Accelerating DeepWalk via Context-Level Parameter Update and Huffman Tree Pruning

Chang Gong[1], Weiguo Zheng[1(✉)], and Hongwei Feng[2]

[1] School of Data Science, Fudan University, Shanghai, China
changgong22@m.fudan.edu.cn, zhengweiguo@fudan.edu.cn
[2] Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China
hwfeng@fudan.edu.cn

Abstract. As a classic unsupervised graph representation learning algorithm, DeepWalk is widely used in various tasks. DeepWalk consists of random walk and Word2Vec-based training, and the latter is more time-consuming due to large-scale softmax computation. To address this, many approximation methods have been proposed. Among them, sampling-based methods struggle to achieve satisfactory accuracy due to large sampling variance and the mismatch between sampling and true distribution. For softmax-based methods, their training efficiency is not high. To this end, we propose CLPU and HTP to reduce redundant operations in these methods. Firstly, we observe that there exist multiple updates for the same node within one window. Based on this, CLPU counts node occurrences and label contributions, and then performs parameter update only once. Secondly, we delve into whether it is necessary to update Huffman tree's upper-level nodes multiple times and propose HTP: focusing on a few auxiliary nodes before leaves instead of the whole traversal path from root, which reduces time complexity from logarithmic to constant. Experiments on 14 datasets demonstrate that CLPU and HTP achieve a speed improvement of 1.9x-2.8x and 2.8x-4.3x, while surpassing other methods in terms of accuracy.

Keywords: DeepWalk · Algorithm Acceleration · Huffman Tree

1 Introduction

Graph representation learning aims at designing algorithms to learn node embeddings for downstream tasks such as node classification [4, 10] and link prediction [14, 17]. Among these algorithms, random-walk-based [4, 14, 15] are most widely used for real-world graphs. Although GNNs [6, 17] have gained popularity in recent years, they still require raw input features and ground-truth labels for training, which are difficult to collect. Meanwhile, GNNs cannot handle web-scale graphs due to issues of neighbor explosion [6]. However, random-walk-based methods do not have these problems and possess scalability. Thus, it is still meaningful to research how to accelerate these algorithms.

Random-walk-based methods consist of random walk on graphs and Word2Vec-based training [10]. Since the former can be parallelized by starting from multiple initial nodes concurrently, its time cost is neg liable. However, the latter needs to update high-dimensional node feature, which is more time-consuming and requires careful scheduling of parallel operations to avoid read-write conflicts between different threads. Therefore, taking DeepWalk [14] as an example, our goal is to improve training efficiency of the latter.

Existing Approaches and Limitations: In Word2Vec-based training, the most time-consuming operation is large-scale full softmax: when calculating node probability, we need to traverse all nodes to compute the denominator of softmax. Many works have attempted to alleviate this issue, which can be divided into two categories: softmax-based [3,8,10,12] and sampling-based approaches [1,2,4,5,7, 11]. The former does not discard softmax layer but adjusts its computation logic, while the latter removes softmax and designs a new loss function to approximate it. Although sampling-based methods are more suitable for GPU acceleration, they still face problems like huge divergence between target node's sampling distribution and real node distribution on graphs, and large sampling variance. For softmax-based methods, the most famous one is Hierarchical Softmax(HS), which transforms the flat structure of full softmax into a tree structure. However, its training efficiency is still limited due to many redundant update operations.

Our Proposed Methods and Contributions: We focus on improving HS. First, we notice that HS updates the same node multiple times within the same window. This finding inspires us to propose a new method **CLPU** to reduce update frequency. Specifically, CLPU updates node feature only once after counting the number of times it appears. It also constructs a better loss function to compensate for the insufficiency in training. Second, we discuss why upper-level nodes in Huffman tree do not require frequent updates and provide two reasons: (1) The range of embedding changes for these nodes gradually narrows with more iterations. (2) Frequent updates to these nodes diminish the distinctiveness of leaf nodes' features. Thus, we propose another method **HTP** to reduce update frequency of these nodes. Specifically, only a few auxiliary nodes near leaves will be updated, instead of all nodes along the tree traversal path. This approach helps to reduce time complexity from logarithmic to constant. Experimental results on 14 datasets show that CLPU and HTP achieve comparable or superior accuracy compared to other baselines, and accelerate training speed with 1.9x-2.8x and 2.8x-4.3x compared to the second fastest one.

Contributions. Our contributions are summarized as follows:

(1) We analyze computational redundancies in Deepwalk, including multiple updates for same node within same window and frequent updates to upper-level nodes in Huffman tree, which clarify the direction for acceleration.
(2) To solve these issues, we propose two methods CLPU and HTP. CLPU updates nodes only once after counting their occurrences and label contributions, and HTP only updates a few auxiliary nodes before leaf nodes instead of the whole tree path.
(3) Extensive experiments confirm effectiveness and efficiency of our methods.

2 Preliminaries

2.1 Word2Vec-based Training in DeepWalk

Word2Vec [10] trains word embeddings in an unsupervised manner, whose core idea is co-occurrence relationship between words in real corpora. To represent this relationship, Word2Vec proposes two methods: CBOW and Skip-gram. We use the latter to introduce Word2Vec-based training in DeepWalk.

Challenge: Large-Scale Full Softmax. For Skip-gram, given center node w, our goal is to predict $context(w)$, which equals to maximize $\prod_{u \in context(w)} \mathbb{P}(u|w)$. We use f and g to represent embeddings of the center and context nodes. Thus, $\mathbb{P}(u|w) = \frac{\exp(f(w)^T g(u))}{\sum_{v \in V} \exp(f(w)^T g(v))}$. For each (w, u) on the sampled path ($u \in context(w)$), we need to calculate the denominator with $\mathcal{O}(V)$ cost, which is unbearable for large-scale V. Thus, Word2Vec proposes a new method to approximate full softmax: Hierarchical Softmax [10, 12] (short as HS).

Huffman Tree and Word2Vec Parameter Update. HS constructs a Huffman tree using node set V. First, it selects two nodes n_1, n_2 with the smallest weights and removes them from V. Then n_1 and n_2 form a subtree, with parent node p's weight being the sum of n_1 and n_2, and we push p back to V. The above two steps are repeated until only one node is left in V, which is the root node of Huffman tree. Here, weight refers to node degree or PageRank [13] value.

From the construction process, node $u \in V$ only appears as a leaf of Huffman tree, with non-leaf nodes being auxiliary ones. Nodes with larger weights have smaller depths, while nodes with smaller weights are further from the root. For each leaf u, there is only one path from root to u: $(u_1, ..., u_{l^u})(u_{l^u} = u)$. If we set left child's label to 1 and right to 0, the above path corresponds to a unique binary representation. Thus, we have $\mathbb{P}(u|w) = \prod_{i=2}^{l^u} \mathbb{P}(u_i|w, u_{i-1})$. For each $\mathbb{P}(u_i|w, u_{i-1})$, we model it as

$$\mathbb{P}(u_i|w, u_{i-1}) = (\sigma(f(w)^T g(u_{i-1})))^{label(u_i)}(1 - \sigma(f(w)^T g(u_{i-1})))^{1-label(u_i)},$$

where σ is sigmoid function. Notice that $\mathbb{P}(u|w)$ is calculated by $(l^u - 1)$ times multiplication, which reduces time complexity from $\mathcal{O}(V)$ to $\mathcal{O}(\log V)$. Then we can calculate $\log \mathbb{P}(u|w)$, take derivatives with respect to $f(w)$ and $g(u_{i-1})$ and perform parameter update, which is shown in Algorithm 2 in detail (η is learning rate). The output node embeddings are stored in $f(w)$.

3 Improvements of DeepWalk

3.1 Context-Level Parameter Update

Motivation. Algorithm 2 has a problem: maximizing $\log \mathbb{P}(u|w)$ does not necessarily maximize $\sum_{u \in context(w)} \log \mathbb{P}(u|w)$. Moreover, the order in which $context(w)$ is traversed also affects final result, which needs to design an optimal traversal

order. A straightforward idea is to exchange lines 8 and 9 of Algorithm 2, that is, to collect all information in $context(w)$ to update $f(w)$. However, this does not significantly reduce the number of updates (since most updates occur in lines 4-6), and the efficiency improvement is quite limited.

Notice that in Huffman tree, paths corresponding to different leaf nodes may share a long prefix. According to Algorithm 2, many separate updates would be required for the shared prefix. This pattern of repeatedly updating the same auxiliary nodes within the same window is common throughout HS computation process. However, with a constant learning rate, we can count the number of times the prefix nodes are visited and then update them together, which reduces the number of parameter updates.

Approach. Based on the above discussion, for each $context(w)$, we count occurrence number and label contributions of different nodes in tree paths corresponding to $context(w)$, and then perform a single parameter update for $f(w)$ and $g(u)$, if u is visited during tree path traversal. Specifically, we calculate derivative of $\sum_{u \in context(w)} \log \mathbb{P}(u|w)$ with respect to $f(w)$ and $g(u_{i-1})$:

$$\frac{\partial \sum_{u \in context(w)} \log \mathbb{P}(u|w)}{\partial f(w)} = \sum_{u \in context(w)} \sum_{i=2}^{l^u} (label(u_i) - \sigma(f(w)^T g(u_{i-1})))g(u_{i-1}),$$

$$\frac{\partial \sum_{u \in context(w)} \log \mathbb{P}(u|w)}{\partial g(u_{i-1})} = \sum_{v \in S} (label(v) - \sigma(f(w)^T g(v)))f(w),$$

where $S = \cup_{u \in context(w)} \{u_j | 2 \leq j \leq l^u, u_j = u_{i-1}\}$. Notice that S can contain duplicate elements, and $label(v)$ is determined as 0 or 1 during the traversal of tree paths. Clearly, the gradient computation of $\sum_{u \in context(w)} \log \mathbb{P}(u|w)$ with respect to $g(u_{i-1})$ is not simply summing that of $\log \mathbb{P}(u|w)$. According to the above calculations, we obtain Algorithm 1, which is named as **CLPU** (**C**ontext-**L**evel **P**arameter **U**pdate). Specifically, given $(w, context(w))$, we first use two hashmaps to count the occurrences of different auxiliary nodes on the tree path corresponding to each leaf node $u \in context(w)$, as well as the sum of their labels (lines 1-6). Then, based on the aforementioned formula, we perform only one update for $g(key)$ if key is visited during traversal (lines 7-11), and finally update $f(w)$ (line 12).

Analysis. The main cost of original algorithm lies in lines 4-6, and each update is vector-level operation. Even with vectorized instructions, it is still time-consuming. In contrast, the main overhead of CLPU comes from lines 8-10, requiring only $|label\_map.keys()|$ executions. Since most nodes share a long prefix, the number of distinct nodes encountered during tree path traversal is limited, thereby ensuring a notable efficiency improvement.

3.2 Huffman Tree Pruning

Motivation. In Algorithm 2, nodes at higher levels of Huffman tree are frequently visited, such as root, which is updated during each traversal of tree

Algorithm 1 Context-Level Parameter Update

$num\_map = \varnothing, label\_map = \varnothing$
for $u \in context(w)$ **do**
  for $i = 2 : l^u$ **do**
    $num\_map[u_i] + +, label\_map[u_i] = label\_map[u_i] + label(u_i)$
  end for
end for
for $key \in label\_map.keys()$ **do**
  $label\_sum = label\_map[key], count\_sum = num\_map[key]$
  $t = label\_sum - count\_sum * \sigma(f(w)^T g(key))$
  $e = e + t * g(key), g(key) = g(key) + \eta * t * f(w)$
end for
$f(w) = f(w) + \eta * e$

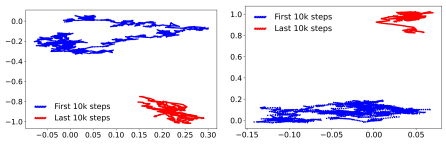
paths. In contrast, deeper leaf nodes are seldom updated, and their predecessors are also rarely updated. This is detrimental to the learning of these nodes.

To balance update frequency for nodes with different weights and reduce time costs, an intuitive idea is to decrease the number of updates for upper-level nodes. We verify its rationality from two aspects:

(1) The range of embedding changes for upper-level nodes gradually contracts with an increasing number of iterations. In other words, after a sufficient number of update operations, these nodes' embedding will fluctuate within a small range. We use two graphs to verify this phenomenon. Figure 1 shows position changes of Huffman tree's root node on a plane (embedding dimension is 2). By comparing the first (blue) and last (red) $10k$ iterations, we find that the range of embedding changes has contracted, which means there is considerable redundancy in the updates for these nodes.

(2) Frequent updates of upper-level nodes weaken the discernibility of leaf nodes. Considering the update of $f(w)$: $f(w) = f(w) + \eta * \sum_{i=2}^{l^u}(label(u_i) - \sigma(f(w)^T g(u_{i-1})))g(u_{i-1})$, it can be viewed as $f(w)$ moving one step towards $g(u_{i-1})$ in high-dimensional space. If for each update of $f(w)$, we accumulate gradient values from root node u_1, then $f(w)$ will be significantly influenced by $g(u_1)$, while auxiliary nodes near w contribute less to $f(w)$. As shown in Figure 2, u^1 and u^2 denote two leaf nodes that are initially distant in feature space. However, after a substantial number of steps directed towards $g(u_1)$ (blue arrows), they cluster as the first aspect discussed (blue u^1, u^2). Despite occasional movements in different directions (yellow and green arrows), $f(u^1)$ and $f(u^2)$ are still influenced by $g(u_1)$, leading to a diminishment of their unique features.

Approach. Based on the discussion above, we conclude that frequent updates are unnecessary for upper-level auxiliary nodes. Furthermore, the second aspect implies that accumulating gradient values starting from the root node each time is redundant. Instead, focus should be placed on the contributions of the auxiliary nodes that are adjacent to leaves. Thus, we introduce a new strategy with



(a) Wiki (b) Blog

Fig. 1: Embedding change of Huffman tree's root node during the first (blue) and the last (red) $10k$ steps.

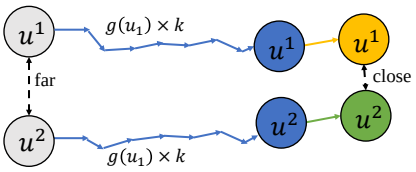


Fig. 2: Accumulating too many upper-level nodes (e.g., $g(u_1)$) diminishes the distinctive features between leaf nodes u^1, u^2.

Algorithm 2 Hierarchical Softmax

1: **for** $u \in context(w)$ **do**
2: $e = 0$
3: **for** $i = 2 : l^u$ **do**
4: $t = label(u_i) - \sigma(f(w)^T g(u_{i-1}))$
5: $e = e + t * g(u_{i-1})$
6: $g(u_{i-1}) = g(u_{i-1}) + \eta * t * f(w)$
7: **end for**
8: $f(w) = f(w) + \eta * e$
9: **end for**

Algorithm 3 Huffman Tree Pruning

1: **for** $u \in context(w)$ **do**
2: $e = 0$
3: **for** $i = max(2, l^u - C + 1) : l^u$ **do**
4: $t = label(u_i) - \sigma(f(w)^T g(u_{i-1}))$
5: $e = e + t * g(u_{i-1})$
6: $g(u_{i-1}) = g(u_{i-1}) + \eta * t * f(w)$
7: **end for**
8: $f(w) = f(w) + \eta * e$
9: **end for**

notable efficacy: during each traversal from root node to leaf node u, we only perform update for C auxiliary nodes preceding u (i.e., $u_i (l^u - C + 1 \leq i \leq l^u)$) and accumulate these nodes' contributions to update $f(w)$. We refer to this approach as **HTP**(**H**uffman **T**ree **P**runing), and its complete process is shown in Algorithm 3. The only modification is in line 3, which shortens the traversal of leaf node u's path (i.e., $(u_2, ..., u_{l^u})$) to a constant-length traversal (i.e., $(u_{l^u - C+1}, ..., u_{l^u})$).

Analysis. HTP effectively circumvents the issue of frequent updates for upper-level nodes. Each update focuses on auxiliary nodes near leaves, which can determine the distinctiveness between leaves. For upper-level nodes, their features are not kept fixed and can be updated using leaf nodes with shallow depths. Although HTP differs from the original algorithm only in one line (line 3), it achieves a significant improvement in efficiency. Compared to the latter whose update number is $\mathcal{O}(\log N)$, HTP only requires constant updates (w.r.t N). In experiments, we set $C \in \{6, 7, 8\}$, and have already obtained satisfactory results.

4 Experiments

4.1 Experimental Settings

Tasks and Datasets. We evaluate algorithm's performance on two tasks: node classification and link prediction. We select 14 datasets, which are collected from

Table 1: Micro-F_1 score(%) comparison on node classification task.

	Cora	**Wiki**	**Blog**	**CS**	**Pubmed**	**Physics**	**Flickr**	**Arxiv**
#Node	2,708	4,777	10,312	18,333	19,717	34,493	89,250	169,343
#Edge	10,556	184,812	667,966	163,788	88,651	495,924	899,756	2,315,598
Depth	11.6	12.7	14.3	14.5	15.0	15.4	16.9	17.9
HS	79.8±1.6	33.9±1.1	**33.3±0.6**	88.5±0.3	81.2±0.5	93.6±0.2	52.2±0.4	67.8±0.2
SS	80.8±0.7	35.8±0.6	31.9±0.3	88.0±0.2	81.0±0.3	93.1±0.3	52.1±0.1	68.3±0.1
IS	80.2±1.1	35.6±0.8	32.0±0.6	88.1±0.2	80.9±0.3	93.0±0.2	52.3±0.2	68.5±0.1
NEG	**81.3±1.1**	35.9±0.9	31.5±0.5	88.4±0.3	80.3±0.4	93.1±0.2	52.4±0.2	68.7±0.2
NCE	81.2±1.5	35.8±1.0	32.5±0.9	88.3±0.3	80.7±0.4	93.3±0.2	**52.7±0.2**	68.6±0.2
CLPU	80.4±1.1	**36.2±0.8**	32.4±0.8	**88.8±0.3**	**81.3±0.4**	**93.8±0.2**	52.4±0.4	**68.8±0.1**
HTP	80.6±1.1	36.0±1.3	32.7±0.7	88.7±0.3	81.2±0.4	93.7±0.1	52.1±0.3	68.5±0.2

DGL [16] and SNAP [9]. Detailed statistics can be found in Table 1 and Table 2, where "Depth" represents average depth of all leaf nodes in Huffman tree.

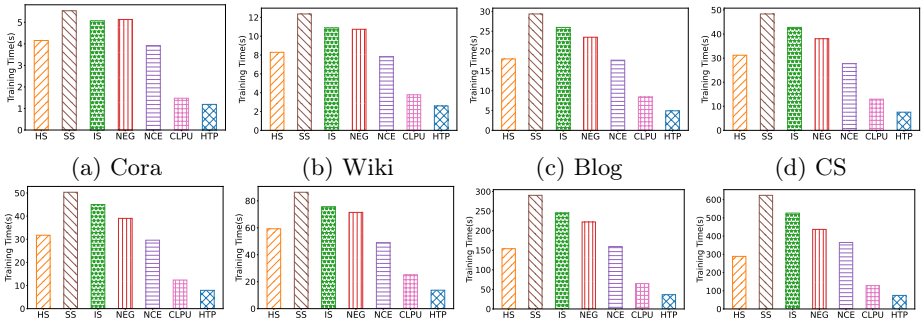
Baselines. We compare with five approaches for accelerating softmax: Hierarchical Softmax [10, 12] (short as HS), Sampled Softmax [7] (short as SS), Importance Sampling [1] (short as IS), Noise Contrastive Estimation [5, 11] (short as NCE) and one of its approximation Negative Sampling [4, 10] (short as NEG).

4.2 Node Classification

Accuracy Comparison. We first use DeepWalk to train node embedding and then employ logistic regression for node classification. 70% of nodes are used for training, while others are for testing. As shown in Table 1, CLPU and HTP achieve similar accuracy with other methods. Moreover, CLPU performs best on several large graphs (indicated in **bold font**). These results are as expected: CLPU updates parameters by integrating calculation results within a context, leading to a more accurate gradient estimation. For HTP, it reduces update number for upper-level nodes, which is equivalent to increasing update frequency of auxiliary nodes near leaves and helps them learn distinctive features. Although simplifying update strategy of HS might reduce accuracy in theory, both methods eliminate useless operations in HS, which have a marginal impact on accuracy.

Efficiency Comparison. Compared to minor fluctuations in accuracy, we expect our algorithms to significantly outperform other baselines in terms of efficiency. We focus on DeepWalk training acceleration and neglect the time cost of logistic regression since this part is shared among all methods. Results are shown in Figure 3. Clearly, both CLPU and HTP demonstrate a substantial reduction in training time. Specifically, they outperform HS by achieving speedups of 2.14x-2.82x for CLPU and 3.20x-4.33x for HTP.

Methods based on sampling negative examples or noise (like SS, IS, NCE, and NEG) often require a large number of negative samples to reduce softmax estimation variance and for each negative sample, parameter update is performed in a sequential manner. Consequently, these methods incur a high computational



(a) Cora (b) Wiki (c) Blog (d) CS

(e) Pubmed (f) Physics (g) Flickr (h) Arxiv

Fig. 3: Training time comparison on node classification task.

cost, which even exceeds that of HS in some cases. However, CLPU and HTP do not have these issues.

4.3 Link Prediction

AUC Comparison. To further validate the efficacy of our methods, we perform experiments on task link prediction. Specifically, we train DeepWalk to get node u's embedding $f(u)$ and calculate edge score $s(e) = f(u)^T f(v)$. Then AUC is computed as $AUC = \frac{\sum_{p \in pos, n \in neg} \mathbb{1}(s(p) > s(n))}{|pos| \times |neg|}$. Notice that compared to node classification, there is no need to perform logistic regression on the output embeddings of DeepWalk, which avoids its impact on final results.

The results of AUC comparison are shown in Table 2. Numbers in parentheses represent the increase or decrease in AUC compared to HS. Clearly, most baselines tend to perform significantly worse than HS. However, both CLPU and HTP continue to exhibit high AUC values. This suggests that the embeddings produced by CLPU and HTP are not only on par with other baselines when utilized as initial inputs for downstream tasks, but also possess superior representational capabilities when directly used for result evaluation.

Efficiency Comparison. Similarly, we also examine the runtime cost of CLPU and HTP in link prediction task. As shown in Table 3, aside from NCE, which has a runtime comparable to HS, the other baselines significantly lag behind HS. In contrast, CLPU and HTP still achieve substantial speedups of 1.9x-2.7x and 2.8x-4.2x over HS, marking a significant enhancement in performance.

4.4 Hyperparameter Study

We further conduct hyperparameter experiments for C, where $C \in [1, 10]$, and results are shown in Figure 4. Clearly, training time increases linearly with C, which aligns with expectations. However, the relationship between accuracy fluctuations and changes in C is not straightforward: on Github and Twitch, AUC shows a swift improvement as C increases, while on Amazon, it initially rises

Table 2: AUC comparison on link prediction task.

	Facebook	**Lastfm**	**Deezer**	**Github**	**Twitch**	**Amazon**
#Node	4,039	7,624	28,281	36,282	160,693	334,863
#Edge	88,234	27,806	92,752	238,543	3,407,103	925,872
Depth	12.4	13.5	15.3	16.1	18.4	18.7
HS	95.5	89.4	86.9	77.8	82.8	93.7
SS	88.4 (-7.1)	79.3 (-4.1)	75.5 (-11.4)	62.3 (-15.5)	72.8 (-10.0)	94.3 (+0.6)
IS	90.2 (-5.3)	81.8 (-7.6)	76.1 (-10.8)	64.9 (-12.9)	71.5 (-11.3)	93.0 (-0.7)
NEG	89.9 (-5.6)	77.8 (-11.6)	69.2 (-17.6)	52.2 (-25.6)	58.1 (-24.7)	94.6 (+0.9)
NCE	89.8 (-5.7)	78.6 (-10.8)	73.6 (-13.3)	59.4 (-18.4)	69.0 (-13.8)	94.5 (+0.8)
CLPU	95.3 (-0.2)	89.5 (+0.1)	87.0 (+0.1)	78.0 (+0.2)	82.7 (-0.1)	94.2 (+0.5)
HTP	95.7 (+0.2)	89.1 (-0.3)	87.3 (+0.4)	77.4 (-0.4)	82.4 (-0.4)	94.3 (+0.6)

Table 3: Training time comparison on link prediction task.

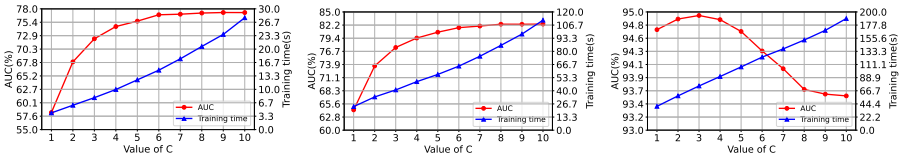
	Facebook	**Lastfm**	**Deezer**	**Github**	**Twitch**	**Amazon**
HS	6.7	12.0	44.9	61.0	275.2	518.1
SS	11.3 (×0.6)	20.5 (×0.6)	93.4 (×0.5)	146.2 (×0.4)	668.1 (×0.4)	1432.8 (×0.4)
IS	8.2 (×0.8)	17.2 (×0.7)	64.2 (×0.7)	89.8 (×0.7)	501.0 (×0.5)	1124.1 (×0.5)
NEG	10.1 (×0.7)	19.7 (×0.6)	71.0 (×0.6)	116.6 (×0.5)	565.6 (×0.5)	820.6 (×0.6)
NCE	5.8 (×1.2)	10.9 (×1.1)	39.0 (×1.2)	51.5 (×1.2)	304.6 (×0.9)	785.8 (×0.7)
CLPU	2.9 (×2.3)	4.6 (×2.6)	16.5 (×2.7)	32.7 (×1.9)	148.7 (×1.9)	230.8 (×2.2)
HTP	2.0 (×3.4)	3.2 (×3.8)	10.6 (×4.2)	21.6 (×2.8)	68.1 (×4.0)	177.8 (×2.9)

with C but then declines. However, for most graphs, setting $C \in \{6, 7, 8\}$ yields optimal results. Concurrently, the "Depth" of these graphs is between 16 and 19, which is approximately twice the value of the chosen C. Thus, for new graphs, we can set $C = \frac{1}{2}d$, where d is approximately equal to $\log N$.

5 Conclusion

In this paper, we analyze redundant operations in DeepWalk, focusing on repeated updates of same auxiliary nodes within same window and frequent updates of upper-level nodes in Huffman tree. To address this, we design two strategies CLPU and HTP. The former counts the occurrences of the same nodes and their label contributions, reducing multiple updates to a single operation. The latter only considers the contributions of a few auxiliary nodes preceding leaves during the traversal of tree path. Experimental results on 14 datasets confirm the effectiveness and efficiency of CLPU and HTP.

Acknowledgments. This work was substantially supported by Key Projects of the National Natural Science Foundation of China (Grant No. U23A20496).



(a) Github (b) Twitch (c) Amazon

Fig. 4: AUC and time comparison with different C on link prediction.

References

1. Bengio, Y., Senecal, J.S.: Quick training of probabilistic neural nets by importance sampling. In: AISTATS (2003)
2. Bengio, Y., Senecal, J.S.: Adaptive importance sampling to accelerate training of a neural probabilistic language model. IEEE Transactions on Neural Networks (2008)
3. Costa-jussà, M.R., Fonollosa, J.A.R.: Character-based neural machine translation. In: ACL (2016)
4. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. Proceedings of the 22nd ACM SIGKDD (2016)
5. Gutmann, M., Hyvärinen, A.: Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In: AISTATS (2010)
6. Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: NeurIPS (2017)
7. Jean, S., Cho, K., Memisevic, R., Bengio, Y.: On using very large target vocabulary for neural machine translation. arXiv preprint arXiv:1412.2007 (2014)
8. Kim, Y., Jernite, Y., Sontag, D., Rush, A.M.: Character-aware neural language models. In: AAAI (2016)
9. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (2014)
10. Mikolov, T., Chen, K., Corrado, G.S., Dean, J.: Efficient estimation of word representations in vector space. In: ICLR (2013)
11. Mnih, A., Teh, Y.W.: A fast and simple algorithm for training neural probabilistic language models. In: ICML (2012)
12. Morin, F., Bengio, Y.: Hierarchical probabilistic neural network language model. In: AISTATS (2005)
13. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab (1999)
14. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: online learning of social representations. In: Proceedings of the 20th ACM SIGKDD (2014)
15. Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., Mei, Q.: Line: Large-scale information network embedding. In: TheWebConf (2015)
16. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J.J., Li, J., Smola, A., Zhang, Z.: Deep graph library: Towards efficient and scalable deep learning on graphs. arXiv preprint arXiv:1909.01315 (2019)
17. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: ICLR (2019)