *arXiv preprint arXiv:1911.06904*.

Evans, R.; Saxton, D.; Amos, D.; Kohli, P.; and Grefenstette, E. 2018. Can neural networks understand logical entailment? *arXiv preprint arXiv:1802.08535*.

Fey, M., and Lenssen, J. E. 2019. Fast graph representation learning with pytorch geometric. *CoRR* abs/1903.02428.

Garcez, Artur S d'Avila, A. S., and Zaverucha, G. 1999. The connectionist inductive learning and logic programming system. *Applied Intelligence* 11(1):59–77.

Garcez, A. S. d.; Broda, K.; Gabbay, D. M.; et al. 2002. *Neural-symbolic learning systems: foundations and applications*. Springer Science & Business Media.

Glorot, X.; Anand, A.; Aygun, E.; Mourad, S.; Kohli, P.; and Precup, D. 2019. Learning representations of logical formulae using graph neural networks. In *Neural Information Processing Systems, Workshop on Graph Representation Learning*.

Goldwaser, A., and Thielscher, M. 2020. Deep reinforcement learning for general game playing. In *AAAI*, 1701–1708.

Gunawan, A.; Ruan, J.; Thielscher, M.; and Narayanan, A. 2020. Exploring a learning architecture for general game playing. In *AI 2020: Advances in Artificial Intelligence - 33rd Australasian Joint Conference, AI 2020, Proceedings*, 294–306. Springer.

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In Bengio, Y., and LeCun, Y., eds., *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*.

Kuhlmann, G., and Stone, P. 2007. Graph-based domain mapping for transfer learning in general games. In *European Conference on Machine Learning*, 188–200. Springer.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. S. 2016. Gated graph sequence neural networks. In Bengio, Y., and LeCun, Y., eds., *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Lin, Y.; Wang, A. S.; Undersander, E.; and Rai, A. 2022. Efficient and interpretable robot manipulation with graph neural networks. *IEEE Robotics and Automation Letters*.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification.

Olšák, M.; Kaliszyk, C.; and Urban, J. 2019. Property invariant embedding for automated reasoning. *arXiv preprint arXiv:1911.12073*.

Paliwal, A.; Loos, S.; Rabe, M.; Bansal, K.; and Szegedy, C. 2020. Graph representations for higher-order logic and theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34(03), 2967–2974.

Rawson, M., and Reger, G. 2020. Directed graph networks for logical entailment. Technical report, EasyChair.

Schiffel, S. 2010. Symmetry detection in general game playing. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Schkufza, E.; Love, N.; and Genesereth, M. 2008. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *Australasian Joint Conference on Artificial Intelligence*, 56–66. Springer.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017b. Mastering the game of go without human knowledge. *nature* 550(7676):354–359.

Silver, T.; Chitnis, R.; Curtis, A.; Tenenbaum, J.; Lozano-Perez, T.; and Kaelbling, L. P. 2020. Planning with learned object importance in large problem instances using graph neural networks. *arXiv preprint arXiv:2009.05613*.

Tekol, Y., and contributors. 2020. PySwip v0.2.10.

Thost, V., and Chen, J. 2021. Directed acyclic graph neural networks. In *International Conference on Learning Representations*.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. *International Conference on Learning Representations*.

Zhao, L., and Akoglu, L. 2020. Pairnorm: Tackling oversmoothing in gnns. In *International Conference on Learning Representations*.