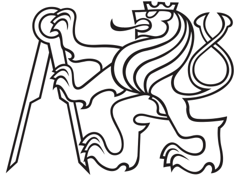
Bachelor's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Architectures of Neural Networks for Heuristic Functions in General Planning Problems

Vítězslav Šimek

Supervisor: doc. Ing. Tomáš Pevný, Ph.D.
May 2024

BACHELOR‘S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name: **Šimek Vít zslav** Personal ID number: **508482**

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:

Architectures of Neural Networks for Heuristic Functions in General Planning Problems

Bachelor’s thesis title in Czech:

Architektury neuronových sítí pro heuristické funkce v obecných plánovacích problémech

Guidelines:

Hyper-Graph Neural Networks (HGNNs) are a class of neural networks designed to approximate functions over hyper-graphs. General planning problems can be represented as a set of predicates that evaluate to true in an initial state and in a goal state, which can be represented as a hyper-graph. The thesis is to study different methods, how the hyper-graph can be defined and different methods, and how HGNN can be constructed. The student should take the following steps.
1. Study Graph Neural Networks and their training.
2. Search the relevant prior art and learn, how planning problem can be transformed into hyper-graphs and their relation to lifted relational neural network.
3. Learn how to use NeuroPlanner.jl and Mill.jl libraries and understand their internal mechanism.
4. Use NeuroPlanner.jl and Mill.jl to implement different methods using HGNN to implement heuristic function for planning.
5. Experimentally compare implemented architectures on a representative suit of planning problems (ideally from learning track of ISPC challenge).

Bibliography / sources:

[1] Learning Generalized Policies Without Supervision Using GNNs, Simon Stahlberg, Blai Bonet, Hector Geffner, 2022
[2] Asnets: Deep learning for generalised planning, S Toyer, S Thiébaux, F Trevizan, L Xie, 2022
[3] GOOSE: Learning Domain-Independent Heuristics, Dillon Ze Chen and Sylvie Thiebaux and Felipe Trevizan, 2023

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Tomáš Pevný, Ph.D. Artificial Intelligence Center FEE

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

Date of bachelor’s thesis assignment: **16.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

doc. Ing. Tomáš Pevný, Ph.D.	prof. Dr. Ing. Jan Kybic	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

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to express my sincere gratitude to doc. Ing. Tomáš Pevný, Ph.D. for their invaluable guidance, patience and expertise, which played a pivotal role in the completion of this work. I would also like to thank my family and friends for their support on my journey through discovering artificial intelligence.

Declaration

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

Prague, May 24, 2024

. . . .

Signature

Abstract

This bachelor thesis compares the architectures of Neural Networks for heuristic functions in general planning problems. It explores various methods for encoding planning problems and utilizing Graph Neural Networks. Additionally, it proposes several optimizations to enhance learning speed and discusses the advantages and disadvantages of the introduced architectures. A significant portion of the work is dedicated to the implementation of these architectures and their optimizations.

Keywords: planning problems, artificial intelligence, graph neural networks, architectures, optimization

Supervisor: doc. Ing. Tomáš Pevný, Ph.D.
Resslova 307/9
120 00 Praha 2

Abstrakt

Tato bakalářská práce porovnává architektury neuronových sítí pro heuristické funkce v obecných plánovacích problémech. Zkoumá různé metody kódování plánovacích problémů a využití grafových neuronových sítí. Dále navrhuje několik optimalizací pro zrychlení učení a diskutuje výhody a nevýhody zavedených architektur. Významná část práce je věnována implementaci těchto architektur a jejich optimalizací.

Klíčová slova: plánovací problémy, umělá inteligence, grafové neuronové sítě, architektury, optimalizace

Překlad názvu: Architektury neuronových sítí pro heuristické funkce v obecných plánovacích problémech

Contents

1 Introduction 1

2 Background 3
2.1 Classical planning . . . 3
2.2 Planning languages . . . 3
2.3 PDDL formulas . . . 3
2.4 Solving PDDL problems . . . 5
2.5 Graphs . . . 6

3 Introduction to Graph Neural Networks 9
3.1 Feed Forward Neural Networks . . 9
3.1.1 Artificial Neuron . . . 9
3.1.2 Layer . . . 10
3.1.3 Neural Network Architecture 11
3.1.4 Non-linear activation functions 11
3.2 Graph Neural Networks . . . 12
3.2.1 GNN input . . . 12
3.2.2 Message Passing . . . 12
3.2.3 Readout Function . . . 13
3.2.4 Types of problems GNN solves 13

4 Representing State as a Graph 15
4.1 NeuroPlanner . . . 16
4.2 Representing STRIPS state as a graph for GNN models . . . 16
4.3 Definitions . . . 17
4.4 Architectures . . . 17
4.4.1 Object Binary Structure . . . 17
4.4.2 Atom Binary Structure . . . 19
4.4.3 Object-Atom Binary Structure 19
4.4.4 Object-Pair Binary Structure 22
4.5 Expressiveness . . . 23

5 Implementation details 25
5.1 Graph Connectivity Representation . . . 25
5.1.1 Adjacency Matrix . . . 25
5.1.2 Adjacency List . . . 26
5.1.3 Scattered Bags . . . 26
5.1.4 Compressed Bags . . . 27

6 Discussion 29
6.1 Edge Representation . . . 29
6.1.1 Adjacency matrix . . . 29
6.1.2 Adjacency list . . . 29
6.1.3 Scattered Bags . . . 29
6.1.4 Compressed Bags . . . 30
6.2 Multi graph or edge features . . . 30
6.3 Nullary and Unary Predicates . . 30
6.4 Architectures . . . 32

7 Experimental section 33
7.1 Preliminary experiment . . . 33
7.2 Optimizations . . . 34
7.3 Preparation . . . 35
7.3.1 Architecture implementations 35
7.3.2 Used Cluster . . . 36
7.3.3 Search for hyper parmeters . . 36
7.3.4 Dataset . . . 36
7.3.5 Planner . . . 36
7.4 Results . . . 36

8 Conclusion 39

Bibliography 41

A Attached files 43

Figures

2.1 Graphs 6
2.2 Hypergraph 7
2.3 A hypergraph and its equivalent bipartite graph. Nodes of hypergraph are nodes on the left of the bipartite graph, and hyperedges are represented as nodes on the right side of the bipartite graph. Node v is connected to the node e in the bipartite graph if node v occurs in the hyperedge e in the hypergraph. 8

4.1 Encoding of the STRIPS defined in 4.4 using the Object Binary structure 18
4.2 Encoding of the STRIPS state defined in 4.4 using the Atom Binary structure 20
4.3 Encoding of the STRIPS state defined in 4.4 using the Object-Atom Binary structure 21
4.4 Encoding of the STRIPS state defined in 4.4 using the Object-Pair Binary structure 23

5.1 Graph $G = \langle V, E \rangle$ 26
5.2 Adjacency matrix of the graph G defined in 5.1 26

6.1 Graphs encoding two types of edge feature vectors 31

Tables

7.1 Finished number of combinations over the different seeds 34
7.2 The proportion of solved problems 34
7.3 Average time in μs of transforming a state of given problem into a graph using scattered bags format (SBags), compressed bags format using a vector (CBags) and compressed bags format using a matrix (CBMat) . . 35
7.4 Number of finished training and testing of GNN models using different combinations of hyperparameters . 37
7.5 The proportion of solved problems in given time 37

Chapter 1
Introduction

Solving planning problems has become another field where artificial intelligence find use. The heuristic function is one of the main concepts for effective solving of planning problems. Graph neural networks are able to learn this function when provided a set of solved planning problems. They can be then used as a function to provide heuristic value for states for search algorithms. However, it is not clear how to encode a PDDL state into graph structure. The focus of this thesis is on how a PDDL state can be encoded and comparison of such encodings. We want to see the strengths and weaknesses of different encodings and find out on what makes an encoding great. To achieve this, we add implementations of encodings to the NeuroPlanner.jl [1] Julia library using structures from Mill.jl [2] Julia library. For qualitative results we will use problems provided from the International Planning Competition from the International Conference on Automated Planning and Scheduling, where the most revered researchers cooperate and present their ideas about planning and scheduling.

Chapter 2

Background

2.1 Classical planning

In this section we would like to introduce a classical planning and core features for definition of planning problems and plans. Furthermore we will take a look at learning how to clear a planning problem.

2.2 Planning languages

Through out the years of development of artificial intelligence there has been several languages for defining classical planning tasks. In this section we would like to introduce some of them.

In the beginning the first planning tasks were introduced using the STRIPS language. Planner using Stanford Research Institute Problem Solver (STRIPS) language gave Shakey [3], the robot, the ability to execute commands and follow given plan. STRIPS language was then used as a base language for most languages used to describe planning problems.

One of the advancements of STRIPS language is Action Description Language (ADL). ADL schema consists of action name, an optional parameter list, and four optional groups of clauses labeled PRECOND, ADD, DELETE, and UPDATE. With this advancement of STRIPS language the ADL overcomes the limited expressiveness and semantic difficulties of the STRIPS language [4].

Another advancement and the one we will be using in this thesis is called Planning Domain Definition Language (PDDL) released in 1998 [5]. PDDL has since its release become a community standard for the representation of planning domains and problems [6]. PDDL was an attempt to standardise planning languages and is used as a standard on many intenational planning competitions.

2.3 PDDL formulas

An object is a component used to define the entities that exist, they are specific instances of types. There can be multiple types of objects and multiple

instances of one type of object.

A predicate is a tuple of types of variables. Each predicate can have its variables bounded by objects and therefore describe a part of the state of the world. An atom over a set of objects O is an expression of the form $P(o_1, \ldots, o_n)$ where P is an n-ary predicate symbol and each $o_i \in O$.

Consider a set of predicate symbols P and a set of objects O. A state $S = \langle O, \langle P^S \mid P \in R \rangle \rangle$ where $P^S \subseteq O^n$ is the interpretation of the n-ary predicate symbol $P \in R$.

The atom $P(o_1, \ldots, o_n)$ holds in state S iff $\langle o_1, \ldots, o_n \rangle \in P^S$.

A domain represents a world structure, defines types of objects and their connections and actions for state transitions. Domain is defined as a tuple $D = (N, R, A, T)$ where N denotes the domain name. R is a finite set of predicate symbols P with possibly different arity n. A represents a set of actions.

An action in PDDL is a tuple:

$$a = \langle : \text{name}, : \text{parameters}, : \text{precondition}, : \text{effect} \rangle$$

An action defines a transition from state S to state S'. The $:$ parameters define a list of variables upon which the particular action operates. The $:$ precondition is a list of atoms that must hold in S for the action to be applicable. The $:$ effect describes the transition from S to S'. All variables must be bound for an action to be applicable [5]. A transition function f maps state S using action a to state S'.

$$S' = f(S, a)$$

A domain definition of sokoban looks like this:

```
(define (domain sokoban)
(:requirements :typing)
(:types location direction box)

(:constants down up left right - direction)

(:predicates
             (at-robot ?l - location)
             (at ?o - box ?l - location)
             (adjacent ?l1 - location ?l2 - location
                ?d - direction)
             (clear ?l - location)
)

(:action move
:parameters (?from - location ?to - location
                ?dir - direction)
:precondition (and (clear ?to) (at-robot ?from)
                (adjacent ?from ?to ?dir))
```

```
:effect (and (at-robot ?to) (not (at-robot ?from)))
)


(:action push
:parameters  (?rloc - location ?bloc - location
    ?floc - location ?dir - direction ?b - box)
:precondition (and (at-robot ?rloc) (at ?b ?bloc)
                (clear ?floc) (adjacent ?rloc ?bloc ?dir)
                (adjacent ?bloc ?floc ?dir))

:effect (and (at-robot ?bloc) (at ?b ?floc)
                (clear ?bloc) (not (at-robot ?rloc))
                (not (at ?b ?bloc)) (not (clear ?floc)))
)
)
```

In this domain there are three types of objects - a location, a direction and a box. There are already defined objects, instances of type direction. There are 4 predicates that define a state of the world. We can see 2 actions, move and push, that define a transition from one state to another.

Problem is a tuple $P = \langle D, O, I, G \rangle$ where

- D is a problem domain
- O is a set of objects
- I is an initial state
- G is a goal state

A plan $\langle a_1, \ldots, a_n \rangle$ is a tuple of n actions, that defines a transition from state S to state S'.

$$f(S, a_1) = S_1$$

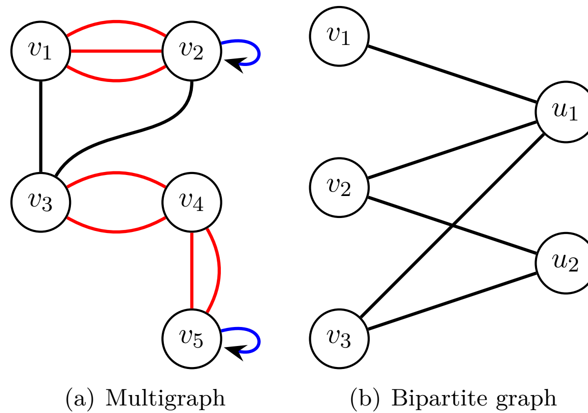
$$S_{i+1} = f(S_i, a_i)$$

$$S_n = S'$$

An optimal plan is a tuple with minimal number of actions, that define a transition from state S to S'.

2.4 Solving PDDL problems

Solving PDDL problems is a task that many researchers cooperate in. The main goal of a researcher is to implement a planner that solves as many problems as it can in the least amount of time. Planners optimized by neural network firstly learn the domain architecture and train on already solved problems.



(a) Multigraph (b) Bipartite graph

Figure 2.1: Graphs

A PDDL problem is solved if there is a plan that makes a transition from an initial state to a goal state. There are two types of planners, ones that satisfy the goal state and try to find a solution in the least amount of time. And the others optimizing planners that search to find the optimal plan.

A heuristic function is a mapping of states into real valued numbers, called heuristic values. This mapping is used for finding a path to the goal state. A heuristic value of a given state represents an approximation of the cost to the goal state from the given state. This means that the lower the heuristic value of a state is that much closer to the goal state it is. However this is only an approximation and not necessarily correct. On the other hand this gives planners a valuable information and helps them reach the goal state in shorter amount of time.

A planner is a traversal algorithm that is used to find a path in a space of states from the initial state to the goal state. For example a BFS algorithm searches the whole space from the initial point in all direction at the same time, it finds the optimal solution, however might take a lot of time. A^* on the other hand uses heuristic function to traverse state space around the least heuristic value of the states, it prefers states that are based on the heuristic function closer to the goal state [7]. PDDL problems are usually NP hard and we use heuristic function to help us solve them.

2.5 Graphs

Graph G is a pair $G = \langle V, E \rangle$ where V is a set of vertices (nodes) and E is a set of unordered pairs $\{u, v\}$ called edges. If there exists an edge between vertices u and v we say that they are connected.

A directed graph G is a pair $G = \langle V, E \rangle$ where V is a set of vertices and E is a set of ordered pairs of distinct vertices. We say that this ordered pair (u, v) is a directed edge going from u to v. If an edge is allowed to connect the same vertex to itself we call this edge a loop.

A multigraph is a generalization, that allows multiple edges between two

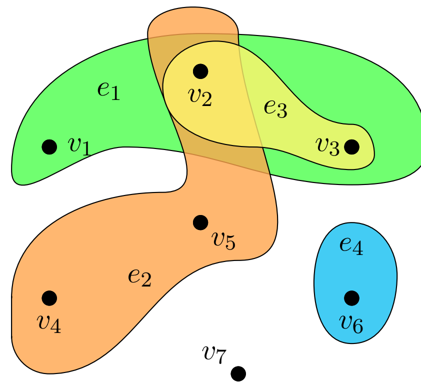
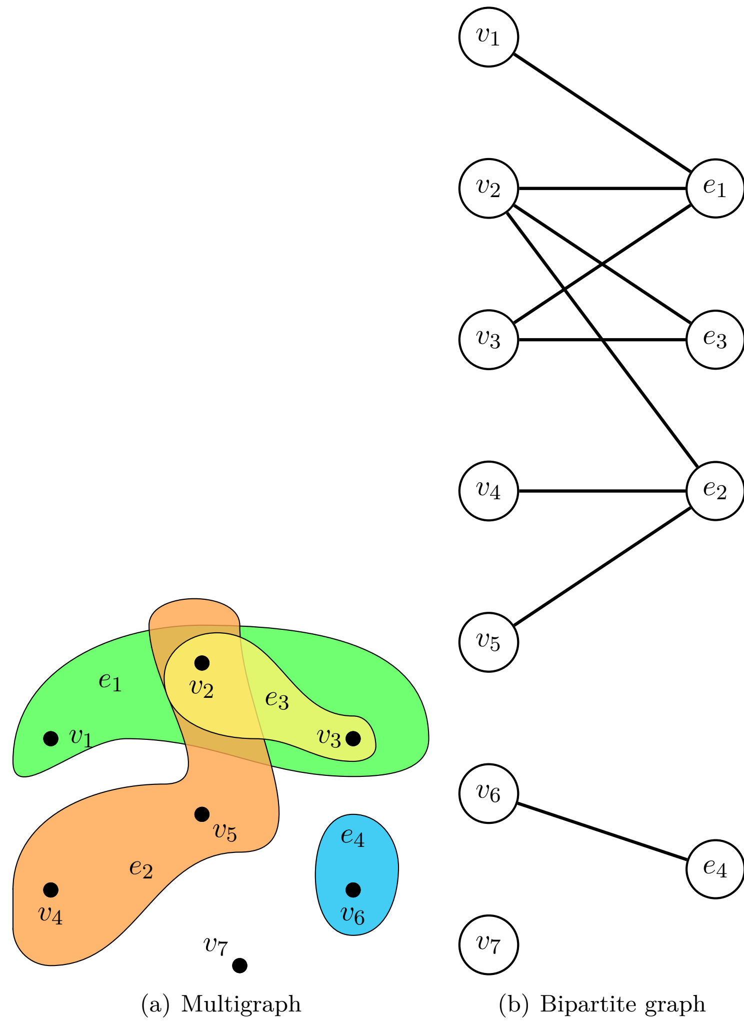


Figure 2.2: Hypergraph

nodes. We can see an example of a multigraph in Figure 2.1(a).

Hypergraph G is a generalization that allows to have edges connecting multiple nodes, we call these edges hyperedges. Hypergraph is defined as a pair $G = \langle V, E_1, \ldots, E_n \rangle$ where V is a set of vertices and E_i is a set of i vertices of V. A hypergraph is shown in Figure 2.2

Bipartite graph G is a tuple $G = \langle U, V, E \rangle$ where vertices are divided into two disjoint and independent sets U and V, every edge in E connects one vertex in U to one in V. Every hypergraph G_H can be transformed into bipartite graph and vice versa. An example of a bipartite graph is shown in Figure 2.1(b). An example of a hypergraph and its equivalent bipartite graphs is shown in Figure 2.3.



(a) Multigraph (b) Bipartite graph

Figure 2.3: A hypergraph and its equivalent bipartite graph. Nodes of hypergraph are nodes on the left of the bipartite graph, and hyperedges are represented as nodes on the right side of the bipartite graph. Node v is connected to the node e in the bipartite graph if node v occurs in the hyperedge e in the hypergraph.

Chapter 3

Introduction to Graph Neural Networks

In this section we will briefly introduce artificial neural networks. They are the core method in deep learning for classification, regression, pattern recognition, data mining, voice recognition and many others. This section will introduce Feed Forward Neural Networks and then further Graph Neural Networks (GNNs), which the advanced methods are build upon.

3.1 Feed Forward Neural Networks

Feed Forward Neural Network is an Artificial Neural Network, which is an affine projection from $\mathbb{R}^d$ to $\mathbb{R}^s$. It is an attempt to recreate biological neural connections in human brain. All the computation human brain conducts lies within its network of neurons. Human brain has approximately 100 billion neurons [8, 9]. Some of them are connected resulting in passing of signals. When one neuron receives a signal it modifies it and sends it to all other neurons that are connected to the neurons end. This way the human brain can receive signals from receptive fields transform them and based on results create assumptions. Through the connected study of biological neurons and machine learning an artificial neural network was introduced [10].

3.1.1 Artificial Neuron

The perceptron became first artificial neuron mimicking work of a biological neuron [11]. Multi-layer perceptron soon became the core concept of artificial neural networks.

An artificial neuron consists of two parts affine and non-linear. The affine projection consists of multiplication and summation and the non-linear part consists of applying any non-linear function on the result of the affine projection. The importance of the non-linearity will be described in the following sections.

Neuron definition:

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T \cdot \mathbf{x} + b)$$

where:

- $\mathbf{x}$ is a vector from $\mathbb{R}^d$
- $\mathbf{w}$ is a vector from $\mathbb{R}^d$
- b is the bias from $\mathbb{R}$, that shifts the result from the origin
- σ is an activation function

Most commonly used activation functions are:

- Binary Step Function

$$\sigma(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

- Sigmoid

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

- Rectified Linear Unit (ReLU)

$$\sigma(x) = \max(x, 0)$$

- Hyperbolic tangent

$$\sigma(x) = \tanh(x)$$

3.1.2 Layer

One layer in neural network is a multitude of neurons. All neurons in one layer are the same, their weights are of the same dimension and their activation functions are the same as well. The number of neurons in a layer defines its output dimension.

We can then define a layer as follows:

$$f(\mathbf{x}) = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$

where:

- $\mathbf{W}$ is a matrix of weights from $\mathbb{R}^{s \times d}$, where i-th row represents weights for the i-th neuron
- $\mathbf{b}$ is a vector of biases from $\mathbb{R}^s$
- σ is an activation function taking in a vector of values and applying element-wise σ activation function

The result of this function is a vector from $\mathbb{R}^s$.

3.1.3 Neural Network Architecture

Neural network is an oredered set of layers, where they are connected to each other. One type of neural network is a multi-layer perceptron, which is defined as a multitude of layers where each layer is fully-connected to the next one, one neuron takes input from all neurons of the previous layer and send its output to all the neurons of the next layer.

All layers in neural network are divided into three groups, the first layer, hidden layers and the last layer.

The first layer is called the input layer. It takes the input of dimension d and transforms it to dimension h.

The last layer is called the output layer. It defines the output dimension of the whole neural network. This layer does not necessarily comply with the non-linear activation function condition. The activation function of the last layer depends on the task. If the task is of regression type the activation function can be identity, which is linear. If the task is of classification type the activation function is commonly softmax, which takes in the output of a whole layer without the inner activation functions as a vector and outputs a normalized vector of the same dimension, generating a distribution.

Advanced neural networks have one or more hidden layers. We call neural networks with many layers deep and with only few layers shallow. Both deep and shallow networks find their usage.

Shallow neural networks have relatively small number of hidden layers or none at all. On one hand they have many advantages they can be trained in a short amount of time, with a small number of training samples and perform really well. On the other hand they can not learn complex data structures. If the task given to the neural network is too hard for its architecture given small number of layers it will not perform well.

Deep neural networks have been in the past years the most researched and used in machine learning. They can learn very complex data structures and complete tasks with nearly perfect accuracy. However they can have billions of hyperparameters to be trained and their training needs large number of training samples. The training takes a lot of time and many optimization methods need to be used to train such network.

3.1.4 Non-linear activation functions

The main concept of non-linear activation functions is scalability with layers. If all activation functions were linear the whole neural network could be reduced into one matrix multiplication and addition of the bias term, with learning only a few of hyperparameters. However with the activation functions being non-linear the network is impossible to be reduced. Therefore the whole neural network has many trainable parameters, such as weights and biases.

3.2 Graph Neural Networks

In the real world data do often have a more complex structure that cannot be defined by a single vector. Such data structures can be represented as graphs. Because there is no common way to represent graphs as vectors and therefore cannot be passed through normal neural networks Graph Neural Networks (GNNs) have been introduced.

One of the common complex structures that are used machine learning are molecules. They consist of atoms of elements together create a molecule, with each atom having different features and each edge having different features as well. GNN models often work on molecules and predict their features. Planning problems can be represented as graphs and GNN models can be learnt to predict paths. Another graph-like structure comes from social networks. Where people are connected to other people by being friends, sharing the same hobby and more. Furthermore a movie recommender is based on GNN models as well, if a person watched only action movies the model will predict that other movies labeled action will be of interest to this person. A sentence can also be defined as a directed graph and therefore GNN models can make prediction on the whole sentence and or on its words.

3.2.1 GNN input

GNN models generally take as input graphs whose vertices and possibly edges are labeled by real-valued feature vectors. These vectors are then processed through a series of parameterized differentiable transformations into the output value [12].

Feature vector is a vector of constant dimension. It contains information about features of one type of element in graph, i.e., contains information about nodes, edges or the whole graph. Its dimension is constant for each type of element. Nodes and edges can have feature vectors of different dimension. We call a vector one-hot, when it contains zeros in all its entries except one, where is a one. We call a vector multi-hot, when the number of entries with a one is greater than one and all the others contain a zero. In this work we will use feature vectors for nodes and edges. multigraph often uses one-hot feature vectors for edges and normal graph often uses multi-hot feature vectors for edges.

3.2.2 Message Passing

In this section we would like to introduce how GNN models work and how they process input. We will define them the same way as they do in the following work [13].

There is no simple matrix multiplication and addition that can be used on graphs as do normal neural networks. GNN models work on a different type of processing. It can be compared to convolutions however it is something different. GNN models are generally based on the message passing

architecture.

GNN models have layers but each layer represents one message pass. In each message pass feature vectors of nodes are updated. The neighbourhood of node v in a graph G is

$$\mathcal{N}_G(v) = \{\{u \mid \langle v, u \rangle \in E\}\}$$

where $\mathcal{N}_G(v)$ can be a multi set (denoted by $\{\{\dots\}\}$) containing multiple instances of one node .

The GNN model maintains node embeddings (feature vectors) $\mathbf{u}^{(i)}(v) \in \mathbb{R}^k$, $\forall\, v \in V$, $i = 0, \dots, L$ where L is a number of message passing layers (iterations). The node embeddings $\mathbf{u}^{(0)}(v), \forall v \in V$ are initially fixed. Message passing layers iteratively update node embeddings with node embeddings of nodes that are in its one-hop neighbourhood with the general message passing equation:

$$\mathbf{u}^{(t+1)}(v) = \mathrm{comb}_t(\mathbf{u}^{(t)}(v), \mathrm{agg}_t(\{\{\mathbf{u}^{(t)}(u) \mid u \in \mathcal{N}(v)\}\}))$$

where combination functions $comb_t$ map pairs of vectors into $\mathbb{R}^k$ vector, aggregation functions agg_t map arbitrary number of $\mathbb{R}^k$ vectors into single one. common aggregation functions are mean, max, sum.

Each aggregation function has some advantages and some disadvantages. Mean function is invariant to the number of nodes in the one-hop neighbourhood, it does not create nodes with higher values in its embeddings. However it does create a smaller differences over many message-passing layers. Max function is invariant to the number of nodes as well and maximizes every embedding. Sum on the other hand does create differences between nodes based on the number of connections it has. This create another feature for nodes and possibly GNN models can learn something more using this aggregation function.

Crucial is how well can agg distinguish two multi sets of features. It is known that agg functions based on sum are more expressive than those based on mean with max being the least expressive [14].

3.2.3 Readout Function

The final layer is called a readout layer and works similarly to the previous layers, however it does not update node embeddings and aggregates all the feature vectors and aggregates them into a single output value:

$$\mathrm{ro}(\{\{\mathbf{u}^{(L)}(v) \mid v \in V\}\})$$

where ro is the readout aggregation function.

3.2.4 Types of problems GNN solves

For graphs there are three general types of prediction: graph-level, node-level and edge-level. Each of these prediction can be either classification or regression and all of them can be solved by GNN models.

Graph-level tasks are generally tasks that classify the whole graph. For example when a new molecule is invented GNN models classify its properties and determine whether the molecule is stable, useful and more. In planning the classification is a little bit different, we want to predict the heuristic function of a state in a given problem.

There are many GNN models that predict on the node-level. Node-level tasks are focused on predicting role or type of each node in a graph. In a sentence each word is represented by a node and can be made prediction on them, for example what kind of parts-of-speech is each word. In social networking GNN models can for example predict whether a person, that newly joined a community will be happy or start with the same hobby as all the other members do.

Edge-level tasks address predicting relationships between objects. If the input to the GNN model is a fully-connected graph with unlabeled edges the output can be labels for each edge. With the example of social networking GNN models can predict whether one person will connect on some level with another person. In biochemistry we can predict whether an atom in a molecule will attach to another atom and with what kind of intensity and with what type of connection.

Chapter 4

Representing State as a Graph

In this section we will introduce the main problem with learning heuristic function for general planning problems. We will introduce some architectures of planning problems.

By an encoding we understand a transformation of PDDL state into a graph representation. It is not clear how to encode one PDDL state and there are multiple ways. Let h denote the encoding of a state and g a GNN model and let them compose into $f = g \circ h$. Consider two states S' and S'' then these two states are not separated by f iff they are distinct and have the same image under f.

$$S' \neq S'' \wedge f(S') = f(S'')$$

The main problem is the state differentiation in neural network, this defines the power of the representation. Neural networks can generate same output for different input, which is problematic. The less the states that are indistinguishable by the neural network the better. There comes in architectures for state encodings.

Each encoding has different advantages and disadvantages. The benchmark for encodings is the number of problems a GNN model using given encoding can solve in given time. This makes the comparison difficult, because there are many factors that influence the transformation of a state into graphs, i.e, memory allocation, computational complexity, the number of indistinguishable states and more.

Implementation of each architecture is very important, we cannot measure two architectures where one encoding of a state is optimized and the other encoding of a state is suboptimal. There are high demands on transformation of states into graphs. We have therefore used profiler to ensure each architecture is well optimized.

GNN models take as input graphs. There are 3 types of information that we need to represent: Node embeddings, edge embeddings and connectivity. The first two are straightforward: node (edge) features will be represented by a matrix $M \in \mathbb{R}^{n \times d}$ where n is the number of vertices (edges) and d is the dimension of feature vectors. Each node (edge) is assigned an index i and its feature vector is stored as a column in $M_{:,i}$. Representing connectivity can be easy and there are many common ways to represent connectivity however, for

our purposes suboptimal. There are several ways such as adjacency matrix or sparse matrix. We will go into detail of this problem in the following chapters.

4.1 NeuroPlanner

All the introduced architectures and other representation formats are realized as an extension of the NeuroPlanner.jl [1] library. It contains other architectures of GNN models and realizes a way to learn GNN models provided with a set of training problems with plans.

4.2 Representing STRIPS state as a graph for GNN models

We define our input into GNN models as a unification of initial and goal states. We transform both states into a graph possibly multi or hyper and then merge them together. This way the GNN models know the initial state and know the goal state and can learn what needs to be transformed to reach the goal state.

Objects are commonly represented as vertices in a graph. However, in the following chapters we will introduce an architecture that builds on representing objects as edges between nodes representing atoms.

There are four possible representations of nullary predicates

- Graph feature
- Hyperedge over all nodes
- Self loops over all nodes
- Feature in all node embeddings

For unary predicates there are less possible representation since they need to be bound to given object.

- Self loops
- Feature in node embedding for given object

Representation of predicates with greater arity is not unified. However, they are commonly represented as either hyperedges connecting multiple nodes or simplified and represented as edges connecting two nodes. One way to simplify it is to create an edge between two nodes if they are both present in the predicate. Another way without simplification is to create a hyperedge over all the nodes that are present creating a hypergraph. Another way is to create a bipartite graph, where all predicates are represented as nodes. This creates two sets of nodes and edges are only between objects and predicates.

Node Features. Construction of feature vectors depends on given architecture and its representation of nullary and unary predicates. Consider an architecture where all nullary and unary predicates are represented as features of a vector. For given state $S = \langle O, \ \langle P^S \mid P \in R \rangle \rangle$ and its graph representation $G = (V, E)$ the feature vector $\mathbf{u} \in \{0, 1\}^n$ of a node $u \in V$ is constructed from nullary and unary predicate symbols P_i as follows: $\mathbf{u}_i = 1$ iff $P_i(u)$ holds in S or P_i is a nullary predicate symbol [12].

Edge Features. For multi and hypergraphs edge embeddings are represented as one-hot encodings. Each edge in graph $G = (V, E)$ representing state $S = \langle O, \ \langle P^S \mid P \in R \rangle \rangle$ represents a predicate and after enumerating predicates the feature vector $\mathbf{u} \in \{0, 1\}^n$ of an edge $e \in E$ is one-hot vector where $\mathbf{u}_i = 1$ iff e represents predicate symbol P_i.

For normal graph each edge represents all the connection between two nodes and thus may represent many predicates. For a state $S = \langle O, \ \langle P^S \mid P \in R \rangle \rangle$ represented by a graph $G = (V, E)$ the feature vector of $\mathbf{u} \in \{0, 1\}^n$ of an edge $e \in E$ is constructed as follows: $\mathbf{u}_i = 1$ for all predicate symbols P that e represents.

4.3 Definitions

For a tuple of objects $\vec{c}$, c_i denotes its i-th element. Let B be a set of objects and $\vec{c}$ a tuple of objects, the expression $B \subseteq \vec{c}$ denotes that all objects in B also occur in $\vec{c}$. For two tuples $\vec{b}$, $\vec{c}$, the expression $\vec{b} \cap \vec{c}$ denotes the set of their shared objects. Analogously, $\vec{b} \cup \vec{c}$ is the set of all objects occurring in at least one of the tuples. Finally, $\vec{b} \triangle \vec{c} = (\vec{b} \cup \vec{c}) \setminus (\vec{b} \cap \vec{c})$ is the symmetric difference of elements occurring in $\vec{b}$ and $\vec{c}$ [12].

4.4 Architectures

In the following sections we will present few architectures from Horcik and Sir [12]. Each encoding converts a state S into either labeled graph, multi-graph or bipartite (multi) graph.

As an example consider a set of predicate symbols, $R = \{N, A, B, U, T\}$, where N nullary predicate symbol, A, B are unary unary predicate symbols, U is binary and T is ternary, a set of objects $O = \{a, b_1, b_2\}$. And a STRIPS state represented as a set of atoms.

$$S = \langle O = \{a, b_1, b_2\}, \ \{N, \ A(a), \ B(b_1), \ B(b_2), \ U(b_1, \ b_2), \ T(a, \ b_1, \ b_2)\} \rangle$$

4.4.1 Object Binary Structure

Given a state $S = \langle O, \ \langle P^S \mid P \in R \rangle \rangle$ defined in 4.4 the encoding of this state using the object binary structure is a graph $G = \langle V, \ E \rangle$ with feature vectors for nodes and edges.

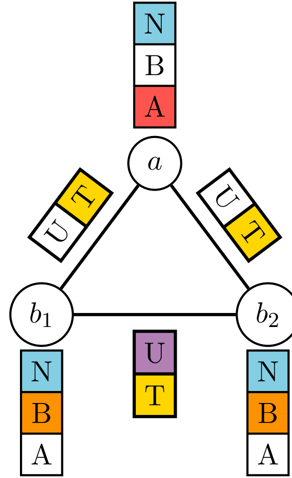


Figure 4.1: Encoding of the STRIPS defined in 4.4 using the Object Binary structure

The transformation maps all objects in O into vertices V in the graph $G = \langle V, E \rangle$. The feature vector $\mathbf{u} \in \{0,1\}^n$ of a node $u \in V$ representing an object $o \in O$ is constructed from nullary and unary predicates $P_i \in R$ as follows: $\mathbf{u}_i = 1$ iff $P_i(o)$ holds in S or P_i is a nullary prediacte. Nullary predicates are shared across all vertices.

$$V = \{v \mid o \in O\}$$

If two objects o_1, o_2 jointly occur in an atom with predicate symbol $R \in P$ of arity at least 2 that holds in S we define a connection between their respective nodes v_1, v_2. We say that this connection represents the predicate R.

$$\langle v_1, v_2 \rangle \in E \text{ iff } \{o_1, o_2\} \subseteq \vec{c} \text{ for some } \vec{c} \in R^S$$

All connections are symmetric by definition, v_1, v_2 might occur in any position in $\vec{c}$. If we consider an architecture using a multigraph, all relations are separately transformed into edges with one-hot feature vectors. The feature vector $\mathbf{u} \in \{0,1\}^n$ of an edge $e \in E$ is one-hot vector where $\mathbf{u}_i = 1$ iff e represents $R_i \in P$.

The encoding of the STRIPS state defined in 4.4 using the object binary structure is shown in Figure 4.1. The nullary and unary predicate symbols are depicted as node feature vectors. Edge feature vector consist of predicate symbols of arity at least 2 and are represented as a feature vector as well. Object a occurs in atom $A(a)$ and also a nullary atom N, likewise objects b_1 and b_2 both occur in atoms $B(b_1)$ and $B(b_2)$ and a nullary atom N. An edge connecting objects b_1 and b_2 has features of both predicates of arity at least two, since both objects occur in atoms $T(a, b_1, b_2)$ and $U(b_1, b_2)$

4.4.2 Atom Binary Structure

This next architecture is the opposite of the previous one. It transforms state $S = \langle O,\ \langle P^S \mid P \in R \rangle\rangle$ into multigraph with nodes representing atoms that hold in S and connections between them represent their contained and shared objects. We connect two atoms if they share common objects. We also connect two atoms if the symmetric difference of objects occurring in them occurs also in another atom.

We create a graph $G = \langle V, E \rangle$, where nodes are atoms that hold in S.

$$U = \{u \mid \alpha \in P^S\}$$

There exist an oriented edge e labeled by $E_{i,j}$ between nodes u, v iff $u = P(\vec{a})$, $v = P'(\vec{b})$ and $a_i = b_j$. Also an edge e is labeled by predicate symbol $\hat{P}$, where $\hat{P}$ is constructed from predicates with arity at least 2 $P \in R$, if the symmetric difference $\vec{a} \triangle \vec{b} \subseteq \vec{c}$ for some $\vec{c} \in P^S$.

Node feature vectors are one-hot and contain predicate symbols, where each atom knows its predicate. Let m be the highest arity of predicate in R. Edge feature vectors are either multi-hot or one-hot using multigraph and are constructed from pairs of indices $\langle i, j \rangle \in [1, m]^2$ and predicate symbols of arity at least 2. We can see that edges are symmetrical, if there is an edge from node u to node v then there is also an edge from v to u. This is also propagated to all the edge features of predicate symbols.

In Figure 4.2 we can see the encoding of the STRIPS state defined in 4.4 using the atom binary structure. Nodes are atoms labeled by one-hot feature vector of predicate symbols. There are oriented edges from nodes with labels describing the common objects and predicate symbols. There is an edge feature vector in the top right corner describing edge features. We can see that there is an oriented edge connecting atom $T(a, b_1, b_2)$ and $U(b_1, b_2)$ labeled by $E_{2,1}$, $E_{3,2}$, since the second object b_1 in $T(a, b_1, b_2)$ is also a first object in $U(b_1, b_2)$. Likewise the third object b_2 in $T(a, b_1, b_2)$ is also a second object in $U(b_1, b_2)$. Note that there is a connection between N and $T(a, b_1, b_2)$ labeled by $\hat{T}$, since if we remove their common objects $\{\}$, which is an empty set, the remaining objects $\{a, b_1, b_2\}$ occur in atom $T(a, b_1, b_2)$.

4.4.3 Object-Atom Binary Structure

The encoding of a state $S = \langle O,\ \langle P^S \mid P \in R \rangle\rangle$ using the object-atom binary structure is a bipartite graph or a hypergraph. However, there is little difference since any hypergraph can be represented as a bipartite graph and vice versa. The implementation depends on the preference, we will further explain details and our implementation in the next sections. We will describe the architecture with a bipartite graph.

Object-atom binary structure is build on both objects and atoms and is represented by a bipartite graph $G = \langle U,\ V,\ E \rangle$. All objects $o \in O$ are mapped into vertices U and all atoms $a \in P^S$ are mapped into vertices V. There is a connection between an object and an atom if the object occurs in

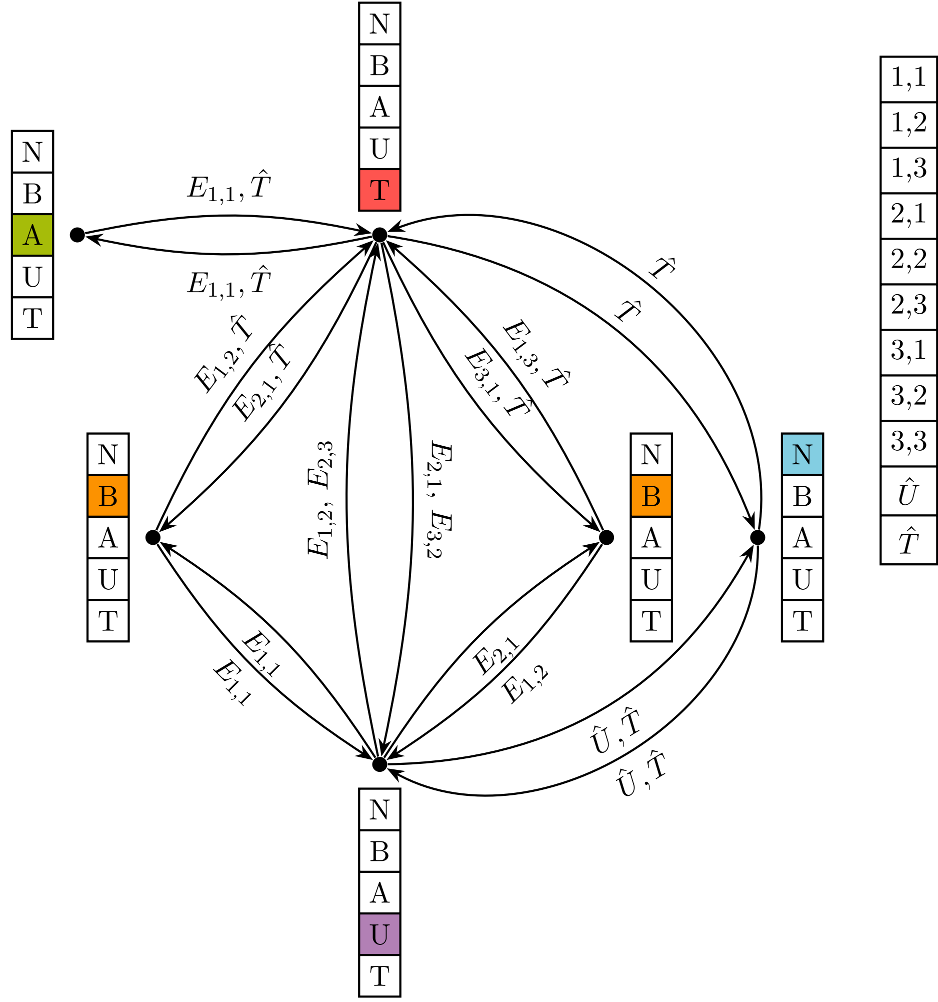


Figure 4.2: Encoding of the STRIPS state defined in 4.4 using the Atom Binary structure

the atom. We add two extra unary predicates C and F to distinguish objects from atoms.

$$U = \{u \mid o \in O\}$$

$$V = \{v \mid \alpha \in P^S\}$$

Set of atoms P_e^S is an enriched set with the unary atoms C and F and a an enriched set of predicate symbols R_e unifies with predicate symbols C, F.

$$S_e = S \cup \{C(o) \mid o \in O\} \cup \{\alpha \mid \alpha \in P^S\}$$

$$R_e = R \cup \{C, F\}$$

The feature vector $\mathbf{u} \in \{0,1\}^n$ of a node $u \in U$ representing object $o \in O$ is constructed from unary predicates $P_i \in R_e$ as follows: $\mathbf{u}_i = 1$ iff $P_i(o) \in P_e^S$.

The feature vector $\mathbf{u} \in \{0,1\}^n$ of a node $v \in V$ representing atom $\vec{a} \in S$ is constructed from predicates $P_i \in P_e^S$ as follows: $\mathbf{u}_i = 1$ iff $\vec{a}$ is an instance of P_i or $P_i = F$.

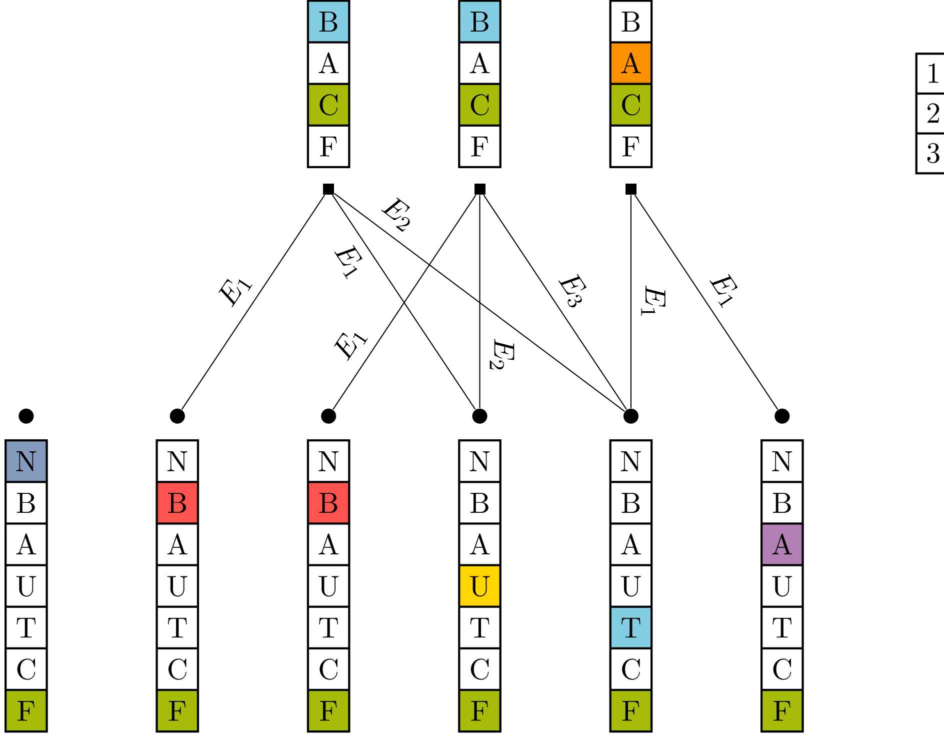


Figure 4.3: Encoding of the STRIPS state defined in 4.4 using the Object-Atom Binary structure

Let m be the highest arity of a predicate in R and for each $i \in [1, m]$ sets of edges E_i representing objects occurring in the i-th position within an atoms. There exists an edge $e \in E_i$ between nodes $u \in U$ and $v \in V$ if the object o represented by node u occurs in the i-th position within an atom $\vec{a}$ represented by v.

$$\langle u,\ v\rangle \in E_i \text{ iff } o = a_i$$

The feature vector $\mathbf{u} \in \{0,1\}^n$ of an edge $e \in E_i$ is one-hot vector where $\mathbf{u_i} = 1$. All edges are the unified into single set $E = \bigcup_{i=1}^{m} E_i$.

The encoding of the STRIPS state defined in 4.4 using the object-atom binary structure is a bipartite graph and is shown in Figure 4.3. The objects (i.e., elements of O) are depicted as black boxes and the atoms (i.e., elements of S) are depicted as black dots. The atoms are labeled by their predicates and objects by the unary predicates from P they satisfy. Edges connect objects and atoms in which they occur. Numbered label indicates a position at which they occur. For instance, the object b_2 is connected to $B(b_2)$ via E_1, to $U(b_1,\ b_2)$ via E_2, and to $T(a,\ b_1,\ b_2)$ via E_3.

Note that this structure can be represented as a hypergraph, where predicates with arity at least 2 are represented as hyper edges with one-hot feature vectors.

4.4.4 Object-Pair Binary Structure

It is generally known that using pairs of objects increases expressiveness of given architecture, see 4.5. Therefore object-pair binary structure builds on 2-sets of objects. We create these 2-sets as 2-tuples whose components are sorted by a fixed total order $\leq$ on the set of objects. We also consider singletons, such as $\langle o, o \rangle$. State $S = \langle O,\ \langle R^S \mid R \in P \rangle \rangle$ is represented as a graph $G = \langle V, E \rangle$

$$V = \{\vec{u} = \langle o_1, o_2 \rangle \mid o_1,\ o_2 \in O,\ o_1 \leq o_2\}$$

Two nodes $\vec{u}$ and $\vec{v}$ are connected by an edge $e \in E$ if they share a single object.

$$E = \{\langle \vec{u}, \vec{v} \rangle \mid \vec{u}, \vec{v} \in V,\ \vec{u} \neq \vec{v},\ |\vec{u} \cap \vec{v}|\ \ = 1\}$$

Feature vector of a node $\langle o_1, o_2 \rangle$ contains three types of features and is multi-hot. Nullary predicate symbols are in all feature vectors. Unary predicates are separated for both object, they are denoted by subscript 1 for o_1 and 2 for o_2.

Predicate symbols $P \in R$ of arity at least two are represented by a one if $\{o_1, o_2\} \subseteq \vec{c}$ for some $\vec{c} \in R^S$. For every unary predicate $P \in R$ we create two copies $P_1,\ P_2$. For unary predicate $P \in R$ and an ordered pair $\vec{o} = \langle o_1, o_2 \rangle \in V$, we define

$$P_1(\vec{o}) \in P^S \text{ iff } P(o_1) \in P^S$$

$$P_2(\vec{o}) \in P^S \text{ iff } P(o_2) \in P^S$$

A node representing a pair of objects $\vec{o} = \langle o_1, o_2 \rangle$ has a feature of predicate $P \in R$ of arity at least two if both inner objects of $\vec{o}$ occur in an atom $\vec{c} \in P^S$. Also an edge e connecting two nodes $\vec{u} = \langle u_1, u_2 \rangle$ and $\vec{v} = \langle v_1, v_2 \rangle$ is labeled by predicate symbol $\hat{P}$, where $\hat{P}$ is constructed from predicates $P \in R$ with arity at least 2 , if the symmetric difference $\vec{u} \triangle \vec{v} \subseteq \vec{c}$ for some $\vec{c} \in P^S$.

The encoding of the STRIPS state described in 4.4 using the object-pair binary structure is shown in Figure 4.4. We ordered objects as follows: $a < b_1 < b_2$ and therefore created ordered pairs as shown in circles as nodes. Note that nullary atom N is shared across all ordered pairs. For ordered pair $\langle a, b_1 \rangle$ there are three valid atoms shown in feature vector, nullary atom N, unary atom $A(a)$ with subscript 1 since object a is in the first position in the ordered pair, unary atom $B(b_1)$ with subscript 2 for the second object, and ternary atom $T(a, b_1, b_2)$ since both objects occur in the atom. Note that there is an edge between ordered pairs $\langle b_1, b_1 \rangle$ and $\langle b_1, b_2 \rangle$, because it contains the same object b_1, labeled with feature vector of two valid atoms $\hat{U}$ and $\hat{T}$. The symmetric difference of these two ordered pairs is $\{b_1, b_2\}$ which is a subset of elements in atom $U(b_1, b_2)$ and in atom $T(a, b_1, b_2)$, therefore the feature vector for this edge contains a one in both its positions.

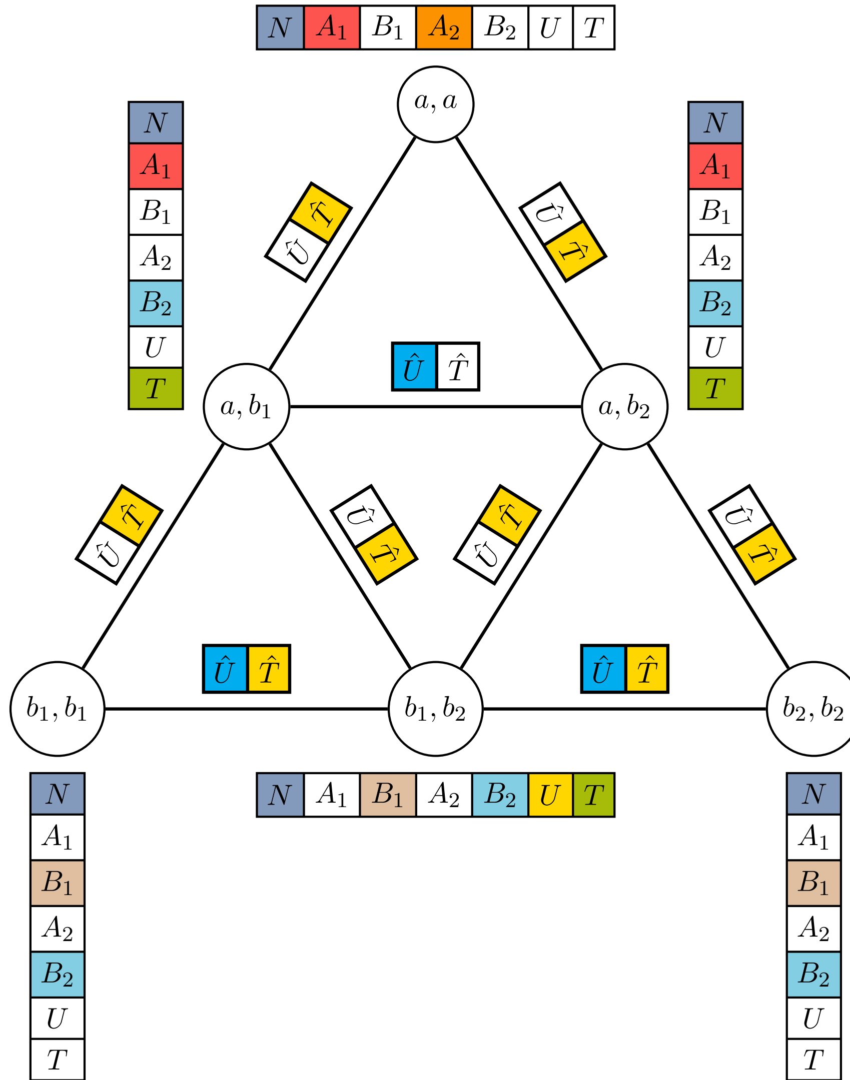


Figure 4.4: Encoding of the STRIPS state defined in 4.4 using the Object-Pair Binary structure

4.5 Expressiveness

In the previous section we introduced architectures representing states as a data structure suitable as input for GNN models. Each architecture has its expressive power. The expressive power of an architecture is defined by the number of distinguishable states. The message passing computation is theoretically similar to the Color Refinement algorithm. Color Refinement can be used as an isomorphism test for two graphs that is as strong as the 1-dimensional Weisfeiler-Lehman test (1-WL) [15]. 1-WL test cannot distinguish all the non-isomorphic graphs. It can distinguish those graphs that are not C_2-equivalent [16]. Therefore it is not surprising that message-passing GNN models have the same limitations [12]. The expressive power of GNN

model can be enhanced by mimicking higher order WL test [15]. 1-WL test can be computed over a graph build from all k-tuples of nodes. These variants can then distinguish graphs up to C_k-equivalence. On one hand this increases the expressive power of an encoding and could perform better. On the other hand building such graphs creates problems. The number of k-tuples grows rapidly and the number of edges as well. As a result variants building on these k-tuples were introduced [17]. However, any computations over these graphs will need cutting-edge models of processing units.

Chapter 5

Implementation details

In this section we would like to discuss representation of edges in a graph and its memory usages. We build graph structures using Mill.jl [18] library. We propose new representation of edges based on the Compressed Sparse Column (CSC) format [19]. This format is characterized by its efficiency in terms of memory usage and creation time. This means it consumes less memory, requires less time to generate and less number of memory allocations compared to other commonly used formats, such as adjacency matrix. We do not use vector of vectors but one flat vector, whose maximum length can be usually estimated.

5.1 Graph Connectivity Representation

There are common representations of edges however, each representation is suitable for different project. Using some of these is suboptimal for our purposes. We want to use as less memory as we can and create the structure as fast as possible.

We will use a graph $G = \langle V, E \rangle$ to demonstrate representation formats. Graph G consists of 5 vertices $V = (A, B, C, D, E)$ and 5 edges $E = (\{1,5\}, \{2,3\}, \{2,5\}, \{3,4\}, \{4,5\})$ between them. See Figure 5.1.

5.1.1 Adjacency Matrix

Edges in a graph can be represented by a matrix $A \in \{0,1\}^{n \times n}$ where each i-th column and i-th row represent a i-th vertex. The entry on i-th column and j-th row is 1 if i-th vertex is connected to j-th vertex, otherwise zero. This creates a symmetric matrix since edges are indirected and if i-th vertex is connected to j-th vertex, then j-th vertex is connected to i-th vertex as well.

$$M_{i,j} = 1 \text{ iff } (i,j) \in E$$

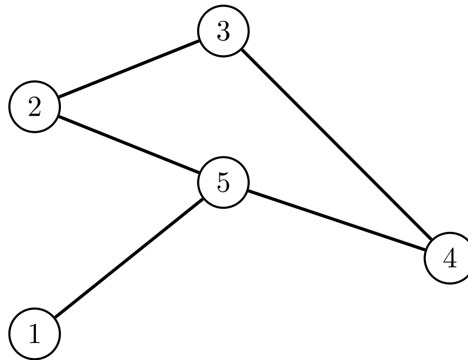


Figure 5.1: Graph $G = \langle V, E \rangle$

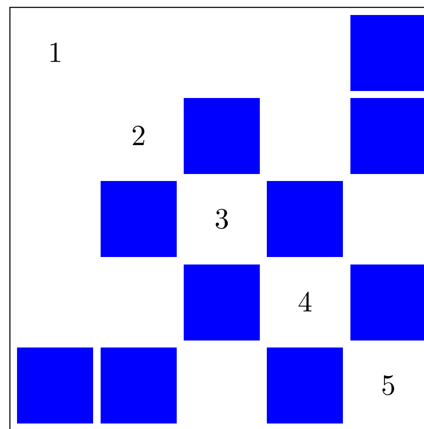


Figure 5.2: Adjacency matrix of the graph G defined in 5.1

5.1.2 Adjacency List

This simple representation takes all edges and forms an adjacency list. This list consists of pairs of vertices, where each pair represents an edge connecting two vertices. This representation can be used to represent hyper edges as well, where the list would contain tuples of vertices connected by a hyper edge.

The following list is an adjacency list representing graph G defined in 5.1.

$$[(1,5),\ (2,5),\ (2,3),\ (3,4),\ (4,5)]$$

5.1.3 Scattered Bags

A structure called Scattered bags was already implemented in Mill.jl [18]. We say that a bag is a set of indices of edges in which a vertex occurs. Scattered bags is a vector of vectors. Edges are iteratively pushed into the inner vectors. Each inner vector at the end contains indices of edges a given vertex occurs in. This representation can represent hyper edges as well.

Consider the graph G defined in 5.1. We enumerate the edges as follows:

$$\begin{array}{lcl} 1 & - & \{1,5\} \\ 2 & - & \{2,3\} \\ 3 & - & \{2,5\} \\ 4 & - & \{3,4\} \\ 5 & - & \{4,5\} \end{array}$$

The scattered bags structure for edge representation looks as follows:

$$((1),(2,3),(2,4),(4,5),(1,3,5))$$

We can see that a vertex 5 occurs in three edges, 1, 2, and 3. On the other hand vertex 1 occurs only in one edge, edge 1.

5.1.4 Compressed Bags

We propose a Compressed Bags format, which stores undirected edges in an optimized way. It is friendly in memory usage and time complexity of creation. This format optimizes all encodings and supports transformation of states into graphs with unknown number of edges beforehand but with the number of maximum possible edges. It also supports hyper edges connecting arbitrary number of vertices.

This format has been derived from Compressed Sparse Column format. It represents edges connecting an arbitrary number of vertices using two vectors. However, one compressed bags structure can represent edges connecting a given number of vertices. Therefore for two hyper edges connecting different number of vertices two compressed bags structures are needed.

This format consists of two vectors. One can be sought at like connected many vectors of possibly different size side by side. Each of these vectors represents one vertex. These vectors contains indices of edges in whose the given vertex occurs. The other one keeps in memory the ranges of vertices, where their inidices of edges start and where they end. The concept is essentially similar to the scattered bags however, it is more memory efficient.

Construction of Compressed Bags

Let n be the number of vertices in graph G, k a number of edges in G each connecting l vertices. For construction we need to create a $l \times k$ matrix M of integers and a $\mathbf{c} \in \mathbb{N}^n$ and set the values of $\mathbf{c}$ to zero. If the number of edges is unknown before hand we compute the maximum number of edges for given architecture and allocate the needed amount of memory for that. We assume that an edge $e = (u_1, \ldots, u_l)$ is a tuple of vertices it connects and that $E = \{e_1, \ldots, e_n\}$ is a set of edges in G. We enumerate all the vertices and assign them their indices. We call $\mathbf{c}$ *counts* as it counts number of observations of vertices for all edges and M *indices* as it it as matrix of indices of vertices.

For each edge e_i we insert its values (indices of vertices) into the i-th column of the matrix M, also as we iterate over the vertices in the edge we update the number of occurrences of each vertex in $\mathbf{c}$.

Consider the graph G defined in 5.1, the matrix M of indices of vertices and a vector $\mathbf{c}$ of counts will look like this

$$M = \begin{bmatrix} 1 & 2 & 2 & 3 & 4 \\ 5 & 3 & 5 & 4 & 5 \end{bmatrix}, \mathbf{c} = \begin{pmatrix} 1 & 2 & 2 & 2 & 3 \end{pmatrix}$$

Then we compute the ranges of each bag knowing the number of occurrences of each vertex. Knowing the starting and ending index of each bag lets us iteratively insert the indices of edges into their respective bags. The index of an edge is the number of a column in M in which it is stored.

$$\mathbf{ii} = \left(\begin{array}{c|cc|cc|cc|ccc} 1 & 2 & 3 & 2 & 4 & 4 & 5 & 1 & 3 & 5 \end{array} \right)$$

$$\mathbf{bags} = \begin{pmatrix} 1:1 & 2:3 & 4:5 & 6:7 & 8:10 \end{pmatrix}$$

As we can see the vector of indices **ii** contains indices of edges for each vertex. We depicted the seperation of vertices as vertical lines, i.e., the first vertex occurs in edge number 1, the penultimate occurs in edges 4 and 5 and the last vertex occurs in edges 1, 3 and 5. The **bags** vector tells us where the seperating lines are for each vertex, i.e., vertex number 1 has indices of edges starting in position 1 and ending at position 1 as well, meaning that it occurs in only 1 edge. Vertex number 3 has a starting position of indices at 4 and ending at position 4, it occurs in two edges and their indices are at ii_4 and ii_5. We start indexing into vector from 1, the first element in vector is ii_1.

Chapter 6

Discussion

In this section we will discuss shortcomings of and strengths of using multi graph with one-hot feature vectors over multi-hot feature vectors. Afterwards we will present two representations of nullary and unary predicates in graphs. And show see if there are advantages to one representation over the other.

6.1 Edge Representation

Each representation has its advantages and disadvantages. Some are more efficient in construction, some are more suitable for smaller graphs and some for larger graphs.

6.1.1 Adjacency matrix

On one hand this representation is straightforward and intuitive. On the other hand the adjacency matrix is often very sparse and keeping $n \times n$ matrix in memory is inefficient, where n is the number of nodes. Another point is that this adjacency matrix can represent only edges connecting two vertices, it would need more structures to support hyper edges. And therefore it would use more sparse matrices and require more memory capacity.

6.1.2 Adjacency list

This representation is memory friendly, takes only the needed amount of space and can represent hyper edges connecting arbitrary number of nodes. However, the usage for GNN models is rather unfortunate as message passes aggregate information about vertices in the one-hop neighbourhood and an adjacency list is not sorted and no vertex knows its neighbors.

6.1.3 Scattered Bags

This representation was already implemented in the Mill.jl library and was initially used for the encodings. However after further inspection we found out that a lot of time and memory allocations take place especially in the section of forming edges and saving. The main problem was when the number of

edges exceeded the allocated space. Furthermore scattered bags is essentially a vector of vectors and if one bag is full and we want to push another index inside of it we need to allocate new memory, copy the contents and free the previous chunk of memory. For big graph this could be done many times and therefore slow down the whole process.

6.1.4 Compressed Bags

Compressed Bags representation is efficient in its construction as well as in its memory usage. There is no problem with having a large chunk of memory preallocated therefore this format is optimal for our uses. Furthermore it is suitable for storing hyper edges and we can experiment with implementation of hyper graphs.

6.2 Multi graph or edge features

Consider an architecture that encodes objects as vertices and predicates as edges, i.e., object binary or object-pair binary structures. These structures can be represented either by multi graphs where each edge with one-hot feature vector representing only one predicate or by graphs where edges have multi-hot feature vectors possibly representing multiple predicates.

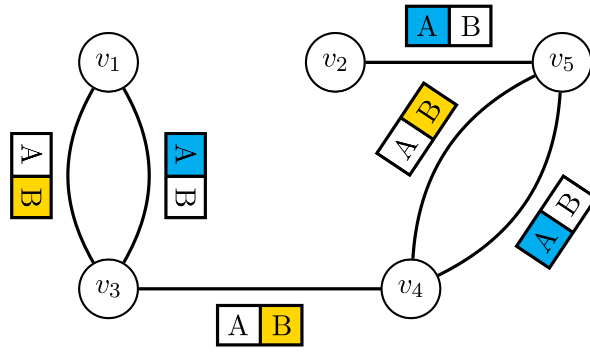
Having a multi graph with each edge representing only one predicate may possibly learn more and generalize better, however this is only a thought, it is not based on experiments. It is supported by having multiple message passes between nodes connected by multiple edges. Each message pass gathers data only from edges representing one type of predicate therefore the information is concentrated. However, it increases the number of message passes and therefore takes more time to process one input through the GNN model. This can possibly result in learning from less problems and even possibly solve less problems in given times. But it might be compensated by concentrated message passes.

Creating a graph with less edges comes in with the advantage of less memory usage. Since there are less feature vectors since there are less edges. This is very great for large graphs. Furthermore there is less message passes and the messages are more concentrated combining information using all the predicates.

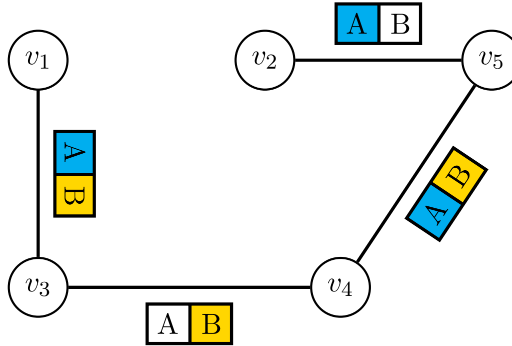
6.3 Nullary and Unary Predicates

Consider the same architecture with objects as nodes and predicates as edges. There are essentially two ways to represent nullary and unary predicates.

One way is to represent them as multi-hot feature vectors. Nullary predicates are shared among all nodes. And unary predicates are relevant only in the nodes whose objects occur in atoms. This representation is straightforward and initializes feature vectors based on the state of the world. It



(a) Multi Graph with one-hot feature vectors



(b) Graph with multi-hot feature vectors

Figure 6.1: Graphs encoding two types of edge feature vectors

is initially set and afterwards iteratively updated through message passes. There is one question that might arise. Are the nullary and unary predicates relevant after many message passes, do they not lose importance? After many message passes the feature vectors will look completely different. And the initial nullary or unary predicates might hold little to none importance.

The other representation transforms nullary and unary predicates into edges. Nullary predicates might be represented as hyper edges over all the nodes, or as loops. Unary predicates are then represented as loops as well. Node feature vectors need to be initialized from single element vectors of ones. On one hand there is a big advantage of representing nullary and unary predicates as edges. Each message pass uses all edges and therefore all predicates. This can create a big advantage with many message passes as in every message pass nullary and unary predicates have the same amount of importance as predicates with arity at least 2 since these have been used in every message pass. On the other hand each object is represented essentially by a single element vector, a scalar. This can decrease variety of feature vectors and multiple objects can be represented by the same values.

There might arise the same problem as in the previous section, where more edges mean more message passes and therefore mean spending more time in updating feature vectors.

6.4 Architectures

Each introduced architecture is different. However, there are two that share a common problem, object-pair binary and atom binary. Graphs transformed from PDDL states using object-pair binary encoding contain $\frac{n^2}{2} + \frac{n}{2}$ nodes and at least that amount of edges, where n is the number of objects in a given problem. Atom binary encoding does not have that amount of nodes but it has a quadratic amount of edges with respect to the atoms. This creates very large graphs. One advantage of it is the expressiveness however, the computational cost and memory needs are massive. These two encodings might have a problem on training on large domains such as sokoban or rovers.

Chapter 7
Experimental section

In this section we would like show results of our experimental part. We implemented introduced architectures and then tested their performance on problems from International Planning Competition (IPC) 2023.

7.1 Preliminary experiment

Initially we conducted a preliminary experiment on simpler problems. With this experiment we wanted to find out whether nullary and unary predicates should be implemented as features of nodes or whether it would be better to implement them as another layer of edges. We implemented 2 architectures where nodes represent objects and hyperedges represent atoms. Feature object-atom represents nullary and unary predicates as features of nodes and predicate object-atom represents them as hyperedges.

This experiment was conducted over a set hyper parameters. The searched for hyperparameters were a number of message passing layers, 1, 2 and 3, a dimension of dense layers, 4, 8 and 16, and the usage of a residual layer or a linear combination of its input and output, total number of combinations is 18. This was conducted three times with different seed. In the table 7.1 we can see the number of finished training and testing experiments. An experiment with a set of hyperparameters was finished if it did not exceed the allocated amount of memory or the given time. We can see that the amount of finished experiments did not much change over the different initializations with seeds.

In the next table 7.2 we can see the results of testing trained models on given problems. We can see that both of these encodings solved nearly all of the testing problems but there are some differences. Feature object-atom solved overall more problems and looked more stable on all the problems. However, this might differ on other encodings, a more complex experiment on many more encodings would need to be conducted to create a clear conclusion on which implementation is better. Unless stated otherwise we mean feature object-atom, because it worked better in this experiment

problem	Predicate Object-Atom	Feature Object-Atom
blocks	11/12/12	17/18/17
elevators	17/17/16	16/17/13
ferry	16/16/18	17/18/18
npuzzle	18/18/18	18/18/18
spanner	18/18/18	18/18/18

Table 7.1: Finished number of combinations over the different seeds

problem	Predicate Object-Atom	Feature Object-Atom
blocks	0.98	0.99
elevators	0.98	1.00
ferry	0.88	0.91
npuzzle	0.89	0.89
spanner	0.99	0.99

Table 7.2: The proportion of solved problems

7.2 Optimizations

Firstly we would like to show the optimizations of formats for storing edges. We initially stored edges as a vector of vectors. Then we implemented Compressed bag format using a vector, as decsribed in the previous section. And afterwards we implemented the format using a matrix.

We will show this on the transformation of a state into a graph using object binary and atom binary encodings and all the problem domains. The table 7.3 shows the initial implementation using scattered bags format for edge representations (SBags). We computed the mean value of transforming a state into its graph representation over all the training problems for each domain in microseconds. We can see that especially sokoban and rovers domain take a substantial amount of time. On the other hand domains such as ferry or blocksworld take little to none time. This is caused by the number of objects and predicates in the problems. Sokoban has generally tens to hundreds of objects and atoms in one problem however, blocksworld problems have generally up to 12 objects and similar amount of atoms.

We then implemented the compressed bags format for edge representation using vector for construction. The results can be seen in Table 7.3 (CBags). This substantially decreased the amount of time it took to transform a state into its graph representation. In many domains it decreased the time by half, for example rovers or childsnack domain and overall it decresead the time on all domains. However, some domains such as sokoban and rovers especially using the atom binary encoding take a lot of time to compute.

We further optimized the compressed bags format using a matrix for more cache hits. As can be seen in the Table 7.3 (CBagsMat) it helped and sped up the transformation nearly on all domains. The biggest difference can be seen in the domains of big problems, such as rovers or sokoban. However, even on

problem	ObjectBinary SBags	ObjectBinary CBags	ObjectBinary CBMat	AtomBinary SBags	AtomBinary CBags	AtomBinary CBMat
ferry	18.0	12.9	10.6	94.4	49.9	53.8
rovers	207.8	108.4	108.7	8110.4	4906.9	4812.0
blocksworld	22.9	14.7	14.5	179.2	89.4	89.8
floortile	96.6	55.4	52.7	1344.5	888.5	806.6
satellite	72.8	40.3	38.6	770.5	538.5	496.4
spanner	38.2	25.5	24.4	190.8	98.2	93.3
childsnack	33.4	21.0	154.2	68.1	93.3	74.9
miconic	81.3	51.6	50.4	2725.0	1725.5	1679.7
sokoban	223.3	119.8	116.3	12285.1	10670.8	10611.4
transport	54.7	28.9	28.7	445.7	213.4	213.0

Table 7.3: Average time in μs of transforming a state of given problem into a graph using scattered bags format (SBags), compressed bags format using a vector (CBags) and compressed bags format using a matrix (CBMat)

the small ones, such as spanner, we managed to push the amount down as well. On the other small problems such as ferry or childsnack it took a little more time using the atom binary encoding. This might occur because of the allocation of big matrix for small number of elements and then its access.

7.3 Preparation

7.3.1 Architecture implementations

With respect to our input format into the GNN models we implemented object atom structure as a hypergraph. On one hand the bipartite graph creates interesting representation. However, the unification of the two graphs, one for the initial state and the other for the goal state, is not intuitive. We would need to create new nodes for atoms in the goal state and restructure the whole representation, which would essentially only take more time. Therefore we decided to go with the implementation of hypergraph, where stacking hypergraphs is easy since the nodes are the same and edges can be easily added to the structure. Other encodings of states were implemented as multigraphs, edges having one-hot feature vectors. As was found in the preliminary experiment we decided to implement all the introduced encodings using implementation of nullary and unary predicates as features of nodes.

GNN models for each domain and architecture were created using Mill.jl [18] library. This library is able to create a GNN model based on one encoding and few variables, such as number of layers, dimension of dense layer and either usage of residual layer or linear combination of output of residual layer and its input. Afterwards this model can accept any state in the same representation as the initial state and be trained.

7.3.2 Used Cluster

We trained and evaluated GNN models on a cluster AMD EPYC 7543 which is a part of AMD's EPYC 7003 series. This particular model has 32 cores and 64 threads. It has a base clock speed of 2.8 GHz and 256MB of L3 Cache. We used only 8 cores with 64 GB of RAM.

7.3.3 Search for hyper parmeters

As said in the previous section each GNN model can be initialized with different values. We used a grid search over a number of hyperparameters, such as a number of message passing layers, a dimension if hidden layers and an initial seed. Our values for the number of message passing layers was chosen to be small, for we think that the model does not need many layers to learn the features of a state, exactly speaking we chose between 1, 2 and 3 layers. For the dimension of hidden layers we chose from 4, 16, 32 and 64. Each of these combinations was experimented on three times with different initialization using a seed, 1, 2 or 3.

7.3.4 Dataset

Our dataset consists of multiple classical PDDL domain problems from IPC 2023, to name a few sokoban, blocksworld, ferry. Each domain consists of roughly 50 problems with plans for training and about 90 problems for testing.

7.3.5 Planner

Since we want to learn a heuristic function using GNN models, we used the A^* and Greedy planners for all the testing.

7.4 Results

As predicted in the previous chapter, object pair encoding does have a problem even with small problems. We tried to train GNN model using this architecture however, it did run out of memory and or out of the given amount of time on many tries. Even on the cluster this was impossible to compute with our limited time and resources. Therefore we did not compute the statistics for this encoding.

We conducted experiments of training and testing GNN models using atom binary, object atom and object binary encodings over a set of hyperparameters. We trained each GNN model for 100 epochs and then tested its performance on the testing dataset.

Each experiment of training and testing one architecture on one domain was given 3 days of time. If the experiment ran out of memory or given time, the experiment was not finished and we continued on another. Each combination of hyperparameters was given 3 hours of time, considering that the total number of combinations was 24. We optimized each GNN model

problem	AtomBinary	ObjectAtom	ObjectBinary
blocksworld	18/18/18	17/18/17	24/24/23
ferry	16/17/16	18/18/17	24/24/23
floortile	5/4/2	18/17/18	24/24/23
miconic	—	17/18/17	24/24/22
rovers	—	7/8/8	12/12/11
satellite	0/3/1	17/16/17	24/23/22
sokoban	—	13/12/13	22/22/22
spanner	18/18/18	17/18/18	24/24/23
transport	—	17/16/18	23/23/24

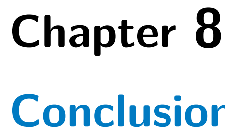
Table 7.4: Number of finished training and testing of GNN models using different combinations of hyperparameters

problem	AtomBinary	ObjectAtom	ObjectBinary
blocksworld	0.22	0.33	0.33
ferry	0.53	0.56	0.56
floortile	0.28	0.43	0.43
miconic	—	0.61	0.64
rovers	—	0.43	0.46
satellite	0.19	0.25	0.25
sokoban	—	0.28	0.30
spanner	0.41	0.52	0.51
transport	—	0.09	0.10

Table 7.5: The proportion of solved problems in given time

using the L^* loss function [20]. In the table 7.4 we can see the number of finished training and testing of GNN models using atom binary, object atom and object binary encodings over different combinations of hyper parameters. The experiments were conducted three times for three different initializations. We can see that the object binary architecture finished nearly all of the experiments besides the rovers domain. Object atom architecture completed two thirds of the experiments and also had problems with the rovers domain. Atom binary architecture however, did not complete 4 out of the 9 domains, depicted by —, and on satellite and floortile did poorly.

We also computed the proportion of solved problems for each domain and architecture. Object binary encoding, on the other hand, solved about half of the problems on few domains and fewer on others. The object atom encoding solved the same amount as object binary or slightly fewer. In the Spanner domain the object atom encoding solved few more problems. With respect to the amount of finished experiments it is clear that the atom binary architecture also did not solve any problems in the miconic, rovers, sokoban and transport domains and solved less problems in all other domains. The only relevant and similar to the other encodings is the ferry domain.



Chapter 8
Conclusion

Graph neural networks provide an effective method for understanding the complexities of graphs. Generalizations of graphs are useful since they provide formats for encoding complex structured data. GNN models also provide great scalability since the only restriction for the input is the structure of the graph, not its number of nodes or vertices. However, understanding details of general planning problems and generalizing over complex and large hyper graphs is complicated. It is not clear how to encode states of planning problems. The main goal of this thesis was to implement the introduced architectures and experimentally compare their effectiveness.

We showed that one section of graph neural networks learn using the message passing algorithm. Afterwards we introduced multiple transformations of planning problems into generalizations of graphs, such as hypergraphs or multigraphs. We implemented these encodings using the Julia programming language as an extension of the NeuroPlanner.jl library, leveraging the structures provided by the Mill.jl library. Furthermore, we optimized the structures in the Mill.jl library for enhanced performance and efficiency in our implementations. Each implementation of architectures was optimized to the point of all being on the same level for qualitative experimental results.

We conducted an experiment to see whether the implementation of nullary and unary predicates as feature vectors of nodes is better than their implementation as edges and loops. With this knowledge we implemented specific sections of the architectures so that they are optimal for our purposes. Afterwards an experiment of comparing the architectures was conducted using problems from International Planning Competition.

GNN model of each encoding was trained on specific domains and then tested on provided problems. Object binary encoding solved the most problems, followed closely by object-atom binary. These seemingly simple architectures generated the best results. Atom binary architecture, potentially strong encoding in terms of expressive power, produced regrettable results. The computations were demanding and the results were not as satisfactory as we thought they would be. It solved substantially less problems than the previous two architectures. The Object-pair binary architecture proved to be more computationally demanding than our current capabilities could handle. With the increase in expressive power of an encoding, its computa-

tional demands also grow. Thus, we conclude that the expressive power of an encoding is less crucial for satisficing planning compared to factors like straightforward implementation and computational ease. Therefore encodings with low computational cost such as object binary are more suitable for satisficing planning rather than the more expressive with high computational cost such as atom binary.

Bibliography

[1] Tomáš Pevný. Neuroplanner.jl. `https://github.com/pevnak/NeuroPlanner.jl`, 2023. GitHub repository.

[2] Tomas Pevny and Simon Mandlik. Mill.jl framework: a flexible library for (hierarchical) multi-instance learning.

[3] Nils J. Nilsson. Shakey the robot. 1984.

[4] E Pednault. Exploring the middle ground between strips and the situation calculus. In *Proc. of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332.

[5] 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. 08 1998.

[6] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, December 2003.

[7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[8] John S Ebersole and Timothy A Pedley. *Current practice of clinical electroencephalography*. Lippincott Williams & Wilkins, 2003.

[9] Gordon M Shepherd. *The synaptic organization of the brain*. Oxford university press, 2003.

[10] K. Gurney. *An Introduction to Neural Networks (1st ed.)*. CRC Press., 1997.

[11] Pitts W. McCulloch, W.S. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5*, page 115–133, 1943.

[12] Rostislav Horcik and Gustav Šír. Expressiveness of graph neural networks in planning domains. In *34th International Conference on Automated Planning and Scheduling*, 2024.

[13] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning generalized policies without supervision using gnns, 2022.

[14] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[15] Martin Grohe. The logic of graph neural networks, 2022.

[16] Jin Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, December 1992.

[17] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks, 2021.

[18] Simon Mandlík and Tomáš Pevný. Mapping the internet: Modelling entity interactions in complex heterogeneous networks. *CoRR*, abs/2104.09650, 2021.

[19] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.

[20] Leah Chrestien, Tomás Pevný, Stefan Edelkamp, and Antonín Komenda. Optimize planning heuristics to rank, not to estimate cost-to-goal, 2023.

Appendix A

Attached files

The attached file is a compressed file of the whole NeuroPlanner.jl [1] repository as of 5. 23. 2024. The simple unpacking of the file and instantiating of the NeuroPlanner module should provide the user all the implemented features such as encodings, compressed bags and more. However the library will be updated and further optimized over time so we suggest the reader to clone the github repository [1].