# 4 Structured Data

In this chapter, we show methods and techniques that learn models for structured data in resource-aware environments. In practice, data models can often be structured as a graph where different data points are represented as nodes and the relationship between data points is captured by edges. Graphs occur in many applications because they serve well to represent objects of the physical world as compositions of parts. Molecules, for instance, can be described by a graph where the atoms are represented by nodes and their bonds by the edges. Another example are mathematical formulas, whose composition is semantically well modeled by graphs (see Section 4.5). Moreover, interactions between nodes of a graph can even be structured over time leading to spatio-temporal probabilistic graphs (see Section 4.1).

Once a particular type of a graph model is determined the models can be trained to do machine learning tasks such as classifying graphs or, when we interpret a graph as a transitional system, predicting the probability of a change from one state to another. The learning methods we use in this chapter can be divided mainly into *discriminative* Graph Neural Networks (GNNs) and *generative* Random Fields. GNNs use a learning approach that is derived from Convolutional Neural Networks (CNNs) by aggregating information of the neighborhood of each node through a message passing function (see Sections 4.2, 4.3, 4.5). Random Fields are a probabilistic model that captures the dependencies between multiple random variables and is trained to answer queries for a probability of event $A$ under the condition that event $B$ already happened (see Section 4.1). GNNs and Random Fields are different methods but both can be used to express the same kind of problems. For example, to infer conditional probabilities for each event we can use multiple GNNs in a layered approach [376]. However, the way in which they take care of the computational resources is rather different.

Here is an overview of this chapter. In Section 4.1, a new model is proposed to train spatio-temporal networks with Random Fields called the *Spatio-Temporal Random Field*. This model reduces the memory consumption without loss of the accuracy through a theoretically well based universal reparameterization. In Section 4.2, the *Weisfeiler-Leman algorithm* is explained with a focus on theoretical runtimes and the scalability of the algorithm. Then, the connection between the Weisfeiler-Leman algorithm and learning methods using graph kernels and GNNs, is surveyed. In Section 4.3, a unified framework for differentiable message passing in GNNs is introduced, and techniques for increasing its scalability are proposed. Section 4.4 proposes a framework to compute cuts in directed graphs with high quality, which scales well in shared memory and can be used in semi-supervised learning as well as in data compression. Section 4.5 presents a new technique to search for scientific papers, which uses mathematical formulas instead of words. A GNN is trained on a huge dataset extracted from arXiv and it is shown that the model scales well in practice.


https://doi.org/10.1515/9783110785944-004

## 4.1 Spatio-Temporal Random Fields

*Nico Piatkowski*
*Katharina Morik*

**Abstract:** Parameter sharing is a key technique in various state-of-the-art machine learning approaches. The underlying idea is simple yet effective. Given a highly over-parametrized model whose input data obeys some repetitive structure, multiple subsets of parameters are tied together. On the one hand, this reduces the number of parameters, which simplifies the corresponding estimation problem. On the other hand, information is transferred from one part of the data space to another, thus allowing the model to learn patterns that never explicitly occurred in the training data. In the context of resource constrained data analysis, the primary interest lies in the reduced memory requirements, induced by the lower parameter space dimension and a presumably lower sample complexity. In this contribution, the concept that underlies parameter sharing is transferred to the spatio-temporal domain. More precisely, a re-parametrization of undirected probabilistic graphical models, known as Markov Random Fields (MRFs) is proposed for non-stationary time series of finite length. MRFs are equivalent to deep latent variable models [568] but obey an easier-to-interpret structure. Data for such spatio-temporal models arises naturally in distributed sensor networks. The corresponding machine learning models are, however, far too large to be processed directly at the sensor level. Re-parametrized probabilistic models exhibit a very sparse parameter space that facilitates probabilistic inference directly from a compressed model. This section studies different variants of the underlying re-parametrization and compares them in numerical experiments on benchmark data. Furthermore, we propose how the learning procedure can be embedded directly into a sensor network: proximal optimization is applied in a distributed setting. It turns out that the parameter optimization is purely local and that communication between sensor nodes is required only for the gradient computation. Different real-world applications, including traffic models and sensor network models underpin the practical relevance of compressed Spatio-Temporal Random Fields (STRF).

### 4.1.1 Introduction

Spatio-temporal sensor data is an archetypical instance of structured data. Inherent dependencies that span over space and time constitute demanding challenges when aiming for reliable models with reasonable resource requirements. Here, we consider the task of *spatio-temporal state prediction*, where the spatio-temporal structure is represented by an undirected graph $G = (V, E)$ that is either known or inferred from

data. Nodes within the network represent locations at different points in time $t$ from a finite time horizon $T$. Based on a set of $N$ partially observed joint realizations, a generative model $\mathbb{P}_{\boldsymbol{\theta}}$ is learned, where $\boldsymbol{\theta}$ is the trainable parameter. This task arises frequently in the analysis of sensor networks e.g., communication networks [577] or satellite image data [229]. For the sake of clarity, modeling the traffic in a highway network will serve as our running example. That is, the model must answer queries for all parts of the network and all points in time. Examples of such predictions are:

- Given the traffic densities of all roads in a street network at discrete time points $t_1, t_2, t_3$ (e.g., 8 o'clock on Monday, Tuesday, Wednesday): indicate the probabilities of traffic levels on a particular road $A$ at some other time point, not necessarily following the given ones (e.g., 7 o'clock on Thursday).
- Given a traffic jam at place $A$ at time $t_s$: output other places with a probability higher than 0.7 for the state "jam" in the time interval of $t_s < t < t_t$.

One particular interest lies in learning probabilistic models for answering such queries in resource-constrained environments. This addresses huge amounts of data on fast computing facilities moderate data volume on embedded or ubiquitous devices. Results and methods that are presented in this contribution are based on [566] and [567].

### 4.1.2 Previous Work

In this section, an overview of previous contributions to spatio-temporal modeling is given. The task of *traffic forecasting* is often solved by simulations [467]. This presupposes a model instead of learning it. In the course of urban traffic control, events are merely propagated that are already observed, e.g., a jam at a particular highway section results in a jam at another highway section, or the prediction is based on a physical rule that predicts a traffic jam based on a particular congestion pattern [287]. Many approaches apply statistical time series methods such as auto-regression and moving average models [705]. They do not take into account spatial relations but restrict themselves to the prediction of the state at one location given a series of observations at this particular location. An early approach, that of Whittaker, Garside, and Lindveld [703], relies on the street network topology for deriving spatial relations. The training is done via Kalman filters, which imply a strictly linear conditional independence structure, that is not expressive enough for answering queries like the ones stated above. A statistical relational learning approach to traffic forecasting uses explicit rules for modeling spatio-temporal dependencies [441]. Here, training is done by a Markov Logic Network delivering conditional probabilities of congestion classes. The discriminative model is restricted to binary classification tasks and the spatial dependencies need to be given by hand-tailored rules. Moreover, the model is not sparse and training is not scalable. Even for a small number of sensors, training takes hours of computation. When the estimation of models for spatio-temporal data on ubiquitous devices is considered,

such as when learning to predict smartphone usage patterns based on time and visited places, minutes are the order of magnitude in demand. Hence, even this advanced approach does not yet meet the demands of the spatio-temporal prediction task in resource-constrained environments.

Some geographically weighted regression or non-parametric k-Nearest Neighbor ($k$NN) methods model a task similar to spatio-temporal state prediction [263, 477, 743]. The regression expresses the temporal dynamics and the weights express spatial distances. Another way to introduce the spatial relations into the regression is to encode the spatial network into a kernel function [440]. The $k$NN method by [409] models correlations in spatio-temporal data not only by their spatial but also by their temporal distance. As stated for the spatio-temporal state prediction task, the particular place and time in question need not be known in advance, because the lazy learner $k$NN determines the prediction at query time. However, this approach does not deliver probabilities along with the predictions, either. For some applications, traffic prognoses for car drivers, a probabilistic assertion is not necessary. However, in applications of disaster management, the additional information regarding likelihood is desirable.

As is easily seen, generative Markov models fit the task of spatio-temporal state prediction. For notational convenience, let us assume only one variable $X$. Any *generative probabilistic model* represents the joint $\mathbb{P}(X, Y)$ and allows us to derive $\mathbb{P}(Y|X) = \frac{\mathbb{P}(X,Y)}{\mathbb{P}(X)}$ as well as $\mathbb{P}(X|Y) = \frac{\mathbb{P}(X,Y)}{\mathbb{P}(Y)}$. In contrast, *discriminative probabilistic models* represent $\mathbb{P}(Y|X)$ directly and must be trained specifically for each $Y$—this property is inherent since each realization of $Y$ requires a different normalization constant. In our example a distinct model would need to be trained for each place. Hence, a huge set of discriminative models would be necessary to express one generative model. A discussion of discriminative versus generative models can be found in a study by [531]. Here, we refer to the capability of interpolation (e.g., between points in time) of generative models and their informativeness in delivering probability estimates instead of merely binary decisions.

Spatial relations are naturally expressed by *graphical models*. For instance, discriminative graphical models such as Conditional Random Fields (CRFs) have been used for object recognition over time [182], while generative graphical models such as Markov Random Fields (MRFs) have been applied to video or image data [322, 723]. The number of training instances does not influence the model complexity of MRFs. However, the number of parameters can easily exceed millions. In particular when using MRFs for spatio-temporal state prediction, the numerous spatial and temporal relations soon lead to inefficiency.

We have argued in favor of using generative graphical models that model both, spatial and temporal dependencies, at the same time. However, some problems have until now prohibited this:

- The original parametrization is not well suited for producing sparse models.
- Trained models tend to overfit to the training data.

– Training high-dimensional models is not feasible.

In the following, we shall review existing work on graphical models (Section 4.1.3) and regularization methods (Section 4.1.4) so that we can then introduce a new method for spatio-temporal state prediction that does not suffer from the listed disadvantages.

### 4.1.3 Graphical Models

The formalism of probabilistic graphical models provides a unifying framework for capturing complex dependencies among random variables, and building large-scale multivariate statistical models [692]. Let $G = (V, E)$ be an undirected graph with the set of vertices $V$ and the set of edges $E \subset V \times V$. Note that the subset relation is strict, since self-edges are not allowed. Moreover, we represent undirected edges as sets (as opposed to ordered tuples). For each node (or vertex) $v \in V$, let $\boldsymbol{X}_v$ be a random variable, taking values $x_v$ in some space $\mathcal{X}_v$. The concatenation of all $n = |V|$ variables yields a multivariate random variable $\boldsymbol{X}$ with state space $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n$. Training delivers a full probability distribution over the random variable $\boldsymbol{X}$. Let $\boldsymbol{\phi}$ be an *indicator function* or *sufficient statistic* that indicates if a configuration $\boldsymbol{x}$ obeys a certain event $\{\boldsymbol{X}_\alpha = \boldsymbol{x}_\alpha\}$ with $\alpha \subseteq V$. We use the short-hand notation $\{\boldsymbol{x}_\alpha\}$ to denote the event $\{\boldsymbol{X}_\alpha = \boldsymbol{x}_\alpha\}$. The functions of $\boldsymbol{x}$ defined in the following can be also considered as functions of $\boldsymbol{X}$. We replace $\boldsymbol{x}$ by $\boldsymbol{X}$ when it makes their meaning clearer. Restricting $\alpha$ to vertices and edges,[1] one gets

$$\boldsymbol{\phi}_{\{v=x\}}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x}_v = x \\ 0 & \text{otherwise,} \end{cases} \qquad \boldsymbol{\phi}_{\{(v,w)=(x,y)\}}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } (\boldsymbol{x}_v, \boldsymbol{x}_w) = (x, y) \\ 0 & \text{otherwise} \end{cases}$$

with $\boldsymbol{x} \in \mathcal{X}$, $\boldsymbol{x}_v \in \mathcal{X}_v$ and $y \in \mathcal{X}_w$. Let us now define vectors for collections of those indicator functions:

$$\begin{aligned} \boldsymbol{\phi}_v(\boldsymbol{x}) &:= \left[\boldsymbol{\phi}_{\{v=x\}}(\boldsymbol{x})\right]_{x \in \mathcal{X}_v}, \\ \boldsymbol{\phi}_{(v,w)}(\boldsymbol{x}) &:= \left[\boldsymbol{\phi}_{\{(v,w)=(x,y)\}}(\boldsymbol{x})\right]_{(x,y) \in \mathcal{X}_v \times \mathcal{X}_w}, \\ \boldsymbol{\phi}(\boldsymbol{x}) &:= \left[\boldsymbol{\phi}_v(\boldsymbol{x}), \boldsymbol{\phi}_e(\boldsymbol{x}) : \forall v \in V, \forall e \in E\right]. \end{aligned} \tag{4.1}$$

The vectors are constructed for fixed but arbitrary orderings of $V$, $E$ and $\mathcal{X}$. The dimension of $\boldsymbol{\phi}(\boldsymbol{x})$ is thus $d = \sum_{v \in V} |\mathcal{X}_v| + \sum_{(v,u) \in E} |\mathcal{X}_v| \times |\mathcal{X}_u|$. Now, consider a dataset $\mathcal{D} = \left\{\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^N\right\}$ with instances $\boldsymbol{x}^i$. Each $\boldsymbol{x}^i$ consists of an assignment to every node in the graph. It defines a full joint state of the random variable $\boldsymbol{X}$.

1 In general, one may consider indicator functions not only for nodes and edges, but for all cliques (fully connected subgraphs) in $G$. Our description still applies to higher order models, since we can convert them into models using only nodes and edges [692, Appendix E].

The quantities

$$\hat{\boldsymbol{\mu}}_{\{v=x\}} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{\phi}_{\{v=x\}}(\boldsymbol{x}^i), \quad \hat{\boldsymbol{\mu}}_{\{(v,w)=(x,y)\}} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{\phi}_{\{(v,w)=(x,y)\}}(\boldsymbol{x}^i) \tag{4.2}$$

are known as *empirical moments* and they reflect the empirical frequency estimates of the corresponding events. We say that a given probability density function $p$ with base measure[2] $\nu$ and expectations $\mathbb{E}_p\left[\boldsymbol{\phi}_{\{\boldsymbol{x}_\alpha\}}(\boldsymbol{x})\right]$ is *locally consistent* with data $\mathcal{D}$ if and only if $p$ satisfies the *moment matching condition*

$$\mathbb{E}_p\left[\boldsymbol{\phi}_{\{\boldsymbol{x}_\alpha\}}(\boldsymbol{x})\right] = \hat{\boldsymbol{\mu}}_{\{\boldsymbol{x}_\alpha\}}, \forall \alpha \in V \cup E,$$

i.e. the density $p$ is consistent with the data w.r.t. the empirical moments.

This problem is underdetermined in that there are many densities $p$ that are consistent with the data, so that we need a principle for choosing among them. The principle of maximum entropy is to choose, among the densities consistent with the data, the densities $p^\star$ whose *Shannon entropy* $\mathcal{H}(p)$ is maximal. $\mathcal{H}$ is given by

$$\mathcal{H}(p) := -\int_{\mathcal{X}} p(\boldsymbol{x}) \log_2\left(p(\boldsymbol{x})\right) d\nu(\boldsymbol{x}).$$

This is turned into the constrained optimization problem

$$\max_{p\in\mathbb{P}} \mathcal{H}(p) \text{ subject to } \mathbb{E}_p\left[\boldsymbol{\phi}_{\{\boldsymbol{x}_\alpha\}}(\boldsymbol{x})\right] = \hat{\boldsymbol{\mu}}_{\{\boldsymbol{x}_\alpha\}}, \quad \forall \alpha \in V \cup E.$$

It can be shown that the optimal solution $p^\star$ takes the form of an exponential family of densities

$$p_{\boldsymbol{\theta}}(\boldsymbol{X} = \boldsymbol{x}) = \exp[\langle \boldsymbol{\theta}, \boldsymbol{\phi}(\boldsymbol{x})\rangle - A(\boldsymbol{\theta})],$$

parametrized by a vector $\boldsymbol{\theta} \in \mathbb{R}^d$. Note that the parameter vector $\boldsymbol{\theta}$ and the sufficient statistics vector $\boldsymbol{\phi}(\boldsymbol{x})$ have the same length $d$. The term

$$A(\boldsymbol{\theta}) := \log \int_{\mathcal{X}} \exp[\langle \boldsymbol{\theta}, \boldsymbol{\phi}(\boldsymbol{x})\rangle] d\nu(\boldsymbol{x})$$

is called *log partition function*. It is defined with respect to a reference measure $\nu$ such that $\mathbb{P}(X \in S) = \int_S p_{\boldsymbol{\theta}}(x) d\nu(x)$ for any measurable set $S$. Expanding $\boldsymbol{\phi}(\boldsymbol{x})$ by means of Equation 4.1 reveals the usual density of pairwise undirected graphical models, also known as *pairwise MRFs*

$$\begin{aligned} p_{\boldsymbol{\theta}}(\boldsymbol{X} = \boldsymbol{x}) &= \frac{1}{\exp A(\boldsymbol{\theta})} \prod_{v\in V} \exp[\langle \boldsymbol{\theta}_v, \boldsymbol{\phi}_v(\boldsymbol{x})\rangle] \prod_{(v,w)\in E} \exp[\langle \boldsymbol{\theta}_{(v,w)}, \boldsymbol{\phi}_{(v,w)}(\boldsymbol{x})\rangle] \\ &= \frac{1}{\Psi(\boldsymbol{\theta})} \prod_{v\in V} \psi_v(\boldsymbol{x}) \prod_{(v,w)\in E} \psi_{(v,w)}(\boldsymbol{x}). \end{aligned}$$

**2** Notice that when the underlying state space $\mathcal{X}$ is discrete, then $\nu$ is the counting measure and we may identify the density $p$ with the measure $\mathbb{P}$.

Here, $\Psi = \exp A$ is the cumulant-generating function of $p_{\boldsymbol{\theta}}$, and $\psi_\alpha$ refers to the *potential functions*.

Inference, that is, computing the marginal probabilities or maximum a-posteriori states of each vertex, can be carried out by message propagation algorithms [404, 560, 690], variational methods [692], or quadrature-based methods [572, 573]. In order to fit the model on some dataset, the model parameters have to be estimated. If the dataset contains only fully observed instances, the parameters may be estimated by the maximum likelihood principle. The estimation of parameters in the case of partially unobserved data is a challenging topic on its own. Here, we assume that the dataset $\mathcal{D}$ contains only fully observed instances. The *likelihood* $\mathcal{L}$ and the *average log-likelihood* $\ell$ of parameters $\boldsymbol{\theta}$ given a set of i.i.d. data $\mathcal{D}$ are defined as

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) := \prod_{i=1}^{N} p_{\boldsymbol{\theta}}(\boldsymbol{x}^i) \quad \text{and} \quad \ell(\boldsymbol{\theta}; \mathcal{D}) := \frac{1}{N} \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}^i) = \langle \boldsymbol{\theta}, \hat{\boldsymbol{\mu}} \rangle - A(\boldsymbol{\theta}). \tag{4.3}$$

The latter is usually maximized due to numerical inconveniences of $\mathcal{L}$. The most frequently applied optimization methods are iterative proportional fitting [160], gradient descent and quasi-newton methods such as L-BFGS or the conjugate gradient [538]. Section 4.1.5 will show how to model spatio-temporal dependencies within this formalism.

### 4.1.4 Regularization

As we can see, the number of parameters in $\boldsymbol{\theta}$ grows quite rapidly as we consider more complex graphical models. A large number of parameters is generally not preferable, since it may lead to overfitting, and it resists the implementation of a memory-efficient predictor. Therefore, some regularization is necessary to achieve a sparse and robust model.

Popular choices of regularizers are the $l_1$ and $l_2$ norms of the parameter vector, $\|\boldsymbol{\theta}\|_1$ and $\|\boldsymbol{\theta}\|_2$. By minimizing the $L_1$ norm, we coerce the values for less informative parameters to zero (similar to LASSO [660]), and by the $l_2$ norm we find smooth functions parametrized by $\boldsymbol{\theta}$ (similar to the penalized splines [559]). Using both together is often referred to as the *elastic net* [748]. For graphical models, elastic nets appeared in the context of structure learning (estimating the neighborhoods) [156] in a manner similar to the approach of [484]. For the state prediction task, there exist two short workshop papers [569, 571] using the elastic net. However, their analytical and empirical validation of such an approach is rather limited.

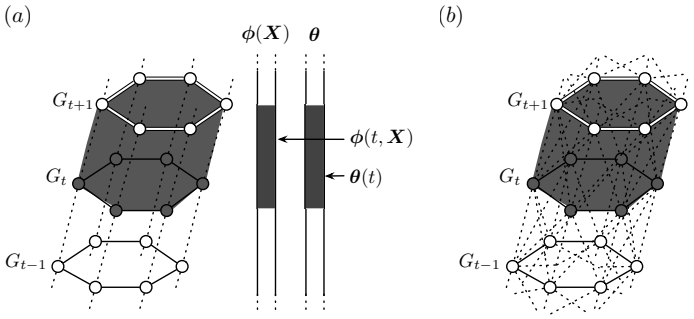



**Fig. 4.1:** A spatio-temporal model consisting of multiple snapshot graphs $G_t$ for $t = 1, 2, \ldots, T$. The spatial and temporal edges are represented by solid and dotted lines, respectively. (a) A layer $L_t$ is shown as the shaded region with simple temporal edges ($L_t$ does not include the elements of $G_{t+1}$), along with the corresponding sufficient statistic and parameter subvectors $\boldsymbol{\phi}(t, X)$ and $\boldsymbol{\theta}(t)$. (b) An extended model with "crossing" temporal edges between consecutive snapshots. This extended model is adopted in our experiments.

### 4.1.5 From Linear Chains to Spatio-Temporal Models

Sequential undirected graphical models, also known as linear chains, are a popular method in the natural language processing community [407, 654]. There, consecutive words or corresponding word features are connected to a sequence of labels that reflects an underlying domain of interest like entities or part of speech tags. If we consider a sensor network $G$ that generates measurements over space such as a word, then it would be appealing to think of the instances of $G$ at different time points, like words in a sentence, to form a temporal chain $G_1 - G_2 - \cdots - G_T$. We will now present a formalization of this idea followed by some obvious drawbacks. Hereafter, we will discuss how to tackle those drawbacks and derive a tractable class of generative graphical models for the spatio-temporal state prediction task.

We first define the part of the graph corresponding to the time step $t$ as the *snapshot graph* $G_t = (V_t, E_t)$, for $t = 1, 2, \ldots, T$. Each snapshot graph $G_t$ replicates a given *spatial graph* $G_0 = (V_0, E_0)$, which represents the underlying physical placement of sensors, i.e., the spatial structure of random variables that does not change over time. We also define the set of spatio-temporal edges $E_{t-1;t} \subset V_{t-1} \times V_t$ for $t = 2, \ldots, T$ and $E_{0;1} = \emptyset$, that represent dependencies between adjacent snapshot graphs $G_{t-1}$ and $G_t$, assuming a Markov property among snapshots, so that $E_{t;t+h} = \emptyset$ whenever $h > 1$ for any $t$. Note that the actual time gap between any two time frames $t$ and $t + 1$ can be chosen arbitrarily.

The entire graph, denoted by $G$, consists of the snapshot graphs $G_t$ stacked in the order of time frames $t = 1, 2, \ldots, T$ and the temporal edges connecting them: $G := (V, E)$ for $V := \cup_{t=1}^{T} V_t$ and $E := \cup_{t=1}^{T} \{E_t \cup E_{t-1;t}\}$. We sketch the structure of $G$ in Figure 4.1.

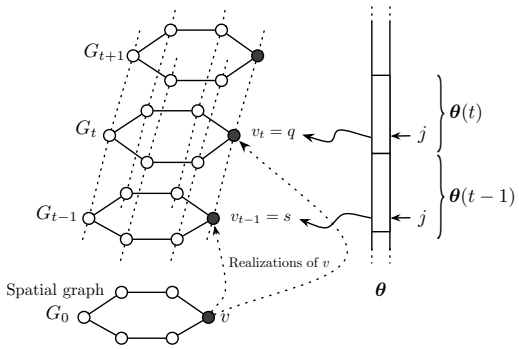



**Fig. 4.2:** An example of indexing for a node and state pair over time. A sensor modeled by the node $v$ in the spatial graph $G_0$ shows its measurements $v_{t-1}$ and $v_t$ at time frames $t-1$ and $t$, respectively. The pairs $v_{t-1} = s$ and $v_t = q$ are located at the same index $j$ in the subvectors $\boldsymbol{\theta}(t-1)$ and $\boldsymbol{\theta}(t)$.

For the sake of a simple description, we define a *layer* $L_t$ as the partial subgraph of $G$ containing all vertices of $V_t$ and all edges of $E_t \cup E_{t;t+1}$, for $t = 1, 2, \ldots, T$. For instance, a layer $L_t$ is depicted as a shaded region in Figure 4.1. Let $a \in \mathcal{X}_v$ and $b \in \mathcal{X}_w$ and define the subvectors of $\boldsymbol{\phi}(X)$ and $\boldsymbol{\theta}$ that correspond to a layer $L_t$ as follows:

$$\begin{aligned} \boldsymbol{\phi}(t, \boldsymbol{X}) &:= (\boldsymbol{\phi}_{v=a}(\boldsymbol{X}_v), \boldsymbol{\phi}_{(v,w)=(a,b)}(\boldsymbol{X}_v, \boldsymbol{X}_w) \mid v \in L_t,\ (v,w) \in L_t,), \\ \boldsymbol{\theta}(t) &:= (\boldsymbol{\theta}_{v=a}, \boldsymbol{\theta}_{(v,w)=(a,b)} \mid v \in L_t,\ (v,w) \in L_t). \end{aligned} \tag{4.4}$$

By construction, the layers $L_1, L_2, \ldots, L_T$ define a non-overlapping partitioning of a graph $G$, which allows us to write

$$\langle \boldsymbol{\phi}(\boldsymbol{X}), \boldsymbol{\theta} \rangle = \sum_{t=1}^{T} \langle \boldsymbol{\phi}(t, \boldsymbol{X}), \boldsymbol{\theta}(t) \rangle.$$

The subvectors $\boldsymbol{\phi}(t, \boldsymbol{X})$ and $\boldsymbol{\theta}(t)$ have the same length $d' := d/T$ for all $t = 1, 2, \ldots, T$. Note that the subvectors should be "aligned", in the sense that the $j$th elements in all subvectors must point to the same node:state or edge:states pair over time. We illustrate this in Figure 4.2.

The spatial graph $G_0$ and the sizes of the vertex state spaces $\mathcal{X}_v$ determine the number of model parameters $d$. In order to compute this quantity, we consider the construction of $G$ (as shown in Figure 4.1 (b)) from $G_0$. First, all vertices $v$ and all edges $(u, v)$ from $G_0$ are copied exactly $T$ times and added to $G = (V, E)$, whereas each copy is indexed by time step $t$, i.e. $v \in V_0 \Rightarrow v_t \in V_t$, $1 \le t \le T$ and likewise for the edges. Then, for each vertex $v_t \in V$ with $t \le T-1$, a temporal edge $(v_t, v_{t+1})$ is added to $G$. Finally, for each edge $(v_t, u_t) \in E$ with $t \le T-1$, the two spatio-temporal edges $(v_t, u_{t+1})$ and $(v_{t+1}, u_t)$ are also added to $G$. The number of parameters per vertex $v$ is $|\mathcal{X}_v|$ and

accordingly $|\mathcal{X}_v||\mathcal{X}_u|$ per edge $(v, u)$. Thus, the total number of model parameters is

$$\begin{aligned} d = &\sum_{v\in V_0}\sum_{t=1}^{T}|\mathcal{X}_{v_t}| + \sum_{v\in V_0}\sum_{t=1}^{T-1}|\mathcal{X}_{v_t}|\,|\mathcal{X}_{v_{t+1}}| + \sum_{(u,v)\in E_0}|\mathcal{X}_{v_T}|\,|\mathcal{X}_{u_T}| \\ &+ \sum_{(u,v)\in E_0}\sum_{t=1}^{T-1}\left(|\mathcal{X}_{v_t}|\,|\mathcal{X}_{u_{t+1}}| + |\mathcal{X}_{v_{t+1}}|\,|\mathcal{X}_{u_t}| + |\mathcal{X}_{v_t}|\,|\mathcal{X}_{u_t}|\right). \end{aligned} \tag{4.5}$$

If we assume that all vertices $v, u \in V$ share a common state space and that state spaces do not change over time, i.e. $\mathcal{X}_{v_t} = \mathcal{X}_{u_{t'}}, \forall v, u \in V, 1 \le t, t' \le T$, the expression simplifies to

$$d = \underbrace{T|V_0|\,|\mathcal{X}_{v_t}|}_{\text{\# of vertex parameters}} + \underbrace{\left[(T-1)(|V_0| + 3|E_0|) + |E_0|\right]|\mathcal{X}_{v_t}|^2}_{\text{\# of edge parameters}}$$

with some arbitrary but fixed vertex $v_t$. Note that the last two assumptions are only needed to simplify the computation of dimension $d$; the spatio-temporal random field that is described in the following section is not restricted by any of these assumptions.

This model now truly expresses temporal and spatial relations between all locations and points in time for all features. However, the memory requirements of such models are quite high due to the large problem dimension. Even loading or sending models may cause issues when mobile devices are the platform. Furthermore, the training does not scale well because of step-size adaption techniques that are based on sequential (i.e., non-parallel) algorithms.

### 4.1.6 Spatio-Temporal Random Fields

Now we describe how we modify the naive spatio-temporal graphical model discussed above. We have two goals in mind: (i) to achieve compact models retaining the same prediction power, and (ii) to find the best of such models via scalable distributed optimization.

#### 4.1.6.1 Towards Better Sparsification

The memory consumption of MRFs is dominated by the size of its parameter vector: the graph $G$ can be stored within $\mathcal{O}(|V| + |E|)$ space (temporal edges do not have to be constructed explicitly), and the size of intermediate variables required for inference is $\mathcal{O}(2|E||\mathcal{X}_v|)$. That is, if $|\mathcal{X}_v| \ge 2$ for all $v$, the dimension $d$ in Equation 4.5 and therefore the memory consumption of the parameter vector are always a dominant factor. Also, since each parameter is usually accessed multiple times during inference, it is desirable to have them in a fast storage, e.g. a cache memory.

An important observation on the parameter subvector $\boldsymbol{\theta}(t)$ is that it is unlikely to be a zero vector when it models an informative distribution. For example, if the

nodes can have one of the two states {high, low}, suppose that the corresponding parameters at time $t$ satisfy $[\boldsymbol{\theta}(t)]_v = 0$ for all $v$ and equally for all edge weights. Then it implies $\mathbb{P}(\boldsymbol{X}_v = \text{high}) = \mathbb{P}(\boldsymbol{X}_v = \text{low})$, a uniform marginal distribution. The closer the parameters of a classical MRF tend towards $\mathbf{0}$, the closer are the corresponding marginals to the uniform distribution.

When all consecutive layers are sufficiently close in time, the transition of distributions over the layers will be smooth in many real-world applications. But the optimal $\boldsymbol{\theta}$ is likely to be a dense vector, and it will require a large memory and possibly a long time to make predictions with it as we deal with large graphical models. This creates the necessity for a different parametrization.

#### 4.1.6.2 Reparametrization

In our reparametrization, we consider a piecewise linear representation of $\boldsymbol{\theta}(t)$ with new parameter vectors $\boldsymbol{\Delta}_{\cdot i} \in \mathbb{R}^{d'}$ for $i = 1, 2, \ldots, T$,

$$\boldsymbol{\theta}(t) = \sum_{i=1}^{t} \frac{1}{t-i+1} \boldsymbol{\Delta}_{\cdot i}, \quad t = 1, 2, \ldots, T. \tag{4.6}$$

Our motivation is best shown by the differences in $\boldsymbol{\theta}$ between two consecutive layers, $\Delta_{(t-1):t} := \boldsymbol{\theta}(t) - \boldsymbol{\theta}(t-1) = \boldsymbol{\Delta}_{\cdot t} - \sum_{i=1}^{t-1} \frac{1}{(t-i+1)(t-i)} \boldsymbol{\Delta}_{\cdot i}$. That is, the difference (slope) is mostly captured by the first term $\boldsymbol{\Delta}_{\cdot t}$, and by the remainder terms $\boldsymbol{\Delta}_{\cdot (t-i)}$ with quadratically decaying weights in $\mathcal{O}(i^{-2})$, for $i = 1, 2, \ldots, t$. We note that a simpler alternative might be setting $\boldsymbol{\theta}(t) = \sum_{i=1}^{t} \boldsymbol{\Delta}_{\cdot i}$, but our approach leads to better conditions in optimization which allow for faster convergence.

With the new parameters, if the changes between two consecutive layers are near zero, that is, $\boldsymbol{\theta}(t) \approx \boldsymbol{\theta}(t-1)$, then we expect $\boldsymbol{\Delta}_{\cdot t} \approx 0$. This is a novel property of the new parametrization, since with the classical parameters $\boldsymbol{\theta}$ the condition does not necessarily entail $\boldsymbol{\theta}(t) \approx 0$. In other words, $\boldsymbol{\Delta}_{\cdot t} = 0$ implies no changes in the distribution from $t-1$ to $t$, but $\boldsymbol{\theta}(t) = 0$ implies the distribution at $t$ suddenly becoming a uniform distribution, regardless of the previous state at layer $t-1$. An example is illustrated in Figure 4.3.

Since we have defined $\boldsymbol{\theta}$ as a concatenation of vectors $\boldsymbol{\theta}(1), \boldsymbol{\theta}(2), \ldots, \boldsymbol{\theta}(T)$, the reparametrization reads as follows:

$$\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}(1) \\ \boldsymbol{\theta}(2) \\ \vdots \\ \boldsymbol{\theta}(T) \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Delta}_{\cdot 1} \\ \frac{1}{2}\boldsymbol{\Delta}_{\cdot 1} + \boldsymbol{\Delta}_{\cdot 2} \\ \vdots \\ \sum_{i=1}^{T} \frac{1}{t-i+1} \boldsymbol{\Delta}_{\cdot i} \end{bmatrix}, \quad \boldsymbol{\Delta} := \begin{bmatrix} | & | & & | \\ \boldsymbol{\Delta}_{\cdot 1} & \boldsymbol{\Delta}_{\cdot 2} & \cdots & \boldsymbol{\Delta}_{\cdot T} \\ | & | & & | \end{bmatrix}.$$

For convenience, we define the *slope matrix* $\boldsymbol{\Delta} \in \mathbb{R}^{d' \times T}$ as above, which contains $\boldsymbol{\Delta}_{\cdot 1}$, $\boldsymbol{\Delta}_{\cdot 2}, \ldots, \boldsymbol{\Delta}_{\cdot T}$ as its columns. In the following we sometimes use the notations $\boldsymbol{\theta}(\boldsymbol{\Delta})$ and $\boldsymbol{\theta}(t, \boldsymbol{\Delta})$, whenever it is necessary to emphasize the fact that $\boldsymbol{\theta}$ and $\boldsymbol{\theta}(t)$ are functions of

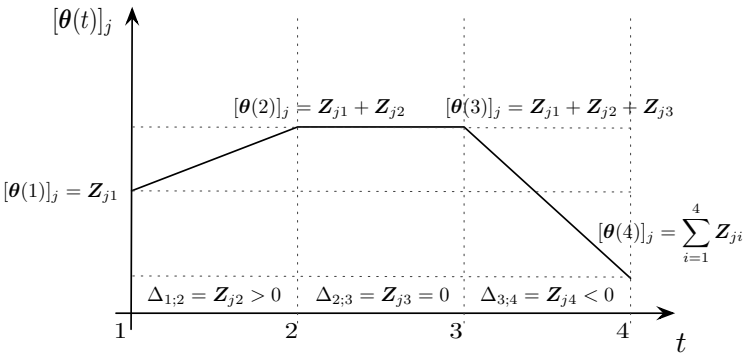



**Fig. 4.3:** A simplified example of the reparametrization of $[\boldsymbol{\theta}(t)]_j$, the $j$th element in the subvector $\boldsymbol{\theta}(t)$, over the timeframes $t = 1, 2, 3, 4$. We store slopes $\boldsymbol{\Delta}_{jt}$ instead of the actual values of the piecewise linear function $[\boldsymbol{\theta}(t)]_j$ between two consecutive timeframes $t - 1$ and $t$ (except for $\boldsymbol{\Delta}_{j1}$ which works as an intercept). Near-zero slopes $\boldsymbol{\Delta}_{jt} \approx 0$ ($\boldsymbol{\Delta}_{j3} = 0$ above) can be removed from computation and memory.

$\boldsymbol{\Delta}$ under the new parametrization. Finally, another property of our reparametrization is that it is linear. Therefore an important property for optimization carries over: $A(\boldsymbol{\theta}(\boldsymbol{\Delta}))$ is convex in $\boldsymbol{\Delta}$ as $A(\boldsymbol{\theta})$ is convex in $\boldsymbol{\theta}$ [692].

We note that due to the summation in Equation 4.6 our reparametrization with $\boldsymbol{\Delta}$ introduces some additional overhead compared with the classical parametrization with $\boldsymbol{\theta}$. In particular, whenever an algorithm has to read a value from $\boldsymbol{\theta}$, it has do be decompressed instantly, which adds asymptotic complexity $\mathcal{O}(T)$ to every access. However, if we obtain a *sparse representation* with $\boldsymbol{\Delta}$, then it can be stored in small memory (possibly even in CPU cache memory) and therefore the chances for cache misses or memory swapping will be reduced. This becomes an important factor when, say, we deploy a learned model to applications running on mobile devices. Chapter 7 presents approaches to memory-aware learning in other classes of learning methods.

#### 4.1.6.3 Analysis

We define the $l_1$ and $l_2$ regularizers for the slope matrix $\boldsymbol{\Delta}$ as follows,

$$\|\boldsymbol{\Delta}\|_1 := \sum_{j=1}^{d'} \|\boldsymbol{\Delta}_{j\cdot}\|_1, \quad \|\boldsymbol{\Delta}\|_F^2 := \sum_{j=1}^{d'} \|\boldsymbol{\Delta}_{j\cdot}\|_2^2. \tag{4.7}$$

The two regularizers induce sparsity and smoothness respectively, as we have discussed in Section 4.1.4. The difference is that due to the reparametrization, now differences between parameters $\boldsymbol{\theta}(t-1)$ and $\boldsymbol{\theta}(t)$ are penalized, not the actual values they contain, which are unlikely to be zero.

The proposed reparametrizations can result in large improvements regarding a model's memory consumption. Clearly, the amount of reduction depends on the specific

dataset. It is hence even more astonishing that the reparametrization itself can be applied without any harm—it can represent any natural parameter. Let us consider a proper definition of our former intuition. For the sake of generality, let $C$ be any clique (e.g., an edge) of the underlying graph.

**Definition 4** (Piecewise Linear Reparametrization [567])**.** *Let $G$ be a spatio-tem-poral graph of length $T$, and let $\boldsymbol{D}(h) \in [0;1]^{h \times h}$ be a lower unitriangular[3] matrix. Any MRF with graph $G$ and piecewise linear clique-wise reparametrization*

$$\boldsymbol{\theta}_{C=\boldsymbol{x}'} = \eta_{\boldsymbol{D}(h)}(\boldsymbol{\Delta}_{C=\boldsymbol{x}'}) = \boldsymbol{D}(h)\boldsymbol{\Delta}_{C=\boldsymbol{x}'} \tag{4.8}$$

*where $h = T - (\max\{t' \mid v(t') \in C\} - \min\{t' \mid v(t') \in C\})$ is called a* spatio-temporal random field.

Based on that definition, we can derive some useful properties.

**Lemma 5** (Universality of the Reparametrization)**.** *The spatio-temporal repara-metrization is universal. That is, the piecewise linear reparametrization is a bijection.*

**Proof** Indeed, any $\boldsymbol{\Delta} \in \mathbb{R}^d$ can be mapped to some $\boldsymbol{\theta} \in \mathbb{R}^d$ by multiplication with $\boldsymbol{D}$ according to Definition 4. To see that the converse also holds, note that for each $t \in [T]$, $\det \boldsymbol{D}(h) = \prod_{i=1}^{t} \boldsymbol{D}(h)_{i,i} = 1$, due to unitriangularity. Each $\boldsymbol{D}(h)$ is thus invertible and so is the block diagonal matrix $\boldsymbol{D}^{\circ}$. So for any given natural parameter $\boldsymbol{\theta}_{C=\boldsymbol{y}}$, we can find the corresponding reparametrization via $\boldsymbol{\Delta}_{C=\boldsymbol{y}} = \boldsymbol{D}^{-1}\boldsymbol{\theta}_{C=\boldsymbol{y}}$. That is, $\eta_{\boldsymbol{D}}$ is bijective and hence universal. ■

Since $\eta_{\boldsymbol{D}}$ is universal, any natural parameter can be represented via some $\boldsymbol{\Delta}$. Moreover, $\eta_{\boldsymbol{D}}$ is a linear function of $\boldsymbol{\Delta}$. The convexity of a function is preserved by composing it with a linear function. Hence, the reparametrized negative average log-likelihood $\ell(\eta_{\boldsymbol{D}}(\boldsymbol{\Delta}); \mathcal{D}) = A(\eta_{\boldsymbol{D}}(\boldsymbol{\Delta})) - \langle \eta_{\boldsymbol{D}}(\boldsymbol{\Delta}), \tilde{\boldsymbol{\mu}} \rangle$ is a convex function of $\boldsymbol{\Delta}$.

Up to now, we have not saved any memory since $\boldsymbol{\Delta}$ and $\boldsymbol{\theta}$ have the same dimension. By imposing $l_1$- and $l_2$-regularization on the reparametrized objective, we arrive at the problem

$$\min_{\boldsymbol{\Delta} \in \mathcal{R}^d} \underbrace{A(\eta_{\boldsymbol{D}}(\boldsymbol{\Delta})) - \langle \eta_{\boldsymbol{D}}(\boldsymbol{\Delta}), \tilde{\boldsymbol{\mu}} \rangle + \frac{\lambda_2}{2}\|\boldsymbol{\Delta}\|_F^2 + \lambda_1 \|\boldsymbol{\Delta}\|_1}_{\ell^{\mathrm{ST}}(\boldsymbol{\Delta}; \mathcal{D})} . \tag{4.9}$$

The following theorem shows that the intuition that we used to design our reparametrization has indeed the desired effect—it allows us to convert redundancy into sparsity by detecting negligible changes in consecutive natural parameters. Moreover, a polynomial number of samples suffices to achieve a small estimation error with high probability.

**3** An unitriangular matrix is triangular and all entries on its main diagonal are 1.

**Theorem 6** (STRF Consistency)**.** *Consider a random variable* $\boldsymbol{X}$ *with exponential family density, parameter* $\boldsymbol{\theta}^\star \in \mathbb{R}^d$ *whose reparametrization has minimal norm among all equivalent parameters, and a generalized sequence structure of length* $T$*. We are given a dataset* $\mathcal{D}$ *with* $N = |\mathcal{D}|$ *samples from* $\boldsymbol{X}$*. Suppose* $\|\nabla^2 A(\boldsymbol{\theta}^\star)^{-1}\|_\infty \leq \kappa$ *and* $\|\boldsymbol{\Delta}\|_\infty \leq \gamma$*, and set* $\lambda_1 = 4T\sqrt{\log(d)/N}$ *and* $\lambda_2 = \gamma^{-1}\lambda_1$*. If* $N \geq 324\kappa^4 d^{12}\log(d)/(T-d^2)^2$*, then, for an arbitrary decay matrix* $\boldsymbol{D}$*:*

- *the distance between the true parameter* $\boldsymbol{\theta}^\star$ *and the estimate* $\eta_{\boldsymbol{D}}(\hat{\boldsymbol{\Delta}})$ *is bounded, i.e.,*

$$\|\eta_{\boldsymbol{D}}(\hat{\boldsymbol{\Delta}}) - \boldsymbol{\theta}^\star\|_\infty \leq 3\kappa d^2 \lambda_1 \ ,$$

- *any sparsity in the estimate implies some redundancy in the true parameter, i.e.,* $\hat{\boldsymbol{\Delta}}_{C=\boldsymbol{x}'}(t) = 0 \Rightarrow$

$$\begin{aligned} & |\boldsymbol{\theta}^\star_{C=\boldsymbol{x}'}(t-1) - \boldsymbol{\theta}^\star_{C=\boldsymbol{x}'}(t)| \\ \leq \ & \frac{3d^2\kappa\lambda_1}{T} + (t-1)\left(\max_{i=1}^{t-1} |\hat{\boldsymbol{\Delta}}_{C=\boldsymbol{x}'}(i)| + \frac{3d^2\kappa\lambda_1}{T}\right) \ , \end{aligned}$$

*for any clique* $C$ *and time-point* $t$*. Both statements hold with probability at least* $1-(2/d)$*.*

A proof for this statement can be found in [567].

### 4.1.7 Experimental

We evaluate the performance of our suggested method on two real-world datasets, where each set is described by a spatial graph $G_0 = (V_0, E_0)$ with a set of sensors $V_0$ and connections $E_0$, and a set of historical sensor readings $\mathcal{D}$. We evaluate two approaches: MRFs with the original parametrization (MRF) and the spatio-temporal random fields[4] (STRF) presented in this section.

First we discuss the model training. We investigate the prediction quality and sparsity of resulting models with respect to regularization parameters. We also present the impact of separable optimization on training time. Next, the quality of prediction on test sets is discussed, regarding the sparsity (and thereby the size in memory) of trained models. Finally, we discuss the qualitative results regarding the interpretability of the STRF model.

Throughout the experiments, our STRF algorithm has produced solutions satisfying our target optimality of $< 10^{-5}$ within ten iterations. A description of the traffic and temperature datasets as well as the quality measures (accuracy Acc and number-of-non-zero-ratio NNZ) used for this evaluation can be found in [566].

**4** An implementation is part of the Python package pxpy which is available at https://pypi.org/project/pxpy.

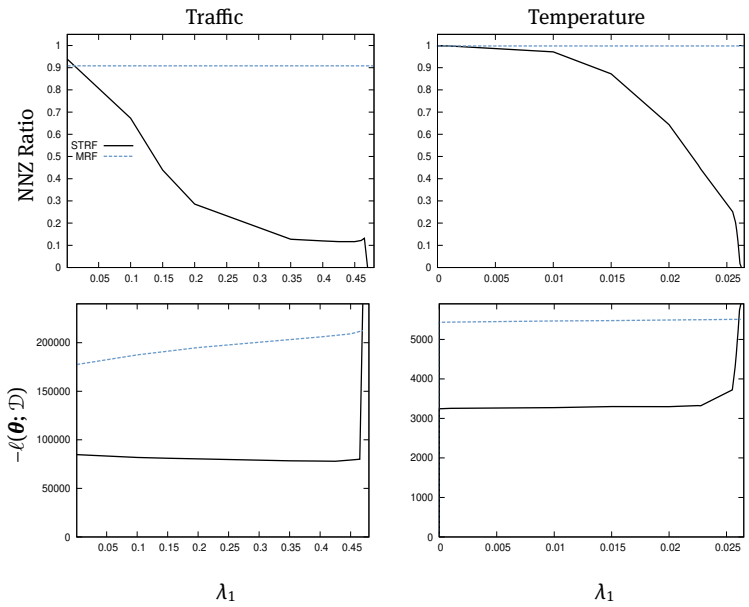



**Fig. 4.4:** The effect of regularization on models for varying sparsity parameter $\lambda_1$ (left: traffic data, right: temperature data, top: NNZ ratio, bottom: negative log-likelihood). All measurements were obtained after ten iterations, which was enough for STRF to reach the target optimality.

### 4.1.8 Regularized Training of Spatio-Temporal Random Fields

In our model, the $l_2$ regularizer imposes "smoothness" on the dynamics of parameters over time, providing a controllable way to avoid overfitting noisy observations. The degree of smoothness is controlled by $\lambda_2$, whereas the compression ratio is controlled by $\lambda_1$. Positive values of $\lambda_2$ help in our method, since the curvature estimation becomes better conditioned.

#### 4.1.8.1 Sparsity of Trained Models and Their Training Accuracy

Figure 4.4 shows the performance of STRF (our method) and MRF (classical parametrization) in terms of the negative log-likelihood and the NNZ ratio for a range of values for $\lambda_1$. The parameter $\lambda_2$ was fixed to $10^{-1}$ (the characteristics were almost identical for various $\lambda_2$ values we tried in the range of $[0, 1]$). For MRF, we augmented the objective with the $l_1$ and $l_2$ regularizers discussed in Section 4.1.4, and then applied a subgradient descent method with fixed step size ($\eta = 10^{-2}$). Our results show that (i) the subgradient method does not properly perform regularization for MRF, regardless of the choices of $(\lambda_1, \lambda_2)$; (ii) the negative log-likelihood decreases as $\lambda_1$ is increased, which is expected because at the strongest $l_1$ regularization will force all marginals to be uniform distributions; (iii) our method STRF identifies sparse models accordingly to given regularization strength, while retaining similar likelihood values to MRF. More

precisely, focusing on the curves for STRF, likelihood keeps improving until $\lambda_1$ reaches $0.47$. Beyond this value, the model is compressed too much, losing its prediction power. Overall, the pair $(\lambda_1, \lambda_2) = (0.4655, 1.0)$ with NNZ ratio 0.101573 has been identified as a good choice for the traffic data, and the pair $(\lambda_1, \lambda_2) = (0.0255, 1.0)$ with NNZ ratio 0.248136 has been identified as a good choice for the temperature data, since both lead to sparse models with reasonable likelihood values. We use these values in the following experiments.

Since the number of edge parameters is a dominant factor in the dimension $d$ of the parameter space, it would be desirable that STRF sufficiently compresses edge parameters. Considering the NNZ ratio of vertex and edge parameters separately, it turns out that STRF has such a property: with the good parameter values above, the NNZ ratio of vertices is about $0.95$, whereas that of the edges is about $0.09$.

### 4.1.9 Prediction on Test Sets

Here we investigate (i) the test-set performance of the sparse models, obtained with the good parameter values of $\lambda_1$ and $\lambda_2$ found in training, and (ii) how the sparsity of trained models affect the testing time.

The test-set accuracy of the models, obtained by the regularization parameters described in Section 4.1.8.1, is presented in Figure 4.5. Here our method STRF, the classical MRF, the $k$NN algorithm with several values of $k$, and the random guessing method, are compared. The prediction quality of the models produced by STRF is almost identical to that of MRF, although the STRF models are much smaller in size (10.2 % and 24.8 % of the MRF models in size, for traffic and temperature, respectively). The $k$NN algorithm sometimes performs better than STRF and MRF, but remember that $k$NN cannot capture probabilistic relations and requires access to full training data, which is not the case for STRF and MRF.

### 4.1.10 Conclusion

In this contribution, we presented an improved graphical model designed for the efficient probabilistic modeling of spatio-temporal data. It is based on a combination of parametrization and regularization, such that the estimated parameters are sparse and the estimated marginal probabilities are smooth without losing prediction accuracy. We investigated the sparsity, smoothness, prediction accuracy, and scalability of the model on real-world datasets. The experiments showed that often around 10 % of the original model size suffices to achieve almost the same prediction accuracy. Moreover, the method is amenable to parallelization and scales well with an increasing number of CPUs.

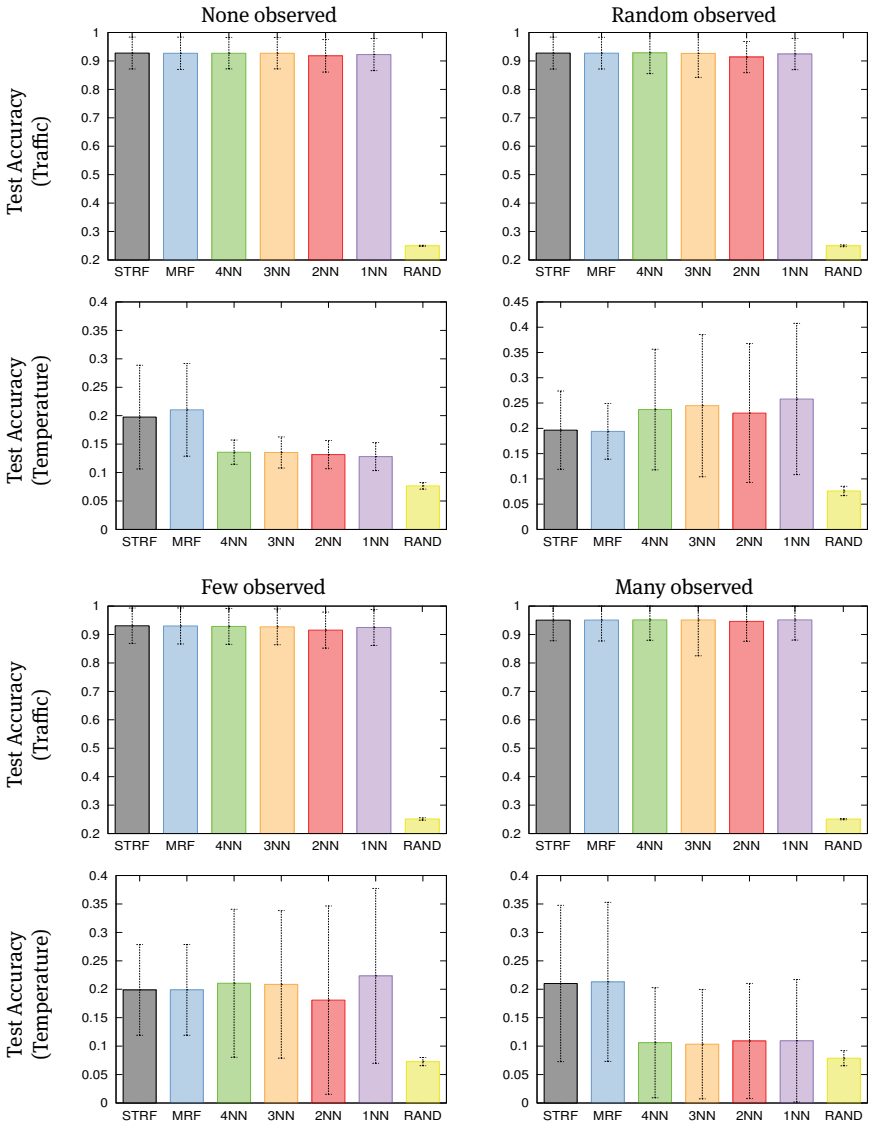



**Fig. 4.5:** Test accuracy of STRF, MRF, and *k*-nearest neighbor algorithm on the traffic dataset for four scenarios: unconditioned (first column, first two rows), random observed layers (second column, first two rows), conditioned on Monday (first column, last two rows), conditioned on Monday to Saturday (first column, last two rows).

## 4.2 The Weisfeiler-Leman Method for Machine Learning with Graphs

*Nils Kriege*
*Christopher Morris*

**Abstract:** The Weisfeiler-Leman method is a classic heuristic for graph isomorphism testing, which iteratively encodes vertex neighborhoods of increasing radius by vertex colors. Two graphs whose vertex colors do not match are called non-isomorphic. The method is fundamental for recent advances in machine learning with graphs, e.g., graph kernels and graph neural networks. This contribution overviews the development of graph kernels based on the Weisfeiler-Leman algorithm, which are among the most successful graph kernels today. We describe the Weisfeiler-Leman heuristic for graph isomorphism testing, from which the classical Weisfeiler-Leman subtree kernel directly follows. Further, we summarize the theory of optimal assignment kernels and present the Weisfeiler-Leman optimal assignment kernel for graphs and the related Wasserstein Weisfeiler-Leman graph kernel. We discuss kernel functions based on the $k$-dimensional Weisfeiler-Leman algorithm, a strict generalization of the Weisfeiler-Leman heuristic. We show that a local, sparsity-aware variant of this algorithm can lead to scalable and expressive kernels. Moreover, we survey other kernels based on the principle of Weisfeiler-Leman refinement. Finally, we shed some light on the connection between Weisfeiler-Leman-based kernels and neural architectures for graph-structured input.

### 4.2.1 Introduction

Graph-structured data is ubiquitous across application domains ranging from chemo- and bioinformatics [40, 647] to image [633] and social network analysis [193]. In drug discovery, molecules are represented as graphs [379] and the search for promising drug candidates that bind to a specific target protein can be greatly accelerated by machine learning methods suitable for graph data. Moreover, proteins themselves [64] as well as their interactions and complexes [646] (also see 2.6 in Volume 3) can be adequately modeled as graphs. The increasing amount of data in these areas offers enormous potential in studying diseases and their cures. However, due to the size and complexity of the data, automated methods for their analysis are required.

To develop successful machine learning models in these domains, we need techniques that can exploit the rich information inherent in the graph structure and the feature information contained within vertices and edges. In recent years, numerous approaches have been proposed for machine learning with graphs—most notably, methods based on graph kernels [398] and graph neural networks (GNN) [122, 252, 272].

Here, graph kernels based on the 1-*dimensional Weisfeiler-Leman algorithm* (1-WL) [28, 271], and corresponding GNNs [509, 714] have recently advanced the state of the art in supervised node and graph learning.

The 1-WL was introduced as a heuristic for the graph isomorphism problem and is widely used as a subroutine in graph isomorphism and canonization algorithms following the individualization-refinement paradigm [480]. It allows recognizing two graphs as non-isomorphic. More precisely, 1-WL assigns colors to the nodes of two graphs in an iterative process, such that isomorphic graphs are assigned matching node colors. Whenever two graphs obtain different colorings, they are guaranteed to be non-isomorphic. However, two graphs with matching colors may still be non-isomorphic. The abilities and limitations of the 1-WL for this task have been studied for decades and are well understood [271]. In machine learning with graph-structured data, the goal is less clear, and a general objective is to compute a meaningful similarity between graphs. Two graphs that are non-isomorphic but differ only by one edge, say, should still be considered highly similar. In practical applications, it has been observed that the Weisfeiler-Leman technique is often suitable to approximate computationally demanding graph similarity measures based on the minimum number of edit operations required to transform one graph into the other [397, 646]. (See 2.6 in Volume 3 for details.) Moreover, Weisfeiler-Leman type algorithms are remarkably successful in machine learning tasks. However, their abilities and limitations in these applications are not well understood and are the subject of current research.

Here, we give an overview of the recent progress of graph kernels based on the Weisfeiler-Leman paradigm. That is, we review the 1-WL and its more expressive generalization, the $k$-WL. Starting from the Weisfeiler-Leman subtree kernel [627], a simple graph kernel based on the 1-WL, we survey the area with a focus on assignment-based kernels and an extension based on the $k$-WL. Moreover, we overview the connections between the Weisfeiler-Leman algorithm and graph neural networks.

### 4.2.2 Preliminaries

In the following, we introduce notation and give the necessary background on graph s. As usual, let $[n] = \{1, \dots, n\} \subset \mathbb{N}$ for $n \geq 1$, and let $\{\!\{\dots\}\!\}$ denote a multiset.

#### 4.2.2.1 Graphs

A *graph* $G$ is a pair $(V, E)$ with a finite set of *vertices* $V$ and a set of *edges* $E \subseteq \{\{u, v\} \subseteq V \mid u \neq v\}$. We denote the set of vertices and the set of edges of $G$ by $V(G)$ and $E(G)$, respectively. For ease of notation, we denote the edge $\{u, v\}$ in $E(G)$ by $(u, v)$ or $(v, u)$. In the case of *directed graphs* the order of the nodes is distinguished and $E \subseteq \{(u, v) \in V \times V \mid u \neq v\}$. A *labeled graph* $G$ is a triple $(V, E, l)$ with a label function $l\colon V(G) \cup E(G) \to \Sigma$, where $\Sigma$ is some finite alphabet. Then $l(v)$ is the *label* of $v$ in $V(G) \cup E(G)$. The

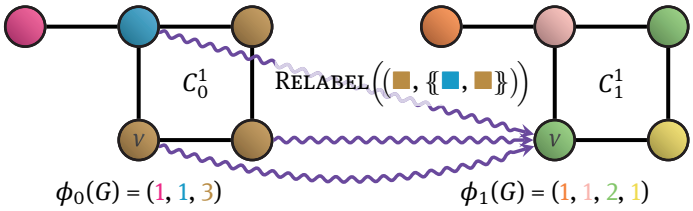



**Fig. 4.6:** Illustration of the coloring scheme of the 1-WL.

*neighborhood* of $v$ in $V(G)$ is denoted by $\delta(v) = N(v) = \{u \in V(G) \mid (v, u) \in E(G)\}$. Let $S \subseteq V(G)$ then $G[S] = (S, E_S)$ with $E_S = \{(u, v) \in E(G) \mid u, v \in S\}$ is the subgraph of $G$ *induced* by $S$. A *tree* is a connected graph without cycles. A *rooted tree* is a tree with a designated vertex called *root* in which the edges are directed such that they point away from the root. Let $p$ be a vertex in a rooted tree; we call its out-neighbors *children* with parent $p$.

We say that two graphs $G$ and $H$ are *isomorphic* if there exists a bijection $\varphi\colon V(G) \to V(H)$ that preserves the edges, i.e., $(u, v)$ is in $E(G)$ if and only if $(\varphi(u), \varphi(v))$ is in $E(H)$ for all $u$ and $v$ in $V(G)$. If $G$ and $H$ are isomorphic, we write $G \simeq H$ and call $\varphi$ an *isomorphism* between $G$ and $H$. Moreover, we call the equivalence classes induced by $\simeq$ *isomorphism types*. In the case of labeled graphs, we additionally require that $l(v) = l(\varphi(v))$ for all $v$ in $V(G)$ and $l((u, v)) = l((\varphi(u), \varphi(v)))$ for all $(u, v)$ in $E(G)$.

#### 4.2.2.2 Kernels

A *kernel* on a non-empty set $\mathcal{X}$ is a symmetric, positive semidefinite function $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Equivalently, a function $k$ is a kernel if there is a *feature map* $\phi\colon \mathcal{X} \to \mathcal{H}$, where $\mathcal{H}$ is a Hilbert space endowed with the inner product $\langle \cdot, \cdot \rangle$, such that $k(x, y) = \langle \phi(x), \phi(y) \rangle$ for all $x$ and $y$ in $\mathcal{X}$. Let $\mathcal{G}$ be the set of all graphs, then a kernel on $\mathcal{G}$ is called a *graph kernel*.

### 4.2.3 The Weisfeiler-Leman Algorithm

The 1-WL is a classical heuristic for the graph isomorphism problem [28, 273, 700]. Here, we formally introduce the 1-WL and its generalization, the $k$-WL, which form the basis for the graph kernels described in the following sections.

#### 4.2.3.1 The 1-dimensional Weisfeiler-Leman Algorithm

Intuitively, the 1-WL aims to capture the structure of a graph by iteratively aggregating labels or *colors* of adjacent vertices. Two equally colored vertices get a different color if their neighborhood is colored differently. See Figure 4.6 for an illustration.

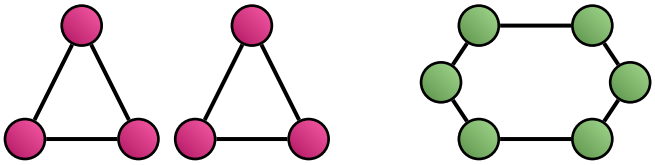


**Fig. 4.7:** Two graphs that cannot be distinguished by the 1-WL.

Formally, let $(G, l)$ be a labeled graph. In each iteration $i \geq 0$, the algorithm computes a *coloring* $C_i^1 \colon V(G) \to \mathbb{S}$, where $\mathbb{S}$ is some arbitrary codomain. In the first iteration, we color the vertices according to the labeling $l$, i.e., $C_0^1(v) = l(v)$ for $v$ in $V(G)$. For $i \geq 0$, $C_{i+1}^1$ is defined by

$$C_{i+1}^1(v) = \textsc{Relabel}\left(C_i^1(v), \{\!\{C_i^1(w) \mid w \in \delta(v)\}\!\}\right).$$

Here, RELABEL is an injection that maps the pair consisting of the current color and the multiset of colors of adjacent vertices to a new color. Hence, two vertices with the same color in iteration $i$ get a different color in the next iteration if the number of neighbors colored with a certain color is different. Observe that it is straightforward to extend the 1-WL to labeled, directed graphs. We run the algorithm until convergence, i.e.,

$$C_i^1(v) = C_i^1(w) \iff C_{i+1}^1(v) = C_{i+1}^1(w),$$

holds for all $v$ and $w$ in $V(G)$. We call the partition of $V(G)$ induced by $C_i^1$ the *stable partition*. For such $i$, we define $C_\infty^1(v) = C_i^1(v)$ for $v$ in $V(G)$. For two graphs $G$ and $H$, we run the algorithm in "parallel" on both graphs. Then the 1-WL *distinguishes* between them if

$$|V(G) \cap (C_\infty^1)^{-1}(c)| \neq |V(H) \cap (C_\infty^1)^{-1}(c)|,$$

for some color $c$ in the codomain of $C_\infty^1$. If the 1-WL distinguishes two graphs, the graphs are not isomorphic.

#### 4.2.3.2 *k*-dimensional Weisfeiler-Leman Algorithm

The 1-WL is not able to distinguish between all pairs of non-isomorphic graphs. See Figure 4.7 for such a pair. The $k$-WL is a natural generalization of the 1-WL, which gets more powerful by coloring $k$-tuples defined over the set of vertices.

Formally, let $G$ be a graph, and let $k \geq 2$. Moreover, let $\mathbf{v}$ be a tuple in $V(G)^k$, then $G[\mathbf{v}]$ is the subgraph induced by the components of $\mathbf{v}$, where the vertices are labeled with integers from $\{1, \ldots, k\}$ corresponding to indices of $\mathbf{v}$. In each iteration $i \geq 0$, the algorithm computes a *coloring* $C_i^k \colon V(G)^k \to \mathbb{S}$, where $\mathbb{S}$ is some arbitrary codomain. In the first iteration ($i = 0$), two tuples $\mathbf{v}$ and $\mathbf{w}$ in $V(G)^k$ get the same color if the map $v_i \mapsto w_i$ is an isomorphism between $G[\mathbf{v}]$ and $G[\mathbf{w}]$. Now, for $i \geq 0$, $C_{i+1}^k$ is defined by

$$C_{i+1}^k(\mathbf{v}) = \textsc{relabel}(C_i^k(\mathbf{v}), M_i(\mathbf{v})),$$

where the multiset

$$M_i(\mathbf{v}) = \big(\{\!\{C_i^k(\phi_1(\mathbf{v}, w)) \mid w \in V(G)\}\!\}, \ldots, \{\!\{C_i^k(\phi_k(\mathbf{v}, w)) \mid w \in V(G)\}\!\}\big), \tag{4.10}$$

and

$$\phi_j(\mathbf{v}, w) = (v_1, \ldots, v_{j-1}, w, v_{j+1}, \ldots, v_k).$$

That is, $\phi_j(\mathbf{v}, w)$ replaces the $j$-th component of the tuple $\mathbf{v}$ with the vertex $w$. We run the algorithm until convergence, i.e.,

$$C_i^k(\mathbf{v}) = C_i^k(\mathbf{w}) \iff C_{i+1}^k(\mathbf{v}) = C_{i+1}^k(\mathbf{w}),$$

for all $\mathbf{v}$ and $\mathbf{w}$ in $V(G)^k$ holds, and call the partition of $V(G)^k$ induced by $C_i^k$ the *stable partition*. For such $i$, we define $C_\infty^k(\mathbf{v}) = C_i^k(\mathbf{v})$ for $\mathbf{v}$ in $V(G)^k$. The procedure of determining if two graphs are non-isomorphic is the same as for the 1-WL. With increasing $k$ the algorithm gets more and more powerful [117]. That is, for each $k \geq 2$ there exists a pair of graphs that the $k$-WL cannot distinguish but the $(k+1)$-WL can.

Let $A$ and $B$ be two heuristics for the graph isomorphism problem, e.g., the $k$-WL, then we write $A \sqsubseteq B$ ($A \sqsubset B$, $A \equiv B$), if algorithm $A$ is more powerful (strictly more powerful, equally powerful) than $B$ in terms of distinguishing non-isomorphic graphs. Using this notation we write

$$(k+1)\text{-WL} \sqsubset k\text{-WL},$$

for $k \geq 2$, to state the result mentioned in the last paragraph.

### 4.2.4 Kernels Based on the Weisfeiler-Leman Algorithm

The Weisfeiler-Leman algorithm forms the basis for some of the most successful graph kernels. Here, we give an overview on kernels based on the 1-WL, followed by kernels based on the $k$-WL. Moreover, we survey other kernels related to the Weisfeiler-Leman paradigm.

#### 4.2.4.1 Weisfeiler-Leman Subtree Kernel

The idea of the *Weisfeiler-Leman subtree graph kernel* [627] is to compute the 1-WL for $h \geq 0$ iterations resulting in a label function $C_i^1 \colon V(G) \to \mathbb{S}_i$ for each iteration $0 \leq i \leq h$. Now after each iteration, we compute a *feature vector* $\phi_i(G)$ in $\mathbb{R}^{|\mathbb{S}_i|}$ for each graph $G$. Each component $\phi_i(G)_c$ counts the number of occurrences of vertices labeled with $c$ in $\mathbb{S}_i$. The overall feature vector $\phi_{\mathrm{WL}}(G)$ is defined as the concatenation of the feature vectors of all $h$ iterations, i.e.,

$$\phi_{\mathrm{WL}}(G) = \big[\phi_0(G), \ldots, \phi_h(G)\big].$$

The Weisfeiler-Leman subtree kernel for $h$ iterations is then computed as

$$k_{\mathrm{WL}}(G, H) = \langle \phi_{\mathrm{WL}}(G), \phi_{\mathrm{WL}}(H) \rangle,$$

where $\langle \cdot, \cdot \rangle$ denotes the standard inner product. The running time for a single feature vector computation is in $\mathcal{O}(hm)$ and $\mathcal{O}(Nhm + N^2hn)$ for the computation of the gram matrix for a set of $N$ graphs [627], where $n$ and $m$ denote the maximum number of vertices and edges over all $N$ graphs, respectively.

#### 4.2.4.2 Weisfeiler-Leman Optimal Assignment Kernels

The Weisfeiler-Leman subtree kernel counts pairs of vertices with the same label. A different approach is to *assign* each vertex of $G$ to a vertex of $H$. Constructing an assignment that maximizes the structural overlap and agreement of vertex attributes is a general concept for comparing graphs and also forms the basis of *graph matching* or *network alignment*. This principle was proposed to obtain graph kernels, where the similarity between two vertices is determined by an arbitrary base kernel [236]. However, it was soon observed that the resulting similarity measure is in general not positive semidefinite [685]. Subsequent research has identified a specific class of base kernels, for which the similarity derived from optimal assignments is guaranteed to be a valid kernel, i.e., positive semidefinite [395]. We summarize the theory of valid assignment kernels and then describe how a suitable base kernel can be obtained from the 1-WL.

**Valid Optimal Assignment Kernels** We consider the general setting, where the elements of two sets are to be assigned to each other. Let $[\mathcal{X}]^n$ denote the set of all $n$-element subsets of a set $\mathcal{X}$ and $\mathfrak{B}(X, Y)$ the set of all bijections between $X$ and $Y$ in $[\mathcal{X}]^n$ for $n$ in $\mathbb{N}$. The *optimal assignment kernel* $K^k_{\mathfrak{B}}$ on $[\mathcal{X}]^n$ is defined as

$$K^k_{\mathfrak{B}}(X, Y) = \max_{B \in \mathfrak{B}(X,Y)} \sum_{(x,y) \in B} k(x, y), \tag{4.11}$$

where $k$ is a *base kernel* on $\mathcal{X}$. For the application to sets of different cardinality, the smaller set can be augmented by dummy elements $d$ with $k(d, \cdot) = 0$.

Similar to the concept of an ultrametric, which must satisfy the strong triangle inequality, the so-called *strong kernel* was introduced as a kernel satisfying $k(x, y) \geq \min\{k(x, z), k(z, y)\}$ for all $x, y, z$ in $\mathcal{X}$. It was shown that the function $K^k_{\mathfrak{B}}$ is a valid kernel if $k$ is a strong kernel [395]. Strong kernels are equivalent to kernels obtained from a hierarchical partition of their domain. Formally, let $T$ be a rooted tree such that the leaves of $T$ are the elements of $\mathcal{X}$ and $\omega \colon V(T) \to \mathbb{R}_{\geq 0}$ a weight function. We refer to the tuple $(T, \omega)$ as a *hierarchy*. A hierarchy on $\mathcal{X}$ induces a similarity $k(x, y)$ for $x$ and $y$ in $\mathcal{X}$ as follows. For $v$ in $V(T)$ let $P(v) \subseteq V(T)$ denote the set of vertices in $T$ on the path from $v$ to the root $r$. Then the similarity between $x$ and $y$ in $\mathcal{X}$ is

$$k(x, y) = \sum_{v \in P(x) \cap P(y)} \omega(v).$$

For every strong kernel $k$ there is a hierarchy that induces $k$ and, vice versa, every hierarchy induces a strong kernel [395].

The optimal assignment kernel of Equation 4.11 can be computed in linear time from the hierarchy $(T, \omega)$ of the base kernel $k$ by histogram intersection. For a node $v$ in $V(T)$ and a set $X \subseteq \mathcal{X}$, let $X_v$ denote the subset of $X$ that is contained in the subtree rooted at $v$. Then the optimal assignment kernel is

$$K_{\mathfrak{B}}^{k}(X, Y) = \sum_{v \in V(T)} \min\{|X_v|, |Y_v|\} \cdot \omega(v), \tag{4.12}$$

which can be seen as the histogram intersection kernel for appropriately defined histograms representing the sets $X$ and $Y$ under the strong base kernel $k$ [395].

**Optimal Assignment Kernels from the 1-WL** The 1-WL produces a hierarchy on the vertices of a (set of) graphs, where the $i$th level consists of nodes $\mathbb{S}_{i+1}$ with an artificial root at level 0. The parent-child relationships are given by the color refinement process, where the root has children $\mathbb{S}_1$. This hierarchy with a uniform weight function induces the strong base kernel

$$k(u, v) = \sum_{i=0}^{h} k_\delta(C_i^1(u), C_i^1(v)), \quad k_\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

on the vertices. The kernel counts the number of iterations required to assign different colors to the vertices and reflects the extent to which the vertices have a structurally similar neighborhood. The optimal assignment kernel with this base kernel is referred to as *Weisfeiler-Leman optimal assignment kernel* and was shown to achieve better accuracy results in many classification experiments than the Weisfeiler-Leman subtree kernel. Moreover, the weights of the hierarchy associated with a strong base kernel can be optimized via multiple kernel learning [396].

### 4.2.4.3 Wasserstein Weisfeiler-Leman Graph Kernels

Related to assignment kernels are techniques based on the Wasserstein distance. Given two vectors $a$ and $b$ in $\mathbb{R}_+^n$ with entries that sum to the same value and a ground cost matrix $D$ in $\mathbb{R}_+^{n \times n}$, the *Wasserstein distance* (or *earth mover's distance*, *optimal transport distance*)[5] is

$$W(a, b) = \min_{T \in \Gamma(a,b)} \langle T, D \rangle, \quad \Gamma(a, b) = \left\{ T \in \mathbb{R}_+^{n \times n} \colon T\mathbf{1} = a, T^\top \mathbf{1} = b \right\}, \tag{4.14}$$

where $\Gamma(a, b)$ is the set of so-called *transport plans* and $\langle \cdot, \cdot \rangle$ denotes the Frobenius dot product. Although $\Gamma(a, b)$ allows doubly stochastic matrices, the Wasserstein distance

5 Depending on the context, slightly different definitions are used in the literature. Often, they require that $a$ and $b$ be distributions.

is a generalization of the min-version of Equation 4.11. The ground cost matrix, providing the dissimilarity between entries of $a$ and $b$, has a role analogous to the base kernel.

The Wasserstein distance can be applied to the vertices of two graphs using ground costs obtained by 1-WL [663]. The entries of $D$ are given by

$$d(u, v) = \frac{1}{h+1} \sum_{i=0}^{h} \rho(C_i^1(u), C_i^1(v)), \quad \rho(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise.} \end{cases} \tag{4.15}$$

Equation 4.15 is closely related to Equation 4.13 and can be regarded as its associated normalized distance. The Wasserstein distance $W(a, b)$ of Equation 4.14 is then combined with a distance-based kernel [283], specifically a variant of the Laplacian kernel. The resulting function was shown to be positive semidefinite. The authors also proposed extending the 1-WL to continuous attributes replacing discrete colors with real-valued vectors. Then, the ground costs of the Wasserstein distance are obtained from the Euclidean distance between these vectors. However, in this case, it is not guaranteed that the resulting function is positive semidefinite.

The Weisfeiler-Leman assignment kernel and the Wasserstein Weisfeiler-Leman kernel employ the 1-WL and improve the classification accuracy observed in practice on many datasets over the Weisfeiler-Leman subtree kernel. However, they are not more powerful in distinguishing non-isomorphic graphs. One approach to obtain kernels more expressive in this sense is to use the $k$-WL.

#### 4.2.4.4 Kernels Based on the $k$-WL

The $k$-WL was also used to derive graph kernels [504, 506]. Essentially, the kernel computation works the same way as in the 1-dimensional case, i.e., a feature vector is computed for each graph based on color counts. To make the algorithm more scalable, the authors of [506] resorted to color all subgraphs on $k$ vertices instead of all $k$-tuples, resulting in a less expressive algorithm. Moreover, the authors proposed that only a subset of the original neighbors be considered to exploit the sparsity of the underlying graph. Further, they offered a sampling-based approximation algorithm to speed up the kernel computation for a large graph, showing that the kernel can be approximated in constant time, i.e., independent of the number of vertices and edges, with an additive approximation error. Finally, they showed empirically that the proposed kernel beats the Weisfeiler-Leman subtree kernel on a subset of tested benchmark datasets.

Similarly, Morris, Rattan, and Mutzel [504] proposed graph kernels based on the $k$-WL. Again they proposed a local variant of the $k$-WL, named $\delta$-$k$-LWL, that only considers a subset of the original neighborhood. However, they considered $k$-tuples and proved that a variant of their method is slightly more powerful than the original $k$-WL while taking the original graph's sparsity into account. That is, instead of Equation 4.10, the $\delta$-$k$-LWL uses

$$M_i^{\delta}(\mathbf{v}) = \left( \{\!\{ C_i^{k,\delta}(\phi_1(\mathbf{v}, w)) \mid w \in \delta(v_1) \}\!\}, \ldots, \{\!\{ C_i^{k,\delta}(\phi_k(\mathbf{v}, w)) \mid w \in \delta(v_k) \}\!\} \right).$$

Hence, the labeling function is defined by

$$C^{k,\delta}_{i+1}(\mathbf{v}) = \textsc{relabel}(C^{k,\delta}_{i}(\mathbf{v}), M^{\delta}_{i}(\mathbf{v})). \tag{4.16}$$

Empirically, they show that one of their variants of the $k$-WL achieves a new state of the art across many standard benchmark datasets while being several orders of magnitude faster than the $k$-WL.

#### 4.2.4.5 Other Kernels Based on the Weisfeiler-Leman Algorithm

In the following, we survey other graph kernels that build on the Weisfeiler-Leman paradigm.

**Weisfeiler-Leman Kernel Framework** A general technique to modify and strengthen graph kernels is to modify their labels such that additional information is encoded. This can be achieved by computing the first $h \geq 0$ colors $C^1_0, \ldots, C^1_h$ of the 1-WL [627]. Then, given an arbitrary graph kernel used as base kernel, the corresponding Weisfeiler-Leman kernel is the sum of the base kernel applied to pairs of graphs with the label $C^1_i$ for $i$ in $\{0, \ldots, h\}$. The *Weisfeiler-Leman subtree kernel* described in Section 4.2.4.1 is obtained for a base kernel counting common vertex labels. Another instance of the approach commonly used is obtained by using the shortest-path kernel [65].

**Hash Graph Kernel Framework** In chem- or bioinformatics, edges and vertices of graphs are often annotated with real-valued information, e.g., physical measurements [508]. Previous graph kernels that can take these attributes into account are relatively slow and employ the kernel trick [65, 222, 394]. Therefore, these approaches do not scale to large graphs and datasets. Moreover, kernels such as the Weisfeiler-Leman subtree kernel cannot adequately deal with such continuous information due to its discrete nature. To overcome this, the hash graph kernel framework was introduced [507]. The idea is to iteratively turn the continuous attributes into discrete labels using randomized hash functions. This allows the application of fast, explicit graph feature maps, e.g., the Weisfeiler-Leman subtree kernel, which are limited to discrete annotations. In each iteration, the algorithm samples new hash functions and computes the feature map. Finally, the feature maps of all iterations are combined into one feature map. In order to obtain a meaningful similarity between attributes in $\mathbb{R}^d$, one requires that the probability of collision $\Pr[h_1(x) = h_2(y)]$ of two independently chosen random hash functions $h_1, h_2 \colon \mathbb{R}^d \to \mathbb{N}$ equals an adequate kernel on $\mathbb{R}^d$. Equipped with such a hash function, approximation results were derived for several state-of-the-art kernels that can handle continuous information [507]. In particular, we derived a variant of the Weisfeiler-Leman subtree kernel, which can handle continuous attributes. The extensive experimental study showed that instances of the hash kernel framework achieve state-of-the-art classification accuracies while being orders of magnitudes faster than kernels that were specifically designed to handle continuous information.

**Neighborhood Aggregation in Graph Kernels** The idea of neighborhood aggregation is widely used, and there are often subtle differences in definition. For completeness, we mention several graph kernels following this general idea. The *neighborhood hash kernel* [314] is similar in spirit to the Weisfeiler-Leman subtree kernel, but represents simple labels by bit-vectors and uses logical operations and hashing to encode the direct neighborhood for efficiency. Propagation kernels proposed in [528] provide a generic framework to define kernels on graphs based on an information propagation scheme for labels and attributes. Propagation, e.g., based on random walks, is performed individually on the two input graphs and a kernel is obtained by comparing label distributions after every propagation step. In the case of continuous (multi-dimensional) attributes, a single hash function is used to obtain a discrete label. In [537] a general message passing framework for kernels was proposed, where the concept of optimal assignments (see Section 4.2.4.2) was introduced in the neighborhood aggregation step. *Persistent Weisfeiler-Leman kernels* [597] combine 1-WL with persistent homology to extract topological features such as cycles. Recent theoretical results that link 1-WL to graph homomorphisms [167] were used to define graph kernels that have the same expressive power as the 1-WL, but a different feature space [533].

### 4.2.5 Graph Neural Networks and Their Connection to the Weisfeiler-Leman Algorithm

GNNs emerged as an alternative to graph kernels for graph classification and other machine learning tasks on graphs such as node classification or regression. Standard GNNs can be viewed as a neural version of the 1-WL, where colors are replaced by continuous feature vectors and neural networks are used to aggregate over node neighborhoods [252, 292, 375]. In effect, the GNN framework can be viewed as implementing a continuous form of graph-based "message passing", where local neighborhood information is aggregated and passed on to the neighbors [252]. By deploying a trainable neural network to aggregate information in local node neighborhoods, GNNs can be trained in an end-to-end fashion together with the parameters of the classification or regression algorithm, possibly allowing for greater adaptability and better generalization compared with the kernel counterpart of the classical 1-WL.

A GNN model consists of a stack of neural network layers, where each layer aggregates local neighborhood information, i.e., features of neighbors, around each node and then passes this aggregated information on to the next layer. See Figure 4.8 for an illustration of the architecture.

In the following, we formally define GNNs and outline their connection to the Weisfeiler-Leman algorithm. Let $G = (V, E, l)$ be a labeled graph with an initial node coloring $f^{(0)} \colon V(G) \to \mathbb{R}^{1\times d}$ that is *consistent* with $l$. This means that each node $v$ is annotated with a feature $f^{(0)}(v)$ in $\mathbb{R}^{1\times d}$ such that $f^{(0)}(u) = f^{(0)}(v)$ if and only if $l(u) = l(v)$. Alternatively, $f^{(0)}(v)$ can be an arbitrary real-valued feature vector associated with $v$. Examples include continuous atomic properties in chemoinformatic applications or

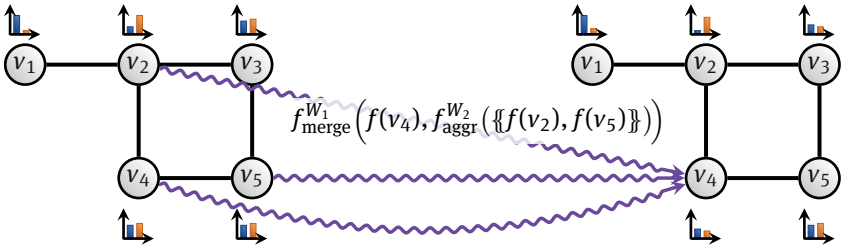



**Fig. 4.8:** Illustration of the feature aggregation scheme of GNNs. The new feature of the node $v_4$ is computed from its old feature and the features of its neighbors $v_2$ and $v_5$.

vector representations of text in social network applications. A basic GNN model can be implemented as follows [292]. In each layer $t > 0$, we compute a new feature

$$f^{(t)}(v) = \sigma\left(f^{(t-1)}(v) \cdot W_1^{(t)} + \sum_{w \in N(v)} f^{(t-1)}(w) \cdot W_2^{(t)}\right) \tag{4.17}$$

in $\mathbb{R}^{1\times e}$ for $v$, where $W_1^{(t)}$ and $W_2^{(t)}$ are parameter matrices from $\mathbb{R}^{d\times e}$ , and $\sigma$ denotes a component-wise non-linear function, e.g., a sigmoid or a ReLU.[6]

One may also replace the sum defined over the neighborhood in the above equation by different permutation-invariant, differentiable functions, e.g., mean or max, and one may substitute the outer sum by, say, a column-wise vector concatenation [252]. Thus, in full generality a new feature $f^{(t)}(v)$ is computed as

$$f_{\text{merge}}^{W_1}\left(f^{(t-1)}(v), f_{\text{aggr}}^{W_2}\left(\{\!\{f^{(t-1)}(w) \mid w \in N(v)\}\!\}\right)\right), \tag{4.18}$$

where $f_{\text{aggr}}^{W_2}$ aggregates over the set of neighborhood features and $f_{\text{merge}}^{W_1}$ merges the node's representations from step $(t-1)$ with the computed neighborhood features. Both $f_{\text{aggr}}^{W_2}$ and $f_{\text{merge}}^{W_1}$ may be arbitrary differentiable functions, e.g., neural networks, and, by analogy to Equation 4.17, we denote their parameters as $W_1$ and $W_2$, respectively.

A vector representation $f_{GNN}$ over the whole graph can be computed by aggregating the vector representations computed for all nodes, e.g.,

$$f_{GNN}(G) = \sum_{v \in V(G)} f^{(T)}(v),$$

where $T > 0$ denotes the last layer. More refined approaches use differential pooling operators based on sorting [736] or soft assignments [724]. To adapt the parameters $W_1$ and $W_2$ of Equations 4.17 and 4.18 to a given data distribution, they are optimized in an end-to-end fashion (usually via stochastic gradient descent) together with the parameters of a neural network used for classification or regression. Efficient GPU-based implementations of many GNN architectures can be found in [225] and [696]. See also Section 4.3.

**6** For clarity of presentation we omit biases.

#### 4.2.5.1 Connections to the Weisfeiler-Leman Algorithm

A recent line of work [468, 509, 714] connects the power or expressivity of GNNs to that of the Weisfeiler-Leman algorithm. The results show that GNN architectures generally do not have more power to distinguish between non-isomorphic (sub)graphs than the 1-WL.

Formally, let $(G, l)$ be a labeled graph, and let $\mathbf{W}^{(t)} = \left(W_1^{(t')}, W_2^{(t')}\right)_{t' \leq t}$ denote the GNN parameters given by Equations 4.17 and 4.18 up to iteration $t$. We encode the initial labels $l(v)$ by vectors $f^{(0)}(v)$ in $\mathbb{R}^{1\times d}$ using a 1-hot encoding. The first theoretical result shown in [509] states that the GNN architectures do not have more power to distinguish between non-isomorphic (sub-)graphs than the 1-WL. More formally, let $f_{\text{merge}}^{W_1}$ and $f_{\text{aggr}}^{W_2}$ be any two functions chosen in Equation 4.18. For every encoding of the labels $l(v)$ as vectors $f^{(0)}(v)$, and for every choice of $\mathbf{W}^{(t)}$, the coloring $C_i^1$ of the 1-WL always refines the coloring $f^{(t)}$ induced by a GNN parameterized by $\mathbf{W}^{(t)}$.

**Theorem 7.** *Let $(G, l)$ be a labeled graph. Then for all $t \geq 0$ and for all choices of initial colorings $f^{(0)}$ consistent with $l$, and weights $\mathbf{W}^{(t)}$,*

$$c_l^{(t)} \sqsubseteq f^{(t)}\,.$$

The second result of [509] states that there exists a sequence of parameter matrices $\mathbf{W}^{(t)}$ such that GNNs have the same power in terms of distinguishing non-isomorphic (sub-)graphs as the 1-WL. This even holds for the simple architecture Equation 4.17, provided we choose the encoding of the initial labeling $l$ in such a way that linearly independent vectors encode different labels.

**Theorem 8.** *Let $(G, l)$ be a labeled graph. Then for all $t \geq 0$ there exists a sequence of weights $\mathbf{W}^{(t)}$, and a 1-GNN architecture such that*

$$c_l^{(t)} \equiv f^{(t)}\,.$$

Hence, in the light of the above results, GNNs may be viewed as an extension of the 1-WL, which in principle have the same power but are more flexible in their ability to adapt to the learning task at hand and can handle continuous node features.

#### 4.2.5.2 Higher-order Graph Neural Networks

The above results also have been lifted to the $k$-dimensional case. For example, Maron, Ben-Hamu, Serviansky, and Lipman [468] devised an architecture based on simple matrix operations that has the same power as the 3-WL. In a recent work, Morris, Rattan, and Mutzel [504] devised neural architectures, denoted $\delta$-$k$-LGNN, that resemble the construction for GNNs.

Formally, given a labeled graph $G$, let each tuple $\mathbf{v}$ in $V(G)^k$ be annotated with an initial feature $f^{(0)}(\mathbf{v})$ determined by the isomorphism type of $G[\mathbf{v}]$. In each layer $t > 0$,

we compute a new feature $f^{(t)}(\mathbf{v})$ as

$$f_{\text{merge}}^{W_1}\Big(f^{(t-1)}(\mathbf{v}), f_{\text{agg}}^{W_2}\big(\{\!\{f^{(t-1)}(\phi_1(\mathbf{v}, w)) \mid w \in \delta(v_1)\}\!\}, \ldots, \{\!\{f^{(t-1)}(\phi_k(\mathbf{v}, w)) \mid w \in \delta(v_k)\}\!\}\big)\Big),$$

in $\mathbb{R}^{1\times e}$ for a tuple $\mathbf{v}$, where $W_1^{(t)}$ and $W_2^{(t)}$ are learnable parameter matrices from $\mathbb{R}^{d\times e}$.[7] Moreover, $f_{\text{merge}}^{W_2}$ and the permutation-invariant $f_{\text{agg}}^{W_1}$ can be arbitrary differentiable functions, responsible for merging and aggregating the relevant feature information, respectively. Note that one can naturally handle discrete node and edge labels as well as directed graphs. The following result shown in [504] demonstrates the expressive power of the $\delta$-$k$-LGNN in terms of distinguishing non-isomorphic graphs.

**Theorem 9.** *Let $(G, l)$ be a labeled graph. Then for all $t \geq 0$ there exists a sequence of weights $\mathbf{W}^{(t)}$ such that*

$$C_t^{k,\delta}(\mathbf{v}) = C_t^{k,\delta}(\mathbf{w}) \iff f^{(t)}(\mathbf{v}) = f^{(t)}(\mathbf{w}).$$

*Hence, for all graphs, the following holds for all $k \geq 1$:*

$$\delta\text{-}k\text{-}LGNN \equiv \delta\text{-}k\text{-}LWL.$$

### 4.2.6 Conclusion and Future Work

The Weisfeiler-Leman method has been studied for decades in graph theory and recently turned out to be an essential technique in machine learning with graphs [505], achieving high accuracy on many real-world datasets [508]. While the Weisfeiler-Leman algorithm's expressivity limits machine learning methods to distinguishing non-isomorphic graphs, the generalization abilities of such methods are understood to a lesser extent, indicating an avenue for future research. Moreover, heterogeneous networks with different edge types or graphs annotated with temporal information will become increasingly important. The adaption of the Weisfeiler-Leman paradigm to such settings has only recently been considered, e.g., for temporal graphs [544], and the development of new suitable learning methods has only just begun.

7 For clarity of presentation we omit biases.

## 4.3 Deep Graph Representation Learning

*Matthias Fey*
*Frank Weichert*

**Abstract:** Learning with graph-structured data such as molecules, social, biological, and financial networks, requires effective representations that successfully capture their rich structural properties. In recent years, numerous approaches have been proposed for machine learning on graphs — most notably, approaches based on graph kernels and *Graph Neural Networks (GNNs)*. Graph neural networks exploit relational inductive biases of the underlying data by following a differentiable neural message passing scheme, and show-case promising performance on a variety of different tasks due to their expressive power in capturing different graph structures. However, despite the indisputable potential of GNNs in learning such representations, one of the challenges that have so far precluded their wide adoption in industrial and social applications is the difficulty to scale them to large graphs. In particular, the embedding of a given node depends recursively on all its neighbor's embeddings, leading to high inter-dependency between nodes that grows exponentially with respect to the number of layers.

Here, we demonstrate the generality of message passing through a unified framework that is suitable for a wide range of operators and learning tasks. This generality of message passing led to the development of *PyTorch Geometric*, a well-known deep learning library for implementing and working with graph-based neural network building blocks. Furthermore, we discuss scalable approaches for applying graph neural networks to large-scale graphs. In particular, we show that scalable approaches based on sub-sampling of edges or non-trainable propagations weaken the expressive power of message passing. In order to overcome this restriction, we present *GNN AutoScale*, a framework for scaling arbitrary message passing neural networks to large graphs. GNN AutoScale prunes entire sub-trees of the computation graph by utilizing historical node embeddings from prior training iterations while provably being able to maintain the expressive power of the original architecture.

### 4.3.1 Introduction

Graphs are widely used for abstracting complex systems of interacting objects, such as social networks, knowledge graphs, molecular graphs, and biological networks, as well as for modeling 3D objects, manifolds, and source code [320]. To develop successful machine learning models in these domains, we need techniques that can exploit the rich information inherent in the graph structure, as well as the feature information contained within a graph's nodes and edges. Recently, graph neural networks emerged

as a powerful approach and the de facto standard for representation learning on graphs. GNNs are able to capture local graph structure and feature information in a trainable fashion to derive powerful node representations suitable for a given task at hand [291, 455]. To achieve this, they follow a simple neighborhood aggregation procedure or neural message passing scheme motivated by two major perspectives: The generalization of classical CNNs to irregular domains, and their strong relations to the Weisfeiler-Lehman algorithm [226, 509, 715] (see Section 4.2.5).

The recent work in the fields of *geometric deep learning* and *relational representation learning* provides a large number of graph-based operators, which allows for precise control of the properties of extracted graph-based features [134, 225, 252, 292, 375, 378, 588, 683, 697, 714, 715]. Nonetheless, all those recent operators can be described by a simple message passing formulation, leading to a unified framework suitable across a wide range of operators and learning tasks [252]. The generality of message passing led to the development of the *PyTorch Geometric* library, a deep learning framework for implementing and working with graph-based neural networks [225].

While GNNs have become better understood and models have become more sophisticated, advancements in this field should be more noticeable with access to increasing data. However, applying mini-batch training of GNNs is challenging since the embedding of a given node depends recursively on all its neighbor's embeddings, leading to high inter-dependency between nodes that grows exponentially with respect to the number of layers [455]. Several recent works address this problem via different sampling techniques (leading to sub-sampling of edges) [455, 600], or by decoupling propagations from predictions [234, 321, 378, 710, 726]. Although empirical results suggest that the aforementioned methods can scale GNN training to large graphs, these techniques are either restricted to shallow networks, non-exchangeable operators, or reduced expressivity. In particular, existing approaches consider only specific GNN operators and it is not yet well known whether these techniques can be successfully applied to the wide range of GNN architectures available.

In the next sections, we will discuss and introduce the aforementioned general neural message passing framework, and show how common GNN operators fit into this scheme. We proceed by introducing the PyTorch Geometric library [225], which makes it easy to implement those GNN operators in practice. Furthermore, we present our *GNN AutoScale* framework for scaling arbitrary message passing GNNs to large-scale graphs [224].

### 4.3.2 Representation Learning on Graphs via Neural Message Passing

We begin by refining the general neural message passing scheme from Section 4.2.5 that is utilized in state-of-the-art graph neural networks and, along the way, introduce the necessary notation and background. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ or $\boldsymbol{A} \in \{0, 1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ denote a *graph* with node feature vectors $\boldsymbol{x}_v$ for all $v \in \mathcal{V}$ and (optional) edge features $\boldsymbol{e}_{v,w}$ in case

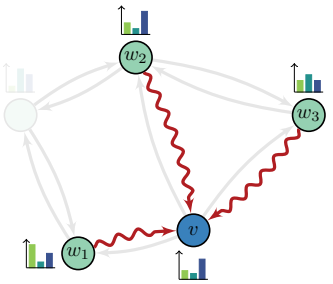



**Fig. 4.9:** Message passing flow in a GNN layer. Each direct neighbor of a node crafts a message that is sent along the given edge. Each node aggregates their incoming messages to update its current node representation.

$(v, w) \in \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Here, we are mostly interested in learning final node representations $\boldsymbol{h}_v \in \mathbb{R}^D$ for all $v \in \mathcal{V}$ in an end-to-end fashion that are suitable for a given downstream task (such as node, link, or graph classification). In node classification, each node $v \in \mathcal{V}$ is associated with a label $y_v$, and the goal is to learn a representation $\boldsymbol{h}_v$ from which $y_v$ can be easily predicted. In link prediction, we want to find the missing links in an incomplete graph, and we can directly use $\boldsymbol{h}_v$ and $\boldsymbol{h}_w$, $v, w, \in \mathcal{V}$, for predicting the existence of an edge between the given node pair. In graph classification, each individual graph is associated with a label $y$, and we can use $\{\{\boldsymbol{h}_v : v \in \mathcal{V}\}\}$ alltogether to predict the label $y$ in a permutation-invariant fashion.

Graph neural networks operate on graph-structured data $\mathcal{G}$ by following a *neural message passing scheme*, where a representation of a node is iteratively updated by aggregating representations of its neighbors [252]. After $L$ iterations of aggregation, the representation of a node captures both structural *and* feature information within its $L$-hop neighborhood. Formally, the $(\ell + 1)$-th layer of a GNN is defined as

$$\boldsymbol{h}_v^{(\ell+1)} = \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)} \left( \boldsymbol{h}_v^{(\ell)}, \left\{\!\!\left\{ \left( \boldsymbol{h}_w^{(\ell)}, \boldsymbol{h}_v^{(\ell)}, \boldsymbol{e}_{w,v}^{(\ell)} \right) : w \in \mathcal{N}(v) \right\}\!\!\right\} \right) \tag{4.19}$$

$$= \textsc{Update}_{\boldsymbol{\theta}}^{(\ell+1)} \left( \boldsymbol{h}_v^{(\ell)}, \bigoplus_{w \in \mathcal{N}(v)} \textsc{Message}_{\boldsymbol{\theta}}^{(\ell+1)} \left( \boldsymbol{h}_w^{(\ell)}, \boldsymbol{h}_v^{(\ell)}, \boldsymbol{e}_{w,v}^{(\ell)} \right) \right) \tag{4.20}$$

where $\boldsymbol{h}_v^{(\ell)}$ represents the feature vector of node $v$ obtained in layer $\ell$ and $\mathcal{N}(v) = \{w : (w, v) \in \mathcal{E}\}$ defines the neighborhood set of $v$. We initialize $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$. Since different nodes can have identical feature vectors, a GNN operates on *multisets* $\{\!\{\ldots\}\!\}$, defined as a 2-tuple $\mathcal{X} = (\mathbb{P}^d, c)$, where $\mathbb{P}^d$ denotes the underlying set of $\mathcal{X}$ and $c : \mathbb{P}^d \to \mathbb{N}_{\geq 1}$ counts its multiplicity. A general illustration of this message passing flow is given in Figure 4.9. Most recent GNN operators $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$ can be decomposed into differentiable and parametrized $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ functions parametrized by weights $\boldsymbol{\theta}$, as well as permutation-invariant aggregation functions $\bigoplus$, e.g., taking the sum, mean or maximum of features [225]. Message and Update can be chosen in different ways, depending on the task at hand. For example, Message functions can transform incoming features

either linearly or non-linearly [252, 588, 697]; aggregative functions can model static [714], structure-dependent [375], or data-dependent aggregations [683]; and UPDATE is typically used to preserve central node information via skip-connections [292] or residuals [134, 378].

Ideally, a maximally powerful GNN could distinguish non-isomorphic graph structures by mapping them to different representations in the embedding space. In recent studies [509, 714], it has been shown that the representational power of GNNs is bounded by the capacity of the *Weisfeiler-Leman (WL)* graph isomorphism test [701], (see Section 4.2.3), which uniquely refines the coloring of a node $c_v^{(\ell)} \colon \mathcal{V} \to \Sigma$ based on the colors of their neighbors. In fact, a GNN's expressiveness is equivalent to the WL test if all its layers $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$ are injective, i.e., if they *never* map two different neighborhoods to the same representation. As a result, numerous GNN operators have been proposed that are equally powerful as the WL test [155, 714], as well as higher-order variants to increase their representational power even further [67, 227, 468, 504, 509, 519] (see Section 4.2). We now briefly review how current state-of-the-art GNN operators fit into the given neural message passing scheme (omitting final non-linearities due to simplicity).

**Graph Neural Networks (GNN)** [375] can be considered as one of the pioneers of graph-structured deep learning methods, and they are motivated by a first-order approximation of spectral graph convolutions. Its underlying GNN operator uses a symmetrically normalized mean aggregation of linearly transformed neighboring node representations

$$\boldsymbol{h}_v^{(\ell+1)} = \overbrace{\frac{1}{c_{v,v}} \boldsymbol{W}\boldsymbol{h}_v^{(\ell)} + \underbrace{\sum_{w \in \mathcal{N}(v)}}_{\bigoplus} \underbrace{\frac{1}{c_{w,v}} \boldsymbol{W}\boldsymbol{h}_w^{(\ell)}}_{\textsc{Message}_{\boldsymbol{\theta}}^{(\ell+1)}}}^{\textsc{Update}_{\boldsymbol{\theta}}^{(\ell+1)}}, \tag{4.21}$$

where $c_{w,v} = \sqrt{\deg(w) + 1}\sqrt{\deg(v) + 1}$ with $\deg(\cdot)$ denoting node degree, and $\boldsymbol{W}$ being a trainable weight matrix.

**Graph Attention Networks (GAT)** [683] builds upon the idea of GCNs where the structure-dependent normalization coefficients are replaced by an anisotropic, learnable aggregation guided by attention

$$\boldsymbol{h}_v^{(\ell+1)} = \overbrace{\alpha_{v,v} \boldsymbol{W}\boldsymbol{h}_v^{(\ell)} + \underbrace{\sum_{w \in \mathcal{N}(v)}}_{\bigoplus} \underbrace{\alpha_{w,v} \boldsymbol{W}\boldsymbol{h}_w^{(\ell)}}_{\textsc{Message}_{\boldsymbol{\theta}}^{(\ell+1)}}}^{\textsc{Update}_{\boldsymbol{\theta}}^{(\ell+1)}}, \tag{4.22}$$

where attention coefficients are computed via

$$\alpha_{w,v} = \frac{\exp\left(\text{LeakyReLU}\left(\boldsymbol{a}^\top \left[\boldsymbol{W}\boldsymbol{h}_v^{(\ell)}, \boldsymbol{W}\boldsymbol{h}_w^{(\ell)}\right]\right)\right)}{\sum_{k \in \mathcal{N}(v) \cup \{v\}} \exp\left(\text{LeakyReLU}\left(\boldsymbol{a}^\top \left[\boldsymbol{W}\boldsymbol{h}_v^{(\ell)}, \boldsymbol{W}\boldsymbol{h}_k^{(\ell)}\right]\right)\right)}. \tag{4.23}$$

with additional trainable parameters $\boldsymbol{a}$.

**Spline-Based Convolutional Neural Networks** [226] utilize edge information $\boldsymbol{e}_{w,v}$ to learn a data-dependent filter matrix

$$\boldsymbol{h}_v^{(\ell+1)} = \overbrace{\boldsymbol{W}\boldsymbol{h}_v^{(\ell)} + \underbrace{\sum_{w \in \mathcal{N}(v)}}_{\bigoplus} \underbrace{\boldsymbol{g}_{\boldsymbol{\theta}}(\boldsymbol{e}_{w,v})\boldsymbol{h}_w^{(\ell)}}_{\textsc{Message}_{\boldsymbol{\theta}}^{(\ell+1)}}}^{\textsc{Update}_{\boldsymbol{\theta}}^{(\ell+1)}} \tag{4.24}$$

via a parametrized and continuous B-Spline kernel function $\boldsymbol{g}_{\boldsymbol{\theta}}(\cdot)$.

**Graph Isomorphism Networks (GIN)** [714] make use of sum aggregation and MLPs to obtain a maximally powerful GNN operator

$$\boldsymbol{h}_v^{(\ell+1)} = \overbrace{\text{MLP}_{\boldsymbol{\theta}}\left((1+\epsilon)\,\boldsymbol{h}_v^{(\ell)} + \underbrace{\sum_{w \in \mathcal{N}(v)}}_{\bigoplus} \underbrace{\boldsymbol{h}_w^{(\ell)}}_{\textsc{Message}_{\boldsymbol{\theta}}^{(\ell+1)}}\right)}^{\textsc{Update}_{\boldsymbol{\theta}}^{(\ell+1)}}, \tag{4.25}$$

where $\epsilon$ is a trainable scalar in order to distinguish neighbors from central nodes.

**Principal Neighborhood Aggregation (PNA)** [155] networks leverage mulitple aggregators combined with degree-scalers to better capture graph structural properties

$$\boldsymbol{h}_v^{(\ell+1)} = \overbrace{\boldsymbol{W}_2\left[\boldsymbol{h}_v^{(\ell)}, \bigoplus_{w \in \mathcal{N}(v)} \underbrace{\boldsymbol{W}_1\left[\boldsymbol{h}_v^{(\ell)}, \boldsymbol{h}_w^{(\ell)}\right]}_{\textsc{Message}_{\boldsymbol{\theta}}^{(\ell+1)}}\right]}^{\textsc{Update}_{\boldsymbol{\theta}}^{(\ell+1)}}, \tag{4.26}$$

where $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ denote trainable weight matrices, and

$$\bigoplus = \underbrace{\begin{bmatrix} 1 \\ s(\deg(v), 1) \\ s(\deg(v), -1) \end{bmatrix}}_{\text{Scalers}} \otimes \underbrace{\begin{bmatrix} \text{mean} \\ \text{min} \\ \text{max} \end{bmatrix}}_{\text{Aggregators}}, \tag{4.27}$$

with $\otimes$ being the tensor product and

$$s(d, \alpha) = \left(\frac{\log(d+1)}{\frac{1}{|\mathcal{V}|}\sum_{v \in \mathcal{V}} \log(\deg(v)+1)}\right)^{\alpha} \tag{4.28}$$

denoting degree-based scalers. Having introduced the basic concepts of message passing within GNNs, we now look more closely at their practical implementation (Section 4.3.3) and resource efficiency (Section 4.3.4).

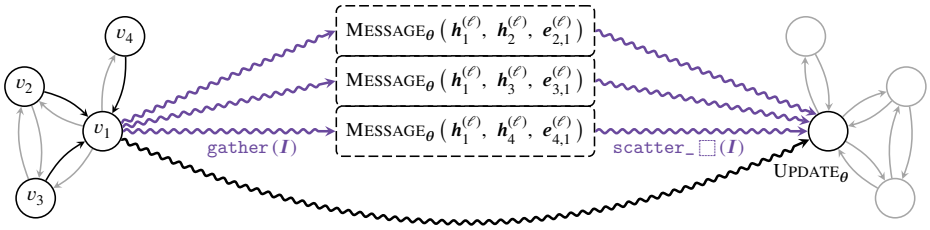



**Fig. 4.10:** Computation scheme of a GNN layer by leveraging gather and scatter methods based on edge indices $\boldsymbol{I}$, hence alternating between node parallel space and edge parallel space.

### 4.3.3 PyTorch Geometric: Implementing Graph Neural Networks

The practical implementation of graph neural networks is challenging, as high GPU throughput needs to be achieved on highly sparse and irregular data of varying size. Here, we introduce and discuss the *PyTorch Geometric* library [225], a library for deep learning on irregularly structured data, built upon PyTorch [555]. In addition to general graph data structures and processing methods, it contains a variety of recently published methods from the domains of relational learning and 3D data processing. PyTorch Geometric achieves high data throughput by leveraging sparse GPU acceleration, by providing dedicated CUDA kernels, and by introducing efficient mini-batch handling for input examples of different sizes. All implemented methods support both CPU and GPU computations and follow an immutable data flow paradigm that enables dynamic changes in graph structures through time. PyTorch Geometric is released under the MIT license and is available on GitHub.[8] It is thoroughly documented and provides accompanying tutorials and examples as a first starting point.[9]

In PyTorch Geometric, we represent a graph $\mathcal{G} = (\boldsymbol{X}, (\boldsymbol{I}, \boldsymbol{E}))$ by a node feature matrix $\boldsymbol{X} \in \mathbb{R}^{N \times F}$ of $N$ nodes holding $F$ features, and a sparse adjacency tuple $(\boldsymbol{I}, \boldsymbol{E})$ of $E$ edges, where $\boldsymbol{I} \in \mathbb{N}^{2 \times E}$ encodes edge indices in COOrdinate (COO) format and $\boldsymbol{E} \in \mathbb{R}^{E \times D}$ (optionally) holds $D$-dimensional edge features. All user-facing APIs, e.g., data-loading routines, multi-GPU support, data augmentation, and model instantiations are heavily inspired by PyTorch to keep them as familiar as possible.

In practice, the realization of Equation 4.20 can be achieved by gathering and scattering node features and a vectorized elementwise computation of MESSAGE and UPDATE functions, as visualized in Figure 4.10. Although working on irregularly structured input, this scheme can be heavily accelerated by the GPU. In contrast to implementations via sparse matrix multiplications, the usage of gather and scatter proves to be advantageous for low-degree graphs and non-coalesced input, and allows for the integration of central node and multi-dimensional edge information directly while aggregating.

**8** GitHub repository: https://github.com/rusty1s/pytorch_geometric.

**9** Documentation: https://pytorch-geometric.readthedocs.io.

We implement different reductions for the scattering of neighboring node features via dedicated CUDA kernels, although execution on other hardware is applicable as well. For more, see Chapter 6.

We provide the user with a general `MessagePassing` interface to allow for rapid and clean prototyping of new research ideas. In order to use the interface, users only need to define the methods $\textsc{Message}_{\boldsymbol{\theta}}$, i.e., `message`, and $\textsc{Update}_{\boldsymbol{\theta}}$, i.e., `update`, and choose an aggregation scheme $\bigoplus$. For implementing `message`, node features are automatically mapped to the respective source and target nodes. Almost all recently proposed neighborhood aggregation functions can be lifted to this interface, including (but not limited to) the methods already integrated in PyTorch Geometric. Overall, PyTorch Geometric currently bundles over 40 different GNN operators proposed in literature, as well as over 15 complete models.

**(Hierarchical) Pooling** PyTorch Geometric also supports graph-level outputs as opposed to node-level outputs by providing a variety of graph-level pooling functions [435, 687, 736]. To further extract hierarchical information and to allow deeper GNN models, various pooling approaches can be applied in a deterministic or data-dependent manner [118, 175, 205, 241, 588, 633, 697, 724].

**Mini-Batch Handling** Our framework supports batches of multiple graph instances (of potentially different size) by automatically creating a single (sparse) block-diagonal adjacency matrix and concatenating feature matrices in the node dimension. Therefore, neighborhood aggregation methods can be applied without any modifications, since no messages are exchanged between disconnected graphs. In addition, an automatically generated assignment vector ensures that node-level information is not aggregated across graphs as when executing global aggregation operators.

**Processing of Datasets** We provide a consistent data format and an easy-to-use interface for the creation and processing of datasets, both for large datasets and for datasets that can be kept in memory during training. In order to create new datasets, users just need to read/download their data and convert it to the PyTorch Geometric data format via the respective `process` method. In addition, datasets can be modified by the use of `transforms`, which take in separate graphs and transform them, say, for data augmentation, for enhancing node features with synthetic structural graph properties, in order to automatically generate graphs from point clouds or to sample point clouds from meshes.

**Empirical Evaluation** We evaluated the correctness of the implemented methods by performing a comprehensive comparative study in homogeneous evaluation scenarios, reaching state-of-the-art performance on several graph benchmark tasks. For example, experiments for the semi-supervised node classification performance of common GNN

**Tab. 4.1:** Performance (accuracy and standard deviation) of semi-supervised node classification experiments for fixed and random splits across 100 runs.

| Method | Cora | | CiteSeer | | PubMed | |
|---|---|---|---|---|---|---|
| | Fixed | Random | Fixed | Random | Fixed | Random |
| Cheby [164] | 81.4±0.7 | 77.8±2.2 | 70.2±1.0 | 67.7±1.7 | 78.4±0.4 | 75.8±2.2 |
| GCN [375] | 81.5±0.6 | 79.4±1.9 | 71.1±0.7 | 68.1±1.7 | 79.0±0.6 | 77.4±2.4 |
| GAT [683] | 83.1±0.4 | 81.0±1.4 | 70.8±0.5 | 69.2±1.9 | 79.0±0.3 | 77.5±2.3 |
| SGC [710] | 81.7±0.1 | 80.2±1.6 | 71.3±0.2 | 68.7±1.6 | 78.9±0.1 | 76.5±2.4 |
| ARMA [52] | 82.8±0.6 | 80.7±1.4 | **72.3**±1.1 | 68.9±1.6 | 78.8±0.3 | 77.7±2.6 |
| APPNP [378] | **83.3**±0.5 | **82.2**±1.5 | 71.8±0.5 | **70.0**±1.4 | **80.1**±0.2 | **79.4**±2.2 |

architectures are easily finished within 1–2 seconds per run, either using fixed or random training splits. Table 4.1 presents the results of state-of-the-art GNNs on several citation datasets [621, 718]. Notably, all experiments show a high reproducibility of the reported results.

### 4.3.4 Scalable and Expressive Graph Neural Networks

While the full-gradient in GNNs is straightforward to compute since we have access to *all* hidden node representations of *all* layers, this is not feasible in large-scale graphs due to memory limitations and slow convergence [455]. Therefore, given a loss function $\phi$, it is desirable to approximate its full-batch gradient stochastically

$$\nabla \mathcal{L} = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} \nabla \phi(\boldsymbol{h}_v^{(L)}, y_v) \approx \frac{1}{|\mathcal{B}|} \sum_{v \in \mathcal{B} \subseteq \mathcal{V}} \nabla \phi(\boldsymbol{h}_v^{(L)}, y_v), \tag{4.29}$$

which considers only a mini-batch $\mathcal{B} \subseteq \mathcal{V}$ of nodes for loss computation. However, this stochastic gradient is still expensive to compute due to the exponentially increasing dependencies of node representations over layers, a phenomenon known as *neighbor explosion* [292]. Specifically, the representation of a given node depends recursively on all its neighbor's representations, and the number of dependencies grows exponentially with respect to the number of layers.

Recent works try to alleviate this problem by proposing various different sampling techniques [455], which can be broadly categorized as node-wise, layer-wise, and subgraph sampling strategies. In general, these techniques can all be viewed as different variants of dropping edges [600]. *Node-wise sampling* [126, 292] recursively samples a fixed number $k$ of 1-hop neighbors, leading to an overall bounded $L$-hop neighborhood of $\mathcal{O}(k^L)$ for each node. In contrast to tracking down inter-layer connections, *layer-wise sampling* techniques independently sample nodes for each layer, leading to a constant sampled size in each layer [126, 323, 747]. Here, variance is further reduced via importance sampling or adaptive sampling techniques. In *subgraph sampling* [137, 731,

732], a full GNN is run on an entire subgraph $\mathcal{G}[\mathcal{B}]$ induced by a sampled batch of nodes $\mathcal{B} \subseteq \mathcal{V}$. Notably, most of these sampling approaches eliminate the neighbor explosion problem, but there are challenges to preserving the edges that present a meaningful topological structure.

Another line of work is based on the idea of decoupling propagations from predictions [234, 378, 710, 726]. Here, input node features are first enhanced by performing several rounds of propagation via, say, the normalized Laplacian matrix or the personalized matrix, before they are inputted into an Multilayer Perceptron (MLP) to perform the final prediction. While this scheme enjoys fast training and inference time, it cannot be applied to any GNN, especially because the propagation is non-trainable. Recently, Huang, He, Singh, Lim, and Benson [321] proposed a simple post-processing step to correct and smooth the predictions of a simple graph-agnostic model via label propagation. While this step is orthogonal to recent GNN advancements, it can only be applied in transductive learning scenarios.

It is well known that the most powerful GNNs adhere to the same representational power as the WL test [701] in distinguishing non-isomorphic structures [509, 714], i.e., $\boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}$ in case $c_v^{(L)} \neq c_w^{(L)}$, where $c_v^{(L)}$ denotes a node's coloring after $L$ rounds of color refinement. However, in order to leverage such expressiveness, a GNN needs to be able to reason about structural differences across neighborhoods directly *during* training. It has been shown that GNNs that scale by sub-sampling edges are not capable of doing so [224]:

**Proposition 10.** *Let $\boldsymbol{f}_{\boldsymbol{\theta}}^{(L)} \colon \mathcal{V} \to \mathbb{R}^d$ be a L-layered GNN as expressive as the WL test in distinguishing the L-hop neighborhood around each node $v \in \mathcal{V}$. Then there exists a graph $\boldsymbol{A} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$ for which $\boldsymbol{f}_{\boldsymbol{\theta}}^{(L)}$ operating on a sampled variant $\tilde{\boldsymbol{A}}$, $\tilde{a}_{v,w} = \begin{cases} \frac{|\mathcal{N}(v)|}{|\tilde{\mathcal{N}}(v)|}, & \text{if } w \in \tilde{\mathcal{N}}(v) \\ 0, & \text{otherwise} \end{cases}$, produces a non-equivalent coloring, i.e., $\tilde{\boldsymbol{h}}_v^{(L)} \neq \tilde{\boldsymbol{h}}_w^{(L)}$ while $c_v^{(L)} = c_w^{(L)}$ for nodes $v, w \in \mathcal{V}$.*

Therefore, a special interest lies in the question if there exist scalable GNN variants that are as expressive as their full-batch counterpart.

#### 4.3.4.1 Scaling Graph Neural Networks via Historical Embeddings

We now introduce the *GNNAutoScale (GAS)* framework [224], which scales graph neural networks by pruning entire sub-trees of the computation graph and filling the missing information by utilizing historical embeddings acquired in previous training iterations [126, 153], leading to constant GPU memory consumption with respect to input node size without dropping any data. Since GNNAutoScale accounts for all data, it provably is able to maintain the expressive power of the underlying graph neural network.

Let $\boldsymbol{h}_v^{(\ell)}$ denote the node embedding in layer $\ell$ of a node $v \in \mathcal{B}$ in a mini-batch $\mathcal{B} \subseteq \mathcal{V}$. For the general message scheme given in Equation 4.20, the execution of $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}$

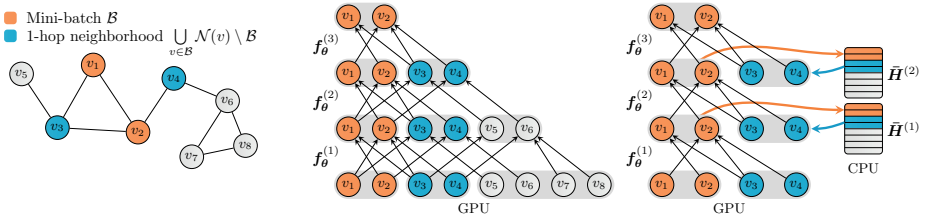



**Fig. 4.11:** Mini-batch processing of GNNs with historical embeddings. ■ denotes the nodes in the current mini-batch and ■ represents their direct 1-hop neighbors. For a given mini-batch (left), GPU memory and computation costs increase exponentially with GNN depth (middle). The usage of historical embeddings avoids this problem as it *prunes* entire sub-trees of the computation graph, which leads to constant GPU memory consumption with respect to input node size (right). Here, nodes in the current mini-batch *push* their updated embeddings to the history $\bar{\boldsymbol{H}}^{(\ell)}$, while their direct neighbors *pull* their most recent historical embeddings from $\bar{\boldsymbol{H}}^{(\ell)}$ for further processing.

can be formulated as:

$$\begin{aligned}
\boldsymbol{h}_v^{(\ell+1)} &= \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} : w \in \mathcal{N}(v) \right\}\!\!\right\}\right) \\
&= \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} : w \in \mathcal{N}(v) \cap \mathcal{B} \right\}\!\!\right\} \cup \left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B} \right\}\!\!\right\}\right) \\
&\approx \boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}\left(\boldsymbol{h}_v^{(\ell)}, \underbrace{\left\{\!\!\left\{ \boldsymbol{h}_w^{(\ell)} : w \in \mathcal{N}(v) \cap \mathcal{B} \right\}\!\!\right\}}_{\textbf{(1)}\text{ Local embeddings}} \cup \underbrace{\left\{\!\!\left\{ \bar{\boldsymbol{h}}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B} \right\}\!\!\right\}}_{\textbf{(2)}\text{ Historical embeddings}}\right)
\end{aligned}$$

Here, we separate the neighborhood information of the multiset into *two* parts: **(1)** the local information of neighbors $\mathcal{N}(v)$ that are part of the current mini-batch $\mathcal{B}$, and **(2)** the information of neighbors that are not included in the current mini-batch. We then approximate the embeddings of out-of-mini-batch nodes with their historical embeddings denoted as $\bar{\boldsymbol{h}}_w^{(\ell)}$. After each step of training, the newly computed embeddings $\boldsymbol{h}_v^{(\ell+1)}$ are pushed to the history and serve as historical embeddings $\bar{\boldsymbol{h}}_w^{(\ell+1)}$ in future iterations.

A high-level illustration of its computation flow is visualized in Figure 4.11. It can be seen that in the original data flow without historical embeddings the required GPU memory increases as the model gets deeper. After a few layers, embeddings for the entire input graph need to be stored, even if only a mini-batch of nodes is considered for loss computation. By contrast, historical embeddings eliminate this problem by approximating entire sub-trees of the computation graph. The required historical embeddings are pulled from an offline storage, instead of being re-computed in each iteration, which keeps the required information for each batch local. For a single batch $\mathcal{B} \subseteq \mathcal{V}$, the GPU memory footprint for one training step is given by $\mathcal{O}(|\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \cup \{v\}| \cdot L)$ and thus only scales linearly with the number of layers $L$. The vast majority of data (the histories) can be stored in RAM or hard drive storage rather than GPU memory.

In contrast to existing scaling solutions based on the sub-sampling edges, GAS provides the following advantages:

1. **GAS trains over all the data:** In GAS, a GNN will make use of all available graph information, i.e., *no* edges are dropped, which results in lower variance and more accurate estimations. Nonetheless, for a single epoch and layer, each edge will be only processed once, putting its time complexity $\mathcal{O}(|\mathcal{E}|)$ on par with its full-batch counterpart. Notably, more accurate estimations will further strengthen gradient estimation during backpropagation. Specifically, the model parameters will be updated based on the node embeddings of *all* neighbors since $\partial \boldsymbol{h}_v^{(\ell+1)} / \partial \boldsymbol{\theta}$ also depends on $\{\!\{ \bar{\boldsymbol{h}}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B} \}\!\}$.
2. **GAS enables constant inference time complexity:** The time complexity of model inference is reduced to a constant factor, since we can directly use the historical embeddings of the last layer to derive predictions for test nodes.
3. **GAS is simple to implement:** Our scheme does not need to maintain recursive layer-wise computation graphs, which makes its overall implementation straightforward and comparable to full-batch training. Only minor modifications are required to *pull* information from and *push* information to the histories.
4. **GAS provides theoretical guarantees:** In particular, if the model weights are kept fixed, $\bar{\boldsymbol{h}}_v^{(\ell)}$ eventually equals $\boldsymbol{h}_v^{(\ell)}$ after a fixed amount of iterations [126].

While sampling strategies loose expressive power due to the sub-sampling of edges, scalable GNNs based on historical embeddings leverage *all* edges during neighborhood aggregation. Therefore, a special interest lies in the question if history-based GNNs are as expressive as their full-batch counterpart. Here, a maximally powerful *and* scalable GNN needs to fulfill the following two requirements: **(1)** it needs to be as expressive as the WL test in distinguishing non-isomorphic structures, and **(2)** it needs to account for the approximation error $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\|$ induced by the usage of historical embeddings. Since it is known that there exists a wide range of maximally powerful GNNs [155, 509, 714], we can restrict our analysis to the latter question.

**Theorem 11.** *Let $\boldsymbol{f}_{\boldsymbol{\theta}}^{(L)}$ be an L-layered GNN. If the historical embeddings do not run too stale, i.e., $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \leq \epsilon$, then there exist $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ functions, $\ell \in \{1, \ldots, L\}$, such that there exists a map $\phi \colon \mathbb{R}^D \to \Sigma$ so that $\phi(\tilde{\boldsymbol{h}}_v^{(L)}) = c_v^{(L)}$ for all $v \in \mathcal{V}$.*

Informally, Theorem 11 (proof in Fey et al. [224]) indicates that scalable GNNs using historical embeddings are able to distinguish non-isomorphic structures (that are distinguishable by the WL test) directly during training, which is what makes reasoning about structural properties possible.

Nonetheless, to allow for high expressiveness, we need to tighten the upper bound of the approximation error induced by the usage of historical embeddings. As denoted before, the output embeddings of $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell+1)}$ are exact if $|\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \cup \{v\}| = |\mathcal{B}|$, i.e., all

neighbors of nodes in $\mathcal{B}$ are also part of $\mathcal{B}$. However, in practice, this can only be guaranteed for full-batch GNNs. Motivated by this observation, we aim to minimize the inter-connectivity between sampled mini-batches, i.e., $\min |\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \setminus \mathcal{B}|$, which minimizes history accesses and therefore reduces overall staleness in return.

We make use of graph partitioning methods such as METIS [175, 361] to achieve this goal. It aims to construct partitions over the nodes in a graph such that intra-links within clusters occur much more frequently than inter-links between different clusters. Intuitively, this results in a high chance that neighbors of a node are located in the same cluster. Notably, modern graph clustering methods are both fast and scalable with time complexities given by $\mathcal{O}(|\mathcal{E}|)$, and only need to be applied once, which leads to an insignificant computational overhead in the pre-processing stage. However, this approach leads to the acceleration of training, since the number of neighbors outside of $\mathcal{B}$ is heavily reduced, and pushing information to histories leads to contiguous memory transfers.

#### 4.3.4.2 Fast Historical Embeddings

Our approach accesses histories to account for any data outside the current mini-batch, which requires frequent data transfers to and from the GPU. Therefore, a special interest lies in the optimization of pulling from and pushing to the histories. We achieve that by making use of *non-blocking* device transfers. Specifically, we immediately start pulling historical embeddings for each layer asynchronously at the beginning of each optimization step, which ensures that GPUs do not run idle while waiting for memory transfers to complete. A separate worker thread gathers historical information into one of multiple pinned CPU memory buffers (denoted by PULL), from where it can be transfered to the GPU via the usage of CUDA streams without blocking any CPU or CUDA execution. Synchronization is done by synchronizing the respective CUDA stream before inputting the transferred data into the GNN layer. The same strategy is applied for pushing information to the history. Considering that the device transfer of $\bar{\boldsymbol{H}}^{(\ell-1)}$ is faster than the execution of $\boldsymbol{f}_{\boldsymbol{\theta}}^{(\ell)}$, this scheme does not lead to any runtime overhead when leveraging historical embeddings and can be twice as fast as its serial non-overlapping counterpart, (cf. Figure 4.12). We have implemented our non-blocking transfer scheme with custom C++/CUDA code to avoid Python's global interpreter lock.

#### 4.3.4.3 Experimental Evaluation

For training large-scale GNNs, GPU memory consumption directly dictates the scalability of the given approach. In Fey et al. [224], we confirmed that GNNs trained via GAS are able to learn expressive node representations, closely resemble the performance of their non-scaling counterparts, and reach state-of-the-art performance on large-scale graphs. Here, we show how GAS maintains a low GPU memory footprint while, in contrast to other scalability approaches, accounting for all information present in the data. We directly compare the memory usage of GCN+GAS training with the memory usage of

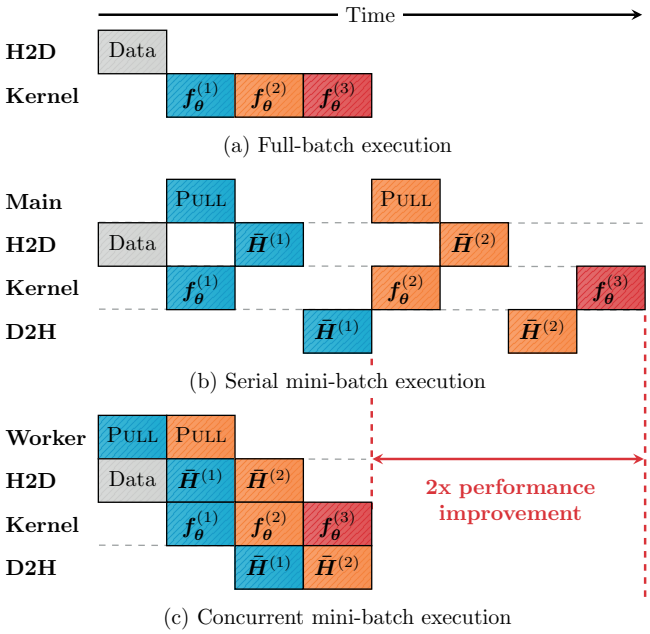



**Fig. 4.12:** Illustrative runtime performances of a serial and concurrent mini-batch execution compared with a full-batch GNN execution. In the full-batch approach (a), all necessary data is first transferred to the device via the Host2Device (H2D) engine, before GNN layers are executed in serial inside the kernel engine. As depicted in (b), a serial mini-batch execution suffers from an I/O bottleneck, in particular because each kernel engine has to wait for memory transfers to complete. The concurrent mini-batch execution (c) avoids this problem by leveraging an additional worker thread and overlapping data transfers, leading to a two times performance improvements compared with a serial execution, which is on par with the standard full-batch approach.

full-batch GCN [375], mini-batch GraphSAGE [292], and Cluster-GCN [137] training on three large-scale datasets [320, 731] (Table 4.2). Notably, GAS is easily able to fit the required data on the GPU, while the memory consumption only increases linearly with the number of layers. Although Cluster-GCN maintains an overall lower memory footprint than GAS, it will only utilize a fraction of the available information, i.e., about 23 % on average.

We now analyze how GAS enables large-scale training due to fast mini-batch execution. Specifically, we are interested in how our concurrent memory transfer scheme reduces the overhead induced by accessing historical embeddings from the offline storage. For this, we evaluate running times of a 4-layer GIN model on synthetic graph data, which allows fine-grained control over the ratio between inter- and intraconnected nodes (Figure 4.13). Here, a given mini-batch consists of exactly 4000 nodes, which are randomly intraconnected to 60 other nodes. We vary the number of inter-connections (connections to nodes outside of the batch) by adding out-of-batch nodes that are

**Tab. 4.2:** GPU memory consumption (in GB) and the amount of data used (%) across different GNN execution techniques. GAS consumes low memory while making use of all the data.

| | # nodes<br># edges<br>**Method** | 717K<br>7.9M<br>YELP | | 169K<br>1.2M<br>ogbn-arxiv | | 2.4M<br>61.9M<br>ogbn-products | |
|---|---|---|---|---|---|---|---|
| 3-layer | Full-batch | 9.44GB/ | 100 % | 2.11GB/ | 100 % | 31.53GB/ | 100 % |
| | GRAPHSAGE | 2.19GB/ | 14 % | 0.93GB/ | 33 % | 4.34GB/ | 5 % |
| | CLUSTER-GCN | 0.23GB/ | 13 % | 0.22GB/ | 40 % | 0.23GB/ | 16 % |
| | **GAS** | 0.79GB/ | 100 % | 0.34GB/ | 100 % | 0.59GB/ | 100 % |
| 4-layer | Full-batch | 12.24GB/ | 100 % | 2.77GB/ | 100 % | 41.10GB/ | 100 % |
| | GRAPHSAGE | 4.31GB/ | 19 % | 1.55GB/ | 37 % | 11.23GB/ | 8 % |
| | CLUSTER-GCN | 0.30GB/ | 13 % | 0.29GB/ | 40 % | 0.29GB/ | 16 % |
| | **GAS** | 1.07GB/ | 100 % | 0.46GB/ | 100 % | 0.82GB/ | 100 % |

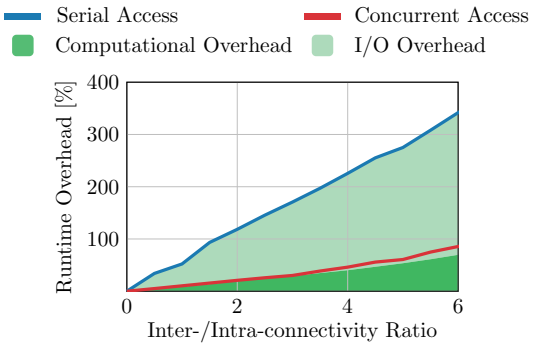



**Fig. 4.13:** Runtime overhead between serial and concurrent history access patterns in relation to the inter-/intraconnectivity ratio of mini-batches. The overall runtime overhead is further separated into computational overhead (overhead of aggregating additional messages) and I/O overhead (overhead of pulling from and pushing to histories). Our concurrent memory transfer scheme reduces historical-caused overhead by a wide margin.

randomly connected to 60 nodes inside the batch. Notably, the naive serial memory transfer increases runtimes up to 250 %, which indicates that frequent history accesses can cause major I/O bottlenecks. By contrast, our concurrent access pattern shows *no* I/O overhead *at all*, and the overhead in execution time is solely explained by the computational overhead of aggregating far more messages during message propagation. Considering the increased amount of additional data available, this overhead is marginal, in particular because most real-world datasets come with inter-/intra-connectivity ratios between 0.1 and 2.5 [224]. Further, the additional overhead of computing METIS partitions in the pre-processing stage is negligible. Computing the partitioning of a graph with 2M nodes takes only about 20–50 seconds (depending on the number of clusters).

### 4.3.5 Conclusion

We introduced graph neural networks for graph machine learning based on deep learning techniques. We demonstrated that graph neural networks follow a general message passing scheme, which is suitable for a wide range of operators and learning tasks. The generality of message passing is show-cased in the PyTorch Geometric library, a well-known deep learning library for implementing and working with graph-based neural networks. Furthermore, we discussed scalable approaches for applying graph neural networks to large-scale graphs. In particular, we showed that scalable approaches based on the sub-sampling of edges or non-trainable propagations weaken the expressive power of message passing. By contrast, our proposed framework, AutoScale, overcomes this restriction by utilizing historical node embeddings while being both fast and memory-efficient to train. While this scheme allows scalable graph machine learning on single or multiple GPUs on the same machine, additional considerations need to be taken into account when data is laid out in a distributed fashion (Section 8.2).

## 4.4 High-Quality Parallel Max-Cut Approximation Algorithms for Shared Memory

*Nico Bertram*
*Jonas Ellert*
*Johannes Fischer*

**Abstract:** We engineer parallel algorithms for approximating the maximum cut in a large directed graph. Our general approach is to first partition the graph into $p$ parts, where $p$ denotes the number of processing elements. The individual processors then independently compute an approximation to their local part of the graph using high-quality sequential approximation algorithms. In a final step, a single Max-Dicut instance of size $\mathcal{O}(p^2)$, capturing the interprocessor edges, is defined and solved exactly, using fast parallel Integer Program solvers or slow approximation algorithms that compute a good approximation. By partitioning the input graph into $p' > p$ parts, we get a smooth trade-off between cut quality and running time. We also show applications of our algorithm in parallel grammar-based text compression.

### 4.4.1 Introduction

Data that occurs in real-world applications can often be structured as *graphs* where data points are represented as nodes and relationships between different data points are captured by edges. Graphs occur in many applications, e.g., road networks, relationships in social networks [193], and bioinformatics [40].

The problem of finding a partitioning of a *directed graph* into two subsets $S$ and $T$ such that the sum of the edge-weights between the two subsets is maximized is one of the classical NP-complete problems. We denote this problem with Max-Dicut. It is closely related to its counterpart in *undirected graphs*, Max-Cut, which was shown to be NP-complete by Karp [359]. In fact, Max-Dicut seems much harder than Max-Cut since every instance of Max-Cut can be easily transformed into an instance of Max-Dicut. It can be shown that this transformation also defines a reduction from a Max-Cut instance to a Max-Dicut instance which shows the NP-hardness of Max-Dicut. That means that there is probably no polynomial time algorithm to compute an optimal solution for Max-Dicut.

One common approach in theory and practice is to solve such hard problems by using approximation algorithms. These algorithms allow for a multiplicative error $\alpha$ with $0 < \alpha < 1$ so that the computed cut is in the worst case by a factor of $\alpha$ worse than the optimal cut. We call this factor $\alpha$ the *performance guarantee* of an algorithm.

One simple randomized algorithm assigns each node with probability $\frac{1}{2}$ either to $S$ or $T$, which leads to a solution with an expected performance guarantee of $\frac{1}{4}$. This algorithm can be derandomized with the method of *conditional expectations* [590, 643]. Buchbinder et al. described a linear time algorithm with a performance guarantee of $\frac{1}{3}$ [90] that can also be randomized to achieve an expected performance guarantee of $\frac{1}{2}$.

Currently, the best-known performance guarantee of $0.874$ uses a formulation of Max-Dicut as an Integer Program that is then relaxed into a *Semidefinite Program* and achieves a performance guarantee of $0.79607$ [256]. This algorithm can be derandomized as well [459]. The performance guarantee was later further improved to $0.859$ [750]. The best-known performance guarantee of $0.874$ was achieved by further improving this approach [426]. In case that the *Unique Games Conjecture* [372] is true, the performance guarantee can be improved up to $0.878$.

There are also attempts to solve Max-Cut by using a machine learning approach. Gu and Yang described a deep neural network combined with learning strategies such as supervised learning and reinforcement learning [275]. Yao et al. [719] used Graph Neural Networks [291] to solve Max-Cut and compared it with the algorithm by Goemans for undirected graphs [256] and a local search procedure [59]. The results for both machine learning approaches are promising. However, they were only evaluated for small graphs; how to apply these approaches for directed graphs remains open.

Max-Cut can also be used in a graph-based semi-supervised learning approach. Wang et al. [695] showed that a bivariate cost function can be reduced to a constrained Max-Cut formulation. Since this formulation has a number of linear constraints on the nodes and the edge weights can be negative, most approximation algorithms cannot be used directly. The authors propose using a greedy gradient Max-Cut algorithm, instead.

To our knowledge, no algorithm exists that produces a Max-Dicut with high quality and performs well in shared memory. One approach to developing such algorithms is to use a graph partitioner [100] to partition a graph into $k$ parts of roughly equal size in terms of node balancing or edge balancing. On each of the $k$ parts we can run a sequential algorithm to compute a local solution with high quality that we have to merge in a final step. In this contribution, we first describe some elementary algorithms to compute a Max-Dicut in a graph. Then, we engineer a framework that computes a Max-Dicut with high quality in shared memory that uses the pattern described above. We also show how we can use a parallel Max-Dicut with high quality in grammar-based string compression to improve the compression ratio.

Parts of this work have already been published in [49].

### 4.4.2 Preliminaries

First, we define cuts in directed graphs. Then, we describe some important approximation algorithms for Max-Dicut.

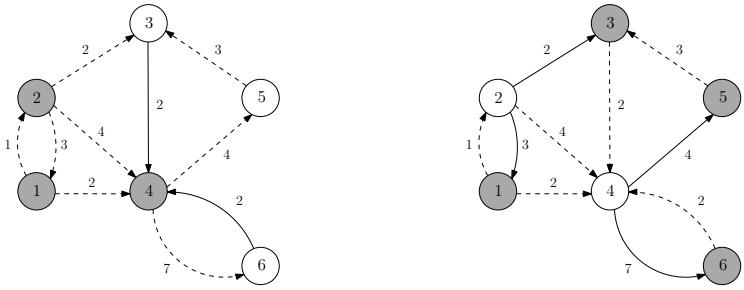



**Fig. 4.14:** A graph with two example cuts. The nodes that are in $S$ are colored in white and the nodes that are in $T$ are colored in gray. The edges that are not counted for the cut are dashed. The cut in (a) has the value 4. The cut in (b) has the value 16, which is the optimal value.

#### 4.4.2.1 Notations

Here, we define the necessary notations for graphs and cuts in directed graphs. A *directed* and *weighted graph* $G$ is a tuple $(V, E, w)$ where $V = \{1, \ldots, n\}$ is the set of *vertices*, $E \subseteq V^2$ is the set of *edges* with $|E| = m$ and $w\colon E \to \mathbb{R}_{>0}$ defines the non-negative *weights* of each edge.

A *cut* in a directed and weighted graph $G = (V, E, w)$ is a partitioning of $V$ into the subsets $S$ and $T$ so that the sum of the edge-weights for edges with origin in $S$ and target in $T$ is maximized. The value of a cut with respect to $S$ and $T$ is defined by $C(S, T) = \sum_{i \in S, j \in T} w(i, j)$. We omit $S$ and $T$ in case they are clear by the context. The maximum cut is then defined by $C_{max} = \max_{S,T} C(S, T)$. We call an edge $(u, v)$ with $u \in S$ and $v \in T$ a *cutting edge*. In Figure 4.14 we see examples of cuts in a directed graph.

#### 4.4.2.2 Algorithms

In the following, we will describe the algorithms we implemented in our framework. First, we describe a naive random approximation and its derandomization. Then, we describe the algorithm by Goemans and Williamson.

**Random Partitioning** One simple algorithm to produce a partitioning of a graph $G$ is to decide for each node $v$ independently with probability $\frac{1}{2}$ whether we assign $v$ to $S$ or $T$. This algorithm calculates a cut with an expected performance guarantee $\frac{1}{4}$ in linear time.

**Theorem 12.** *The described algorithm calculates a cut with an expected performance guarantee of $\frac{1}{4}$ in $\mathcal{O}(n)$ time.*

*Proof.* Since we assign each node in constant time either to $S$ or $T$, the running time is $\mathcal{O}(n)$. Now, we have to show the performance guarantee. Let $G = (V, E, w)$ be a directed graph. Let $W = \sum_{i,j \in V} w_{ij}$. First, we observe that

$$C_{max} \leq W. \tag{4.30}$$

Next, let $e = (u, v) \in E$ be an arbitrary edge. This edge is a cutting edge only if $u \in S$ and $v \in T$. Since we assigned each node randomly with probability $\frac{1}{2}$ to either side of the partition, the probability that $e$ is a cutting edge is exactly $\frac{1}{4}$. So in expectation our calculated cut has the value $E[C] = \frac{1}{4}W$. By Equation 4.30 it follows that $\frac{1}{4}W \geq \frac{1}{4}C_{max}$ and lastly $\frac{E[C]}{C_{max}} \geq \frac{1}{4}$. □

**Derandomization** The random algorithm described above can be derandomized so it deterministically produces a cut with a performance guarantee of $\frac{1}{4}$. This can be done with the method of *conditional expectations* [590, 643]. Suppose we already placed nodes $1, \ldots, i-1$ into either $S$ or $T$. We denote as $E[C \mid 1, \ldots, i-1]$ the expected value of the cut when we place nodes $i, \ldots, n$ at random into each partition. Now, we want to assign node $i$ to either $S$ or $T$. Let us assume that the value $E[C \mid 1, \ldots, i-1] \geq \frac{1}{4}C_{max}$ (when $i = 0$ this assumption is trivially satisfied). Intuitively, we try to put $i$ into the partition that results in the best expected outcome. Since $E[C \mid 1, \ldots, i-1] \geq \frac{1}{4}C_{max}$, at least one of both decisions has to result in an expected value of the cut of $\frac{1}{4}C_{max}$. We can calculate the expected increase for each decision for node $i$ as follows:

$$A = \sum_{\substack{j<i \\ j \in T}} w_{ij} + \frac{1}{2} \sum_{j>i} w_{ij} \tag{4.31}$$

$$B = \sum_{\substack{j<i \\ j \in S}} w_{ji} + \frac{1}{2} \sum_{j>i} w_{ji} \tag{4.32}$$

Equation 4.31 describes the expected increase of the cut when we place $i$ into $S$ and Equation 4.32 describes the expected increase of the cut when we place $i$ into $T$. In both sums the first term refers to the already calculated partitioning of $1, \ldots, i-1$. The second term refers to the expected value when we assign $i+1, \ldots, n$ at random to either $S$ or $T$. When we choose the maximum of $A$ and $B$, we have $E[C \mid 1, \ldots, i] \geq \frac{1}{4}C_{max}$ and, when we assigned all nodes, $E[C] \geq \frac{1}{4}C_{max}$.

**Theorem 13.** *The described algorithm calculates a cut with a performance guarantee of* $\frac{1}{4}$ *in* $\mathcal{O}(m)$ *time.*

**Goemans and Williamson Algorithm** In the following, we describe the Goemans and Williamson algorithm [256] that in its original description had a performance guarantee of 0.79607 but was further improved up to 0.874 [426]. To illustrate the

algorithm, we describe the algorithm for *undirected* graphs that has a performance guarantee of 0.878 and show at the end how to modify the algorithm for *directed* graphs.

First, we need some additional notation. By Prob[$A$] we denote the probability that event $A$ happens. The function sgn($x$) denotes the *sign* function that is defined as

$$\text{sgn}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0. \end{cases}$$

The general idea of the algorithm is to solve a relaxed formulation of Max-Cut as an integer quadratic program (IQP) and then assign each node to either $S$ or $T$ depending on the computed solution. The interesting part about this algorithm is that we relaxed our formulation to a *semidefinite program*. This method, first introduced by Goemans and Williamson, leads to improved performance guarantees for other problems as well such as Max-2-SAT [256].

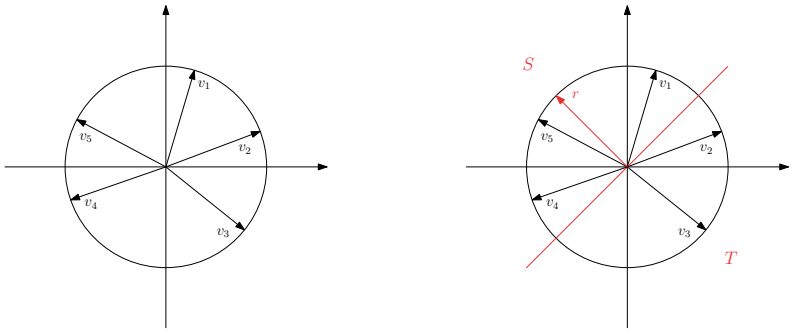
Let $G = (V, E, w)$ be a directed graph. We start with the following formulation of Max-Cut as IQP:

$$\begin{aligned} \text{maximize} \quad & \frac{1}{2} \sum_{i<j} w_{ij}(1 - x_i x_j) \\ \text{subject to} \quad & x_i \in \{-1, 1\} \quad \forall i \in \{1, \ldots, n\} \end{aligned} \tag{4.33}$$

Each node $i$ is represented by a variable $x_i$ that has value $-1$ when $i$ is placed into $S$ and 1 when $i$ is placed into $T$. When we look at the term $(1 - x_i x_j)$, we can see that it evaluates to 2 if nodes $i$ and $j$ are in different partitions and 0 otherwise. Hence, each cutting edge is counted twice, which is why we normalize the calculated value by $\frac{1}{2}$. Solving Equation 4.33 is still NP-hard but now we examine the properties of this formulation when we relax its variables to vectors of dimension $n$. Let $S_n$ be the n-dimensional unit sphere. In Equation 4.34 we see the relaxed formulation.

$$\begin{aligned} \text{maximize} \quad & \frac{1}{2} \sum_{i<j} w_{ij}(1 - v_i v_j) \\ \text{subject to} \quad & |v_i| \in S_n \quad \forall i \in \{1, \ldots, n\} \end{aligned} \tag{4.34}$$

Note, that the optimal solution of Equation 4.34 is an upper bound of the optimal solution of Equation 4.33 because every solution of Equation 4.33 is also a solution of Equation 4.34 (we can transform $x_i$ to a vector $v_i$ by setting the first element to $x_i$ and every other value to 0). In Figure 4.15a we see five vectors that for simplicity's sake are embedded in the unit circle. At first glance, it is hard to see how we should divide the vectors into the partitions $S$ and $T$. Intuitively, $v_1$ and $v_2$ are relatively similar to each other so it should be more likely that they are placed in the same partition as $v_2$ and $v_4$. This similarity can be expressed by the scalar product of vectors $u$ and $v$, which is defined by $u \cdot v = |u| \cdot |v| \cdot \cos(\alpha) = \cos(\alpha)$ where $\alpha$ is the angle between $u$ and $v$.




(a) (b)

**Fig. 4.15:** An example for how to assign nodes to either $S$ or $T$. In (a) we see a solution to Equation 4.34. In (b) we see a random vector $r$ that defines a partitioning of the nodes. Since $v_1$, $v_4$ and $v_5$ lie on the same side of $r$, we set $S = \{1, 4, 5\}$ and $T = \{2, 3\}$. For simplicity, all vectors are embedded in the 2-dimensional unit circle.

To compute an optimal partitioning is still hard but we can compute a partitioning that results in a good solution with high probability. We choose a random vector $r$ uniformly distributed over $S_n$. With $r$ we can define a partitioning by putting all vectors $v_i$ that lie on the same side of $r$ into $S$ i.e., $S = \{i \mid v_i \cdot r \geq 0\}$ and all other vectors into $T$ i.e. $T = \{i \mid v_i \cdot r < 0\}$. This partitioning is visualized in Figure 4.15b. Intuitively, it is more likely that with this partitioning similar vectors are placed in the same partition, which should result in a good solution. This intuition can be formalized in the following lemma.

**Lemma 14.** *Let $v_i$ and $v_j$ be vectors that are optimal solutions of Equation 4.34 and $r \in S_n$ be a random vector drawn uniformly from the n-dimensional unit sphere. Then*

$$Prob[sgn(v_i \cdot r) \neq sgn(v_j \cdot r)] = \frac{1}{\pi} \arccos(v_i \cdot v_j).$$

By Lemma 14 our calculated Cut has an expected value of

$$E[C] = \frac{1}{\pi} \sum_{i<j} w_{ij} \arccos(v_i \cdot v_j).$$

From the fact that $\frac{\arccos(v_i \cdot v_j)}{\pi} \geq \alpha \frac{1}{2}(1 - v_i v_j)$ with $\alpha > 0.878$ we can derive the following theorem:

**Theorem 15.** *Let $v_i$ and $v_j$ be vectors that are optimal solutions of Equation 4.34. Then*

$$E[C] \geq \alpha \frac{1}{2} \sum_{i<j} w_{ij}(1 - v_i v_j) \geq \alpha C_{max}.$$

Now, we still have to show how to get an optimal solution for Equation 4.34. We can transform this formulation into a *semidefinite program (SDP)*. First, we have to define *positive semidefinite matrices*.

**Definition 16.** *Let $M \in \mathbb{R}^{n\times n}$ be a symmetric matrix. $M$ is called* positive semidefinite *if all of its eigenvalues are non-negative. If $M$ is positive semidefinite, we denote this by $M \succeq 0$.*

The following important property holds.

**Lemma 17** ([257, 410])**.** *Let $M \in \mathbb{R}^{n\times n}$ be a positive semidefinite matrix. There exists a matrix $B \in \mathbb{R}^{n\times n}$ such that $M = B^T B$. We can calculate $B$ in $\mathcal{O}(n^3)$ time with a Cholesky Decomposition.*

A *semidefinite program* has the following form where $A_1, \ldots, A_m, B_1, \ldots B_m \in \mathbb{R}^{n\times n}$ are constant matrices and $b_1, \ldots, b_m \in \mathbb{R}$. The variable matrices $X_i \in \mathbb{R}^{n\times n}$ have the constraint that they should be positive semidefinite matrices. To multiply matrices, we use the *Frobenius inner product* defined by: $A \cdot B = \sum_{i,j} A_{ij}B_{ij}$.

$$\begin{aligned} \text{maximize} \quad & A_1 \cdot X_1 + \ldots + A_m \cdot X_m \\ \text{subject to} \quad & X_i \succeq 0 \quad \forall i \in \{1, \ldots, m\} \\ & B_i \cdot X_i = b_i \quad \forall i \in \{1, \ldots, m\} \end{aligned} \tag{4.35}$$

Optimal solutions for a SDP can be computed in $\mathcal{O}(n^c \log(\frac{1}{\epsilon}))$ time for some $c > 0$ by using *interior point methods* [343] where $\epsilon > 0$ is an error parameter.

To convert Equation 4.34 into an SDP in the form of Equation 4.35, we set $y_{ij} = v_i \cdot v_j$. We observe that $y_{ij}$ describes the cosine of the angle between vectors $v_i$ and $v_j$ and $y_{ij} = y_{ji}$. So the matrix

$$Y = \begin{pmatrix} y_{11} & y_{11} & \ldots & y_{1n} \\ y_{21} & y_{22} & \ldots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \ldots & y_{nn} \end{pmatrix}$$

is a symmetric matrix that describes the cosine of the angles between every vector. The element $y_{ii}$ describes the length of vector $v_i$. Since every vector lies on the unit sphere $S_n$, we add the condition that $y_{ii} = 1$ for every $i \in \{1, \ldots, n\}$. So the formulation of Max-Cut as SDP is as follows.

$$\begin{aligned} \text{maximize} \quad & \frac{1}{2} W \cdot ((1)_{n\times n} - Y) \\ \text{subject to} \quad & Y \succeq 0 \\ & y_{ii} = 1 \quad \forall i \in \{1, \ldots, n\} \end{aligned} \tag{4.36}$$

Here $W$ is defined as the weight matrix of $G$ and $(1)_{n\times n}$ is the matrix that contains 1 in each component. By Lemma 17 we can obtain an optimal set of vectors $v_i$ with a Cholesky Decomposition in time $\mathcal{O}(n^3)$.

Now, we show how to modify our formulations to get an approximation algorithm that results in a cut with a performance guarantee of 0.79607 for Max-Dicut. We start again with the formulation of Max-Dicut as IQP:

$$\begin{aligned} \text{maximize} \quad & \frac{1}{4}\sum_{i<j} w_{ij}(1 + x_0x_i - x_0x_j - x_ix_j) \\ \text{subject to} \quad & x_i \in \{-1, 1\} \quad \forall i \in \{1, \dots, n\} \end{aligned} \tag{4.37}$$

Here, we introduce an additional variable $x_0$ that marks which side lies in $S$. More precisely, if $x_0$ is equal to $-1$ all other nodes $i$ with $x_i = -1$ should be assigned to $S$ and to $T$ otherwise. The term $(1 + x_0x_i - x_0x_j - x_ix_j)$ evaluates to 4 if $x_i$ is assigned to $S$ and 0 otherwise. That is why we have to normalize with the value $\frac{1}{4}$. Similar to the undirected Max-Cut, we relax the variables in Equation 4.37 so that they are $n$ dimensional vectors.

$$\begin{aligned} \text{maximize} \quad & \frac{1}{4}\sum_{i<j} w_{ij}(1 + v_0v_i - v_0v_j - v_iv_j) \\ \text{subject to} \quad & |v_i| \in S_n \quad \forall i \in \{1, \dots, n\} \end{aligned} \tag{4.38}$$

For its formulation as a SDP we use the matrices $X \in \mathbb{R}^{n+1\times n+1}$ and $Y, Z \in \mathbb{R}^{n\times n}$ that are defined as follows:

$$X = \begin{pmatrix} y_{00} & y_{01} & \dots & y_{0n} \\ y_{10} & y_{11} & \dots & y_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n0} & y_{n1} & \dots & y_{nn} \end{pmatrix} Y = \begin{pmatrix} y_{11} & y_{11} & \dots & y_{1n} \\ y_{21} & y_{22} & \dots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \dots & y_{nn} \end{pmatrix} Z = \begin{pmatrix} y_{10} & y_{10} & \dots & y_{10} \\ y_{20} & y_{20} & \dots & y_{20} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n0} & y_{n0} & \dots & y_{n0} \end{pmatrix}$$

Then Max-Dicut can be formulated as SDP as follows:

$$\begin{aligned} \text{maximize} \quad & \frac{1}{4}W \bullet ((1)_{n\times n} + Z - Z^T - Y) \\ \text{subject to} \quad & X \succeq 0 \\ & y_{ii} = 1 \quad \forall i \in \{0, \dots, n\} \end{aligned} \tag{4.39}$$

When we compute a solution for Equation 4.39, we choose a random vector vector $r$ uniformly distributed over $S_n$. Since $v_0$ marks the side for the partition $S$, we assign all nodes $i$ where $v_i$ and $v_0$ lie on the same side to $S$. More precisely, we set $S = \{i \mid \text{sgn}(v_i \cdot r) = \text{sgn}(v_0 \cdot r)\}$ and $T = \{i \mid \text{sgn}(v_i \cdot r) \neq \text{sgn}(v_0 \cdot r)\}$.

By analyzing the algorithm similarly to the undirected Max-Cut, we find that our algorithm has a performance guarantee of 0.79607. The following theorem summarizes our results.

**Theorem 18.** *The described algorithm calculates a cut with a performance guarantee of* 0.79607 *in polynomial time.*

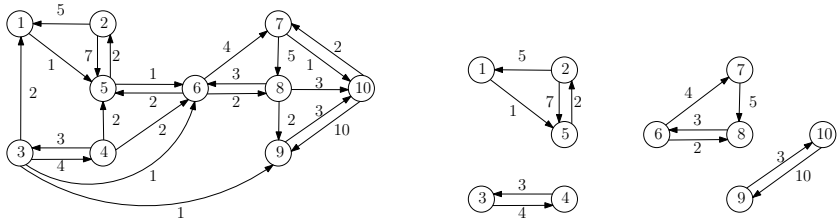



**Fig. 4.16:** The input graph in (a) is partitioned into 4 subgraphs, which can be seen in (b), so that the sum of the edge-weights between the subgraphs is minimized.

### 4.4.3 Framework

In this section we introduce a parallel framework that computes a Max-Dicut with high quality in shared memory. First, we give an overview of the whole framework before we introduce each step individually.

Our approach is to partition an input graph $G$ into $k$ subgraphs $G_i$ of roughly equal size so that the dependency between the subgraphs is minimized i.e. the edge-weights between the subgraphs are minimized. Then on each computed subgraph, we run in parallel a sequential Max-Dicut algorithm to compute several local cuts. In a final step, we have to merge the locally computed cuts. We achieve this by defining a new graph on which Max-Dicut is solved where each node represents a partition computed by the local Max-Dicut algorithms.

#### 4.4.3.1 Graph Partitioning

The first step of our framework is to partition the input graph $G = (V, E, w)$ into $k$ subgraphs so that we can run a Max-Dicut algorithm on each subgraph independently. Our goal is to include as much edge information as possible into each subgraph to improve the quality of the computed Max-Dicut. We achieve this by maximizing the sum of the edge-weights in each subgraph or, vice versa, by minimizing the sum of the edge-weights between the subgraphs. That is to say, we want to minimize $\sum_{i,j\in\{1,\dots,k\}} E_{ij}$ where $E_{ij}$ is the sum of the edge-weights between subgraph $G_i$ and $G_j$. To compute a well balanced graph partitioning in shared memory, there already exist several approaches. In our framework we use the graph partitioner KaHIP [13], which partitions $G$ into the subgraphs $G_i = (V_i, E_i, w), i \in \{1, \dots, k\}$ so that each subgraph has roughly equal size, i.e. we allow for a multiplicative error $\epsilon$ so that $|V_i| \leq (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil$. We also use a partitioning algorithm that naively divides either the nodes or edges into $k$ chunks of equal size. We could also use Metis, as in Section 4.3. However, KaHIP outperforms Metis and is better suited for our application. In Figure 4.16 we see an exemplary input graph that is partitioned into 4 subgraphs.

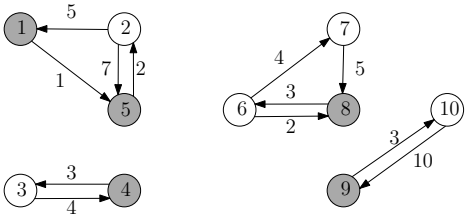



**Fig. 4.17:** On each computed subgraph in Fig. 4.16 we compute a Max-Dicut. The nodes that are in $S_i$ are colored in white and the nodes that are in $T_i$ are colored in gray.

#### 4.4.3.2 Compute Local Solutions

After partitioning the input graph into multiple subgraphs, we run in parallel a sequential Max-Dicut algorithm on each subgraph to compute a local cut for each subgraph. We can compute cuts with higher quality when we use algorithms with better performance guarantees. Since these algorithms are also slower, we have to consider which algorithm achieves the best trade-off between the quality of the cut and the runtime of the framework.

We have implemented the randomized algorithm with an expected performance guarantee of $\frac{1}{4}$ and its derandomized variant, the algorithm with a performance guarantee of $\frac{1}{3}$ that was introduced by Buchbinder [90], and the algorithm with an expected performance guarantee of 0.79607 by Goemans and Williamson [256]. As an example, we show in Figure 4.17 the optimal Max-Dicut for all subgraphs that were computed in Figure 4.16.

#### 4.4.3.3 Merging

In a final step, we have to merge the computed local cuts into a global cut. A naive approach is to define $S = \bigcup_i S_i$ and $T = \bigcup_i T_i$ as the trivial cut. The problem with this approach is that it does not consider the edges between the subgraphs. It might be possible that it is more advantageous to swap the subsets $S_i$ and $T_i$ in the global graph or even to put $S_i$ and $T_i$ into the same partition. To consider each possible combination of merging the cuts, we reduce the problem of merging the local solutions to another Max-Dicut instance. We build a complete graph $H$ with $2k$ nodes in which each node represents a locally computed partition $S_i$ or $T_i$. Let $X$ and $Y$ be two nodes of $H$. We add an edge $(X, Y)$ to $H$ with weight $\sum_{i \in X, j \in Y} w(i, j)$. Then, we can run a Max-Dicut algorithm on $H$. Since the graph has only $2k$ nodes, we can use an expensive algorithm to compute an exact solution. In our framework we implemented a simple brute-force algorithm and an algorithm that solves the formulation of Max-Dicut as an Integer Program. We can also use the approximation algorithm with a performance guarantee of 0.79607 by Goemans and Williamson. In Figure 4.18 we see how the local cuts that were computed in Figure 4.17 are merged into a new Max-Dicut instance. Then, we compute the global solution on this instance.

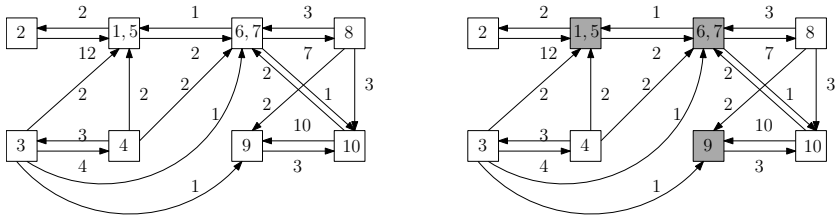



**Fig. 4.18:** We compute a new Max-Dicut instance with 8 nodes (Fig. 4.18a) by merging the nodes that are in the same partition after Fig. 4.17. We run an exact Max-Dicut algorithm on this instance and compute the global Max-Dicut in Fig. 4.18b. The nodes that are in $S$ are colored in white and the nodes that are in $T$ are colored in gray.

**Tab. 4.3:** A summary of our used input graphs.

| Graph | $\|V\|$ | $\|E\|$ |
|---|---|---|
| `recomp_dna1GB_5` | 28 245 | 1 439 986 |
| `road-luxembourg-osm` | 114 600 | 239 332 |
| `rt-retweet-crawl` | 1 112 703 | 4 557 704 |

### 4.4.4 Evaluation

In this section we evaluate our framework. We conducted our experiments on the LiDO3-Cluster of the Technical University of Dortmund[10] on a node with an Intel Xeon CPU E5-2640 processor (20 cores, 2.4 GHz, L1 32K, L2 256K, L3 256M) with 64 GB of RAM. The code was written in `C++` and compiled using GCC 8.4 using OpenMP for parallelization.

We evaluate our framework on the input graphs that are summarized in Table 4.3. The graph `recomp_dna1GB_5` was generated from a *recompression* tool [342] by using the 1 GiB prefix of the text `dna.txt` from the Pizza & Chili text corpus.[11] Here, the nodes represent the characters from the alphabet and we have an edge $(a, b)$ if the pair $ab$ appears in the text. The weight of the edge represents the number of occurrences of the pair $ab$. The graphs `road-luxembourg-osm` and `rt-retweet-crawl` were taken from *Network Repository* [604]. The graph `road-luxembourg-osm` is a road network of Luxembourg and the graph `rt-retweet-crawl` is a Retweet graph of Twitter where each node represents a Twitter user and we have an edge between two users when one user retweets a tweet from the other user.

**10** https://www.lido.tu-dortmund.de/cms/de/LiDO3/index.html, accessed June 9, 2022.
**11** http://pizzachili.dcc.uchile.cl/, accessed June 9, 2022.

#### 4.4.4.1 Experiments

For our experiments, we evaluate each part of our framework separately. First, we compare our partitioning algorithms KaHIP, NodeSlice, and EdgeSlice. Then, we compare our local MAX-DICUT algorithms Derandomization, Buchbinder, and Goemans. For the Goemans algorithm, we compare two variants: one solves the SDP exactly, which we call *Goemans*, the other solves the SDP with a small error where we set $\epsilon = 0.01$, which we will call *Goemans($\epsilon = 0.01$)*. For our merging algorithms, we compare the Buchbinder algorithm, the two Goemans variants described above, and an exact algorithm that solves an *integer linear program* (ILP). When we evaluate the algorithms of one part of our framework, all other parts are fixed, i.e. for the partitioning we use KaHIP, as the local MAX-DICUT algorithm we use Buchbinder, and as merging algorithm we use Goemans ($\epsilon = 0.01$). We conducted all of our experiments five times and took the average of each result for the computed cut as well as the runtime for each step in the framework. We divide the graphs into as many as 2048 parts. Up until 16 parts, we use the same amount of cores as the number of parts. For more than 16 parts, we constantly use 16 cores.

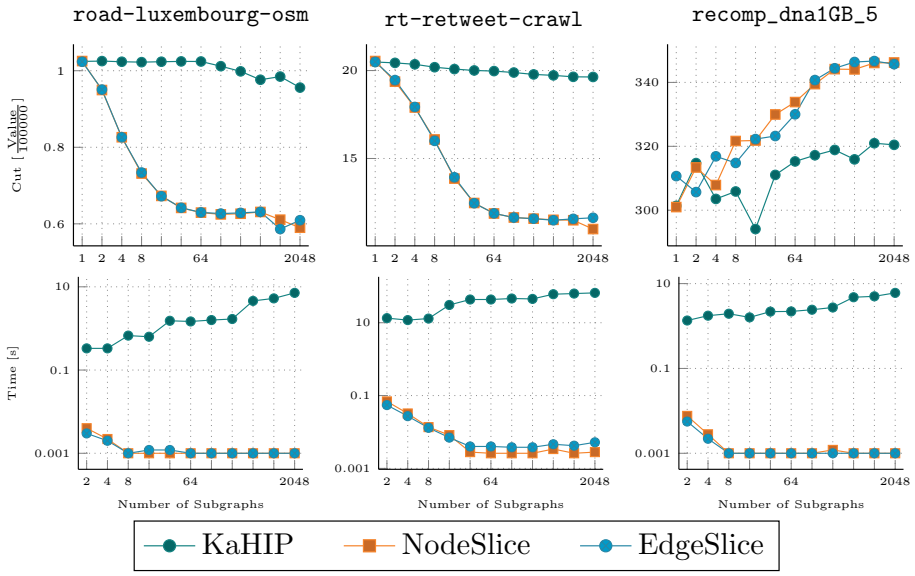



**Fig. 4.19:** The computed cut and the runtime for our partitioning algorithms while the other steps of the framework are fixed algorithms. Missing data points indicate either that the runtime of the whole framework exceeded the time limit or that the memory exceeded the RAM.

In Figure 4.19, we can see our results for our partitioning algorithms. We see that using KaHIP as a partitioner results in an almost constant cut quality for each number of subgraphs for the graph `road-luxembourg-osm` and `rt-retweet-crawl` while the

cut quality when using the naive partitioning algorithms NodeSlice and Edgeslice gets worse when we partition the graph into more subgraphs. However, on the graph `recomp_dna1GB_5` the cut quality when using NodeSlice and EdgeSlice scales better than KaHIP. The runtime of KaHIP is on all inputs significantly slower than NodeSlice and EdgeSlice and does not scale as well as the naive algorithms.

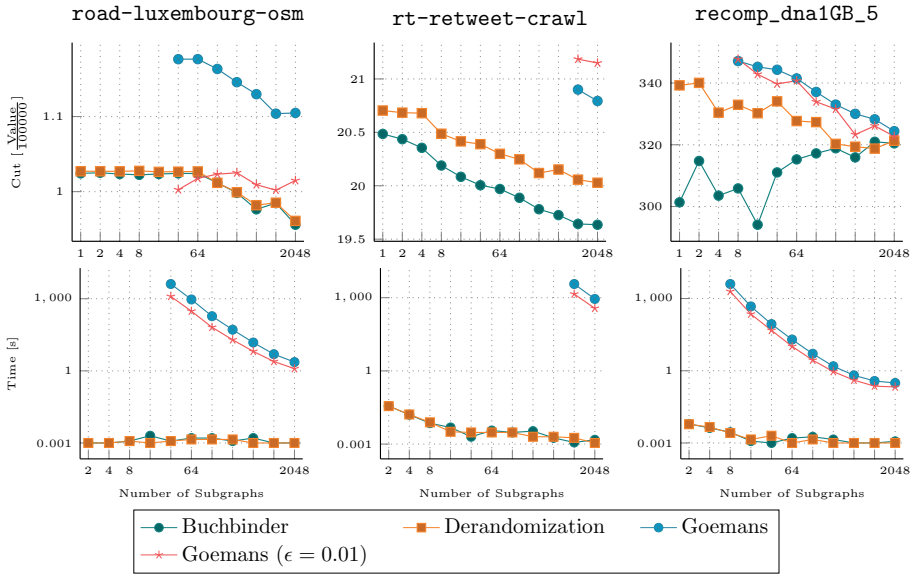



**Fig. 4.20:** The computed cut and the runtime for our local Max-Dicut algorithms while the other steps of the framework are fixed algorithms. Missing data points indicate either that the running time of the whole framework exceeded the time limit or the memory exceeded the RAM.

In Figure 4.20, we can see our results for the local Max-Dicut algorithms. We can see that by using the variants of the Goemans algorithm our framework produces the overall best cut quality. However, these algorithms only compute a solution when we partition the graph into a large number of subgraphs or when we have smaller subgraphs. For larger subgraphs, the Goemans algorithm either takes too long or consumes too much memory. The runtime of the Goemans algorithms is significantly larger than the linear-time algorithms but it gets faster the smaller the subgraphs get.

In Figure 4.21, we can see our results for the merging algorithms. Note that, since our framework uses some random variables, the computed quality may vary between different configurations. Overall, the merging has only a small effect on the cut quality. As one would expect, the exact algorithm that solves an ILP gives the best solution most of the times, closely followed by the exact Goemans algorithm. Using Goemans ($\epsilon = 0.01$), our framework produces a good cut quality most of the times,as well. However, for 128 parts or more the cut quality gets significantly worse on `road-luxembourg-osm`.

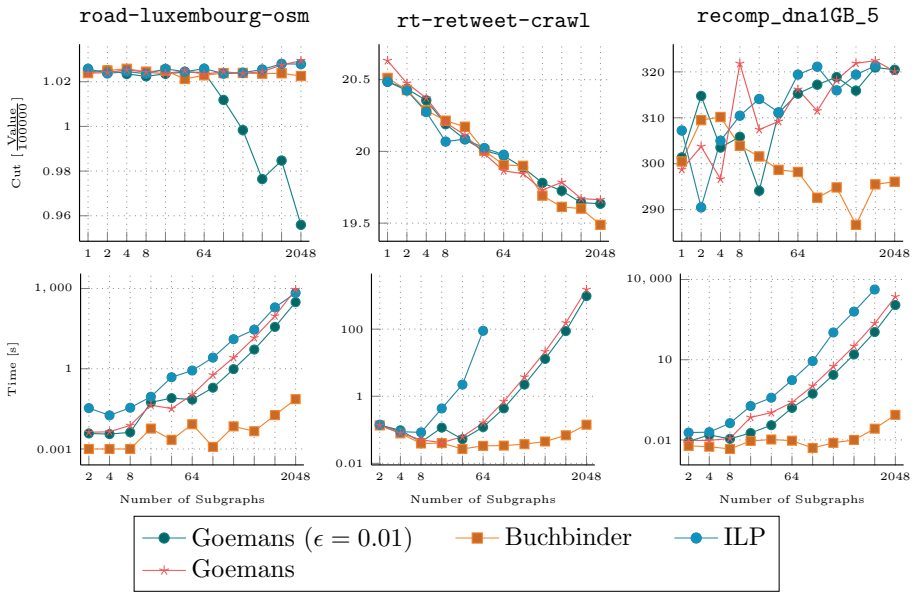



**Fig. 4.21:** The computed cut and the runtime for our merging algorithms; the other steps of the framework are fixed algorithms. Missing data points indicate either that the runtime of the whole framework exceeded the time limit or that the memory exceeded the RAM.

The runtime of ILP is overall the slowest and on `rt-retweet-crawl` becomes too slow for 128 and more parts. The Goemans variants are faster than IPL but are still slower than Buchbinder. We can see that ILP and the Goemans variants slow the larger the Graph $H$ becomes.

### 4.4.5 Application in String Compression

MAX-DICUT is used in building a succinct data structure over strings to answer *Longest Common Extension (LCE) queries* efficiently. An LCE query over a string $S$ asks for two positions $i$ and $j$ for the longest common prefix of the suffixes starting at position $i$ and $j$.

To answer such queries efficiently, one can use the *recompression* technique that was described by Jez [342]. With this technique, a string $S$ is compressed into a context-free grammar that generates exactly $S$. Then, we build an LCE data structure [330] on top of the grammar. The memory usage is $\mathcal{O}(z \log(\frac{n}{z}))$ and the query time is $\mathcal{O}(\log(n))$ where $z$ is the size of the Lempel-Ziv 77 factorization [746] and $n$ is the size of $S$.

During the compression of $S$ into a context-free grammar, we try to find pairs $ab$ and build a rule $X \to ab$ so that as many pairs are covered by a rule. To do that, we build a directed graph $G$ in which each node represents a character of $S$ and we insert

**Tab. 4.4:** The results of different recompression algorithms. We compare the running time in seconds and the compression ratio (compressed text length divided by original text length) for 8 cores on different texts taken from the Pizza & Chili text corpus. In all experiments we use 200 MiB prefix for each text. We mark in bold text the best result on the respective text. Additionally, we provide the size of and the alphabet size $\sigma$ for each text.

| | | `max-dicut_recomp` | | `lp_recomp` | |
|---|---|---|---|---|---|
| **Text** | $\sigma$ | **Time [s]** | **ratio** | **Time [s]** | **ratio** |
| `cere` | 5 | 290.75 | **4.9 %** | **21** | 4.91 % |
| `dna` | 16 | 286.2 | 42.42 % | **26** | **42.29 %** |
| `einstein.en` | 139 | 211.75 | **0.17 %** | **26** | **0.17 %** |
| `english` | 239 | 277.4 | **41.23 %** | **36** | 42.72 % |
| `para` | 5 | 256.6 | **6.81 %** | **22** | 6.82 % |
| `sources` | 230 | 288.2 | **37.79 %** | **33** | 39.91 % |

an edge from $a$ to $b$ if the pair $ab$ appears in $S$. Then, a cut in $G$ represents a partition of the characters into two subsets $\Sigma_1$ and $\Sigma_2$ so that we can compress as many pairs $ab$ with $a \in \Sigma_1$ and $b \in \Sigma_2$ as possible without overlapping pairs. Accordingly, there is a direct correlation between the quality of the computed cut and the compression ratio of $S$.

We integrated our framework for computing a Max-Dicut in a tool that computes the compression with *recompression* in shared memory. We compare the algorithm `max-dicut_recomp` that uses our Max-Dicut framework with the algorithm `lp_recomp` that computes first a naive Max-Cut $(S, T)$ and then compares $C(S, T)$ and $C(T, S)$ in the directed graph and takes the largest value. Additionally, `lp_recomp` tries to take the solution that produces lesser production rules.

Again, we conducted our experiments on the LiDO3-Cluster of the Technical University Dortmund on a node with an Intel Xeon CPU E5-2640 processor (20 cores, 2.4 GHz, L1 32K, L2 256K, L3 256M) with 64 GB of RAM. We compared the compression ratio and runtime for 8 cores of our algorithms on a number of texts taken from the Pizza & Chili text corpus.[12] We repeated our experiments five times and took the average as the final result.

Table 4.4 shows our results. We can see that `max-dicut_recomp` achieves on almost all texts a similar or better compression ratio than `lp_recomp`. On `english` and `sources` the compression ratio increases by 1–2 %. However, to increase the compression ratio for `max-dicut_recomp` we need around 10 times more runtime than `lp_recomp` on all texts on 8.

**12** http://pizzachili.dcc.uchile.cl/, accessed June 9, 2022.

### 4.4.6 Conclusion

In this section we described a framework that calculates a high quality MAX-DICUT in shared-memory that is also easily extendable. We implemented our framework and evaluated it in shared-memory on real-world graphs. The experiments showed that our graph partitioning algorithm KaHIP does not scale well in shared-memory so we plan to use other partitioning algorithms in the future. The best configuration of our framework is to partition our graph into small graphs and use Goemans as our local MAX-DICUT algorithm.

We also integrated our framework into a software that calculates a grammar-based compression. By using our framework, we achieve in most cases better compression rates. However, our new algorithm is much slower than other compression algorithms.

## 4.5 Millions of Formulas

*Lukas Pfahler*

**Abstract:** Amid the increase in the number of research publications, the search for relevant papers has become tedious. In particular, searches across disciplines or schools of thinking are not supported. This is mainly due to the retrieval in terms of keyword queries, as technical terms differ in different sciences and at different times. Relevant articles might better be identified by their mathematical problem descriptions. Just looking at the equations in a paper already gives a hint to whether the paper is relevant. Hence, we propose a new approach for the retrieval of mathematical expressions based on machine learning. We design an unsupervised representation learning task that combines embedding learning, contrastive learning, and self-supervised learning. We want our learned representation to allow the automatic identification of related, relevant mathematical expressions. Using graph convolutional neural networks we embed mathematical expressions in low-dimensional vector spaces that allow efficient nearest-neighbor queries. To train our models, we collect a huge dataset with over 29 million mathematical expressions from over 900 000 publications on arXiv.org. The math is converted into an XML format, which we view as graph data. In this data, we are able to automatically identify equalities and inequalities that we can use for training and testing of our models. Furthermore, our empirical evaluations involve a dataset of manually annotated search queries show the benefits of using embedding models for mathematical retrieval. This contribution is based on a conference paper [563] and more details can be found in [562].

### 4.5.1 Introduction

Machine learning has contributed to many a search engine success story. Unfortunately, the search is most often based on words or text. Technical terms in different disciplines, however, may have different meanings or the same meaning may be referred to by different terms. For instance, various usages of Bayes' law occur in different scientific fields and can be found under different names. For instance in astrophysics, it is known as *information field theory* [200]. Without a knowledge of physics or the use of the name *Bayes*, the law is easily recognized by the formula $P(d|s) = P(d, s)/P(s)$ in any paper. Another example is a 1925 paper by Ising in the physics journal under the title *Ferromagnetismus*. Today, the Ising model is also popular in machine learning, but it is referred to first as the *Hopfield network* and later as the *Boltzmann machine*. This illustrates the aspect of time: words for particular topics change over time. The language of Ising's paper is German; the paper introducing Jensen's inequality in 1906

is written in French. Again, the inequality $f((a+b)/2) \leq f(a)/2 + f(b)/2$ can be easily understood, in both cases. We conclude that the most compact and comprehensive way to transport the main ideas of scientific manuscripts in disciplines like computer science or physics are the equations used. Thus it should also be the way we formulate our search queries when searching for scientific manuscripts. In order to judge the relevance of mathematical expressions for a search query, a system has to generalize between different notations and match the parts of equations, that describe the same concepts, even if they appear in a different form. A human reader resorts to domain knowledge acquired over years of training in his field to judge relevance. We wonder how machine learning models with access to vast amounts of mathematical content can help to automatize this process.

In this work, we propose using graph neural networks to learn a representation of mathematical expressions that captures semantic relatedness. To this end, we design two unsupervised learning tasks, one classic embedding learning task based on contextual similarity and one self-supervised learning task inspired by masked-language models. We curate a dataset of over 28.9 million equations from over 900 000 papers on arXiv.org and represent the equations as graphs with one-hot encoded features. Then we train our models on this large collection of equations. We compile an evaluation dataset with annotated search queries from several different disciplines and showcase the usefulness of our approach for deploying a search engine for mathematical expressions.

### 4.5.2 Math Search and KDD

Mining and indexing mathematical expressions in document collections is a challenging task, mostly tackled in the information retrieval community [277, 745]. We outline how the problem of math search is treated with the tools from Knowledge Discovery in Data and data mining and present related work on the machine learning methods we chose for our approach.

**Representation** The first question we have to consider is how to represent mathematical expressions. Approaches can be divided into two categories: those for visual representation and those for semantic representation. The former category is focused on the layout of an expression. The most prominent choices are LaTex, a Turing-complete language used in the publications on arXiv.org, and `Presentation MathML`[13], an XML dialect for displaying math on the web that we chose in this work. The latter category includes Content MathML and OpenMath, two similar XML dialects that focus on semantics rather than layout, and domain-specific languages for symbolic math solvers

13 https://www.w3.org/TR/MathML3/.

like Mathematica that also allow the manipulation and transformation of formulas. To the best of our knowledge, no large, public collection of semantic math expressions exists and, unfortunately, converting math from a display-representation, where data is available in large quantities, to a semantic representation, which seems more appropriate for searching, is a non-trivial task. Available solutions either use rules and heuristics, e.g. the converter `ml2om` that translates LaTeX to OpenMath[661], or also apply machine learning [693]. We chose to apply machine learning methods directly on the `Presentation MathML` representation. The bottom line of the representation question is that math is expressed in trees, either XML or other parse trees. Our previous work [564] may be the notable exception to this: we chose to represent equations as fixed-size bitmaps. While one could argue that this is an unsuitable choice, the multitude of machine learning or computer-vision approaches that successfully transform images of typeset [170] or hand-written [15, 460] math back to tree-based representations suggests that bitmap representations preserve all required information of tree-based approaches.

**Similarity Measure** The second question is how we compute similarity between formulas. Zanibbi et al. distinguish text-based, tree-based, and spectral approaches [729]. Text-based approaches transform tree-structured math into a sequence by pre-order traversal, say, and then estimate the similarity using methods for sequences such as cosine similarities of bags-of-words or the length of the largest common substring. Tree-based approaches focus on matching trees or subtrees. Typically computing similarities using sub-structures, either sub-sequences or sub-trees, involves solving dynamic-programming problems. Spectral approaches work on paths or partial subtrees in the trees. An example is the work by Zhong and Zanibbi [745], which indexes root-leaf paths of operator trees. From matches of the root-leaf paths, they compute the largest common subexpression to score the similarity of two equations. To convert math from LaTeX to the semantic representation of operator trees, the authors use ca. 100 grammar rules created by domain experts.

A new trend is to use machine learning to learn a similarity measure. A machine learning model maps an equation to a dense, low-dimensional vector. The similarity between these so-called embeddings can be computed via their inner product, which enables fast indexing using a variety of index structures, including faiss and annoy, designed for efficiently handling millions of these dense, low-dimensional vectors. Mansouri et al. [464] propose that equations be embedded using fastText, a method originally designed for computing word embeddings, while in our previous work [564] we compute embeddings with a similar embedding learning task and convolutional neural networks (see Section 4.2).

**Graph Convolutional Neural Networks** We have proposed an embedding model based on Graph Neural Networks (GNN) [563] introduced in this contribution. They are an appealing model choice for this task, as like classic Convolutional Neural Networks

(CNNs) for image processing, they compute feature maps based on local neighborhoods and thus can work on relations between symbols in formulas. While in CNNs we have features associated with each pixel in the pixel grid and neighborhoods are defined by this grid, in GNNs we have features associated with each node of the graph and neighborhoods are defined by the edges in the graph. We define graph structures $x = (X, E)$ as a tuple of node-features $X$ and edges $E$. Let $|x|$ denote the number of nodes in $x$. We assume that $X \in \mathbb{R}^{|x| \times d}$ where $X_i$ are the features of the $i$-th node. A GNN maps an input graph to an output with transformed feature vectors in a $d'$-dimensional output space but with identical edge structure. We use the graph network to compute a vector-valued embedding for mathematical expressions by an average-pooling operation that aggregates all node-embeddings of a graph into a single graph-embedding.

Additionally we investigate the use of transformer architectures [681], more specifically of Bidirectional Encoder Representations (BER)T models [173], for the task of embedding mathematical expressions into vector spaces. Transformers can be viewed as GNNs on a fully connected graph where each layer aggregates neighborhoods using self-attention [681].

**Self-Supervised Learning** We further draw inspiration from a recently proposed class of representation learning tasks called self-supervised learning. Self-supervised learning tasks are unsupervised learning tasks, where parts of the inputs are used to construct proxy tasks. The representations learned in these proxy-tasks can then be used in downstream tasks. For instance, we can rotate images and train a model to predict the rotation angle, as proposed by Gidaris et al. [251]. Using massive amounts of unlabeled data readily available, we can fit models that solve a task like this.

We are particularly interested in , where parts of the input are hidden from a model and the model's task is to predict the hidden parts. This was made popular by the BERT model for pretraining natural language representations [173] and has since then been adopted to other inputs such as pretraining for image classification with convolutional neural networks [669]. We construct a masking task for mathematical expressions and use graph convolutional neural networks to predict the masked parts.

### 4.5.3 The Data

We outline how we gather data from arxiv.org and transform it to graph structured data for our graph convolutional neural network.

#### 4.5.3.1 Dataset

We are working on data obtained from arxiv.org, a service where scientists can upload their manuscripts or pre-prints without reviewing process. We have downloaded all the

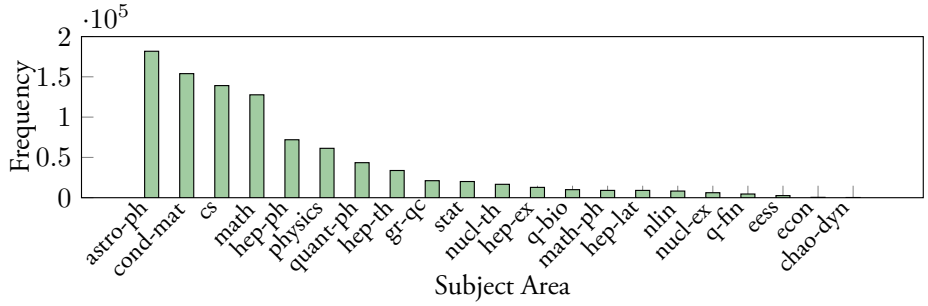



**Fig. 4.22:** Number of papers per subject area in our sample.

LaTeX sources of publications up to April 2019 from the official bulk data repositories.[14] This way we have obtained 934,287 papers. As we can see in Figure 4.22, the large majority of these papers are from disciplines where mathematical expressions are an important part of the publications. The most prominent subject areas are astrophysics, condensed-matter physics, high energy physics, computer science, and mathematics.

From all publications, we extract mathematical expressions by using regular expressions for the most common math-environments such as 'equation', 'align', etc. We do not use inline math snippets but focus on expressions that stand on their own, as they tend to describe more important concepts. Furthermore we extract user-defined commands and macros. Using the library `Katex`[15] we compile the raw LaTeX-equations to the XML-based `MathML` format. Out of all papers downloaded, 760 041 papers contain at least one equation that we were able to convert to `MathML`. In total we have a dataset of 28 973 591 `MathML` equations. Furthermore we have used regular expressions to find arXiv-ids in the bibliographies of the paper to build a citation graph. In total, 540 892 papers have an outgoing edge, with a total number of edges of 4 553 297. Since we only detect those references that use an arXiv-id in, say, an texttturl, our citation graph is only a subgraph of the true citation graph.

To ensure reproducibility we provide the scripts used for processing the public arXiv data dump, extracting the mathematical expressions and converting them to `MathML` as well as collecting meta-data and citations at https://github.com/Whadup/arxiv_library[16].

**14** https://arxiv.org/help/bulk_data_s3.

**15** http://katex.org.

**16** You can find the datasets used in this study at http://github.com/Whadup/arxiv-learning. We also share our citation graph, which might be interesting in other applications.

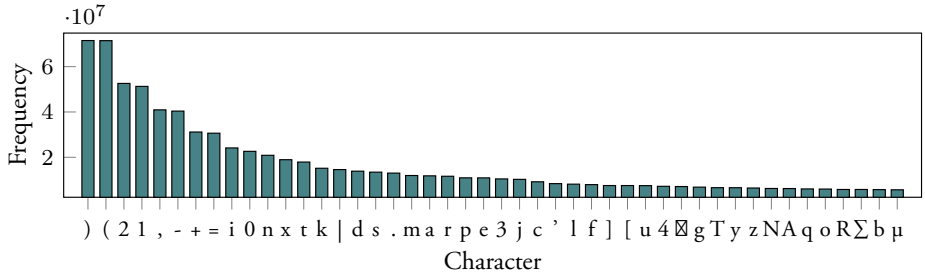



**Fig. 4.23:** The 50 most frequent characters in math environments.

#### 4.5.3.2 Data Representation

In order to feed the MathML to a graph convolutional neural network, we have to convert it to a graph with vectorial node features. The MathML standard defines around 30 different XML-tags such as <mi> for math identifiers or <mo> for math operators. Some of these tags use attributes, to change font or spacing, say. Leaf nodes contain text such numbers, parenthesis, or letters (Greek, Latin, etc...). We view the XML structure as a tree and use its nodes and edges and derive features based on tags, attributes, and text. For each node we use one-hot encoded feature vectors of dimensionality 256. First, we represent each node as a single token, where the token is derived by concatenating tag, attributes and text and use the 256 most frequent tokens that capture the majority of tokens in the data. Attribute values often contain numbers, e.g., for changing the font-size. We round these numbers to one decimal place to reduce the number of possible values. In addition to the one-hot encoded features, we store the position of the node among its sibling nodes.

Then, for the use with transformer models, we compute a sequential representation of our tree-structured data by a pre-order traversal of the tree.

### 4.5.4 Learning to Find Related Equations

In this section we will introduce the graph convolutional neural network used for computing embeddings and present two unsupervised learning tasks used for training the network. indexsubsubsectionModel for Embedding Formulas

**Graph Neural Network** We define a graph convolutional neural network for the task of embedding mathematical expressions into a low-dimensional vector space. The raw MathML is converted to graphs with vectorial features as described in Section 4.5.3.2. We propose using a special first layer that combines the one-hot encoded information at a node with the decimal position attribute. Following Vaswani et al. [681], we encode the position of the $i$-th node $p_i \in \mathbb{N}$ using positional embeddings. We use fixed sinusoid embeddings [681] denoted by $E(p_j)$, but in order to still allow the model to control the

influence of the positional embeddings, we introduce a learnable scaling coefficient $\alpha$ initialized to 1.

$$x_i^{(1)} = \text{ReLU}\left(\sum_{j \in \mathcal{N}(i) \cup i} W^{(1)} x_j + \alpha E(p_j) + b^{(1)}\right)$$

The first layer is followed by 3 fully-connected graph convolution layers of width 512, where the $l$-th layer is defined by

$$x_i^{(l)} = \text{ReLU}\left(\sum_{j \in \mathcal{N}(i) \cup i} W^{(l)} x_j^{(l-1)} + b^{(l)}\right)$$

which linearly transforms all nodes using a weight matrix $W^{(l)}$, adds a bias term $b^{(l)}$, aggregates by computing the sum over all neighbors $\mathcal{N}(i)$ and applies the ReLU activation component-wise. All graph convolution layers output feature maps with 512 dimensions. In our tree-structured data we assume all edges are bi-directional; hence the set of neighbors consists of the parent node and all child nodes. We apply batch-normalization before the first and third graph convolution layer. For the remainder of this paper, let $\phi(x) \in \mathbb{R}^{|x| \times 512}$ denote the output of the last graph convolution layer given the input $x$. To obtain a single embedding for an input graph, we compute the mean of all node features. This mean is transformed in another linear layer to reduce the dimensionality to 64. For the remainder of this paper, let $\bar{\phi}(x) \in \mathbb{R}^{64}$ denote this embedding of $x$.

When scoring similarities between embeddings with margin losses, we need to control the norm of the embeddings, otherwise the notion of adherence to a margin becomes meaningless. Ding et al. [179] and others have proposed normalizing all embeddings to unit length. We propose a softer normalization inspired by batch normalization[335] that also allows us to obtain embeddings with norms smaller than 1. For every training batch of graphs, we compute the mean of the norm as well as its standard deviation. Then we inversely scale each embedding by the mean plus the standard deviation. This way, most embeddings have a norm smaller than 1. We keep a running average of the means and standard deviations. At inference time, we use these running averages for scaling.

**Transformer** The original transformer model — as proposed by Vaswani et al. [681] — is slightly modified in BERT [173], which only uses encoder layers. In our work we use the same transformer model architecture as BERT — including the same encoder layers, activation functions, optimization algorithms, and learning rate schedules. The transformer architecture introduces the multi-head-attention layers as the key mechanism for learning the relations between each pair of tokens in the input sequence. This is applicable on mathematical formulas too, because understanding the relations between the symbols of a mathematical formula is crucial for understanding the meaning of the formula. We also extend the vocabulary by the special classification, separation,

masking, and unknown token—as did [173]—in order to predict masked tokens and thereby integrate in the model the ability to correct mathematical expressions.

We explore three differently-sized variants of the BERT architectures for embedding mathematical expressions. While BERT has a hidden-size-dependent number of attention heads, we keep them constant. We set the number of different multi-head-attention heads $D$ to 4. By doing so the hidden size $H$ has the largest impact on the performance of the multihead attention. As for the intermediate projection size, we kept this always bigger than the hidden size so that we can have a linear projection on a higher space. The resulting models are summarized in Table 4.5.

**Tab. 4.5:** Math-BERT model configurations.

| Model | L | H | I | D | Params | GFLOP |
|---|---|---|---|---|---|---|
| SMALL | 4 | 128 | 768 | 4 | 1.2 m | 0.7 |
| BASE | 8 | 256 | 768 | 4 | 6.0 m | 3.6 |
| LARGE | 12 | 512 | 768 | 4 | 25.0 m | 15 |

#### 4.5.4.1 Representation Learning Tasks

We propose that our embeddings are trained using two self-supervised learning tasks simultaneously by adding their respective losses.

**Contextual Similarity** For learning relations between equations, we rely on the established contextual similarity task that was first made popular by word embeddings [493] and has hence been used in many representation learning approaches, including our approach [564] for learning similarities between equations. The main idea is that objects that frequently appear in shared contexts are related. We define the context of mathematical expressions as the paper containing the equation and conjecture that two equations are related if they appear in the same paper, as originally proposed in [564]. We extend this approach and further define two equations as related if one paper references the other using a citation graph. This way we hope to connect equations that describe the same context but use different notation. In addition, we discriminate between sampling expressions from the same paper and from the same section. We hope that within sections, equations are more related to each other. For obtaining positive examples of related equations, we

1. sample a paper uniformly at random and select an expression from this paper uniformly at random;
2. randomly select whether we sample from the same section, same paper or along a citation;
3. sample a positive example using that method; when we cannot find a positive example using that method, we jump back to (1).

For learning similarities we also require negative examples. To obtain these, we sample a paper uniformly at random and select an expression from this paper uniformly at random. The random process that generates these weak labels for similarity learning introduces a lot of noise, as many equations we claim to be related are in fact unrelated and some of the pairs we say are unrelated are related. We leave the investigation of more advanced sampling schemes to future work.

Using the sampled equations $x$ with positive $x^+$ and negative partners $x^-$, we apply similarity learning. We have to choose a suitable loss function and investigate two different losses: Triplet and Histogram. The triplet loss [38] that we have previously used [564], contrasts the similarity between a positive pair of examples and a negative pair of examples and demands that the similar pair has a higher similarity by a user-defined margin $\Delta$, usually set to 1.

$$\ell_t(x, x^+, x^-) = \max(0, \Delta - \langle \bar{\phi}(x), \bar{\phi}(x^+) \rangle + \langle \bar{\phi}(x), \bar{\phi}(x^-) \rangle) \tag{4.40}$$

We have proposed using the histogram loss as first published by Ustinova and Lempitsky[676]. It does not work on a triplet of equations, but on a mini-batch of size $m$ positive pairs $X^+$ and a batch of negative pairs $X^-$ with respect to anchor examples $X$. We collect all similarities between positive pairs in a vector $s^+ = (\langle \bar{\phi}(x_i), \bar{\phi}(x_i^+) \rangle)_{i=1,\ldots,m}$ and of all negative pairs in $s^-$. We divide the interval $[-1, 1]$ into $R-1$ equally-sized bins with boundaries $-1 = t_1, t_2, \ldots, t_R = 1$ and width $\Delta = 2/(R-1)$ and build histograms for the positive similarities and the negative similarities. Now we demand that the positive histogram leans more toward the +1 similarity than the negative histogram. We formalize this intuition as

$$\ell_h(s^+, s^-) = \frac{1}{m^2} \sum_{r=1}^{R} \sum_{r'=1}^{r} \left( \sum_{i=1}^{m} \delta_r[s_i^-] \right) \left( \sum_{i=1}^{m} \delta_{r'}[s_i^+] \right) \tag{4.41}$$

where instead of hard assignments, we use the triangular kernel

$$\delta_r[s] = \begin{cases} (s - t_{r-1})/\Delta \text{ if } s \in [t_{r-1}, t_r] \\ (t_{r-1} - s)/\Delta \text{ if } s \in [t_r, t_{r+1}] \\ 0 \text{ otherwise} \end{cases}$$

to put similarities into bins. This way we obtain a differentiable loss function. We hope that histogram loss is more robust with regard to the massive noise in our labels as each positive example is contrasted with all negative examples.

**Masking Task** We propose extending the contextual similarity task by another task and optimizing the sum of both tasks for training our embedding models. The main idea of our second task is, that the symbols in mathematical expressions do not appear independent from each other, but have strong dependencies. Thus if we hide a fraction

of the symbols in an equation, we should be able to approximately reconstruct the hidden symbols from the remaining symbols. This task is reminiscent of masked language modeling tasks made popular by BERT [173] for natural language processing. In order to successfully solve this task, a model has to learn about the frequencies of symbols and their dependencies from the data, as is illustrated in Figure 4.24.

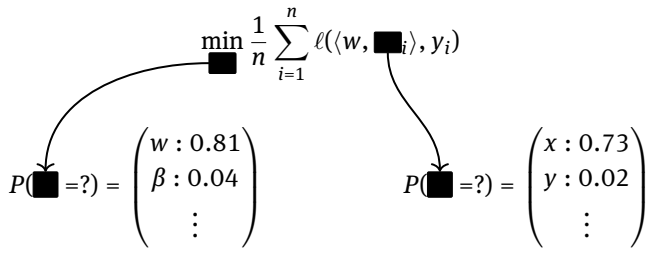



**Fig. 4.24:** Example of the masking task with fictional values.

More formally, we proceed as follows. For each input graph $x$ with features $X$, we randomly set the feature vector of 15 % of the nodes to all zero obtaining the graph $x_{\blacksquare}$. Then we compute $\phi(x_{\blacksquare}) \in \mathbb{R}^{|x| \times 512}$. Now for each masked node, we solve a classification task: given $\phi_i(x_{\blacksquare})$, predict the right token, i.e. the combination of XML-tag, XML-attributes, and character. This classification task is solved using a single linear layer of dimensionality 256 with softmax-activation and cross-entropy-loss.

$$\ell_i = \ell(\text{softmax}(W)\phi_i(x_{\blacksquare}) + b, X_i)$$

The loss is only evaluated for the masked tokens and we compute the mean over all masked tokens to obtain a loss value for $x_{\blacksquare}$.

Adding this task to the contextual similarity task has the additional advantage that we now learn a representation that not only captures context information, but also preserves information about the raw input symbols.

#### 4.5.4.2 Data Augmentation

Data augmentation eases the generalization of machine learning models and is particularly popular for image classification tasks where we can augment images by randomly rotating, scaling, padding, etc. For mathematical expressions, we propose the following random data augmentation. Since we know that a renaming of symbols in equations

rarely changes the semantic, we propose randomly permuting the character features of all nodes that correspond to a math identifier, encoded in <mi> tags according to the MathML standard. For each equation we process, we sample a number of flips from a Poisson distribution with an expected value of 32. Then starting with the identity permutation that does not change the order of our 192 features, we construct a permutation with the desired number of flips by incrementally exchanging two random characters.

### 4.5.5 Experimental Results

In this section we perform an experimental evaluation of our embedding model. In particular, we focus on the use-case of a search engine for mathematical expressions. We begin by investigating the effects of the individual components of our model on a small, closed subset of the data. Then we investigate the effectiveness of our method on all 29.9 million equations.

#### 4.5.5.1 Analysis on the Machine Learning Subset

We begin our analysis only on arXiv publications where the primary subject classification is machine learning (cs.LG). This is a natural choice, as we have some expertise to judge the quality of our results, a task which we are in no way equipped for across all subject fields.

Of these 9936 publications, we sample two subsets, train and test sizes of 7949 and 1987, respectively, and a total number of equations of 237 335 and 54 767, respectively. We use the train-set for building our embedding models and use the test-set to investigate generalization properties.

For training, we sample 1 million triplets $(x, x^+, x^-)$. Of these triples, 45.9 % have a positive pair from the same section, 42.2 % from the same paper, and 13.9 % along an edge in the citation graph. We sample 100k triplets for testing with similarly distributed positive examples.

We perform an ablation study on our proposed embedding model and compare it with prior work. This section investigates the influence of our design choices. We decided (a) to use the histogram loss instead of the triplet loss, and (b) to add a masking task, (c) to data augmentation.

We measure the ranking score, i.e. the fraction of all triples in the training data where same-class pairs of equations have higher similarities than across-class pairs. As we see in Table 4.6, our evaluations indicate that all of our design choices contribute favorably to the overall performance on hold-out data, as deactivating any component decreases the score. We note that the biggest gain is achieved by switching from triplet-loss to histogram-loss. We believe that this is due to the massive noise in our labels.

We also compare with our previous model [564] and see that we beat it by a small margin. However, this comparison is not entirely fair, as their model was trained on a

**Tab. 4.6:** Ablation Study.

| Influence factor | Ranking hold-out | Ranking eval | Accuracy eval |
|---|---|---|---|
| Full model | 76.5 (±0.0) | 57.7 (±0.0) | 60.6 (±0.0) |
| No histogram loss | 72.5 (–4.0) | 49.6 (–8.1) | 30.9 (–29.7) |
| No masking | 75.2 (–1.3) | 54.3 (–3.4) | 50.0 (–10.6) |
| No augmentation | 75.3 (–1.2) | 53.6 (–4.1) | 50.0 (–10.6) |
| Bitmap CNN original[564] | 76.2 (–0.3) | 71.9 (+14.2) | 68.3 (+7.7) |
| Bitmap CNN retrained | 70.0 (–6.5) | 50.0 (–7.7) | 52.9 (–7.7) |

larger dataset of around 25 000 papers, probably including some of the papers in our test set. We use their code to re-train on our subset of equations and yield a substantial margin of 6.5 percentage points.

We also use our previous evaluation data [564]. It consists of 103 equations labeled into 13 categories related to machine learning including k-means, LSTMs, empirical risk minimization, etc. Since only bitmaps are available, we transcribe the equations manually. There are three issues with this evaluation set. First, it is too small to produce significant numbers. Second, some equations in the dataset appear in the training data. This is not only the case for our subset, but also for the training data used in [564]. Third, many equations within a category are obviously from the same paper, hence we have seen some of the pairs in our training data. Nevertheless we use the evaluation data. Indeed in our use-case of search engines, the crawled equations will always be in the training data and only the user queries will be unseen equations. In a way, we simulate this with the evaluation data.

Following the original experimental protocol, we measure the 1-nearest-neighbor accuracy obtained in leave-one-out validation (named Accuracy) as well as the above Ranking score. In Table 4.6, we again see that our model is only surpassed by the pre-trained model that uses a larger training dataset. This motivates the use of a much larger dataset.

#### 4.5.5.2 Large-Scale Experiments

For training on all the papers in our dataset, we sample two different sets of training triplets, one with 5 million triplets and one with 20 million triplets. We train our models on a Nvidia GTX1080 GPU with 8 GB memory, which allows us to process mini-batches of 128 triplets, or 384 equations. During training, we process around 1300 triplets per second, not counting the time for reading data from hard disk. In total, one of the 20 epochs of training on 20 million triplets takes 6:30h on our system. We use annoy to construct an index for approximate nearest neighbor retrieval. In total, our index uses 13 GB of hard disk storage to manage all mathematical expressions in our dataset.

**Tab. 4.7:** Evaluation Scores.

| Dataset | Ranking eval | Accuracy eval |
|---|---|---|
| 1mio ML-subset triplets | 57.7 | 60.6 |
| 5mio full ArXiv triplets | **76.2** | 80.9 |
| 20mio full ArXiv triplets | 75.3 | **84.0** |
| Bitmap CNN original[564] | 71.9 | 68.3 |

Before we evaluate our models in a search engine study, we again check the performance on the aforementioned evaluation data. The results in Table 4.7 indicate the power of using large amounts of training data, although it is unclear if using 20 million training triplets is an advantage over using only 5 million. Our large-scale models beat all the models trained on smaller amounts of data. Even though the smaller models were trained on only machine learning-related data, we obtain better scores on the machine learning evaluation data by training on all disciplines.

Let us now inspect two example search queries. In Figures 4.25 and 4.26 we see the two examples from the introduction, Bayes law and Ising models, and their respective nearest neighbors under our model trained on 5 million triplets. We see that we can find other definitions of Bayes' law as well as the related law of total probability. When we perform a query for the Ising model and look at the first 20 results, we find papers where the model is called the Boltzmann machine as well as papers that refer to the Ising model. This illustrates the power of querying for mathematical expressions instead of using keywords.

#### 4.5.5.3 Search Engine Study

Finally we want to study the usefulness of our embedding approach for a search engine application more systematically. Traditionally, validating search engines using the measures precision or recall requires relevance scores for each result for each evaluation query. We see that this requires much manual annotation work since we have to manually identify each relevant equation for each query. Unfortunately, we were not able to find available evaluation data. The best fit is the NTCIR-12 task evaluation data [729] consisting of 37 annotated queries. But this is not appropriate for our ap-

| | |
|---|---|
| Query: | $P(d \mid s) = \frac{P(d,s)}{P(s)}$ |
| 1st result: | $P(s \mid d) = \frac{P(d\|s)P(s)}{P(d)}$ |
| 4th result: | $P(d) = \int P(d \mid s)P(s)ds$ |

**Fig. 4.25:** Example: Bayes' law. We report the first result and the first result that does not show Bayes' law, but, in this case, the related law of total probability. The first result is from: R. H. Leike, T. A. Enßlin, *Charting nearby dust clouds using Gaia data only*, 2019.

| | |
|---|---|
| Query: | $\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i$ |
| 'Boltzmann' Result: | $E = -\sum_i b_i\, s_i - \sum_{i<j} w_{ij} s_i s_j.$ |
| 'Ising' Result: | $\mathcal{H} = -\sum_{i<j} C_{ij}\, J_{ij} \sigma_i \sigma_j - \sum_i h_i \sigma_i$ |

**Fig. 4.26:** Example: Ising model. We find equations related to both Ising models and Boltzmann machines. First result is from: Weinstein, *Learning the Einstein-Podolsky-Rosen correlations on a Restricted Boltzmann Machine*, 2017. Second result is from: Ferrari et al., *Finite size corrections to disordered systems on Erdős–Rényi random graphs*, 2013.

proach, as most queries are a combination of math as well as keywords. When we ignore the keywords, the remaining query becomes very generic, for instance $x + y$, which makes it very unlikely that we accurately find the articles labeled as relevant. In addition, the overall focus of the NTCIR-12 task is recovery of exact matches, whereas our focus is on retrieving *related* expressions.

Consequently, we curate and publish our own evaluation dataset. To reduce the manual annotation labour, we want to apply a heuristic for the relevance judgement. To this end, we have asked our colleagues, many from disciplines other than computer science and data science, to provide us with equations that we should query. For each equation, they provide a set of keywords or keyphrases that should appear in the section around the result. If one of the keywords is present, we count the result as correct. In this way, we can evaluate our search result without manually checking result lists. If a keyword has more than 10 characters, we also count it, if we find a substring that has a Levenshtein distance less than 2. In total, we have 53 evaluation queries publicly available and editable online.[17]

We inspect two different information retrieval metrics that do not require the number of relevant documents in advance: Precision@$k$ and unnormalized Mean Average Precision. Precision@$k$ is defined as the fraction of relevant documents within the first $k$ results. We report it for lists of 10, 100, and 1000 results and compute its mean over our evaluation queries.

Unnormalized Mean Average Precision is derived from the standard mean average precision metric. Since we do not now the number of relevant documents in advance, we omit this term, limit the search to a maximum of 1000 results, and obtain the following definition

$$\text{uMAP} = \sum_{k=1}^{1000} P(k)\Delta_k$$

where $P(k)$ is Precision@$k$ and $\Delta_k$ specifies if the $k$-th result is relevant. Again we compute the mean over all evaluation queries. Compared with Precision@$k$, uMAP

**17** Crowd-sourced evaluation data can be accessed and edited here: https://www.overleaf.com/8721648589nrjxgwmtzfvm.

**Tab. 4.8:** Search Engine Performance

| | P@10 | P@100 | P@1000 | uMAP |
|---|---|---|---|---|
| BoW | 0.4567 | 0.3170 | 0.2083 | 106.17 |
| 5Mio | **0.5038** | **0.3817** | **0.2984** | **165.04** |
| 20Mio | 0.4547 | 0.3709 | 0.2897 | 156.51 |

considers the order of the search results and rewards relevant results early in the result lists.

For reference, we include retrieval based on a bag-of-words (BoW) representation. To this end, we use our data representation as in Section 4.5.3.2, but compute the sum over all nodes in the graph to obtain a single 256-dimensional vector of the whole tree. We retrieve the nearest neighbors using cosine similarity.

In Table 4.8, we see that our approach beats the bag-of-words margin, in particular for larger values of $k$. We see for Precision@10, the performance between BoW and our embedding model is very similar. This is because for many queries the top-10 results are mostly near-perfect matches that are easily identified. However when looking at more results, we are able to find almost 50 % more relevant equations.

#### 4.5.5.4 Retrieval of Equalities and Inequalities

We have extracted equalities and inequalities in the test set of our data using regular expressions. Using a simple heuristic, we filter the resulting (in-)equalities, such that left-hand-side (LHS) and right-hand-side (RHS) do not differ in length dramatically, thereby eliminating formulas such as definitions, where the LHS is only a single symbol. We derive three different datasets, one with only equalities (LHS and RHS split at "="), one with inequalities (split at < and ≤) and one with mixed relations (split at =<>≤ and ≥). This data allows us to use the LHS of the (in-)equalities as queries in hopes of retrieving RHS. We have made our finetuning-data available at https://whadup.github.io/arxiv_learning/ as well.

Following other machine learning-based approaches for mathematical retrieval [464, 563, 564], we use our models to encode formulas into a dense vector space and retrieve results using approximate nearest neighbor search [48]. In the case of our BERT models, we use output embedding of the CLS token as the representation for the whole formula and finetune the model to output meaningful embeddings for this first token. We finetune our models on half of the available data and test on the remaining half.

**Finetuning Task** We propose using contrastive learning to learn to identify the RHS given the LHS. The learning task in contrastive learning is identifying the right partner for each input in a minibatch of datapoints. Hence the representation learning problem is formulated as a classification problem. Let $X^l, X^r \in \mathbb{R}^{m \times d}$ contain the output embeddings of a minibatch of LHSs and RHSs. We normalize each embedding to unit length

and denote the normalized embeddings by $\bar{X}^l$ and $\bar{X}^r$. We use the InfoNCE loss[547], i.e. the negative log-likelihood of softmax probabilities parameterized by the pairwise cosine similarities between the LHSs and RHSs:

$$\ell_\tau(\bar{X}^l, \bar{X}^r) = m^{-1} \sum_{i=1}^{m} \log \frac{\exp(\langle \bar{X}_i^l, \bar{X}_i^r \rangle / \tau)}{\sum_{j \neq i} \exp(\langle \bar{X}_i^l, \bar{X}_j^r \rangle / \tau)} \tag{4.42}$$

where $\tau > 0$ is a hyperparameter that controls the temperature of the output probability distribution, which we set to $10^{-2}$. The contrastive learning task is more difficult for larger batchsizes $m$, as there are more candidate RHSs to chose from and thus the underlying classification problem becomes more difficult. But it has been shown that the utility of the model increases for larger batchsizes [134, 496], which we also investigate in our application.

**Baseline Models** In addition to our models we include several baseline models:

- First, we test a simple *bag-of-words* (BoW) model that is trained on a bag of `MathML` tree nodes. This model does not use a pre-training phase, but is only tuned on the finetuning data. The BoW model maps the sparse BoW representation to a $d$-dimensional vectorial embedding though a single matrix multiplication. In comparison with our BERT models, we do not restrict the vocabulary size of the inputs. The representation is trained using the same contrastive learning task with InfoNCE loss. We test $d \in \{64, 128, 256\}$ and report the best result after varying learning rates and number of training epochs in a grid search.
- Second we evaluate a pretraining approach based on the BoW model. The word-embedding-based approach *fastText* by Joulin et al.[350] is trained by predicting which tokens appear in the contexts together. Mansoury et al. use it for learning embeddings of formulae by serializing a `MathML` layout tree similar to the one we use in this work [464], hence we include it in our comparison. We finetune these embeddings by learning a linear mapping into a $d$-dimensional vector space, $d \in \{64, 128, 256\}$ using the same contrastive learning task.

We begin by training our models and the baseline models with a minibatch-size of 1024. Then we also investigate the effect of varying the batch size. Our implementations of all methods is available at http://github.com/Whadup/arxiv-learning.

**Results** For testing, we compute embeddings for all LHSs and RHSs in the test data and store them in an index structure. We use annoy [48], an indexing method for an approximate nearest-neighbor search based on an ensemble of random projection trees. We use an ensemble of 16 trees with default hyperparameters, but we found that the results were very insensitive to our particular parameter choices.

Then we query the $k$-nearest neighbors, $k \in \{1, 10, 100\}$, for each formula from the test set and check if the corresponding other side of the (in-)equality is in the result set. This way we can compute recall values to measure the quality of our embeddings.

**Tab. 4.9:** Results of the mathematical retrieval experiment. We report recall@K for $K \in \{1, 10, 100\}$.

| Model | Equalities (36864) | | | Relations (40960) | | | Inequalities (13312) | | |
|---|---|---|---|---|---|---|---|---|---|
| | R@1 | R@10 | R@100 | R@1 | R@10 | R@100 | R@1 | R@10 | R@100 |
| SMALL-PRE | 0.379 | 0.577 | 0.714 | 0.432 | 0.626 | 0.749 | 0.397 | 0.661 | 0.795 |
| SMALL-NO-PRE | 0.400 | 0.584 | 0.700 | 0.405 | 0.632 | 0.769 | 0.371 | 0.632 | 0.769 |
| BASE-PRE | **0.511** | 0.71 | 0.805 | 0.503 | 0.697 | 0.791 | 0.484 | 0.765 | 0.86 |
| BASE-NO-PRE | 0.434 | 0.623 | 0.729 | 0.446 | 0.63 | 0.734 | 0.409 | 0.682 | 0.797 |
| LARGE-PRE | 0.507 | 0.704 | 0.799 | 0.496 | 0.683 | 0.777 | 0.489 | 0.765 | 0.864 |
| LARGE-NO-PRE | 0.452 | 0.637 | 0.737 | 0.46 | 0.640 | 0.736 | 0.427 | 0.703 | 0.817 |
| BOW | 0.483 | 0.653 | 0.739 | 0.491 | 0.658 | 0.743 | 0.503 | 0.738 | 0.821 |
| FASTTEXT [350] | 0.480 | 0.650 | 0.739 | 0.488 | 0.651 | 0.742 | 0.488 | 0.713 | 0.810 |
| GNN [563] | 0.507 | **0.833** | **0.884** | **0.512** | **0.834** | **0.883** | **0.504** | **0.870** | **0.922** |

We summarize our findings in Table 4.9. Our BERT approach substantially outperforms both the BoW approaches, without (BOW) and with pretraining (FASTTEXT). This suggests that our model is capable of matching formulas based on characteristics that go beyond merely counting the number of matching tokens. However, the graph neural network GNN outperforms the sequential models in most scenarios, sometimes even substantially. It is, however, noteworthy that of the transformer models, the mid-size model is most useful.

For the mid-size and large models we observe the benefit of pretraining, as models that were trained from scratch perform worse than their pretrained counterparts. For the small models we do not consistently see this effect.

Overall, the recall at 10 for our approaches is already pretty high, which indicates that our representation learning on structured data is useful in search engine applications where users generally want to inspect only a small number of results.

### 4.5.6 Conclusion

Finding relevant literature across disciplines is essential for research. The search results should contain papers that are both relevant and stimulating. Very often, a look at the formulas in a paper gives a compact description of the problems and solutions it discusses. Hence, the goal is to find related papers based on the mathematical expressions. This task is different from mathematical information retrieval, but it shares the problem of determining the right representation of mathematical expressions.

In order to handle the large amounts of data that are common in search engine applications, we need models that allow efficient computation of the vector representations. Our approach based on graph-neural networks is a good fit for this demand as it makes use of the sparsely connected input graphs. As such it is much more com-

putationally efficient than the other transformer models that we considered in this contribution.

We have demonstrated that representation learning on structured input is a useful approach for mathematical retrieval. Self-supervised and embedding learning successfully learned real-valued representations of tree-structures that allow efficient nearest-neighbor searches.