UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie
master in de informatica

Masterthesis

From graded mu-calculus to recurrent graph neural networks

Joppe Wouters
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Jan VAN DEN BUSSCHE

2023
2024

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

From graded mu-calculus to recurrent graph neural networks

Joppe Wouters
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Jan VAN DEN BUSSCHE

Acknowledgements

I would like to begin this thesis by thanking a few people, without whom it would never have been possible. First and foremost, I would like to express my sincere gratitude to my promotor, prof. dr. Jan Van den Bussche, for his invaluable guidance and support during this thesis. His willingness to provide continuous feedback and engage in brainstorming sessions was crucial to understanding this fascinating topic better and solving the challenges I encountered. I also want to thank my friends from university who made this academic journey really fun. Finally, I am deeply grateful to my parents for their unwavering belief in me and for allowing me to pursue these studies.

Samenvatting

Kunstmatige intelligentie (AI) en neurale netwerken krijgen de meeste aandacht in het huidige technologische landschap. Mensen denken dat we ons in een tijdperk bevinden waarin AI 'alles' kan. De uitdrukkingskracht van zulke neurale netwerken kan beter verklaard worden door logica, die gebruikt kan worden als tegenvoorbeeld van wat mensen denken dat neurale netwerken kunnen. Formele logica speelt een cruciale rol in het beschrijven van de uitdrukkingskracht van programmaklassen. Deze aanpak werd gepionierd door Edgar F. Codd in de jaren 70 toen hij het relationele model voor datamanagement introduceerde, gebruikmakend van relationele algebra en relationele calculus voor het formaliseren.

Onderzoekers hebben recentelijk graaf neurale netwerk (GNN) architecturen geformaliseerd door gebruik te maken van logica. Het vermogen van GNNs is om knopen in grafen te classificeren, waarbij de classifier, een booleaanse classifier, uitgedrukt kan worden als een logische formule. Het vooraanstaande onderzoek is het werk van Barceló et al. [BKM$^+$20], waarop deze thesis is geïnspireerd. Barceló et al. onderzochten en bewezen de equivalentie tussen AC-GNNs, een GNN architectuur gebaseerd op message passing, en formules in graded modal logic (GML). GNNs beginnen erg populair te worden voor specifieke toepassingen, maar er is nog veel onbekend. Daarom breiden we het opmerkelijke onderzoek van Barceló et al. uit en verdiepen we ons in de logische uitdrukkingskracht van recurrente graaf neurale netwerken (RecGNNs). Deze RecGNNs zijn te vergelijken met GNNs waaraan recursie is toegevoegd; de equivalente logica omvat ook recursie. Deze logica staat bekend als fixpoint GML of graded μ-calculus.

Achtergrond informatie

Om een goede basis te vormen voor onze voorgestelde aanpak, verdiepen we ons eerst in de achtergrond door een aantal relevante studiegebieden te bespreken.

Deze thesis maakt gebruik van graded modal logic, een fragment van predikatenlogica (FO) dat gebruikt kan worden voor lokale graaf evaluatie. In tegenstelling tot predikatenlogica gebruikt GML geen variabelen en richt het zich op individuele knopen binnen een graafstructuur $G(V, E)$. Formules worden geëvalueerd op een specifieke knoop $v \in V$. Predicaten, kunnen geïnterpreteerd worden als de kleuren van knopen in een gekleurde graaf. Voorheen aangeduid als bijvoorbeeld $\mathrm{Col}(x)$ in FO, maar deze worden nu gebruikt als Col. Modale connectieven worden ook geïntroduceerd, de diamond operator $\Diamond^N$ en de vierkant operator $\Box^N$, met N een natuurlijk getal. De FO-expressie $\exists^{\geq N} y(E(x, y) \wedge \varphi(y))$ wordt aangeduid met $\Diamond^N \varphi$ in GML. De vierkant operator wordt gedefinieerd als de negatie van de diamond operator. Graded modal logic formules worden inductief gedefinieerd door het combineren van predikaten (i.e., Col), negatie, conjunctie en de diamant operator. Operatoren zoals de disjunctie en de vierkant operator kunnen met behulp van de wetten van De Morgan worden afgeleid uit de beschikbare operatoren.

Op basis van GML kunnen we deze logica uitbreiden met de least fixpoint operator μ en greatest fixpoint operator ν. Deze uitbreiding creëert een krachtigere logica die bekend staat als graded μ-calculus. We kunnen de greatest fixpoint simuleren door de negatie te nemen van de least fixpoint.

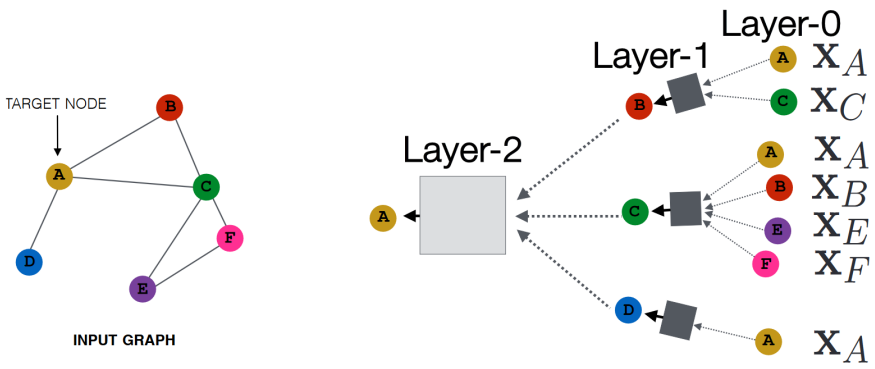

Figure 1: Message passing vergroot het receptieve gebied van knoop A [Les23].

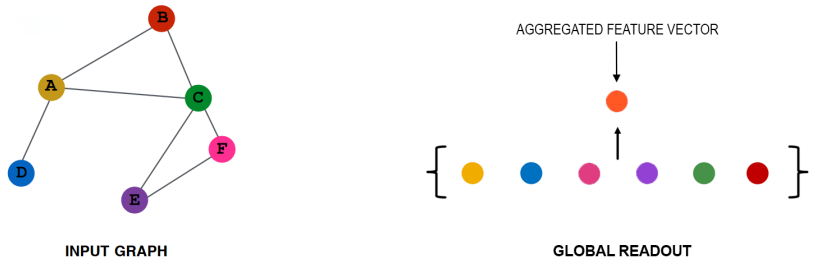

Figure 2: Global readout van de feature vectors van elke knoop in de inputgraaf.

Graaf neurale netwerken zijn ontwikkeld om de beperkingen van traditionele neurale netwerken op te lossen bij het verwerken van gegevens die gestructureerd zijn zoals een graaf. Traditionele neurale netwerken presteren goed met gegevens in rasterformaten, zoals afbeeldingen en sequenties, maar ondervinden moeilijkheden met relationele gegevens zoals kennisdiagrammen, sociale netwerken en moleculaire structuren. In dergelijke gevallen bestaan de gegevens uit individuele entiteiten (knopen) en hun verbindingen (bogen) met andere entiteiten. Traditionele neurale netwerken gaan uit van data-onafhankelijkheid, iets wat niet toepasbaar is in graafstructuren waar de relaties tussen entiteiten ook behoren tot de datastructuur.

GNNs gebruiken lagen die vergelijkbaar zijn met die van traditionele neurale netwerken, maar beschikken over unieke mechanismen om graafstucturen te verwerken. Deze omvatten lokale en globale pooling-lagen die respectievelijk informatie van buren van een knoop en de hele graaf samenvoegen. Deze aggregatieprocessen, message passing en global readout zijn cruciaal voor het leren van patronen in graafstructuren. Tijdens message passing verzamelt elk knoop informatie van zijn buren, update zijn toestand en verspreidt deze informatie verder door de netwerklagen, zie Figuur 1. Een global readout verzamelt informatie van elk knoop in de graaf en voegt deze informatie samen, zie Figuur 2. Het type GNN dat beide mechanismen gebruikt staat bekend als de aggregate-combine-readout GNN (ACR-GNN).

Recurrente graaf neurale netwerken zijn een gespecialiseerde klasse van GNNs die gebruik maken van recurrente mechanismen. Een RecGNN is een meerlaagse ACR-GNN die wordt geïtereerd en functioneert als een netwerk met één enkele invoer: de inputgraaf. In elke iteratie dient de output als input voor de volgende iteratie. Dit iteratieve proces gaat door totdat de classificaties van de knopen stabiliseren, wat betekent dat de waarden in de laatste laag van een iteratie overeenkomen met die van de vorige iteratie.

Aanpak

De eerste stap van onze aanpak begint met het bestuderen van het bewijs van Barceló et al. Dat bewijs fungeert als onze basis, waardoor we elke tussenstap moeten begrijpen. Bij het analyseren van hun constructie werd een inefficiëntie opgemerkt bij het berekenen van de features. Voordat de basis werd uitgebreid, waren enkele optimalisaties nodig om deze inefficiënties te verminderen.

De optimalisatie ligt in de feature vectors, die door het neurale netwerk heen worden vergroot. In plaats van elk feature vanaf het begin op te slaan, voegt elke laag zijn corresponderende features toe aan de vectoren. Deze constructie van flexibele feature vectors optimaliseert ook het aantal lagen dat nodig is in de ACR-GNN, wat het aantal rekenkundige bewerkingen aanzienlijk vermindert. Vanwege deze veranderingen moeten we de constructie van Barceló et al., die bestaat uit de essentiële GML-operatoren, opnieuw definiëren.

De volgende stap is het toevoegen van recursie aan zowel de logische als de GNN aspecten. Aangezien de output van de ene iteratie wordt gebruikt als input in de volgende iteratie, is een extra laag vereist die de uitgebreide feature vector terug naar zijn oorspronkelijke grootte transformeert. In graded modal logic kan elke operator in één iteratie worden berekend, maar voor fixpoint operatoren kunnen meerdere iteraties noodzakelijk zijn. Het aantal iteraties dat nodig is voor convergentie is inherent onbekend door de afhankelijkheid van deze iteraties van de inputgraaf. Om dit probleem op te lossen worden extra readiness features geïntroduceerd. Elk feature krijgt een extra feature dat de readiness (gereedheid) weergeeft. Vanwege onze definitie propageert deze gereedheid zich door de subformules van de logische formule die we willen simuleren.

Hoe wordt de least fixpoint operator μ_X geïmplementeerd? Deze operator moet voor elk knoop controleren of de oude waarde gelijk is aan de nieuwe. Er kan een gelijkheidvergelijking worden gemaakt tussen de feature van de corresponderende fixpoint relatie X en een andere operator, het rechtstreekse kind van de μ-operator binnen de abstracte syntaxboom (AST). Het is belangrijk op te merken dat, als binaire boom, elke μ-operator in de AST slechts één kind heeft. Aan de hand van deze vergelijking kunnen we bepalen of elke knoop in de inputgraaf is aangepast of zijn fixpoint heeft bereikt. Om te controleren of alle knopen in de inputgraaf hun fixpoint hebben bereikt, is een global readout nodig. Deze stap vereist echter kennis van het totale aantal knopen in de graaf. Om dit te kunnen bepalen, introduceren we een extra kleur voor de knopen, aangeduid met n, die *waar* is voor elke knoop in de inputgraaf. Hierdoor kan de global readout nagaan of een globaal fixpoint effectief is bereikt.

Geneste fixpoint operatoren introduceren een extra laag van complexiteit. Wanneer een geneste structuur wordt gebruikt, moeten de relaties die geassocieerd zijn met deze binnenste operatoren gereset worden bij het bereiken van hun respectievelijke fixpoints. Deze reset wordt uitgevoerd door de omsluitende (buitenste) fixpoint operator. De buitenste operator kan dat door de gereedheid van zijn directe kind operator te verifiëren. Gereedheid betekent de afronding van berekeningen binnen de deelboom van het kind. Bijgevolg worden alle operatoren binnen die deelboom, inclusief eventuele verdere geneste fixpoint operatoren, beschouwd als klaar om te resetten.

Implementatie

Implementatiegewijs kan onze aanpak voor een fixpoint operator in twee delen worden verdeeld. Gedurende het eerste deel controleren we de lokale gereedheid voor elke knoop in de inputgraaf door de gelijkheid tussen de fixpoint relatie X en het directe kind van de μ-operator te controleren. Simultaan resetten we ook de fixpointrelaties van geneste fixpointoperatoren indien deze aanwezig en gereed zijn. De gelijkheid wordt bepaald door een driedelig proces dat wordt vereist door de volgende logische expressie:

$$(X \wedge kind) \vee (\neg X \wedge \neg kind).$$

Het tweede deel voert een global readout uit om de globale gereedheid te controleren. Deze computationele fase omvat vijf subprocessen om de convergentie te controleren en de fixpoint relatie te updaten. De extra kleur voor de knopen n wordt gebruikt om het aantal knopen te kunnen berekenen die aanwezig zijn in de inputgraaf om convergentie voor elke knoop te controleren. De volgende logische expressie wordt hiervoor gebruikt:

$$\neg\left[\sigma(n - som)\right].$$

Het updaten van de fixpoint relatie is enkel toegestaan als de gehele deelboom klaar is omdat geneste fixpoint relaties nog niet gereed kunnen zijn. Zowel deze controle als het updaten zelf gebeurt in de laatste twee stappen van het tweede deel.

De bovenstaande constructie van een algoritme voor het simuleren van logische operatoren uit de graded μ-calculus met RecGNNs laat de mogelijkheden zien voor het simularen van positieve graded μ-calculus formules aan de hand van deze netwerken. Daarnaast hebben we een bijbehorende compiler geïmplementeerd die een graded μ-calculus uitdrukking als input neemt. Zulke graded μ-calculus uitdrukkingen hebben een beperkt aantal operatoren, dus het omzetten van infix notatie naar prefix notatie is relatief eenvoudig. Door deze notatie te gebruiken kan de bijbehorende abstracte syntaxboom geconstrueerd worden. De compiler kan de RecGNN-architectuur met de juiste laagvolgorde en parameters construeren door deze boom te doorlopen met behulp van een post-order traversal. Elke knoop krijgt een ID toegewezen gedurende deze traversal, en de basiskleuren van de knopen worden geïdentificeerd. Deze basiskleuren zijn nodig om de initiële feature vector te construeren.

Door gebruik te maken van de binaire boomstructuur zijn de benodigde parameters voor elke laag, zoals kindknopen of geneste μ-operatoren, gemakkelijk te identificeren. Dit is mogelijk door de boom dusdanig te construeren dat elke knoop over efficiënte toegangsmechanismen beschikt. Het kan rechtstreeks zijn directe kinderen bereiken en eenvoudig de deelboom verkennen met zichzelf als de wortel.

Conclusie

Deze thesis onderzoekt de uitbreiding van het werk van Barceló et al. door een recurrent graaf neuraal netwerk te construeren dat in staat is om graded μ-calculus uitdrukkingen te simuleren. Het onderzoek bouwt voort op de bestaande implementatie van logische operatoren van graded modal logic en demonstreert met succes de haalbaarheid van deze aanpak. De voornaamste bijdrage is de uitbreiding om de least fixpoint operator, die specifiek is voor graded μ-calculus, te kunnen gebruiken. Bovendien gaat het onderzoek verder dan de implementatie van operatoren door het ontwikkelen van een compiler die automatisch een graded μ-calculus uitdrukking vertaalt naar de bijbehorende RecGNN-architectuur.

Summary

Artificial intelligence (AI) and neural networks get the most attention and excitement in today's technological landscape. People think we are in an era where AI can do 'everything'. The expressiveness of such neural networks can be better explained through logic, which can be used as a counterexample to what people think neural networks can do. Formal logic plays a crucial role in characterizing the expressive power of program classes. This approach was pioneered by Edgar F. Codd in the 1970s when he introduced the relational model for data management, employing relational algebra and relational calculus for formalization.

Researchers have recently formalized graph neural network (GNN) architectures through logic. The ability of the GNN will be to classify nodes over graphs, with the classifier a boolean node classifier expressible as a logical formula. The state of the art is the work by Barceló et al. [BKM$^+$20], which is the inspiration of this thesis. Barceló et al. considered and proved the equivalence between AC-GNNs, a graph neural network architecture based on message passing neural networks, and formulas in graded modal logic (GML). Thus, GNNs are becoming very popular for specific use cases, but a lot remains unknown. That is why we will extend the remarkable research by Barceló et al. and dive into the logical expressiveness of recurrent graph neural networks (RecGNNs). These RecGNNs are like GNNs with recursion added; the equivalent logic also includes recursion. This logic is known as fixpoint GML or graded μ-calculus.

Background information

To establish a strong foundation for our proposed approach, we first delve into the background by reviewing several relevant fields of study.

This work utilizes graded modal logic, a fragment of first-order (FO) logic that can be used for local graph evaluation. Unlike first-order logic, GML does not use variables by focusing on individual nodes within a graph structure $G(V, E)$. Formulas are evaluated on a specific node $v \in V$. Predicates, which are interpreted as node colors in a colored graph, previously denoted as $\mathrm{Col}(x)$ in FO, will now be used as Col. Modal connectives are also introduced, the diamond operator $\Diamond^N$ and the box operator $\Box^N$, with N a positive integer. The FO expression $\exists^{\geq N} y(E(x, y) \wedge \varphi(y))$ is denoted by $\Diamond^N \varphi$. The box operator is defined as the negation of the diamond operator. Graded modal logic formulas are built inductively, combining basic propositions (i.e., Col), negation, conjunction, and the diamond operator. Operators like disjunction and the box operator can be derived using De Morgan's laws from the available operators.

Building upon GML, we can extend that logic by incorporating the least fixpoint operator μ and greatest fixpoint operator ν. This extension creates a more powerful logic known as graded μ-calculus. Note that the greatest fixpoint can be simulated as the negation of the least fixpoint.

Graph neural networks were developed to address the limitations of traditional neural networks in handling data structured as graphs. Traditional neural networks perform well with data in

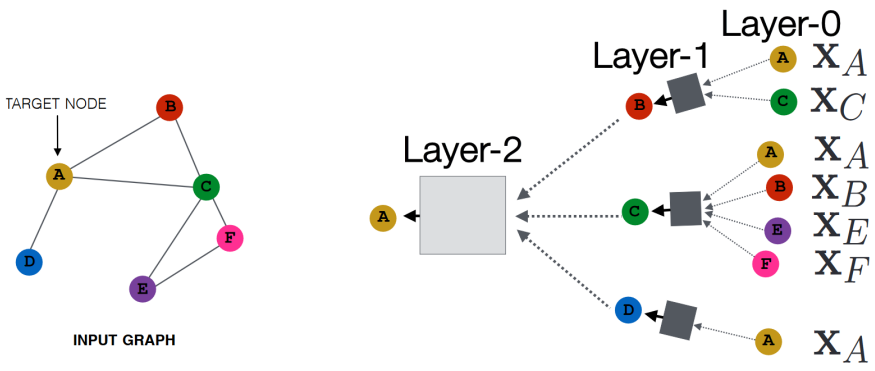

Figure 3: Message passing increases the receptive field of node A [Les23].

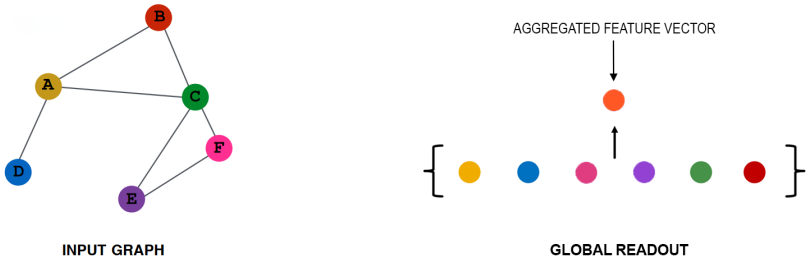

Figure 4: Global readout of the feature vectors of all the nodes in the graph.

grid formats, such as images and sequences, but struggle with relational data like knowledge graphs, social networks, and molecular structures. In such cases, the data consists of individual entities (nodes) and their connections (edges) with other entities. Traditional neural networks assume data independence, which is not applicable in graph-structured data where relationships between entities are crucial.

GNNs use layers similar to traditional neural networks but have unique mechanisms to handle graph data. These include local and global pooling layers that aggregate information from node neighbors and the entire graph, respectively. These aggregation processes, message passing, and global readout are crucial for learning from graph-structured data. During message passing, each node gathers information from its neighbors, updates its state, and propagates this information through the network layers, see Figure 3. A global readout will collect information from every input graph node and aggregate it into one feature vector, see Figure 4. The type of GNN which utilizes both mechanisms is known as the aggregate-combine-readout GNN (ACR-GNN).

Recurrent graph neural networks represent a specialized class of GNNs that incorporate principles of recurrence. A RecGNN is a multi-layer ACR-GNN that is iterated, functioning as a one-to-many network with a single input: the input graph. In each iteration, the output serves as the input for the subsequent iteration. This iterative process continues until the node classifications stabilize, meaning the values in the final layer of an iteration match those of the previous iteration.

Approach

The first step of our approach starts with studying the proof of Barceló et al. That proof functions as our foundation, and we need to understand every intermediate step. When analyzing their construction, an inefficiency was noticed in computing the features. Before extending the foundation, some optimizations were required to mitigate these inefficiencies.

The optimization lies within the feature vectors, which will be expanded throughout the neural network. Instead of holding every feature from the beginning, each layer will add its corresponding features to the vectors. This construction of flexible feature vectors also optimizes the number of layers needed in the ACR-GNN, significantly reducing the number of computations. Because of these changes, we must redefine the construction of Barceló et al., which consists of the essential GML operators.

The next step is to add recursion to both the logical and GNN aspects. Since the output of one iteration is used as an input in the following iteration, we needed an extra layer that transforms the expanded feature vector back to its original size. Every operator in graded modal logic can be computed in one iteration, but fixpoint operators may take several iterations. The number of iterations required for convergence is inherently unknown beforehand due to the dependence of these iterations on the input graph. Extra readiness features are introduced to mitigate this problem. Each feature gets an additional feature, representing its readiness. Due to our definition, the readiness will propagate through the subformulas of the logical formula we want to capture.

How is the least fixpoint operator μ_X implemented? Such an operator must check for each node if the old value equals the new one. Equality comparison can be established between the feature of the corresponding fixpoint relation X and another operator, the direct child of the μ-operator within the abstract syntax tree (AST). It is important to note that, as a binary tree, each μ-operator in the AST possesses only one child. This comparison allows us to determine whether every node in the input graph's value has been updated or reached its fixpoint. Verifying if all nodes in the input graph have reached their fixpoint necessitates a global readout operation. This step requires knowledge of the total number of nodes in the graph. To address this, we introduce an additional node color denoted by n, which is $true$ for every node in the input graph. This allows the global readout to determine if a global fixpoint has been reached effectively.

Nested fixpoint operators introduce another layer of complexity. When encountering a nested structure, the relations associated with these inner operators must be reset upon reaching their respective fixpoints. This reset is triggered by the enclosing (outer) fixpoint operator. The outer operator achieves this by verifying the readiness of its direct child operator. Readiness signifies the completion of computations within the child's subtree. Consequently, all operators within that subtree, including any further nested fixpoint operators, are considered ready for reset.

Implementation

Implementation-wise, our approach for a fixpoint operator can be divided into two parts. During the first part, we check the local readiness for every node in the input graph by checking the equality between the fixpoint relation and the direct child of the μ-operator. Simultaneously, we will also reset the fixpoint relations of nested fixpoint operators if present and ready. The equality is determined through a three-step process necessitated by the following logical expression:

$$(X \wedge child) \vee (\neg X \wedge \neg child).$$

The second part will perform a global readout to check the global readiness. This computational stage employs five steps to verify convergence and update the fixpoint relation. The extra node

color n is used to calculate the number of nodes present in the input graph to check convergence for every node. The following logical expression does this:

$$\neg\left[\sigma(n - count)\right].$$

Updating the fixpoint relation is only allowed when the entire subtree is ready because nested fixpoint relations cannot be ready yet. This check and the updating itself are done in the last two steps.

The above construction of an algorithm for simulating graded μ-calculus logical operators using RecGNNs demonstrates the capability for capturing positive graded μ-calculus formulas with these networks. We also implemented a corresponding compiler that takes a graded μ-calulus expression as its input. Such graded μ-calculus expressions have limited operators, so converting them from infix notation to prefix notion is relatively easy. This notation enables the construction of the associated abstract syntax tree. The compiler can construct the RecGNN architecture with the correct layer order and parameters by traversing this tree using a post-order traversal. Each node gets an ID during this traversal, and the base node colors are identified. These node colors are necessary to start the construction of the initial feature vector.

By leveraging the binary tree structure, the necessary parameters for each layer, such as child nodes or nested μ-operators, are easily identifiable. This is achieved by constructing the tree so each node possesses efficient access mechanisms. It can directly retrieve its immediate children and effortlessly explore the subtree with itself as the root.

Conclusion

This thesis investigates the extension of Barceló et al.'s work by constructing a recurrent graph neural network capable of evaluating graded μ-calculus expressions. Building upon the existing implementation of logical operators from graded modal logic, the research successfully demonstrates the feasibility of this approach. The core contribution lies in extending the framework to handle the least fixpoint operator specific to graded μ-calculus. Furthermore, the research goes beyond operator implementation by introducing a compiler that automatically translates a graded μ-calculus expression into a corresponding RecGNN architecture.

Contents

Chapter 1

Introduction

Artificial intelligence (AI) and neural networks get the most attention and excitement in today's technological landscape. These terms are no longer only used by academics and employees in computer science and other relevant fields. Everyone has heard of OpenAI and Google's latest innovations and even used their tools. People think we are in an era where AI can do 'everything'. We can better explain the expressiveness of such neural networks through logic, which can be used as a counterexample to what people think neural networks can do.

1.1 Motivation

Logic is helpful for formalizing program classes to understand that class's expressive power. Edgar F. Codd introduced such formalization in the 1970s for relational algebra and relational calculus, the relational model for data management [Cod70]. Initially, Codd proposed an abstract model of databases with relations as the data structures and an algebra for describing the queries. The following articles from Codd presented a query language based on the predicate calculus of first-order logic, equivalent to the algebra used in his model. The model does not specify a query language, but SQL is the most dominant query language for relational databases.

Codd's relational model significantly impacts how database management systems (DBMS) work today. The query compiler executes a given query and utilizes the equivalence between SQL and relational algebra. The first step is translating the SQL query to a relational algebra expression. So, it is still crucial for DBMS but has become classical. Researchers have recently formalized graph neural network (GNN) architectures through logic. The ability of the GNN will be to classify nodes over graphs, with the classifier a boolean node classifier expressible as a logical formula.

1.2 Problem definition

The state of the art is the work by Barceló et al. [BKM$^+$20], which is the inspiration of this thesis. Barceló et al. considered and proved the equivalence between AC-GNNs, a graph neural network architecture based on message passing neural networks (MPNNs), and formulas in graded modal logics (GML). Other works [GSVdB22] discuss similar equivalences using MPNNs. GNNs are becoming very popular for their use cases; a lot remains unknown. Because of this, we will extend the remarkable research by Barceló et al. and dive into the logical expressiveness of recurrent graph neural networks (RecGNNs). These RecGNNs are like GNNs with recursion added; the equivalent logic will also include recursion. This logic is known as fixpoint GML or graded μ-calculus.

In this thesis, a feasibility study, we will attempt to answer the following research question:

> Can we extend the work of Barceló et al. to a type of GNN that uses recursion and can evaluate a graded μ-calculus expression?

There are many GNN architectures and their corresponding implementation available. For our research question, we will use the most basic variant, the message passing neural network, without any improvements. This is like the aggregate-combine-readout GNN (ACR-GNN) from Barceló et al. but we will extend this architecture by adding recurrence. This means that the output from one iteration will be the input of the next iteration, and so on. We will construct a compiler-like program that outputs a RecGNN architecture based on a graded μ-calculus formula as input to answer our research question. The architecture is a sequence of layers that can evaluate the logical formula together. In addition to the compiler, an evaluator class has been implemented. This evaluator takes an input graph as its parameter and uses it on a given architecture. By applying the logical formula encoded within the architecture, the evaluation process determines the properties of each node in the input graph.

Chapter 2

Logics as Query Languages

This chapter establishes a foundation for understanding logic's role in database querying. It begins by exploring the core principles of Codd's relational model and its connection to the prevalent SQL language. Following this, fixpoints will be introduced, utilizing Datalog and relational calculus formulas to demonstrate their application within this domain. Readers with established expertise in this area may find this chapter familiar; however, it is part of the motivation of this thesis. This chapter will also discuss some concepts on which a considerable amount of time was spent trying to understand the setting of this thesis.

2.1 Codd's relational model

The relational model is a typical but valuable introduction to relational databases. We will dive into this model briefly by recapping relational algebra and relational calculus, focusing on the equivalences between these languages. The mentioned query language in the model, specifically SQL, will be discussed in section 2.2.

2.1.1 Relational algebra

Relational algebra is the more operational language that provides clear instructions to achieve the desired outcome. It uses a set of operators applied in a particular order to achieve the result by manipulating the data. Relations are the data and are the fundamental building blocks in relational algebra. Operators will perform various tasks on these relations. Of the operators that can be used, we meanly consider the essential ones. These include:

- projection (π);
- selection (σ);
- cartesian product ($\times$);
- union ($\cup$);
- difference ($-$);
- rename (ρ).

Assuming familiarity with the operators, we will proceed with their application in this context without revisiting their functionalities. The expressions, created by combining operators on defined relations, clearly express data manipulation tasks and can be used to reason about and optimize queries for the database system. This logic helps us understand the underlying principles of data management by formalizing it. On the other hand, the language is less intuitive for novice users than SQL, but that is not the importance of this notion. To conclude the

recap of relational algebra, a running example will be introduced that will be used to highlight the equivalence between relational algebra and relational calculus, which we will discuss next.

Consider the following relational schema:

- $R(A, B)$
- $S(C, D)$
- $T(E, F, G)$

We want to return the pairs (x, y) such that a tuple exists in R with $(x,\ i)$, the tuple $(i,\ j)$ exists in S and finally T has a tuple with $(j,\ 5,\ y)$. The simplest, without any optimizations, corresponding relational algebra expression is:

$$\pi_{R.A,T.G}\ \sigma_{R.B=S.C \wedge S.D=T.E \wedge T.F=5}(R \times S \times T).$$

All mentioned relations in the text are present in the expression. On the output of the cartesian products, there is a selection operator (σ) that holds all the constraints and conditions described in the text. The projection operator (π) outputs the correct tuple, the pair (x, y).

2.1.2 Relational calculus

Relational calculus is a declarative language because it does not provide instructions on achieving the desired outcome like relational algebra. Instead, it focuses on the 'what' aspect of what the user wants to achieve compared to the 'how' aspect, which is the main focus of relational algebra. Relational calculus employs a logical paradigm that allows the formulation of queries using predicate logic, which is more specific first-order logic without function symbols. First-order logic will express the constraints and conditions like the selection operator in relational algebra.

Propositional logic

Before discussing first-order logic, we look at propositional logic. Propositional variables represent elements from a defined set, typically described by $\{p, q, r, \dots\}$, possibly with subscripts, which can be $true$ or $false$. How do we connect these variables and constants like a sentence? That is why there are connectives that will complete the syntax for propositional calculus. The unary connective negation ($\neg$) and several binary connectives:

- conjunction ($\wedge$);
- disjunction ($\vee$);
- implication ($\rightarrow$);
- equivalence ($\leftrightarrow$).

De Morgan's laws on connectives

Note that only the negation connective and conjunction connective are part of the essential connectives. It is easy to see that we can construct the latter three connectives using the first two through De Morgan's laws. Given this principle's pivotal role in this thesis, a concise explication of the constructions is given.

Firstly, consider the expression $A \vee B$ (disjunction connective), which can be rewritten as $\neg(\neg A \wedge \neg B)$. The disjunction connective is generally considered essential because it is often used in a single propositional formula. Next, consider the expression $A \rightarrow B$ (implication connective), which is logically equivalent to $\neg A \vee B$. Finally, we have $A \leftrightarrow B$, which can be written as $(A \rightarrow B) \wedge (B \rightarrow A)$ which in its turn is logically equivalent to $(\neg A \vee B) \wedge (A \vee \neg B)$.

Truth assignment

A truth assignment for a set V of propositional variables is a function $\xi : V \to \{true, false\}$. The truth value of a propositional formula φ, written as $\varphi[\,\xi\,]$, is defined inductively. For example:

- $true[\,\xi\,] = true$;
- $\varphi = p$ with p a propositional variable, then $\varphi[\,\xi\,] = \xi(p)$;
- $\varphi = (\neg\psi)$ then $\varphi[\,\xi\,] = true$ iff $\psi[\,\xi\,] = false$;
- $(\psi_1 \vee \psi_2)[\,\xi\,] = true$ iff at least one $\psi_1[\,\xi\,] = true$ or $\psi_2[\,\xi\,] = true$.

We say that φ is true under ξ only if $\varphi[\,\xi\,] = true$ and similarly for false. If a formula φ has at least one truth assignment that makes it true, then we call φ satisfiable; otherwise, it is unsatisfiable. When every truth assignment makes φ true, we call it valid.

First-order logic

Until now, we have only discussed propositional logic, which is the foundation of first-order logic. Now, we extend that with the more complex first-order logic. The main new features of first-order logic are the addition of predicates and quantifiers. Predicates specify a relation between variables, which represent arguments to predicates. These are typically denoted by capital letters of their corresponding relation from the relational schema. Each predicate can have some input variables, and we call this number of variables the arity of the predicate. Thus, a boolean variable from propositional logic can be considered a predicate name of arity 0.

The variables V used in first-order logic range over elements of a defined domain D, just like propositional logic does for its variables. Quantifiers $\exists$ (existential) and $\forall$ (universal) will range over these elements to assign a specific value to variable x if we consider the formula $\exists x(\varphi)$. This assignment is done for each variable V using a function ν that maps each variable to a value in D. What are considered formulas? This is defined inductively:

- Each atomic formula;
- $x = y$ with x, y variables;
- Given two formulas, denoted by φ_1 and φ_2, these formulas can be combined using a connective from propositional logic. This combination results in a new formula, such as $\varphi_1 \wedge \varphi_2$;
- Moreover, if φ is a formula and x a variable, then $(\exists x(\varphi))$ and $(\forall x(\varphi))$ are also formulas.

Next, we want to discuss free and bound variables in a formula. A variable x is considered free in a formula φ if it occurs somewhere outside the scope of a quantifier $\exists x$ or $\forall x$. Reconsider the assignment function ν, which only has to assign free variables to evaluate the formula φ.

Lastly, we want to extend our running example with the corresponding relational calculus formula. We start with the desired output pair (x, y) and add the constraints and mentioned relations later. One corresponding relational calculus formula is the following:

$$\{(x, y) \mid \exists i\ R(x, i) \wedge \exists j\ S(i, j) \wedge \exists y\ T(j, 5, y)\}.$$

Note that we use a comprehension operator $|$, which we did not discuss above. This separates the definition of the pair (x, y) from the condition. Everything before the comprehension operator is our output, similar to what the projection operator does in relational algebra. After the comprehension operator, the condition contains all mentioned relations and constraints or conditions. For every relation, we use a predicate with the same letter and equal arity as defined in the relational schema.

The cartesian product from relational algebra is the logical conjunction that combines the relation predicates. The projection operator is the tuple before the comprehension operator,

and the selection operator is the combination of an existential quantifier with its variable used in a relation predicate.

Comparing the relational calculus formula with the text defining it, it is easy to see that it is more like the text than our relational algebra expression. That is because relational calculus is more declarative.

2.2 SQL

Codd's relational model does not mention a specific query language so we will use the most prominent one, Structured Query Language (SQL). This is like relational calculus, a declarative language because the SQL query that a user writes focuses solely on the 'what' aspect.

When the query is executed, the query compiler will choose the best plan based on statistics without the user's involvement. The query compiler thus focuses on the 'how' aspect, which makes SQL very user-friendly. Note that this compiler first translates the SQL query to relational algebra. Because this is part of our motivation, only the standard queries of the form select-from-where are considered. This also perfectly aligns with the operators from relational algebra and calculus from section 2.1. Take the following select-from-where statement:

```
SELECT select-list
FROM R1, R2 T1, ...
WHERE where-condition
```

When there are no subqueries present in the where-condition, the statement can be translated to the following relational algebra expression:

$$\pi_{\text{select-list}}\sigma_{\text{where-condition}}(R1 \times \rho_{T1}(R2) \times \dots).$$

The where-condition will hold all constraints and conditions the query has. If the where-condition solely has conjunctions of atomic equality conditions, these are called conjunctive queries. An example of such queries is the running example from section 2.1. Following the mentioned conversion, the query will look like this:

```
SELECT R.A, T.G
FROM R, S, T
WHERE R.B = S.C AND S.D = T.E AND T.F = 5;
```

2.3 Datalog

SQL is the most prominent query language, but Datalog is a more versatile query language for this thesis. A stepping stone to the entire language will be nonrecursive Datalog without negation [AHV95]. Datalog is a variant of logic programming designed for relational data while excluding function symbols. So, this language, nonrecursive Datalog without negation, has the same expressive power as relational algebra and relational calculus. They can express the class of queries specified as first-order queries.

2.3.1 Datalog syntax

A Datalog program P consisting of a set of rules over a schema R looks like this:

$$\begin{aligned} S_1 &\leftarrow body_1 \\ S_2 &\leftarrow body_2 \\ &\vdots \\ S_k &\leftarrow body_k \end{aligned}$$

Line	Station	Next station
Red	Oxford Circus	Tottenham Court Road
Red	Oxford Circus	Bond Street
Red	Tottenham Court Road	Holborn
Red	Holborn	Chancery Lane
Blue	Holborn	Covent Garden
Blue	Covent Garden	Leicester Square

Table 2.1: Part of the Tube database (London Underground).

where no relation name in R occurs in the rule head, which are the relations $\{S_1, S_2, \ldots, S_k\}$ who appear before the arrow. The part after the arrow is the body of the rule. Diving into the program, we will take a closer look at one rule, which is an expression of the form:

$$S_1(t) \leftarrow R_1(t_1), \ldots, R_k(t_k),$$

where $k \geq 1$ and $R_1, \ldots, R_k$ are relation names from R with each a tuple t_i of the matching arity. These tuples consist of several variables, and the variables from t must occur at least once in one of $t_1, \ldots, t_k$. A nonrecursive Datalog program also does not have the relation from the rule head in its body. Continuing the running example, which is a conjunctive query, we would get the following Datalog program:

$$Ans(x, y) \leftarrow R(x, i),\ S(i, j),\ T(j, 5, y).$$

consisting of only one rule with three relations in a conjunction.

The next step is adding recursion to Datalog by allowing the relation from the rule heads to be present in its body. We will give an example based on a graph-like structure, which is the main focus of this thesis. Consider the underground lines of the London Underground. The Tube map can be considered a directed graph from and to different underground stations. Suppose all connections are in a Tube database like Table 2.1, and we want to see the neighboring stations of Oxford Circus. This is a simple, nonrecursive Datalog program: $Ans(x) \leftarrow Tube(l, \text{`Oxford Circus'}, x)$. What if we want to check all reachable stations from Oxford Circus? We need a recursive program like:

$$\begin{aligned} Reachable(x, y) &\leftarrow Tube(l, x, y). \\ Reachable(x, y) &\leftarrow Reachable(x, z),\ Tube(l, z, y). \\ Ans(y) &\leftarrow Reachable(\text{`Oxford Circus'}, y). \end{aligned} \tag{2.1}$$

where the first rule states that $Reachable(x, y)$ is true for every pair of stations in the Tube database because of the direct connection. The second rule defines the reachability between two stations x and y if there exists another station z that is reachable from x and has a direct link from z to y according to the Tube relation. The last rule answers our question about the stations reachable from Oxford Circus.

The answer will first provide a list of the stations directly connected, one hop away, to Oxford Circus: Tottenham Court Road and Bond Street. It will then expand the list to include the directly connected stations of the stations in the current answer. At step k, we get information about the stations k hops away. So, the answer will look like this: Tottenham Court Road, Bond Street, Holborn, Chancery Lane, and so on.

2.3.2 Fixpoint semantics

After discussing the example of the reachability in the graph of the London Underground, we introduce the immediate consequence operator, an operational semantics for Datalog programs. Like the example, the operator will start with known facts and produce new ones by deducing

them by the rules of the Datalog program. The immediate consequence operator of a Datalog program P is denoted as T_P.

Before diving deeper into this operator, we need to discuss some notations. Let P be a Datalog program; an extensional relation only occurs in the body of the rules of P. An intentional relation is a relation that occurs in the head of some rule of P. Then, the extensional schema, denoted $edb(P)$, consists of all extensional relation names, and the intentional schema, denoted $idb(P)$, consists of all intentional relation names. In the London Underground example P_{LU}, the extensional schema is $\{Tube\}$, and the intentional schema is $\{Reachable, Ans\}$. The schema of P denoted $sch(P)$, is defined as the union of $edb(P)$ and $idb(P)$. The immediate consequence operator needs an input instance $\mathbf{I}$ containing the program's data. In our example, $\mathbf{I}$ is the Tube database. We can say that $\mathbf{I}$ is an instance over $edb(P)$ and let $\mathbf{B}(P, \mathbf{I})$ be the instance over $sch(P)$.

The immediate consequence operator T_P is monotone, which means that for each $\mathbf{I}$, $\mathbf{J}$, $\mathbf{I} \subset \mathbf{J}$ implies $T(\mathbf{I}) \subset T(\mathbf{J})$. When computing $T_P(\mathbf{I})$, $T_P^2(\mathbf{I})$, etcetera we can see that

$$\mathbf{I} \subseteq T_P(\mathbf{I}) \subseteq T_P^2(\mathbf{I}) \subseteq \cdots \subseteq \mathbf{B}(P, \mathbf{I}).$$

Let N be the number of facts in $\mathbf{B}(P, \mathbf{I})$, then the fixpoint will be reached after at most N steps. Meaning, $\forall i \geq N : T_P^i(\mathbf{I}) \subseteq T_P^N(\mathbf{I})$. This fixpoint is denoted by $T_P^\omega(\mathbf{I})$. Continuing the London Underground example, taking only the six rows of Table 2.1, we get the input instance

$$\mathbf{I} = \{Tube(\text{'Red'}, \text{'Oxford Circus'}, \text{'Tottenham Court Road'}), \dots\}.$$

Then we have

$$\begin{aligned}
&T_{P_{LU}}(\mathbf{I}) = \mathbf{I} \cup \{R(\text{'Oxford Circus'}, \text{'Tottenham Court Road'}), \dots\} \\
&T_{P_{LU}}^2(\mathbf{I}) = T_{P_{LU}}(\mathbf{I}) \cup \{R(\text{'Oxford Circus'}, \text{'Holborn'}), R(\text{'Holborn'}, \text{'Leicester Square'}), \dots\} \\
&T_{P_{LU}}^3(\mathbf{I}) = T_{P_{LU}}^2(\mathbf{I}) \cup \{R(\text{'Oxford Circus'}, \text{'Chancery Lane'}), \dots\} \\
&T_{P_{LU}}^4(\mathbf{I}) = T_{P_{LU}}^3(\mathbf{I}) \cup \{R(\text{'Oxford Circus'}, \text{'Leicester Square'})\} \\
&T_{P_{LU}}^5(\mathbf{I}) = T_{P_{LU}}^4(\mathbf{I}).
\end{aligned} \tag{2.2}$$

After four iterations, the fixpoint has been reached, thus $T_{P_{LU}}^\omega(\mathbf{I}) = T_{P_{LU}}^4(\mathbf{I})$. Each iteration added an extra intermediate station between the start and finish if one exists in the Tube database. The longest connection possible is from Oxford Circus to Leicester Square, which has three stops. It is easy to see that the fixpoint is reached at iteration four, where iteration one copies the facts from relation $Tube$ to the relation R (abbreviation of relation $Reachable$).

2.3.3 Datalog with negation

With the foundation of Datalog syntax established through the illustrative example of computing the transitive closure of the London Underground network, we proceed to extend the expressiveness of rule bodies by incorporating negation. Datalog with recursion in combination with negation is denoted as Datalog$\neg$. Consider computing the complement of the transitive closure of the London Underground. Re-using the relation R (*Reachable*) from 2.1 results in the following Datalog program Q:

$$\begin{aligned}
R(x, y) &\leftarrow Tube(l, x, y). \\
R(x, y) &\leftarrow R(x, z),\ Tube(l, z, y). \\
N(x, y) &\leftarrow Tube(l, x, y), \neg R(x, y).
\end{aligned} \tag{2.3}$$

where the complement of the transitive closure is computed in the intentional relation N. For this example, the extensional schema looks like $\{Tube\}$, and the intentional schema looks like $\{R, N\}$. Examining the program's output will show a surprising deviation from expectations.

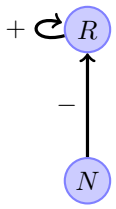

Figure 2.1: Precedence graph for computing the complement of the transitive closure.

In essence, relation R represents all reachable station pairs, where one station can be directly or indirectly accessed from the other. Conversely, relation N encompasses all unreachable station pairs, indicating no travel route exists between them.

However, relation R will result in the pairs present in 2.2. Simultaneously, the program will also compute the pairs for relation N. Because this is a concurrent operation, relation R is still empty in iteration one, resulting in the addition of every pair from the Tube database. In iteration two, relation R is still growing, but relation N has already converged. This deviates from the expected outcome, which is a precise characterization of all unreachable station pairs. This is not the expected result because relation N depends on relation R. This can be visualized in the precedence graph in Figure 2.1. Each node is an *idb*, and there exists an edge $p \to q$ if there is a rule for p that uses q in its body. A label $-$ is added if q is negated in that rule, and label $+$ otherwise.

2.3.4 Stratified Datalog

The Datalog program Q from 2.3 gives a wrong output because relation R has not been converged yet when relation N is being computed. Intuitively, we do this by first computing R before computing N. Stratified Datalog is one solution to this problem, and it partitions the program into strata as our intuition does.

A stratification of a Datalog¬ program P partitions P into Datalog sub-programs $P^1, \ldots, P^m$ (strata) such that:

- All rules defining the same *idb* relation must be in the same stratum.
- A nonnegated relation R present in the body of a rule R' can be an *edb*, a relation from previous strata, or a relation from the stratum where R' is defined.
- A negated relation R present in the body of a rule R' can be an *edb* or a relation from previous strata.

Stratification works because we freeze an *idb* defined in a stratum. In the following strata, we can use it like an *edb*. Comparing this definition to the precedence graph in Figure 2.1, strata are divided by edges labeled with $-$. Thus, Datalog program 2.3 can be stratified in two Datalog subprograms as follows:

$$\begin{aligned} &\text{(P1)}\ R(x,y) \leftarrow Tube(l,x,y). \\ &\text{(P1)}\ R(x,y) \leftarrow R(x,z),\ Tube(l,z,y). \\ &\text{(P2)}\ N(x,y) \leftarrow Tube(l,x,y), \neg R(x,y). \end{aligned} \tag{2.4}$$

Notice that (P1) defines the *idb* R, and (P2) defines N; N has a negated *idb* in the body of a rule that defines it, but this *idb* appears in the previous stratum (P1). A Datalog¬ program can have many possible stratification, but not every Datalog¬ program can be stratified. A Datalog¬ is stratifiable if the precedence graph has no directed cycles with a negative edge [AHV95].

2.4 Least fixpoint

Having established the foundational concept of fixpoints using Datalog and the stratified version, this section uses the same concept on relational calculus formula. Building upon the relational calculus syntax, this approach introduces a fixpoint operator for extended expressiveness. The fixpoint operator is denoted by μ_T, with T the relation on which the fixpoint is applied. The relation T starts empty, $T = \varnothing$, and the fixpoint operator will add new tuples to the fixpoint relation T in each iteration until it reaches its fixpoint. This type of fixpoint is the least fixpoint, which will be the focus of this thesis due to its prominence in the field. In contrast, there are other types, like partial fixpoints that do not guarantee to reach a fixpoint or inflationary fixpoints that ensure a fixpoint by forcing the growth of the fixpoint relation. While some time was spent understanding these types, they are not relevant enough to be discussed in this thesis.

2.4.1 Positive formulas

Before exploring least fixpoints, a discussion on positive formulas becomes essential. This recognition stems from the necessity of exercising caution when combining recursion with negation, as previously demonstrated in subsection 2.3.3.

To ensure convergence to a fixpoint, we can use monotonicity. When a query is monotone, a least fixpoint always exists. Unfortunately, monotonicity is an undecidable property, rendering it unsuitable for our purposes. An alternative is only allowing positive formulas, which also leads to convergence. This means that the relation T on which the fixpoint is applied, i.e., $\mu_T(\varphi(T))$, only appears under an even number of negations in the syntax tree of the formula. Other relations may occur under an odd number of negations because they do not interfere with the fixpoint similar to the *edb* relations from Datalog.

Continuing with the transitive closure of the London Underground, we would get the following calculus formula:

$$\mu_R \left[Tube(l_1, x, y) \vee R(x, y) \vee \exists z (R(x, z) \wedge Tube(l_2, z, y)) \right] (x, y). \tag{2.5}$$

All tuples from relation $Tube$ will be added to R in the first iteration. The content of relation R matches the content of the immediate consequence operator from subsection 2.3.2. It is trivial that Equation 2.5 is a positive formula because no negations are present. The subformula $\varphi(T)$ is typically enclosed within parentheses or brackets to ensure an unambiguous interpretation. This resembles the concept of partitions or strata employed in stratified Datalog.

Translating the Datalog program 2.3, where the complement of the transitive closure is computed, into a calculus formula will result in the following formula:

$$\left[Tube(l, x, y) \wedge \neg \mu_R \left(Tube(l_1, x, y) \vee R(x, y) \vee \exists z (R(x, z) \wedge Tube(l_2, z, y)) \right) (x, y) \right] (x, y). \tag{2.6}$$

This exemplifies the compelling parallel between the fixpoint operator and stratified Datalog. While stratified Datalog employs strata for iterative evaluation, the fixpoint operator implicitly leverages the same principle. Crucially, each embedded fixpoint operator must be converged to its fixpoint before evaluating the entire formula. This ensures the derivation of the correct result. Since the fixpoint relation R is not negated within the formula, it can be classified as a positive formula. When the fixpoint is reached, the whole subformula $\mu_R(\varphi(R))$ can be used like an *edb* and can be negated.

2.4.2 Multiple fixpoints

Having established the behavior of formulas with a single fixpoint operator, we now quickly delve into more complex scenarios involving multiple operators to explore the concepts of nested fixpoints and simultaneous fixpoints. One scenario involving multiple fixpoints is the following:

$$\mu_X \left[\varphi(X) \right] \wedge \mu_Y \left[\psi(Y) \right].$$

The formula can have two or more fixpoint operators that iterate independently toward their respective fixpoints. These operators do not influence each other's convergence, and each can reach its fixpoint at a different iteration. Consequently, evaluation of the entire formula proceeds only after all fixpoint operators have converged. The precedence graph associated with such formulas will have isolated sub-graphs for each fixpoint operator, signifying the absence of mutual dependencies. This characteristic aligns with the concept of a single stratum in stratified Datalog, where all rules can be evaluated within the same level.

Nested fixpoints involve two or more fixpoint operators embedded within each other. Unlike the previous scenario, these fixpoint operators exhibit mutual influence on their convergence. A nested fixpoint operator's evaluation hinges on the prior convergence of all its embedded fixpoint operators. Consequently, the evaluation process commences with the most deeply nested operator, iteratively proceeding outwards. The formula below has one nested fixpoint operator.

$$\mu_X \left[\varphi(X) \wedge \mu_Y[\psi(X, Y)]\right].$$

Here $\psi(X, Y)$ involves both X and Y. This corresponds to a nested iteration where subformula ψ can use fixpoint relations X and Y in contrast to $\varphi(X)$, which can only use the fixpoint relation X. The nested fixpoint can be referred to as a simultaneous fixpoint because it depends on multiple fixpoint relations. Formulas with nested fixpoint operators generally possess a single, connected precedence graph. This absence of isolated sub-graphs stems from the inherent dependencies between nested operators.

2.4.3 Greatest fixpoint

This section briefly introduces the greatest fixpoints due to their close connection to the least fixpoints. While the least fixpoints begin with an empty relation, the greatest fixpoints commence with the universal relation, encompassing 'everything'. Notably, every monotonic operator guarantees the existence of a least fixpoint when starting iterations from the empty set. Dually, every monotonic operator possesses a greatest fixpoint obtainable by initializing iterations with 'everything'. The greatest fixpoint is denoted by ν_T, where T represents the relation on which the fixpoint is applied.

Every greatest fixpoint $\nu_T[\varphi(T)]$ can be written as a least fixpoint using double negation and this logical equivalence:

$$\neg\nu_T[\varphi(T)] \equiv \mu_T[\neg\varphi(\neg T)]. \tag{2.7}$$

Similar to how De Morgan's laws enable the simulation of logical operators like implication and equivalence, it is possible to simulate a greatest fixpoint using a least fixpoint.

Chapter 3

Graded Modal Logics

This chapter expands on prior explorations of query languages by introducing graded modal logic (GML) and its extension, μ-calculus. Both hold particular relevance to graph structures, the central theme of this thesis. To understand the motivations behind GML and its limitations, the fragment FOC_2 of first-order logic is first examined. This analysis will highlight its restricted expressiveness when dealing with graphs and pave the way for GML and μ-calculus, which will be delved into next. Throughout the chapter, numerous connections between the discussed concepts and graph structures will be established using illustrative examples.

3.1 FOC_2

Before delving into graded modal logic, we start with FOC_2 as a stepping stone. The logic FOC_2 is the two-variable fragment of first-order logic extended with counting quantifiers. Why do we begin with FOC_2 classifiers? It connects to model checking and its well-known satisfiability problem: determining whether a formula is true in a specific state within a Kripke structure. Unfortunately, for first-order logic in its entirety, many problems like satisfiability and model-checking verification are undecidable. However, only highly restricted fragments of first-order logic become decidable. Notably, these decidable fragments are typically defined using the concept of bounded quantifier alternation [Var96].

Having established the motivation for the logic FOC_2, some examples are considered to illustrate this logic further. All formulas discussed in this chapter will be evaluated using graph structures that can be interpreted as relational databases. Unary predicates are best understood as node colors, and the only binary predicate $E(x, y)$ denotes an edge in the graph between node x and node y. The predicates can be interpreted as database relationships when comparing a graph to a relational database. Logical (node) classifiers $\varphi(x)$ are unary formulas and will be evaluated over all graph nodes using the free variable x. Nodes v are classified as $true$ if $(G, v) \models \alpha$ and the other nodes, for which $(G, v) \not\models \alpha$, are classified as $false$.

First, let us take a look at a logical node classifier in first-order logic that classifies x according to whether the formula holds for x or not:

$$\alpha(x) := \mathrm{Orange}(x) \wedge \exists y(E(x, y) \wedge \mathrm{Blue}(y)). \tag{3.1}$$

Equation 3.1 has one free variable, x, because it occurs outside the scope of a quantifier. Variable y is bounded because it occurs within the quantifier $\exists y$. The formula above evaluates to $true$ for nodes v whose color is Orange and have at least one Blue neighbor.

The following discussion delves into how FOC_2 limits the expressive power of FO logic. First, subscript two, which relates to the number of variables used in the formula, will be explained.

It is easy to see that this reduces the expressive power. However, certain formulas can be rewritten using two variables by reusing variables. If we extend Equation 3.1 to:

$$\beta(x) := \text{Orange}(x) \wedge \exists y(E(x,y) \wedge \text{Blue}(y)) \wedge \exists z(E(x,z) \wedge \text{Purple}(z)), \tag{3.2}$$

which uses three variables x, y, z, but it is possible to write an equivalent formula for $\beta(x)$ using two variables. Variables must be reused, where variable z will be replaced by y because their scopes do not interfere. Another example will alternate between two variables to create a path of nodes. An example is:

$$\gamma(x) := \text{Orange}(x) \wedge \exists y\left[E(x,y) \wedge \exists x(E(y,x) \wedge \text{Blue}(x))\right],$$

which satisfies Orange nodes for which a path of length two leads to a Blue node.

The last difference between FOC_2 and FO logic is the addition of counting quantifiers, hence the C in the logic abbreviation. Using the following equation as an example:

$$\delta(x) := \text{Orange}(x) \wedge \exists y\left(\neg E(x,y) \wedge \exists z_1, z_2\left[(E(y,z_1) \wedge E(y,z_2) \wedge z_1 \neq z_2 \wedge \text{Blue}(z_1) \wedge \text{Blue}(z_2)\right]\right)$$

which evaluates to *true* for nodes x whose color is Orange and there is another node y, that is not connected to x and has at least two Blue neighbors z_1 and z_2. The counting quantifier $\exists^{\geq N}$ for every positive integer N can express $\delta(x)$ by only using two variables. This counting quantifier $\exists^{\geq N}$ functions similarly to the regular quantifier $\exists$ and reduces the number of variables used in that subformula. The regular quantifier $\exists x(\varphi)$ expresses the existence of a node x satisfying property φ. The counting quantifier $\exists^{\geq N} x(\varphi)$ expresses the existence of at least N distinct nodes x satisfying property φ. With the counting quantifier $\exists^{\geq 2}$, and reusing variable x, formula $\delta(x)$ can be expressed by the classifier:

$$\varepsilon(x) := \text{Orange}(x) \wedge \exists y\left(\neg E(x,y) \wedge \exists^{\geq 2} x\left[(E(y,x) \wedge \text{Blue}(x)\right]\right),$$

which uses two variables x, y, and the counting quantifier, so it qualifies as a logic formula in FOC_2. Note that an existential quantifier is not required in the formula.

Thus, FOC_2 allows the use of every first-order logic operator with additionally the counting quantifiers but restricts the number of variables to two. In consideration of the emphasis placed on graph structures within this work, the predicate set is defined as follows:

- predicates like $\text{Orange}(x)$ and $\text{Blue}(x)$ describing abstract node properties, which are unary relations (predicates with arity one);
- the edge predicate $E(x,y)$ describing an edge between node x and y in the input graph G. This is the only binary relation (predicate with arity two) allowed.

Note that the essential operators are the same as those in first-order logic. Other operators like the $\vee$ operator can be simulated using De Morgan's laws discussed in section 2.1.2. FOC_2 is strictly less expressive than FO because counting quantifiers can be mimicked in FO using more variables and inequalities.

3.2 Graded modal logic

Now that the logic FOC_2 has been explained, the connection to graded modal logic (GML) can be elucidated. FOC_2 is an excellent stepping stone because it uses first-order logic semantics. In contrast, graded modal logic uses different semantics but can be embedded into the two-variable fragment of first-order logic [Grä99].

3.2.1 Syntax using first-order semantics

First, we will compare graded modal logic to FOC_2 using the FO logic syntax. It uses the same predicates and operators but forces all subformulas using the existential quantifier to be

guarded by the edge predicate E. The guarded edge predicate ensures the locality property because we can no longer check whether some node somewhere in the graph exists with a property: $\exists y(\varphi)$. One can only check whether some neighbor y of node x satisfies φ, denoted as $\exists y(E(x,y) \wedge \varphi(y))$. This fragment of FO logic is defined as follows. A graded modal logic formula can take one of the following forms using the graph predicates and generalizing all node colors as $\mathrm{Col}(x)$:

- $\mathrm{Col}(x)$, where Col is a node color;
- $\neg\varphi(x)$;
- $\varphi(x) \wedge \psi(x)$;
- $\exists^{\geq N} y(E(x,y) \wedge \varphi(y))$,

where φ and ψ are graded model logic formulas and N a positive integer. Note that this is an inductive definition similar to the definition of FO logic.

3.2.2 Syntax of graded modal logic

Having established the syntax through familiar first-order logic notation, this section delves into the semantics of graded modal logic. Graded modal logic does not use variables because the logic is always locally evaluated. Using graphs, every graded modal logic formula φ is evaluated on a graph structure $G(V,E)$ and a node $v \in V$. Predicates previously denoted as $\mathrm{Col}(x)$ in FO will now be used as Col. Modal connectives are also introduced, the diamond operator $\Diamond^N$ and the box operator $\Box^N$, with N a positive integer. The FO expression $\exists^{\geq N} y(E(x,y) \wedge \varphi(y))$ is denoted by $\Diamond^N\varphi$. The expression $\Diamond\varphi$ defines given (G,x) that there exists a neighbor y of x for which φ evaluates to *true*. Denoted as $(G,x) \models \Diamond^N\varphi$, with $N = 1$. The diamond operator, $\Diamond$, is defined as a satisfaction relation together with the box operator $\Box$ [dR00]. The box operator is defined as $(G,x) \models \Box^N\varphi$ iff $(G,x) \models \neg\Diamond^N\neg\varphi$. Intuitively, the box operator $\Box^N\varphi$ evaluates to true given (G,x) if there are strictly less than N neighbors of x for which φ evaluates to $false$. The expression $\Box\varphi$ defines given (G,x) that every neighbor of x evaluates to *true*.

A graded modal logic formula can be inductively defined as:

- Col;
- $\neg\varphi$;
- $\varphi \wedge \psi$;
- $\Diamond^N\varphi$,

where φ and ψ are graded model logic formulas and N a positive integer. Again, using De Morgan's laws, operators like $\vee$ and $\Box$ can be simulated using the operators above.

3.2.3 Examples using graphs

Examples previously discussed, like Equation 3.2, qualify as a valid graded modal logic expression. Using the newly introduced semantics, the formula will look like:

$$\beta := \mathrm{Orange} \wedge \Diamond(\mathrm{Blue}) \wedge \Diamond(\mathrm{Purple}),$$

which is equivalent to Equation 3.2 and more comprehensible after an understanding of modal logic syntax is established. Using the graph structure from Figure 3.1, the nodes from that graph that satisfy formula β are nodes two and four. Nodes two and four are Orange and can reach node five, which is Purple. Both nodes can also reach a Blue node, nodes one and six, respectively. All other nodes do not satisfy β.

De Morgan's laws have been referenced many times throughout this thesis, enabling the utilization of essential operators. The following example will be a formula using the box operator.

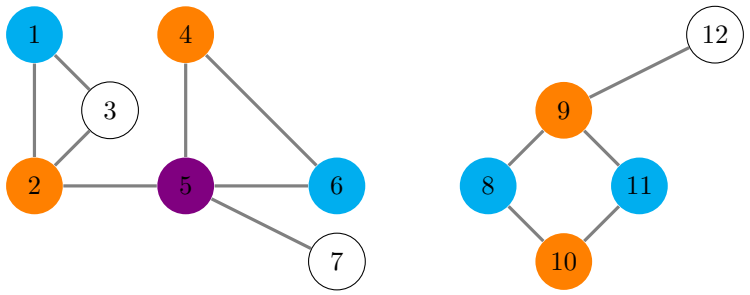

Figure 3.1: Graph structure used in the examples.

Subsequently, this formula will be converted to an equivalent formula using the diamond operator. Starting with the following formula:

$$\gamma_1 := \text{Blue} \wedge \Box(\text{Orange}),$$

which evaluates to *true* for nodes x whose color is Blue and every neighbor of that node is Orange. Using the logical equivalence between the operators $\Box$ and $\Diamond$, this yields the following equivalent formula:

$$\gamma_2 := \text{Blue} \wedge \neg\Diamond(\neg\text{Orange}),$$

which intuitively states that a node must be Blue and that a neighbor whose color is not Orange does not exist. This last constraint can be restated as follows: every neighbor must be Orange. Using the logical equivalence $\Box(\varphi) \equiv \neg\Diamond(\neg\varphi)$, formula γ_1 can be written as formula γ_2. A relevant graph structure like the one in Figure 3.1 can be discussed to elucidate this formula further. The nodes in the graph that satisfy the formula γ_1 or γ_2 are nodes eight and 11. Both nodes are Blue and can only reach Orange nodes nine and ten.

3.3 Graded μ-calculus

Having established graded modal logic, this section explores its extension by incorporating the least fixpoint operator μ and the greatest fixpoint operator ν. This addition results in the creation of a fixpoint logic. While the concepts of least and greatest fixpoints have been introduced using relational calculus in section 2.4, their functionality remains consistent within modal logics. Only the least fixpoint operator μ will be added to the essential operators because this fixpoint operator can simulate the greatest fixpoint operator ν; see Equation 2.7. Because the inner workings of these operators are already explained, some examples using graph structures will be discussed while using the modal logic semantics that are used in the graded μ-calculus.

A prime example of a graded μ-calculus formula using the least fixpoint is checking whether you can reach a node with a specific property. Given the graph structure depicted in Figure 3.1, this example aims to identify all nodes that can reach a Purple node. This can be denoted as:

$$\mu_X(\text{Purple} \vee \Diamond X). \tag{3.3}$$

It is easy to see that every node with ID one to seven can reach a Purple node, node five. The other isolated graph containing nodes with id eight till 12 can not reach such a Purple node and thus will not be added to relation X. One can iteratively compute the unary relation X by defining X_i for every iteration i until it reaches its fixpoint at iteration j, for which holds $X_{j+1} = X_j$. Applying this method on Equation 3.3 using the graph depicted in Figure 3.1 one

would get:

$$X_0 = \varnothing$$
$$X_1 = \{5\}$$
$$X_2 = \{2, 4, 5, 6, 7\}$$
$$X_3 = \{1, 2, 3, 4, 5, 6, 7\}$$
$$X_4 = \{1, 2, 3, 4, 5, 6, 7\}.$$

The fixpoint is reached in iteration four because every node from the left sub-graph is added to the fixpoint relation X. A convenient interpretation of relation X is the collection of nodes situated on any path that ultimately reaches a Purple node. Iteratively, one can interpret every iteration as follows:

$X_0 :=$ empty by the definition of μ

$X_1 :=$ all Purple nodes

$X_2 :=$ all Purple nodes or nodes with a Purple neighbor

$X_3 :=$ all Purple nodes or nodes with a Purple neighbor or nodes whose neighbor has ...

$X_4 := \ldots$

Reading and understanding can become increasingly cumbersome as the description progresses through multiple iterations. An alternative formulation that leverages the definition established in the previous iteration can be employed to address this readability concern. This approach simplifies the representation and enhances comprehension.

$X_0 :=$ empty by the definition of μ

$X_1 :=$ all Purple nodes

$X_2 :=$ all Purple nodes or nodes with a neighbor in X_1

$X_3 :=$ all Purple nodes or nodes with a neighbor in X_2

$X_4 :=$ all Purple nodes or nodes with a neighbor in X_3.

This intuitive recursive definition is much easier to read and understand. Technically, X_1 is defined as all Purple nodes or nodes with a neighbor in X_0, but X_0 is defined as the empty relation; thus, no node qualifies for the second condition.

Next, a model example will be discussed using the greatest fixpoint operator ν to identify all nodes that can only reach Orange nodes. This concept is expressed as follows:

$$\nu_X(\text{Orange} \wedge \Box X). \tag{3.4}$$

Note that no node of the graph in Figure 3.1 satisfies this constraint, resulting in an empty fixpoint relation X. Through the iterative process, relation X will progressively shrink until it becomes empty. This behavior stems from the definition of ν_X, which initializes the relation X as all nodes within the graph structure. As the iterations proceed, the relation is refined to progressively exclude nodes that do not satisfy the specified criteria, resulting in an empty set.

$$X_0 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$
$$X_1 = \{2, 4, 9, 10\}$$
$$X_2 = \varnothing$$
$$X_3 = \varnothing.$$

Iteratively, one can interpret every iteration as follows:

$X_0 =$ all nodes by the definition of ν

$X_1 =$ all Orange nodes

$X_2 =$ all Orange nodes and every neighbor is Orange

$X_3 =$ all Orange nodes and every neighbor is Orange and can only have Orange neighbors.

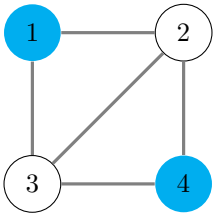

Figure 3.2: Considering Equation 3.5, the associated graph structure has property P assigned to nodes 1 and 4.

Thus, the least fixpoint can be described as a reachability constraint, and the greatest fixpoint describes a constraint on the elements that can be part of a solution path. The following example showcasing a nested fixpoint will conclude this chapter. The graph structure used for this example is depicted in Figure 3.2, and the following formula is used as the node classifier:

$$\nu_Y \left[\mu_Z (\Diamond Y \wedge (P \vee \Diamond Z))\right] \tag{3.5}$$

The formula holds for all nodes in the graph. The iterative definitions of fixpoint relations Y and Z will demonstrate this. Initialization begins with assigning 'everything' and 'nothing' to these relations, respectively. Next, the fixpoint of μ_Z will be computed because it is the most deeply nested fixpoint operator. Lastly, the greatest fixpoint ν_Y will be updated, resulting in:

$$\begin{aligned}
Y_0 &= \{1,2,3,4\} \\
&\quad Z_0^0 = \varnothing \\
&\quad Z_1^0 = \{1,4\} \\
&\quad Z_2^0 = \{1,2,3,4\} \\
&\quad Z_3^0 = \{1,2,3,4\} \\
Y_1 &= Z_3^0 = \{1,2,3,4\}
\end{aligned}$$

After one iteration, the greatest fixpoint is converged, and the least fixpoint is computed during that iteration. If a further iteration is required for the greatest fixpoint, the nested least fixpoint resets to the empty set and is computed again with the updated Y relation.

It is important to note that while all examples throughout this chapter have utilized undirected graphs, the concepts can also be applied to directed graphs without making any adjustments.

Chapter 4

Graph Neural Networks

Graph neural networks (GNNs) are vital for comprehension in subsequent chapters. There are many graph neural network classes, but we will focus on the most basic class that uses message passing. Before diving into GNNs, we will briefly explain the difference between 'traditional' neural networks and graph neural networks.

4.1 Motivation for GNNs

The primary motivation for developing GNNs stems from the limitations of traditional neural networks when dealing with data inherently structured graphs. Traditional neural networks excel at processing data that can be encoded as a grid structure, like images and sequences. Nonetheless, they struggle with relational data like knowledge graphs, social networks, and molecule structures. Here, the data lies in individual entities (nodes) but also in their connections or relations with other entities. This goes against traditional neural networks' core assumption that entities are independent of each other. This comparison is depicted in Figure 4.1. Data stored in the nodes are typically numerical values known as features and can be represented as a one-hot encoding. We can intuitively observe these values as colors applied to nodes with that specific value. Assigning features to nodes in a graph is analogous to creating a feature map on the graph [Gro21].

One can think of incorporating the input graph structure into the connections of neurons between layers in a traditional neural network. For each node and feature, there would be one neuron representing it in each layer, essentially unfolding the GNN into a traditional neural network. For example, the information from the adjacency matrix must be integrated within the weights between the neurons. Still, the neural network would only work on that specific input, a graph structure, and be complex and inefficient. These limitations are solved when using a GNN by the concept of message passing.

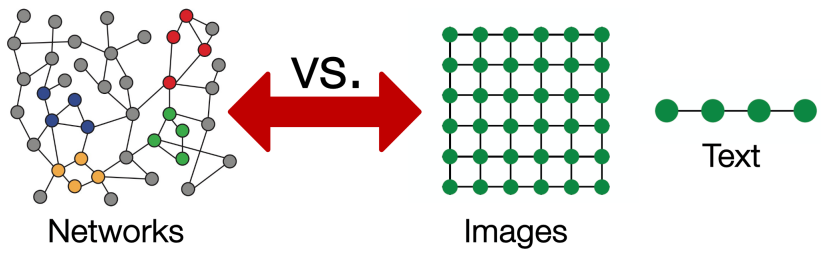

Figure 4.1: Relational data versus grids and simple sequences [Les23].

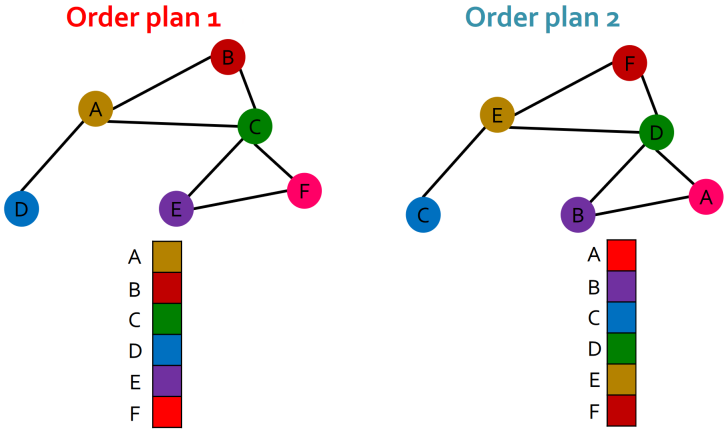

Figure 4.2: Two order plans on the same graph structure [Les23].

GNNs share some similarities with traditional neural networks in their layered architecture and basic processing principles, such as utilizing an activation function to transform the input data and pass the output to the next layer. However, they also possess a unique toolkit tailored to handle the complexities of graph-structured data.

4.2 Inner workings of GNNs

Graph neural networks (GNNs) deviate from traditional neural networks in their architectural design. However, understanding permutation equivariance is crucial before exploring the core building blocks of GNNs, namely local and global pooling layers.

4.2.1 Permutation equivariance

Graph structures do not have a canonical order of the nodes, which means that the order in which the nodes are listed alters graph properties like the adjacency matrix. Although the same graph is used, it can result in different order plans; see Figure 4.2. A GNN processes information from a graph to learn patterns or make predictions. The term 'permutation equivariant' can be explained by using the term permutation invariant as a stepping stone. A function is permutation invariant if it permutes the input and the output stays the same. An example is mapping a graph to a vector, like the molecular structure of H_2O, into the text H_2O. The order of atoms, whether we consider hydrogen or oxygen first, should not affect its properties.

Now, let us consider a function that is permutation equivariant. If such a function permutes the input, it will permute the output accordingly. Define G_1 as the graph of order plan 1 in Figure 4.2 and G_2 as the graph of order plan 2. An example here is mapping the G_1 to its colored version by defining a color and assigning node labels. The other graph G_2 can be defined using a permutation p on G_1 like follows $G_2 = p(G_1)$ where p will map $A \to E$, $B \to F$ etcetera, see Figure 4.3. A function f is permutation equivariant if $f(G_2) = p(f(G_1))$ using the permutation we just defined.

Thus, permutation equivariance ensures that the network's output does not change when the graph is permuted. This can be equivalently stated as achieving invariance under graph isomorphism. The graph and node representations should be the same for order plan 1 and order plan 2. GNNs consist of multiple permutation equivariant or invariant layers. Traditional neural network architectures are not permutation equivariant or invariant because switching the order of the input leads to different outputs. This also explains why incorporating the input graph structure into the connections of a traditional neural network does not work. The order of the nodes should not influence the architecture and, more importantly, the training phase of the

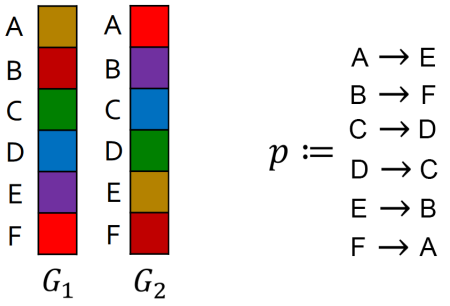

Figure 4.3: Visualization of permutation p for graphs G_1 and G_2.

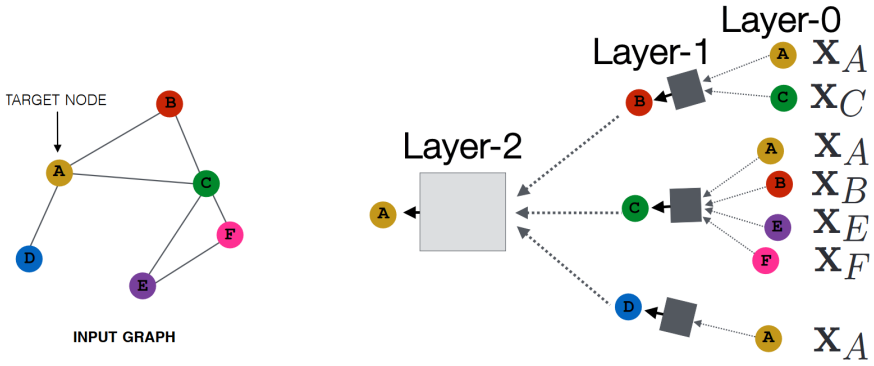

Figure 4.4: Message passing increases the receptive field of node A [Les23].

neural network. For example, assigning consistent IDs to nodes across different graphs in the training dataset could lead to the network exploiting this pattern instead of focusing on the inherent node features it aims to learn. Building upon the limitations outlined in section 4.1 and the reasoning presented here, it becomes evident that GNNs are an essential tool for effectively processing graph data.

4.2.2 Local pooling layers

The key design element of local pooling layers is message passing, where a node in the input graph iteratively updates by exchanging information with its neighbors in the input graph. We already mentioned the presence of many GNN architectures. The difference between them lies in implementing their message passing algorithms, like recursive and convolutional approaches. Message passing on the node's neighbors will increase the receptive field beyond its immediate neighbors. The edges of the input graph define the neighbors from whom we receive a message and to whom we send a message. This concept is depicted in Figure 4.4.

The number of local pooling layers defines the receptive field of a node in the graph. Figure 4.4 visualizes what three layers do for the receptive field of node A. At the start in layer-0, every node in the input graph only knows its features. Furthermore, in layer-1, we see that the receptive field of node B is increased to its neighbors A and C. Lastly, in layer-2, we can see that node A receives a message from its neighbors B, C, and D, who each know the features of their neighbors. Hence, node A's perceptive field contains nodes B, C, D, E, and F.

Note that we can visualize this for every node in the input graph for any arbitrary number of layers. At layer-k, a node of the input graph receives information from nodes that are k hops away.

The boxes are the only thing still undiscussed from Figure 4.4. What is in the box, and what do they represent? The box can be split into two parts. In the first part, all the messages from

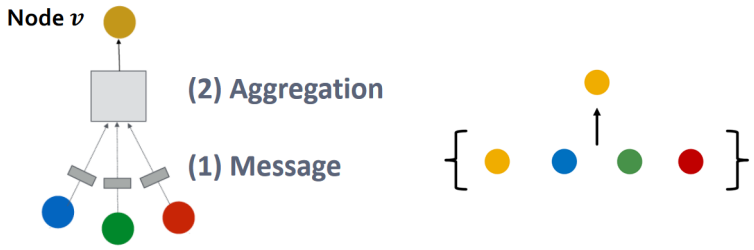

Figure 4.5: Aggregate and combine the feature vector of node v [Les23].

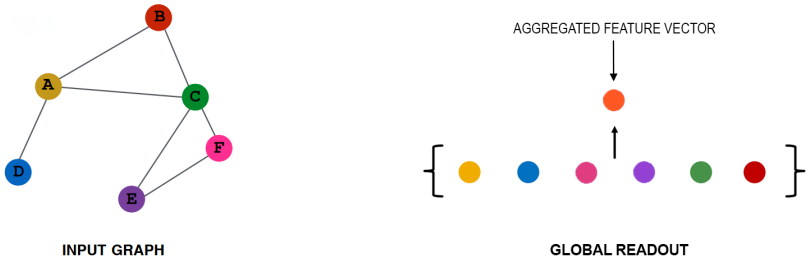

Figure 4.6: Global readout of the feature vectors of all the nodes in the graph.

the neighbors are collected by some arbitrary aggregation function. The second part utilizes an arbitrary combination function to combine the feature vector from the receiving node with the aggregated feature vectors from its neighbors with possibly a bias vector and an activation function. You could argue that this can be seen as a small neural network; see Figure 4.5.

Thus, each node of the input graph aggregates information from its immediate neighbors using message passing. These messages are then aggregated to summarize representation and combined with the node's own features. This creates an updated node representation capturing information from a wider perceptive field.

Note that Figure 4.4 has an undirected input graph. A node will send a message to its connected nodes on these graphs. A directed input graph works the same where a node A will send a message to node B only if a directed edge (B, A) exists. The message will be sent in the opposite direction of the edge because node B has to get the information from nodes it can reach by following the directed edge.

4.2.3 Global pooling layers

The last essential layer is the type of global pooling, also known as a (global) readout layer. Unlike the local pooling layers, where a node only aggregates information from its immediate neighbors, the global readout mechanism combines information from the entire input graph into a single vector representation using an arbitrary combination function, see Figure 4.6. In other words, this layer acts as a compression mechanism to capture the global features of the whole input graph, which a local pooling layer lacks. Notice that no message passing is involved here because the global readout directly operates on the representations of all nodes in the input graph.

4.3 Applications of GNNs

GNNs are used on relational data represented as a graph, so the various outputs they can produce are also graph-related. The output depends on the specific task and network architecture that the user chose. The most common types of GNN outputs are node classification, graph classification, and a newly generated graph.

Node classification will predict a class label for each node in the input graph, i.e., categorizing the nodes. These nodes can be anything from persons in a social network to research papers in a citation network. Via message passing, each node gathers information to predict the label or classify the node based on that information.

Node classification has a wide range of applications. Here are a few examples:

- Social network analysis: Classify persons based on their interests or demographics.
- Research papers citation network: Classify the paper into its specific research topic (e.g., machine learning, databases, etc.).
- Fraud detection: Identify a fraudulent person in a financial network using the transactions that are the edges of the graph.
- Recommendation system on social media: Recommend user content based on their connections on the platform.

Unlike node classification, which focuses on labeling individual points within a graph, graph classification aims to categorize the entire graph. It analyses the graph structure, meaning the connections between the nodes and the node properties, to predict the overall class. This is achieved through message passing and global pooling layers to aggregate information. Some examples are:

- Toxicity of molecules: Classify molecules as toxic or non-toxic based on their biological structure or properties.
- Document classification: A document can be classified based on the relationships between words or phrases in natural language processing (NLP).

Some GNN architectures are designed to generate new graphs based on the learned patterns or relations from the training data. This can be used for:

- Protein interaction: Analyze protein connections in organism A, on which the GNN is trained, to understand how similar proteins in organism B might interact.
- Chemistry: Generate lifelike molecules with desired properties and discover new chemical structures.

4.4 Classes of GNNs

While GNNs offer a powerful toolkit for various graph-centric tasks, this thesis will delve specifically into the domain of node classification. Here, our focus will lie on boolean node classifiers, where the GNN outputs a binary label ($true/false$ or 1/0) for each node in the input graph. The specific classes of GNNs relevant to this thesis are the aggregate-combine GNN (AC-GNN) and its extension, the aggregate-combine-readout GNN (ACR-GNN), which we will discuss in more depth.

4.4.1 AC-GNNs

A GNN takes a graph $G = (V, E)$ as input where each vertex $v \in V$ has a feature vector x_v. This feature vector can be interpreted as a finite list of colors where each vertex v can have multiple colors. The feature vector is a one-hot encoding of node colors. For directed graphs,

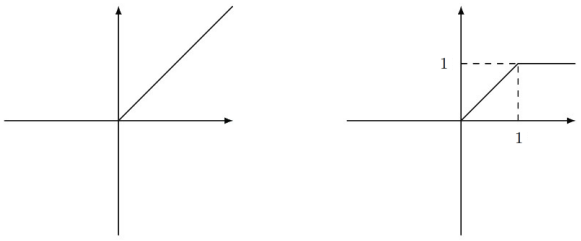

Figure 4.7: Comparison of activation functions ReLU and truncated ReLU.

the neighborhood $\mathcal{N}_G(v)$ of a node $v \in V$ is the set of $\{u \mid \{v, u\} \in E\}$. This is the set of nodes with an incoming edge from node v. For undirected graphs, the neighborhood $\mathcal{N}_G(v)$ of a node $v \in V$ is the set of $\{u \mid (\{v, u\} \in E) \vee (\{u, v\} \in E)\}$. In other words, the set of nodes that have an edge to node v.

An aggregate-combine GNN has multiple layers, say L layers, and can be denoted by $\mathcal{A} = (\{\text{AGG}^{(i)}\}_{i=1}^{L}, \{\text{COM}^{(i)}\}_{i=1}^{L}, \text{CLS})$. The tuple consists of three elements:

1. $\{\text{AGG}^{(i)}\}_{i=1}^{L}$: the set of aggregation functions, for each layer i, that aggregates the feature vectors of the neighbors into a multiset.

2. $\{\text{COM}^{(i)}\}_{i=1}^{L}$: the set of combination functions, for each layer i, that combines the feature vector with the aggregated multiset with additionally some weights and biases.

3. CLS: the boolean classification that extracts a particular component from each feature vector x_v to classify that node.

Using the recursive formula 4.1, an AC-GNN computes the feature vectors $x_v^{(i)}$ for every node v of graph G. Each node v starts with the initial feature vector $x_v^{(0)}$.

$$x_v^{(i)} = \text{COM}^{(i)}\left(x_v^{(i-1)}, \text{AGG}^{(i)}(\{\!\{x_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\})\right), \text{ for } i = 1, \ldots, L. \tag{4.1}$$

Our definition of AC-GNNs so far does not specify the aggregate, combination, and classification functions. Using different functions results in various classes of GNNs. The most straightforward choice is to take the sum of the feature vectors as the aggregation function and Equation 4.2 for the combination function. Where the feature vectors $x_v^{(i)}$ are row vectors, f is a non-linearity function like ReLU, $C^{(i)}$ and $A^{(i)}$ are matrices, and $b^{(i)}$ is a bias vector.

$$\text{COM}^{(i)}(x_1, x_2) = f(x_1 C^{(i)} + x_2 A^{(i)} + b^{(i)}). \tag{4.2}$$

Barceló et al. call an AC-GNN simple and homogeneous if the previously discussed functions are used for the aggregation and combination, and these functions are the same across the layers [BKM$^+$20]. Furthermore, they use the truncated or clipped ReLU activation function defined by $\sigma(x) = \max(0, \min(1, x))$, see Figure 4.7. The truncated ReLU is like ReLU, a threshold operation where any input value below zero is mapped to zero. The difference lies in the mapping for positive input values where truncated ReLU has a clipping ceiling where any input value larger than that value is mapped to that ceiling; in our case, that ceiling is 1.

For simple homogeneous AC-GNNs, Equation 4.1 would look like:

$$x_v^{(i)} = \mathrm{COM}\left(x_v^{(i-1)}, \mathrm{AGG}(\{\!\{x_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\})\right)$$
$$= \sigma\left(x_v^{(i-1)}C + \sum_{u \in \mathcal{N}_G(v)} x_u^{(i-1)}A + b\right).$$

4.4.2 ACR-GNNs

In subsection 4.2.3, the principle of a global readout is explained, but this mechanism is not implemented in an AC-GNN. These aggregate-combine GNNs can only gather local information for their classification. The aggregate-combine-readout GNN (ACR-GNN) is an extension of the AC-GNN that includes global feature computation on each layer. Readout functions $\{\mathrm{READ}^{(i)}\}_{i=1}^{L}$ are specified for each layer, which aggregate the current feature vectors $x_v^{(i)}$ of all nodes v of graph G. Thus, an aggregate-combine-readout GNN can be denoted by $\mathcal{A} = (\{\mathrm{AGG}^{(i)}\}_{i=1}^{L}, \{\mathrm{COM}^{(i)}\}_{i=1}^{L}, \{\mathrm{READ}^{(i)}\}_{i=1}^{L}, \mathrm{CLS})$. Each feature vector $x_v^{(i)}$ can be computed by extending Equation 4.1 to:

$$x_v^{(i)} = \mathrm{COM}^{(i)}\left(x_v^{(i-1)}, \mathrm{AGG}^{(i)}(\{\!\{x_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\}), \mathrm{READ}^{(i)}(\{\!\{x_u^{(i-1)} \mid u \in G\}\!\})\right). \quad (4.3)$$

The readout function is a global operation, so it is an aggregation function for all nodes in graph G. So, the feature vector $x_v^{(i)}$ combines its feature vector, the aggregated feature vectors of its neighbors, and the aggregated feature vectors of all nodes. For this class of GNNs, Barceló et al. also introduced the term a simple ACR-GNN, which uses the sum as the readout function. Equation 4.2 can be generalized by adding another feature vector:

$$\mathrm{COM}^{(i)}(x_1, x_2, x_3) = f(x_1C^{(i)} + x_2A^{(i)} + x_3R^{(i)} + b^{(i)}). \quad (4.4)$$

4.4.3 Other classes

The two classes of GNNs we discussed, the AC-GNN and ACR-GNN, are the most basic but fundamental classes. They possess the essential layers that differentiate these neural networks from traditional ones. Like traditional neural networks, which can be extended to a convolutional neural network, GNNs can also be extended to those classes. Other GNN classes are graph convolutional networks [KW17], graph attention networks [VCC$^+$18], and so on.

Chapter 5

The Expressive Power of AC-GNNs

Building on the concepts introduced in chapter 3 on graded modal logics and chapter 4 on the application of graph neural networks for node classification, this chapter explores the connection between these two domains. Specifically, it examines how node classifiers in graded modal logic, functions classifying nodes in graphs as $true$ or $false$, can be used to formalize a class of GNNs, namely the simple homogeneous AC-GNN, which is explained in subsection 4.4.1. This concept is the state of the art by Barceló et al. [BKM$^+$20].

5.1 Introduction

Let's delve into how graded modal logic relates to the inner workings of aggregate-combine GNNs. Barceló et al. showed that graded modal logic is the 'largest' class of logical classifiers captured by AC-GNNs. In other words, the only first-order formulas AC-GNNs can learn accurately are those in graded modal logic. The alignment between graded modal logic, a well-established field, and graph neural networks, a relatively recent development, is particularly noteworthy.

Proposition 5.1.1. *Each graded modal logic classifier is captured by a simple homogeneous AC-GNN.*

Before introducing the concept and construction of Proposition 5.1.1, it is necessary to establish a formal definition of what it means for a logic classifier to be captured. To this end, we present the following definition.

Definition 5.1.1. *A GNN classifier $\mathcal{A}$ captures a logical classifier φ if for every graph G and node v in G, it holds that $\mathcal{A}(G, v) = true$ if and only if $(G, v) \models \varphi$.*

In other words, this means that a GNN classifier and a logical classifier coincide over every node in every possible input graph. The technique used in the proof of Proposition 5.1.1 is the inspiration of this thesis; an exact copy of that proof is present in Appendix A. Note that the state of the art uses the first-order semantics for graded modal logic. The following sections will focus on the underlying concept of the construction and the construction itself using the modal logic semantics.

5.2 Key idea

The key idea is to use the vectors' dimensions, used by the AC-GNN $\mathcal{A}_\varphi$ to label nodes as subformulas of the captured classifier φ, a graded modal logic formula. A feature i of a node is 1 if the

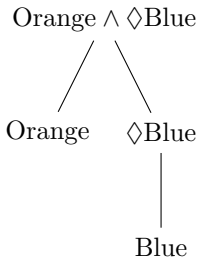

Figure 5.1: Abstract Syntax Tree of graded modal logic formula α.

node satisfies the subformula φ_i corresponding to that feature. Let $sub(\varphi) = (\varphi_1, \varphi_2, \ldots, \varphi_L)$ be an enumeration of the subformulas of φ where each φ_i is a subformula of φ_j if $i \leq j$. Thus, $\mathcal{A}_\varphi$ with L layers will have feature vectors in $\mathbb{R}^L$ where each component represents a different subformula in $sub(\varphi)$. Note that $\varphi = \varphi_L$ is the feature vector's last component which will be evaluated after L layers. Each node gets value 1 at the last component if and only if the node satisfies φ. Based on the value of that component, the classification function CLS will classify that node.

We interpreted this representation of subformulas as a list of nodes from the parse tree of φ because the proof does not provide a specific way to obtain this list. For example, take $\alpha = \text{Orange} \wedge \Diamond\text{Blue}$. The enumeration of subformulas will look like:

$$sub(\alpha) = (\text{Orange}, \text{Blue}, \Diamond\text{Blue}, \text{Orange} \wedge \Diamond\text{Blue})\,.$$

This sequence can be obtained by traversing the formula's abstract syntax tree (AST) in postorder traversal; see Figure 5.1. Postorder traversal is necessary because of our definition of $sub(\alpha)$, which defines that each φ_i is a subformula of φ_j if $i \leq j$. Regarding an AST, all the children, which are the left and right subtree of an AST-node v, must be evaluated before v itself. Thus, the postorder traversal algorithm must be used.

5.3 Construction

Having explained the idea behind the construction of the feature vectors used by the simple homogeneous AC-GNN $\mathcal{A}_\varphi$, we will dive into the definitions of the aggregation and combine functions:

$$\begin{aligned} \text{AGG}(X) &= \sum_{x \in X} x, \\ \text{COM}(x, y) &= \sigma(xC + yA + b), \end{aligned}$$

where σ is the truncated ReLU activation function, $A, C \in \mathbb{R}^{L \times L}$, and $b \in \mathbb{R}^L$. Given a colored graph $G = (V, E)$, each node v has an initial feature vector $x_v^{(0)} = (x_1, x_2, \ldots, x_L)$ such that $(x_v^{(0)})_l = 1$ if subformula φ_l is the initial color assigned to v, and $(x_v^{(0)})_l = 0$ otherwise. Remark that a node can have multiple initial colors. From subsection 4.4.1, we know that feature vectors are computed as follows:

$$\begin{aligned} x_v^{(i)} &= \text{COM}\left(x_v^{(i-1)}, \text{AGG}(\{\!\{x_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\})\right) \\ &= \sigma\left(x_v^{(i-1)} C + \sum_{u \in \mathcal{N}_G(v)} x_u^{(i-1)} A + b\right). \end{aligned} \tag{5.1}$$

Thus, the value of the l-th component of $x_v^{(i)}$ can be denoted as:

$$(x_v^{(i)})_l = \sigma\left(\sum_{k=1}^{L}(x_v^{(i-1)})_k C_{kl} + \sum_{u\in\mathcal{N}_G(v)}\sum_{k=1}^{L}(x_u^{(i-1)})_k A_{kl} + b_l\right). \tag{5.2}$$

The entries of the l-th columns of A, C, and b are dependent on the subformula φ_l which will be discussed as separate cases for each essential operator in graded modal logic as follows:

Case 0. if $\varphi_l = \text{Col}$ with Col one of the node colors, then $C_{ll} = 1$,

Case 1. if $\varphi_l = \varphi_j \wedge \varphi_k$ then $C_{jl} = C_{kl} = 1$ and $b_l = -1$,

Case 2. if $\varphi_l = \neg\varphi_k$ then $C_{kl} = -1$ and $b_l = 1$,

Case 3. if $\varphi_l = \Diamond^N\varphi_k$ then $A_{kl} = 1$ and $b_l = -N + 1$,

and all other values in the l-th columns of A, C and b are 0.

Note that matrix A is responsible for the message passing of the neighbors' features because it is only used in *Case 3* and multiplies with the feature vectors of the neighbors in Equation 5.2. Matrix C is responsible for passing information about properties or subformulas about the node itself. For example, take *Case 0* where $C_{ll} = 1$, which copies the property from the $i-1$-th layer to the next layer. Vector b, which acts as a bias vector, ensures that the correct input will be given to the truncated ReLU activation function. For example, take *Case 1* where the subformula φ_j is $true$ and φ_k is $false$. Without the bias, the truncated ReLU would output 1 because it gets 1 as input. Thus, the bias ensures the input will be 0, resulting in 0 as the output of the truncated ReLU.

5.4 Example

To illustrate this concept further, let's consider an example that will be given for the graded modal logic formula $\alpha = \text{Orange} \wedge \Diamond\text{Blue}$, which we already used throughout this chapter. Recall that the enumeration of the subformulas looks like:

$$sub(\alpha) = (\text{Orange}, \text{Blue}, \Diamond\text{Blue}, \text{Orange} \wedge \Diamond\text{Blue}).$$

The simple homogeneous AC-GNN $\mathcal{A}_\alpha$ will take Figure 5.2 as input graph $G = (V, E)$. It is easy to see that after evaluation, the nodes 2 and 4 will be classified as $true$ and the other nodes as $false$. Having a formula and input graph, the only thing left to define is the AC-GNN $\mathcal{A}_\alpha$. The GNN will have four layers because the enumeration $sub(\alpha)$ has four items or subformulas. Also, each feature vector will have a length of four because of this. The initial or base colors are Orange and Blue, and the feature map for Figure 5.2 will look like:

$$FM^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \tag{5.3}$$

where each row i represents the feature vector for node i and the columns represent the enumeration of $sub(\alpha)$. For example, the first row representing node 1 has a 1 in column two, which represents the color Blue, and a 0 in column one because the node is not Orange.

Next, the construction of the matrices C, A and vector b, we get:

$$C = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \; A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \; b = \begin{bmatrix} 0 & 0 & 0 & -1 \end{bmatrix}.$$

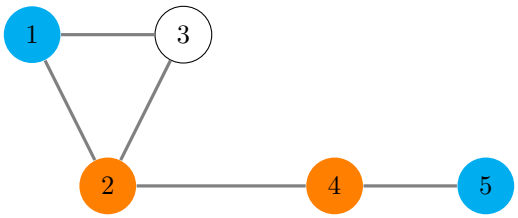

Figure 5.2: Input graph $G = (V, E)$ for evaluating graded modal logic formula α.

We will explain this using the four cases for each subformula.

Subformula 1: which is Orange, an initial color, so it falls into *Case 0*.

Subformula 2: which is Blue, an initial color, so it falls into *Case 0*.

Subformula 3: which is the $\Diamond$ operator on subformula 2 with $N = 1$, so it falls into *Case 3*.

Subformula 4: which is the $\wedge$ operator on subformula 1 and 3, so it falls into *Case 1*.

Following the definition of the AC-GNN $\mathcal{A}_\alpha$, we will provide a detailed breakdown of the iterative computations occurring within each of the four layers. The feature map will be used for the iterative computations to compute all the feature vectors simultaneously. Note that the adjacency matrix of the input graph G will be used to get the neighbors of node v, denoted $\mathcal{N}_G(v)$. The adjacency matrix is symmetrical because the graph is undirected and looks like:

$$Adj = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Now every component of Equation 5.1 is defined, filling them in gives:

$$\begin{aligned} FM^{(i)} = \text{truncated\_ReLU}(&FM^{(i-1)} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\ &+ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} FM^{(i-1)} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & -1 \end{bmatrix}), \end{aligned} \tag{5.4}$$

where we use the feature map instead of the individual feature vectors. This way, the adjacency matrix can be used for message passing, making it easier to comprehend. The bias vector b will be added to each feature map row, representing a feature vector.

Starting with the initial feature map, see Equation 5.3 and iteratively compute the following feature map using Equation 5.4. We get the following feature map after iteration one:

$$FM^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

because nodes 2,3 and 4 have a Blue neighbor. Node 1 for nodes 2 and 3, and node 5 for node 4. The first two columns are copied because they represent the initial colors. The fourth column,

which represents the $\wedge$ operator, is zero for every node because no node has a one in the first and third columns in the feature map that was the input. In this case, the initial feature map. The next iteration will give:

$$FM^{(2)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

In the second iteration, every node gained information about its neighbors (one hop away), so the third feature or column is computed. Resulting in an output for the fourth feature, which is *true* for nodes 2 and 4. Iterations three and four will no longer change the feature map.

5.5 Discussion

Note that an iteration will compute each feature in the feature vector, even if it cannot be computed because the subformulas have not been computed yet. For example, the last feature, which represents the conjunction between features one and three, cannot be computed in iteration one because that is the first iteration where feature three can be computed. The first iteration where the last feature can be computed is iteration two. Furthermore, the current configuration may include an excessive number of layers. In our example, layers three and four appear to be redundant, as their computations likely yield no significant changes to the feature map since the necessary information is already captured in layer two. This results in unnecessary computations that cannot be avoided because of the construction of Barceló et al.

The activation function truncated ReLU is very useful for *Case 3*, representing the diamond operator. By definition, the operation evaluates to *true* if there are N or more neighbors that satisfy a subformula φ. Truncated ReLU ensures that every number $n \geq N$ will be transformed to 1, which means *true*. The added threshold in the truncated ReLU is unnecessary for *Case 1* and *Case 2* because the bias vector handles all the required transformations. A regular ReLU would work for these cases. *Case 0* does not need a bias or an activation function because they are always copied.

Thus, *Case 3* is the only one that needs the truncated ReLU to work correctly. A solution exists that only uses regular ReLU functions. A two-round ReLU and a slight change in definition can simulate the truncated version. Recall that the truncated ReLU is defined as $\max(0, \min(1, x))$, and the regular Relu is defined as $\max(0, x)$. We can rewrite the truncated ReLU using the following mathematical equivalence as a two-round ReLU.

$$\min(a, b) = a - \max(0, a - b).$$

For *Case 3* ($\varphi_l = \Diamond^N \varphi_k$), we get:

$$\begin{aligned} \sigma(-N + 1 + m) &= \max(0, \min(1, -N + 1 + m)) \\ &= \max(0, 1 - \max(0, 1 + N - 1 - m)) \\ &= \max(0, 1 - \max(0, N - m)) \\ &= \max(0, 1 - \mathrm{ReLU}(N - m)) \\ &= \mathrm{ReLU}(1 - \mathrm{ReLU}(N - m)), \end{aligned}$$

where σ is the truncated ReLU activation, and $m = |\{u \mid u \in \mathcal{N}(v) \text{ and } u \models \varphi_k\}|$ for every node v in G. Note that definition of *Case 3* which currently is:

Case 3. if $\varphi_l = \Diamond^N \varphi_k$ then $A_{kl} = 1$ and $b_l = -N + 1$,

must be redefined to:

Case 3. if $\varphi_l = \Diamond^N \varphi_k$ then $A_{kl} = -1$ and $b_l = N$.

This way, matrix A will take care of the $-m$, and the bias will take care of N. Thus for this case, the activation function is defined as:

$$\sigma(x) = \mathrm{ReLU}(1 - \mathrm{ReLU}(x)).$$

Having established the ability to simulate truncated ReLU with ReLU, this work prioritizes the truncated version due to its compactness and ease of use. The issue of unnecessary computations for certain features and the number of layers will be addressed in chapter 7, where a dedicated solution will be presented.

Chapter 6

Recurrent Neural Networks

Recurrent neural networks (RNNs) are a powerful class of neural networks that contain a cycle within their network connections, meaning that the value of some unit, e.g., a past output, can be utilized as an input. This makes them adept at handling sequential data. While powerful, these networks are complex to reason about and train. While various RNN architectures exist, we will explore the fundamental class that exemplifies this unique cyclic structure and compare it with 'traditional' neural networks. To conclude this chapter, we define a recurrent graph neural network.

6.1 Motivation for RNNs

While powerful for various tasks, traditional neural networks struggle with sequential data. This data, where the order of elements is significant, is ubiquitous in real-world applications like language processing and time series forecasting. This goes against the assumption of traditional neural networks that entities are independent of each other. Thus, the learned features will and can not be shared across different positions of text in terms of language processing. Another problem is the input and output size, which is fixed for traditional networks. Sequential data usually have a different length, resulting in an input size dependent on the sequential input data. The same holds for the output size.

While intuitively appealing, constructing a traditional neural network to mimic a recurrent neural network has inherent limitations. One approach might involve chaining multiple smaller networks together, where each subsequent network receives the output of the previous one. Still, the neural network would only work for sequential data with the same size. Furthermore, the output size is also fixed. This construction also struggles with long sequences as the number of layers and computational complexity rapidly grows. Additionally, it is possible to lose information over many layers. All these limitations are solved when using a RNN.

6.2 Inner workings of RNNs

The most basic class, vanilla RNNs, will be used to explain the key difference between a traditional neural network and a recurrent neural network. Figure 6.1 shows this difference where a RNN has a feedback loop. The construction mentioned in the previous section is exactly how a vanilla RNN works. A RNN can be unrolled, making it easier to visualize and analyze the process. Unrolling refers to transforming a recurrent structure (like a loop) into an acyclic one (without loops) by explicitly replicating the computations performed at each iteration. In Figure 6.1, the unrolled RNN consists of three layers or iterations that each output something. Suppose the RNN wants to predict the weather based on the weather over the past few days. Having data from yesterday, a traditional neural network can predict today's weather but not

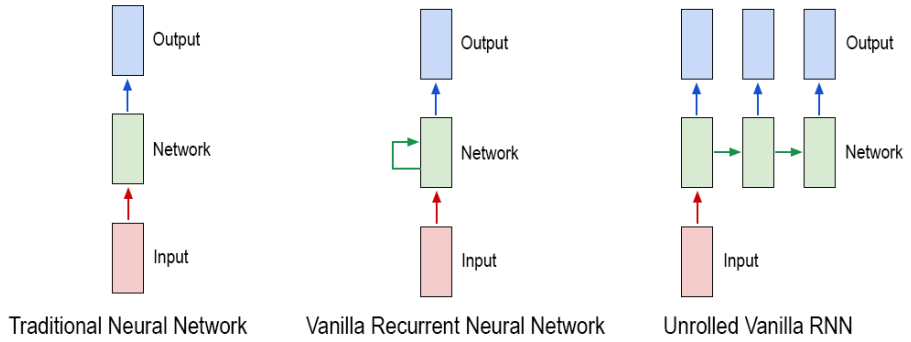

Figure 6.1: Comparison between a traditional neural network and a (unrolled) vanilla recurrent neural network.

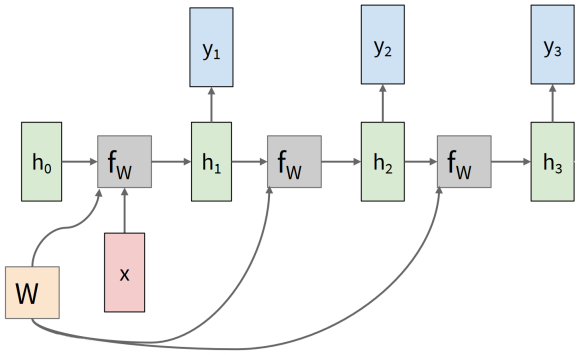

Figure 6.2: Computational graph of a three-layer RNN [LA24].

the weather for tomorrow or the day after tomorrow, and so on. If we want to predict the weather for the day after tomorrow, a three-layer RNN is necessary. The first output will be today's weather, the second will be tomorrow's weather, and the third will be the weather of the day after tomorrow.

RNNs have an 'internal or hidden state' that is updated as a sequence is processed; in the case of the weather example, the hidden state can hold information about the weather of the past few days/iterations. A sequence of vectors x can be processed by applying a recurrent formula at every time step:

$$\begin{aligned} h_t &= f_W(h_{t-1}, x_t) \\ &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t), \end{aligned} \tag{6.1}$$

which is the formula to update the hidden state based on the old hidden state with the input vector at some time step. The output can be described as:

$$y_t = f_{W_{hy}}(h_t) = W_{hy}h_t,$$

where the network is represented by a function f, which gets as input the hidden state from this time step. One input suffices because the hidden state h_t combines the old state with the input; see Equation 6.1. A bias vector can be added to both equations depending on the definition. A more detailed version of the three-layer RNN from our example will be Figure 6.2.

Note that a RNN can be represented without being unrolled, thus only seeing one layer. So, each layer/iteration, which means a smaller neural network, uses the same network architecture, weights, and biases. This is depicted in Figure 6.2 where each box grey box uses the same function f_W and weight matrix W at every time step.

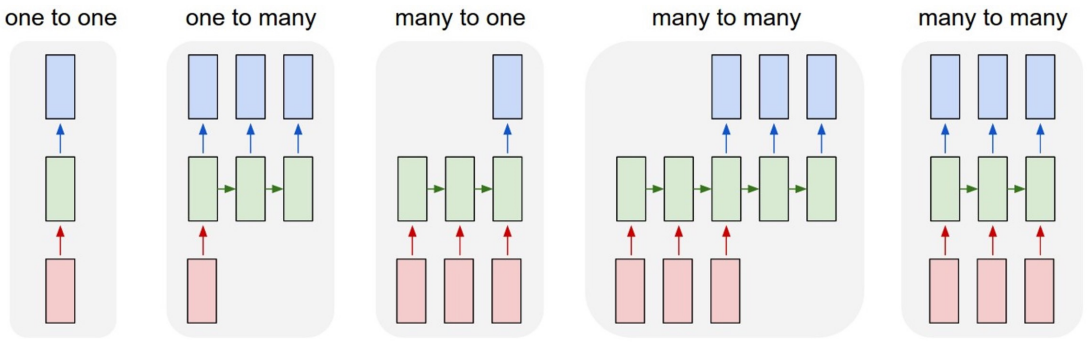

Figure 6.3: Different types of networks based on their number of inputs and outputs [LA24].

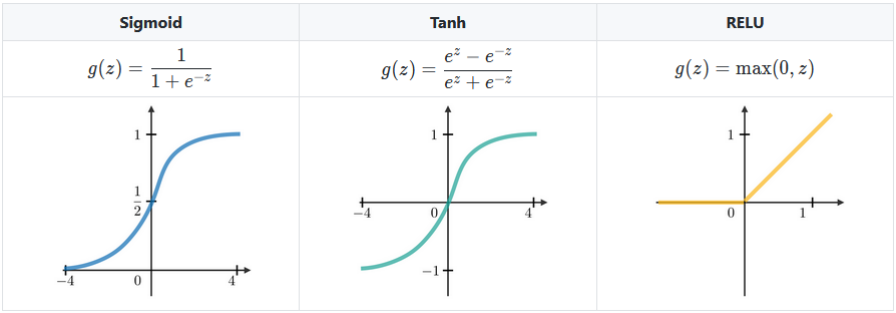

Figure 6.4: Most common activation functions used in RNN modules [AA19].

6.2.1 Types of RNNs

Many types of RNNs exist, and they differ in their number of inputs and outputs. The RNN depicted in Figure 6.2 is a one-to-many network. A traditional neural network can be interpreted as a one-to-one network; see Figure 6.3. An example of a many-to-one RNN is a network that classifies a sequence of video frames to one particular class. Staying in the context of video frames, the many-to-many RNN can be used for video captioning, where the number of inputs and outputs can differ. When the inputs and outputs are the same, an example is video classification on frame level.

Note that one can simulate a one-to-many using a many-to-many by only using the first input and setting the remaining inputs to zero. These types can also be concatenated to create a sequence-to-sequence model. Suppose we use a many-to-one RNN to encode the input sequence into a single vector, followed by a one-to-many RNN to produce the output sequence from a single input vector. This RNN can be seen as two smaller RNNs concatenated.

6.2.2 Vanishing gradient problem

In Equation 6.1, the hyperbolic tangent activation function was used, but other common activation functions used in RNNs are sigmoid and ReLU, see Figure 6.4. Training involves backpropagation, a technique that adjusts the weights within the network based on the difference between the predicted and usual outputs. The gradients representing these adjustments are calculated and used to update the weights in each layer.

Regarding RNNs, information from the beginning of a sequence needs to flow through many layers/iterations to the final output. During backpropagation, these gradients are multiplied at each layer. This multiplicative gradient can be exponentially decreasing with respect to the number of layers and weights. When the gradients are multiplied back through many layers with

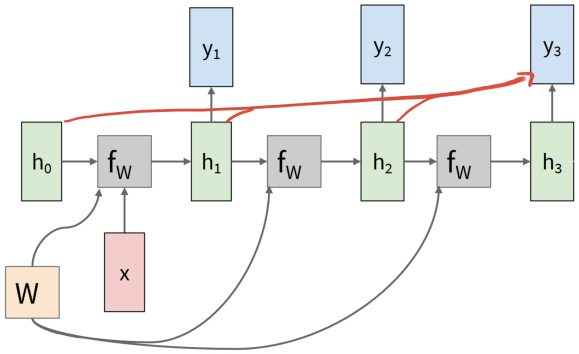

Figure 6.5: High-level idea of LSTMs, an extension of vanilla RNNs.

an activation function having values less than one, especially if the weights are also small, the gradient values will shrink layer over layer. This makes it difficult for the network to learn from distant parts of the sequence because the faraway states do not contribute to the parameters' gradient computation. On the other hand, we have gradients exponentially exploding, the exploding gradient problem, which is attributed to large values in matrix multiplication, causing the loss function to oscillate or diverge rather than converge to a minimum. [Che16].

The vanishing and exploding gradient problem is the main weakness of RNNs, but it can be resolved using several techniques like gradient clipping. This technique limits the gradient by setting a maximum value to prevent it from exploding during backpropagation. However, gradient clipping introduces an additional parameter that needs to be carefully selected. If the threshold is too high, it may fail to prevent the instability caused by exploding gradients. Conversely, if the threshold is too low, it may hinder the learnability of the network by restricting the gradient updates too much. Thus, using gradient clipping is not without limitations.

Another solution is using different activation functions apart from the hyperbolic tangent. Tanh has values between -1 and 1. If the inputs are centered around zero, the tanh function's output will also be close to zero. These near-zero values cause the gradients to shrink during backpropagation. A different activation function like ReLU has other characteristics, but no activation function can completely eliminate the vanishing gradient problem.

6.3 Long short-term memory (LSTM)

While gradient clipping can solve the exploding gradient problem, no solution is given for the vanishing gradient problem. Vanilla RNNs can not solve this phenomenon, so the solution is to change to a different RNN architecture called long short-term memory (LSTM). These models were proposed to handle gradient vanishing and exploding problems. LSTM models incorporate the correlation between x_t, h_{t-1}, and h_t using a memory unit instead of a simple tanh function. Furthermore, LSTM models can handle long sequences better than vanilla RNNs [Che16].

Before delving into the specifics of LSTMs, we start with a basic explanation of how LSTMs differ from vanilla RNNs. Figure 6.5 shows the high-level idea of LSTMs where there are two separate paths to make predictions about y_3. The outputs y_1 and y_2 are unimportant for this example but are computed similarly. The path already existed for vanilla RNNs is used for short-term memories, and the added red path is for long-term memories. Unrolling a vanilla RNN was simple because the only thing connecting the recurrent layers (units) was the feedback loop. LSTMs have a much more complicated unit because they use two feedback paths.

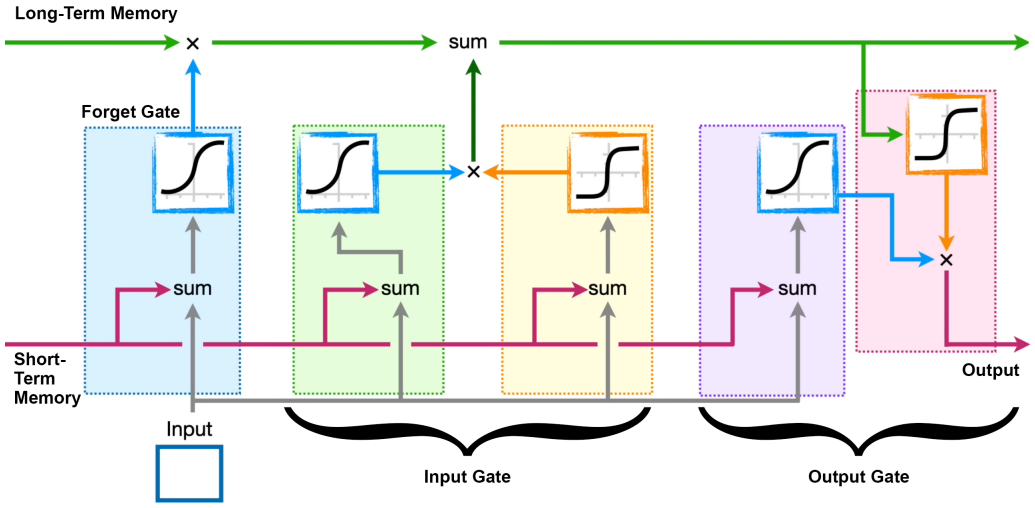

Figure 6.6: Unit structure of LSTM, including three gates: input gate, forget gate and output gate [Sta22].

6.3.1 Informal explanation

An excellent introduction to LSTMs is given in a YouTube video by Josh Starmer [Sta22], where he uses Figure 6.6 throughout the video. The unit structure with three gates is intuitively visualized and is a good stepping stone when trying to understand LSTMs.

LSTM units can be separated into multiple gates, each with a responsibility. In Figure 6.6, you can see that the LSTM unit has an input for that time step but also gets the short-term and long-term memory as inputs, the purple and green arrows, respectively. No weights or biases can affect long-term memory. This prevents the vanishing or exploding gradient problem.

The first element within the LSTM architecture, visualized as a blue box, is the forget gate. This gate plays a crucial role in managing the long-term memory state of the network. It takes the network's short-term memory and the current input to the LSTM unit as input. Based on this combined information, the forget gate determines which elements of the previous long-term memory state are still relevant and which can be discarded. The forget gate employs a sigmoid activation function to map the weighted sum of its inputs and a bias term to a value between 0 and 1. The forget gate facilitates selective memory control within the LSTM, allowing the network to focus on the most relevant past information while discarding outdated or irrelevant elements.

The second element within the LSTM architecture is the input gate, visualized as two interconnected units: a green box and an orange box. We will start with the orange box, which is crucial in creating candidate memory for the long-term memory. This unit takes two inputs: the network's short-term memory and the current input. These inputs are combined and processed through a hyperbolic tangent activation function. The tanh function outputs values between -1 and 1, allowing the network to learn a diverse range of potential memory updates.

The green box operates the same as the blue box. It acts as a gating mechanism to determine the proportion of this candidate's memory that will be incorporated into the long-term memory. This gate is responsible for updating the long-term memory.

The third and last element in the LSTM unit is the output gate, visualized as two interconnected units: a purple box and a pink box. This gate plays a critical role in determining both the output of the current LSTM unit and the updated state of the network's short-term memory.

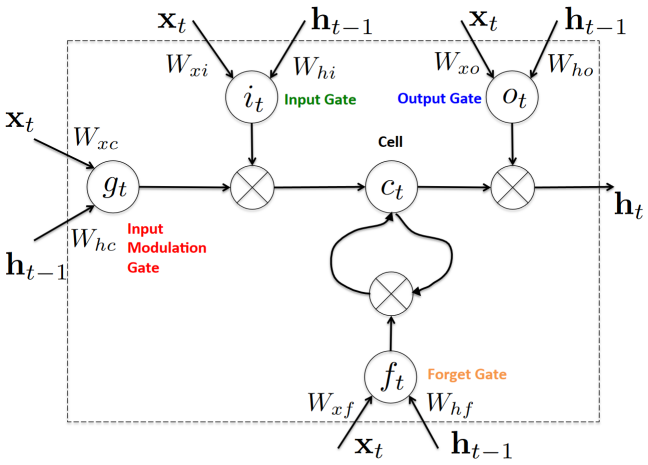

Figure 6.7: It is a unit structure of LSTM, including four gates: input modulation gate, input gate, forget gate, and output gate [Che16].

The pink box uses a hyperbolic tangent activation function on the updated long-term memory to create the candidate memory for the short-term memory. The purple box is executed next on this candidate memory, a gating mechanism like the blue and green boxes. The output is the new short-term memory and acts as the output if this LSTM unit is the last in the recurrence.

6.3.2 Formal explanation

Next, we will put the previous explanation into formal definitions and a more formal visualization, see Figure 6.7. The formal definition has an extra gate, the input modulation gate, which combines the input x_t and long-term memory h_{t-1}. The short-term memory is denoted as the memory cell c_t. The gates can be defined as follows:

$$
\begin{aligned}
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f),\\
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i),\\
g_t &= \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c),\\
c_t &= f_t \circ c_{t-1} + i_t \circ g_t,\\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o),\\
h_t &= o_t \circ \tanh(c_t), z_t = \text{softmax}(W_{hz}h_t + b_z),
\end{aligned}
$$

where f_t indicates the forget gate, i_t the input gate, o_t the output gate, and g_t the input modulation gate. h_t is the output of the LSTM unit, and z_t is used to make predictions by adding a linear model over the hidden state h_t and outputting the likelihood with the softmax activation function.

The LSTM unit structure in Figure 6.7 can also be unrolled like vanilla RNNs to understand the overall structure and error backtracking better. This is visualized in Figure 6.8.

6.4 RNNs in practice

Recurrent neural networks have revolutionized the field of deep learning by enabling effective modeling of sequential data. This capability makes them highly valuable for a wide range of real-world applications. Here are a few examples:

- Natural language processing (NLP): Tasks that require understanding the context of words within a sequence.

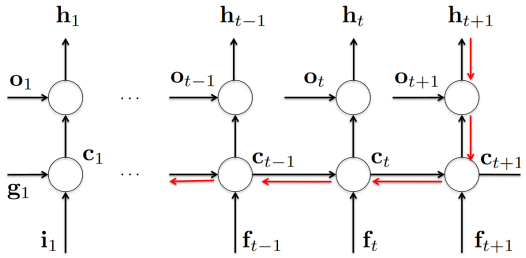

Figure 6.8: This graph unfolds the memory unit of LSTM to make it easy to understand error backpropagation [Che16].

- Speech recognition: Analyzing the sequence of sounds in speech. This powers automated transcription services and voice search functionalities.
- Time series forecasting: Learning patterns and trends within time series data such as stock prices or weather patterns.
- Video analysis: Processing video sequences for object tracking and video captioning.

Despite their strengths, RNNs face challenges such as the vanishing gradient problem, which can hinder their ability to learn long-term dependencies. However, improvements such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), similar to LSTMs, have been developed to address these limitations and enhance the capabilities of RNNs for processing long sequences. RNNs have become a cornerstone of deep learning for sequential data and paved the way for transformers. They rely on an attention mechanism that allows them to simultaneously attend to all sequence elements. This can be more efficient for specific tasks.

6.5 RecGNNs

This thesis focuses on recurrent graph neural networks (RecGNNs), which we will introduce next. We define a RecGNN as a multi-layer GNN that is iterated. It is a one-to-many network with only one input, the input graph. The output of an iteration functions as the input of the next iteration. When the values of the final layer in one iteration are the same as the values of that layer in the previous iteration, we say that the RecGNN has stabilized. Thus, message passing iterations are repeatedly applied until the node classifications become stable. This type of definition is very similar to the ones from the original GNN paper [SGT+09] and a follow-up [GM10], where they define a model that recursively applies a message passing layer until the updates of the feature vectors are smaller than a certain threshold.

Like vanilla RNNs and LSTMs, a RecGNN will use the same network architecture in each iteration, which means we can define a RecGNN like a regular GNN with an additional feedback loop. This type of network allows us to learn progressively complex representations of the nodes and, ultimately, the entire input graph. So, the number of message passing iterations is not fixed but dependent on the input graph.

Chapter 7

From Graded μ-calculus to RecGNNs

The remarkable work by Barceló et al. on the connection between graded modal logic (GML) and graph neural networks (GNNs) has spurred significant interest in the field, which is discussed in chapter 5. However, much remains unknown about the expressive power of GNNs, particularly recurrent variants. This chapter extends this inquiry by investigating the logical expressiveness of simple homogeneous recurrent graph neural networks (RecGNNs), defined in section 6.5. Because it extends the work by Barceló et al., an extension of GML must be used that incorporates recursion. This logic is graded μ-calculus, explained in section 3.3. We will show that these graded μ-calculus classifiers can be captured by RecGNNs through the construction of a compiler-like program that outputs the corresponding RecGNN architecture given a graded μ-calculus expression.

7.1 Key idea

Graded μ-calculus extends graded modal logic by adding fixpoint operators. This connection builds upon the foundational work by Barceló et al., making their research an excellent foundation for our exploration.

7.1.1 Optimizing the foundation

As discussed in section 5.5, an inefficiency exists in computing features that are not yet considered 'ready', and the number of layers may be excessive, as shown in the example.

Expanding the feature vectors

To avoid these inefficiencies, we will start by changing the construction proposed by Barceló et al. Initially, we will only consider graded modal logic expressions to optimize our foundation. The change lies within the feature vectors, which initially will only contain all the node colors and will be expanding throughout the neural network. Define the number of node colors κ; then, the initial feature vectors are defined in $\mathbb{R}^\kappa$. Each layer will copy the existing features to the new feature vector and only compute the feature of the subformula corresponding to that layer. Note that the features in the feature vectors resemble the subformulas defined in $sub(\varphi) = (\varphi_1, \varphi_2, \ldots, \varphi_L)$ but all node colors are defined as a subformula before defining logical operators like the conjunction or negation. Recall the example from section 5.4:

$$\alpha = \text{Orange} \wedge \Diamond \text{Blue}. \tag{7.1}$$

Also, recall that this GML has the following subformulas:

$$sub(\alpha) = (\text{Orange}, \text{Blue}, \Diamond\text{Blue}, \text{Orange} \wedge \Diamond\text{Blue})\,.$$

Note that every node color is defined before the operations for this small example. In this case, κ is defined as $\kappa = 2$; hence, the initial feature vectors will be defined in $\mathbb{R}^2$.

Mitigating Redundant Layers

Using the flexible feature vectors described above, the number of layers is no longer related to L, the number of subformulas in $sub(\varphi)$. With this construction, the number of layers is the number of logical operators present in $sub(\varphi)$; in other words, the number of subformulas that are not node colors. Thus, each layer corresponds to a subformula and cannot do any unnecessary computations for upcoming features because they don't exist yet due to our construction. It is trivial to see that this plan results in a GNN architecture with fewer layers than L. Continuing with our example using Equation 7.1, the architecture that can capture this formula consists of two layers. Recall that a feature vector is computed as follows:

$$\begin{aligned} x_v^{(i)} &= \text{COM}\left(x_v^{(i-1)}, \text{AGG}(\{\!\{x_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\})\right) \\ &= \sigma\left(x_v^{(i-1)} C + \sum_{u \in \mathcal{N}_G(v)} x_u^{(i-1)} A + b\right). \end{aligned} \tag{7.2}$$

The first layer will use feature vectors $x_v^{(0)} \in \mathbb{R}^2$ and outputs feature vectors $x_v^{(1)} \in \mathbb{R}^3$. This is achieved by defining $A, C \in \mathbb{R}^{2\times 3}$, and $b \in \mathbb{R}^3$. Writing Equation 7.2 specifically for this layer, we get:

$$x_v^{(1)} = \sigma\left(x_v^{(0)} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} + \sum_{u \in \mathcal{N}_G(v)} x_u^{(0)} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}\right).$$

The second and also last layer uses the output of the previous layer, which is $x_v^{(1)} \in \mathbb{R}^3$ and outputs feature vectors $x_v^{(2)} \in \mathbb{R}^4$. This layer can be denoted as:

$$x_v^{(2)} = \sigma\left(x_v^{(1)} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} + \sum_{u \in \mathcal{N}_G(v)} x_u^{(1)} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & -1 \end{bmatrix}\right).$$

The last feature, located in the fourth column, will be extracted using the classification function CLS to classify a node v as *true* if that component of $x_v^{(2)}$ corresponding to the graded μ-calculus formula φ holds 1.

Redefining the construction

Having established a new base for our construction, the cases defined by Barceló et al. discussed in section 5.3 must be redefined. Define L as the number of subformulas in $sub(\varphi)$ and $\kappa \leq L$ as the number of unique node colors in $sub(\varphi)$. This results in $\theta = L - \kappa$, defined as the number of logical operators in $sub(\varphi)$. This number θ will correspond to the number of layers needed in the GNN to evaluate the formula φ because the initial feature vectors will contain all node colors. The size and entries of the matrices A, C, and b of a specific layer depend on the incoming feature vector size and which operator it corresponds to. Define the input feature vector as $x_v^{(i-1)} \in \mathbb{R}^m$ thus; the output feature vector will be $x_v^{(i)} \in \mathbb{R}^n$ with $n = m + 1$ because each layer will add one feature to the feature vector. This results in the following cases with $A, C \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^n$:

Case 1. if $\varphi_n = \varphi_j \wedge \varphi_k$ then $C_{jn} = C_{kn} = 1$ and $b_n = -1$,

Case 2. if $\varphi_n = \neg\varphi_k$ then $C_{kn} = -1$ and $b_n = 1$,

Case 3. if $\varphi_n = \Diamond^N \varphi_k$ then $A_{kn} = 1$ and $b_n = -N + 1$,

and $C_{ii} = 1$, for $i = 1, \ldots, m$. All other values of A, C, and b are 0.

Each case will copy the first m features to the new feature vector $x_v^{(i)}$ due to the first m rows and columns of matrix C. This submatrix along the main diagonal corresponds to the identity matrix. This is because all elements on the diagonal are equal to one, while all other entries within this submatrix are zero. The last column of A, C, and b will aid the layer in simulating the corresponding logical operator. The first two cases only use matrix C, which is responsible for local properties. On the other hand, the diamond operator from *Case 3* is dependent on the neighbors, so it utilizes matrix A, which is responsible for message passing.

In *Case 1*, φ_n is the conjunction of two subformulas φ_j and φ_k. This new feature can be computed using only local features, specifically the features at index j and k. The values of these features are added up in the last column of the new feature vector by defining $C_{jn} = 1$ and $C_{kn} = 1$. The value of b_n, which is -1, ensures that the feature only results in 1 if the sum of the two features equals two. Without the bias, the feature would be 1 or *true* even if only one feature of the two is 1.

The following case is *Case 2*, where φ_n is the negation of a subformula φ_k. Like *Case 1*, this layer will use matrix C to compute the new feature. The column of matrix C will be defined as $C_{kn} = -1$, which will copy the k-th feature multiplied with -1 to the last column of the new feature vector. We need a bias to ensure that *false* will result in *true*. Assigning b_n a value of 1 achieves the desired outcome. Without the bias, the feature would result in 0 or *false* for every value.

In the last case, *Case 3*, subformula φ_n is the diamond operator on a subformula φ_k. This operator does not rely on local features for its computation. Consequently, the final column of the matrix C contains only zeros. Matrix A will be multiplied with the feature vectors of the neighbors, thereby being responsible for computing the feature of the diamond operator. Matrix A copies the k-th feature of every neighbor and sums them up. Subformula φ_n only evaluates to *true* if at least N neighbors satisfy the k-th subformula; hence, the bias vector will add $-N + 1$ to the computed sum to get the desired result.

7.1.2 Adding recursion

The subsequent step entails introducing recursion to both the logical and GNN aspects. This necessitates an expansion of the previously mentioned construction. Before proposing this expansion, the RecGNN architecture is analyzed. Notably, the prior notion of node colors will be extended to encompass fixpoint relations, i.e., relation X from fixpoint operator μ_X. In other words, fixpoint relations will themselves be considered valid node colors. This extension will also influence κ, which now contains the number of unique base colors and fixpoint relations. Base colors cannot share the same identifier as a fixpoint relation to avoid ambiguity and ensure well-defined behavior. In other words, a property denoted by the same letter used for a fixpoint relation cannot exist within the input graph.

Transformation layer

As defined in section 6.5, a RecGNN will repeatedly apply iterations until the node classifier becomes stable. Since the output feature vectors of one iteration serve as the input feature vectors for the subsequent iteration, their dimensions must remain consistent. Our construction expands the feature vector, and this expansion is related to the number of layers present in the GNN architecture. To ensure dimensional compatibility, an additional final layer, a transformation layer, must be appended to the architecture. This layer maps the current feature vector $x_v^{(i-1)}$ back to the initial size, guaranteeing that the output $x_v^{(i)}$ remains in $\mathbb{R}^\kappa$. This layer can be interpreted as a simple linear layer or a dense layer that implements the following operation:

$$x_v^{(i)} = \sigma(x_v^{(i-1)} C).$$

Define $x_v^{(i-1)} \in \mathbb{R}^m$ thus the matrix C is defined in $\mathbb{R}^{m \times \kappa}$ and $C_{ii} = 1$, for $i = 1, \ldots, \kappa$. All other values of C are 0. This transformation layer effectively returns the expanded feature vector to its original dimensionality by discarding the information added during the intermediate layers. Before this layer applies the transformation, the final classification function, denoted as CLS, will extract the final component of the feature vector to classify the nodes.

Readiness feature

Every operator in graded modal logic can be computed in one iteration, but fixpoint operators may take several iterations. The number of iterations required for convergence is inherently unknown beforehand due to the dependence of these iterations on the input graph. While the CLS function extracts the final component in each iteration to perform classification, determining whether a fixpoint has been reached remains an open question. Extra readiness features are introduced to mitigate this problem. Each feature gets an additional feature, representing its readiness. Thus, the initial feature vectors will be defined in $\mathbb{R}^{2\kappa}$, and each layer will expand this size with two features: the subformula feature and its readiness.

How is this readiness computed? All node colors, the leaves in the logic formula's abstract syntax tree, have a readiness feature valued at one, meaning *true*. The concept of readiness propagates upward through the abstract syntax tree due to how nodes in the tree utilize the conjunction to determine their own readiness based on their children's readiness. Upon reaching a state of readiness in the root node of the abstract syntax tree, the interpretation of the CLS output becomes valid, as readiness signifies a complete and interpretable state.

7.1.3 The least fixpoint operator

This section will explain the key ideas for implementing the least fixpoint operator μ_X. Such an operator must check for each node if the old value equals the new one. Equality comparison can be established between the feature of the corresponding fixpoint relation X and another operator, the direct child of the μ-operator within the AST. It is important to note that, as a binary tree, each μ-operator in the AST possesses only one child. This comparison allows us to determine whether every node in the input graph's value has been updated or reached its fixpoint.

Determining whether all nodes in the input graph have reached their fixpoint necessitates a global readout operation, of which the inner workings are discussed in subsection 4.4.2. This step requires knowledge of the total number of nodes within the input graph. To address this, we introduce an additional node color n with *true* assigned to every node. By incorporating this feature into the global readout, we can verify whether a global fixpoint has been achieved. So, the definition of κ will be expanded again, incorporating this additional node color n. Note that, like the fixpoint relations, this color identifier cannot already be used in the input graph. Next, if the direct child operator is ready and the fixpoint is reached, we update the fixpoint relation X feature.

Fixpoint operators can exhibit nested structures. In such cases, the relations associated with these nested fixpoint operators must be reset once they have reached their respective fixpoints. This reset operation is performed by the enclosing (outer) fixpoint operator. The outer fixpoint operator verifies this using the readiness of its direct child operator. Since readiness signifies the completion of computations within a subtree, all operators within the subtree, including any nested fixpoint operators, are ready.

7.2 Construction

Following our modification to the feature vector construction and the motivation for employing ACR-GNNs, this section delves into a detailed explanation of the various layer types, with our main focus on the construction of the least fixpoint operator μ. Like the work by Barceló et

al., this construction will utilize the same aggregation, combine, and readout functions:

$$\begin{aligned} \mathrm{AGG}(X) &= \sum_{x \in X} x, \\ \mathrm{READ}(X) &= \sum_{x \in X} x, \\ \mathrm{COM}(x, y, z) &= \sigma(xC + yA + zR + b), \end{aligned}$$

with σ the truncated ReLU activation function. Building on the definitions provided by Barceló et al., our RecGNN can be classified as simple and homogeneous. It is simple due to its reliance on aggregation, combine, and readout functions and homogeneous because these functions remain consistent across all network layers. The feature vectors will be computed using the following equation:

$$\begin{aligned} x_v^{(i)} &= \mathrm{COM}\left(x_v^{(i-1)}, \mathrm{AGG}(\{\!\{x_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\}), \mathrm{READ}(\{\!\{x_u^{(i-1)} \mid u \in G\}\!\})\right) \\ &= \sigma\left(x_v^{(i-1)} C + \sum_{u \in \mathcal{N}_G(v)} x_u^{(i-1)} A + \sum_{u \in G} x_u^{(i-1)} R + b\right). \end{aligned} \tag{7.3}$$

The initial feature vector $x_v^{(0)}$ will contain all the different node colors in the input graph, which are used in the formula φ. The fixpoint relations with the additional node color n, *true*, for every node are also present. The motivation for this additional color is given in subsection 7.1.3. Each feature will also have its readiness feature set to *true* for every node color. The readiness will propagate upward throughout the abstract syntax tree of the graded μ-calculus formula and signals if the whole formula is computed. Continuing with the same notation, we use κ to define the initial feature vector, denoted as $x_v^{(0)} \in \mathbb{R}^{2\kappa}$. Note that the additional color n and its readiness are always the first two features in the vector.

7.2.1 Basic layers

The 'basic' layers implemented in our construction are the layers from section 7.1.1, additionally with the disjunction layer because the operator is often used. None of these layers utilizes the readout function; only the layer simulating the diamond operator uses message passing. A breakdown of each layer is presented in this section. Each operator can be implemented using one layer using Equation 7.3. Define the input feature vector as $x_v^{(i-1)} \in \mathbb{R}^m$ thus the output feature vector will be $x_v^{(i)} \in \mathbb{R}^n$ with $n = m + 2$. In the feature vector $x_v^{(i)}$, the first m features will be a copy of feature vector $x_v^{(i-1)}$. The two extra features will be the operator's logical value, simulated by the layer, and its readiness. With the logical value at $(x_v^{(i)})_{n-1}$ and its readiness at $(x_v^{(i)})_n$. This results in $A, C \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^n$. Every implementation will use the foundation defined in section 7.1.1. Because every layer copies the m features from the previous feature vector, matrix C must have $C_{ii} = 1$, for $i = 1, \ldots, m$.

The first implementation will consider the logical operator $\wedge$, assuming that the current subformula from $sub(\varphi)$ can be written as $\varphi_j \wedge \varphi_k$. Then $C_{j(n-1)} = C_{k(n-1)} = 1$ and $b_{n-1} = -1$ to compute the output of the logical operator. This operator has two subformulas, meaning that the operator's readiness will be defined as the conjunction of the subformulas readiness. This results in a similar operation for the readiness, $C_{jn} = C_{kn} = 1$ and $b_n = -1$.

The second logical operator will be the disjunction operator $\vee$ if the current subformula can be denoted as $\varphi_j \vee \varphi_k$ then $C_{j(n-1)} = C_{k(n-1)} = 1$. The truncated ReLU activation function eliminates the need for a bias term in this operator's simulation. The computation of the readiness of this operator will also use the conjunction, thus $C_{jn} = C_{kn} = 1$ and $b_n = -1$.

The third operator is the negation $\neg$, which is used when the current subformula is $\neg\varphi_k$. Then $C_{k(n-1)} = -1$ and $b_{n-1} = 1$. This operator is a unary operator, so it inherits the readiness of its single subformula, $C_{kn} = 1$.

The three operators discussed previously operate exclusively on local features. Consequently, they solely rely on matrix C and bias vector b from Equation 7.3 for computations. Therefore, we classified these layers as 'local' due to the absence of message passing or global readout mechanisms.

The next layer simulates the diamond operator $\Diamond^N$, used when the current subformula can be written as $\Diamond^N \varphi_k$. Then $A_{k(n-1)} = 1$ and $b_{n-1} = -N + 1$. Because it is a unary operator, the readiness is directly inherited from its subformula. Thus, $C_{kn} = 1$. This layer stands out as the sole message passing layer due to its exclusive utilization of matrix A.

The last 'basic' layer is the final layer, which does not simulate logical operators. This layer transforms the current feature vector to its initial size by defining matrix $C \in \mathbb{R}^{m \times \kappa}$ and $C_{ii} = 1$, for $i = 1, \ldots, \kappa$. All other values of C are 0.

7.2.2 μ-layer

The following section delves into a more detailed description of the construction of the least fixpoint operator, μ, the final logical operator to be discussed. While the general implementation idea was previously presented in subsection 7.1.3, a more in-depth examination is provided here. Simulating this operator using a single layer based on Equation 7.3 proves insufficient. A combination of helper layers using Equation 7.3 is necessary to achieve this simulation. These helper layers can be divided into two parts: the first focuses on determining local readiness, while the second performs a global readout.

Checking local readiness

This part of the computation utilizes three helper layers to accomplish two tasks: check the equality between the fixpoint relation and the direct child of the μ-operator and reset the fixpoint relations of nested fixpoint operators if present. Because we only use matrix multiplications and additions with binary values, equality can not be computed in one step. The following logical expression is used to simulate equality between fixpoint relation X and the direct child:

$$(X \wedge child) \vee (\neg X \wedge \neg child). \tag{7.4}$$

The first helper layer will expand the feature vector with four features. The first additional feature is the negation of the readiness of the direct child, denoted as $\neg child\_r$, which is used in the next helper layer to reset the nested fixpoint relations if necessary. The last three additional features are $X \wedge child$, $\neg X$, and $\neg child$, which are used in the equality check.

The second helper layer will add another feature for the equality check, which is $\neg X \wedge \neg child$. Simultaneously, we will use the feature $\neg child\_r$ in a conjunction with each nested fixpoint relation to perform a rest if necessary. This reset works because the readiness of the direct child implies that the entire subtree, including all nested fixpoint operators, is ready. In the case of a $false$ readiness value, the negation operation produces $true$. The overall value remains unchanged since a conjunction with $true$ has no effect. Conversely, a $true$ readiness value resulting in $false$ after negation, combined with a value through conjunction, always results in $false$, effectively resetting the values.

The process introduces five additional features. However, only two new features persist alongside the existing ones because we remove three intermediate features and redefine the last two in helper layer three. The second to last feature defines the equality between X and the direct child by performing the last step of Equation 7.4, computing the disjunction of the features $(X \wedge child)$ and $(\neg X \wedge \neg child)$. The last feature is the readiness of the μ-operator, which is initialized as $false$. Note that these helper layers introducing intermediate features do not need extra readiness features for every introduced feature.

Performing a global readout

Until this point, only locality checks have been performed. To check if a fixpoint is reached, all nodes in the graph must have a *true* equality feature. This computational stage employs five helper layers to verify convergence and update the fixpoint relation. During this final phase, no additional features are introduced; hence, we end with two extra features like any other layer we defined.

The first helper layer of this part is the only layer using a global readout by using matrix R from Equation 7.3. The extra node color is used to calculate the number of nodes present in the input graph to check convergence for every node. This is done by applying the first part of the following logical expression:

$$\neg\left[\sigma(n - count)\right], \tag{7.5}$$

with n the sum of nodes using the extra node color and *count* the sum of *true* equality features. Note that the negation cannot be computed before we have materialized subformula $\sigma(n - count)$. This subresult is stored in the last feature, representing the readiness of the fixpoint operator. Simultaneously, in the first helper layer, the feature of the direct child is copied to the second to last feature, the feature of the fixpoint operator. The second helper layer will take the negation of the readiness of the fixpoint operator, so it simulates Equation 7.5.

The last step in determining the readiness feature is to do a conjunction of the readiness with the readiness of the direct child. Verifying readiness solely at the current level is insufficient. While the intermediate steps determine if a fixpoint has been reached, they do not guarantee the readiness of every underlying operator. For the μ-operator to be considered ready, its direct child operator must be ready. Helper layer three will check this constraint by taking the conjunction between the two readiness features, thus making sure we are ready locally and ready globally.

While the first three helper layers of this part are responsible for computing the readiness of the fixpoint operator, the subsequent two helper layers will update the fixpoint relation if necessary by computing the feature of the fixpoint operator itself. A fixpoint relation is only allowed to update if the entire subtree, including nested fixpoint operators, is ready. This is done by defining the feature of the fixpoint operator as the conjunction between itself and the readiness feature of the direct child. This computation is executed in helper layer four. This will reset the fixpoint operator feature if the subtree is not ready yet. In the fifth helper layer, the fixpoint relation is updated by taking the disjunction of this feature with the fixpoint relation feature. The disjunction operator is allowed because we only consider positive formulas.

Are these last two steps essential, and can we not just update the fixpoint relation by copying the fixpoint operator feature to the feature of its fixpoint relation? A fixpoint relation is only allowed to be updated if the entire subtree is ready because nested fixpoint relations could not be ready yet. Suppose an update is performed when a nested fixpoint is not converged. In that case, it perhaps leads to undesired results for the nested fixpoint relation because it suddenly has an updated version of the outer fixpoint relation.

7.2.3 Examples

Finishing this section with an example: $\alpha = \mu_X\left[P \vee \Diamond X\right]$ with its abstract syntax tree depicted in Figure 7.1 and the following subformulas:

$$sub(\alpha) = \left(P, X, \Diamond X, P \wedge \Diamond X, \mu_X\left[P \vee \Diamond X\right]\right).$$

The initial feature vector will be defined in $\mathbb{R}^6$ because we have the following base features:

$$\left[n, n\_r, P, P\_r, X, X\_r\right].$$

The corresponding RecGNN architecture will have four main layers with eight helper layers, thus 11 layers in total, because the eight helper layers represent a μ-layer.

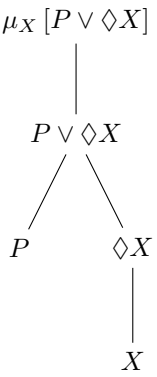

Figure 7.1: Abstract Syntax Tree of graded μ-calculus formula α.

Layer 1. diamond_layer(N: 1, *child*: 4, *child_r*: 5)

Layer 2. or_layer(*child*1: 2, *child*1_*r*: 3, *child*2: 6, *child*2_*r*: 7)

Layer 3. mu_layer(X: 4, *child*: 8, *child_r*: 9, list of nested μ's: [ ])

Part 1: mu_local_readiness_check(X: 4, *child*: 8, *child_r*: 9, list of nested μ's: [ ])

Helper layer 1. responsible for computing $\neg child\_r$, $X \wedge child = X \wedge (P \wedge \Diamond X)$, $\neg X$, and $\neg child = \neg(P \wedge \Diamond X)$.

Helper layer 2. responsible for computing $\neg X \wedge \neg child = \neg X \wedge \neg(P \wedge \Diamond X)$. No resets must be performed because there are no nested μ-operators.

Helper layer 3. responsible for computing $(X \wedge child) \vee (\neg X \wedge \neg child) =$
$(X \wedge (P \wedge \Diamond X)) \vee (\neg X \wedge \neg(P \wedge \Diamond X))$.

Part 2: mu_global_readout(X: 4, *child*: 8, *child_r*: 9)

Helper layer 1. performing a global readout to compute $\sigma(n - count)$.

Helper layer 2. responsible for computing $\neg [\sigma(n - count)]$.

Helper layer 3. responsible for computing $\mu\_r = \mu\_r \wedge child\_r$.

Helper layer 4. responsible for computing $\mu = \mu \wedge child\_r$.

Helper layer 5. responsible for updating the fixpoint relation $X = X \vee \mu$.

Layer 4. final_layer(classification feature: 10)

each corresponding to a subformula from $sub(\alpha)$. The input of the final layer will look like:

$$[n, n\_r, P, P\_r, X, X\_r, \Diamond, \Diamond\_r, \vee, \vee\_r, \mu, \mu\_r] .$$

A more complicated graded μ-calculus formula is $\beta = \nu_Y [\mu_Z(\Diamond Y \wedge (P \vee \Diamond Z))]$. This formula has a nested fixpoint consisting of a greatest and a least fixpoint, which has already been discussed in subsection 2.4.3. Our construction supports this formula but has to be converted to match the implementation, which does not use the greatest fixpoint operator ν. After converting this formula, we get the following:

$$\beta = \neg\mu_Y \neg [\mu_Z(\Diamond\neg Y \wedge (P \vee \Diamond Z))] .$$

The subformulas for formula β will be:

$$sub(\beta) = (P, Y, Z, \neg Y, \Diamond\neg Y, \Diamond Z, P \vee \Diamond Z, \Diamond\neg Y \wedge (P \vee \Diamond Z), \mu_Z(\Diamond\neg Y \wedge (P \vee \Diamond Z)),$$
$$\neg [\mu_Z(\Diamond\neg Y \wedge (P \vee \Diamond Z))] , \mu_Y \neg [\mu_Z(\Diamond\neg Y \wedge (P \vee \Diamond Z))] , \neg\mu_Y \neg [\mu_Z(\Diamond\neg Y \wedge (P \vee \Diamond Z))]).$$

The formula has three node colors; hence, the initial feature vector will be defined in $\mathbb{R}^8$ and looks like:

$$[n, n\_r, P, P\_r, Y, Y\_r, Z, Z\_r]\,.$$

The RecGNN architecture will have 24 layers:

Layer 1. negation_layer(4, 5)

Layer 2. diamond_layer(1, 8, 9)

Layer 3. diamond_layer(1, 6, 7)

Layer 4. or_layer(2, 3, 12, 13)

Layer 5. and_layer(10, 11, 14, 15)

Layer 6. mu_layer(6, 16, 17, [ ])

Part 1: mu_local_readiness_check(6, 6, 17, [ ])

Helper layer 1. responsible for computing $\neg child\_r$, $X \wedge child$, $\neg X$, and $\neg child$.

Helper layer 2. responsible for computing $\neg X \wedge \neg child$. No resets must be performed because there are no nested μ-operators.

Helper layer 3. responsible for computing $(X \wedge child) \vee (\neg X \wedge \neg child)$.

Part 2: mu_global_readout(6, 16, 17)

Helper layer 1. performing a global readout to compute $\sigma(n - count)$.

Helper layer 2. responsible for computing $\neg\,[\sigma(n - count)]$.

Helper layer 3. responsible for computing $\mu\_r = \mu\_r \wedge child\_r$.

Helper layer 4. responsible for computing $\mu = \mu \wedge child\_r$.

Helper layer 5. responsible for updating the fixpoint relation $X = X \vee \mu$.

Layer 7. negation_layer(18, 19)

Layer 8. mu_layer(4, 20, 21, [6])

Part 1: mu_local_readiness_check(4, 20, 21, [6])

Helper layer 1. responsible for computing $\neg child\_r$, $X \wedge child$, $\neg X$, and $\neg child$.

Helper layer 2. responsible for computing $\neg X \wedge \neg child$. Possibly reset the feature at index 6 if the whole subtree is ready.

Helper layer 3. responsible for computing $(X \wedge child) \vee (\neg X \wedge \neg child)$.

Part 2: mu_global_readout(4, 20, 21)

Helper layer 1. performing a global readout to compute $\sigma(n - count)$.

Helper layer 2. responsible for computing $\neg\,[\sigma(n - count)]$.

Helper layer 3. responsible for computing $\mu\_r = \mu\_r \wedge child\_r$.

Helper layer 4. responsible for computing $\mu = \mu \wedge child\_r$.

Helper layer 5. responsible for updating the fixpoint relation $X = X \vee \mu$.

Layer 9. negation_layer(22, 23)

Layer 10. final_layer(24)

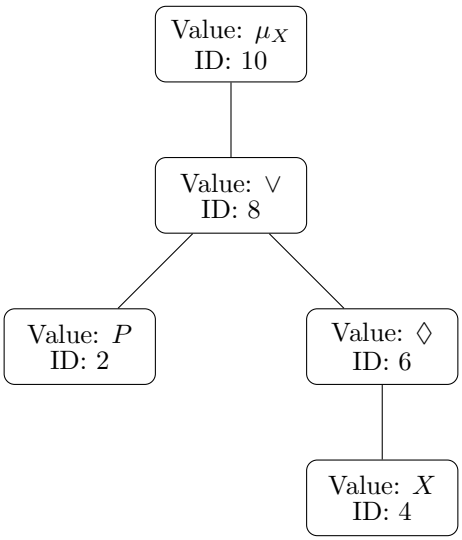

Figure 7.2: Traversing the AST of formula $\mu_X[P \vee \Diamond X]$ in post-order to ID each node.

7.3 Compiler

The above construction of an algorithm for simulating graded μ-calculus logical operators using recurrent graph neural networks (RecGNNs) demonstrates the capability for capturing positive graded μ-calculus formulas with these networks. However, the current approach necessitates the manual creation of the RecGNN architecture. This limitation highlights the need for a compiler to automatically translate a graded μ-calculus expression into its corresponding RecGNN architecture.

We implemented a corresponding compiler that takes a graded μ-calulus expression as its input. Such graded μ-calculus expressions have limited operators, so converting them from infix notation to prefix notion is relatively easy. We use a stack during this conversion with some precedence rules. This notation enables the construction of the associated abstract syntax tree using a stack and reading the prefix formula backward. The compiler can construct the RecGNN architecture with the correct layer order and parameters by traversing this tree using a post-order traversal. Each node gets an ID during this traversal, and the base node colors are identified; see Figure 7.2. These node colors are necessary to start the construction of the initial feature vector.

The parameters needed for every layer, like the child nodes or the nested μ-operators, can easily be identified through our binary tree. The data structure of the tree is constructed so each node can access its direct children and easily search through the subtree with itself as the root.

Chapter 8

Implementation of the method

In chapter 7, the focus lies on the core concepts and construction methods necessary to create a RecGNN architecture capable of capturing a positive graded μ-calculus formula. While the focus remained on the theoretical underpinnings, the implementation specifics were not addressed. This chapter delves into the details of the implementation of this method.

8.1 Implementation details

First, the implementation details of the method itself are discussed. The method is implemented in Python. We exclusively rely on Python's `NumPy` package, utilizing its efficient matrix operations and `NumPy` arrays to construct the matrices in the RecGNN layers. `NumPy` offers highly optimized matrix operations, leading to efficient computations and the ability of the arrays to be easily manipulated and combined, enabling the implementation of various layer functionalities without excessive code complexity.

Central to our implementation are feature vectors, represented as `NumPy` arrays. These arrays encapsulate the state of a node at a specific layer during the computation. Each node within the input graph is assigned a feature vector stored within a feature map data structure, a 2D array. Traditionally, constructing the adjacency matrix representing the connections between nodes in the input graph can be a time-consuming, manual process. To address this challenge, we leverage the `NetworkX` package. `NetworkX` offers a streamlined approach to defining nodes and edges within the input graph. Once the graph G is specified, the appropriate adjacency matrix can be effortlessly retrieved using the `NetworkX` function: `adj_mat = nx.adjacency_matrix(G).todense()` and subsequently employed as input for our method.

Moreover, we utilize standard Python packages like `re`, which supports regular expressions used in our compiler. The `argparse` package offers a user-friendly command-line interface, and the packages `os` and `json` are used to call and create the dump file, which can be made during compilation.

8.2 RecGNN evaluator

The implementations of the layers representing a logical operator are functions in the class `RecGNN_Evaluator`. This class takes the adjacency matrix and initial feature vectors as inputs. Besides the inputs, two variables are stored within the class: the initial feature vector size and a boolean to indicate readiness. These two variables are used in the final layer to convert the feature vector back to the initial size and to determine whether to stop iterating.

The truncated activation function has its dedicated function utilizing the separation of concerns design principle. The function `carb(self, tmp_feature_vectors, C, A, R, b)`, the shared component of every layer, which will use the `truncated_relu(self, x)`. The function name `carb` is an abbreviation of the matrices C, A, R and b used in Equation 7.3. Completing the class functionality, a set of methods is implemented to address various logical operations.

8.2.1 Basic layers

The basic layers implemented are conjunction, disjunction, negation, diamond operator layers, and the final layer. Their corresponding functions are:

```
def and_layer(self, child1_feature_idx, child1_readiness_idx,
              child2_feature_idx, child2_readiness_idx)

def or_layer(self, child1_feature_idx, child1_readiness_idx,
             child2_feature_idx, child2_readiness_idx)

def negation_layer(self, child_feature_idx, child_readiness_idx)

def diamond_layer(self, N, child_feature_idx, child_readiness_idx)

def final_layer(self, classification_feature_idx)
```

where each parameter is an integer. Each layer creates and defines the matrices C, A, R and b and calls the `carb` function to execute the layer.

8.2.2 μ-layer

Next is the layer that simulates the least fixpoint, using eight helper layers. These are separated into two parts that use three and five layers, respectively. The layer itself is called:

```
def mu_layer(self, x_feature_idx, child_feature_idx, child_readiness_idx,
         list_mu_child_feature_idx):

    __mu_local_readiness_check(self, x_feature_idx, child_feature_idx,
            child_readiness_idx, list_mu_child_feature_idx)

    __mu_global_readout(self, x_feature_idx, child_feature_idx,
            child_readiness_idx)
```

where every parameter except `list_mu_child_feature_idx` is an integer. The list parameter is a list consisting of integers. Like the basic layers, each helper layer will create and define the matrices it utilizes and pass them to the `carb` function to execute the layer.

8.3 Mu compiler

In the following section, we will delve into the details of the file containing the Mu compiler. We will begin by examining its functionality, followed by discussing its usage.

8.3.1 Functionality

The `mu_compiler` class encapsulates the conversion process from a graded μ-calculus formula to its corresponding RecGNN architecture. The first step in the compiler is the lexer. The lexer tokenizes the formula by converting the logical formula from infix notation to prefix notation. To simplify the process, each operator is encoded to a single char. By doing this, we can use a character-based stack during the conversion. This process is shown here:

```
def __lexer(self):
    self.__convert_to_char_ops()
    self.infix_to_prefix(self.formula)
    self.__reverse_char_ops()
```

During this last conversion, the formula will be represented as a list of tokens, which can be used to build the syntax tree by parsing the list in reverse order. The next step is called the parser, where the syntax tree is constructed from the tokenized prefix notation using a stack. During this step, the node colors are identified and given an ID used as its index in the coming feature vector. The parser's final part will ID all syntax tree's internal nodes by traversing it in post-order. Along the way, it also searches for fixpoint operators and saves them in a list. The parser looks like this:

```
def __parser(self):
    self.__save_predicates()
    self.__id_predicates()

    self.__build_syntax_tree() # Tree with id'ed leaves (predicates).
    self.__id_internal_nodes(self.syntax_tree)
```

The final step is to build the RecGNN architecture by traversing the ID'ed syntax tree in post-order and creating a layer for every internal node we see. For every type of layer, every parameter can easily be found by looking in the subtree. Note that this function `__build_recgnn_logical_layers(self, node)` is recursive because the depth of the AST is dependent on the logical formula. After traversing the whole tree, the final layer is added to the architecture by hitting the recursive base cases at the leaves.

8.3.2 Usage

The Mu compiler operates through a command-line interface powered by the `argparse` package. It takes user-provided arguments to configure its behavior. Based on these arguments, it will show the architecture or create an output and dump file for later analysis on a specific input graph. The help argument (`--help`) can also be called and will show the following:

```
usage: mu_compiler.py [-h] [-v] [-o OUTPUT] [-d DUMP] formula

Convert a graded mu-calculus formula to a RecGNN architecture.
Instructions:
1. The implementation language consists of the following operands:
   - Unary operators: Mu X, !, <>^N
   - Binary operators: &&, ||
   - Predicates: P,Q,X,Y...
   - Forbidden predicate: n (lowercase n)
2. The formula must be positive.
3. Use double negations to remove Nu's and [] operands.

An example formula is: '! [Mu Y [! Mu Z [<>!Y && (P || <>Z) ] ] ]'

positional arguments:
  formula               the graded mu-calculus formula to evaluate.

options:
  -h, --help            show this help message and exit
  -v, --verbosity
  -o OUTPUT, --output OUTPUT
                        the python file to write the architecture to.
  -d DUMP, --dump DUMP  the JSON file to dump the architecture to.
```

The compiler only requires a graded μ-calculus formula as a minimum. The other options are not required and will only enhance the user's understanding. The verbosity option will show the intermediate steps during the compilation. The output option will write the RecGNN architecture to a Python file where the user only has to define the adjacency matrix and feature map of the user-provided input graph. The last option is to dump the RecGNN architecture into a JSON file.

8.4 RecGNN syntax layers

The compiler offers an additional functionality to enhance user understanding of how formulas are processed within the architecture. This functionality allows for visualizing all relevant matrices C, A, R, and b associated with each layer. By providing this detailed layer-wise breakdown, the compiler facilitates a deeper comprehension of the operator transformations applied by the formula across the entire network. Note that if a matrix contains only zeros, it will not be present in the dump for that specific layer.

During compilation, the compiler will run another Python file: `recgnn_syntax_layers.py`, responsible for creating the JSON dump. Besides the architecture, the compiler will pass extra information like the original formula and a list of used predicates via arguments to incorporate this along with the RecGNN architecture. The parameters this Python file supports are:

```
usage: recgnn_syntax_layers.py [-h] formula predicates colors arch output

positional arguments:
  formula       The graded mu-calculus formula
  predicates    The predicates in the formula
  colors        The base colors in the formula
  arch          The architecture of the network
  output        the JSON file to dump the RecGNN architecture to.

options:
  -h, --help    show this help message and exit
```

8.5 Example

Finishing this chapter with a demo of the compiler on our example: $\alpha = \mu_X[P \vee \Diamond X]$. The following command will call the compiler with the formula α as an argument:

```
$ python utils/mu_compiler.py "Mu X [P || <>X]" -o test.py -d test.json -v

Converted formula: µ (P | 1X)
Prefix formula: µ|P1X
Prefix list: ['Mu X', '||', 'P', '<>^1', 'X']
Predicates: ['P', 'X']
Current id: 6
Syntax tree:
(Value: Mu X, ID: None, Left: (Value: ||, ID: None, Left:
(Value: P, ID: 2, Left: None, Right: None), Right:
(Value: <>^1, ID: None, Left: (Value: X, ID: 4, Left: None, Right: None),
Right: None)), Right: None)
Syntax tree with ids:
(Value: Mu X, ID: 10, Left: (Value: ||, ID: 8, Left:
(Value: P, ID: 2, Left: None, Right: None), Right:
(Value: <>^1, ID: 6, Left: (Value: X, ID: 4, Left: None, Right: None),
Right: None)), Right: None)
```

```
diamond_layer(1, 4, 5)
or_layer(2, 3, 6, 7)
mu_layer(4, 8, 9, [])
final_layer(10)
Predicates: ['n', 'n_r', 'P', 'P_r', 'X', 'X_r']
Base predicates: ['P']
```

Note that this example has the verbosity set to *true*, so we can see the intermediate steps. The output and dump files associated with the compilation process are available in Appendix B.

The console output shows that the initial feature vector will contain six features because two predicates are detected in the formula: `['P', 'X']`. Adding the additional node color n and the readiness features, we end up with `['n', 'n_r', 'P', 'P_r', 'X', 'X_r']` as the initial feature vector with $\kappa = 3$. The layers from the RecGNN architecture in the compiler output are precisely those from $sub(\alpha) = (P, X, \Diamond X, P \wedge \Diamond X, \mu_X [P \vee \Diamond X])$ minus the predicates.

Next, we will define an input graph, depicted in Figure 3.1, and predicate P will correspond to Purple nodes. The fixpoint has been reached at iteration four, and the RecGNN classifies nodes with ID one till seven as *true*, just like we computed by hand in section 3.3. Part of the output of the evaluation:

```
Iteration 4 :

Starting feature vectors:
[[1 1 0 1 1 1]
 [1 1 0 1 1 1]
 [1 1 0 1 1 1]
 [1 1 0 1 1 1]
 [1 1 1 1 1 1]
 [1 1 0 1 1 1]
 [1 1 0 1 1 1]
 [1 1 0 1 0 1]
 [1 1 0 1 0 1]
 [1 1 0 1 0 1]
 [1 1 0 1 0 1]
 [1 1 0 1 0 1]]
Is fixpoint reached? True
Current classification: [1 1 1 1 1 1 1 0 0 0 0 0]
------------------- RESULT -------------------
Fixpoint reached at iteration 4
Fixpoint classification: [1 1 1 1 1 1 1 0 0 0 0 0]
```

So, our compiler successfully created the RecGNN architecture, which captures the given graded μ-calculus formula. Moreover, no more iterations are needed with the evaluator than manual classification while achieving the same correct classification for every node in the input graph.

Chapter 9

Conclusion

To conclude this thesis, we highlight our key findings and the valuable lessons learned throughout the research and implementation process. We then discuss the challenges encountered and how we navigated them.

This thesis explores the feasibility of extending Barceló et al.'s work by creating a recurrent graph neural network (RecGNN) capable of evaluating a graded μ-calculus expression. We achieved a successful study answering this research question by leveraging the existing implementation of logical operators from GML [BKM$^+$20]. We focused on extending this framework to handle the least fixpoint operator μ specific to graded μ-calculus. Finally, we have gone beyond operator implementation by introducing a compiler that automatically translates a graded μ-calculus expression into a RecGNN architecture.

As someone completely new to these fields, I began by immersing myself in the literature on graded modal logic, graded μ-calculus, and graph neural networks. A strong foundation in these areas was crucial for the research to come. Next, we studied the proof of Barceló et al., of which a copy is present in Appendix A. That proof functioned as our foundation, and we needed to understand every intermediate step they took. By applying their construction to an example, we quickly realized that it has a computation inefficiency. We dived into this problem and came to the solution of expanding the feature vectors throughout the GNN. This construction of flexible feature vectors also optimized the number of layers needed in the AC-GNN, significantly reducing the number of computations. Next, we tried incorporating the least fixpoint operator μ into this optimized foundation. Unfortunately, we learned that this type of GNN cannot simulate a fixpoint because it needs global information, and local information does not suffice. So, we used the ACR-GNNs, which can perform a global readout and gather information from every node in the input graph. We introduced readiness features for every operator to show that we have reached a fixpoint if every readiness feature is *true*. Since the output of one iteration is used as an input in the following iteration, we needed an extra layer that transforms the expanded feature vector back to its original size.

The implementation part showed that creating a RecGNN architecture responding to the given graded μ-calculus formula is possible. Designed for flexibility, the compiler seamlessly handles any graded μ-calculus formula, provided it can be effectively translated into our chosen implementation language. Before implementing this compiler, we had to manually create the RecGNN architecture, which can be time-consuming if the formula is large.

Reflecting on this thesis, I noticed that I improved in several things. Undertaking the literature review, thinking outside the box, and completing the thesis writing task as a culmination of my master's presented a challenge of a new scale. Throughout this process, I gained valuable insights into qualitative literature search, scientific writing, and the field of theoretical computer science. I think my strongest point lies in my ability to plan my work, which allowed me to

finish this thesis in a timely fashion. The brainstorming sessions throughout the year improved my outside-the-box thinking and communication skills.

Looking at the final results, I am confident that this thesis successfully demonstrates the capability of RecGNNs to capture graded μ-calculus formulas. However, as with any research endeavor, there is always room for further exploration. I encountered challenges during the process that consumed valuable time. Despite these hurdles, the final outcome convincingly argues for the feasibility of the proposed approach and provides a clear justification for the choices made throughout the research.

As mentioned before, there is always room for further exploration. In our implementation, we used the truncated ReLU activation function, which is also used in the work of Barceló et al. In section 5.5, we explored the potential for utilizing a standard ReLU activation function. The analysis suggested its compatibility with minor adjustments to the current definition of Barceló et al.'s work. Building upon this observation, future work could involve a dedicated investigation into the modifications required to successfully integrate a standard ReLU into our implementation. Also, the number of features used in the RecGNN can be further investigated and optimized. This would further improve the current implementation. To further strengthen the validity of our findings, we can conduct controlled experiments using synthetic data. These experiments aim to definitively demonstrate that AC-GNN architectures lack the capability to learn classifiers in graded μ-calculus, while ACR-GNNs possess this ability

Bibliography

[AA19] Afshine Amidi and Shervine Amidi. Stanford cs 230: Deep learning. `https://stanford.edu/~shervine/teaching/cs-230/`, 2019. Last consulted on 4 May 2024.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[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. In *8th International Conference on Learning Representations, ICLR 2020*. OpenReview.net, 2020.

[Che16] Gang Chen. A gentle tutorial of recurrent neural network with error backpropagation. *CoRR*, abs/1610.02583, 2016.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[dR00] Maarten de Rijke. A note on graded modal logic. *Stud Logica*, 64(2):271–283, 2000.

[GM10] Claudio Gallicchio and Alessio Micheli. Graph echo state networks. In *International Joint Conference on Neural Networks, IJCNN 2010, Barcelona, Spain, 18-23 July, 2010*, pages 1–8. IEEE, 2010.

[Grä99] Erich Grädel. Why are modal logics so robustly decidable? *Bull. EATCS*, 68:90–103, 1999.

[Gro21] Martin Grohe. The logic of graph neural networks. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–17. IEEE, 2021.

[GSVdB22] Floris Geerts, Jasper Steegmans, and Jan Van den Bussche. On the expressive power of message-passing neural networks as global feature map transformers. In Ivan Varzinczak, editor, *Foundations of Information and Knowledge Systems - 12th International Symposium, FoIKS 2022, Helsinki, Finland, June 20-23, 2022, Proceedings*, volume 13388 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2022.

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[LA24] Fei-Fei Li and Ehsan Adeli. Stanford cs231n: Deep learning for computer vision. `https://cs231n.stanford.edu/`, April 2024. Last consulted on 4 May 2024.

[Les23] Jure Leskovec. Stanford cs224w: Machine learning with graphs. `http://cs224w.stanford.edu/`, October 2023. Last consulted on 27 April 2024.

[SGT+09] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

[Sta22] StatQuest with Josh Starmer. Long short-term memory (lstm), clearly explained. `https://www.youtube.com/watch?v=YCzL96nL7j0`, November 2022. Last consulted on 4 May 2024.

[Var96] Moshe Y. Vardi. Why is modal logic so robustly decidable? In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop 1996, Princeton, New Jersey, USA, January 14-17, 1996*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 149–183. DIMACS/AMS, 1996.

[VCC+18] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

Appendix A

Proof of Proposition 5.1.1

We will first recall the proposition.

Proposition 5.1.1. Each graded modal logic classifier is captured by a simple homogeneous AC-GNN.

We first define formally the semantics of the graded modal logic over simple undirected node-colored graphs, assuming the FO syntax introduced in the paper.

Definition A.1. *We define when a node v in a graph G satisfies a graded modal logic formula $\varphi(x)$, written as $v \models \varphi$ in G (where 'in G' may be omitted when clear), recursively as follows:*

- *if $\varphi(x) = \mathrm{Col}(x)$, then $v \models \varphi$ if and only if* Col *is the color of v in G,*
- *if $\varphi(x) = \varphi'(x) \wedge \varphi''(x)$, then $v \models \varphi$ if and only if $v \models \varphi'$ and $v \models \varphi''$, and similarly with $\neg\varphi'(x)$, and*
- *if $\varphi(x) = \exists^{\geq N} (E(x,y) \wedge \varphi'(y))$, then $v \models \varphi$ if and only if the set of nodes $\{u \mid u \in \mathcal{N}_G(v)$ and $v \models \varphi'\}$ has cardinality at least N.*

We can now proceed to the proof of the proposition.

Proof of Proposition 5.1.1. Let $\varphi(x)$ be a graded modal logic formula. We will construct an AC-GNN $\mathcal{A}_\varphi$ that is further simple and homogeneous. Let $\mathrm{sub}(\varphi) = (\varphi_1, \varphi_2, \ldots, \varphi_L)$ be an enumeration of the sub-formulas of φ such that if φ_k is a subformula of φ_l then $k \leq l$. The idea of the construction of $\mathcal{A}_\varphi$ is to have feature vectors in $\mathbb{R}^L$ such that every component of those vectors represents a different formula in $\mathrm{sub}(\varphi)$. Then $\mathcal{A}_\varphi$ will update the feature vector $\boldsymbol{x}_v^{(i)}$ of node v ensuring that component l of $\boldsymbol{x}_v^{(l)}$ gets a value 1 if and only if the formula φ_l is satisfied in node v.

We note that $\varphi = \varphi_L$ and thus, the last component of each feature vector after evaluating L layers in every node gets a value 1 if and only if the node satisfies φ. We will then be able to use a final classification function CLS that simply extracts that particular component.

Formally, the simple homogeneous AC-GNN $\mathcal{A}_\varphi$ has L layers and uses the aggregeation and combine functions

$$\mathrm{AGG}(X) = \sum_{\boldsymbol{x} \in X} \boldsymbol{x},$$
$$\mathrm{COM}(\boldsymbol{x}, \boldsymbol{y}) = \sigma\left(\boldsymbol{x}\boldsymbol{C} + \boldsymbol{y}\boldsymbol{A} + \boldsymbol{b}\right),$$

where $\boldsymbol{A}, \boldsymbol{C} \in \mathbb{R}^{L \times L}$, and $\boldsymbol{b} \in \mathbb{R}^L$ are defined next, and σ is the truncated ReLU activation defined by $\sigma(x) = \min(\max(0, x), 1)$. The entries of the l-th columns of $\boldsymbol{A}, \boldsymbol{C}$ and $\boldsymbol{b}$ depend on the sub-formulas of φ as follows:

Case 0. if $\varphi_l(x) = \mathrm{Col}(x)$ with Col one of the (base) colors, then $C_{ll} = 1$,

Case 1. if $\varphi_l(x) = \varphi_j(x) \wedge \varphi_k(x)$ then $C_{jl} = C_{kl} = 1$ and $b_l = -1$,

Case 2. if $\varphi_l(x) = \neg\varphi_k(x)$ then $C_{kl} = -1$ and $b_l = 1$,

Case 3. if $\varphi_l(x) = \exists^{\geq N}(E(x,y) \wedge \varphi_k(y))$ then $A_{kl} = 1$ and $b_l = -N + 1$,

and all other values in the l-th columns of $\boldsymbol{A}, \boldsymbol{C}$ and $\boldsymbol{b}$ are 0.

We now prove that $\mathcal{A}_\varphi$ indeed captures φ. Let $G = (V, E)$ be a colored graph. For every node v in G we consider the initial feature vector $\boldsymbol{x}_v^{(0)} = (x_1, \ldots, x_L)$ such that $x_l = 1$ if sub-formula φ_l is the initial color assigned to v, and $x_l = 0$ otherwise. By definition, AC-GNN $\mathcal{A}_\varphi$ will iterate the aggregation and combine functions defined above for L rounds (L layers) to produce feature vectors $\boldsymbol{x}_v^{(i)}$ for every node $v \in G$ and $l = 1, \ldots, L$ as follows:

$$\begin{aligned} \boldsymbol{x}_v^{(i)} &= \mathrm{COM}(\boldsymbol{x}_v^{(i-1)}, \mathrm{AGG}(\{\!\!\{\boldsymbol{x}_u^{(i-1)} \mid u \in \mathcal{N}_G(v)\}\!\!\})) \\ &= \sigma\left(\boldsymbol{x}_v^{(i-1)}\boldsymbol{C} + \sum_{u \in \mathcal{N}_G(v)} \boldsymbol{x}_u^{(i-1)}\boldsymbol{A} + \boldsymbol{b}\right). \end{aligned} \tag{A.1}$$

We next prove that for every $\varphi_l \in \mathrm{sub}(\varphi)$, every $i \in \{l, \ldots, L\}$, and every node v in G it holds that

$$(\boldsymbol{x}_v^{(i)})_l = 1 \text{ if } v \models \varphi_l, \text{ and } (\boldsymbol{x}_v^{(i)})_l = 0 \text{ otherwise,} \tag{A.2}$$

where $(\boldsymbol{x}_v^{(i)})_l$ is the l-th component of $\boldsymbol{x}_v^{(i)}$ that is, the l-th component of $\boldsymbol{x}_v^{(i)}$ has a 1 if and only if v satisfies φ_l in G. In the rest of the proof we will be continuously using the value of $(\boldsymbol{x}_v^{(i)})_l$ whose general expression is

$$(\boldsymbol{x}_v^{(i)})_l = \sigma\left(\sum_{k=1}^{L}((\boldsymbol{x}_v^{(i-1)})_k C_{kl} + \sum_{u \in \mathcal{N}(v)}\sum_{k=1}^{L}(\boldsymbol{x}_u^{(i-1)})_k A_{kl} + b_l\right). \tag{A.3}$$

We proceed to prove (A.2) by induction on the number of sub-formulas of every φ_l. If φ_l has one sub-formula, then $\varphi_l(x) = \mathrm{Col}(x)$ with Col a base color. We next prove that $(\boldsymbol{x}_v^{(1)})_l = 1$ if and only if v has Col as its initial color. Since $\varphi_l(x) = \mathrm{Col}(x)$ we know that $C_{ll} = 1$ and $C_{kl} = 0$ for every $k \neq l$ (see Case 0 above). Moreover, we know that $b_l = 0$ and $A_{kl} = 0$ for every k. Then, from Equation (A.3) we obtain that

$$(\boldsymbol{x}_v^{(1)})_l = \sigma\left(\sum_{k=1}^{L}((\boldsymbol{x}_v^{(0)})_k C_{kl} + \sum_{\{v,u\} \in E}\sum_{k=1}^{L}(\boldsymbol{x}_u^{(0)})_k A_{kl} + b_l\right) = \sigma((\boldsymbol{x}_v^{(0)})_l).$$

Then, given that $(\boldsymbol{x}_v^{(0)})_l = 1$ if the inital color of v is Col and $(\boldsymbol{x}_v^{(0)})_l = 0$ otherwise, we have that $(\boldsymbol{x}_v^{(1)})_l = 1$ if $(G, v) \models \varphi_l$ and $(\boldsymbol{x}_v^{(1)})_l = 0$ otherwise. From this it is easy to prove that for every $i \geq 1$ the vector $(\boldsymbol{x}_v^{(i)})_l$ satisfies the same property. Now assume that φ_l has more than one sub-formula, and assume that for every φ_k with $k < l$ the property (A.2) holds. Let $i \geq l$. We are left to consider the following cases, corresponding to the cases for the shape of the formula above.

Case 1. Assume that $\varphi_l(x) = \varphi_j(x) \wedge \varphi_k(x)$. Then $C_{jl} = C_{kl} = 1$ and $b_l = -1$. Moreover, we have $C_{ml} = 0$ for every $m \neq j$, k and $A_{nl} = 0$ for every n (see Case 1 above). Then, from Equation (A.3) we obtain that

$$(\boldsymbol{x}_v^{(i)})_l = \sigma\left((\boldsymbol{x}_v^{(i-1)})_j + (\boldsymbol{x}_v^{(i-1)})_k - 1\right).$$

Since the number of each proper sub-formula of φ_l is strictly less than both l and i, by induction hypothesis we know that $(\boldsymbol{x}_v^{(i-1)})_j = 1$ if and only if $v \models \varphi_j$ and $(\boldsymbol{x}_v^{(i-1)})_j = 0$ otherwise.

Similarly, $(\boldsymbol{x}_v^{(i-1)})_k = 1$ if and only if $v \models \varphi_k$ and $(\boldsymbol{x}_v^{(i-1)})_k = 0$ otherwise. Now, since $(\boldsymbol{x}_v^{(i)})_l = \sigma((\boldsymbol{x}_v^{(i-1)})_j + (\boldsymbol{x}_v^{(i-1)})_k - 1)$ we have that $(\boldsymbol{x}_v^{(i)})_l = 1$ if and only if $(\boldsymbol{x}_v^{(i-1)})_j + (\boldsymbol{x}_v^{(i-1)})_k - 1 \geq 1$ that can only happen if $(\boldsymbol{x}_v^{(i-1)})_j = (\boldsymbol{x}_v^{(i-1)})_k = 1$. Then $(\boldsymbol{x}_v^{(i)})_l = 1$ if and only if $v \models \varphi_j$ and $v \models \varphi_k$ that is, if and only if $v \models \varphi_l$ (since $\varphi_l(x) = \varphi_j(x) \wedge \varphi_k(x)$), and $(\boldsymbol{x}_v^{(i)})_l = 0$ otherwise. This is exactly what we wanted to prove.

Case 2. Assume that $\varphi_l(x) = \neg\varphi_k(x)$. Then $C_{kl} = -1$ and $b_l = 1$. Moreover, we have $C_{ml} = 0$ for every $m \neq k$ and $A_{nl} = 0$ for every n (see Case 2 above). Then, from Equation (A.3) we obtain that

$$(\boldsymbol{x}_v^{(i)})_l = \sigma\left(-(\boldsymbol{x}_v^{(i-1)})_k + 1\right).$$

By induction hypothesis we know that $(\boldsymbol{x}_v^{(i-1)})_k = 1$ if and only if $v \models \varphi_k$ and $(\boldsymbol{x}_v^{(i-1)})_k = 0$ otherwise. Since $(\boldsymbol{x}_v^{(i)})_l = \sigma(-(\boldsymbol{x}_v^{(i-1)})_k + 1)$ we have that $(\boldsymbol{x}_v^{(i)})_l = 1$ if and only if $1 - (\boldsymbol{x}_v^{(i-1)})_k \geq 1$ that can only happen if $(\boldsymbol{x}_v^{(i-1)})_k = 0$. Then $(\boldsymbol{x}_v^{(i)})_l = 1$ if and only if $v \not\models \varphi_k$ that is, if and only if $v \models \neg\varphi_k$, which holds if and only if $v \models \varphi_l$, and $(\boldsymbol{x}_v^{(i)})_l = 0$ otherwise. This is exactly what we wanted to prove.

Case 3. Assume that $\varphi_l(x) = \exists^{\geq N}(E(x,y) \wedge \varphi_k(y))$. Then $A_{kl} = 1$ and $b_l = -N + 1$. Moreover for every m we have that $C_{ml} = 0$ (see Case 3 above). Then, from Equation (A.3) we obtain that

$$(\boldsymbol{x}_v^{(i)})_l = \sigma\left(-N + 1 + \sum_{\{u,v\} \in E} (\boldsymbol{x}_u^{(i-1)})_k\right).$$

By induction hypothesis we know that $(\boldsymbol{x}_u^{(i-1)})_k = 1$ if and only if $u \models \varphi_k$ and $(\boldsymbol{x}_u^{(i-1)})_k = 0$ otherwise. Then we can write $(\boldsymbol{x}_v^{(i)})_l = \sigma(-N + 1 + m)$ where

$$m = |\{u \mid u \in \mathcal{N}(v) \text{ and } u \models \varphi_k\}|.$$

Thus, we have that $(\boldsymbol{x}_v^{(i)})_l = 1$ if and only if $m \geq N$, that is if and only if there exists at least N nodes connected with v that satisfy φ_k, and $(\boldsymbol{x}_v^{(i)})_l = 0$ otherwise. From that we obtain that $(\boldsymbol{x}_v^{(i)})_l = 1$ if and only if $v \models \varphi_l$ since $\varphi_l(x) = \exists^{\geq N}(E(x,y) \wedge \varphi_k(y))$, which is what we wanted to prove.

To complete the proof we only need to add a final classification after the L iterations of the aggregate and combine layers that simply classifies a node v as `true` if the component of $\boldsymbol{x}_v^{(L)}$ corresponding to φ holds 1. □

Appendix B

Compiler output

B.1 Python file

```
import numpy as np
import networkx as nx
from utils.recgnn_evaluator import RecGNN_Evaluator
from utils.print_colors import prRed, prGreen

# Here come the adjacency matrix and feature vectors (matrix)

# TIP: Use NetworkX to create the graph and its adjacency matrix.
# adj_matrix = np.array([])
# ['n', 'n_r', 'P', 'P_r', 'X', 'X_r'], Fill in the remaining values with 0
# or 1 based on the nodes and their properties.

# feature_vectors = np.array([[0, 1, ..., 1, 0, 1]])

def evalGNN(adj_matrix, feature_vectors, iterations=50):
    evaluator = RecGNN_Evaluator(adj_matrix, feature_vectors)
    fixpoint_iteration = None
    fixpoint_classification = None
    iteration = 0
    while iteration < iterations:
        print("-----------------------------------------------")
        print("Iteration", iteration+1, ":")
        print("")
        evaluator.print_feature_vectors("Starting feature vectors:")

        # Add the layers here
        evaluator.diamond_layer(1, 4, 5)
        evaluator.or_layer(2, 3, 6, 7)
        evaluator.mu_layer(4, 8, 9, [])
        classification = evaluator.final_layer(10)

        print("Current classification:", classification)
        if evaluator.get_overall_readiness():
            if fixpoint_iteration == None:
                fixpoint_iteration = iteration+1
                fixpoint_classification = classification
```

```
                break

        iteration += 1

    print("------------------- RESULT --------------------")
    if fixpoint_iteration == 0:
        prRed("No fixpoint reached after " + str(iterations) + " iterations.")
    else:
        prGreen("Fixpoint reached at iteration " + str(fixpoint_iteration))

    print("Fixpoint classification:", fixpoint_classification)

evalGNN(adj_matrix, feature_vectors, 20)
```

B.2 JSON dump


```
{
  "Formula": "Mu X [P || <>X]",
  "Initial feature vector size": "6",
  "Predicates": "['n', 'n_r', 'P', 'P_r', 'X', 'X_r']",
  "Base predicates": "['P']",
  "Layers": [
    {
      "Number": 1,
      "Operator": "diamond_layer(N=1, child_feature_idx=4,
                  child_readiness_idx=5)",
      "Input size": 6,
      "Output size": 8,
      "Matrices": {
        "C": [
          "[ 1  0  0  0  0  0  0  0]",
          "[ 0  1  0  0  0  0  0  0]",
          "[ 0  0  1  0  0  0  0  0]",
          "[ 0  0  0  1  0  0  0  0]",
          "[ 0  0  0  0  1  0  0  0]",
          "[ 0  0  0  0  0  1  0  1]"
        ],
        "A": [
          "[ 0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  1  0]",
          "[ 0  0  0  0  0  0  0  0]"
        ]
      }
    },
    {
      "Number": 2,
      "Operator": "or_layer(child1_feature_idx=2, child1_readiness_idx=3,
                  child2_feature_idx=6, child2_readiness_idx=7)",
      "Input size": 8,
      "Output size": 10,
      "Matrices": {
```

```
      "C": [
        "[ 1  0  0  0  0  0  0  0  0  0]",
        "[ 0  1  0  0  0  0  0  0  0  0]",
        "[ 0  0  1  0  0  0  0  0  1  0]",
        "[ 0  0  0  1  0  0  0  0  0  1]",
        "[ 0  0  0  0  1  0  0  0  0  0]",
        "[ 0  0  0  0  0  1  0  0  0  0]",
        "[ 0  0  0  0  0  0  1  0  1  0]",
        "[ 0  0  0  0  0  0  0  1  0  1]"
      ],
      "b": "[ 0  0  0  0  0  0  0  0  0 -1]"
    }
  },
  {
    "Number": 3,
    "Operator": "mu_layer(x=4, child_feature_idx=8, child_readiness_idx=9,
                list_mu_child_feature_idx=[])",
    "Input size": 10,
    "Output size": 12,
    "Helper layers": [
      {
        "Helper": 1,
        "Description": "mu_local_readiness_check: step 1 of equality check",
        "Input size": 10,
        "Output size": 14,
        "Matrices": {
          "C": [
            "[ 1  0  0  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  1  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  1  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  1  0  0  0  0  0  0 -1  0  1]",
            "[ 0  0  0  0  0  1  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  0  1  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  0  0  1  0  0  0 -1  1]",
            "[ 0  0  0  0  0  0  0  0  0  1 -1  0  0  0]"
          ],
          "b": "[ 0  0  0  0  0  0  0  0  0  0  1  1  1 -1]"
        }
      },
      {
        "Helper": 2,
        "Description": "mu_local_readiness_check: step 2 of equality check",
        "Input size": 14,
        "Output size": 15,
        "Matrices": {
          "C": [
            "[ 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  1  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  1  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0]",
```

```
        "[ 0  0  0  0  0  0  0  1  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  1  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  1  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  1  0  0  1]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  0  1  0  1]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0]"
      ],
      "b": "[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1]"
    }
  },
  {
    "Helper": 3,
    "Description": "mu_local_readiness_check: step 3 of equality check",
    "Input size": 15,
    "Output size": 12,
    "Matrices": {
      "C": [
        "[ 1  0  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  1  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  1  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  1  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  1  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  1  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  1  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  1  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  1  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  1  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  1  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  1]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  1]"
      ]
    }
  },
  {
    "Helper": 4,
    "Description": "mu_global_readout step 1: perform global readout",
    "Input size": 12,
    "Output size": 12,
    "Matrices": {
      "C": [
        "[ 1  0  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  1  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  1  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  1  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  1  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  1  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  1  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  1  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  1  0  1  0]",
        "[ 0  0  0  0  0  0  0  0  0  1  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
        "[ 0  0  0  0  0  0  0  0  0  0  0  0]"
```

```
        ],
        "R": [
          "[ 0  0  0  0  0  0  0  0  0  0  0  1]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0 -1]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  0]"
        ]
      }
    },
    {
      "Helper": 5,
      "Description": "mu_global_readout step 2: negate the result",
      "Input size": 12,
      "Output size": 12,
      "Matrices": {
        "C": [
          "[ 1  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  1  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  1  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  1  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  1  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  1  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  1  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  1  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  1  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  1  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  1  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0 -1]"
        ],
        "b": "[ 0  0  0  0  0  0  0  0  0  0  0  1]"
      }
    },
    {
      "Helper": 6,
      "Description": "mu_global_readout step 3: check readiness",
      "Input size": 12,
      "Output size": 12,
      "Matrices": {
        "C": [
          "[ 1  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  1  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  1  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  1  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  1  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  1  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  1  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  1  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  1  0  0  0]",
```

```
            "[ 0  0  0  0  0  0  0  0  1  0  0  0]",
            "[ 0  0  0  0  0  0  0  0  0  1  0  1]",
            "[ 0  0  0  0  0  0  0  0  0  0  1  0]",
            "[ 0  0  0  0  0  0  0  0  0  0  0  1]"
          ],
          "b": "[ 0  0  0  0  0  0  0  0  0  0  0 -1]"
        }
      },
      {
        "Helper": 7,
        "Description": "mu_global_readout step 4:
                        set result based on readiness",
        "Input size": 12,
        "Output size": 12,
        "Matrices": {
          "C": [
            "[ 1  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  1  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  1  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  1  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  1  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  1  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  1  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  0  1  0  0  0  0]",
            "[ 0  0  0  0  0  0  0  0  1  0  0  0]",
            "[ 0  0  0  0  0  0  0  0  0  1  1  0]",
            "[ 0  0  0  0  0  0  0  0  0  0  1  0]",
            "[ 0  0  0  0  0  0  0  0  0  0  0  1]"
          ],
          "b": "[ 0  0  0  0  0  0  0  0  0  0 -1  0]"
        }
      },
      {
        "Helper": 8,
        "Description": "mu_global_readout step 5: update fixpoint relation",
        "Input size": 12,
        "Output size": 12,
        "Matrices": {
          "C": [
            "[ 1  0  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  1  0  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  1  0  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  1  0  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  1  0  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  1  0  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  1  0  0  0  0  0]",
            "[ 0  0  0  0  0  0  0  1  0  0  0  0]",
            "[ 0  0  0  0  0  0  0  0  1  0  0  0]",
            "[ 0  0  0  0  0  0  0  0  0  1  0  0]",
            "[ 0  0  0  0  1  0  0  0  0  0  1  0]",
            "[ 0  0  0  0  0  0  0  0  0  0  0  1]"
          ]
        }
      }
    ]
```

```
    },
    {
      "Number": 4,
      "Operator": "final_layer(classification_feature_idx=10)",
      "Input size": 12,
      "Output size": 12,
      "Matrices": {
        "C": [
          "[ 1  0  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  1  0  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  1  0  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  1  0  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  1  0  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  1  0  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  1  0  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  1  0  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  1  0  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  1  0  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  1  0]",
          "[ 0  0  0  0  0  0  0  0  0  0  0  1]"
        ]
      }
    }
  ]
}
```