RWTH AACHEN UNIVERSITY

Logic and the Power of Recurrent Graph Neural Networks

Martin Grohe

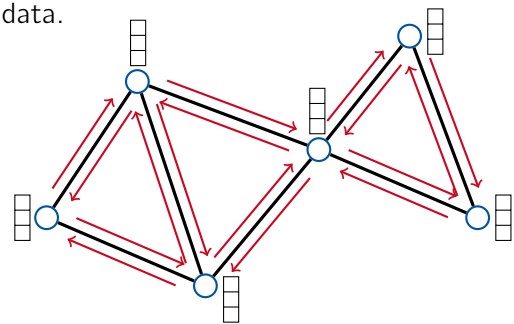
Graph Neural Networks

Graph Neural Networks

- Graph neural networks (GNNs) are deep learning architectures for machine learning problems on graphs.

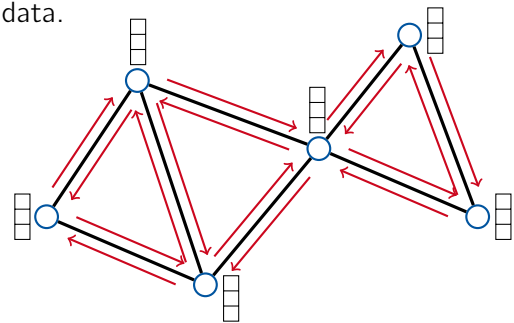
Graph Neural Networks

- Graph neural networks (GNNs) are deep learning architectures for machine learning problems on graphs.
- A GNN may be viewed as distributed message passing algorithm operating on the vertices of its input graph. The algorithm is controlled by parameters typically learned from data.



Graph Neural Networks

- Graph neural networks (GNNs) are deep learning architectures for machine learning problems on graphs.
- A GNN may be viewed as distributed message passing algorithm operating on the vertices of its input graph. The algorithm is controlled by parameters typically learned from data.



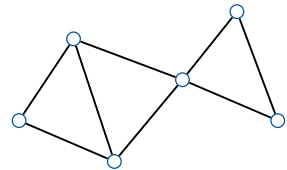
- In this talk, we study the expressiveness of GNNs:

  Which functions on graphs can be computed by a GNN?

Computation of GNNs

GNN N with d layers maps graph G to sequence of signals

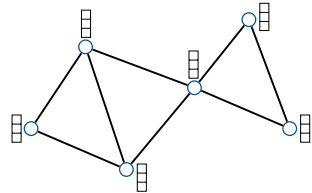
$$\mathcal{I}^{(t)} : V(G) \to \mathbb{R}^{p_t} \quad \text{for } t = 0, \ldots, d$$



Computation of GNNs

GNN N with d layers maps graph G to sequence of signals

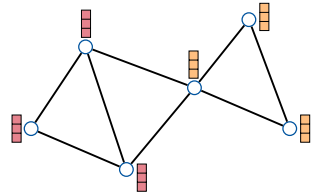
$$\mathcal{J}^{(t)} : V(G) \to \mathbb{R}^{p_t} \quad \text{for } t = 0, \ldots, d$$



Computation of GNNs

GNN N with d layers maps graph G to sequence of signals

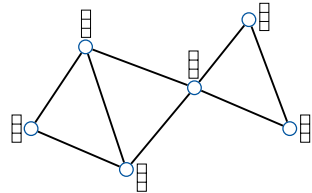
$$\mathcal{I}^{(t)} : V(G) \to \mathbb{R}^{p_t} \quad \text{for } t = 0, \ldots, d$$



Computation of GNNs

GNN N with d layers maps graph G to sequence of signals

$$\mathcal{I}^{(t)} : V(G) \to \mathbb{R}^{p_t} \quad \text{for } t = 0, \ldots, d$$

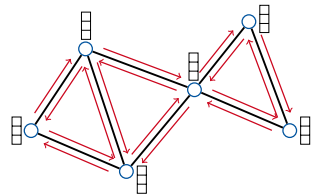


Initialisation: $\mathcal{I}^{(0)}(v) \in \mathbb{R}^{p_0}$ encodes node labels or initial signal on the graph.

Computation of GNNs

GNN N with d layers maps graph G to sequence of signals

$$\mathcal{I}^{(t)} : V(G) \to \mathbb{R}^{p_t} \quad \text{for } t = 0, \ldots, d$$



Initialisation: $\mathcal{I}^{(0)}(v) \in \mathbb{R}^{p_0}$ encodes node labels or initial signal on the graph.

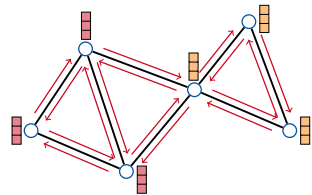
Aggregation: $\mathcal{A}^{(t)}(v) := \mathrm{agg}_t\Big(\big\{\!\!\big\{ \mathcal{I}^{(t-1)}(w) \;\big|\; w \in N_G(v) \big\}\!\!\big\}\Big)$

- agg_t aggregation function: coordinatewise sum, mean or max

Computation of GNNs

GNN N with d layers maps graph G to sequence of signals

$$\mathcal{I}^{(t)} : V(G) \to \mathbb{R}^{p_t} \quad \text{for } t = 0, \ldots, d$$



Initialisation: $\mathcal{I}^{(0)}(v) \in \mathbb{R}^{p_0}$ encodes node labels or initial signal on the graph.

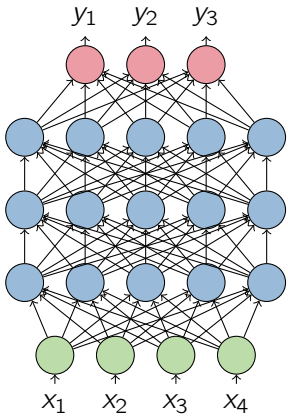
Aggregation: $\mathcal{A}^{(t)}(v) := \mathrm{agg}_t\Big(\big\{\!\!\big\{ \mathcal{I}^{(t-1)}(w) \;\big|\; w \in N_G(v) \big\}\!\!\big\}\Big)$

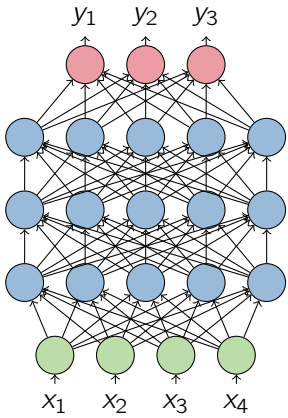
- agg_t aggregation function: coordinatewise sum, mean or max

Combination: $\mathcal{I}^{(t)}(v) := \mathrm{comb}_t\big(\mathcal{I}^{(t-1)}(v), \mathcal{A}^{(t)}(v)\big)$

- $\mathrm{comb}_t : \mathbb{R}^{2p_{t-1}} \to \mathbb{R}^{p_t}$ combination function computed by a feedforward neural network (a.k.a. multilayer perceptron)

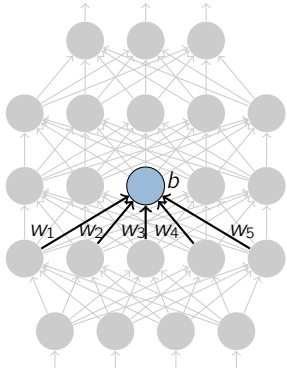
Feedforward Neural Networks





Feedforward Neural Networks

- Nodes and edges are weighted.



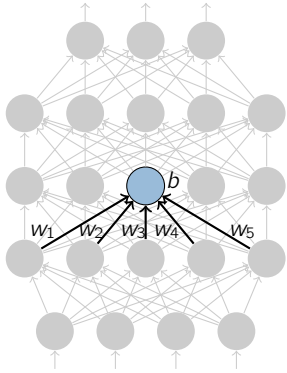
Feedforward Neural Networks

- Nodes and edges are weighted.
- Node with weight $b \in \mathbb{R}$ and incoming edges with weights $w_1, \ldots, w_k \in \mathbb{R}$ computes function

$$x_1, \ldots, x_k \mapsto \sigma\left(b + \sum_{i=1}^{k} w_i x_i\right),$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function.

Feedforward Neural Networks

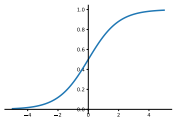


- Nodes and edges are weighted.
- Node with weight $b \in \mathbb{R}$ and incoming edges with weights $w_1, \ldots, w_k \in \mathbb{R}$ computes function

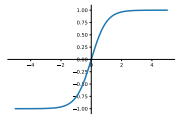
$$x_1, \ldots, x_k \mapsto \sigma\left(b + \sum_{i=1}^{k} w_i x_i\right),$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function.

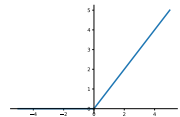
Typical activation functions are:



sigmoid

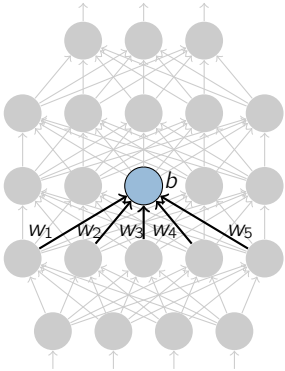


hyperbolic tangent



rectified linear unit (ReLU)

Feedforward Neural Networks



- Nodes and edges are weighted.
- Node with weight $b \in \mathbb{R}$ and incoming edges with weights $w_1, \ldots, w_k \in \mathbb{R}$ computes function

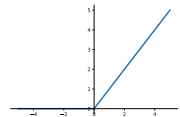
$$x_1, \ldots, x_k \mapsto \sigma\left(b + \sum_{i=1}^{k} w_i x_i\right),$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function.

In this talk, we always use

$$\mathrm{ReLu}(x) \coloneqq \max\{0, x\}$$

as activation function.



rectified linear unit (ReLU)

Universal Approximation Theorem

Theorem (Cybenko 1989, Hornik 1991)

Let $f : K \to \mathbb{R}^n$ be a continuous function defined on a compact set $K \subseteq \mathbb{R}^m$. Then for every $\epsilon > 0$ there is a feedforward neural network N computing a function f_N such that

$$\sup_{\mathbf{x} \in K} \|f(\mathbf{x}) - f_N(\mathbf{x})\| < \epsilon.$$

Computation of GNNs (revisited)

GNN N with d layers maps graph G to sequence of signals

$$\mathcal{I}^{(t)} : V(G) \to \mathbb{R}^{p_t} \quad \text{for } t = 0, \ldots, d$$

Initialisation: $\mathcal{I}^{(0)}(v) \in \mathbb{R}^{p_0}$ encodes node labels or initial signal on the graph.

Aggregation: $\mathcal{A}^{(t)}(v) := \mathrm{agg}_t\Big(\big\{\!\!\big\{ \mathcal{I}^{(t-1)}(w) \;\big|\; w \in N_G(v) \big\}\!\!\big\}\Big)$

- agg_t aggregation function: coordinatewise sum, mean or max

Combination: $\mathcal{I}^{(t)}(v) := \mathrm{comb}_t\big(\mathcal{I}^{(t-1)}(v), \mathcal{A}^{(t)}(v)\big)$

- $\mathrm{comb}_t : \mathbb{R}^{2p_{t-1}} \to \mathbb{R}^{p_t}$ combination function computed by a feedforward neural network (a.k.a. multilayer perceptron)

Global Aggregation

$$\mathscr{g}^{(t)} := \mathrm{agg}'_t\big(\{\!\{\mathscr{I}^{(t-1)}(w) \mid w \in V(G)\}\!\}\big),$$

where the aggregation function agg'_t is coordinatewise sum, mean or max.

Global Aggregation

$$\mathscr{g}^{(t)} \coloneqq \mathrm{agg}'_t\big(\{\!\{\mathscr{z}^{(t-1)}(w) \mid w \in V(G)\}\!\}\big),$$

where the aggregation function agg'_t is coordinatewise sum, mean or max.

Then combination becomes

$$\mathscr{z}^{(t)}(v) \coloneqq \mathrm{comb}_t\big(\mathscr{z}^{(t-1)}(v), \mathscr{a}^{(t)}(v), \mathscr{g}^{(t)}\big)$$

with $\mathrm{comb}_t : \mathbb{R}^{3p_{t-1}} \to \mathbb{R}^{p_t}$ computed by feedforward neural network.

Global Aggregation

$$\mathscr{g}^{(t)} \coloneqq \mathrm{agg}'_t\big(\{\!\{\mathscr{j}^{(t-1)}(w) \mid w \in V(G)\}\!\}\big),$$

where the aggregation function agg'_t is coordinatewise sum, mean or max.

Then combination becomes

$$\mathscr{j}^{(t)}(v) \coloneqq \mathrm{comb}_t\big(\mathscr{j}^{(t-1)}(v), \mathscr{a}^{(t)}(v), \mathscr{g}^{(t)}\big)$$

with $\mathrm{comb}_t : \mathbb{R}^{3p_{t-1}} \to \mathbb{R}^{p_t}$ computed by feedforward neural network.

Random Initialisation

$\mathscr{j}^{(0)}(v) = (s_1, \ldots, s_{p_0}, s_{p_0+1}) \in \mathbb{R}^{p_0+1}$ where $(s_1, \ldots, s_{p_0})$ endodes node labels or initial signal and s_{p_0+1} is chosen uniformly at random from $[0, 1]$.

Global Aggregation

$$\mathscr{g}^{(t)} \coloneqq \mathrm{agg}'_t\big(\{\!\{\mathscr{I}^{(t-1)}(w) \mid w \in V(G)\}\!\}\big),$$

where the aggregation function agg'_t is coordinatewise sum, mean or max.

Then combination becomes

$$\mathscr{I}^{(t)}(v) \coloneqq \mathrm{comb}_t\big(\mathscr{I}^{(t-1)}(v), \mathscr{a}^{(t)}(v), \mathscr{g}^{(t)}\big)$$

with $\mathrm{comb}_t : \mathbb{R}^{3p_{t-1}} \to \mathbb{R}^{p_t}$ computed by feedforward neural network.

Random Initialisation

$\mathscr{I}^{(0)}(v) = (s_1, \ldots, s_{p_0}, s_{p_0+1}) \in \mathbb{R}^{p_0+1}$ where $(s_1, \ldots, s_{p_0})$ endodes node labels or initial signal and s_{p_0+1} is chosen uniformly at random from $[0, 1]$.

In this talk, we always consider GNNs with global aggregation. We sometimes consider random initialisation, but mention it explicitly when we do.

Node-Level Functions

GNNs compute a function $\mathscr{f}$ that maps each graph G (possibly with node labels or an initial signal) to a signal $\mathscr{f}(G, \cdot) : V(G) \to \mathbb{R}^q$ defined by

$$\mathscr{f}(G, v) = \mathscr{z}^{(d)}(v).$$

Functions Computed by GNNs

Node-Level Functions

GNNs compute a function $\mathscr{f}$ that maps each graph G (possibly with node labels or an initial signal) to a signal $\mathscr{f}(G, \cdot) : V(G) \to \mathbb{R}^q$ defined by

$$\mathscr{f}(G, v) = \mathscr{z}^{(d)}(v).$$

Graph-Level Functions

To compute a function $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ mapping graphs to vectors of real numbers,

Functions Computed by GNNs

Node-Level Functions

GNNs compute a function $\mathscr{f}$ that maps each graph G (possibly with node labels or an initial signal) to a signal $\mathscr{f}(G, \cdot) : V(G) \to \mathbb{R}^q$ defined by

$$\mathscr{f}(G, v) = \mathscr{z}^{(d)}(v).$$

class of all graphs

Graph-Level Functions

To compute a function $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ mapping graphs to vectors of real numbers,

Functions Computed by GNNs

Node-Level Functions

GNNs compute a function $\mathscr{f}$ that maps each graph G (possibly with node labels or an initial signal) to a signal $\mathscr{f}(G, \cdot) : V(G) \to \mathbb{R}^q$ defined by

$$\mathscr{f}(G, v) = \mathscr{z}^{(d)}(v).$$

Graph-Level Functions

To compute a function $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ mapping graphs to vectors of real numbers, we aggregate the values and then apply a readout function:

$$\mathscr{F}(G) = \mathrm{ro}\Big(\mathrm{agg}\Big(\Big\{\!\!\Big\{ \mathscr{z}^{(d)}(v) \;\Big|\; v \in V(G) \Big\}\!\!\Big\}\Big)\Big).$$

- agg aggregation function: sum, mean or max
- $\mathrm{ro} : \mathbb{R}^p \to \mathbb{R}^q$ readout function computed by a feedforward neural network

Invariance

In this talk, we focus on graph-level functions.

In this talk, we focus on graph-level functions.

Invariance

Let $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ be computed by a GNN. Then for all isomorphic graphs G, H,

$$\mathscr{F}(G) = \mathscr{F}(H).$$

In this talk, we focus on graph-level functions.

Invariance

Let $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ be computed by a GNN. Then for all isomorphic graphs G, H,

$$\mathscr{F}(G) = \mathscr{F}(H).$$

Graph Properties

When comparing GNNs with logic and circuit complexity, we often look at (isomorphism invariant) Boolean functions $\mathscr{P} : \mathscr{G} \to \{0, 1\}$, that is, properties of graphs.

In this talk, we focus on graph-level functions.

Invariance

Let $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ be computed by a GNN. Then for all isomorphic graphs G, H,

$$\mathscr{F}(G) = \mathscr{F}(H).$$

Graph Properties

When comparing GNNs with logic and circuit complexity, we often look at (isomorphism invariant) Boolean functions $\mathscr{P} : \mathscr{G} \to \{0, 1\}$, that is, properties of graphs.

Remark

GNNs with random initialisation compute an isomorphism invariant random variable.

In this talk, we focus on graph-level functions.

Invariance

Let $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ be computed by a GNN. Then for all isomorphic graphs G, H,

$$\mathscr{F}(G) = \mathscr{F}(H).$$

Graph Properties

When comparing GNNs with logic and circuit complexity, we often look at (isomorphism invariant) Boolean functions $\mathscr{P} : \mathscr{G} \to \{0, 1\}$, that is, properties of graphs.

Remark

GNNs with random initialisation compute an isomorphism invariant random variable.

To compute a function, we can run the GNN several times with independent random initialisations and then take the average value, or for Boolean functions, the majority.

The Logic of GNNs

2-Variable Counting Logic

$\left.\begin{matrix} C^2 \\ FO^2{+}C \end{matrix}\right\}$ 2-variable fragments of first-order logic with counting quantifiers $\exists^{\geq n} x \varphi$

2-Variable Counting Logic

$\left.\begin{matrix} C^2 \\ FO^2{+}C \end{matrix}\right\}$ 2-variable fragments of first-order logic with counting quantifiers $\exists^{\geq n} x \varphi$

Difference:

- in C^2, the number n in a counting quantifier $\exists^{\geq n} x$ is a constant;

2-Variable Counting Logic

$\left.\begin{matrix} C^2 \\ FO^2{+}C \end{matrix}\right\}$ 2-variable fragments of first-order logic with counting quantifiers $\exists^{\geq n} x \varphi$

Difference:

- in C^2, the number n in a counting quantifier $\exists^{\geq n} x$ is a constant;
- in $FO^2{+}C$ it is a variable or term, and we allow arithmetic on such number terms.

2-Variable Counting Logic

$\left.\begin{matrix} C^2 \\ FO^2{+}C \end{matrix}\right\}$ 2-variable fragments of first-order logic with counting quantifiers $\exists^{\geq n} x \varphi$

Difference:

- in C^2, the number n in a counting quantifier $\exists^{\geq n} x$ is a constant;
- in $FO^2{+}C$ it is a variable or term, and we allow arithmetic on such number terms.

Example

- A C^2-formula expressing that vertex x has degree 8:

$$\exists^{\geq 8} y E(x, y) \wedge \neg \exists^{\geq 9} y E(x, y).$$

2-Variable Counting Logic

$\left.\begin{matrix} C^2 \\ FO^2{+}C \end{matrix}\right\}$ 2-variable fragments of first-order logic with counting quantifiers $\exists^{\geq n} x \varphi$

Difference:

- in C^2, the number n in a counting quantifier $\exists^{\geq n} x$ is a constant;
- in $FO^2{+}C$ it is a variable or term, and we allow arithmetic on such number terms.

Example

- A C^2-formula expressing that vertex x has degree 8:

$$\exists^{\geq 8} y E(x,y) \wedge \neg \exists^{\geq 9} y E(x,y).$$

- An $FO^2{+}C$-formula expressing that vertex x has even degree:

$$\exists n \Big(\exists^{\geq 2n} y \, E(x,y) \wedge \neg \exists^{\geq 2n+1} y \, E(x,y) \Big).$$

C^2-Equivalence

For every logic L, two graphs are L-equivalent if they satisfy the same sentences of L.

C^2-Equivalence

For every logic L, two graphs are L-equivalent if they satisfy the same sentences of L.

Theorem (Immerman and Lander 1990, Tinhofer 1991, Dvorak 2010)

For all graphs G, H, the following are equivalent:

1. *G, H are C^2-equivalent.*
2. *G, H are FO^2+C-equivalent.*

C^2-Equivalence

For every logic L, two graphs are L-equivalent if they satisfy the same sentences of L.

Theorem (Immerman and Lander 1990, Tinhofer 1991, Dvorak 2010)

For all graphs G, H, the following are equivalent:

1. *G, H are C^2-equivalent.*
2. *G, H are FO^2+C-equivalent.*
3. *The Colour Refinement algorithm does not distinguish G, H.*

C^2-Equivalence

For every logic L, two graphs are L-equivalent if they satisfy the same sentences of L.

Theorem (Immerman and Lander 1990, Tinhofer 1991, Dvorak 2010)

For all graphs G, H, the following are equivalent:

1. *G, H are C^2-equivalent.*
2. *G, H are FO^2+C-equivalent.*
3. *The Colour Refinement algorithm does not distinguish G, H.*
4. *G and H are fractionally isomorphic, that is, there is a doubly stochastic matrix X such that $A_G X = X A_H$.*

C^2-Equivalence

For every logic L, two graphs are L-equivalent if they satisfy the same sentences of L.

Theorem (Immerman and Lander 1990, Tinhofer 1991, Dvorak 2010)

For all graphs G, H, the following are equivalent:

1. *G, H are C^2-equivalent.*
2. *G, H are FO^2+C-equivalent.*
3. *The Colour Refinement algorithm does not distinguish G, H.*
4. *G and H are fractionally isomorphic, that is, there is a doubly stochastic matrix X such that $A_G X = X A_H$.*
5. *For all trees T, the number of homomorphisms from T to G equals the number of homomorphisms from T to H.*

Colour Refinement

The Colour Refinement algorithm iteratively computes a colouring of the vertices of graph G.

Colour Refinement

The Colour Refinement algorithm iteratively computes a colouring of the vertices of graph G.

Initialisation All vertices get the same colour.

Colour Refinement

The Colour Refinement algorithm iteratively computes a colouring of the vertices of graph G.

Initialisation All vertices get the same colour.

Refinement Step Two nodes v, w get different colours if there is some colour c such that v and w have different numbers of neighbours of colour c.

Colour Refinement

The Colour Refinement algorithm iteratively computes a colouring of the vertices of graph G.

Initialisation All vertices get the same colour.

Refinement Step Two nodes v, w get different colours if there is some colour c such that v and w have different numbers of neighbours of colour c.

Refinement is repeated until colouring stays stable.

Colour Refinement

The Colour Refinement algorithm iteratively computes a colouring of the vertices of graph G.

Initialisation All vertices get the same colour.

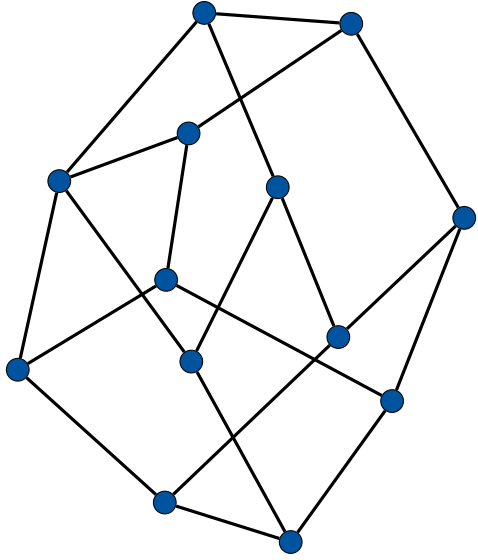
Refinement Step Two nodes v, w get different colours if there is some colour c such that v and w have different numbers of neighbours of colour c.

Refinement is repeated until colouring stays stable.

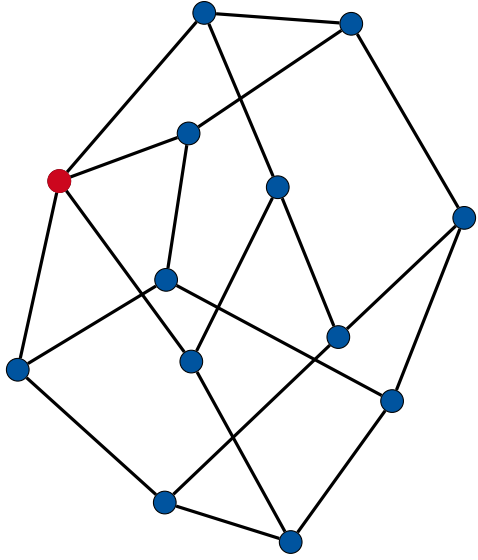
Remark

The algorithm is also refered to as naive vertex classification and as 1-dimensional Weisfeiler-Leman algorithm. One needs to be careful though, because there are two slightly different versions of the algorithm, and this difference does play a role in our work (though not in this talk).

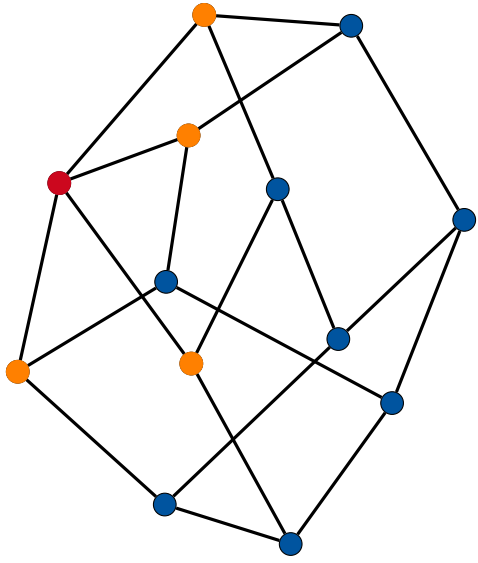
Example



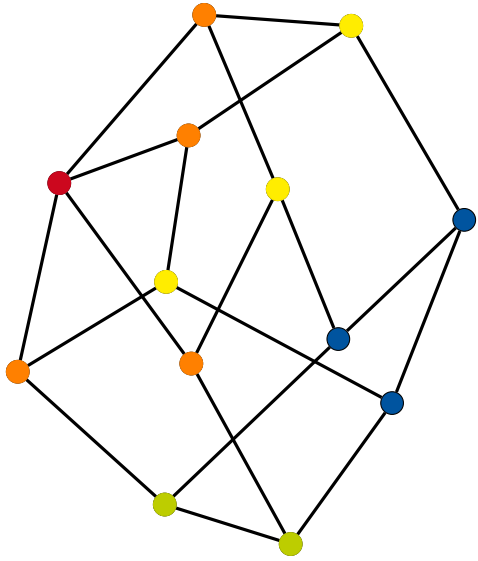
Example



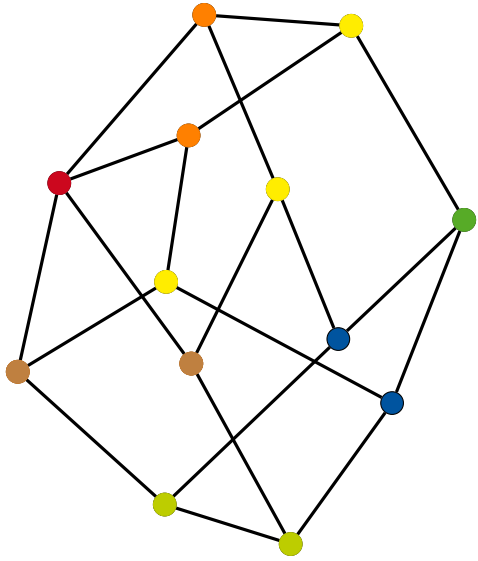
Example



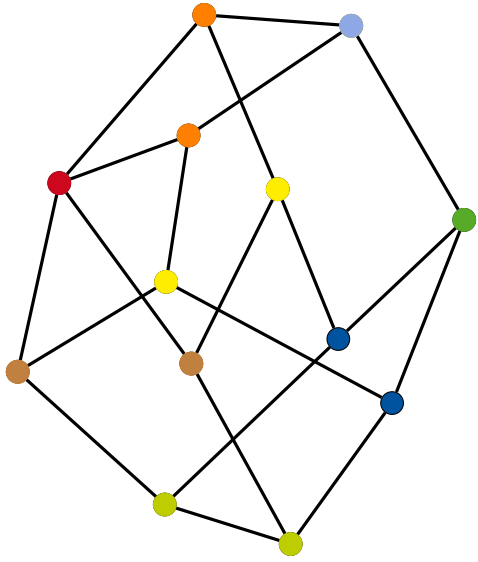
Example



Example



Example



Distinguishing Graphs by Colour Refinement

Colour Refinement distinguishes two graphs G, H if their colour histograms differ, that is, some colour appears a different number of times in G and H.

Distinguishing Graphs by Colour Refinement

Colour Refinement distinguishes two graphs G, H if their colour histograms differ, that is, some colour appears a different number of times in G and H.

Thus Colour Refinement can be used as an incomplete isomorphism test.

Distinguishing Graphs by Colour Refinement

Colour Refinement distinguishes two graphs G, H if their colour histograms differ, that is, some colour appears a different number of times in G and H.

Thus Colour Refinement can be used as an incomplete isomorphism test.

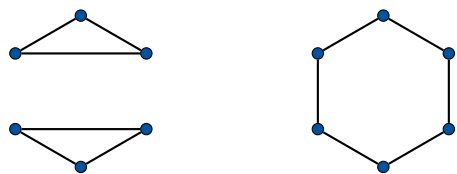
- works on almost all graphs (Babai, Erdös, Selkow 1980)

Distinguishing Graphs by Colour Refinement

Colour Refinement distinguishes two graphs G, H if their colour histograms differ, that is, some colour appears a different number of times in G and H.

Thus Colour Refinement can be used as an incomplete isomorphism test.

- works on almost all graphs (Babai, Erdös, Selkow 1980)
- fails on some very simple graphs:



The Distinguishing Power of GNNs

Theorem (Morris, Ritzert, Fey, Hamilton, Lenssen, Rattan, G. 2019, Xu, Hu, Leskovec, Jegelka 2019)

For all graphs G, H, the following are equivalent:

1. *G and H are C^2-equivalent.*
2. *G and H are not distinguishable by a GNN, that is, for all GNNs N we have $\mathscr{F}_N(G) = \mathscr{F}_N(H)$;*

The Distinguishing Power of GNNs

Theorem (Morris, Ritzert, Fey, Hamilton, Lenssen, Rattan, G. 2019, Xu, Hu, Leskovec, Jegelka 2019)

For all graphs G, H, the following are equivalent:

1. *G and H are C^2-equivalent.*
2. *G and H are not distinguishable by a GNN, that is, for all GNNs N we have $\mathscr{F}_N(G) = \mathscr{F}_N(H)$;*

Corollary

Every function $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ computable by a GNN is C^2-invariant, that is, if G, H are C^2-equivalent then $\mathscr{F}(G) = \mathscr{F}(H)$.

The Distinguishing Power of GNNs

Theorem (Morris, Ritzert, Fey, Hamilton, Lenssen, Rattan, G. 2019, Xu, Hu, Leskovec, Jegelka 2019)

For all graphs G, H, the following are equivalent:

1. *G and H are C^2-equivalent.*
2. *G and H are not distinguishable by a GNN, that is, for all GNNs N we have $\mathscr{F}_N(G) = \mathscr{F}_N(H)$;*

Corollary

Every function $\mathscr{F} : \mathscr{G} \to \mathbb{R}^q$ computable by a GNN is C^2-invariant, that is, if G, H are C^2-equivalent then $\mathscr{F}(G) = \mathscr{F}(H)$.

Remark

(Morris et al. 2019) ensure that graphs G, H that are not C^2-equivalent can be distinguished by a GNN N of size polynomial in $|G|, |H|$.

Expressing Graph Properties

Theorem (Barceló, Kostylev, Monet, Pérez, Reutter, Silva 2019)

Let $\mathscr{P}$ be a graph property expressible in the logic C^2. Then there is a GNN (with rational weights) computing $\mathscr{P}$.

Expressing Graph Properties

Theorem (Barceló, Kostylev, Monet, Pérez, Reutter, Silva 2019)

Let $\mathscr{P}$ be a graph property expressible in the logic C^2. Then there is a GNN (with rational weights) computing $\mathscr{P}$.

Theorem (G. 2023)

Let $\mathscr{P}$ be a graph property computable by a GNN with rational weights. Then $\mathscr{P}$ is expressible in in $\mathsf{FO}^2{+}\mathsf{C}$.

Theorem (Barceló, Kostylev, Monet, Pérez, Reutter, Silva 2019)

Let $\mathscr{P}$ be a graph property expressible in the logic C^2. Then there is a GNN (with rational weights) computing $\mathscr{P}$.

Theorem (G. 2023)

Let $\mathscr{P}$ be a graph property computable by a GNN with rational weights. Then $\mathscr{P}$ is expressible in in $\mathsf{FO}^2{+}\mathsf{C}$.

Corollary

$$\mathsf{C}^2 \quad \subset \quad \mathsf{GNN} \quad \subset \quad \mathsf{FO}^2{+}\mathsf{C}.$$

Theorem (Barceló, Kostylev, Monet, Pérez, Reutter, Silva 2019)

Let $\mathscr{P}$ be a graph property expressible in the logic C^2. Then there is a GNN (with rational weights) computing $\mathscr{P}$.

Theorem (G. 2023)

Let $\mathscr{P}$ be a graph property computable by a GNN with rational weights. Then $\mathscr{P}$ is expressible in in FO^2+C.

Corollary

$$\mathsf{C}^2 \quad \subset \quad \mathsf{GNN} \quad \subset \quad \mathsf{FO}^2\text{+C}.$$

It can be shown that both inclusions are strict.

Theorem (G. 2023)

For all graph properties $\mathscr{P}$, the following are equivalent.

1. *$\mathscr{P}$ is computable by a polynomial-size bounded-depth family of GNNs with random initialisation and with arbitrary real weights (that may use activations like sigmoid or tanh besides ReLU).*

Theorem (G. 2023)

For all graph properties $\mathscr{P}$, the following are equivalent.

1. *$\mathscr{P}$ is computable by a polynomial-size bounded-depth family of GNNs with random initialisation and with arbitrary real weights (that may use activations like sigmoid or tanh besides ReLU).*
2. *$\mathscr{P}$ is computable by a polynomial-size bounded-depth family of GNNs with random initialisation and with rational weights, using only sum aggregation.*

Theorem (G. 2023)

For all graph properties $\mathscr{P}$, the following are equivalent.

1. *$\mathscr{P}$ is computable by a polynomial-size bounded-depth family of GNNs with random initialisation and with arbitrary real weights (that may use activations like sigmoid or tanh besides ReLU).*
2. *$\mathscr{P}$ is computable by a polynomial-size bounded-depth family of GNNs with random initialisation and with rational weights, using only sum aggregation.*
3. *$\mathscr{P}$ is expressible in FO^2+C with built-in relations.*

Theorem (G. 2023)

For all graph properties $\mathscr{P}$, the following are equivalent.

1. *$\mathscr{P}$ is computable by a polynomial-size bounded-depth family of GNNs with random initialisation and with arbitrary real weights (that may use activations like sigmoid or tanh besides ReLU).*
2. *$\mathscr{P}$ is computable by a polynomial-size bounded-depth family of GNNs with random initialisation and with rational weights, using only sum aggregation.*
3. *$\mathscr{P}$ is expressible in $\mathsf{FO}^2{+}\mathsf{C}$ with built-in relations.*
4. *$\mathscr{P}$ is in the complexity class TC^0.*

Recurrent GNNs

Recurrent GNNs

So far, GNNs have a fixed number d of layers, and each layer t has its own functions agg_t, agg'_t, comb_t.

Recurrent GNNs

So far, GNNs have a fixed number d of layers, and each layer t has its own functions agg_t, agg'_t, comb_t.

Recurrent GNN Computation

A recurrent GNN has a single layer and applies it repeatedly. That is, we have aggregation functions $\mathrm{agg}, \mathrm{agg}'$ and a combination function comb and for $t \geq 1$ let:

$$\jmath^{(t)}(v) = \mathrm{comb}\Big(\jmath^{(t-1)}(v), \mathrm{agg}\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in N_G(v)\}\!\}\big), \mathrm{agg}'\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in V(G)\}\!\}\big)\Big)$$

So far, GNNs have a fixed number d of layers, and each layer t has its own functions agg_t, agg'_t, comb_t.

Recurrent GNN Computation

A recurrent GNN has a single layer and applies it repeatedly. That is, we have aggregation functions $\mathrm{agg}, \mathrm{agg}'$ and a combination function comb and for $t \geq 1$ let:

$$\mathcal{J}^{(t)}(v) = \mathrm{comb}\Big(\mathcal{J}^{(t-1)}(v), \mathrm{agg}\big(\{\!\{\mathcal{J}^{(t-1)}(w) \mid w \in N_G(v)\}\!\}\big), \mathrm{agg}'\big(\{\!\{\mathcal{J}^{(t-1)}(w) \mid w \in V(G)\}\!\}\big)\Big)$$

Termination and Readout

- Each node has flag (say, the last coordinate in $\mathcal{J}^{(t)}(v)$) indicating whether the local computation at the node is complete.

So far, GNNs have a fixed number d of layers, and each layer t has its own functions agg_t, agg'_t, comb_t.

Recurrent GNN Computation

A recurrent GNN has a single layer and applies it repeatedly. That is, we have aggregation functions $\mathrm{agg}, \mathrm{agg}'$ and a combination function comb and for $t \geq 1$ let:

$$\jmath^{(t)}(v) = \mathrm{comb}\Big(\jmath^{(t-1)}(v), \mathrm{agg}\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in N_G(v)\}\!\}\big), \mathrm{agg}'\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in V(G)\}\!\}\big)\Big)$$

Termination and Readout

- Each node has flag (say, the last coordinate in $\jmath^{(t)}(v)$) indicating whether the local computation at the node is complete.
- The global computation halts once every node has set its flag.

So far, GNNs have a fixed number d of layers, and each layer t has its own functions agg_t, agg'_t, comb_t.

Recurrent GNN Computation

A recurrent GNN has a single layer and applies it repeatedly. That is, we have aggregation functions $\mathrm{agg}, \mathrm{agg}'$ and a combination function comb and for $t \geq 1$ let:

$$\jmath^{(t)}(v) = \mathrm{comb}\Big(\jmath^{(t-1)}(v), \mathrm{agg}\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in N_G(v)\}\!\}\big), \mathrm{agg}'\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in V(G)\}\!\}\big)\Big)$$

Termination and Readout

- Each node has flag (say, the last coordinate in $\jmath^{(t)}(v)$) indicating whether the local computation at the node is complete.
- The global computation halts once every node has set its flag.
- At that point, the signal is aggregated and a readout functions is applied.

Recurrent GNNs

So far, GNNs have a fixed number d of layers, and each layer t has its own functions agg_t, agg'_t, comb_t.

Recurrent GNN Computation

A recurrent GNN has a single layer and applies it repeatedly. That is, we have aggregation functions $\mathrm{agg}, \mathrm{agg}'$ and a combination function comb and for $t \geq 1$ let:

$$\jmath^{(t)}(v) = \mathrm{comb}\Big(\jmath^{(t-1)}(v), \mathrm{agg}\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in N_G(v)\}\!\}\big), \mathrm{agg}'\big(\{\!\{\jmath^{(t-1)}(w) \mid w \in V(G)\}\!\}\big)\Big)$$

Termination and Readout

- Each node has flag (say, the last coordinate in $\jmath^{(t)}(v)$) indicating whether the local computation at the node is complete.
- The global computation halts once every node has set its flag.
- At that point, the signal is aggregated and a readout functions is applied.

We do not require convergence of the signals.

Universality of Recurrent GNNs

Theorem (Rosenbluth and G. 2025+)

Let $\mathscr{F} : \mathscr{G} \to \mathbb{Q}^n$ be computable. Then the following are equivalent.

(i) *$\mathscr{F}$ is C^2-invariant.*

(ii) *$\mathscr{F}$ is computable by a recurrent GNN.*

Eran Rosenbluth

Universality of Recurrent GNNs

Theorem (Rosenbluth and G. 2025+)

Let $\mathscr{F} : \mathscr{G} \to \mathbb{Q}^n$ be computable. Then the following are equivalent.

(i) *$\mathscr{F}$ is C^2-invariant.*

(ii) *$\mathscr{F}$ is computable by a recurrent GNN.*

Eran Rosenbluth

Furthermore,

- *we may choose the GNN in (ii) to have rational weights and to only use SUM aggregation;*
- *if $\mathscr{F}$ is computable in polynomial time then the GNN stops after polynomially many iterations and only uses internal states of polynomial bitlength.*

Corollary

Every computable function $\mathscr{F} : \mathscr{G} \to \mathbb{Q}^n$ is computable by a recurrent GNN with random initialisation only using rational weight and SUM aggregation.

Corollary

Every computable function $\mathscr{F} : \mathscr{G} \to \mathbb{Q}^n$ is computable by a recurrent GNN with random initialisation only using rational weight and SUM aggregation.

Furthermore, if $\mathscr{F}$ is computable in polynomial time then the GNN runs in polynomial time and only uses internal states of polynomial bitlength.

Corollary

Every computable function $\mathscr{F} : \mathscr{G} \to \mathbb{Q}^n$ is computable by a recurrent GNN with random initialisation only using rational weight and SUM aggregation.

Furthermore, if $\mathscr{F}$ is computable in polynomial time then the GNN runs in polynomial time and only uses internal states of polynomial bitlength.

Remarks

- Global aggregation simplifies the theorem and its proof. We also have a version without global aggregation, but it requires to pass the size of the graph as part of the initial signal, and it only applies to connected graphs.

Corollary

Every computable function $\mathscr{F} : \mathscr{G} \to \mathbb{Q}^n$ is computable by a recurrent GNN with random initialisation only using rational weight and SUM aggregation.

Furthermore, if $\mathscr{F}$ is computable in polynomial time then the GNN runs in polynomial time and only uses internal states of polynomial bitlength.

Remarks

- Global aggregation simplifies the theorem and its proof. We also have a version without global aggregation, but it requires to pass the size of the graph as part of the initial signal, and it only applies to connected graphs.
- There are also a versions of the results for node-level functions (actually, we prove these first and derive the graph-level results as a consequence).

Proof of the Theorem (Outline)

We want to compute $\mathscr{F}(G)$ for some graph G by a recurrent GNN.

Proof of the Theorem (Outline)

We want to compute $\mathscr{F}(G)$ for some graph G by a recurrent GNN.

Phase 1: Computing the Colours

We simulate colour refinement. After Phase 1, the signal at every node v holds a representation D_v of v's its colour.

Proof of the Theorem (Outline)

We want to compute $\mathscr{F}(G)$ for some graph G by a recurrent GNN.

Phase 1: Computing the Colours

We simulate colour refinement. After Phase 1, the signal at every node v holds a representation D_v of v's its colour.

Phase 2: Inversion

Locally at every node v, from the colour D_v we compute a graph G_v that is C^2-equivalent to the input graph G.

Proof of the Theorem (Outline)

We want to compute $\mathscr{F}(G)$ for some graph G by a recurrent GNN.

Phase 1: Computing the Colours

We simulate colour refinement. After Phase 1, the signal at every node v holds a representation D_v of v's its colour.

Phase 2: Inversion

Locally at every node v, from the colour D_v we compute a graph G_v that is C^2-equivalent to the input graph G.

Phase 3: Computation of $\mathscr{F}$

Locally at every node v, we compute $\mathscr{F}(G_v)$. Since $\mathscr{F}$ is C^2-invariant, we have $\mathscr{F}(G_v) = \mathscr{F}(G)$.

Proof of the Theorem (Outline)

We want to compute $\mathscr{F}(G)$ for some graph G by a recurrent GNN.

Phase 1: Computing the Colours

We simulate colour refinement. After Phase 1, the signal at every node v holds a representation D_v of v's its colour.

Phase 2: Inversion

Locally at every node v, from the colour D_v we compute a graph G_v that is C^2-equivalent to the input graph G.

Phase 3: Computation of $\mathscr{F}$

Locally at every node v, we compute $\mathscr{F}(G_v)$. Since $\mathscr{F}$ is C^2-invariant, we have $\mathscr{F}(G_v) = \mathscr{F}(G)$.

Thus after Phase 3, every node holds the correct function value, and readout becomes trivial.

Phase 3: Computation of $\mathscr{F}$

Locally at every node v, we need to compute $\mathscr{F}(G_v)$ from the graph G_v that is encoded in ths state of node v after Phase 2.

Phase 3: Computation of $\mathscr{F}$

Locally at every node v, we need to compute $\mathscr{F}(G_v)$ from the graph G_v that is encoded in ths state of node v after Phase 2.

Local computation of a GNN at a node that ignores the message passing is essentially the computation of a recurrent feedforward neural network.

Phase 3: Computation of $\mathscr{F}$

Locally at every node v, we need to compute $\mathscr{F}(G_v)$ from the graph G_v that is encoded in ths state of node v after Phase 2.

Local computation of a GNN at a node that ignores the message passing is essentially the computation of a recurrent feedforward neural network.

Thus we can use the following result

Theorem (Siegelmann and Sontag 1992)

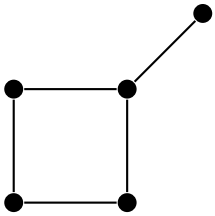
Every computable function $f : \mathbb{Q}^m \to \mathbb{Q}^n$ is computable by a recurrent feedforward neural network.

Phase 1: Computing the Colours

What are the “colours”?

Phase 1: Computing the Colours

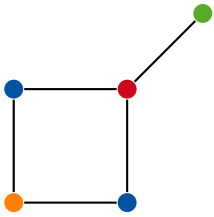
What are the “colours”?



Graph G

Phase 1: Computing the Colours

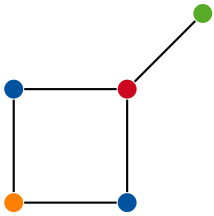
What are the “colours”?



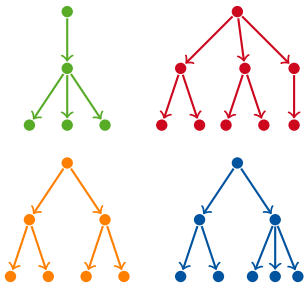
Graph G
coloured

Phase 1: Computing the Colours

What are the “colours”?



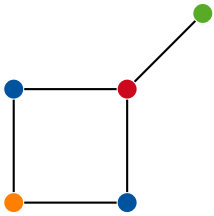
Graph G
coloured



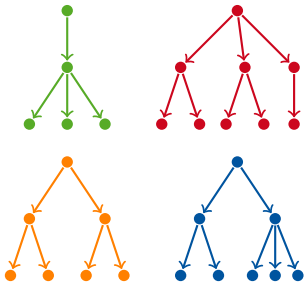
Tree representation

Phase 1: Computing the Colours

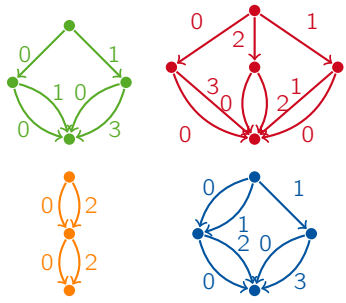
What are the "colours"?



Graph G coloured



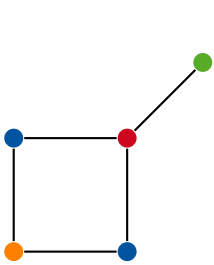
Tree representation



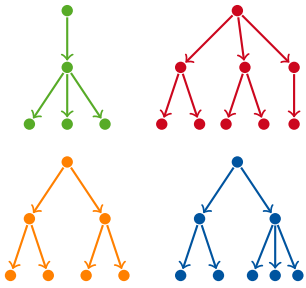
Compact dag representation

Phase 1: Computing the Colours

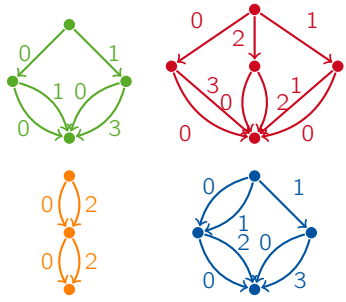
What are the "colours"?



Graph G coloured



Tree representation



Compact dag representation

To simulate colour refinement by a recurrent GNN, we encode the compact dag representation of colours by rational numbers and in several rounds by decreasing c for each node v count the number of neighbours of v of colour c.

The following lemma is an easy adaptation of a result on "inverting C^2-invariants" due to Martin Otto.

The following lemma is an easy adaptation of a result on "inverting C^2-invariants" due to Martin Otto.

Lemma (Otto 1997)

There is a polynomial time algorithm that, given the dag representation D_v of the colour of a node v in a graph G, computes a graph G_v that is C^2-equivalent to G.

Concluding Remarks

Concluding Remarks

- Our results on recurrent GNNs necessarily require rationals of unbounded precision in the (internal) signals computed by the GNN, even if we are only interested in computing a Boolean function.

Concluding Remarks

- Our results on recurrent GNNs necessarily require rationals of unbounded precision in the (internal) signals computed by the GNN, even if we are only interested in computing a Boolean function.

  It would be interesting, to understand the precise bitlength required. For example, which functions can we compute with logarithmic bitlength (in the size of the input graph)?

Concluding Remarks

- Our results on recurrent GNNs necessarily require rationals of unbounded precision in the (internal) signals computed by the GNN, even if we are only interested in computing a Boolean function.

  It would be interesting, to understand the precise bitlength required. For example, which functions can we compute with logarithmic bitlength (in the size of the input graph)?

- Is there a version of our main result on recurrent GNNs for higher-order recurrent GNNs and functions invariant under the corresponding higher-order Weisfeiler-Leman algorithm?

Concluding Remarks

- Our results on recurrent GNNs necessarily require rationals of unbounded precision in the (internal) signals computed by the GNN, even if we are only interested in computing a Boolean function.

  It would be interesting, to understand the precise bitlength required. For example, which functions can we compute with logarithmic bitlength (in the size of the input graph)?
- Is there a version of our main result on recurrent GNNs for higher-order recurrent GNNs and functions invariant under the corresponding higher-order Weisfeiler-Leman algorithm?
- We have a good understanding of the expressiveness of GNNs. But expressiveness results only tell half the story, because they ignore learning. However, many of the results presented here have good experimental support.

Concluding Remarks

- Our results on recurrent GNNs necessarily require rationals of unbounded precision in the (internal) signals computed by the GNN, even if we are only interested in computing a Boolean function.

  It would be interesting, to understand the precise bitlength required. For example, which functions can we compute with logarithmic bitlength (in the size of the input graph)?
- Is there a version of our main result on recurrent GNNs for higher-order recurrent GNNs and functions invariant under the corresponding higher-order Weisfeiler-Leman algorithm?
- We have a good understanding of the expressiveness of GNNs. But expressiveness results only tell half the story, because they ignore learning. However, many of the results presented here have good experimental support.

If you are looking for a postdoc position, please contact me.

A Few References

Grohe, M. (2021). The Logic of Graph Neural Networks.
In: Proc. LICS 2021.
arXiv:2104.14624

Grohe, M. (2024). The Descriptive Complexity of Graph Neural Networks.
In: TheoretiCS 3(25), 2024.
arXiv:2303.04613

Rosenbluth, E. and Grohe, M. (2025) Repetition Makes Perfect: Recurrent Sum-GNNs Match Message Passing Limit.
arXiv:2505.00291