Adversarial Permutation Guided Node Representations for Link Prediction

Indradyumna Roy, Abir De, Soumen Chakrabarti

Indian Institute of Technology Bombay

{indraroy15, abir, soumen}@cse.iitb.ac.in

Abstract

After observing a snapshot of a social network, a link prediction (LP) algorithm identifies node pairs between which new edges will likely materialize in future. Most LP algorithms estimate a score for currently non-neighboring node pairs, and rank them by this score. Recent LP systems compute this score by comparing dense, low dimensional vector representations of nodes. Graph neural networks (GNNs), in particular graph convolutional networks (GCNs), are popular examples. For two nodes to be meaningfully compared, their embeddings should be indifferent to reordering of their neighbors. GNNs typically use simple, symmetric set aggregators to ensure this property, but this design decision has been shown to produce representations with limited expressive power. Sequence encoders are more expressive, but are permutation sensitive by design. Recent efforts to overcome this dilemma turn out to be unsatisfactory for LP tasks. In response, we propose PERMGNN, which aggregates neighbor features using a recurrent, order-sensitive aggregator and directly minimizes an LP loss while it is 'attacked' by adversarial generator of neighbor permutations. PERMGNN has superior expressive power compared to earlier GNNs. Next, we devise an optimization framework to map PERMGNN's node embeddings to a suitable locality-sensitive hash, which speeds up reporting the top-K most likely edges for the LP task. Our experiments on diverse datasets show that PERM-GNN outperforms several state-of-the-art link predictors, and can predict the most likely edges fast.

1 Introduction

In the link prediction (LP) task, we are given a snapshot of a social network, and asked to predict future links that are most likely to emerge between nodes. LP has a wide variety of applications, *e.g.*, recommending friends in Facebook, followers in Twitter, products in Amazon, or connections on LinkedIn. An LP algorithm typically considers current non-edges as potential edges, and ranks them by decreasing likelihoods of becoming edges in future.

1.1 Prior Work and Their Limitations

LP methods abound in the literature, and predominantly follow two approaches. The first approach relies strongly on hand-engineering node features and edge likelihoods

based on the network structure and domain knowledge (Katz 1997; Liben-Nowell and Kleinberg 2007; Backstrom and Leskovec 2011). However, such feature engineering often demands significant domain expertise. The second approach learns low dimensional node embeddings which serve as node features in LP tasks. Such embedding models include Node2Vec (Grover and Leskovec 2016), Deep-Walk (Perozzi, Al-Rfou, and Skiena 2014), etc., and various graph neural networks (GNN), *e.g.*, GCN (Kipf and Welling 2016a), GraphSAGE (Hamilton, Ying, and Leskovec 2017), GAT (Veličković et al. 2017), etc.

Limited expressive power of GNNs. While deep graph representations have shown significant potential in capturing complex relationships between nodes and their neighborhoods, they lack representational power useful for LP. A key reason for this weakness is the use of symmetric aggregates over a node u's neighbors, driven by the desideratum that the representation of u should be invariant to a permutation of its neighbor nodes (Zaheer et al. 2017; Ravanbakhsh, Schneider, and Poczos 2016; Qi et al. 2017). Such networks have recently been established as low-pass filters (Wu et al. 2019; NT and Maehara 2019), which attenuate high frequency signals. This prevents LP methods based on such node representations from reaching their full potential. Although recent efforts (Lee et al. 2019; Bloem-Reddy and Teh 2019; Shi, Oliva, and Niethammer 2020; Stelzner, Kersting, and Kosiorek 2020; Skianis et al. 2020; Zhang and Chen 2018) on modeling inter-item dependencies have substantially improved the expressiveness of set representations in applications like image and text processing, they offer only modest improvement for LP, as we shall see in our experiments. Among these approaches, SEAL (Zhang and Chen 2018) improves upon GNN performance but does not readily lend itself to efficient top-K predictions via LSH.

Limitations of sequence driven embeddings. We could arrange the neighbors of u in some arbitrary canonical order, and combine their features sequentially using, say, a recurrent neural network (RNN). This would capture feature correlations between neighbors. But now, the representation of u will become sensitive to the order in which neighbors are presented to the RNN. In our experiments, we see loss degradation when neighbors are shuffled. We seek to resolve this central dilemma. An obvious attempted fix would be to present many permutations (as Monte Carlo samples)

of neighbor nodes but, as we shall see, doing so in a data-oblivious manner is very inefficient in terms of space and time.

1.2 Our Proposal: PERMGNN

In response to the above limitations in prior work, we develop PERMGNN: a novel node embedding method specifically designed for LP. To avoid the low-pass nature of GNNs, we eschew symmetric additive aggregation over neighbors of a node u, instead using a recurrent network to which neighbor node representations are provided sequentially, in some order. The representation of u is computed by an output layer applied on the RNN states.

To neutralize the order-sensitivity of the RNN, we cast LP as a novel min-max optimization, equivalent to a game between an adversary that generates worst-case neighbor permutations (to maximize LP loss) and a node representation learner that refines node representations (to minimize LP loss) until they become insensitive to neighborhood permutations. To facilitate end-to-end training and thus avoiding exploration of huge permutation spaces, the adversarial permutation generator is implemented as a Gumbel-Sinkhorn neural network (Mena et al. 2018).

Next, we design a hashing method for efficient LP, using the node representation learnt thus far. We propose a smooth optimization to compress the learned embeddings into binary representations, subject to certain hash performance constraints. Then we leverage locality sensitive hashing (Gionis, Indyk, and Motwani 1999) to assign the bit vectors to buckets, such that nodes likely to become neighbors share buckets. Thus, we can limit the computation of pairwise scores to within buckets. In spite of this additional compression, our hashing mechanism is accurate and fast.

We evaluate PERMGNN on several real-world datasets, which shows that our embeddings can suitably distill information from node neighborhoods into compact vectors, and offers accuracy boosts beyond several state-of-the-art LP methods[1], while achieving large speed gains via LSH.

1.3 Summary of Contributions

(1) **Adversarial permutation guided embeddings:** We propose PERMGNN, a novel node embedding method, which provides high quality node representations for LP. In a sharp contrast to additive information aggregation in GNNs, we start with a permutation-sensitive but highly expressive aggregator of the graph neighbors and then desensitize the permutation-sensitivity by optimizing a min-max ranking loss function with respect to the smooth surrogates of adversarial permutations.
(2) **Hashing method for scalable predictions:** We propose an optimized binary transformation to the learnt node representations, that readily admits the use of a locality-sensitive hashing method and shows fast and accurate predictions.
(3) **Comprehensive evaluation:** We provide a rigorous evaluation to test both the representational power of PERM-GNN and the proposed hashing method, which show that our proposal usually outperforms classical and recent methods.

Further probing the experimental results reveal insightful explanations behind the success of PERMGNN.

2 Preliminaries

In this section, we describe necessary notation and the components of a typical LP system.

2.1 Notation

We consider a snapshot of an undirected social network $G = (V, E)$. Each node u has a feature vector $\boldsymbol{f}_u$. We use $\mathrm{nbr}(u)$ and $\overline{\mathrm{nbr}}(u)$ to indicate the set of neighbors and non-neighbors of u. Our graphs do not have self edges, but we include u in $\mathrm{nbr}(u)$ by convention. We define $\mathrm{nbr}(u) = \{u\} \cup \{v \mid (u, v) \in E\}$, $\overline{\mathrm{nbr}}(u) = \{v \mid v \neq u, (u, v) \notin E\}$ and also $\overline{E}$ to be the set of non-edges, *i.e.*, $\overline{E} = \cup_{u \in V} \overline{\mathrm{nbr}}(u)$. Finally, we define Π_δ to be the set of permutations of the set $[\delta] = \{1, 2, ..., \delta\}$ and $\mathcal{P}_\delta$ to be the set of all possible 0/1 permutation matrices of size $\delta \times \delta$.

2.2 Scoring and Ranking

Given a graph snapshot $G = (V, E)$, the goal of a LP algorithm is to identify node-pairs from the current set of non-edges $\overline{E}$ (often called potential edges) that are likely to become edges in future. In practice, most LP algorithms compute a **score** $s(u, v)$ for each potential edge $(u, v) \in \overline{E}$, which measures their likelihood of becoming connected in future. Recently invented network embedding methods (Kipf and Welling 2016a; Grover and Leskovec 2016; Salha et al. 2019) first learn a latent representation $\mathbf{x}_u$ of each node $u \in V$ and then compute scores $s(u, v)$ using some similarity or distance measure between the corresponding representations $\mathbf{x}_u$ and $\mathbf{x}_v$. In the test fold, some nodes are designated as *query* nodes q. Its (current) non-neighbors v are sorted by decreasing $s(q, v)$. We are primarily interested in LP systems that can retrieve a small number of K nodes with largest $s(q, v)$ for all q in $o(N^2)$ time.

3 Proposed Approach

In this section, we first state the limitations of GNNs. Then, we present our method for obtaining high quality node embeddings, with better representational power than GNNs.

3.1 GNNs and Their Limitations

GNNs start with a graph and per-node features $\boldsymbol{f}_u$ to obtain a neighborhood-sensitive node representation $\mathbf{x}_u$ for $u \in V$. To meaningfully compare $\mathbf{x}_u$ and $\mathbf{x}_v$ and compute $s(u, v)$, information from neighbors of u (and v) should be aggregated in such a way that the embeddings become invariant to permutations of the neighbors of u (and v). GNNs ensure permutation invariance by additive aggregation. Given an integer K, for each node u, a GNN aggregates structural information k hops away from u to cast it into $\mathbf{x}_u$ for $k \leq K$. Formally, a GNN first computes intermediate embeddings $\{\mathbf{z}_u(k) \mid k \in [K]\}$ in an iterative manner and then computes

$\mathbf{x}_u$, using the following recurrent propagation rule.

$$\overline{\mathbf{z}}_u(k-1) = \text{AGGR}\left(\{\mathbf{z}_v(k-1)\,|\,v \in \text{nbr}(u)\}\right); \quad (1)$$

$$\mathbf{z}_u(k) = \text{COMB}_1\left(\mathbf{z}_u(k-1),\,\overline{\mathbf{z}}_u(k-1)\right); \quad (2)$$

$$\mathbf{x}_u = \text{COMB}_2(\mathbf{z}_u(1),\dots,\mathbf{z}_u(K)) \quad (3)$$

Here, for each node u with feature vector $\boldsymbol{f}_u$, we initialize $\mathbf{z}_u(0) = \boldsymbol{f}_u$; AGGR and COMB$_{1,2}$ are neural networks. To ensure permutation invariance of the final embedding $\mathbf{x}_u$, AGGR aggregates the intermediate $(k-1)$-hop information $\mathbf{z}_v(k-1)$ with an additive (commutative, associative) function, guided by set function principles (Zaheer et al. 2017):

$$\text{AGGR}\left(\{\mathbf{z}_v(k-1)\,|\,v \in \text{nbr}(u)\}\right)$$
$$= \sigma_1\left(\sum_{v \in \text{nbr}(u)} \sigma_2\left(\mathbf{z}_v(k-1)\right)\right). \quad (4)$$

Here σ_1, σ_2 are nonlinear activations. In theory (Zaheer et al. 2017, Theorem 2), if COMB$_{1,2}$ are given 'sufficient' hidden units, this set representation is universal. In practice, however, commutative-associative aggregation suffers from limited expressiveness (Pabbaraju and Jain 2019; Wagstaff et al. 2019; Garg, Jegelka, and Jaakkola 2020; Cohen-Karlik, David, and Globerson 2020), which degrades the quality of $\mathbf{x}_u$ and $s(\cdot,\cdot)$, as described below. Specifically, their expressiveness is constrained from two perspectives.

Attenuation of important network signals. GNNs are established to be intrinsically low pass filters (NT and Maehara 2019; Wu et al. 2019). Consequently, they can attenuate high frequency signals which may contain crucial structural information about the network. To illustrate, assume that the node u in Eqs. (1)–(3) has two neighbors v and w and $\mathbf{z}_v(k-1) = [+1, -1]$ and $\mathbf{z}_w(k-1) = [-1, +1]$, which induce high frequency signals around the neighborhood of u. In practice, these two representations may carry important signals about the network structure. However, popular choices of σ_2 often diminish the effect of each of these vectors. In fact, the widely used linear form of σ_2 (Hamilton, Ying, and Leskovec 2017; Kipf and Welling 2016a) would completely annul their effects (since $\sigma_2(\mathbf{z}_v(k-1)) + \sigma_2(\mathbf{z}_w(k-1)) = \mathbf{0}$) in the final embedding $\mathbf{x}_u$, which would consequently lose capacity for encapsulating neighborhood information.

Inability to distinguish between correlation structures. In Eq. (4), the outer nonlinearity σ_1 operates over the sum of all representations of neighbors of u. Therefore, it cannot explicitly model the variations between the joint dependence of these neighbors. Suppose the correlation between $\mathbf{z}_v(k-1)$ and $\mathbf{z}_w(k-1)$ is different from that between $\mathbf{z}_{v'}(k-1)$ and $\mathbf{z}_{w'}(k-1)$ for $\{v, v', w, w'\} \subseteq \text{nbr}(u)$. The additive aggregator in Eq. (4) cannot capture the distinction.

Here, we develop a mitigation approach which exploits sequential memory, e.g., LSTMs, even though they are order-sensitive, and then neutralize the order sensitivity by presenting adversarial neighbor orders. An alternative mitigation approach is to increase the capacity of the aggregator (while keeping it order invariant by design) by explicitly modeling dependencies between neighbors, as has been attempted in image or text applications (Lee et al. 2019; Bloem-Reddy and Teh 2019; Shi, Oliva, and Niethammer 2020; Stelzner, Kersting, and Kosiorek 2020).

3.2 Our Model: PERMGNN

Responding to the above limitations of popular GNN models, we design PERMGNN, the proposed adversarial permutation guided node embeddings.

Overview. Given a node u, we first compute an embedding $\mathbf{x}_u$ using a *sequence* encoder, parameterized by θ:

$$\mathbf{x}_u = \rho_\theta\left(\{\boldsymbol{f}_v\,|\,v \in \text{nbr}(u)\}\right), \quad (5)$$

where $\text{nbr}(u)$ is presented in some arbitrary order (to be discussed). In contrast to the additive aggregator, ρ is modeled by an LSTM (Hochreiter and Schmidhuber 1997), followed by a fully-connected feedforward neural network (See Figure 1). Such a formulation captures the presence of high frequency signal in the neighborhood of u and the complex dependencies between the neighbors $\text{nbr}(u)$ by combining their influence via the recurrent states of the LSTM.

However, now the embedding $\mathbf{x}_u$ is no longer invariant to the permutation of the neighbors $\text{nbr}(u)$. As we shall see, we counter this by casting the LP objective as an instance of a min-max optimization problem. Such an adversarial setup refines $\mathbf{x}_u$ in an iterative manner, to ensure that the resulting trained embeddings are permutation invariant (at least as far as possible in a non-convex optimization setting).

PERMGNN architecture. Let us suppose $\boldsymbol{\pi} = [\pi_1, ..., \pi_{|\text{nbr}(u)|}] \in \Pi_{|\text{nbr}(u)|}$ is some arbitrary permutation of the neighbors of node u. We take the features of neighbors of u in the order specified by $\boldsymbol{\pi}$, i.e., $\left(v_{\pi_1}, v_{\pi_2}, \dots, v_{\pi_{|\text{nbr}(u)|}}\right)$, and pass them into an LSTM:

$$\mathbf{y}_{u,1}, ..., \mathbf{y}_{u,|\text{nbr}(u)|} = \text{LSTM}_\theta\left(\boldsymbol{f}_{v_{\pi_1}}, ..., \boldsymbol{f}_{v_{\pi_{|\text{nbr}(u)|}}}\right). \quad (6)$$

Here $\left(\mathbf{y}_{u,k}\right)_{k \in [|\text{nbr}(u)|]}$ is a sequence of intermediate representation of node u, which depends on the permutation $\boldsymbol{\pi}$. Such an approach ameliorates the limitations of GNNs in two ways:

(1) Unlike GNNs, the construction of $\mathbf{y}_\bullet$ is not limited to symmetric aggregation, and is therefore able to capture crucial network signals including those with high frequency (Borovkova and Tsiamas 2019).

(2) An LSTM (indeed, any RNN variant) is designed to capture the influence of one token of the sequence on the subsequent tokens. In the current context, the state variable $\boldsymbol{h}_k$ of the LSTM combines the influence of first $k-1$ neighbors in the input sequence, *i.e.*, $v_{\pi_1}, \dots v_{\pi_{k-1}}$ on the k-th neighbor v_{π_k}. Therefore, these recurrent states allow $\mathbf{y}_\bullet$ to capture the complex dependence between the features $\boldsymbol{f}_\bullet$.

Next, we compute the final embeddings $\mathbf{x}_u$ by using an additional nonlinearity on the top of the sequence $(\mathbf{y}_{u,k})_{k \in [|\text{nbr}(u)|]}$ output by the LSTM:

$$\mathbf{x}_{u;\boldsymbol{\pi}} = \sigma_\theta\left(\mathbf{y}_{u,1}, \mathbf{y}_{u,2}, \dots, \mathbf{y}_{u,|\text{nbr}(u)|}\right) \in \mathbb{R}^D. \quad (7)$$

Note that the embeddings $\{\mathbf{x}_u\}$ computed above depends on $\boldsymbol{\pi}$, the permutation of the neighbors $\text{nbr}(u)$ given as the input sequence to the LSTM in Eq. (6).

Removing the sensitivity to $\boldsymbol{\pi}$. One simple way to ensure permutation invariance is to compute the average of $\mathbf{x}_{u;\boldsymbol{\pi}}$ over all permutations $\boldsymbol{\pi} \in \Pi_{|\text{nbr}(u)|}$, similar to Murphy et al. (2019a). At a time and space complexity of at least $O(\sum_{u \in V} |\Pi_{|\text{nbr}(u)|}|)$, this is quite impractical for even
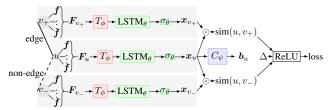
Figure 1: PERMGNN min-max loss and hashing schematic.

moderate degree nodes. Replacing the exhaustive average by a Monte Carlo sample does improve representation quality, but is still very expensive. Murphy et al. (2019b) proposed a method called π-SGD, which samples one permutation per epoch. While it is more efficient than sampling multiple permutations, it shows worse robustness in practice.

Adversarial permutation-driven LP objective. Instead of brute-force sampling, we setup a two-party game, one being the network for LP, vulnerable to π, and the other being an adversary, which tries to make the LP network perform poorly by choosing a 'bad' π at each node.

1: pick initial π^u at each node u
2: **repeat**
3:      fix $\{\pi^u : u \in V\}$; optimize θ for best LP accuracy
4:      fix θ; find next π^u at all u for worst LP accuracy
5: **until** LP performance stabilizes

Let $\pi^u \in \Pi_{|\text{nbr}(u)|}$ be the permutation used to shuffle the neighbors of u in Eq. (6). Conditioned on π^u, π^v, we compute the score for a node-pair (u, v) as

$$s_\theta(u, v | \pi^u, \pi^v) = \text{sim}(\mathbf{x}_{u;\pi^u}, \mathbf{x}_{v;\pi^v}), \qquad (8)$$

where $\text{sim}(\boldsymbol{a}, \boldsymbol{b})$ denotes the cosine similarity between $\boldsymbol{a}$ and $\boldsymbol{b}$. To train our LP model to give high quality ranking, we consider the following AUC loss surrogate (Joachims 2005):

$$\text{loss}(\theta; \{\pi^w\}_{w \in V})$$
$$= \sum_{\substack{(u,v) \in E \\ (r,t) \in \overline{E}}} \left[\Delta + s_\theta(r,t) | \pi^r, \pi^t - s_\theta(u,v | \pi^u, \pi^v) \right]_+ \quad (9)$$

where Δ is a tunable margin and $[a]_+ = \max\{0, a\}$.

As stated above, we aim to train LP model parameters θ in such a way that the trained embeddings $\{\mathbf{x}_u\}$ become invariant to the permutations of $\text{nbr}(u)$ for all nodes $u \in V$. This requirement suggests the following min-max loss:

$$\min_\theta \ \max_{\{\pi^w\}_{w \in V}} \ \text{loss}(\theta; \{\pi^w\}_{w \in V}). \qquad (10)$$

Neural permutation surrogate. As stated, the complexity of Eq. (10) seems no better than exhaustive enumeration of permutations. To get past this apparent blocker, just as max is approximated by softmax (a multinomial distribution), a 'hard' permutation (1:1 assignment) π^w is approximated by a 'soft' permutation matrix $\boldsymbol{P}^w$ — a doubly stochastic matrix — which allows continuous optimization.

Suppose $\boldsymbol{F}_w = \left[\boldsymbol{f}_{v_1}, \boldsymbol{f}_{v_2}, \ldots, \boldsymbol{f}_{v_{|\text{nbr}(w)|}} \right]$ is a matrix whose rows are formed by the features of $\text{nbr}(w)$ presented in some canonical order. Then $\boldsymbol{P}^w \boldsymbol{F}_w$ approximates a permuted feature matrix corresponding to some permuted se-

quence of neighbor feature vectors. The RHS of Eq. (6) can be written as $\text{LSTM}_\theta(\boldsymbol{P}^w \boldsymbol{F}_w)$, which eventually lets us express loss as a function of $\boldsymbol{P}^w$. We can thus rewrite the min-max optimization (10) as

$$\min_\theta \ \max_{\{\boldsymbol{P}^w \, | \, w \in V\}} \ \text{loss}(\theta; \{\boldsymbol{P}^w\}_{w \in V}), \qquad (11)$$

where the inner maximization is carried out over all 'soft' permutation matrices $\boldsymbol{P}^w$, parameterized as follows.

In deep network design, a trainable multinomial distribution is readily obtained by applying a softmax to trainable (unconstrained) logits. Analogously, a trainable soft permutation matrix $\boldsymbol{P}^w$ can be obtained by applying a Gumbel-Sinkhorn network 'GS' (Mena et al. 2018) to a trainable (unconstrained) 'seed' square matrix, say, $\boldsymbol{A}^w$:

$$\boldsymbol{P}^w = \lim_{n \to \infty} \text{GS}^n(\boldsymbol{A}^w), \quad \text{where}$$

$$\text{GS}^0(\boldsymbol{A}^w) = \exp(\boldsymbol{A}^w) \quad \text{and}$$

$$\text{GS}^n(\boldsymbol{A}^w) = \text{COLSCALE}\left(\text{ROWSCALE}\left(\text{GS}^{n-1}(\boldsymbol{A}^w)\right)\right).$$

Here, COLSCALE and ROWSCALE represent column and row normalization. $\text{GS}^n(\boldsymbol{A}^w)$ is the doubly stochastic matrix obtained by consecutive row and column normalizations of $\boldsymbol{A}^w$. It can be shown that

$$\lim_{n \to \infty} \text{GS}^n(\boldsymbol{A}^w) = \underset{\boldsymbol{P} \in \mathcal{P}_{|\text{nbr}(w)|}}{\text{argmax}} \ \text{Tr}\left[\boldsymbol{P}^\top \boldsymbol{A}^w\right]. \qquad (12)$$

GS^n thus represents a recursive differentiable operator that permits backpropagation of loss to $\{\boldsymbol{A}^w\}$. In practice, n is a finite hyperparameter, the larger it is, the closer the output to a 'hard' permutation.

Allocating a separate unconstrained seed matrix $\boldsymbol{A}^w$ for each node w would lead to an impractically large number of parameters. Therefore, we express $\boldsymbol{A}^w$ using a globally shared network $\boldsymbol{T}_\phi$ with model weights ϕ, and the per-node feature matrix $\boldsymbol{F}_w$ already available. I.e., we define

$$\boldsymbol{A}^w := \boldsymbol{T}_\phi(\boldsymbol{F}_w / \tau), \qquad (13)$$

where $\tau > 0$ is a temperature hyperparameter that encourages $\text{GS}^n(\boldsymbol{A}^w)$ toward a 'harder' soft permutation. The above steps allow us to rewrite optimization (11) in terms of θ and ϕ in the form $\min_\theta \max_\phi \text{loss}(\theta; \phi)$. After completing the min-max optimization, the embedding $\mathbf{x}_u$ of a node u can be computed using some arbitrary neighbor permutation. By design, the impact on $\text{sim}(u, v)$ is small when different permutations are used. The extended version of our paper (Roy, De, and Chakrabarti 2021) contains further implementation details.

4 Scalable LP by Hashing Representations

At this point, we have obtained representations $\mathbf{x}_u$ for each node u using PERMGNN. Our next goal is to infer some number of most likely future edges.

Prediction using exhaustive comparisons. Here, we first enumerate the scores for all possible potential edges (the current non-edges) and then report top-K neighbors for each node. Since most real-life social networks are sparse, potential edges can be $\Theta(|V|^2)$ in number. Scoring all of them in large graphs is impractical; we must limit the number of comparisons between potentially connecting node pairs to be as small as possible.

4.1 Data-Oblivious LSH with Random Hyperplanes

When for two nodes u and v, $\text{sim}(u,v)$ is defined as $\cos(\mathbf{x}_u, \mathbf{x}_v)$ with $\mathbf{x}_\bullet \in \mathbb{R}^D$, the classic random hyperplane LSH can be used to hash the embeddings $\mathbf{x}_\bullet$. Specifically, we first draw H uniformly random hyperplanes passing through the origin in the form of their unit normal vectors $\boldsymbol{n}_h \in \mathbb{R}^D, h \in [H]$ (Charikar 2002). Then we set $b_u[h] = \text{sign}(\boldsymbol{n}_h \cdot \mathbf{x}_u) \in \pm 1$ as a 1-bit hash and $\boldsymbol{b}_u \in \pm 1^H$ as the H-bit hash code of node u. Correspondingly, we set up 2^H hash buckets with each node going into one bucket. If the buckets are balanced, we expect each to have $N/2^H$ nodes. Now we limit pairwise comparisons to only node pairs within each bucket, which takes $N^2/2^H$ pair comparisons. By letting H grow slowly with N, we can thus achieve sub-quadratic time. However, such a hashing method is data oblivious—the hash codes are not learned from the distribution of the original embeddings $\mathbf{x}_\bullet$. It performs best when the embeddings are uniformly dispersed in the D-dimensional space, so that the random hyperplanes can evenly distribute the nodes among several hash buckets.

4.2 Learning Data-Sensitive Hash Codes

To overcome the above limitation of random hyperplane based hashing, we devise a data-driven learning of hash codes as explored in other applications (Weiss, Torralba, and Fergus 2009). Specifically, we aim to design an additional transformation of the vectors $\{\mathbf{x}_u\}$ into compressed representations $\{\boldsymbol{b}_u\}$, with the aim of better balance across hash buckets and reduced prediction time.

Hashing/compression network. In what follows, we will call the compression network $C_\psi : \mathbb{R}^D \to [-1,1]^H$, with model parameters ψ. We interpret $\text{sign}\left(C_\psi(\mathbf{x}_u)\right)$ as the required binary hash code $\boldsymbol{b}_u \in \{-1, +1\}^H$, with the surrogate $\tanh(C_\psi(\mathbf{x}_u))$, to be used in the following smooth optimization:

$$\min_\psi \frac{\alpha}{|V|} \sum_{u \in V} \left| \mathbf{1}^\top \tanh(C_\psi(\mathbf{x}_u)) \right|$$
$$+ \frac{\beta}{|V|} \sum_{u \in V} \left\| \left| \tanh(C_\psi(\mathbf{x}_u)) \right| - \mathbf{1} \right\|_1$$
$$+ \frac{\gamma}{|\overline{E}|} \sum_{(u,v) \in \overline{E}} \left| \tanh(C_\psi(\mathbf{x}_u)) \cdot \tanh(C_\psi(\mathbf{x}_v)) \right| \quad (14)$$

Here, $\overline{E}$ is the set of non-edges and $\alpha, \beta, \gamma \in (0,1)$, with $\alpha + \beta + \gamma = 1$ are tuned hyperparameters. The final binary hash code $\boldsymbol{b}_u = \text{sign}(C_\psi(\mathbf{x}_u))$. The salient terms in the objective above seek the following goals.
Bit balance: If each bit position has as many -1s as $+1$s, that bit evenly splits the nodes. The term $\left| \mathbf{1}^\top \tanh(C_\psi(\mathbf{x}_u)) \right|$ tries to bit-balance the hash codes.
No sitting on the fence: The optimizer is prevented from setting $\boldsymbol{b} = \mathbf{0}$ (the easiest way to balance it) by including a term $\sum_h \left| |\boldsymbol{b}[h]| - 1 \right| = \left\| |\boldsymbol{b}| - \mathbf{1} \right\|_1$.
Weak supervision: The third term encourages currently unconnected nodes to be assigned dissimilar bit vectors.

Bucketing and ranking. Note that, we do not expect the dot product between the learned hash codes $\boldsymbol{b}_u \cdot \boldsymbol{b}_v$ to be a good approximation for $\cos(\mathbf{x}_u, \mathbf{x}_v)$, merely that node pairs

---

```
1: Input: Graph G = (V, E); binary hash-codes {b_u};
   query nodes Q; the number (K) of nodes to be recom-
   mended per query node
2: Output: Ranked recommendation list R_q for all q ∈ Q
3: initialize LSH buckets
4: for u ∈ V do
5:     add u to appropriate hash buckets
6: for q ∈ Q do
7:     initialize score heap H_q with capacity K
8: for each LSH bucket B do
9:     for (u, v) ∈ B do
10:        if u ∈ Q then
11:            insert ⟨v, s(u, v)⟩ in H_u; prune if |H_u| > K
12:        if v ∈ Q then
13:            insert ⟨u, s(u, v)⟩ in H_v; prune if |H_v| > K
14: for q ∈ Q do
15:     sort H_q by decreasing score to get ranked list R_q
16: return {R_q | q ∈ Q}
```

Algorithm 1: Reporting ranked list of potential edges fast.

with large $\cos(\mathbf{x}_u, \mathbf{x}_v)$ will be found in the same hash buckets. We form the buckets using the recipe of Gionis, Indyk, and Motwani (1999). We adopt the high-recall policy that node-pair u, v should be scored if u and v share at least one bucket. Algorithm 1 shows how the buckets are traversed to generate and score node pairs, then placed in a heap for retrieving top-K pairs. Details can be found in the extended version of our paper (Roy, De, and Chakrabarti 2021).

5 Experiments

We report on a comprehensive evaluation of PERMGNN and its accompanying hashing strategy. Specifically, we address the following research questions. **RQ1:** How does the LP accuracy of PERMGNN compare with classic and recent link predictors? Where are the gains and losses? **RQ2:** How does PERMGNN compare with brute-force sampling of neighbor permutations? **RQ3:** Exactly where in our adversarially trained network is permutation insensitivity getting programmed? **RQ4:** Does the hashing optimization reduce prediction time, compared to exhaustive computation of pairwise scores? Further experiments are reported by Roy, De, and Chakrabarti (2021).

5.1 Experimental Setup

Datasets. We consider five real world datasets: (1) Twitter (Leskovec and Mcauley 2012), (2) Google+ (Leskovec et al. 2010), (3) Cora (Getoor 2005; Sen et al. 2008), (4) Citeseer (Getoor 2005; Sen et al. 2008) and (5) PB (Ackland et al. 2005).

Baselines. We compare PERMGNN with several hashable LP algorithms. Adamic Adar (AA) and Common Neighbors (CN) (Liben-Nowell and Kleinberg 2007) are classic unsupervised methods. Node2Vec (Grover and Leskovec 2016) and DeepWalk (Perozzi, Al-Rfou, and Skiena 2014) are node embedding methods based on random walks. Graph Convolutional Network (GCN) (Kipf and Welling 2016b),

	Mean Average Precision (MAP)					**Mean Reciprocal Rank (MRR)**				
	Twitter	Google+	Cora	Citeseer	PB	Twitter	Google+	Cora	Citeseer	PB
AA	0.727	0.321	0.457	0.477	**0.252**	0.904	0.553	**0.535**	0.548	0.508
CN	0.707	0.292	0.377	0.401	0.218	**0.911**	0.553	0.460	0.462	**0.516**
Node2Vec	0.673	0.330	0.448	0.504	0.182	0.832	0.551	0.484	0.546	0.333
DeepWalk	0.624	0.288	0.432	0.458	0.169	0.757	0.482	0.468	0.492	0.303
GraphSAGE	0.488	0.125	0.393	0.486	0.077	0.638	0.233	0.425	0.523	0.156
GCN	0.615	0.330	0.408	0.464	0.200	0.789	0.482	0.444	0.505	0.345
Gravity	**0.735**	0.360	0.407	0.462	0.193	0.881	0.540	0.438	0.518	0.330
PERMGNN	**0.735**	**0.385**	**0.480**	**0.560**	0.220	0.880	**0.581**	0.524	**0.600**	0.397

Table 1: MAP and MRR for all LP algorithms (PERMGNN and baselines) on the ranked list of all potential edges ($K = \infty$) across all five datasets, with 20% test set. Numbers in bold font indicate the best performer.

GraphSAGE (Hamilton, Ying, and Leskovec 2017) Gravity (Salha et al. 2019) are node embedding methods based on GNNs. We highlight that SEAL (Zhang and Chen 2018) does not readily lend itself to a hashable LP mechanism and therefore, we do not compare it in this paper. We discuss it in the longer version (Roy, De, and Chakrabarti 2021).

Evaluation protocol. Similar to the evaluation protocol of Backstrom and Leskovec (2011), we partition the edge (and non-edge) sets into training, validation and test folds as follows. For each dataset, we first build the set of query nodes Q, where each query contains at least one triangle around it. Then, for each $q \in Q$, in the original graph, we partition the neighbors $\text{nbr}(q)$ and the non-neighbors $\overline{\text{nbr}}(q)$ which are within 2-hop distance from q into 70% training, 10% validation and 20% test sets, where the node pairs are sampled uniformly at random. We disclose the resulting sampled graph induced by the training and validation sets to the LP model. Then, for each query $q \in Q$, the trained LP model outputs a top-K list of potential neighbors from the test set. Using ground truth, we compute the average precision (AP) and reciprocal rank (RR) of each top-K list. Then we average over all query nodes to get mean AP (MAP) and mean RR (MRR).

5.2 Comparative Analysis of LP Accuracy

First, we address the research question **RQ1** by comparing LP accuracy of PERMGNN against baselines, in terms of MAP and MRR across the datasets.

MAP and MRR summary. Table 1 summarizes LP accuracy across all the methods. We make the following observations. (1) PERMGNN outperforms all the competitors in terms of MAP, in four datasets, except PB, where it is outperformed by AA. Moreover, in terms of MRR, it outperforms all the baselines for Google+ and Citeseer datasets. (2) The performance of GNNs are comparable for Cora and Citeseer. Due to its weakly supervised training procedure, the overall performance of GraphSAGE is poor among the GNN based methods. (3) The classic unsupervised predictors, *i.e.*, AA and CN often beat some recent embedding models. AA is the best performer in terms of MAP in PB and in terms of MRR in Twitter. Since AA and CN encourage triad completion, which is a key factor for growth of several real life networks, they often serve as good link predictors (Sarkar, Chakrabarti, and Moore 2011). (4) The random walk based embeddings,
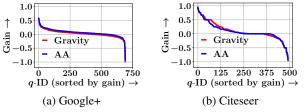

(a) Google+ (b) Citeseer

Figure 2: Query-wise wins and losses in terms of $\text{AP}(\text{PERMGNN}) - \text{AP}(\text{baseline})$, the gain (above x-axis) or loss (below x-axis) of AP of PERMGNN with respect to competitive baselines. Queries Q are sorted by decreasing gain of PERMGNN along the x-axis.
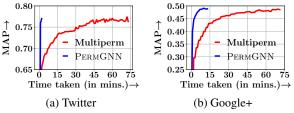

(a) Twitter (b) Google+

Figure 3: Validation MAP against training epochs for Twitter and Google+. PERMGNN converges faster than MultiPerm.

viz. Node2Vec and DeepWalk, show moderate performance. Notably, Node2Vec is the second best performer in Citeseer.

Drill-down. Next, we compare ranking performance at individual query nodes. For each query (node) q, we measure the gain (or loss) of PERMGNN in terms of average precision, *i.e.*, $\text{AP}(\text{PERMGNN}) - \text{AP}(\text{baseline})$ for three competitive baselines, across Google+ and Citeseer datasets. From Figure 2, we observe that, for Google+ and Citeseer respectively, PERMGNN matches or exceeds the baselines for 60% and 70% of the queries.

5.3 PERMGNN vs. Sampling Permutations

Next, we address research question **RQ2** by establishing the utility of PERMGNN against its natural alternative **Multi-Perm**, in which a node embedding is computed by averaging permutation-sensitive representations over several sampled permutations. Figure 3 shows that PERMGNN is $>15\times$ and $>4.5\times$ faster than the permutation averaging based method for Twitter and Google+ datasets. MultiPerm also occupies significantly larger RAM than PERMGNN.
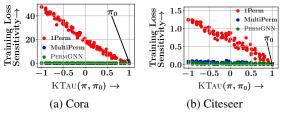
Figure 4: Effect of neighbor order perturbation on training loss. As we move away from the canonical permutation π_0, training loss increases steeply for 1Perm, but remains roughly stable for MultiPerm and PERMGNN.
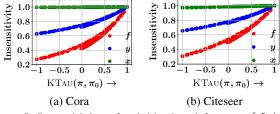

Figure 5: Insensitivity of neighborhood features $\{f_v \mid v \in \mathrm{nbr}(u)\}$ LSTM output $\{\mathbf{y}\}$ and the resultant node embeddings $\mathbf{x}_u$ with respect to neighbor order permutations.

5.4 Permutation Invariance of PERMGNN

Here, we answer the research question **RQ3**. To that end, we first train PERMGNN along with its two immediate alternatives: (i) **1Perm**, where a vanilla LSTM is trained with a single canonical permutation π_0 of the nodes; and, (ii) **Multiperm**, where an LSTM is trained using several sampled permutations of the nodes. Then, given a different permutation π, we compute the node embedding $\mathbf{x}_{u;\pi}$ by feeding the corresponding sequence of neighbors $\pi(\mathrm{nbr}(u))$ (sorted by node IDs of $\mathrm{nbr}(u)$ assigned by π), as an input to the trained models. Finally, we use these embeddings for LP and measure the relative change in training loss. Figure 4 shows a plot of $(\mathrm{LOSS}(\pi) - \mathrm{LOSS}(\pi_0))/\mathrm{LOSS}(\pi_0)$ against the correlation between π and the canonical order π_0, measured in terms of Kendall's τ, $\mathrm{KTAU}(\pi, \pi_0)$. It reveals that 1Perm suffers a significant rise in training loss when the input node order π substantially differs from the canonical order π_0, i.e., $\mathrm{KTAU}(\pi, \pi_0)$ is low. Both Multiperm and PERMGNN turns out to be permutation-insensitive across a wide range of node orderings.

To probe this phenomenon, we instrument the stability of $f, \mathbf{y}, \mathbf{x}$ to different permutations. Specifically, we define insensitivity$(\mathbf{z}; \pi, \pi_0) = \sum_{u \in V} \mathrm{SIM}(\mathbf{z}_{u;\pi}, \mathbf{z}_{u;\pi_0})/|V|$ for any vector or sequence $\mathbf{z}$. We compute insensitivity of the input sequence $\{f_v : v \in \mathrm{nbr}(u)\}$, the intermediate LSTM output $\{\mathbf{y}\}$ and the final embedding $\mathbf{x}_u$ with respect to different permutations π. Figure 5 summarizes the results, and shows that as information flows through PERMGNN stages, from input feature sequence to the final embeddings, the insensitivity of the underlying signals increases. Thus, our adversarial training smoothly turns permutation-sensitive input sequences into permutation invariant node embeddings, without any explicit symmetric aggregator.
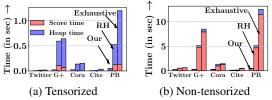

Figure 6: Running time for our LSH based scalable prediction, random-hyperplane based LSH method, exhaustive comparison.

5.5 Performance of Hashing Methods

Finally, we address **RQ4** by studying the performance of our LSH method (Section 4.2). Specifically, we compare the time spent in similarity computation and heap operations of our hashing method against random hyperplane based hashing (Section 4.1), compared to exhaustive computation of pairwise scores (as a slow but "relatively perfect" baseline). Since vectorized similarity computation inside Torch may be faster than numpy, we provide results on both implementations. Figure 6 summarizes results in terms of running time. It shows that: (1) hashing using C_ψ leads to considerable savings in reporting top-K node-pairs with respect to both random hyperplane based hashing and exhaustive enumeration, and (2) the gains increase with increasing graph sizes (from Google+ to PB). Because LSH-based top-K retrieval may discard relevant nodes after K, it is more appropriate to study ranking degradation in terms of decrease in NDCG (rather than MAP). Suppose we insist that NDCG be at least 85, 90, or 95% of exhaustive NDCG. How selective is a hashing strategy, in terms of the factor of query speedup (because of buckets pruned in Algorithm 1)? Table 2 shows that our hashing method provides better pruning than random hyperplane for a given level of NDCG degradation.

| | Minimum NDCG as % of exhaustive NDCG | | | | | |
| | 85% | | 90% | | 95% | |
	Twitter	Google+	Twitter	Google+	Twitter	Google+
Our Hashing	6.67	12.5	6.67	10	6.25	5.5
RH	1.78	3.45	1.78	3.45	1.78	3.45

Table 2: Speedup achieved by different hashing methods under various permitted NDCG degradation limits.

6 Conclusion

We presented PERMGNN, a novel LP formulation that combines a recurrent, order-sensitive graph neighbor aggregator with an adversarial generator of neighbor permutations. PERMGNN achieves LP accuracy comparable to or better than sampling a number of permutations by brute force, and is faster to train. PERMGNN is also superior to a number of LP baselines. In addition, we formulate an optimization to map PERMGNN's node embeddings to a suitable locality-sensitive hash, which greatly speeds up reporting of the most likely edges. It would be interesting to extend PERMGNN to other downstream network analyses, e.g., node classification, community detection, or knowledge graph completion.

Acknowledgements

References

Ackland, R.; et al. 2005. Mapping the US political blogosphere: Are conservative bloggers more prominent? In *BlogTalk Downunder 2005 Conference, Sydney*. BlogTalk Downunder 2005 Conference, Sydney.

Adamic, L. A.; and Adar, E. 2003. Friends and neighbors on the Web. *Social Networks* 25(3): 211 – 230. ISSN 0378-8733. doi:http://dx.doi.org/10.1016/S0378-8733(03)00009-1. URL http://pkudlib.org/qmeiCourse/files/FriendsAndNeighbors.pdf.

Backstrom, L.; and Leskovec, J. 2011. Supervised random walks: predicting and recommending links in social networks. In *WSDM Conference*, 635–644. URL http://cs.stanford.edu/people/jure/pubs/linkpred-wsdm11.pdf.

Bloem-Reddy, B.; and Teh, Y. W. 2019. Probabilistic symmetry and invariant neural networks. *arXiv preprint arXiv:1901.06082* .

Borovkova, S.; and Tsiamas, I. 2019. An ensemble of LSTM neural networks for high-frequency stock market classification. *Journal of Forecasting* 38(6): 600–619.

Charikar, M. S. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*, 380–388. URL https://dl.acm.org/doi/pdf/10.1145/509907.509965.

Cohen-Karlik, E.; David, A. B.; and Globerson, A. 2020. Regularizing Towards Permutation Invariance in Recurrent Models. In *NeurIPS*. URL https://arxiv.org/abs/2010.13055.

Cuturi, M. 2013. Sinkhorn distances: Lightspeed computation of optimal transport. In *NeurIPS*, 2292–2300. URL https://papers.nips.cc/paper/4927-sinkhorn-distances-lightspeed-computation-of-optimal-transport.pdf.

Garg, V. K.; Jegelka, S.; and Jaakkola, T. 2020. Generalization and representational limits of graph neural networks. *arXiv preprint arXiv:2002.06157* .

Getoor, L. 2005. Link-based classification. In *Advanced methods for knowledge discovery from complex data*, 189–207. Springer.

Gionis, A.; Indyk, P.; and Motwani, R. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB Conference*, 518–529. See http://citeseer.nj.nec.com/gionis97similarity.html.

Grover, A.; and Leskovec, J. 2016. node2vec: Scalable feature learning for networks. In *SIGKDD*.

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, 1024–1034.

Hochreiter, S.; and Schmidhuber, J. 1997. Long Short-Term Memory. *Neural Computation* 9(8): 1735–1780. URL https://www.mitpressjournals.org/doi/pdfplus/10.1162/neco.1997.9.8.1735.

Joachims, T. 2005. A support vector method for multivariate performance measures. In *ICML*, 377–384. ISBN 1-59593-180-5. doi:http://doi.acm.org/10.1145/1102351.1102399. URL http://www.machinelearning.org/proceedings/icml2005/papers/048_ASupport_Joachims.pdf.

Katz, B. 1997. From Sentence Processing to Information Access on the World Wide Web. In *AAAI Spring Symposium on Natural Language Processing for the World Wide Web*, 77–94. Stanford CA: Stanford University. See http://www.ai.mit.edu/people/boris/webaccess/.

Kipf, T. N.; and Welling, M. 2016a. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* .

Kipf, T. N.; and Welling, M. 2016b. Variational graph autoencoders. *arXiv preprint arXiv:1611.07308* .

Kulis, B.; and Darrell, T. 2009. Learning to hash with binary reconstructive embeddings. In *NeurIPS*, 1042–1050. URL http://papers.nips.cc/paper/3667-learning-to-hash-with-binary-reconstructive-embeddings.pdf.

Lee, J.; Lee, Y.; Kim, J.; Kosiorek, A.; Choi, S.; and Teh, Y. W. 2019. Set transformer: A framework for attention-based permutation-invariant neural networks. In *ICML*.

Leskovec, J.; Chakrabarti, D.; Kleinberg, J.; Faloutsos, C.; and Ghahramani, Z. 2010. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research* 11(Feb): 985–1042.

Leskovec, J.; and Mcauley, J. J. 2012. Learning to discover social circles in ego networks. In *NeuIPS*.

Liben-Nowell, D.; and Kleinberg, J. 2007. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology* 58(7): 1019–1031. ISSN 1532-2890. doi:10.1002/asi.20591. URL https://onlinelibrary.wiley.com/doi/full/10.1002/asi.20591.

Lichtenwalter, R. N.; Lussier, J. T.; and Chawla, N. V. 2010. New perspectives and methods in link prediction. In *SIGKDD Conference*, 243–252. Washington, DC, USA: ACM. ISBN 978-1-4503-0055-1. doi:10.1145/1835804.1835837. URL http://users.cs.fiu.edu/~lzhen001/activities/KDD_USB_key_2010/docs/p243.pdf.

Liu, W.; Wang, J.; Ji, R.; Jiang, Y.-G.; and Chang, S.-F. 2012. Supervised hashing with kernels. In *IEEE CVPR*, 2074–2081. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6247912.

Mena, G.; Belanger, D.; Linderman, S.; and Snoek, J. 2018. Learning latent permutations with gumbel-sinkhorn networks. *arXiv preprint arXiv:1802.08665* URL https://arxiv.org/pdf/1802.08665.pdf.

Murphy, R. L.; Srinivasan, B.; Rao, V.; and Ribeiro, B. 2019a. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. *ICLR* URL https://arxiv.org/pdf/1811.01900.

Murphy, R. L.; Srinivasan, B.; Rao, V.; and Ribeiro, B. 2019b. Relational pooling for graph representations. *arXiv preprint arXiv:1903.02541* .

NT, H.; and Maehara, T. 2019. Revisiting graph neural networks: All we have is low-pass filters. *arXiv preprint arXiv:1905.09550* .

Pabbaraju, C.; and Jain, P. 2019. Learning Functions over Sets via Permutation Adversarial Networks. *arXiv preprint arXiv:1907.05638* URL https://arxiv.org/pdf/1907.05638.

Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. Deepwalk: Online learning of social representations. In *KDD*, 701–710.

Qi, C. R.; Su, H.; Mo, K.; and Guibas, L. J. 2017. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 652–660.

Ravanbakhsh, S.; Schneider, J.; and Poczos, B. 2016. Deep learning with sets and point clouds. *arXiv preprint arXiv:1611.04500* .

Roy, I.; De, A.; and Chakrabarti, S. 2021. Adversarial Permutation Guided Node representations for Link Prediction. In *arXiv:2012.08974*.

Salha, G.; Limnios, S.; Hennequin, R.; Tran, V.-A.; and Vazirgiannis, M. 2019. Gravity-Inspired Graph Autoencoders for Directed Link Prediction. In *CIKM*, 589–598. URL https://doi.org/10.1145/3357384.3358023.

Sarkar, P.; Chakrabarti, D.; and Moore, A. W. 2011. Theoretical justification of popular link prediction heuristics. In *COLT*.

Schlichtkrull, M.; Kipf, T. N.; Bloem, P.; Van Den Berg, R.; Titov, I.; and Welling, M. 2018. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, 593–607. URL https://arxiv.org/pdf/1703.06103.

Sen, P.; Namata, G.; Bilgic, M.; Getoor, L.; Galligher, B.; and Eliassi-Rad, T. 2008. Collective classification in network data. *AI magazine* 29(3): 93–93.

Shi, Y.; Oliva, J.; and Niethammer, M. 2020. Deep Message Passing on Sets. In *AAAI*, 5750–5757.

Sinkhorn, R. 1967. Diagonal equivalence to matrices with prescribed row and column sums. *The American Mathematical Monthly* 74(4): 402–405. URL https://www.jstor.org/stable/pdf/2314570.pdf.

Skianis, K.; Nikolentzos, G.; Limnios, S.; and Vazirgiannis, M. 2020. Rep the set: Neural networks for learning set representations. In *International Conference on Artificial Intelligence and Statistics*, 1410–1420. PMLR.

Stelzner, K.; Kersting, K.; and Kosiorek, A. R. 2020. Generative Adversarial Set Transformers. In *Workshop on Object-Oriented Learning at ICML 2020*. URL https://www.ml.informatik.tu-darmstadt.de/papers/stelzner2020ood_gast.pdf.

Tang, J.; Qu, M.; Wang, M.; Zhang, M.; Yan, J.; and Mei, Q. 2015. LINE: Large-scale information network embedding. In *WWW Conference*, 1067–1077.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* .

Wagstaff, E.; Fuchs, F. B.; Engelcke, M.; Posner, I.; and Osborne, M. 2019. On the limitations of representing functions on sets. *arXiv preprint arXiv:1901.09006* .

Wang, Z.; Ren, Z.; He, C.; Zhang, P.; and Hu, Y. 2019. Robust Embedding with Multi-Level Structures for Link Prediction. In *IJCAI*, 5240–5246. URL https://www.ijcai.org/Proceedings/2019/0728.pdf.

Weiss, Y.; Torralba, A.; and Fergus, R. 2009. Spectral hashing. In *NeurIPS*, 1753–1760. URL https://papers.nips.cc/paper/3383-spectral-hashing.pdf.

Wu, F.; Zhang, T.; Souza Jr, A. H. d.; Fifty, C.; Yu, T.; and Weinberger, K. Q. 2019. Simplifying graph convolutional networks. *arXiv preprint arXiv:1902.07153* .

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2018a. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* .

Xu, K.; Li, C.; Tian, Y.; Sonobe, T.; Kawarabayashi, K.-i.; and Jegelka, S. 2018b. Representation learning on graphs with jumping knowledge networks. *arXiv preprint arXiv:1806.03536* .

Yadati, N.; Nitin, V.; Nimishakavi, M.; Yadav, P.; Louis, A.; and Talukdar, P. 2018. Link prediction in hypergraphs using graph convolutional networks. Manuscript. URL https://openreview.net/forum?id=ryeaZhRqFm.

You, J.; Ying, R.; and Leskovec, J. 2019. Position-aware graph neural networks. *arXiv preprint arXiv:1906.04817* .

Zaheer, M.; Kottur, S.; Ravanbakhsh, S.; Poczos, B.; Salakhutdinov, R. R.; and Smola, A. J. 2017. Deep sets. In *Advances in neural information processing systems*, 3391–3401.

Zhang, M.; and Chen, Y. 2018. Link prediction based on graph neural networks. In *NeurIPS*.