



PDF Download  
3696410.3714938.pdf  
16 January 2026  
Total Citations: 0  
Total Downloads: 3489

 Latest updates: <https://dl.acm.org/doi/10.1145/3696410.3714938>

RESEARCH-ARTICLE

## Node2binary: Compact Graph Node Embeddings using Binary Vectors

NILOY TALUKDER, Indiana University Indianapolis, Indianapolis, IN, United States

CROIX GYUREK, University of Waterloo, Waterloo, ON, Canada

MOHAMMAD MAHFUZ AL HASAN, Indiana University Indianapolis, Indianapolis, IN, United States

Open Access Support provided by:

University of Waterloo

Indiana University Indianapolis

Published: 22 April 2025

[Citation in BibTeX format](#)

WWW '25: The ACM Web Conference 2025

April 28 - May 2, 2025  
Sydney NSW, Australia

Conference Sponsors:  
SIGWEB

# NODE2BINARY: Compact Graph Node Embeddings using Binary Vectors

Niloy Talukder  
Indiana University Indianapolis  
Indianapolis, IN, USA  
ntalukde@iu.edu

Croix Gyurek  
University of Waterloo  
Waterloo, ON, CA  
cgyurek@uwaterloo.ca

Mohammad Al Hasan  
Indiana University Indianapolis  
Indianapolis, IN, USA  
alhasan@iu.edu

## Abstract

With the adoption of deep learning models to low-power, small-memory edge devices, energy consumption and storage usage of such models have become a key concern. The problem exacerbates even further with ever-growing data and equally-matched bulkier models. This concern is particularly pronounced for graph data due to its quadratic storage, irregular (non-grid) geometry, and very large size. Typical graph data, such as road networks, infrastructure networks, and social networks, easily exceeds millions of nodes, and several gigabytes of storage is needed just to store the node embedding vectors, let alone the model parameters. In recent years, the memory issue has been addressed by moving away from memory-intensive double precision floating-point arithmetic towards single-precision or even half-precision, often by trading-off marginally small performance. Along this effort, we propose NODE2BINARY, which embeds graph nodes in as few as 128 binary bits, thereby reducing the memory footprint of vertex embedding vectors by several orders of magnitude. NODE2BINARY leverages a fast community detection algorithm to convert the given graph into a hierarchical partition tree and then find embeddings of graph vertices in binary space by solving a combinatorial optimization (CO) task over the tree edges. CO is NP-hard, but NODE2BINARY uses an innovative combination of discrete gradient descent and randomization to solve this task effectively and efficiently. Extensive experiments over four real-world graphs show that NODE2BINARY achieves competitive performance compared to the state-of-the-art graph embedding methods in both node classification and link prediction tasks.

## CCS Concepts

• **Computing methodologies** → **Knowledge representation and reasoning**; • **Information systems** → **Social networks**; • **Mathematics of computing** → *Combinatorial optimization*.

## Keywords

Binary Space Embedding, Graph Embedding, Discrete Gradient Descent, Randomized Algorithm

## ACM Reference Format:

Niloy Talukder, Croix Gyurek, and Mohammad Al Hasan. 2025. NODE2BINARY: Compact Graph Node Embeddings using Binary Vectors. In *Proceedings of the ACM Web Conference 2025 (WWW '25)*, April 28-May 2, 2025, Sydney,



This work is licensed under a Creative Commons Attribution International 4.0 License.

WWW '25, April 28-May 2, 2025, Sydney, NSW, Australia  
© 2025 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-1274-6/2025/04  
<https://doi.org/10.1145/3696410.3714938>

NSW, Australia. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3696410.3714938>

## 1 Introduction

In today's connected world, networks have become a formidable data structure for representing many complex systems, such as social networks, biological networks, transportation networks, information networks, and so on. The information captured in such networks holds immense value, both commercial and strategic, so developing knowledge discovery models for analyzing these networks have received increasing attention by both researchers and practitioners. Since most knowledge discovery models are suited for a vector space, a fundamental task for supporting such analysis is to embed a network in a latent space where each vertex is represented by a low-dimensional vector. This task is known as graph representation learning (GRL). A key objective of GRL is to find vertex representation (aka vertex embedding) so that the proximity (based on graph topology, or property) among the vertices are preserved in the embedding space. Such a low-dimensional embedding is very useful in a variety of applications, such as visualization [1, 17], vertex classification [10, 13], and link prediction [4].

Over the years, as real-life graphs grew larger, often exceeding millions of vertices, graph representation learning (GRL) task has become computationally challenging. Lately, the GRL task is mostly solved by graph neural networks or its variants, which are computationally expensive—this also contribute to the challenge. Besides computation, for large graphs the memory footprint of the embedding vectors is also large. For a graph with 10 million vertices, if each embedding vector is 100 dimensions (a typical number), the memory footprint of these vectors is more than 60 GB using double precision real numbers. If single precision real numbers are used, the number reduces to 30 GB, still a formidable amount of storage. As machine learning tasks are now being solved in the edge devices, which are battery powered with limited memory storage; processing such a large dataset is difficult due to memory bottleneck. If memory is not an issue, transferring such large amount of data to and from a server would drain the battery of such devices. Hence, there is a surge of interest in machine learning research to develop embedding methods that embed entities in fewer bits, resulting in efficient memory and storage requirements. However, there exist no prominent works for solving GRL that generate compact embedding vectors.

Over the last decade, myriad methods have been proposed for solving the GRL task; prominent methods follow methodologies derived from matrix factorization [1, 9], structural-preserving optimization [12], node-context sampling using random walk [4, 10], and graph neural networks [17]. Some of these methods consider network topology, and others consider both topology and node/edge

attributes. The matrix factorization based methods factor a matrix capturing node similarity: Laplacian matrix, its variants, or specifically designed matrix capturing higher order node proximity are often used for this task [1]. Such methods are costly and usually not scalable for very large networks. To overcome the lack of scalability, random walk based methods [4] are proposed. These methods perform random walk sampling from each node to build a node-sequence capturing the context of the given node. Performance of such models is highly dependent on the quality of sampling strategy that generates the node-context; also, a large number of samples are needed for capturing the node context effectively. Methods based on structure preservation learn node embedding by optimizing a loss function that directly captures first- or higher-order node similarity. But such methods usually do not perform very well [13, 17].

With the popularity of deep learning, various graph neural network architectures have also been proposed for learning node embedding. The earliest among these is GCN (graph convolution network) [7], which uses graph convolution or neighborhood aggregation, a method for enriching node representation by merging its features with those of its neighbors. But GCN is a transductive model thus cannot be generalized to unseen graph nodes. GraphSAGE [6] overcomes this limitation by efficiently using network topology and node attributes to generate embeddings for new nodes. While graph neural networks use latest technologies from the deep learning community for the task of GRL, there are two crucial limitations. First, almost all of the architectures assume the presence of node attributes, which is not the case for general node embedding task. Also, many of the models are trained in a supervised setup, but for a general GRL task, node/edge labels are not typically available.

In this work, we propose NODE2BINARY, a novel node embedding method which embeds vertices of a given graph using binary bit vectors. NODE2BINARY adopts an out-of-the-box idea for binary embedding which stems from a community centric viewpoint of the graph. NODE2BINARY first builds a community partition tree, which is a hierarchical clustering of the vertices of the input graph. Using this community view of the graph, NODE2BINARY imposes constraints over the community partition tree edges to learn meaningful embedding of the graph vertices. Given that the embedding space for NODE2BINARY is binary, the learning task becomes a constrained combinatorial optimization, which NODE2BINARY solves by using a synthesis of discrete gradient descent and randomized local search. Extensive experimental results show the superiority of NODE2BINARY over a number of baseline graph embedding models.

Our contributions can be summarized as follows:

- We propose NODE2BINARY, a novel framework for solving the graph representation learning task in binary space. The proposed method is generic and can be used for learning node embedding vectors for graphs from any domain.
- We propose a novel method to solve combinatorial optimization problem associated to learning node embedding vectors of an input graph in binary space. The method uses discrete gradient descent and an innovative randomized algorithm.
- Experiments on four real-world networks demonstrate that NODE2BINARY maintains competitive performance compared to the state-of-the-art methods, with superior performance at increasingly smaller number of bits.

## 2 Related Work

The main motivation of embedding in binary space is to have a compact node representation which is efficient both computationally and storage-wise. In some existing works, binary vectors are used to embed nodes of a general network, where the primary objective is to perform node similarity search by hashing binary vectors developed through fast sketching methods. The latest among these works is called NODESIG [23] which uses stable random projection for learning binary embedding of vertices. Most of the binary embedding works take inspiration from the random projection based fast nearest neighbor search using locality sensitive hashing [18, 19]. Our work is different from this line of works, which use hashing for embedding nodes in binary space. A recent work [5] embeds entities having is-a relation by using binary bit vectors. However, it is only suitable for embedding a directed network (say, tree or DAG) which satisfies the transitivity property along the edge direction.

Several recent methods [2, 8] use graph coarsening on the original graph, then they apply the methodologies of a traditional graph-embedding method (such as node2vec [4], NetMF [11] etc.) on the coarsest graph to produce binary embeddings. Some simple approaches utilize random projection [3, 22] and spectral graph sparsification techniques [21] to learn scalable network embeddings, but they sacrifice performance in doing so. Our work is different from the above works since we learn vertex embedding vectors by solving combinatorial optimization directly in the binary space.

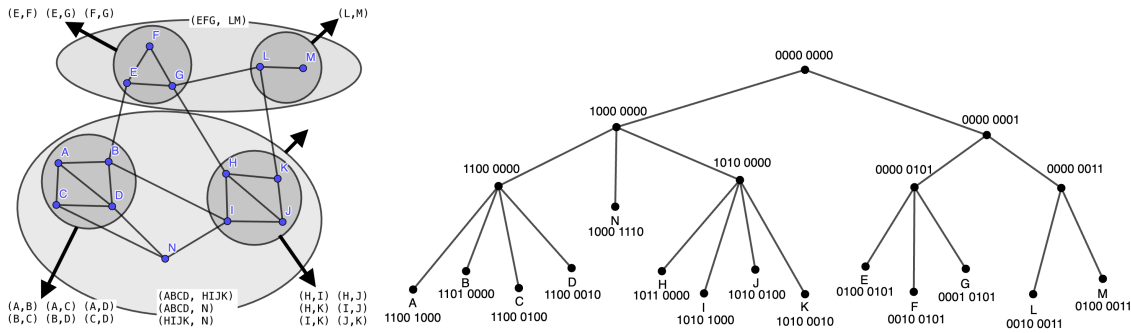
## 3 Methodology

**Notations:**  $G = (V, E)$  is the input graph for which we are soliciting vertex representation vectors. NODE2BINARY builds a hierarchical Partition tree of the vertices  $V$  of  $G$ , denoted with the symbol  $\mathcal{T}_G$ . Italic letters  $a, b$  are used to denote the vertices of the graph  $G$  and also nodes of the tree  $\mathcal{T}_G$ . NODE2BINARY learns embedding vectors for each vertex of  $G$  and also for each node of  $\mathcal{T}_G$ . To represent these embedding vectors we use boldface letters,  $\mathbf{a}$  for node  $a$ . The letter  $d$  is a positive integer number denoting the embedding dimension. We use the letter  $n$  to denote the number of nodes in  $\mathcal{T}_G$ . Greek letters, such as  $\alpha, \beta, \gamma$  are scalars. They are generally reserved for user-defined hyperparameters. The symbol  $\Delta_{\mathbf{x}}$  is used for gradient with respect to a vector variable,  $\mathbf{x}$ .

### 3.1 Problem Formulation and Framework

Graph embedding in binary space is defined as below: Given an undirected graph  $G(V, E)$ , learn an embedding function  $f : V \rightarrow \{0, 1\}^d$  that embeds *similar* vertices in  $V$  close to each other in the embedding space. The embedding space is discrete, as each vertex is mapped to a corner of a unit-length  $d$ -dimensional hypercube. We also assume that  $V$  and  $E$  do not have attributes, so two vertices in  $G$  are considered similar if they appear within the same local context in the graph space. Embedding function  $f$  must preserve this similarity in the embedding space; the similarity between two bit vectors can be defined by Hamming distance.

To solve this embedding task, NODE2BINARY uses an innovative combinatorial optimization framework. To set up this framework, NODE2BINARY first partitions the vertex-set of  $G$  into hierarchical disjoint communities, which can be organized into a hierarchical partition tree. Then the binary embedding vectors of the vertices of



**Figure 1: Community Partition Tree formation from the Original Graph. (Spaces in binary embeddings are for ease of reading.)**

$G$  are learnt by respecting the following two requirements: (1) In a hierarchical (tree-like) partitioning of a network into communities, where the root node of the partitioning is the entire graph, and each of the leaf nodes is a single vertex, embedding vector of a vertex is more similar to that of its parent than any of its ancestors in the tree; and (2) if two vertices are siblings in the partition tree, their embedding vectors are similar. Given that the embedding space is binary, satisfying these requirements leads to a combinatorial optimization problem, which NODE2BINARY solves by using an innovative combination of discrete gradient descent and randomization. We discuss the overall methods of NODE2BINARY in the following subsections.

### 3.2 Creating Community Partition (CP) Tree

We want the embedding vectors of two vertices to have small Hamming distance, if the vertices are well-connected. To realize the notion of well-connectedness between two vertices, we adopt a community-centric view of the graph, where the graph is hierarchically partitioned into smaller communities. This partitioning can also be shown by a tree often known as a *dendrogram* in hierarchical clustering literature. We refer to this tree as *community partition* (CP) tree. For a graph  $G = (V, E)$ , the CP tree is denoted as  $\mathcal{T}_G$ . Nodes in  $\mathcal{T}_G$  represents a community and an edge from a parent to a child in  $\mathcal{T}_G$  represents community to sub-community relation. Note that in this community-hierarchy, a parent community is partitioned into disjoint children community recursively until a community contains only one vertex of  $G$ . Thus the root node of  $\mathcal{T}_G$  represents a community containing all the vertices in  $V$ , and each of the leaf nodes of  $\mathcal{T}_G$  is a vertex of  $G$ . For a toy example, see Figure 1. On the left side of this figure, we show a graph in its hierarchically-partitioned form. The graph is partitioned into two communities, each of which is split into smaller communities. The CP tree of this graph is shown on the right side of the figure.

In such a partitioning if two vertices belong to the same community, they are considered well-connected and they should have similar bit vector representations. We also extend the idea of similarity between vertices to similarity between communities and to achieve that we assign bit vector representation for the communities (the nodes in the CP tree) as well. This is illustrated in Figure 1, where each of the tree node (in the right side of the figure) is assigned a bit vector. Then the embedding task becomes assigning a binary vectors to each of the nodes of the CP tree satisfying certain requirements (which is discussed in the next subsection).

We use Leiden algorithm [16] to construct the CP tree ( $\mathcal{T}_G$ ) from the given graph ( $G$ ). The algorithm is applied recursively to generate CP tree. The leaf nodes correspond to vertices in the graph  $G$ , and internal nodes correspond to communities ordered by inclusion. We summarize the steps of the formation of our CP tree in Algorithm 1.

---

#### Algorithm 1 CreateTree

---

**Require:** Graph  $G = (V, E)$ , number of layers  $\ell$   
**Ensure:** CP tree  $\mathcal{T}_G$ , whose nodes are set of vertices

- 1:  $\mathcal{T}_G \leftarrow$  single-node tree with node-set  $V$
- 2: **if**  $\ell = 1$  or  $|V| = 1$  **then**
- 3:   For each  $v \in V$ , append child  $\{v\}$  to  $\mathcal{T}_G$
- 4: **else**
- 5:    $C \leftarrow$  Leiden( $G$ )
- 6:   **for** community  $C \in \mathcal{C}$  **do**
- 7:      $G_C \leftarrow$  induced subgraph of  $C$
- 8:     Append child CreateTree( $G_C, \ell - 1$ ) to  $\mathcal{T}_G$
- 9:   **end for**
- 10: **end if**

---

### 3.3 Mathematical Framework of NODE2BINARY

To map all the vertices which belong to a community to similar bit vectors, we ensure that all vertices in a community adhere to certain *behaviors*. To realize this notion, we utilize bit vectors assigned to each community. As stated earlier, each node in the CP tree ( $\mathcal{T}_G$ ) is assigned a bit vector. If we hypothesize that bit vector indices are indicator functions of possessing a given behavior, for a representation vector of a community  $C$ , the ‘1’ indices denote that the members of  $C$  adhere to those behaviors. If  $C$ ’s members are partitioned into sub-communities, sub-community members also possess those behaviors inherited from their parent ( $C$ ). This requirement extends to the entire partition hierarchy; i.e., if a community possesses a behavior (‘1’ bit), its sub-communities will also possess the same behavior, reflected in their bit vector representations. In this way, embedding vector of every node in  $\mathcal{T}_G$  inherits the 1 bits from its parents, and may add 1s of its own. The terminal nodes in  $\mathcal{T}_G$  are vertices of the graph  $G$ , hence the bit vectors of these nodes also follow this requirement. For illustration, see the CP tree in Figure 1. In this tree, the leaf node  $H$  (which is also a vertex of the input graph) is assigned an embedding vector 10110000; there are 3 ‘1’ bits, of which two (the first and third indices) are inherited from its parent, and one (the fourth index) is its own.

Given a CP tree, the embedding model of NODE2BINARY solves a combinatorial optimization problem for assigning the bit vectors of each node in this tree. If node  $a$  is a child of node  $b$  in  $\mathcal{T}_G$ , then  $f(b)[i] \Rightarrow f(a)[i], \forall i \in [1:d]$ , where  $d$  is the embedding dimension and ' $\Rightarrow$ ' is logical implication. This can be succinctly shown as  $\mathbf{b} \Rightarrow \mathbf{a}$ , where  $\mathbf{b} = f(b)$  and  $\mathbf{a} = f(a)$ .

This requirement leads to a combinatorial feasibility problem over the nodes of the CP tree  $\mathcal{T}_G$  of a given graph  $G$ . The number of constraints equals the number of edges in  $\mathcal{T}_G$  times the dimension, as we need to satisfy the implication condition for each tree edge and for each index of the embedding vectors. A solution to this problem is a  $d$ -dimensional bit vector for each node of the partition tree so that the parent to child implication from above is satisfied.

If  $c$  and  $d$  are children of a node  $b$  in  $\mathcal{T}_G$ , then embedding vectors of  $c$  and  $d$  inherits the '1' bits of  $b$ ; this makes  $c$  and  $d$  similar, but if  $b$  does not have many 1 bits, the constraint makes a very weak similarity. So, we enforce sibling similarity in the objective function to prefer that  $c$  and  $d$  are also similar in bit indices where  $b$  has a '0' bit. However, this requirement is not a constraint as we do not want  $c$  and  $d$  to have identical embedding vectors. So, for each node  $b \in \mathcal{T}_G$ , we take  $k$  random sample of its children and build a loss function to minimize Hamming distance between pairs of children. The full problem looks like

$$\begin{aligned} & \text{find } f : V(\mathcal{T}_G) \rightarrow \{0, 1\}^d \\ & \text{to minimize } \sum_{(c,d) \in S} \text{HammingDistance}(f(c), f(d)) \quad (1) \\ & \text{such that } \forall a, b, \text{ if } a \text{ is a child of } b, \forall i, f(b)[i] \leq f(a)[i] \end{aligned}$$

where  $S$  is the set of sibling pairs.

Solving this type of problem exactly is NP-Hard, so we convert it into an unconstrained optimization problem with a loss function equal to the number of unsatisfied constraints.

### 3.4 Loss Function

Our training goal is twofold. The first is to ensure that *tree relation* (*parent-child*) on nodes (including internal nodes) of  $\mathcal{T}_G$  maps to *bit-wise implication* in the embedding space; the second is to ensure that *sibling nodes* in  $\mathcal{T}_G$  have *similar* embedding vectors as defined by Hamming distance.

Our *loss function* is defined as follows. Let  $H$  be the set of *hierarchy pairs* of nodes  $(a, b)$  from  $\mathcal{T}_G$  satisfying  $b \Rightarrow a$  ( $a$  is child and  $b$  is parent node), and  $S$  be the set of *sibling pairs*  $(c, d)$  in  $\mathcal{T}_G$ . We have two goals: ensure that (1) nodes in the *hierarchy* pass their "1" bits down to their children, and (2) two *sibling* nodes have embeddings with small Hamming distance between them. Condition (1) is equivalent to ensuring that  $a$ 's embedding vector ( $\mathbf{a}$ ) inherits all 1 bits from  $b$ 's embedding vector ( $\mathbf{b}$ ), i.e. there is no  $j$  such that  $\mathbf{a}_j = 0$  and  $\mathbf{b}_j = 1$ . Any violation of this condition constitutes a loss value of 1. If  $H$  is the set of all pairs  $(a, b)$  such that  $b$  is an ancestor of  $a$  (we call  $(a, b)$  a positive pair), positive loss is simply the count of all such violations in all bitvector indices over the pairs in  $H$ . We also randomly generate a collection of negative pairs, and for those pairs, the loss value is simply the count of pairs for which the implication holds (which should not hold as they are negative pairs). Finally for sibling pairs  $(c, d)$ , the loss value is simply the sum of Hamming distance over all such pairs. We linearly add these

three losses to generate the loss value as shown below:

$$\text{Loss} = \alpha \text{Loss}^{\text{hier},+} + \beta \text{Loss}^{\text{hier},-} + \gamma \text{Loss}^{\text{sib}} \quad (2)$$

$$\text{Loss}^{\text{hier},+} = \sum_{(a,b) \in H} \sum_{j=1}^d (1 \text{ if } (\mathbf{a}_j, \mathbf{b}_j) = (0, 1) \text{ else } 0) \quad (3)$$

$$\text{Loss}^{\text{hier},-} = |W|, \text{ where } W = \{(a, b) \notin H : \mathbf{a}_j \geq \mathbf{b}_j, \forall j\} \quad (4)$$

$$\text{Loss}^{\text{sib}} = \sum_{(c,d) \in S} \text{HammingDistance}(\mathbf{c}, \mathbf{d}) \quad (5)$$

### 3.5 Training Algorithm

Once the CP tree is formed and the *sibling pairs* are collected, we train our model to learn binary embedding vectors of all nodes in CP tree by minimizing the loss value using gradient descent like local optimization. Since the model parameters (node embedding vectors) are binary vector, gradient of loss with respect to the model parameters cannot be obtained by taking derivatives similar to real space; we instead approximate gradient by finite difference method. If  $f(\mathbf{x})$  is a function whose inputs are binary, that is,  $\mathbf{x} \in \{0, 1\}^k$ , then the *discrete gradient* of  $f$  with respect to  $\mathbf{x}$  using finite difference can be defined as

$$(\Delta_{\mathbf{x}} f(\mathbf{x}))_j = f(\mathbf{x} \text{ with bit } j \text{ flipped}) - f(\mathbf{x}) \quad (6)$$

Note that, by using above formula,  $\Delta_{\mathbf{x}} f$  is positive when *flipping*  $\mathbf{x}$  increases  $f$ . Since our loss functions  $\text{Loss}^{\text{hier},+}$ ,  $\text{Loss}^{\text{hier},-}$ , and  $\text{Loss}^{\text{sib}}$  are two-argument functions, the finite difference based gradient computation with respect to one argument involves the consideration of bit pattern in the other argument. These scenarios are explained in the Appendix A.1, in Table 4 for  $\text{Loss}^{\text{hier},+}$ , Table 5 for  $\text{Loss}^{\text{hier},-}$ , and Table 6 for  $\text{Loss}^{\text{sib}}$ . By using the bit pattern in Column 3 and 4 of Table 5 for a positive pair  $(a, b)$ , where  $a$  is a child and  $b$  is a parent node, the boolean functions for gradient of  $\text{Loss}^{\text{hier},+}$  with respect to parameters  $\mathbf{a}$ , and  $\mathbf{b}$  are as follow:

$$\Delta_{\mathbf{a}} \text{Loss}^{\text{hier},+} = \sum_{b:(a,b) \in H} \mathbf{b} * (2\mathbf{a} - 1) \quad (7)$$

$$\Delta_{\mathbf{b}} \text{Loss}^{\text{hier},+} = \sum_{a:(a,b) \in H} (\mathbf{a} - 1) * (2\mathbf{b} - 1) \quad (8)$$

Note that the loss value is computed over all positive pairs  $(a, b)$  in  $H$ . The first of the above equations ensures the property that  $\Delta_{\mathbf{a}} \text{Loss}^{\text{hier},+}(\mathbf{a}, \mathbf{b})$  is  $-1$  in position  $j$  iff  $(\mathbf{a}_j, \mathbf{b}_j) = (0, 1)$ . This encourages the model to flip  $\mathbf{a}_j$ , thus restoring the partial order relation. Contrariwise, if  $(\mathbf{a}_j, \mathbf{b}_j) = (1, 1)$ , the gradient of  $\mathbf{a}_j$  is set to  $+1$ , which discourages  $\mathbf{a}_j$  from being flipped. A similar logic applies to the  $\mathbf{b}$  gradient. We provide a detailed justification in Appendix A.1.

For negative samples, i.e. pairs  $(a', b')$  where  $a'$  is *not* a child of  $b'$ , the situation is more complicated because our loss function is the *number* of violated constraints, i.e. the number of pairs  $(a', b')$  where  $\mathbf{b} \Rightarrow \mathbf{a}$  in *all* bit positions. That is, we want  $(0, 1)$  pairs in  $(a', b')$ 's embeddings. The "all" in the loss function requires a more involved gradient computation. For each pair  $(a', b')$ , let  $G(a', b')$  be the number of "good"  $(0, 1)$  pairs; we aim to ensure  $G(a', b') \geq 1$

for all  $(a', b')$ . Then we can derive

$$\Delta_{\mathbf{a}'} \text{Loss}^{\text{hier}, -} = \sum_{b': (a', b') \in H^-} \begin{cases} -\mathbf{a}' * \mathbf{b}' & G(a', b') = 0 \\ \mathbf{b}' * (1 - \mathbf{a}') & G(a', b') = 1 \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

$$\Delta_{\mathbf{b}'} \text{Loss}^{\text{hier}, -} = \sum_{a': (a', b') \in H^-} \begin{cases} -(1 - \mathbf{a}') * (1 - \mathbf{b}') & G(a', b') = 0 \\ \mathbf{b}' * (1 - \mathbf{a}') & G(a', b') = 1 \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

Finally, the Hamming-loss gradient is the same for both inputs:

$$\Delta_{\mathbf{a} \text{ or } \mathbf{b}} \text{Loss}^{\text{sib}} = 1 - 2(\mathbf{a} \oplus \mathbf{b}) \quad (11)$$

The reader can verify that this vector is +1 in index positions where  $\mathbf{a}, \mathbf{b}$  agree, and -1 where they disagree. This encourages nodes in the same community to have similar embeddings.

With discrete space, we cannot adjust the embeddings by a “small amount” as in traditional gradient descent. Instead, we compute a probability for inverting bit  $a_j$  based on the negation of overall gradient  $\Delta(\mathbf{x}_j)$ ,  $\text{FlipProb}(\Delta(\mathbf{x}_j))$ , defined as

$$\text{FlipProb}(\Delta(\mathbf{x}_j)) = \frac{1}{2} \tanh(-2(r_\ell \Delta(\mathbf{x}_j) + b_\ell)) \quad (12)$$

where the *learning rate*  $r_\ell$  controls the flipping rate and the *bias*  $b_\ell$  allows flipping bits with zero gradient, to avoid local maxima. This function ensures that at most half of all bits are flipped each iteration, which prevents the model from oscillating.

NODE2BINARY’s algorithm is summarized in Algorithm 2. We start by building the tree,  $\mathcal{T}_G$ , as in Algorithm 1 and generating the sibling pairs  $S$  and hierarchy pairs  $H$  by considering all tree edges and their transitive closure. We initialize embedding vectors of the tree nodes to the zero matrix. On each training iteration, we compute the discrete gradient matrix  $\Delta \in \mathbb{Z}^{n \times d}$ , pass  $\Delta$  into  $\text{FlipProb}$  to get a probability matrix  $P$ , and flip each bit  $a_j$  with probability  $P(a, j)$ . The symbol  $\text{Emb}$  denotes the learned embedding function  $\text{Emb} : \mathcal{T}_G \rightarrow \{0, 1\}^d$ .

---

#### Algorithm 2 Training Algorithm

---

**Require:** Graph  $G = (V, E)$ , dimension  $d$ , iterations  $t$ , parameters  $(\alpha, \beta, \gamma, r_\ell, b_\ell)$

- 1:  $\mathcal{T}_G \leftarrow \text{CreateTree}(G)$
  - 2:  $S \leftarrow \text{CreateSiblings}(\mathcal{T}_G)$
  - 3:  $H \leftarrow \{(a, b) : a, b \in \mathcal{T}_G, b \Rightarrow a\}$      $\triangleright (a, b)$  are edges of  $\mathcal{T}_G$ ’s transitive closure
  - 4:  $\text{Emb}(a) \leftarrow (0, \dots, 0)$  for all  $a \in \mathcal{T}_G$
  - 5: **for**  $\tau = 1$  to  $t$  **do**
  - 6:     $H^- \leftarrow$  negative samples of  $H$
  - 7:     $\Delta \leftarrow \alpha \Delta^{\text{hier}, +}(H, \text{Emb}) + \beta \Delta^{\text{hier}, -}(H^-, \text{Emb}) + \gamma \Delta^{\text{sib}}(S, \text{Emb})$
  - 8:     $P \leftarrow \text{FlipProb}(\Delta)$
  - 9:    **for**  $a \in \mathcal{T}_G$  **do**
  - 10:      $\text{Emb}(a) \leftarrow \text{Emb}(a) \oplus (\text{random}(0, 1)^d < P(a, \cdot))$
  - 11:    **end for**
  - 12: **end for**
- 

**Table 1: Statistics of the datasets used in the experiments.**

Dataset	PPI	DBLP	Blog	YouTube (Small / Large)
#V	3,890	13,326	10,312	31,703 / 1,138,499
#E	38,705	34,281	333,983	276,126 / 2,990,443
#L	50	2	39	47

### 3.6 Time Complexity and Space Efficiency

Our algorithm is very efficient in both the time and space dimensions. Algorithm 1 can be seen to take  $O(\ell|E|)$  time. It outputs a tree with at most  $\ell|V|$  nodes. Sibling sampling selects  $k$  random siblings of each tree node, which is  $O(k\ell|V|)$ . Finally, the training algorithm is  $O(t(|H| + |S|))$ ;  $|H|$  is bounded by  $\ell^2|V|$ , as each of the  $\ell|V|$  nodes have at most  $\ell$  ancestors, and  $|S| \in O(k\ell|V|)$  from before. Thus, node2binary runs in linear time.

For space complexity, the storage needed for  $n$  nodes each having  $d$ -bit vectors is  $\frac{dn}{8}$  bytes, while double- and single-precision floating point based numbers take  $8dn$  and  $4dn$  bytes, respectively. Thus, node2binary embeddings are 32 or 64 times more space efficient at the same dimension than most competing models.

## 4 Experiments and Results

We evaluate NODE2BINARY’s embedding quality based on two standard graph representation learning tasks: multi-label classification on vertices and link prediction on edges. We organize this section as follows: In §4.1, we provide details of the datasets we used. In §4.2, we discuss the baseline methods we compare our method with, and the evaluation metrics we used. In §4.3 we provide details of our experimental setup. In §4.4 and §4.5, we further discuss the setup and results from our multi-label node classification and link prediction experiments, respectively. For the rest of the section, we discuss the robustness of our method. In §4.6, we compare our method’s scalability with state-of-the-art methods. In §4.7, we discuss parameter sensitivity. In Appendix A.3 we provide the convergence plots and in A.4 we provide an ablation study to show how different components of our objective function affect our performance. Our code is publicly available at <https://github.com/ntalukde/node2binary>.

### 4.1 Datasets

For our experiments we have chosen four moderate to large size real world labeled graphs drawn from biology, collaborations, and social networks which are largely used by our competitors. **PPI** [4] is a subgraph of the protein-protein interaction network for *Homo sapiens*. The labels of the nodes represent its gene sets and also biological states. **DBLP** [20] is a collaboration network which captures the co-authorship of authors. The labels of a node in the co-author graph represents publication venues of the respective author. **BlogCatalog** [14] is a social network of bloggers where labels indicate the topics of interest by the corresponding blogger. Finally, **YouTube** [15] is a social network of users; labels refer to list of subscriptions by the user. We consider a subset of YouTube dataset for node classification and link prediction and use the full dataset with millions of nodes for the scalability experiment. The statistics of the datasets are summarized in Table 1. All our graphs are unweighted and undirected.

**Table 2: Multi-label Node Classification task, according to number of *bits* in embeddings**

		<b>Models</b>							
		<b>DeepWalk</b>	<b>node2vec</b>	<b>LINE</b>	<b>HOPE</b>	<b>NodeSketch</b>	<b>GraphSAGE</b>	<b>NODESIG</b>	<b>node2binary</b>
<b>bits</b>	<b>Node Classification (Metric: F1 %)</b>								
<b>YouTube</b>									
128	6.01	5.94	6.0	5.92	6.69	6.54	<b>8.97</b>	<u>8.79</u>	(8.69 ± 0.07)
256	6.17	6.17	6.23	6.02	6.85	6.52	<u>9.09</u>	<b>9.27</b>	(9.22 ± 0.06)
512	6.44	6.39	6.52	6.26	7.12	6.54	<u>8.75</u>	<b>9.51</b>	(9.44 ± 0.06)
1024	7.0	7.38	6.63	6.55	7.45	6.58	<u>8.38</u>	<b>9.51</b>	(9.47 ± 0.03)
2048	7.92	8.18	7.38	6.77	8.27	6.55	<u>8.87</u>	<b>9.5</b>	(9.48 ± 0.02)
4096	8.89	8.86	8.1	7.89	8.48	6.55	<u>9.5</u>	<b>9.67</b>	(9.52 ± 0.09)
8192	9.4	9.35	9.3	8.42	8.85	6.55	<b>9.93</b>	<u>9.56</u>	(9.45 ± 0.09)
<b>DBLP</b>									
128	<b>82.3</b>	73.43	45.77	39.03	41.89	39.26	73.38	<u>76.44</u>	(73.68 ± 2)
256	<b>88.52</b>	<u>86.39</u>	48.98	40.66	44.13	39.26	79.14	82.87	(81.91 ± 0.8)
512	<u>89.16</u>	87.31	56.95	51.21	47.5	48.73	84.05	<b>94.38</b>	(94.25 ± 0.1)
1024	<u>89.19</u>	88.45	55.20	51.63	48.61	48.69	87.32	<b>95.71</b>	(95.46 ± 0.2)
2048	89.87	89.47	60.42	62.36	53.01	48.73	<u>93.47</u>	<b>95.82</b>	(95.5 ± 0.2)
4096	90.26	89.38	72.08	70.17	55.95	48.7	<b>96.62</b>	<u>94.79</u>	(94.77 ± 0.01)
8192	93.73	89.93	79.08	75.11	59.45	48.7	<b>97.6</b>	<u>94.81</u>	(94.79 ± 0.02)
<b>Blogcatalog</b>									
128	3.12	3.27	3.28	2.83	2.83	2.96	<u>11.85</u>	<b>12.1</b>	(11.83 ± 0.2)
256	6.27	6.98	3.46	2.90	3.81	2.57	<b>15.29</b>	<u>14.64</u>	(14.38 ± 0.1)
512	10.89	12.23	3.90	3.33	4.45	2.84	<b>16.1</b>	<u>16.08</u>	(15.61 ± 0.4)
1024	17.19	<u>18.49</u>	5.74	3.71	4.98	2.56	<b>18.73</b>	15.49	(15.49 ± 0.3)
2048	<u>22.34</u>	<b>23.43</b>	10.51	7.05	6.73	2.56	22.06	15.81	(15.5 ± 0.3)
4096	<u>24.69</u>	<b>26.07</b>	22.42	11.42	8.23	2.56	22.03	13.95	(13.86 ± 0.06)
8192	<u>25.49</u>	<b>26.18</b>	25.41	13.95	9.63	2.56	24.27	13.95	(13.93 ± 0.02)
<b>PPI</b>									
128	4.19	6.07	3.73	3.92	3.48	1.57	<u>13.44</u>	<b>16.89</b>	(16.55 ± 0.3)
256	8.73	9.73	3.59	5.85	4.51	1.57	<u>13.21</u>	<b>17.93</b>	(17.48 ± 0.4)
512	13.03	13.25	3.87	7.03	5.17	1.55	<u>13.74</u>	<b>17.86</b>	(17.21 ± 0.4)
1024	16.34	<u>17.21</u>	5.28	9.14	5.90	2.76	16.06	<b>17.73</b>	(17.13 ± 0.4)
2048	<u>18.04</u>	<b>18.16</b>	7.61	11.26	6.99	1.55	17.96	17.07	(16.95 ± 0.1)
4096	18.7	<b>19.12</b>	16.24	12.35	7.07	1.55	<u>18.93</u>	16.4	(16.27 ± 0.09)
8192	<u>18.95</u>	18.29	18.72	13.96	7.32	1.55	<b>20.96</b>	16.42	(16.23 ± 0.12)

## 4.2 Baseline Methods and Evaluation Metrics

We compare our method against seven carefully chosen baseline models. We can categorize our baseline models as the following: (1) random-walk based models DeepWalk [10] and node2vec [4]; (2) Neural Network based model LINE [13]; (3) matrix-factorization based model HOPE [9]; (4) Hash based model NodeSketch [19]; (5) Inductive model GraphSAGE [6] with Mean aggregation (we use node degrees as features); and (6) Random projection binary model NODESIG [23]. We omitted traditional methods like GraRep [1] and SDNE [17] and chose HOPE and LINE from those categories because of their superior performance.

For our experiments, we used the setup from [4] and considered the commonly used evaluation metrics by our competitors. For multi-label node classification task, we consider Macro-F1 score to show results across different embedding dimensions. We use AUC-ROC score for link prediction experiment.

## 4.3 Experimental Setup

We design experiments to evaluate our and the competitors' performance per bit. Most of our floating-point based competitors are based on double-precision, 64 bits per dimension, except LINE model which is based on single precision. NodeSketch uses integers so it uses 32 bits per dimension and NODESIG is a binary embedding method so it allocates a single bit per dimension. To be fair with all the models, we consider the bit range [128, 256, 512, 1024, 2048, 4096, 8192]. For NODE2BINARY, we run 1000-epoch experiments, evaluating performance every 100 epochs. Since our algorithm is randomized, we repeat each experiment 5 times for each task and dataset. We report the best results along with mean and standard deviation in Tables 2 and 3. Other than dimension  $d$ , we have hyper parameters: depth of the CP tree  $\ell$  and sibling similarity coefficient  $\gamma$ . Hyper parameters for the discrete gradient computations are positive and negative sample weights  $\alpha$  and  $\beta$ , negative sample multiplier  $n^-$ , learning rate  $r_\ell$  and bias  $b_\ell$ . We sample  $k = 10$  siblings per entity for all of our experiments. We reuse CP tree across different dimension once it is formed, to ensure consistency of the

**Table 3: Link Prediction task, according to number of bits in embeddings**

		Models							
		DeepWalk	node2vec	LINE	HOPE	NodeSketch	GraphSAGE	NODESIG	node2binary
bits (dim)	Link Prediction (Metric: AUC %)								
<b>YouTube</b>									
128	55.31	55.56	60.03	52.44	56.22	50	<u>69.54</u>	<b>75.62</b> (73.39 ± 1.31)	
256	44.07	46.81	64.4	65.19	60.66	65.77	<u>74.36</u>	<b>78.31</b> (76.6 ± 1.03)	
512	56.43	58.96	71.31	63.1	68.27	50	<u>75.44</u>	<b>80.49</b> (79.87 ± 0.43)	
1024	62.35	65.01	71.58	63.02	66.62	65.77	<u>79.62</u>	<b>83.16</b> (82.39 ± 0.57)	
2048	63.16	64.37	75.06	62.51	71.59	65.77	<u>82.6</u>	<b>85.08</b> (84.48 ± 0.33)	
4096	64.24	69.45	73.65	62.89	70.65	65.77	<u>83.19</u>	<b>85.52</b> (85.23 ± 0.22)	
8192	63.56	68.89	76.31	61.91	73.46	65.77	<b>86.79</b>	<u>84.53</u> (84.3 ± 0.14)	
<b>Blogcatalog</b>									
128	49.05	46.31	57.48	53.06	51.96	50.5	<u>58.93</u>	<b>76.8</b> (74.92 ± 1.22)	
256	60.86	59.06	56.81	55.09	52.91	50	<u>63.01</u>	<b>80.27</b> (77.93 ± 1.58)	
512	61.61	59.68	62.8	52.39	56.09	50.5	<u>65.89</u>	<b>80.42</b> (79.38 ± 0.58)	
1024	62.83	59.5	<u>66.68</u>	53.35	50.86	50.5	61.64	<b>80.55</b> (79.98 ± 0.63)	
2048	59.84	60.57	<u>69.06</u>	53.31	59.55	50.5	67.35	<b>80.08</b> (79.49 ± 0.44)	
4096	62.0	60.79	71.29	53.5	61.71	50.5	<u>78.47</u>	<b>79.24</b> (78.82 ± 0.26)	
8192	58.51	60.38	69.96	53.59	62.76	50.5	<b>83.48</b>	<u>77.48</u> (77.02 ± 0.28)	
<b>PPI</b>									
128	51.77	51.66	49.17	48.48	<b>63.1</b>	50	51.74	<u>57.35</u> (56.86 ± 0.49)	
256	53.74	53.75	50.28	49.3	<b>62.53</b>	50	52.9	<u>59.44</u> (58.3 ± 0.95)	
512	54.24	52.82	51.05	49.47	<u>62.64</u>	50.42	58.75	<b>63.8</b> (62.28 ± 1.05)	
1024	54.72	53.28	52.71	50.68	<u>60.64</u>	50.42	59.54	<b>63.27</b> (62.78 ± 0.4)	
2048	51.31	53.99	58.58	50.13	62.49	50.42	<b>67.38</b>	<u>63.99</u> (63.41 ± 0.41)	
4096	52.66	52.44	53.7	50.05	<u>64.11</u>	50.42	<b>70.53</b>	62.91 (62.51 ± 0.35)	
8192	51.67	51.8	53.9	50.33	<u>62.7</u>	50.42	<b>70.44</b>	61.35 (60.94 ± 0.41)	
<b>DBLP</b>									
128	47.9	47.3	<u>63.51</u>	53.24	<b>66.34</b>	50	54.61	54.49 (53.56 ± 0.76)	
256	46.68	47.9	<u>63.53</u>	52.93	<b>67.32</b>	50	55.47	55.69 (54.72 ± 0.67)	
512	49.98	49.9	<u>62.92</u>	53.32	<b>68.79</b>	51.76	57.44	56.69 (56.11 ± 0.44)	
1024	48.69	49.04	<u>62.0</u>	53.43	<b>64.33</b>	51.76	58.3	58.4 (58.02 ± 0.29)	
2048	48.99	51.23	59.4	53.4	<b>64.36</b>	51.76	59.21	<u>60.59</u> (60.32 ± 0.31)	
4096	49.96	51.27	56.85	51.99	<b>62.62</b>	51.76	<u>61.51</u>	60.78 (60.61 ± 0.2)	
8192	50.91	52.17	54.78	51.12	<b>61.27</b>	51.76	<u>60.97</u>	59.19 (58.66 ± 0.35)	

results. Apart from the dimension  $d$ , we used best reported hyper parameters in competitors' works. For node2vec [4], following the authors we learned best in-out  $p$  and return  $q$  hyperparameters by 10 fold cross-validation on 10% labeled data using a grid search over  $\{0.25, 0.5, 1, 2, 4\}$ . We ran all models on a Tesla A100 GPU with 128 GB memory.

#### 4.4 Multi-label Node Classification

**Experimental setting:** Node classification utilizes a *labeled* dataset, where each node of the graph has one or more labels from a labelset  $L$ . The task is to correctly classify all the labels for each node; it becomes harder as the size of  $L$  increases. Once we have the embeddings for each node in the dataset, we perform 10-fold cross-validation and randomly sample 50% of nodes to train a One-vs-Rest classifier model with Logistic Regression using 'Liblinear' solver for 1000 iterations, keeping the other 50% of nodes to evaluate classification performance. For YouTube graph, we use a random sample of 50% nodes for classification because of its large size.

**Experimental results:** We report node classification experimental results in Table 2 with datasets in decreasing order by node count. Based on the results we are either best (bold) or second best (underline) most of the time, particularly as the number of bits decreases. For the largest dataset, YouTube, we reach 9.27% F1-score at just 256 bits whereas several competitors need 8192 bits to achieve similar performance. For the second largest, DBLP, we quickly achieve 94% F1-score at 512 bits and maintain superior performance across all dimensions. DBLP is a 2-label dataset, so most models perform better on it. For BlogCatalog, again we perform competitively at lower bit resolution. The closest competitor to NODE2BINARY is another binary model, NODESIG, which performs well at high bit resolution. Among other competitors, DeepWalk and node2vec perform best or second best for this dataset at higher bit resolutions. For PPI we achieve 18% F1-score at 256 bits, unlike most competitors who need 2048 bits to get similar F1-score. The big takeaway from this experiment is that most of our competitors need many bits to perform well, but NODE2BINARY is superior with fewer bits. Even the binary-based NODESIG performs worse than our model at low

dimensions. For a finer-grained analysis of node classification, we vary train-test ratio from 0.1 to 0.9 keeping  $d = 512$  and other setup as before. We report these results in Appendix A.2.

#### 4.5 Link Prediction

**Experimental setting:** Link prediction is the task of determining, whether there exists an edge (link) between given two (user-provided) nodes in a graph. We perform “Hadamard” operation to get an edge embedding vector from the two node embedding vectors. We randomly sample 50% edges in the graph and use it as train set, ensuring that the train subgraph is connected. We also generate a set of negative samples from both train and test set respectively. We train a Logistic Regression classifier using edge embeddings from the train dataset as features with ‘liblinear’ solver with max iterations at 1000. For the Blogcatalog and Youtube datasets, we subsample 50% of nodes (ensuring connectivity) from the original graphs *before* the train-test split due to their larger edge counts. After subsampling, BlogCatalog has 12.7% and YouTube has 15.7% of their original edge counts.

**Experimental results:** We report link prediction experimental results in Table 3 with datasets in decreasing order of edge count. We perform better for the largest datasets, YouTube and BlogCatalog, across all bit-resolutions, except at 8192 bits where we lose to the binary-based NODESIG. For BlogCatalog dataset, we achieve 80% AUC score at just 256 bits, while NODESIG achieves that at 8192 bits. Most of our competitors could not beat 70% AUC-score for this dataset. For PPI, we come out best or second best at lower bit resolutions, losing to NodeSketch and NODESIG at high bit resolutions. These results again show that NODESIG requires large dimensions to perform well, unlike us. For the smallest dataset DBLP, NodeSketch does best across all dimensions. Our model overall came out third best after LINE across all dimensions. GraphSAGE performs poorly across all datasets due to lack of built-in node attributes.

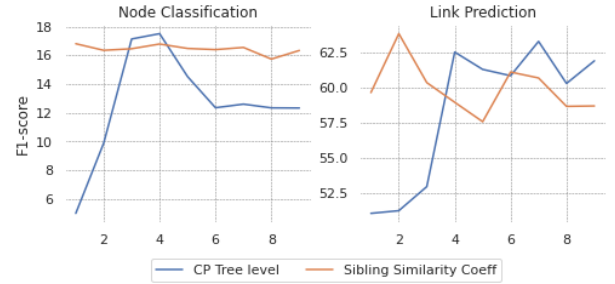
#### 4.6 Scalability Experiment

**Experimental setting:** We compare our algorithm training time with discrete embedding competitors for the large YouTube dataset (Node count 1.1M, edge count: 3M). After forming CP tree, for hierarchical pairs we randomly sample indirect edges to keep total edge counts  $\leq 2M$  and total sibling pairs  $\leq 1M$ . We keep node2binary at dimension 512 and run for 100 iterations. We repeat this experiment 5 times and take the average time. We give our competitors the best hyper parameters from their papers. We omit comparison with random-based, matrix-factorization, neural-network and inductive models since they take much longer to run.

**Experimental Results:** We report experimental results in Table 9 in the Appendix. Among the competitors NodeSketch takes about 0.41x time, and NODESIG 1.12x, compared to our model. Note that, our implementation is in Python, whereas our scalable competitors implemented their models in C or MATLAB which are generally faster programming languages.

#### 4.7 Parameter Sensitivity

We mentioned the hyperparameters used for our model in the Experimental Setup section. Two hyperparameters are critical for



**Figure 2: Parameter sensitivity experiment varying tree level & sibling similarity factor on PPI dataset for  $d = 512$ .**

NODE2BINARY’s performance, so we elaborate their effect on node classification and link prediction performance using PPI dataset.

**Effect of CP Tree level,  $\ell$ :** CreateTree is an important step for our method. We use Leiden algorithm to partition the graph and repeat the process for  $\ell$  layers, where  $\ell$  is a hyperparameter. Our experimental section shows that for node classification performance, it is better to have a small  $\ell$ . It makes sense, since in node classification, adjacent nodes should have similar embeddings. On the contrary, for link prediction, a high  $\ell$  is better as a taller tree can capture graph structure better which leads to better link prediction performance. We illustrate the effect of  $\ell$  in Figure 2.

**Effect of sibling similarity coefficient,  $\gamma$ :** For sibling similarity in our work, we introduce a hyperparameter  $\gamma$  as positive similarity weight. We set this parameter based on the ratio between hierarchical pairs and sibling pairs to offset any respective bias. We illustrate the effect of sibling similarity coefficient,  $\gamma$  in Figure 2. We report the best hyperparameters for each task and dataset combination in Table 8 in the Appendix.

## 5 Conclusion

To conclude, we propose NODE2BINARY, a novel method for learning representation vectors of the vertices of a general graph in binary space. It uses a fast community detection algorithm to convert a general graph into a community partition tree and then formulates the node representation vector learning as a combinatorial optimization task over the edges of that tree, which it solves through an innovative combination of discrete gradient descent and randomization. The main strength of NODE2BINARY is that the learned representation vectors that it produces are compact with small memory footprint, making them suitable for space- and energy-constrained environments. Besides, the learning algorithm is very efficient, making the method scalable to graphs with millions of vertices. In comparison with the state-of-the-art graph representation learning methods on standard evaluation tasks with real-world benchmark graphs, NODE2BINARY demonstrates superior results over the competitors in memory-constrained environment. In terms of limitations, NODE2BINARY is a transductive model that cannot generate embeddings for unseen nodes in a graph. A possible future direction of this work is to extend NODE2BINARY for inductive node representation learning.

## 6 Acknowledgments

Dr. Hasan’s research is supported by a National Science Foundation (NSF) grant numbered 2417275.

## References

- [1] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. GraRep: Learning Graph Representations with Global Structural Information. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2015-10-17) (CIKM '15). Association for Computing Machinery, 891–900. <https://doi.org/10.1145/2806416.2806512>
- [2] Haochen Chen, Bryan Perozzi, Yifan Hu, and Steven Skiena. 2017. HARP: Hierarchical Representation Learning for Networks. <https://doi.org/10.48550/arXiv.1706.07845> arXiv:1706.07845 [cs]
- [3] Haochen Chen, Syed Fahad Sultan, Yingtao Tian, Muhao Chen, and Steven Skiena. 2019. Fast and Accurate Network Embeddings via Very Sparse Random Projection. <https://doi.org/10.48550/arXiv.1908.11512> arXiv:1908.11512 [cs]
- [4] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. <https://doi.org/10.48550/arXiv.1607.00653> arXiv:1607.00653 [cs, stat]
- [5] Croix Gyurek, Niloy Talukder, and Mohammad Al Hasan. 2024. Binder: Hierarchical Concept Representation through Order Embedding of Binary Vectors. <https://doi.org/10.48550/arXiv.2404.10924> arXiv:2404.10924 [cs]
- [6] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. <https://doi.org/10.48550/arXiv.1706.02216> arXiv:1706.02216 [cs, stat]
- [7] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. <https://doi.org/10.48550/arXiv.1609.02907> arXiv:1609.02907 [cs, stat]
- [8] Jiongqian Liang, Saket Gururkar, and Srinivasan Parthasarathy. 2020. MILE: A Multi-Level Framework for Scalable Graph Embedding. <https://doi.org/10.48550/arXiv.1802.09612> arXiv:1802.09612 [cs]
- [9] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016-08-13) (KDD '16). Association for Computing Machinery, 1105–1114. <https://doi.org/10.1145/2939672.2939751>
- [10] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. <https://doi.org/10.1145/2623330.2623732>
- [11] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (2018-02-02). 459–467. <https://doi.org/10.1145/3159652.3159706> arXiv:1710.02971 [cs, stat]
- [12] Leonardo F. R. Ribeiro, Pedro H. P. Savarese, and Daniel R. Figueiredo. 2017. struc2vec: Learning Node Representations from Structural Identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017-08-04). 385–394. <https://doi.org/10.1145/3097983.3098061> arXiv:1704.03165 [cs, stat]
- [13] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale Information Network Embedding. In *Proceedings of the 24th International Conference on World Wide Web* (2015-05-18). 1067–1077. <https://doi.org/10.1145/2736277.2741093> arXiv:1503.03578 [cs]
- [14] Lei Tang and Huan Liu. 2009. Relational learning via latent social dimensions. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2009-06-28) (KDD '09). Association for Computing Machinery, 817–826. <https://doi.org/10.1145/1557019.1557109>
- [15] Lei Tang and Huan Liu. 2009. Scalable learning of collective behavior based on sparse social dimensions. In *Proceedings of the 18th ACM conference on Information and knowledge management* (New York, NY, USA, 2009-11-02) (CIKM '09). Association for Computing Machinery, 1107–1116. <https://doi.org/10.1145/1645953.1646094>
- [16] V. A. Traag, L. Waltman, and N. J. van Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. 9, 1 (2019), 5233. <https://doi.org/10.1038/s41598-019-41695-z>
- [17] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural Deep Network Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016-08-13) (KDD '16). Association for Computing Machinery, 1225–1234. <https://doi.org/10.1145/2939672.2939753>
- [18] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. 2018. Efficient attributed network embedding via recursive randomized hashing. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden, 2018-07-13) (IJCAI'18). AAAI Press, 2861–2867.
- [19] Dingqi Yang, Paolo Rosso, Bin Li, and Philippe Cudre-Mauroux. 2019. NodeSketch: Highly-Efficient Graph Embeddings via Recursive Sketching. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (New York, NY, USA, 2019-07-25) (KDD '19). Association for Computing Machinery, 1162–1172. <https://doi.org/10.1145/3292500.3330951>
- [20] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (New York, NY, USA, 2012-08-12) (MDS '12). Association for Computing Machinery, 1–8. <https://doi.org/10.1145/2350190.2350193>
- [21] Jie Zhang, Yuxiao Dong, Yan Wang, Jie Tang, and Ming Ding. 2019. ProNE: fast and scalable network representation learning. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China, 2019-08-10) (IJCAI'19). AAAI Press, 4278–4284.
- [22] Ziwei Zhang, Peng Cui, Haoyang Li, Xiao Wang, and Wenwu Zhu. 2018. Billion-scale Network Embedding with Iterative Random Projection. <https://doi.org/10.48550/arXiv.1805.02396> arXiv:1805.02396 [cs, stat]
- [23] Abdulkadir Çelikkanat, Fragkiskos D. Malliaros, and Apostolos N. Papadopoulos. 2023. NodeSig: Binary Node Embeddings via Random Walk Diffusion. In *Proceedings of the 2022 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (Istanbul, Turkey, 2023-06-28) (ASONAM '22). IEEE Press, 68–75. <https://doi.org/10.1109/ASONAM55673.2022.10068621>

## A Appendix

## A.1 Computation of Gradients

The discrete gradient functions we use for updating our embeddings are designed to guide the model towards fulfilling the tree and sibling constraints. We can think about the gradient as encouraging or discouraging flipping certain bits. If flipping a bit increases loss value, the associated gradient is positive, and vice-versa. For positive tree pairs  $(a, b)$ , for each bit  $j$ , we want to avoid having  $a_j = 0$  and  $b_j = 1$ . Therefore, for each pair of vectors  $a, b$  with  $a$  is a child of  $b$ , we assign gradient values to encourage corresponding bit pairs  $(a_j, b_j)$  to flip away from  $(0, 1)$ . If  $(a_j, b_j) = (0, 1)$ , then we set the gradient of positive pair loss with respect to both  $a_j$  and  $b_j$  to  $-1$ , so that at least one of them flips. If  $(a_j, b_j) = (0, 0)$ , then we are fine, but we want to avoid flipping  $b_j$ , which would create a  $(0, 1)$  pair, so we set the gradient of  $b_j$  to 1. The other cases are similar and shown in Table 4. In the table, a comment column is added to highlight the expected behavior of the trained model.

Table 4: Logic truth table for Computing gradient of positive pair hierarchical loss

$a_j$	$b_j$	$\Delta_{a_j} \text{Loss}^{\text{hier},+}$	$\Delta_{b_j} \text{Loss}^{\text{hier},+}$	Comments
0	0	0	1	Don't flip $b_j$
0	1	-1	-1	Flip either bit
1	0	0	0	Allowed, do nothing
1	1	1	0	Don't flip $a_j$

Table 5: Logic truth table for computing gradient of negative pair hierarchical loss gradient

$a'_j$	$b'_j$	$\Delta_{a'_j} \text{Loss}^{\text{hier},-}$	$\Delta_{b'_j} \text{Loss}^{\text{hier},-}$	Comments
0	0	0	-1	Flip $b'_j$ if $G = 0$
0	1	1	1	Protect if $G = 1$
1	0	0	0	Allowed, do nothing
1	1	-1	0	Flip $a'_j$ if $G = 0$

For the negative samples  $(a \not\equiv b)$ , we want to enforce that for at least one bit index (out of  $d$  indices) the representation vectors  $(a'$  and  $b')$  exhibits a  $(0, 1)$  pattern—negation of implication. So, we only care about pairs  $(a', b')$  that are on the threshold—those satisfying  $G(a', b') \leq 1$  where  $G$  is the count of  $(0, 1)$  pairs over all the bit indices in representation vectors,  $a'$  and  $b'$ . In other words, if  $G(a', b') > 1$ , the gradient of negative pair hierarchical loss is 0 with respect to both  $a'$  and  $b'$ , as defined in Eq. (9-10). Now for the

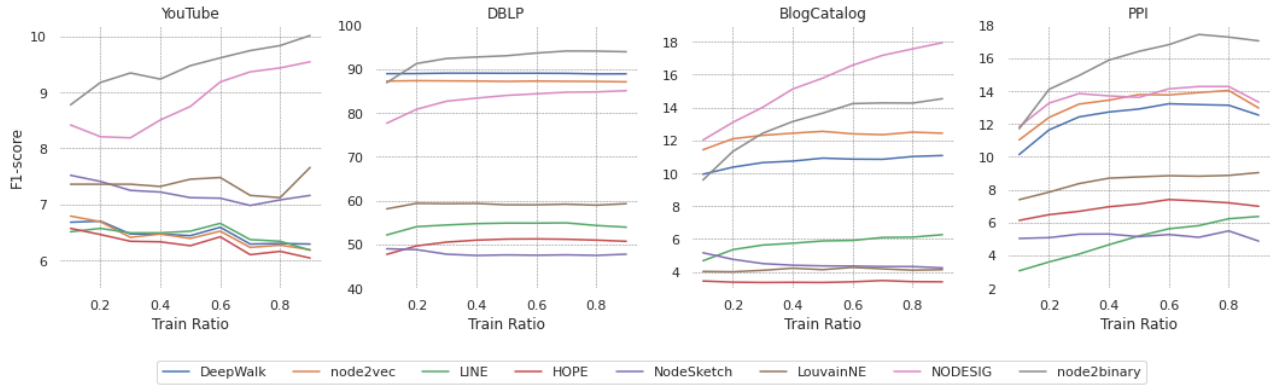


Figure 3: Node Classification experiment by varying train ratio from 0.1 to 0.9 with embedding dimension set at 512.)

Table 6: Logic truth table for computing gradient of sibling loss

$c_j$	$d_j$	$\Delta_{c_j} Loss$	$\Delta_{d_j} Loss$	Comments
0	0	1	1	Don't flip
0	1	-1	-1	Flip
1	0	-1	-1	Flip
1	1	1	1	Don't flip

threshold cases: if  $G = 1$ , we protect the solo  $(0, 1)$  pair by setting the gradient to  $+1$  for that bit-index; if  $G = 0$ , we flip the  $a'$  side of  $(1, 1)$  pairs or the  $b'$  side of  $(0, 0)$  pairs, by setting those gradients to  $-1$ . See Table 5.

In a similar way, we show in Table 6, the gradients for sibling loss, which aims to make the two bits equal. Note that, when  $c_j \neq d_j$ , we set both gradients to  $-1$ , because we don't care which bit flips, only that one of them does. This means there is a chance *both* flip, causing  $c_j \neq d_j$  again. This does not hinder the progress in optimization, because the flip probability as computed in Eq. 12, on average, flips at most half of the bits in the model.

## A.2 Fine-grained Node Classification Task

We perform this experiment to show the effect on node classification performance for varying degree of train-test ratio for NODE2BINARY and competing methods. We give all models 512 bits. The experiment results is shown in Figure 3. In all four plots, along the x-axis is train ratio, and along the y-axis is F1-score (the higher the better). Our model NODE2BINARY is the gray line. For the largest dataset (YouTube), all our competitors perform poorly. For the second largest (DBLP), NODE2BINARY performs superior except at the lowest training ratio, 0.1. For BlogCatalog dataset NODE2BINARY comes second to NODESIG. For PPI, our model achieves best F1 score across all training ratio. Overall our model achieves better F1-score in all datasets.

## A.3 NODE2BINARY Model Convergence Results

To validate our model convergence, we plot both Hierarchical loss and Sibling Similarity loss for 100 iterations for both node classification and link prediction tasks. For this experiment, we use PPI dataset with dimension 512. We show our results in Figure 4. For

both tasks, with successive iterations, Hierarchical loss and sibling similarity loss converges to smaller values.

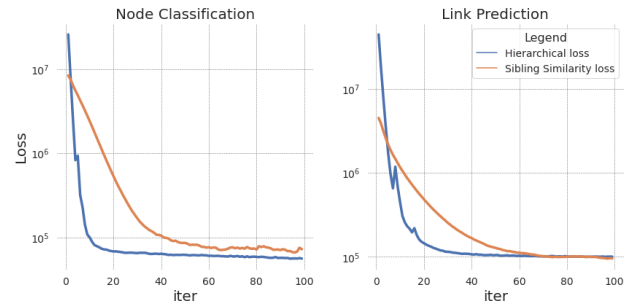


Figure 4: Convergence results (first 100 iterations) for both tasks on PPI dataset setting embedding dimension at 512.

## A.4 Ablation Study

Table 7: Ablation study for all datasets ( $d=512$ ), NC = Node Classification, LP = Link Prediction.

Component	dataset			
	YouTube	BlogCat	DBLP	PPI
NODE2BINARY (NC)	9.51	16.08	94.38	17.86
n2b - Hier loss (NC)	6.77	3.44	39.46	1.34
n2b - Sib loss (NC)	7.77	6.85	58.3	5.27
NODE2BINARY (LP)	80.49	80.42	56.69	63.8
n2b - Hier loss (LP)	50.05	50	50	50.05
n2b - Sib loss (LP)	70.28	68.62	54.04	60.06

Our model's loss function has two major components. The first is hierarchical loss along the CP tree edges, and the second is minimization of Hamming distance among the sibling pairs in the tree. We perform an ablation study where we set  $\alpha$  and  $\beta$  to their default values and set  $\gamma = 0$  to ignore the effect of homophily for the sibling pairs. Then we set  $\alpha$  and  $\beta$  to 0 and  $\gamma = 1$  to ignore the effect of hierarchy and only consider homophily. We set the rest of the parameters to the default values. We perform this experiment for both tasks for all the datasets ( $d = 512$ ) and report the result in Table 7.

First three rows show results for node classification and the last three rows show results for link prediction. From this experiment we can see that removing hierarchical loss has stronger effect on NODE2BINARY performance compared to removing sibling loss.

### A.5 Optimal Hyper-Parameters

We report the optimal hyper-parameter configuration for the *tree depth*  $\ell$  and *sibling similarity coefficient*  $\gamma$  for each dataset in Table 8. These parameters are independent of the number of bits because we reuse the same CP tree at each dimension.

**Table 8: Best hyper parameters (CP Tree depth, Sibling Similarity Coefficient) for each dataset and task combination.**

Task	dataset			
	YouTube	BlogCat	DBLP	PPI
Node Classification	(4,1)	(5,1)	(3,1)	(3,1)
Link Prediction	(5,1)	(5,1)	(4,2)	(5,3)

### A.6 Reproducibility

Our code and datasets are publicly available at the following link: <https://github.com/ntalukde/node2binary/>. It is implemented in Python with the PyTorch library. Although node2binary is a randomized algorithm, we maintain the same random tree  $\mathcal{T}_G$  for each run to increase consistency. We also provide a command-line option `--seed` in our implementation to seed the random number generator and make the result deterministic.

**Table 9: Algorithm Running Time on YouTube dataset (Task: Link Prediction)**

Model	NodeSketch	NODESIG	NODE2BINARY
Time (s)	2488	6727	CP Tree = 184 Alg Run = 5839
n2b Speed Up	0.41x	1.12x	1x