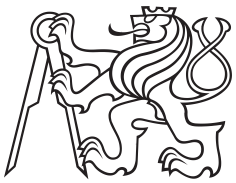
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Applications of Graph Neural Networks in Classical Planning

Bohdan Nazarenko

Supervisor: Ing. Rostislav Horčík, Ph.D.
Study program: Open Informatics
Specialisation: Artificial Intelligence and Computer Science
May 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Nazarenko Bohdan** Personal ID number: **498934**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Applications of Graph Neural Networks in Classical Planning

Bachelor’s thesis title in Czech:

Aplikace grafových neuronových sítí v klasickém plánování

Guidelines:

The thesis aims to implement a planner based on graph neural networks (GNNs). The student is expected to fulfill the following objectives:
1. Familiarize yourself with the language PDDL standard for planning task specifications (McDermott 2000) and the Fast Downward Planning System (Helmert 2006).
2. Survey and analyze the recent results (Ståhlberg, Bonet, and Geffner 2022) on policy learning for a planning domain using GNNs.
3. Implement a planner consisting of two components. The first component learns a policy for a given planning domain. The second one searches for a plan to solve a given planning task by exploiting the learnt policy.
4. Test the implemented planner on the domains from the International Planning Competitions. Compare your results with the results obtained in (Ståhlberg, Bonet, and Geffner 2022).

Bibliography / sources:

[1] McDermott, Drew M. 2000. “The 1998 AI Planning Systems Competition.” AI Magazine 21 (2): 35–56. https://doi.org/10.1609/aimag.v21i2.1506.
[2] Helmert, M. 2006. “The Fast Downward Planning System.” Journal of Artificial Intelligence Research 26: 191–246. https://doi.org/10.1613/jair.1705.
[3] Ståhlberg, Simon, Blai Bonet, and Hector Geffner. 2022. “Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits.” Proceedings of the International Conference on Automated Planning and Scheduling 32 (June): 629–37.

Name and workplace of bachelor’s thesis supervisor:

Ing. Rostislav Horčík, Ph.D. Department of Computer Science FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **09.01.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Rostislav Horčík, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Acknowledgements

I want to thank Ing. Rostislav Horčík, Ph.D., for giving me this topic, being willing to provide me help at any time, guiding, teaching, and for the feedback during whole development. I would also like to thank my family for helping me during my years of study.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 25. May 2023

Abstract

In this work, we study Graph Neural Networks (GNNs) application in Classical Planning appeared in [SBG22]. We reimplement and modify the original code and add new parts. We employ GNNs for learning optimal general policies in Classical Planning domains. We train GNN Models using supervised learning to approximate the optimal value function $V(s)$ for various sampled states s. We reproduce the results of the experiments conducted in [SBG22].

Keywords: GNN, Graph Neural Network, Classical Planning, Planning, Planner, Planning Domain Definition Language, PDDL, Artificial Intelligence, AI

Supervisor: Ing. Rostislav Horčík, Ph.D.
E-322, Charles Square 13, Prague 2

Abstrakt

V této práci studujeme aplikaci grafových neuronových sítí (GNN) v klasickém plánování, která se objevila v článku [SBG22]. Reimplementujeme a upravujeme původní kód, přidáváme nové části kódu. Využíváme GNNs pro učení optimálních obecných strategií v doménách klasického plánování. Trénujeme GNN modely pomocí učení s učitelem k aproximaci optimální hodnotové funkce $V(s)$ pro různé náhodně generované stavy s. Reprodukujeme výsledky experimentů provedených v článku [SBG22].

Klíčová slova: GNN, Grafová neuronová síť, Klasické Plánování, Plánování, Plánovač, PDDL, Umělá Inteligence, AI

Překlad názvu: Aplikace grafových neuronových sítí v klasickém plánování

Contents

Figures

Tables

Chapter 1

Introduction

In recent years, Graph Neural Networks (GNNs) have emerged as a powerful and rapidly evolving field of study. Researchers have been harnessing the potential of GNNs to tackle a wide range of complex problems across various domains. One noteworthy application is highlighted in the paper [SBG22], which focuses on utilizing GNNs to learn general optimal policies for solving Classical Planning problems.

In this bachelor's thesis, we delve into the methods and applications of GNNs in the context of Classical Planning, building upon the research presented in [SBG22]. Our primary objectives encompass reimplementing their existing code solutions, introducing new contributions, providing clear application explanations, and reproducing the experimental results documented in [SBG22]. The structure of the thesis is organized as follows.

Firstly, we provide an in-depth review of Classical Planning and Classical Planning problems. This section establishes a foundational understanding of the domain and the challenges in it.

Next, we delve into the Planning Domain Definition Language (PDDL), which serves as a language for describing Classical Planning problems. We also explore the Fast Downward Planner, an automated planning system that supports PDDL problems and generates solution plans with associated costs.

Subsequently, we revisit the fundamentals of Neural Networks, including the classical training loop used for training neural models. Additionally, we

introduce the architecture of Graph Neural Networks, specifically designed to handle relational structures.

Building upon this knowledge, we delve into the general pipeline of working with GNNs and discuss their specific application in Classical Planning, utilizing relational structures. We explore the algorithms deployed in this context, elucidating their functionality.

Following the theoretical groundwork, we briefly overview our code implementation. This section highlights the key components and functionalities of the implemented system, emphasizing the modifications and enhancements made to facilitate GNN-based planning.

Finally, we conduct two types of experiments to evaluate the effectiveness and performance of our GNN-based planning approach. These experiments aim to validate the contributions made in this thesis and shed light on the capabilities and potential of GNNs in solving Classical Planning problems.

By delving into the methods and applications of GNNs in Classical Planning, this thesis contributes to the growing body of research in the field. The findings and insights derived from our work can potentially advance state-of-the-art planning systems, opening avenues for more efficient and intelligent decision-making in complex real-world scenarios.

Part I

Background

This part provides a brief overview of the subject area, the key concepts, and the current state of knowledge in the field. It consists of two chapters: the first chapter (Chapter 2) discusses Classical Planning; the second chapter (Chapter 3) discusses Graph Neural Networks.

Chapter 2

Classical Planning

This chapter familiarizes the reader with classical planning and describes tools to work with it. The following chapter is drawn from [SBG22], [Hel06], [Lip14], [Mar13], [Gre23] and [GKW+98].

2.1 Classical Planning

Classical planning is a branch of Artificial Intelligence (AI) that develops algorithms and techniques for automated planning in deterministic, fully observable, discrete, and static environments.

In classical planning, the problem consists of domain and domain instances. A domain typically includes a set of rules and constraints that define the allowable actions, preconditions, and effects of those actions within the domain. A domain instance is a specific problem within a given domain defined by objects, facts, and initial and goal states. Then, the task is to generate a sequence of actions (a plan) that can transform the system's initial state to one that satisfies the given goals.

For example, consider a scenario where a delivery company needs to transport packages from one city to another. The initial state could be the location of the packages, cities, predefined routes between them, and available vehicles, while the goal state could be the packages' desired destinations. The

available actions include loading packages onto a truck, driving the truck to the destination city, and unloading the packages at the destination. The transport problem domain aims to find a sequence of actions to move the packages from the initial state to the goal state while minimizing the cost or time required to complete the task. The illustrative picture is shown in Figure 2.1.

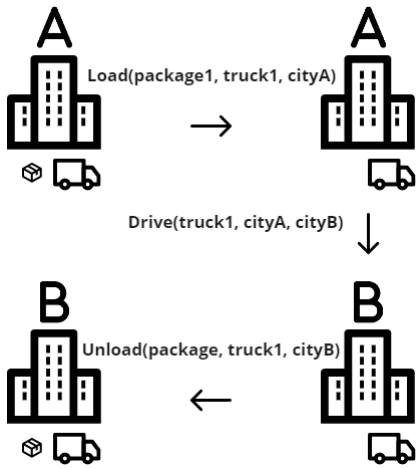


Figure 2.1: Example of transport domain instance

Now we define planning tasks formally. A *classical planning task* is a pair $P = \langle D, I \rangle$ consisting of a planning domain D and a domain instance I. The *planning domain* D is determined by a first-order relational language $\mathcal{L}$ and a set of action schemata $\mathcal{A}$. Relational structures over $\mathcal{L}$ form states of the planning task P and the action schemata define possible transitions between states.

An action schema $a(\vec{x}) \in \mathcal{A}$ is specified by three sets of atomic formulas $\mathsf{pre}_a(\vec{x}), \mathsf{add}_a(\vec{x}), \mathsf{del}_a(\vec{x})$ built up from variables among the action parameters $\vec{x}$, and called *preconditions*, *add effects*, and *delete effects*, respectively. Let $\mathbf{S}$ an $\mathcal{L}$-structure over a set of objects O. Given a sequence of objects $\vec{o}$ of the same length as $\vec{x}$, we can create a ground action $a(\vec{o})$ by substituting objects $\vec{o}$ for the parameters $\vec{x}$. The ground action $a(\vec{o})$ is said to be *applicable* in $\mathbf{S}$ if all its preconditions hold in $\mathbf{S}$, i.e., $\mathbf{S} \models p(\vec{o})$ for each atomic formula $p(\vec{o}) \in \mathsf{pre}_a(\vec{o})$. In that case, we can apply the action $a(\vec{o})$ in $\mathbf{S}$. The result of the application is an $\mathcal{L}$-structure $\mathbf{S}'$ that is a modification of $\mathbf{S}$ so that

1. $\mathbf{S}' \models p(\vec{o})$ for each $p(\vec{o}) \in \mathsf{add}_a(\vec{o})$,
2. $\mathbf{S}' \models \neg p(\vec{o})$ for each $p(\vec{o}) \in \mathsf{del}_a(\vec{o})$, and
3. $\mathbf{S}' \models p(\vec{o})$ iff $\mathbf{S} \models p(\vec{o})$ for the ground atoms $p(\vec{o}) \notin \mathsf{add}_a(\vec{o}) \cup \mathsf{del}_a(\vec{o})$.

The *planning instance* $I = \langle \mathbf{S}_I, \psi_G \rangle$ consists of an $\mathcal{L}$-structure $\mathbf{S}_I$ called the *initial state* and a set of ground atomic formulas ψ_G called the *goal*. To solve the planing task P, we need to find a sequence π of ground actions that transform the initial state into a state satisfying the goal. In this thesis, we focus only on actions of the unit cost. Thus the cost of π is defined as its length. The plan π is said to be *optimal* if there is no shorter plan.

Example 2.1. We illustrate the above definitions with a simple example from the transport domain. The relational language $\mathcal{L}$ consists of three binary predicates at, in, and road. There are three types of objects, namely trucks, packages, and cities. The types can be modelled by unary predicates truck, package, and city. Further, the domain contains three action schemata: $\text{drive}(t, c_1, c_2)$, $\text{load}(p, t, c)$, and $\text{unload}(p, t, c)$. Formally, they are defined as follows:

- $\mathsf{pre}_{\text{drive}}(t, x, y) = \{\text{truck}(t), \text{city}(x), \text{city}(y), \text{at}(t, x), \text{road}(x, y)\}$,
- $\mathsf{add}_{\text{drive}}(t, x, y) = \{\text{at}(t, y)\}$,
- $\mathsf{del}_{\text{drive}}(t, x, y) = \{\text{at}(t, x)\}$,
- $\mathsf{pre}_{\text{load}}(p, t, c) = \{\text{package}(p), \text{truck}(t), \text{city}(c), \text{at}(p, c), \text{at}(t, c)\}$,
- $\mathsf{add}_{\text{load}}(p, t, c) = \{\text{in}(p, t)\}$,
- $\mathsf{del}_{\text{load}}(p, t, c) = \{\text{at}(p, c)\}$,
- $\mathsf{pre}_{\text{unload}}(p, t, c) = \{\text{package}(p), \text{truck}(t), \text{city}(c), \text{in}(p, t), \text{at}(t, c)\}$,
- $\mathsf{add}_{\text{unload}}(p, t, c) = \{\text{at}(p, c)\}$,
- $\mathsf{del}_{\text{unload}}(p, t, c) = \{\text{in}(p, t)\}$.

The instance is specified by an $\mathcal{L}$-structure representing the initial state and a goal condition. Our example comprises five objects: t_1 of type truck, p_1 of type package, and c_1, c_2, c_3 of type city. As all the predicate symbol except of the unary ones are binary, we can represent $\mathcal{L}$-structures as a digraph where the validity of an atom, e.g. $\text{at}(p, c)$, is represented by an arc from p to c labelled by the predicate symbol at. The initial state is depicted in Figure 2.2. The goal is $\psi_G = \{\text{at}(p_1, c_3)\}$. To solve this instance, we first need to drive the truck t_1 to the city c_2, load the package p_1, drive t_1 to c_3, and finally unload p_1. Thus the plan is a sequence of ground actions $\text{drive}(t_1, c_1, c_2)$, $\text{load}(p_1, t_1, c_2)$, $\text{drive}(t_1, c_2, c_3)$, $\text{unload}(p_1, t_1, c_3)$. Its cost is 4 and it is an optimal plan.

The classical planners takes the planning task as its input and search for an (optimal) plan using a search algorithm like A^* navigated by a heuristic

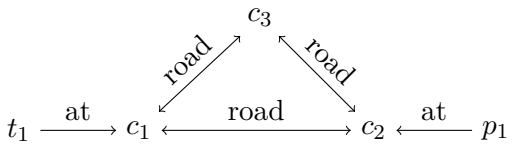


Figure 2.2: The initial state.

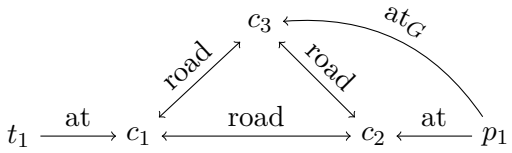


Figure 2.3: The initial state extended by the goal.

function. In this thesis, instead of solving a single instance, we implement a program solving all the domain instances employing machine learning methods. Our solution for a domain is a policy assigning to each state a next action we should apply to get closer to a goal state.

We can work with all domain instances at once because any state in any domain instance is represented by a relational structures over a fixed language $\mathcal{L}$. However, particular instances might differ in their goals. To encode the information about the goal into the state, we expand the language $\mathcal{L}$ by new predicate symbols. For each predicate symbol p in $\mathcal{L}$, we introduce a fresh predicate symbol of the same arity as p denoted p_G. Let $\mathcal{L}_G$ be the resulting language. Given a state $\mathbf{S}$ (i.e., an $\mathcal{L}$-structure) and a goal ψ_G, we expand it to an $\mathcal{L}_G$-structure $\mathbf{S}_G$ so that $\mathbf{S}_G \models p_G(\vec{o})$ for each ground atom $p(\vec{o}) \in \psi_G$. Thus the expanded state $\mathbf{S}_G$ contains the information on the current goal, we want to achieve. For example, the extended initial state from Example example:planning-task is depicted in Figure 2.3.

Let $\mathcal{S}_D$ be the set of all expanded states, i.e., $\mathcal{L}_G$-structures. We represent policies for the domain D as value functions. A *value function* $V \colon \mathcal{S}_D \to \mathbb{R}$ is a real-valued function estimating how far is a goal state from a given state. Each value function determines a *greedy policy* π_V that assigns to a given state $\mathbf{S}$ a ground action that leads to a state $\mathbf{S}'$ with the minimum $V(\mathbf{S}')$. A policy π_V solves a planning instance if following the policy from the initial state always end up in a goal state. Moreover, π_V is optimal if it generates an optimal plan. Using machine learning techniques, we strive to find a value function V that would ideally represent an optimal policy π_V.

2.2 Planning Domain Definition Language

The planning tasks defined in the previous section are usually specified in the Planning Domain Definition Language (PDDL). PDDL is a formal language used to describe classical planning problems in AI. PDDL provides a standard syntax and semantics for representing the problem domain and domain instances using the language of first-order logic. PDDL includes a set of constructs that can specify first-order relational language, action schemata, initial state, goal state, and other relevant aspects of a planning problem. For example, PDDL allows the planner to specify the preconditions and effects of each action, as well as any constraints on the execution of those actions.

PDDL is widely used in the research and development of classical planning systems, and many automated planners support PDDL as a standard input format. In this work, the reference planner (Fast Downward) and GNN planner use PDDL input files.

To specify a planning task in PDDL, we must create two files. The first defines the planning domain, i.e., the first-order relational language and a set of action schemata. The second defines the planning instance, i.e., an initial state and a goal. An example of the domain file and the instance file can be seen in Listings 2.1 and 2.2.

2.3 Fast Downward Planner

Fast Downward (FD) is a popular automated planning system that supports the PDDL planning language. It uses search algorithms and heuristic functions that estimate the distance to the goal state to generate high-quality plans for various planning problems. The planner works in two phases: translate and search. The search phase employs some standard search algorithm like A* endowed with a selected heuristic and searches a plan in a state space represented by the output of the translate phase.

The translate mode translates the input PDDL files (i.e., the domain file and the instance file) into a SAS[1] output. During this process the first-order representation of the planning task is transformed into a propositional representation. Moreover, FD applies several pruning techniques to reduce the

[1] `https://www.fast-downward.org/TranslatorOutputFormat` [accessed 9 May, 2023]

Listing 2.1: Transport domain PDDL file example.

```
(define (domain transport)
  (:requirements :strips :negative-preconditions)
  (:predicates
    (package ?obj)
    (truck ?truck)
    (city ?city)
    (road ?city-from ?city-to)
    (at ?obj ?city)
    (in ?obj ?truck))
  (:action load
    :parameters (?obj ?truck ?city)
    :precondition (and
      (package ?obj)
      (truck ?truck)
      (city ?city)
      (at ?truck ?city)
      (at ?obj ?city))
    :effect (and
      (not (at ?obj ?city))
      (in ?obj ?truck))
  )
  (:action unload
    :parameters (?obj ?truck ?city)
    :precondition (and
      (package ?obj)
      (truck ?truck)
      (city ?city)
      (at ?truck ?city)
      (in ?obj ?truck))
    :effect (and
      (not (in ?obj ?truck))
      (at ?obj ?city))
  )
  (:action drive
    :parameters (?truck ?city-from ?city-to)
    :precondition (and
      (truck ?truck)
      (city ?city-from)
      (city ?city-to)
      (road ?city-from ?city-to)
      (at ?truck ?city-from))
    :effect (and
      (not (at ?truck ?city-from))
      (at ?truck ?city-to))
  )
)
```

Listing 2.2: Transport domain instance PDDL file example.

```
(define (problem transport-1-0)
  (:domain transport)
  (:objects city1 city2 city3 truck1 package1)
  (:init
    (package package1)
    (truck truck1)
    (city city1)
    (city city2)
    (city city3)
    (road city1 city2)
    (road city2 city1)
    (road city2 city3)
    (road city3 city2)
    (road city3 city1)
    (road city1 city3)
    (at truck1 city1)
    (at package1 city2)
  )
  (:goal
    (and
      (at package1 city3)
    )
  )
)
```

resulting state space to be searched. Formally, the propositional representation known as the SAS+ representation [BN95] is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_g \rangle$ where V is a finite set of variables. Each variable $v \in V$ has its domain $\mathsf{dom}(v)$. Using the variables, we define states. A *partial state* is a partial function $s\colon V \to \bigcup_{v \in V} \mathsf{dom}(v)$ such that $s(v) \in \mathsf{dom}(v)$. When s is a total function, i.e. the value $s(v)$ is defined for each variable $v \in V$, s is said to be a *state*. Note that one can represent the partial states as sets of pairs of the form $\langle v, b \rangle$ for $v \in V$ and $b \in \mathsf{dom}(v)$.

Further the propositional representation Π consists of an initial state s_0 and a partial state s_g specifying the goal. A state s is a goal state if $s_g \subseteq s$.

The transition between states are define by the set of operators $\mathcal{O}$. Each operator $a \in \mathcal{O}$ is a pair of two partial states $a = \langle \mathsf{pre}_a, \mathsf{eff}_a \rangle$ encoding the preconditions and effects, respectively. The operator a is applicable in a state s if $\mathsf{pre}_a \subseteq s$, i.e., all the preconditions are satisfied. Applying a to s results

in a state s' defined as follows:

$$s'(v) = \begin{cases} \mathsf{eff}_a(v) & \text{if } \mathsf{eff}_a \text{ is define for } v, \\ s(v) & \text{otherwise.} \end{cases}$$

Given a planning task $P = \langle D, I \rangle$, the Fast Downward translator creates the corresponding SAS output. For our purposes, we need to know how the variables and their domains are built from the planning task P. First, one can split the predicate symbol occurring in P to static and dynamic predicates. A predicate symbol is called *dynamic* if it occurs among the add or delete effects of any action schemata. On the other hand, a predicate is said to be *static* if it occurs only in the action preconditions. So the interpretations of static predicates is determined by the initial state $\mathbf{S}_I$ and is fixed in all reachable states. The interpretations of dynamic predicates might change as we apply actions. Thus to represent a state, it suffices to remember only the interpretations of the dynamic predicates.

To create the variables and their domains, FD utilizes so-called mutex groups. A set of ground atomic formulas M is called a *mutex group* if for any reachable state $\mathbf{S}$, we have $\mathbf{S} \models p$ for at most one atom $p \in M$. The domain $\mathsf{dom}(v)$ of any variable $v \in V$ is created from a mutex group M. As we know that at most of the atoms in M can be true in a state, it is sufficient to know which of them it is. Thus the domain $\mathsf{dom}(v)$ consists of atoms in M. In addition, $\mathsf{dom}(v)$ can be extended by a special value `none-of-those` if it may happen that none of the atoms holds in a reachable state. Consequently, given a state s in the SAS output, we can restore the original structure $\mathbf{S}$ corresponding to s, by collecting the atoms $s(v)$ for $v \in V$ and expanding them by the interpretation of the static predicates.

Fast Downward, by default, uses various pruning techniques to reduce the resulting state space represented by the SAS output. First, it removes unreachable ground atoms, i.e., atoms that can never be true in a reachable state. Further, based on the goal specification, it can remove unnecessary SAS variables that do not have any helpful information that leads to a goal SAS state. As the paper [SBG22] whose results we try to replicate does not apply the pruning based on the goal, we need to keep the unnecessary SAS variables in the SAS output. To do that, one can use Fast Downward Planner's parameter `--translate-options --keep-unimportant-variables`.

Example 2.2. Consider the planning task from Example 2.1. Its SAS output consists of two variables v_1, v_2. The first variable represents the truck's location and the second the package's location. So we have $\mathsf{dom}(v_1) = \{\mathrm{at}(t_1, c_1), \mathrm{at}(t_1, c_2), \mathrm{at}(t_1, c_3)\}$ and $\mathsf{dom}(v_2) = \{\mathrm{at}(p_1, c_1), \mathrm{at}(p_1, c_2), \mathrm{at}(p_1, c_3)\}$. The interpretations of the unary predicates and the predicate road are static.

Listing 2.3: SAS plan for transport domain instance shown in Listing 2.2.

```
(drive truck1 city1 city2)
(load package1 truck1 city2)
(drive truck1 city2 city3)
(unload package1 truck1 city3)
; cost = 4 (unit cost)
```

The initial state $s_0 = \{\langle v_1, \text{at}(t_1, c_1)\rangle, \langle v_2, \text{at}(p_1, c_2)\rangle\}$. The partial state representing the goal is $s_g = \{\langle v_2, \text{at}(p_1, c_3)\rangle\}$.

Example of SAS output (translate mode output) could be generated by Fast Downward with parameter `--translate-options --keep-unimportant-variables` for domain instance given in Listing 2.2.

Example of SAS plan (search mode output) generated by Fast Downward domain instance given in Listing 2.2 can be seen in Listing 2.3.

Chapter 3

Graph Neural Networks

This chapter familiarizes the reader with graph neural networks. The following chapter is drawn from [SBG22], [RN10], [Bou23], [SGT+09], and [ZLLS21].

3.1 Neural Networks

Neural Networks (NN) refer to a cluster of algorithms and mathematical models inspired by the structure and function of the human brain. They comprise layers of interconnected processing nodes or neurons that receive input signals, process them through weighted connections, and generate an output signal. NNs can be trained to recognize patterns, classify data, and make predictions based on input data. They are often used in machine learning applications, such as image recognition, speech recognition, natural language processing, and predictive analytics, pattern recognition. Various architectures can implement NNs, such as feedforward, recurrent, convolutional, graph, and deep neural networks. Illustration of NN layers can be seen in Figure 3.1.

General steps to create a NN model:

- Define the problem and determine the input and output variables.

[1] `https://www.tibco.com/sites/tibco/files/media_entity/2021-05/neutral-network-diagram.svg` [accessed 27 Apr, 2023]

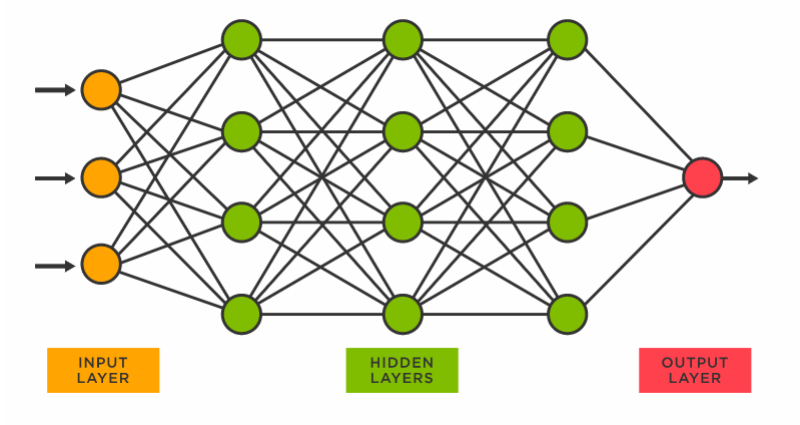


Figure 3.1: Neural Network layers[1].

- Gather and preprocess the data for training the model. Before using the data, it is essential to clean it up, standardize or adjust the features, and divide it into three sets: training, validation, and testing.
- Choose a suitable neural network architecture appropriate for the problem at hand. It could be a feedforward network, recurrent network, convolutional network, or a combination of these.
- Define the number of layers, the number of neurons in each layer, and the activation functions for each neuron. It determines the complexity and capacity of the model.
- Train the network by running a training loop with an optimization algorithm, such as stochastic gradient descent or Adam, to minimize the loss between predicted and actual output (see substeps of this step below).
- Evaluate the performance of the trained model on the validation or test dataset to assess its accuracy and generalization ability.
- Tune the model's hyperparameters (hardcoded values), such as the learning rate, batch size, and regularization strength, to improve its performance and prevent overfitting.
- Deploy the trained model to make predictions on new data. Creating a Neural Network model requires understanding the problem domain, data preprocessing techniques, and neural network architectures and optimization algorithms. Achieving optimal results requires an iterative process that involves experimentation and fine-tuning.

Steps typically involved in the training loop (the train step from above) of a Neural Network:

- Initialize the weights and biases of the Neural Network.
- Input the training data into the network.
- Compute the output of the network using the current weights and biases.
- Calculate the loss between the predicted output and the actual output.
- Use backpropagation to compute the loss gradient for the weights and biases.
- Update the weights and biases using an optimization algorithm, such as stochastic gradient descent.
- Repeat all the steps above except for the first one for a fixed number of iterations or until validation or training loss converges to desired one or no improvements occur.

The training loop may also include techniques such as regularization, early stopping, learning rate schedules, splitting the training dataset into small batches for lowering computational complexity, and model checkpointing to improve the training process and prevent overfitting. Take a look at Figure 3.2.

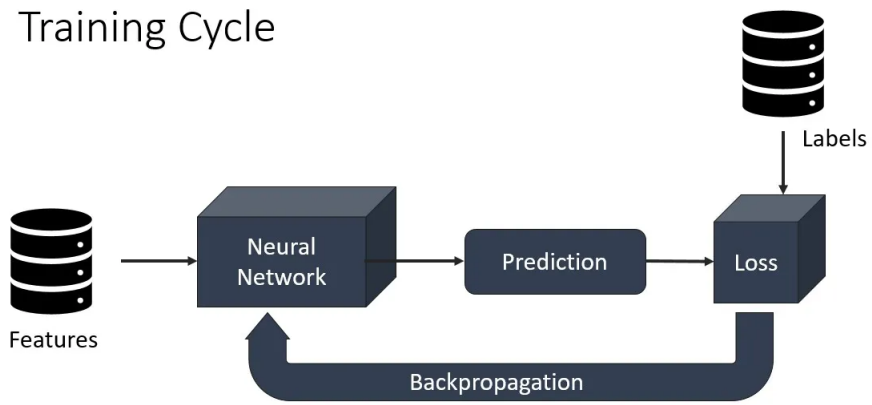


Figure 3.2: Neural Network training loop[2].

This work uses Graph Neural Network (GNN) to predict cost and plan for classical problems.

[2] https://i0.wp.com/galaxyinferno.com/wp-content/uploads/2021/08/Slide3.jpg?w=1280&ssl=1 [accessed 27 Apr, 2023]

3.2 Graph Neural Networks

Graph Neural Networks (GNNs) are machine learning models operating on graph-structured data, such as social networks, molecular structures, and knowledge graphs. GNNs learn to model the complex interactions and dependencies between nodes and edges in a graph by recursively aggregating, combing, and propagating information between neighboring nodes (message passing), using a Neural Network to update node and edge representations at each step. It allows GNNs to capture a graph's structural and feature information and perform tasks such as node classification, link prediction, and graph classification. There are many variants and extensions of GNNs. The choice of GNN architecture depends on the specific problem and the characteristics of the graph data.

GNNs have been applied to various domains, including social networks, chemistry, computer vision, and recommendation systems. They have shown promising results in protein structure prediction, drug discovery, and traffic congestion prediction in road networks.

GNNs are an active area of research, and numerous advancements have occurred in recent years. However, they also present several challenges, including scalability to large graphs, generalization to new graphs, and interpretability of learned representations.

In this work, the Aggregate-Combine Graph Neural architecture of GNN [BKM+20] is used (we will refer to them shortly as GNN) to learn Value Function. Our application of GNNs will be described in the following chapters of this work (see Sections 4.3 and 5.2).

The input of a GNN is a graph $G = (V, E)$ together with feature vectors $\boldsymbol{s}_v^{(0)} \in \mathbb{R}^n$ for each vertex $v \in V$. There are two standard tasks GNNs are used for. The first is *Vertex classification* where the output of the GNN is a map assigning to each vertex a label. The second is *Graph classification* where the GNN assign a single label to the whole input graph. We will employ GNNs only for the second task.

A GNNs consists of several GNN layers $L^{(1)}, \ldots, L^{(d)}$ followed by the final readout layer. A GNN layer $L^{(i)}$ of input dimension n and output dimension m is specified by two functions: an *aggregation function* **agg** and a *combination function* $\mathbf{comb}\colon \mathbb{R}^{2n} \to \mathbb{R}^m$. The aggregation function **agg** maps finite multisets of vectors in $\mathbb{R}^n$ to vectors in $\mathbb{R}^n$. Recall that a multisets

are collection of elements that allow for multiple occurrences of any element.

The layer $L^{(i)}$ transform the feature vectors $\boldsymbol{s}_v^{(i-1)}$ from the previous layer $L^{(i-1)}$ and transforms them to new feature vectors $\boldsymbol{s}_v^{(i)}$. To define the transformation, we need to introduce the set of neighbors of a vertex. Given a vertex $v \in V$, we define the set of its neighbors in G as $N(v) = \{u \in V \mid (u, v) \in E\}$. The feature vector created by the i-th layer $L^{(i)}$ is computed as follows:

$$\boldsymbol{s}_v^{(i)} = \mathbf{comb}\left(\boldsymbol{s}_v^{(i-1)}, \mathbf{agg}(\{\!\{\boldsymbol{s}_u^{(i-1)} \mid u \in N(v)\}\!\})\right)$$

where $\{\!\{\boldsymbol{s}_u^{(i-1)} \mid u \in N(v)\}\!\}$ denotes the multiset of the feature vectors of the vertices from $N(v)$.

See one iteration of GNN Model over one object in Figure 3.3

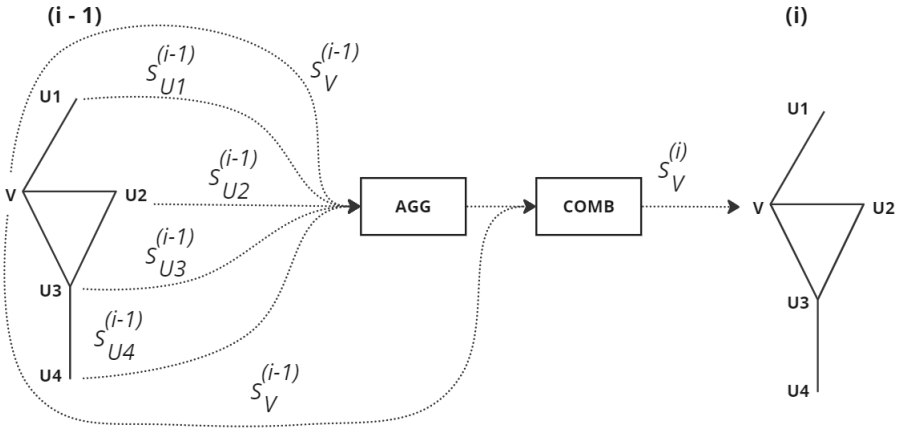


Figure 3.3: One iteration of GNN Model over one object **v** from structure **S**

The final readout layer is specified by a readout function $\mathbf{ro}\colon \mathbb{R}^m \to \mathbb{R}$ where m is the output dimension of the last GNN layer $L^{(d)}$.

The function **comb** and **agg** are usually fixed across the GNN layers. The aggregation function **agg** is typically the sum, pointwise maximum, of the arithmetic mean of the vectors in the multiset. The combination function **comb** together with the readout function **ro** are represented through feed-forward NNs so that their parameters can be learnt by the backpropagation algorithm.

Part II

Application

This part of the work revolves around work conducted by [SBG22]. The majority of the original provided code is modified, improved, reimplemented, or in any other way changed. It consists of two chapters: the first chapter (Chapter 4) discusses what is GNN Data Generator, GNN Model, and GNN Planner; the second chapter (Chapter 5) provides a brief overview of the implementation of the generator, the model, and the planner.

Chapter 4

Application of GNN

The current chapter describes the application of GNN. Firstly, it introduces the application pipeline (see Section 4.1). Then, it explains three parts of the application: GNN Data Generator and its algorithm (see Section 4.2), GNN Model and its algorithm (see Section 4.3), and GNN Planner prediction part and its algorithm (see Section 4.4).

4.1 Pipeline

If we have a problem domain, we want to solve its instances. An alternative to classical solving methods is to apply Graph Neural Network (GNN). Firstly, we generate training data from domain problem instances. Then, we use the generated data to train the GNN Model to approximate the Value Function. Lastly, we implement the greedy policy based on the Value Function represented by the trained GNN Model to solve any domain problem instance. Take a look at Figure 4.1.

4.2 Generator

Bunch of example input-output pairs are required to train any AI model to predict the closest output compared to the real one. In our case, the pairs

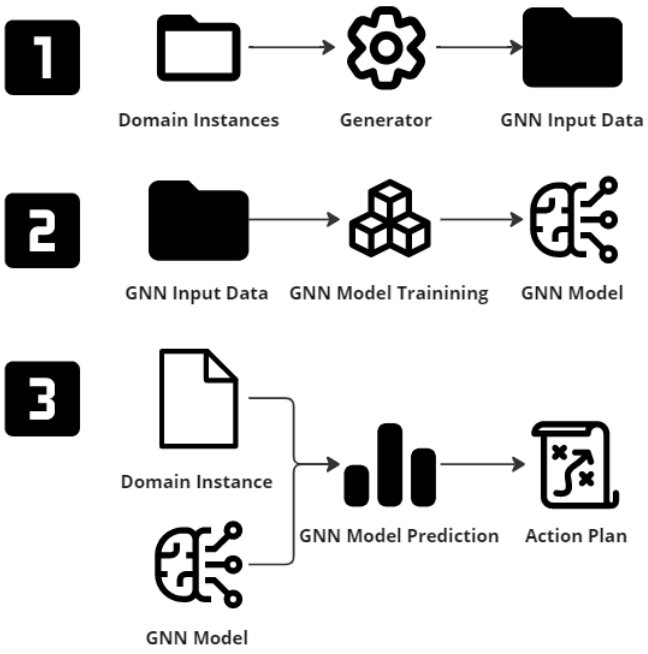


Figure 4.1: GNN application pipeline

consists of a state represented by an $\mathcal{L}_G$-structure $\mathbf{S}$ enriched by the goal (see Section 2.1) and its cost (the length of the shortest plan transforming $\mathbf{S}$ to a goal state.

Firstly, we use Fast Downward Planner's translate mode to get SAS output (see Fast Downward Planner and SAS output description in Section 2.3) from given PDDL files. Then, we work with SAS initial state and SAS operators from that SAS output, where SAS operators provide us functionality of transitioning between SAS states if possible. Because of that, it allows us to explore SAS state space and generate new SAS states.

Next, we parse PDDL files. We get from domain file predicate names, predicates arities and dynamic predicate names. We get from domain instance file object names and initial list of tuples of predicate and its objects.

Lastly, we combine SAS states and parsed information from PDDL files for the current PDDL domain instance using the following sequence of actions:

- 1. Create number mappings for predicate names.
- 2. Create number mappings for object names.
- 3. Map predicates arities using mappings from Step 1.

- 4. Extract list of tuples of static predicates (facts) from initial list of tuples of predicate and its objects using dynamic predicate names. Map facts using mappings from Step 1 and Step 2. Facts are valid for any state generated from the current PDDL instance.
- 5. Map goal SAS state using mappings from Step 1 and Step 2.
- 6. Map generated list of tuples of cost and SAS state using mappings from Step 1 and Step 2.

We save generated data for every PDDL domain instance. It includes predicate names number mappings, predicate arities, object names number mappings, facts (list of tuples of static predicate and its objects), goal (list of tuples dynamic predicate and its objects that should be satisfied), and list of cost-state pairs (state is represented as a list of tuples of dynamic predicate and its objects).

We aggregate generated data from all PDDL domain instances to create relational $\mathcal{L}_G$-structures. Firstly, we create a model from GNN Model's $\mathbf{MLP}_p$ nets from predicates arities (these are fixed for all predicates for any instance of some given domain). See details about $\mathbf{MLP}_p$ in Section 4.3. Then, we transform cost-state pairs into cost-relational-structure pairs, where relational structures are built from some state (list of tuples of dynamic predicate and its objects) with facts (list of tuples of static predicate and its objects) and goals (list of tupels of dynamic predicate and its objects). Lastly, the model's training uses aggregated cost-relational-structure pairs from all generated data from all instances.

This work uses Algorithm 1 for GNN Data Generator. Description:

- Lines 1–4: Initialization steps for an infinite loop.
- Lines 6–25: Infinite generating loop with break conditions.
  - Lines 7–11: Conditions of breaking out of infinite generating loop. Breaking out if one of the following conditions are met: too many unsuccessful iterations in a row without newfound states, the timer is out, or the desired number of generated states is acquired.
  - Line 13: Taking random SAS state from already found `state_cost_pairs`.
  - Line 14: Transitioning `given_length` times from SAS state from the previous line into some neighbor SAS state.
  - Line 15: Firstly, creating SAS output file with SAS state from the previous line as initial state. Next, creating Fast Downward plan.

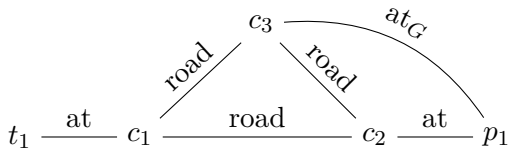


Figure 4.2: The Gaifman graph of the $\mathcal{L}_G$-structure from Figure 2.3.

After that, creating new SAS states using initial SAS state and actions from the plan, creating corresponding costs such that initial SAS state has the original cost from the plan and every following SAS state has lesser and lesser cost by one. Finally, adding only unique states with their costs to all already found state-cost pairs (`state_cost_pairs`).

- Lines 18–22: Updating `patience` variable. If no new state was found - decrease the variable; otherwise, reset the variable.
- Line 24: Saving the current count of state-cost pairs

4.3 GNN Model

The main task of GNN Model is to learn the Value Function (see Section 2.1) from provided data. The model's learnt Value Function determines a greedy policy employed in the actual planner.

As the input for the GNN model are relational structures (not graphs), the architecture presented in Chapter 3 has to be modified accordingly. To each relational $\mathcal{L}$-structure $\mathbf{S}$ with the set of objects O, one can construct so-called Gaifman graph (see e.g. [EF95, Lib04]). Its vertices are the objects from O and two objects $o_1, o_2 \in O$ are connected by an edge if there is a predicate symbol p and tuple of objects $\vec{o}$ such that $\mathbf{S} \models p(\vec{o})$ and both o_1, o_2 occur in $\vec{o}$. Moreover, we label such an edge by the predicate symbol p. Note that each ground atom $p(\vec{o})$ valid in $\mathbf{S}$ defines a clique in the Gaifman graph labelled by p.

Example 4.1. Consider the initial state from Example 2.1 enriched by the goal depicted in Figure 2.3. As all its predicates are at most binary, its Gaifman graph can be obtained from the digraph in Figure 2.3 by forgetting the arc directions; see Figure 4.2.

When implementing a GNN model over the Gaifman graph, we need to distinguish neighbors based on the edge label. The architecture from

[SBG22] introduces for each predicate symbol p its feed-forward neural net $\mathbf{MLP}_p$ aggregating the features vectors of all objects occurring in $\vec{o}$ such that $\mathbf{S} \models p(\vec{o})$. The GNN model first iterates over all ground atoms $p(\vec{o})$ valid in $\mathbf{S}$ and computes a message $\boldsymbol{m}_{p(\vec{o})}$ for each of them by $\mathbf{MLP}_p$. To compute an updated feature vector $\boldsymbol{s}_o^{(i)}$ for an object $o \in O$, we aggregate the multiset of all the messages $\boldsymbol{m}_{p(\vec{o})}$ such that $o \in \vec{o}$ using either sum or smooth maximum (implemented as LogSumExp). Thus we gather all the information from the neighboring objects in the Gaifman graph. Finally, we update the current feature vector $\boldsymbol{s}_o^{(i-1)}$ by the aggregated messages by means of a feed-forward neural net $\mathbf{MLP}_U$. See Figure 4.3. The model also uses two other feed-forward neural nets $\mathbf{MLP}_1$ and $\mathbf{MLP}_2$ to compute the final readout.

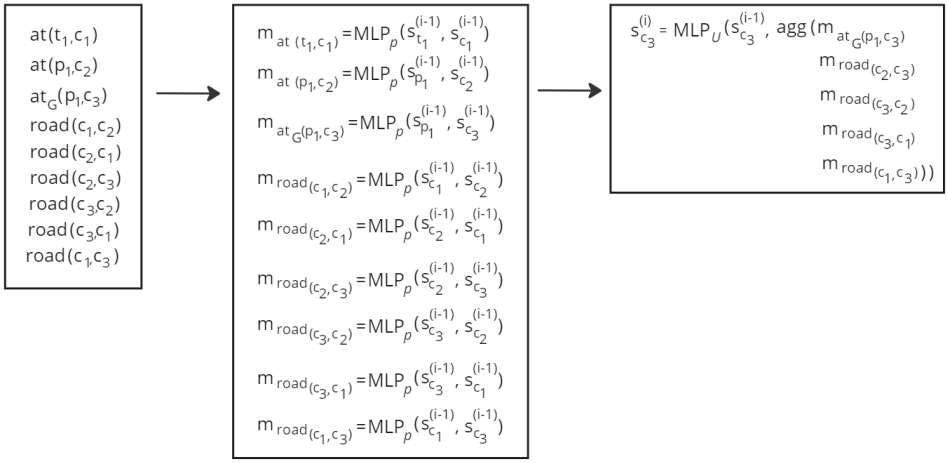


Figure 4.3: One iteration of GNN Model over c_3 object in relational structure from Figure 4.2

Each **MLP** in the GNN model is a feed-forward neural net with two dense layers, the first with the ReLU activation function and the second with a linear activation function.

The details on the GNN model from [SBG22] are shown in Algorithm 2. Description:

- Line 1: Initialization of every object feature vector $\boldsymbol{s}_o^{(0)}$ by concatenating the zero vector $\mathbf{0}$ and a random vector whose components are drawn from the normal distribution $\mathcal{N}(0, 1)$, each of dimension $k/2$, where k represents is the dimension of feature vector and it is hyperparameter (equals 32 in this work as in the original work).
- Line 2–9: Looping L times, where L represents GNN layers (stages) and it is an hyperparameter (equals 30 in this work as in the original work). Updating $\boldsymbol{s}_o$ (feature vectors) throughout L stages, with $\boldsymbol{s}_o^{(i)}$ denoting

the feature vectors after stage i. There are two parts: the first inner loop (Lines 3–5) and the second inner loop (Line 6–8).

- Lines 3–5: Looping throughout every atom $p(\vec{o})$ valid in the given relational structure $\mathbf{S}$ and computing the message $\boldsymbol{m}_{p(\vec{o})}$ using $\mathbf{MLP}_p$.
- Lines 6–8: Looping throughout every object o from O. Updating the feature vector $\boldsymbol{s}_o^{(i-1)}$ to $\boldsymbol{s}_o^{(i)}$ by aggregating and combining messages corresponding to the object o using $\mathbf{MLP}_U$, where the aggregation function **agg** is either sum or smooth maximum.

- Line 10: Construction of the final predicted value for the relational structure $\mathbf{S}$ using $\mathbf{MLP}_1$ and $\mathbf{MLP}_2$ nets.

4.4 Planner

The Planner's main task is to plan actions needed to reach the goal and calculate the plan's cost for a given domain problem instance using provided GNN Model. Firstly, GNN Model is trained using generated data; then, it is applied to create a plan and its cost in the same format created by FD Planner (see Listing 2.3).

We use Fast Downward Planner's translate mode to get SAS output (see Fast Downward Planner and SAS output description in Section 2.3) from given PDDL files. Then, we work with SAS initial state and SAS operators from that SAS output, where SAS operators provide us the functionality of transitioning between SAS states if possible. For any current SAS state, we find all neighbor SAS states, transform them into relational structures, predict cost using trained GNN Model, and choose the neighbor SAS state with the lowest predicted cost till the reaching the goal, dead-end (no neighbors SAS states), or exceeding the given maximum number of neighbor transition.

This work uses Algorithm 3 for GNN Planner. Description:

- Lines 1–5: Initialization steps for an infinite loop.
- Lines 7–33: Infinite generating loop with break conditions.
  - Lines 7–10: Condition of breaking out of infinite generating loop. Breaking out if the goal state is reached.

- Lines 12–16: Condition of breaking out of infinite generating loop. Breaking out and resetting the plan with the plan's cost if steps exceeded `given_steps` number.
- Line 18: Saving the current state to the list of visited states called `visited`
- Lines 20–27: Finding an action that leads to a state with the lowest predicted cost to the goal state.
- Lines 29–32: Updating all variables for the next iteration of the infinite loop.

Algorithm 1 Generator

Prerequisites: `get_state_cost_pairs_to_goal` — a function that creates an actual plan for given state using FD Planner and extracts SAS states with costs from the plan; `count` — a function that returns a count of given data structure of state-cost pairs; `get_random_state` — a function that returns random state from given data structure of state cost pairs; `walk_random` — a function that transitions given state given times; `add_unique` — a function that adds to given data structure of state cost pairs unique states from another given data structure of state cost pairs

Input: `given_count` — maximum number of states to be generated; `given_length` — length of random transitions (random walk), `given_patience` — maximum tries without finding new states, `given_timer` — time to be spend on the problem

Output: `state_cost_pairs` — generated states and their true cost

```
1: state_cost_pairs := get_state_cost_pairs_to_goal(given_state)
2: patience := given_patience
3: timer := given_timer
4: previous_count := count(state_cost_pairs)
5:
6: while true do
7:     if patience ⩽ 0
8:       or timer ⩽ 0
9:       or previous_count ≥ given_count
10:        break
11:    end if
12:
13:    random_state := get_random_state(state_cost_pairs)
14:    random_state := walk_random(random_state, given_length)
15:    add_unique(state_cost_pairs,
16:                get_state_cost_pairs_to_goal(random_state))
17:
18:    if count(state_cost_pairs) == previous_count
19:       patience = patience - 1
20:    else
21:       patience := given_patience
22:    end if
23:
24:    previous_count := count(state_cost_pairs)
25: end while
```

Algorithm 2 Aggregate-Combine Graph Neural Network

Input: Relational structure $\mathbf{S}$ over a set of objects O [state (s)]
Output: $v \in \mathbb{R}$ [value $V(s)$]

// Partial random initialization
$\boldsymbol{s}_o^{(0)} \sim \mathbf{0}^{k/2}\mathcal{N}(0,1)^{k/2}$ for each object $o \in O$
for $i \in \{1, \ldots, L\}$ **do**
  for atom $p(\vec{o}) := p(o_1, \ldots, o_n)$ such that $\mathbf{S} \models p(\vec{o})$ **do**
    // Generate messages
    $\boldsymbol{m}_{p(\vec{o})} := \mathbf{MLP}_p\left(\boldsymbol{s}_{o_1}^{(i-1)}, \ldots, \boldsymbol{s}_{o_n}^{(i-1)}\right)$
  end for
  for $o \in O$ **do**
    // Aggregate messages and update
    $\boldsymbol{s}_o^{(i)} := \mathbf{MLP}_U\left(\boldsymbol{s}_o^{(i-1)}, \mathbf{agg}\left(\{\!\{\boldsymbol{m}_{p(\vec{o})} \mid o \in \vec{o}, \mathbf{S} \models p(\vec{o})\}\!\}\right)\right)$
  end for
end for
// Final Readout
$v := \mathbf{MLP}_2\left(\sum_{o \in O} \mathbf{MLP}_1\left(\boldsymbol{s}_o^L\right)\right)$

Algorithm 3 Planner

Prerequisites: `list` — a function that creates empty list structure; `get_n_state_action_tuples` — a function that returns all neighbor states of given state and actions that produced them; `predict_costs` — a function that predicts costs for states from given states-action tuples; `get_lowest_cost_triple_not_visited` — a function that returns triple from given state-action-cost triples with the lowest predicted cost and not visited state; `add` — a function that adds to a list new entry

Input: `given_model` — trained GNN Model; `given_state` - starting/initial state; `given_goal_state` — the goal state; `given_steps` — number of maximum steps

Output: `plan` and `plan_cost` — list of actions and total action cost

```
1: plan := list()
2: plan_cost := 0
3: state := given_state
4: steps := 0
5: visited := list()
6:
7: while true do
8:     if state == given_goal_state
9:         break
10:    end if
11:
12:    if steps > given_steps
13:        plan := NULL
14:        plan_cost := -1
15:        break
16:    end if
17:
18:    add(visited, state)
19:
20:    n_state_action_tuples := get_n_state_action_tuples(state)
21:    n_state_action_cost_triples := predict_costs(
22:      given_model, n_state_action_tuples
23:    )
24:    n_best_triple := get_lowest_cost_triple_not_visited(
25:      n_state_action_cost_triples, visited
26:    )
27:    best_state, best_action, best_cost := n_best_triple
28:
29:    state := best_state
30:    add(plan, best_action)
31:    plan_cost := plan_cost + 1
32:    steps := steps + 1
33: end while
```

Chapter 5

Modification & Implementation

The current chapter describes the implementation of the generator program (see Section 5.1). Then, it explains the implementation of the GNN Model (see Section 5.2). Finally, it describes the modification of the planner program (see Section 5.3) compared to the provided code from [SBG22].

Download the work from GitLab using link `https://gitlab.fel.cvut.cz/nazarboh/gnn` [accessed 25, 2023].

The implementation of GNN in the paper [SBG22] involves utilizing several popular open-source tools and frameworks, including Python, PyTorch, and PyTorch Lightning, to apply deep learning techniques to create GNN. Python is an interpreted universal programming language popular among the machine learning community due to its ease of use, a large ecosystem of libraries, and support for scientific computing ([Fou23]). PyTorch is a popular deep-learning framework that provides efficient tensor computation with automatic differentiation capabilities ([Con23]). PyTorch Lightning is a higher-level framework built on top of PyTorch in order to simplify the process of training and evaluating deep learning models ([AI23]). These tools have enabled researchers to build and evaluate GNN Models that achieve state-of-the-art performance on their target problems.

This work uses the same tools to modify and implement some additional things. In Sections 5.1 and 5.3, all implemented classes have their own `.py` files named in the snake convention in `Program/Code` folder; otherwise, it is pointed out.

5.1 Generator

To train GNN Value Function, the input data in a special form with many state cost pairs is needed. The original work ([SBG22]) does not provide the generator, which is why the one was implemented. In this section, all technical details of the GNN Data Generator are described (see the application details in Section 4.2).

5.1.1 Prerequisites

The following things are required:

- Python[1]
- Compiled FD Planner[2] (see Section 2.3)
- Python frozendict package[3]

5.1.2 Executable Main Scripts

The generator has two types of main executable Python scripts that are in `Program` folder:

- `main_terminal_gnn_data_generator.py`
- `main_gnn_data_generator.py`

The first type of main parses arguments passed through the command line. The second one parses one argument — the name of the JSON file, then parses arguments from that given JSON file. After argument parsing, an instance of

[1] `https://www.python.org/downloads/` [accessed 5 May, 2023]
[2] `https://www.fast-downward.org/ObtainingAndRunningFastDownward` [accessed 5 May, 2023]
[3] `https://pypi.org/project/frozendict/` [accessed 5 May, 2023]

class `GnnDataGenerator` is created, then generator mode is chosen, and the appropriate instance method is called. The generator has two modes: single and multiple. In folder `Program`, see the `README_GNN_DATA_GENERATOR.md` file for the user manual and all parameter descriptions. Also, take a look at the prepared JSON files for both modes in the `Program` folder: `single_config.json` and `multiple_config.json`.

5.1.3 Generator Single Mode

The `single` mode generates a bunch of cost-state pairs in the form of class-container `GnnData`. The method `single` is a method of the `GnnDataGenerator` class instance. It is used by main scripts (see the description of Executable Main Scripts in Subsection 5.1.2) or by the `multiple` method of the same class instance (see the description of Generator Multiple Mode in Subsection 5.1.4). The method takes one argument of the `SingleConfig` class instance (class from the `configs.py` file in the `Program/Code` folder). This container class captures all arguments passed by the caller. Then, the following sequence of operations in the method is executed:

- 1. Creates a unique number to put in the name of every temporary file.
- 2. Runs Fast Downward Planner as a subprocess to create a translation of given PDDL files into a SAS output file. See details about the SAS format and the SAS output file in Section 2.3.
- 3. Parses created SAS file from step 2 into an instance of the `SasData` class using the functionality of the `SasFileParser` class instance. See details about the `SasFileParser` class in Subsection 5.1.5.
- 4. Reads given PDDL files and captures them in a `PddlData` class instance using the functionality of the `PddlFileReader` class instance. See details about the `PddlFileReader` class in Subsection 5.1.6.
- 5. Creates cost and SAS state pairs for the instance of `SasData` class created in step 3 using the functionality of the `SasCostStatePairsGenerator` class instance. See details about the `SasCostStatePairsGenerator` class in Subsection 5.1.7.
- 6. Adapts the `SasData` class instance from step 5 and the `PddlData` class instance from step 4 into an instance of `GnnData` class using the functionality of the `GnnDataAdapter` class instance. See details about the `GnnDataAdapter` class in Subsection 5.1.8.

- 7. Writes the `SasData` class instance from step 6 into a JSON file using the functionality of the `GnnDataJsonWriter` class instance. See details about the `GnnDataJsonWriter` class in Subsection 5.1.9.
- 8. Cleans up all temporarily created files.

5.1.4 Generator Multiple Mode

The `multiple` mode generates a bunch of cost-state pairs for every PDDL file in a given PDDL folder in the form of class-container `GnnData` by calling the `single` method (see the description of Generator Single Mode in Subsection 5.1.3) within the same class instance. The method `multiple` is a method of the `GnnDataGenerator` class instance. It is used by main scripts (see the description of Executable Main Scripts in Subsection 5.1.2). The method takes one argument of the `MultipleConfig` class instance (class from the `configs.py` file in the `Program/Code` folder). This container class captures all arguments passed by the caller.

5.1.5 SAS File Parser

The main purpose of the `SasFileParser` class is to provide the functionality to parse the SAS output files created by the translate mode of FD Planner. See details about the SAS output file in Section 2.3.

Call the `parse` method of the `SasFileParser` class instance to parse the file. The parsing process is straightforward — it parses the SAS file line by line. The method parses every section of the SAS file using appropriate section parsing instance methods. Every section parser method saves info using SAS structs implemented in the `sas_file_structs.py` file from the `Program/Code` folder. In the end, the method `parse` saves and returns all parsed sections in an instance of `SasData` container class.

5.1.6 PDDL File Reader

The main purpose of the `PddlFileReader` class is to provide the functionality to read useful information from the PDDL domain and the PDDL domain instance files (see the examples of PDDL files in Listings 2.1 and 2.2).

Call the `read` method of the `PddlFileReader` class instance to read the data. The reading process is matching desired information using regular expressions. It reads:

- predicates names from domain file
- predicate arities (number of arguments) from domain file
- effect predicates (predicates that are effected in PDDL actions / dynamic predicates) from domain file
- objects names from domain instance file
- initial predicate states from domain instance file

In the end, the method `read` saves and returns all read data in an instance of `PddlData` container class.

5.1.7 SAS Cost-State Pairs Generator

The main purpose of the `SasCostStatePairsGenerator` class is to provide the functionality to create a bunch of cost and SAS state pairs for a given instance of the `SasData` class. See details about the SAS format and the SAS output file in Section 2.3.

Call the `generate` method of the `SasCostStatePairsGenerator` class instance to generate cost and SAS state pairs. The method `n_random_walk` represents Algorithm 1 described in Section 4.2. The implementation idea: a given instance of the `SasData` class has a list of SAS operators, and every SAS operator has the method called `apply` that can be applied on some state to produce a neighbor SAS state or nothing if the operator is not applicable. This way, in any state, SAS neighbors could be potentially found, and the SAS state could be transitioned into some next neighbor SAS state. See the method `apply` from the SAS `Operator` class in the `sas_file_structs.py` file in the `Program/Code` folder. The method `random_walk` transitions given times random neighbor SAS state out of given start SAS state, then, with final SAS state, the method named `get_state_cost_dictionary_using_planner` is called. This method creates a SAS plan from a given SAS state to the goal SAS state using FD Planner and parses SAS states and costs from it and saves them into SAS state and cost dictionary structure, where the state is a key and the cost is a value. It is implemented this way to keep track of unique states.

5.1.8 GNN Data Adapter

The main purpose of the `GnnDataAdapter` class is to provide the functionality to transform and merge a given instance of the `PddlData` class (see the creation of the `PddlData` instance in Section 5.1.6) and a given instance of the `SasData` class (see the creation of the `SasData` class instance in Sections 5.1.5 and 5.1.7) into an instance of the `GnnData` class.

Before now, we read the PDDL files (the `PddlData` class instance), read the SAS output file (the `SasData` class instance), and created a bunch of SAS states (the `SasData` class instance), but this data should be merged and potentially saved, so the `GnnDataAdapter` class instance does the following:

- 1. Crates the mappings of predicate names from the `PddlData` class instance to numbers. It is the `predicates` variable in the `GnnData` class.
- 2. Maps the predicate arities from the `PddlData` class instance using the mappings from Step 1. It is the variable `predicate_arity_tuples` in the `GnnData` class.
- 3. Crates the mappings of object names from the `PddlData` class instance to numbers. It is the `objects` variable in the `GnnData` class.
- 4. Creates a list of tuples of static predicate and its objects (static predicates are predicates that are not effected in PDDL actions) by extracting effect predicates (dynamic predicates) from the initial list of tuples of predicate and its objects (effect predicates and initial list of tuples of predicate and its objects are from the `PddlData` class instance). It is the `facts` variable in the `GnnData` class.
- 5. Maps the goal SAS state from the `SasData` class instance into number representation (list of tuples of dynamic predicate and objects) using the mappings from Steps 1 and 3. It is the `goals` variable in `GnnData` class.
- 6. Maps every SAS state in cost-state pairs from the `SasData` class instance into number representation (list of tuples of cost and state, where state is a list of tuples of dynamic predicate and its objects) using the mappings from Steps 1 and 3. It is the `cost_state_pairs` variable in the `GnnData` class.

This way, we have some intermediate stored data, but still it is not in the form of GNN Input.

Call the `adapt` method of the `GnnDataAdapter` class instance to adapt all the given data into the needed form, to create and return an instance of the `GnnData` class.

5.1.9 GNN Data JSON Writer

The main purpose of the `GnnDataJsonWriter` class is to provide the functionality to save a given instance of the`GnnData` class into a JSON file.

Call the `write` method of the `GnnDataJsonWriter` class instance to write a given data into a file. The implementation uses the `json.dumps` function to create a JSON string of the class that will be written into the file. The format description:

- Predicates: dictionary of predicate ids and predicate names
- Predicate arities: list of tuples consisting of predicate id and its arity
- Objects: dictionary of object ids and object names
- Facts: list of of tuples consisting of predicate id and list of passed argument objects
- Goals: list of tuples consisting of predicate id and list of passed argument objects
- Cost state pairs: list of tuples consisting of cost and list of tuples consisting of predicate id and list of passed argument objects

Take a look at Listing 5.1 (three dots represent omitted cost-state pairs).

An alternative class s described in Subsection 5.1.10.

5.1.10 GNN Data TXT Writer

The main purpose of `GnnDataTxtWriter` class is to provide the functionality to save a given instance of the `GnnData` class into TXT file.

Listing 5.1: Generator JSON file output (blocks_clear_2_0)

```
{
    "predicates": {
        "0": "on",
        "1": "ontable",
        "2": "clear",
        "3": "handempty",
        "4": "holding"
    },
    "predicate_arity_tuples": [
        [0, 2],
        [1, 1],
        [2, 1],
        [3, 0],
        [4, 1]
    ],
    "objects": {
        "0": "b",
        "1": "a"
    },
    "facts": [],
    "goals": [
        [2, [1]]
    ],
    "cost_state_pairs": [
        [
            0.0,
            [
                [1, [0]],
                [1, [1]],
                [2, [0]],
                [2, [1]],
                [3, []]
            ]
        ],
        ...
    }
}
```

Call the `write` method of the `GnnDataTxtWriter` class instance to write given data into a file. The method uses appropriate write instance methods to create out of `GnnData` class instance properties a list of strings that will be written into a file. The format description is the same as in Subsection 5.1.9, but in TXT form. Take a look at Listing 5.2 (three dots represent omitted cost-state pairs).

An alternative class is described in Subsection 5.1.9.

5.2 GNN Model

The original code has two implemented models: `add` and `max`. These models are in the `Program/Code/Models` folder in `add.py` and `max.py` files. They were slightly restructured and refactored, but the main structure was not changed. (See the application details in Section 4.3 There are four classes: `RelationMessagePassing`, `Readout`, `RelationMessagePassingModel`, and `AddModel` (or `MaxModel`).

The `RelationMessagePassing` class is written as the PyTorch `Module` class, but inherits the PyTorch Lightning `LightningModule` class to use PyTorch Lightning functionality later. The class is different in files `add.py` and `max.py`, because the `forward` instance method has corresponding combination function. The `forward` instance method corresponds to the Lines 3–8 in Algorithm 2. $\mathbf{MLP}_p$ is stored in the `relation_modules` variable and $\mathbf{MLP}_U$ is stored in the `update` variable. $\mathbf{MLP}_p$ nets are initialized using predicates arities (fixed numbers). Nets have corresponding positioning of the predicate number mappings (fixed numbers). Any input should have the same predicate mappings and its arities, which holds true for instances of the same domain, even though they have different amount of objects.

The `Readout` class is written as the PyTorch `Module` class, but inherits the PyTorch Lightning `LightningModule` class to use PyTorch Lightning functionality later. The `forward` instance method corresponds to the Line 10 in Algorithm 2. $\mathbf{MLP}_1$ and $\mathbf{MLP}_2$ are stored in the `pre` and the `post` variables.

The `RelationMessagePassingModel` class is written as the PyTorch `Module` class, but inherits the PyTorch Lightning `LightningModule` class to use PyTorch Lightning functionality later. The class is just combination of the

Listing 5.2: Generator TXT file output (blocks_clear_2_0)

```
BEGIN_OBJECTS
0 b
1 a
END_OBJECTS
BEGIN_PREDICATES
0 on
1 ontable
2 clear
3 handempty
4 holding
END_PREDICATES
BEGIN_PREDICATE_ARITY_TUPLES
0 2
1 1
2 1
3 0
4 1
END_PREDICATE_PREDICATE_ARITY_TUPLES
BEGIN_FACT_LIST
END_FACT_LIST
BEGIN_GOAL_LIST
2 1
END_GOAL_LIST
BEGIN_STATE_LIST
BEGIN_LABELED_STATE
0.0
BEGIN_STATE
1 0
1 1
2 0
2 1
3
END_STATE
END_LABELED_STATE
...
END_STATE_LIST
```

`RelationMessagePassing` and the `Readout` classes. The `forward` instance method corresponds to the Lines 1-13 (all lines) in Algorithm 2.

The `AddModel` class from `add.py` (or the `MaxModel` class from `max.py`) is the PyTorch Lightning wrapper of the `RelationMessagePassingModel` class.

See the structure of GNN Model at Figure 5.3.

`RelationMessagePassing`, `Readout`, and `RelationMessagePassingModel` classes were written as PyTorch `Module` classes, but they inherit PyTorch Lightning `LightningModule` class, because later it provides functionality to work with different types of the processing units.

`AddModel`'s from `add.py` (or `MaxModel`'s from `max.py`) `forward` class method has input format of the tuple that consists of combined relational structure and a list of relation maximum object number (where we represent the relation structure as a dictionary with the key of a predicate and a value of a list of objects argument lists that are in relation with that predicate). The combined relation structure is a dictionary with the key of a predicate and a value of a list of objects that are in relation with that predicate. Relation maximum object numbers are for decoding any encoded relational structure from the combined relational structure.

Notice that no information is lost. Predicate mappings and arities are encoded in GNN Model; there is no need for a list of argument lists because we can take from one large object list just an arity amount objects at a time. The maximum object number for any relational structure restores all objects from the combined structure - the objects from the current structure are bigger than the sum of all previous maximum object numbers and less than that sum plus the current maximum object number. This way, any relational structure information could be restored.

Because of encoding predicate number mappings and arities of these predicates, we can have instances with any amount of objects as long as it has the same predicate numbers and predicate arities (fixed amount of predicates and their arities). See Example 5.1.

Example 5.1. We illustrate the above definitions with a simple example from the transport domain. The relational language $\mathcal{L}$ consists of three binary predicates at, in, and road. There are three types of objects, namely trucks, packages, and cities. The types can be modelled by unary predicates truck, package, and city.

Listing 5.3: Relational structures represented in a dictionary

```
The first structure:      | The second structure:
{                         | {
    0: [[0, 1], [3, 1]]   |     0: [[0, 1]]
    2: [[1, 2]]           |     1: [[2, 0]]
    3: [[3, 2]]           |     3: [[2, 1]]
}                         | }
```

Listing 5.4: GNN Input: Tuple consisting of combined relational structure represented in a dictionary and a list of relational structure's maximum object numbers

```
(
    {
        0: [0, 1, 3, 1, 4, 5]
        1: [6, 4]
        2: [1, 2]
        3: [3, 2, 6, 5]
    },
    [4, 3]
)
```

Imagine we have two $\mathcal{L}_G$-structures depicted in Figures 5.1 and 5.2. The instance from the first figure comprises four objects: t_1 of type truck, p_1 of type package, and c_1, c_2 of type city. The instance from the first figure comprises three objects: t_1 of type truck, p_1 of type package, and c_1 of type city. Then, we map predicates for both $\mathcal{L}_G$-structures with the same numbers: at is 0, in is 1, road is 2, and at_G is 3. After that, we map objects from both structures individually. The mappings for the first relational structure are: t_1 is 0, c_1 is 1, c_2 is 2, and p_1 is 3. The mappings for the second relational structure are: t_1 is 0, c_1 is 1, and p_1 is 2.

These two $\mathcal{L}_G$-structures we can represent as a dictionary, where the key is predicate number mapping and the value is a list of arguments lists. Take a look at Lissting 5.3.

Then, we can encode two relational structures into one, along with a list of the maximum object numbers of every structure. Take a look at Listing 5.4.

The first relational structure objects are from 0 to 4, and the second relational structure objects are from 4 to 7. When the relational structure is decoded, we could decode the amount of that predicate in the original relational structure by knowing objects and predicate arities: predicate 0 (arity 2) in the first relational structure has objects 0, 1, 3, and 1. So we can

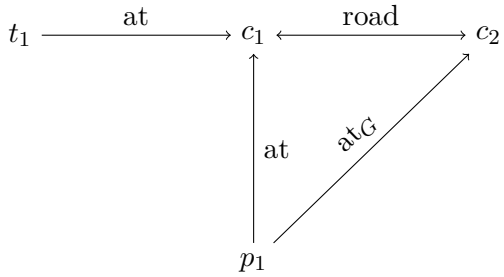


Figure 5.1: First $\mathcal{L}_G$-structure

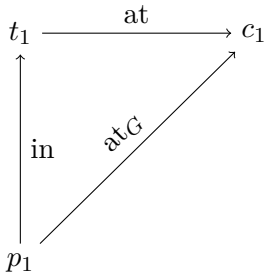


Figure 5.2: Second $\mathcal{L}_G$-structure

restore predicate 0 with list arguments of [0, 1] and [3, 1].

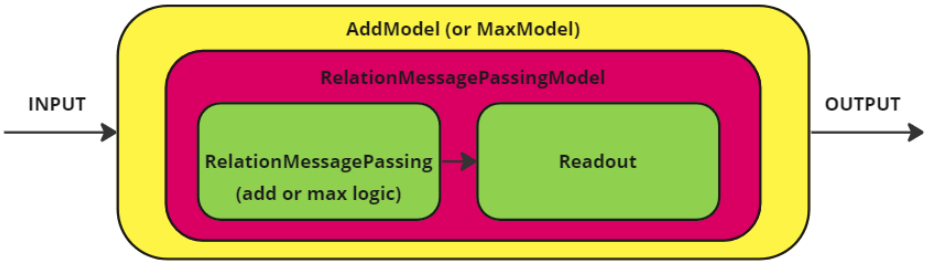


Figure 5.3: Structure of add and max models.

5.3 Planner

The main purpose of GNN Planner is to create a plan of actions with total action cost. The original work ([SBG22]) does not provide the functionality of creating a plan, which is why the one was implemented. Additionally, other functionalities were modified and refactored. In this section, all technical details of GNN Planner are described (see the application details in Section 4.4).

5.3.1 Prerequisites

The following things are required:

- Python[4]
- PyTorch[5]
- PyTorch-Lightning[6]
- (predict mode) Compiled FD Planner[7] (see Section 2.3)
- (predict mode) Python frozendict package[8]

5.3.2 Executable Main Scripts

The planner has two types of main executable Python scripts that are in `Program` folder:

- `main_terminal_gnn.py`
- `main_gnn.py`

The first type of main parses arguments passed through the command line. The second one parses one argument — the name of the JSON file, then parses arguments from that given JSON file. After argument parsing, an instance of the `Gnn` class is created, then planner mode is chosen and appropriate instance method is called. The planner has four modes: train, resume, test, and predict. In folder `Program`, see the `README_GNN.md` file for the user manual and all parameter descriptions. Also, take a look at the prepared JSON files for modes in the `Program` folder: `train_config.json`, `resume_config.json`, `test_config.json`, and `predict_config.json`.

[4] `https://www.python.org/downloads/` [accessed 5 May, 2023]
[5] `https://pytorch.org/get-started/locally/` [accessed 6 May, 2023]
[6] `https://lightning.ai/docs/pytorch/stable/starter/installation.html` [accessed 6 May, 2023]
[7] `https://www.fast-downward.org/ObtainingAndRunningFastDownward` [accessed 5 May, 2023]
[8] `https://pypi.org/project/frozendict/` [accessed 5 May, 2023]

5.3.3 Planner Train Mode

The `train` mode trains a GNN Model using provided train and validation datasets. The method `train` is a method of the `Gnn` class instance. It is used by main scripts (see the description of Executable Main Scripts description in Subsection 5.3.2). The method takes one argument of the `TrainConfig` class instance (class from the `configs.py` file in the (`Program/Code` folder). This container class captures all arguments passed by the caller. Then, the following sequence of operations in method is executed:

- 1. Creates an instance of `GnnDataJsonReader` class. See details about the `GnnDataJsonReader` class in Subsection 5.3.7.
- 2. Creates train dataset — an instance of the `GnnDataset` class using the `GnnDataJsonReader` class instance created in step 1. See details about the `GnnDataset` class in Subsection 5.3.9.
- 3. Creates a dataloader — an instance of the PyTorch `DataLoader` class using the dataset created in step 2, the fuction from the `gnn_dataset.py` file called `collate_batch_of_gnn_dataset_state_cost_pairs` function, and other given parameters. See the explanation of collate functions in Subsection 5.3.11.
- 4. Creates validation dataset — an instance of the `GnnDataset` class using the `GnnDataJsonReader` class instance created in step 1. See details about the `GnnDataset` class in Subsection 5.3.9.
- 5. Creates a dataloader — an instance of the PyTorch `DataLoader` class using the dataset created in step 4, the fuction from the `gnn_dataset.py` file called `collate_batch_of_gnn_dataset_state_cost_pairs` function, and other given parameters. See the explanation of collate functions in Subsection 5.3.11.
- 6. Creates a GNN Model (instance of `AddModel` or `MaxModel` class). See details about the GNN Model in Section 5.2.
- 7. Creates a trainer instance of the Pytorch Lightning `Trainer` class using setup method `load_trainer` of the `Gnn` class instance. See details about the `load_trainer` method in Subsection 5.3.12.
- 8. Trains the model from step 6 with dataloaders from steps 3 and 5 using the method `fit` of the instance of the Pytorch Lightning `Trainer` class from step 7.

5.3.4 Planner Resume Mode

The `resume` mode continues training of a provided GNN Model using given train and validation datasets. The method `resume` is method of the `Gnn` class instance. It is used by main scripts (see description the of Executable Main Scripts in Subsection 5.3.2). The method takes one argument of the `ResumeConfig` class instance (class from the `configs.py` file in the `Program/Code` folder). This container class captures all arguments passed by the caller. Then, the same sequence of operations as in `train` mode from Subsection 5.3.3 is executed, except for step 6, where instead GNN model is loaded from provided PyTorch checkpoint.

5.3.5 Planner Test Mode

The `test` mode calculates the ratio of the error cost summation to the actual cost summation using provided GNN Model and test dataset: $\Sigma(predicted\_cost - true\_cost)/\Sigma(true\_cost)$. The method `test` is method of the `Gnn` class instance. It is used by main scripts (see the description of Executable Main Scripts in Subsection 5.3.2). The method takes one argument of the `TestConfig` class instance (class from the `configs.py` file in the `Program/Code` folder). This container class captures all arguments passed by the caller. Then, the following sequence of operations in method is executed:

- 1. Creates an instance of the `GnnDataJsonReader` class. See details about the `GnnDataJsonReader` class in Subsection 5.3.7.
- 2. Creates test dataset — an instance of the `GnnDataset` class using the `GnnDataJsonReader` class instance created in step 1. See details about the `GnnDataset` class in Subsection 5.3.9.
- 3. Creates a dataloader — an instance of the PyTorch `DataLoader` class using he dataset created in step 2, the fuction from the `gnn_dataset.py` file called `collate_batch_of_gnn_dataset_state_cost_pairs` function, and other given parameters. See the explanation of collate functions in Subsection 5.3.11.
- 4. Loads a GNN Model (the instance of the`AddModel` or the `MaxModel` class) from a provided PyTorch checkpoint. See details about the GNN Model in Section 5.2.
- 5. Creates a trainer instance of the Pytorch Lightning `Trainer` class.

- 6. Tests the model from step 4 with the dataloader from step 3 using the method `test` of the instance of the Pytorch Lightning `Trainer` from step 5.

5.3.6 Planner Predict Mode

The `predict` mode predicts a plan of actions leading the goal state and their total cost in Fast Downward Planner's SAS plan format (see the example of FD Planner's plan in Listing 2.3). The method `predict` is method of the `Gnn` class instance. It is used by main scripts (see the description of Executable Main Scripts in Subsection 5.3.2). The method takes one argument of the `PredictConfig` class instance (class from the `configs.py` file in the`Program/Code` folder). This container class captures all arguments passed by the caller. Then, the following sequence of operations in method is executed:

- 1. Creates a unique number to put in the name of every temporary file.
- 2. Runs Fast Downward Planner as a subprocess to create a translation of given PDDL files into a SAS output file. See details about the SAS format and the SAS output file in Section 2.3.
- 3. Parses created SAS file from step 2 into an instance of the `SasData` class using the functionality of the `SasFileParser` class instance. See details about the `SasFileParser` class in Subsection 5.1.5.
- 4. Reads given PDDL files and captures them in a `PddlData` class instance using the functionality of the `PddlFileReader` class instance. See details about the `PddlFileReader` class in Subsection 5.1.6.
- 5. Loads a GNN Model (the instance of the `AddModel` or the `MaxModel` class) from a provided PyTorch checkpoint. See details about the GNN Model in Section 5.2.
- 6. Creates an instance of the `SasPlanData` class using the instance of `SasData` class from step 3, the instance of `PddlData` class from step 4, the GNN Model from step 5, and the functionality of the `SasPlanPreditor` class instance. See details about the `SasPlanPreditor` class in Subsection 5.3.13.
- 7. Writes the instance of the `SasPlanData` class from step 6 into the Fast Downward Planner's SAS plan format using the instance of the `SasPlanWriter` class. See details about the `SasPlanWriter` class in Subsection 5.3.15.
- 8. Cleans up all temporarily created files.

5.3.7 GNN Data JSON Reader

The main purpose of the `GnnDataJsonReader` class is to provide the functionality to load an instance of the `GnnData` class from a given JSON file.

Call the `read` method of the `GnnDataJsonReader` class instance to read the data from a given JSON file. The implementation uses the `json.load` function to read a JSON string that then is converted using the functionality of the `GnnDataJsonDecoder` class instance into an instance of the `GnnData` class and returned. See the description of the JSON file format in Subsection 5.1.9. See the example of the JSON file in Listing 5.1 (three dots represent omitted cost-state pairs).

An alternative class is described in Subsection 5.3.8.

5.3.8 GNN Data TXT Reader

The main purpose of the `GnnDataTxtReader` class is to provide the functionality to load an instance of the `GnnData` class from a given TXT file.

Call the `read` method of the `GnnDataJsonReader` class instance to read the data from a given TXT file. The method uses appropriate instance methods to read sections of the `GnnData` class instance properties in the TXT file, creates an instance of the `GnnData` class and returns it. The format description is the same as in Subsection 5.1.9, but in the TXT form. See the example of the TXT file in Listing 5.2 (three dots represent omitted cost-state pairs).

An alternative class is described in Subsection 5.3.7.

5.3.9 GNN Dataset

The main purpose of the `GnnDataset` class is to provide the functionality to load an instance of the `GnnData` class from a given folder from all files that are of the `GnnData` form, transform every of them into an instance of the `GnnDatasetEntry` class using an instance of the `GnnDatasetEntryAdapter`

class and aggregate them within itself in one huge dataset. See details about the `GnnDatasetEntryAdapter` class in Subsection 5.3.10.

Call the `GnnDataset` constructor method with path of folder with saved JSON or TXT files of the `GnnData` class form to create the `GnnDataset` class instance. Property `state_cost_pairs` contains all aggregated state-cost pairs from all files from the folder. Property `predicate_arity_tuples` captures all predicate arities.

5.3.10 GNN Dataset Entry Adapter

The main purpose of the `GnnDatasetEntryAdapter` class is to provide the functionality to transform a given instance of the `GnnData` class into an instance of the `GnnDatasetEntry` class.

In this part of the code, transformation of states happens (creation of $\mathcal{L}_G$-structures). To every GNN state in the `cost_state_pairs` variable it adds the `facts` and the `goals` (all variables are from the `GnnData` class instance). Relational structure representation is described in Section 5.2.

Call the `adapt` method of the `GnnDatasetEntryAdapter` class instance to adapt a given instance of the `GnnData` class into an instance of the `GnnDatasetEntry` class.

5.3.11 Collate Functions

All collate functions are implemented in the `gnn_dataset` file in the `Program/Code` folder. The functions are used to do transformation of data (or batch of data) before sending it into the `forward` function of some AI model.

In this work, collate function passed into the PyTorch `DataLoader` (see Subsections 5.3.3, 5.3.4, or 5.3.5), so that every given batch of relational structures will be collated into one tuple consisting of a combined relational structure and a list of the maximum object numbers (GNN input). Also collate function is used in predicate mode of the planner to use `forward` method of GNN Model to produce cost (see Subsection 5.3.14). Tuple consisting of

a combined relational structure and a list of the maximum object numbers (GNN input) representation is described in Section 5.2.

5.3.12 Trainer Setup Method

Trainer setup method `load_trainer` is implemented in class the `Gnn`. The main purpose of the `load_trainer` is to set up the PyTorch Lightning `Trainer` class instance. The method sets up output folder name for trained models, output content of trained models, early stopping criteria, checkpoint criteria, and logger. Then, it returns an instance of `Trainer` class.

5.3.13 SAS Plan Predictor

The main purpose of the `SasPlanPredictor` class is to predict a plan of actions leading the goal state and their total cost in the Fast Downward Planner's SAS plan format (see the example in Listing 2.3), save it in an instance of the `SasPlanData` class, and return it. Also see details about the SAS format and the SAS output file in Section 2.3.

It implements Algorithm 3 described in Section 4.4.

Call the `predict` method of the `SasPlanPredictor` class instance to get predicted SAS plan. It uses the `get_all_neighbor_info_of_sas_state` instance method to find all accessible neighbor SAS states of the current SAS state with their predicted cost, then follows neighbor SAS state with the lowest predicted cost while predicate with objects used to transition into this neighbor SAS state is saved into the plan. To predict cost of some SAS state an instance of the `SasStateCostPredictor` class is used. See details about the `SasStateCostPredictor` class in Subsection 5.3.14.

5.3.14 SAS State Cost Predictor

The main purpose of the `SasStateCostPredictor` class is to predict a cost of some given SAS state. See details about the SAS format and the SAS output file in Section 2.3.

Call the `predict_costs` method of the `SasStateCostPredictor` class instance to get predicted costs of given SAS states. It uses a given `SasData` class instance and a given `PddlData` class instance to get `GnnData` states from given SAS states (instance methods from class `GnnDataAdapter`, see Subsection 5.1.8), then it is transformed into `GnnDataset` states (instance methods from `GnnDatasetEntryAdapter`, see Subsection 5.3.10), later it collated into one state (GNN Input) using function `collate_gnn_dataset_state` (see the explanation of collate functions in Subsection 5.3.11, see GNN Input in Section 5.2), finally it is fed to the `forward` method of a given GNN Model (see details about the GNN Model in Section 5.2) to get predict costs of all SAS states encoded in collated state. Take a look at Figure 5.4.

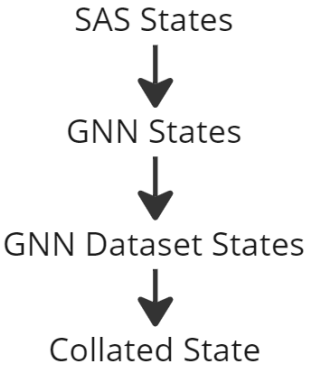


Figure 5.4: States trasformations before passing into GNN Model's forward method.

5.3.15 SAS Plan Writer

The main purpose of the `SasPlanWriter` class is to provide the functionality to write instance of the `SasPlanData` class into the Fast Downward Planner's SAS plan format. See the example in Listing 2.3. See details about the SAS plan in Section 2.3.

Call the `write` method of the `SasPlanWriter` class instance to write given data into a file.

Part III

Experiments

This part of the work reproduces the results conducted by the original work [SBG22] using GNN Data Generator, GNN Model, and GNN Planner. It consists of two chapters: the first chapter (Chapter 6) discusses what to use to do experiments with the generator, the model, and the planner; the second chapter (Chapter 7) provides instructions on how results from [SBG22] work were reproduced.

The results obtained in this study were made possible only due to the availability of high-performance computing resources, specifically the Graphical Processing Units (GPUs) provided by the university (CTU RCI Cluster[9]). The author acknowledges the support of the OP VVV-funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" and the author is also grateful for the access to the computational infrastructure of the project.

[9] `https://login.rci.cvut.cz/wiki/start` [accessed 13 May, 2023]

Chapter 6

Experiments Flow

To measure GNN Model results, we compare Fast Downward Planner's plan and GNN Planner's plan for every specific domain instance problem. See details in Section 6.1.

We use Python scripts to automate and speed up work with GNN, which we run on a server with sizeable computational potential. See details in 6.2.

6.1 Resultor

An executable Python script named `resultor.py` is located in the root folder `GNN`. The purpose of this script is to create actual and predicted plans of domain instance problems, compare them, and save results into a `CSV` table. The script has three modes: generate, examine, and read. See `README_RESULTOR.md` file for the user manual and all parameters description. See details about modes in Subsections 6.1.1, 6.1.2, and 6.1.3.

6.1.1 Resultor Generate Mode

The main purpose of this script mode is to create Fast Downward Planner's plan and GNN Planner's plan for every domain instance problem using

provided trained GNN Model. This way, every instance will have three plans: Fast Downward Planner's plan, GNN Planner's add model plan, and GNN Planner's max model plan. If there is some plan missing, it means the corresponding tool failed.

If Fast Downward Planner's plan already exists for the corresponding domain instance problem, the script does not rerun Fast Downward Planner. `FastDownwardResults` folder consists of precalculated Fast Downward Planner's plans for every domain instance problem for every domain problem. These plans could be used as starting point for any new results folder.

6.1.2 Resultor Examine Mode

The main purpose of this script mode is to compare plans for every domain instance problem for every domain problem and save results into a `CSV` table. It uses the following logic: if the predicted plan costs less or equal to the actual plan - it is an optimal plan; otherwise, it is a suboptimal one. If some plan out of three is missing for a specific domain instance problem, it assumes the corresponding tool failed and saves the failure in the table.

6.1.3 Resultor Read Mode

The main purpose of this script mode is to read saved `CSV` table.

6.2 Jober

Generation of GNN input files, then training of GNN `add` and `max` models, and finally, the creation of results (plans for every domain instance problem) for every domain problem requires significant computation power and ability. For this reason, CTU RCI Cluster[1] was used. It has many CPUs and GPUs with a large amount of memory. The server uses the `SLURM` scheduler, which uses a technique of submitting `jobs` (a `job` is a small bash script with all the cluster metadata and commands to be run). See the `README_CTU_RCI_CLUSTER` file for the user manual.

[1] `https://login.rci.cvut.cz/wiki/start` [accsssed 13 May, 2023]

The Python script `jober.py` has functions that could be run on the server. Master functions are:

- `create_and_run_jobs_single_mode_all` function creates and runs GNN input files generation jobs for every domain instance problem for every domain problem type from `Data/Pddl` folder.
- `create_and_run_jobs_train_mode_all` function creates and runs GNN `add` and `max` model training jobs for every domain problem.
- `create_and_run_jobs_results_all` function creates and runs GNN plan creation jobs for every GNN Model type for every domain instance problem for every domain problem type.

Chapter 7

Results

In this chapter, we discuss GNN hypersetup (see Section 7.1), then two experiments (see Sections 7.2 and 7.3), and finally, expirements' result tables (see Section 7.4).

7.1 Hypersetup

Results below use hypersetup from [SBG22] work:

- k hypervalue is 32 (GNN Planner's training mode parameter called `--hidden_size`); number of dimensions of feature vectors in Algorithm 2
- L hypervalue is 30 (GNN Planner's training mode parameter called `--iterations`); number of layers in Algorithm 2
- Learning rate is 0.0002 (GNN Planner's training mode parameter called `--learning_rate`)
- L1 regularization set to 0.0001 for GNN `add` model and 0.0 for GNN `max` model (GNN Planner's training mode parameter called `--l1_factor`)
- Maximum steps of predicted plan is 100 (GNN Planner's prediction mode parameter called `--step_limit`)

Every domain instance problem data were generated until the generated data met the criteria of either 4 hours of generation, 20000 generated states, or 100 unsuccessful in-row tries to generate a new state.

All the models were trained till the model meets criteria of either 24 hours of training or 1000 epochs of training.

7.2 Experiment 1

The first experiment reproduces results obtained by [SBG22] work. In this experiment, we work with full SAS state space (see the explanation of full SAS state space in 2.3). The results of this experiment are saved in the `results_full.csv` file and shown in Table 7.1a in Section 7.4.

To produce results with full SAS state space data, conduct the following sequence of actions:

- Generate data for all Transport domain instance problems from `Data/Pddl/Transport` folder using GNN Data Generator with `fastdownward-pruning False` parameter and add it to already provided other data by [SBG22] (`Data/JsonFull` folder).
- Train GNN `add` and `max` models for every domain problem (`TrainedModelsFull` folder) using data from `Data/JsonFull.../Train` and `Data/JsonFull.../Validation` folders.
- Generate plans (`ResultsFull` folder) for every domain instance problem (`Data/JsonFull/.../Test` folder) using GNN Planner predict mode with `fastdownward-pruning False` parameter.
- Create a result table by comparing true plans created by Fast Downward Planner (`FastDownwardResults` folder) and predicted plans from `ResultsFull` folder.

7.3 Experiment 2

The second experiment is the same as the first one, but it is conducted in the pruned SAS state space (see the explanation of pruned SAS state space in Sec-

tion 2.3). The results of this experiment are saved in the `results_pruned.csv` file and shown in Table 7.1b in Section 7.4.

To produce results with pruned SAS state space data conduct the following sequence of actions:

- Generate data for all domain instance problems for all different problems from `Data/Pddl` folder using GNN Data Generator with `fastdownward-pruning True` parameter (`Data/JsonPruned` folder).
- Train GNN `add` and `max` models for every domain problem using data from `Data/JsonPruned.../Train` and `Data/JsonPruned.../Validation` folders (`TrainedModelsPruned` folder).
- Generate plans for every domain instance problem (`Data/JsonFull/.../Test` folder) using GNN Planner predict mode with `fastdownward-pruning True` parameter (`ResultsPruned` folder).
- Create a result table by comparing true plans from `FastDownwardResults` folder created by Fast Downward Planner and predicted plans from `ResultsPruned` folder.

7.4 Result Tables

In Table 7.1a, we see that the results of Experiment 1 almost match the results obtained by [SBG22] work. The only difference is that we got worse results for the `Rover` domain. A sampling error likely causes it.

In Table 7.1b, we see that the results of Experiment 2 are slightly better than the results of Experiment 1. Experiment 2 uses pruned SAS state space, which benefits Fast Downward Planner's translation part in GNN Data Generator. After pruning, unnecessary SAS variables and unnecessary SAS operators are omitted, so more potentially goal-leading states are generated (see SAS state space pruning in Section 2.3). It helps GNN Model to learn more correctly.

Domain(#)	GNN-SUM: Optimal	GNN-SUM: Suboptimal, Failed GNN	GNN-MAX: Optimal	GNN-MAX: Suboptimal, Failed GNN
BlocksClear (11)	10	1 + 0 (1)	11	0 + 0 (0)
BlocksOn (11)	11	0 + 0 (0)	11	0 + 0 (0)
Gripper (39)	39	0 + 0 (0)	39	0 + 0 (0)
Logistics (8)	8	0 + 0 (0)	8	0 + 0 (0)
Miconic (95)	95	0 + 0 (0)	95	0 + 0 (0)
ParkingBehind (32)	29	3 + 0 (3)	32	0 + 0 (0)
ParkingCurb (32)	32	0 + 0 (0)	32	0 + 0 (0)
Pathing (6)	3	3 + 0 (3)	6	0 + 0 (0)
Rover (26)	2	4 + 20 (24)	16	2 + 8 (10)
Satellite (20)	20	0 + 0 (0)	20	0 + 0 (0)
Transport (20)	20	0 + 0 (0)	20	0 + 0 (0)
Visitall (12)	12	0 + 0 (0)	12	0 + 0 (0)
Total (312)	281	11 + 20 (31)	302	2 + 8 (10)
Total (%)	90.06%	9.94%	96.79%	3.21%

(a) : Results with Full SAS Space

Domain(#)	GNN-SUM: Optimal	GNN-SUM: Suboptimal, Failed GNN	GNN-MAX: Optimal	GNN-MAX: Suboptimal, Failed GNN
BlocksClear (11)	11	0 + 0 (0)	11	0 + 0 (0)
BlocksOn (11)	11	0 + 0 (0)	11	0 + 0 (0)
Gripper (39)	39	0 + 0 (0)	39	0 + 0 (0)
Logistics (8)	7	1 + 0 (1)	8	0 + 0 (0)
Miconic (95)	95	0 + 0 (0)	95	0 + 0 (0)
ParkingBehind (32)	27	5 + 0 (5)	32	0 + 0 (0)
ParkingCurb (32)	31	1 + 0 (1)	32	0 + 0 (0)
Pathing (6)	6	0 + 0 (0)	6	0 + 0 (0)
Rover (26)	17	0 + 9 (9)	25	1 + 0 (0)
Satellite (20)	20	0 + 0 (0)	20	0 + 0 (0)
Transport (20)	20	0 + 0 (0)	20	0 + 0 (0)
Visitall (12)	12	0 + 0 (0)	12	0 + 0 (0)
Total (312)	296	7 + 9 (16)	311	1 + 0 (1)
Total (%)	94.87%	5.13%	99.68%	0.32%

(b) : Results with Pruned SAS Space

Table 7.1: Results

Chapter 8

Conclusion

In this bachelor's thesis, our primary objective was to explore the various methods and practical applications of Graph Neural Networks (GNNs) in Classical Planning, as suggested by the work referenced [SBG22].To achieve our goals, we undertook a series of essential steps.

Firstly, we presented a clear and concise explanation of the potential applications of GNNs in Classical Planning. This foundational understanding served as a solid starting point for our subsequent work.

Next, we focused on the implementation aspect of our research. We developed a GNN Data Generator, which allowed us to generate suitable datasets for training and evaluation purposes. This tool streamlined the data preparation process and facilitated efficient experimentation.

Furthermore, we refactored and modified the existing codebase of the GNN Model, ensuring its robustness and compatibility with our research objectives. By optimizing and enhancing the code, we improved the overall performance and functionality of the GNN Model, making it more suitable for our specific requirements.

In addition to code refactoring, we made implementations and reimplementations to the GNN Planner, tailoring it to incorporate GNN-based techniques and algorithms. These implementations and reimplementations allowed us to leverage the power of GNNs for planning tasks, enabling more effective and intelligent decision-making processes. One of our key contributions was

implementing a Value-Function-based prediction mode for the planner.

Finally, we successfully reproduced the results obtained in the experiments conducted in the work referenced [SBG22]. This achievement validates our research methodology and implementation, further reinforcing the reliability and credibility of our findings.

Overall, our work has made notable advancements in Classical Planning by exploring the applications and methodologies of GNNs. Our contributions include the development of a GNN Data Generator, code refactoring of the GNN Model, modifications to the GNN Planner, implementation of a Value-Function-based prediction mode, and successful replication of previous experimental results, all following [SBG22].

The outcomes of this thesis provide valuable insights into the potential of GNNs applications in Classical Planning, paving the way for future research and innovation in this area. By harnessing the power of GNNs, we can expect more efficient and intelligent planning systems that can address complex real-world problems effectively.

Appendices

Appendix A

Acronyms

AI Artificial Intelligence. 7, 27, 55

FD Fast Downward. 11, 14, 28, 29, 32, 38, 39, 40, 41, 50, 53, 56, 57, 63, 64, 68, 69

GNN Graph Neural Network. 1, 2, 5, 11, 19, 20, 25, 27, 29, 30, 31, 32, 37, 38, 42, 47, 49, 51, 52, 53, 55, 56, 57, 61, 63, 64, 65, 67, 68, 69, 71, 72

NN Neural Network. 1, 17, 18, 19, 20

PDDL Planning Domain Definition Language. 1, 11, 28, 29, 32, 39, 40, 41, 42, 53

VF Value Function. 20, 27, 30, 38

Appendix B

Bibliography

[AI23] Lightning AI, *Pytorch lightning documentation [online] [https://lightning.ai/docs/pytorch/stable/] [accessed 29 apr, 2023]*, 29 April 2023.

[BKM+20] Pablo Barceló, Egor V. Kostylev, Mikaël Monet, Jorge Pérez, Juan L. Reutter, and Juan Pablo Silva, *The logical expressiveness of graph neural networks*, 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020, OpenReview.net, 2020.

[BN95] Christer Bäckström and Bernhard Nebel, *Complexity results for SAS^+ planning*, Computational Intelligence **11** (1995), no. 4, 625–655.

[Bou23] Daniel Bourke, *Zero to mastery learn pytorch for deep learning [online] [https://www.learnpytorch.io/] [accessed 27 apr, 2023]*, 23 April 2023.

[Con23] PyTorch Contributors, *Pytorch documentation [online] [https://pytorch.org/docs/stable/index.html] [accessed 29 apr, 2023]*, 29 April 2023.

[EF95] Heinz-Dieter Ebbinghaus and Jörg Flum, *Finite model theory*, Perspectives in Mathematical Logic, Springer, 1995.

[Fou23] Python Software Foundation, *Python 3.11.3 documentation [online] [https://docs.python.org/3/] [accessed 29 apr, 2023]*, 29 April 2023.

[GKW+98] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden,

Scott Penberthy, David Smith, Ying Sun, and Daniel Weld, *Pddl - the planning domain definition language*.

[Gre23] Adam Green, *Planning.wiki - the ai planning & pddl wiki [online] [https://planning.wiki/guide] [accessed 26 apr, 2023]*, 26 April 2023.

[Hel06] M. Helmert, *The fast downward planning system*, Journal of Artificial Intelligence Research **26** (2006), 191–246.

[Lib04] Leonid Libkin, *Elements of finite model theory*, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.

[Lip14] Nir Lipovetzky, *Structure and inference in classical planning*, Lulu.com, Morrisville, NC, September 2014.

[Mar13] Radek Marik, *Classical planning and scheduling slides [online] [https://cw.fel.cvut.cz/old/_media/courses/ae3b33kui/lectures/lecture_09_10.pdf] [accessed 26 apr, 2023]*, 16 April 2013.

[RN10] Stuart Russell and Peter Norvig, *Artificial intelligence: A modern approach*, 3 ed., Prentice Hall, 2010.

[SBG22] Simon Ståhlberg, Blai Bonet, and Hector Geffner, *Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits*, Proceedings of the International Conference on Automated Planning and Scheduling **32** (2022), no. 1, 629–637.

[SGT+09] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini, *The graph neural network model*, IEEE Transactions on Neural Networks **20** (2009), no. 1, 61–80.

[ZLLS21] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola, *Dive into deep learning*, arXiv preprint arXiv:2106.11342 (2021).