Memory-Aware Framework for Efficient Second-Order Random Walk on Large Graphs

Yingxia Shao[⋆], Shiyue Huang[§], Xupeng Miao[§], Bin Cui[§], Lei Chen[△]
[⋆]Beijing Key Lab of Intelligent Telecommunications Software and Multimedia, BUPT
[§]School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
[△]Hong Kong University of Science and Technology
shaoyx@bupt.edu.cn, {huangshiyue, xupeng.miao, bin.cui}@pku.edu.cn, leichen@cse.ust.hk

ABSTRACT

Second-order random walk is an important technique for graph analysis. Many applications use it to capture higher-order patterns in the graph, thus improving the model accuracy. However, the memory explosion problem of this technique hinders it from analyzing large graphs. When processing a billion-edge graph like Twitter, existing solutions (e.g., alias method) of the second-order random walk may take up 1796TB memory. Such high memory overhead comes from the memory-unaware strategies for node sampling across the graph.

In this paper, to clearly study the efficiency of various node sampling methods in the context of second-order random walk, we design a cost model, and then propose a new node sampling method following the acceptance-rejection paradigm to achieve a better balance between memory and time cost. Further, to guarantee the efficiency of the second-order random walk within arbitrary memory budgets, we propose a memory-aware framework on the basis of the cost model. The framework applies a cost-based optimizer to assign desirable node sampling method for each node in the graph within a memory budget while minimizing the time cost. Finally, we provide general programming interfaces for users to benefit from the memory-aware framework easily. The empirical studies demonstrate that our memory-aware framework is robust with respect to memory and is able to achieve considerable efficiency by reducing 90% of the memory cost.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '20, June 14–19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380562

CCS CONCEPTS

• **Theory of computation** → **Graph algorithms analysis**; • **Computing methodologies** → *Parallel algorithms.*

KEYWORDS

Random walk; Memory efficient; Graph algorithm; Large-scale

ACM Reference Format:
Yingxia Shao, Shiyue Huang, Xupeng Miao, Bin Cui, Lei Chen. 2020. Memory-Aware Framework for Efficient Second-Order Random Walk on Large Graphs. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3380562

1 INTRODUCTION

Random walk is widely used for graph analysis like proximity computation [17, 32, 38, 42], aggregate estimation [16, 24] and graph embedding [3, 10, 25, 35]. Most existing models use the first-order random walk [14], which assumes the next node to be visited only depends on the current node. However, the first-order random walk abstracts an over-simplification of real-world systems. An example is user trails on the Web [40], where nodes are Web pages and interactions are users navigating from one Web page to another. A user's next page visit not only depends on the last page but also is influenced by the sequence of previous clicks, which are called higher-order dependencies. The first-order random walk fails to capture such higher-order dependencies. Second-order random walk is designed to model the higher-order dependencies, thus improving the accuracy of applications. One popular and recent application of second-order random walk is to learn high-quality embeddings for graph analysis tasks, like node classification [1, 36], link prediction [18, 39]. Node2vec [9, 44] is one of the most successful graph embedding models. It uses the second-order random walk to encode nodes from the same community closely together and outperforms the deepwalk [25], a model using first-order random walk. Besides the graph embedding, second-order

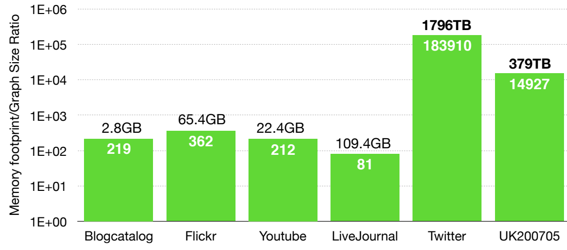


Figure 1: The ratio of total memory footprint to the memory size of the graph when running node2vec. The number at the top of each bar is the corresponding total memory footprint.

random walk is also successfully applied to find meaningful communities [2, 30] and model sequential data [38, 40].

Unlike the first-order random walk, the second-order one visits the next node depending on both the current node and the previous node. Given that the previous and current nodes are u and v, the probability of the next node z is $p(z|vu)$. We call p as *edge-to-edge* transition probability (*e2e* for short). Then for a graph $G(V, E)$, there are in total $|E|d$ different *e2e* probabilities, where d is the average degree of G. The management of $|E|d$ *e2e* probabilities is the bottleneck for running second-order random walks on large graphs.

A straightforward solution is to build the *e2e* distribution on demand, however, this approach has low efficiency and the sampling time cost of each node is linear to the node degree. To improve the sampling efficiency, current solutions store the whole distribution in memory with sophisticated data structures (e.g., alias table [37]). But this incurs the memory explosion problem [35, 44]. From Figure 1, we find that when running node2vec over a billion-edge graph like Twitter, which has about 41.6M nodes and 2.4B edges, current solutions require 1796TB memory that is 183910× larger than the graph size of Twitter, and it is really expensive to set up a machine/cluster to have that much memory. Even for much smaller graphs (e.g., Youtube, LiveJournal), the existing empirical studies [35] also demonstrate that node2vec encounters out-of-memory problem on a commodity server. It is obvious that current memory-intensive solutions are not cost-efficient approaches to process large graphs in the real world. Therefore, the research question is *how to maximize the efficiency of executing second-order random walk on large graphs within a memory budget.*

In this paper, we dissect the random walk operation into a sequence of node sampling, where a node generates samples from a discrete (non-)uniform distribution (e.g., *e2e* distribution). Then we design a cost model to reveal the trade-off between time and memory cost for different node sampling methods. Specifically, given a single node v, the node sampling method in the straightforward solution, called *naive method*, entails $O(d_v)$ time cost and $O(1)$ memory cost,

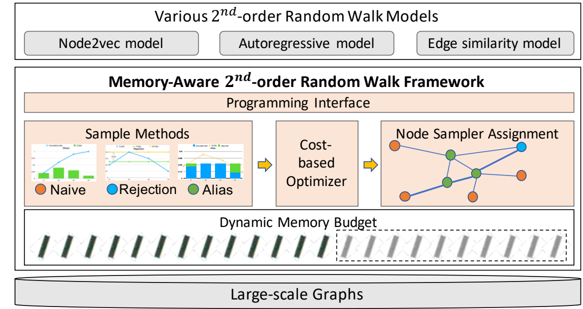


Figure 2: Overview of the memory-aware framework.

where d_v is the degree of node v. The aforementioned efficient node sampling with alias table, called *alias method*, entails $O(1)$ time complexity and $O(d_v^2)$ memory complexity. Inspired by the acceptance-rejection paradigm, we introduce a new sample method called *rejection method*. The rejection method uses the first-order random walk to sample node, accepts the sampled node with probability β, named acceptance ratio, and repeats the sampling until generating an accepted node. The rejection method entails $O(C_v)$ time complexity and $O(d_v)$ memory complexity. Here C_v is a model-related parameter and we prove C_v is smaller than d_v. Compared with the naive method and the alias method, rejection method achieves a better balance between memory and time cost.

From the above cost analysis, we find that these sampling methods have advantages. However, they are fragile and memory-unaware, i.e., they occupy fixed memory footprint, and cannot boost the efficiency when the available memory is larger than the required one or ensure the success of the task when available memory is insufficient. Therefore, we propose a memory-aware second-order random walk framework to guarantee the high efficiency on arbitrary memory budgets. The core of the framework is a cost-based optimizer. Based on the cost model, the optimizer assigns proper node sampling method for each node in the graph while minimizing the total time cost. The assignment problem is modeled as a multiple-choice knapsack problem [33], and we develop an effective greedy algorithm to generate the assignment in high efficiency. Further, considering the dynamic memory budgets in practice, we also introduce an adaptive version of the greedy algorithm, so that we can efficiently update the node sampling assignment when the memory budget changes including increase and decrease.

Finally, with our technical contributions, we carefully implement the memory-aware framework (Figure 2) as a middleware for various second-order random walk models. In this work, we implement two popular second-order random walk models – node2vec and autoregressive model [27] for the experiments. Further, to easily benefit from the aforementioned cost-based optimizer, the framework provides flexible

programming interfaces for users to define their own sample methods and the second-order random walk models. We evaluate the framework on six real-world large graphs including two billion-edge graphs. The empirical studies demonstrate that our memory-aware framework is robust with respect to memory and is able to achieve comparable efficiency by saving 90% of the memory cost which is required by the existing solutions.

We summarize our contributions as follows:

- We propose a memory-aware framework for second-order random walk on large graphs.
- We develop a cost-based optimizer to find a node sampler assignment in high efficiency.
- We introduce a rejection method for second-order random walk.
- We conduct extensive experiments on real-world datasets and show the superiority of our framework compared with the existing methods.

Organization. In Section 2, we present the backgrounds of this work. In Section **??**, we introduce the rejection method for second-order random walk and establish the cost model. Then we elaborate the cost-based optimizer, including node sampler assignment, theoretical analysis and its adaptive version in Section 5, followed by the introduction of the memory-aware framework in Section 5.4. In Section 6, we present the experimental results. Finally, we discuss the related work and give the conclusion in Section 7 and 8.

2 PRELIMINARY

A graph $G = (V, E)$ is defined by a set V of vertices and a set E of pairwise relations (edges) among vertices in V. We use v to represent a vertex in G, $N(v)$ to represent the neighborhood, and the degree of v is d_v. An edge is represented by (u, v) and the corresponding weight is w_{uv}.

2.1 Second-order Random Walk

Random walk on a graph is a stochastic process, that describes a walk that consists of a succession of random selected vertices. *First-order random walk* selects a vertex z based on the state of last vertex v, and it follows the transition distribution $p(z|v) = \frac{w_{vz}}{W_v}$, where $W_v = \sum_{t \in N(v)} w_{vt}$. *Second-order random walk* selects a vertex z based on the states of last two vertices u and v, and the transition distribution is $p(z|vu)$. For simplicity, the distribution $p(z|v)$ is called *node-edge* distribution (n2e for short), while the one $p(z|vu)$ is called *edge-edge* distribution (e2e for short).

The concrete e2e distribution is application dependent. Here we introduce two popular second-order random walk models: node2vec model and autoregressive model.

Algorithm 1 Second-order random walk generation procedure

Input: graph: $G(V, E)$, start node: v, length: *maxLen*
Output: random walk: L

```
1:  L = {v} // an array
2:  for t in 1..maxLen do
3:      if t == 1 then
4:          u_{t-1} = L[t-1]
5:          Node Sampler: draw node u_t from p(u_t|u_{t-1})
6:      else
7:          u_{t-1} = L[t-1], u_{t-2} = L[t-2]
8:          Node Sampler: draw node u_t from p(u_t|u_{t-1}, u_{t-2})
9:      end if
10:     L.append(u_t)
11: end for
12: return L
```

Node2vec model. As briefly mentioned before, node2vec is a network embedding algorithm using second-order random walk. The e2e distribution in node2vec defines the probability of walking from (u, v) to (v, z) as $p_{uvz} = \frac{w'_{vz}}{W'_v}$, where $W'_v = \sum_{t \in N(v)} w'_{vt}$, and w'_{vz} is defined as follows,

$$w'_{vz} = \begin{cases} \frac{w_{vz}}{a} & l_{uz} = 0 \\ w_{vz} & l_{uz} = 1 \\ \frac{w_{vz}}{b} & l_{uz} = 2 \end{cases} \tag{1}$$

where l_{uz} is the unweighted distance between vertices u and z, and a and b are two hyperparameters controlling the affinity of local structure. Both a and b are positive real values.

Autoregressive model. The autoregressive model [27] is used to compute second-order pagerank [38]. The e2e distribution is $p_{uvz} = \frac{p'_{uvz}}{\sum_{t \in N(v)} p'_{uvt}}$, where $p'_{uvz} = (1-\alpha)p_{vz} + \alpha p_{uz}$, $p_{vz} = \frac{w_{vz}}{W_v}$, and $0 \leq \alpha < 1$.

Note that in addition to the two models introduced above, there are many other second-order random walk models, such as the one utilizing network flow data [38], edge similarity model [19]. In this paper, we concentrate on the two models discussed here as representatives.

2.1.1 Node Sampler.

Algorithm 1 lists the general procedure of producing a second-order random walk on graph G. From the algorithm, it is easy to see that the core operation is the selection of a node from the neighborhood (Lines 5 and 8). We name it as *node sampler* in this paper. The main logic of node sampler is to apply different sampling methods to sample nodes. The efficiency of node sampler is heavily related to the sampling methods.

2.2 Three Sampling Methods

Given a discrete probability distribution $P = \{p_i\}, i = 1..n$, where p_i is the probability of element i, and n is the number of elements, the sampling method is to draw an element x

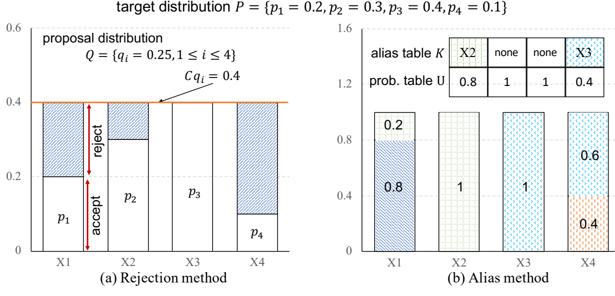


Figure 3: The examples of using rejection method and alias method to sample a target distribution P.

on the basis of distribution P, i.e., $P[x = i] = p_i$. There are many efficient sampling methods to realize the nonuniform sampling. In this paper, we focus on naive method, rejection method, and alias method.

Naive method [22]. This method generates a uniform [0, 1] random variable r, and put $x = i$ if

$$\sum_{k=1}^{i-1} p_k < r \le \sum_{k=1}^{i} p_k \ (p_0 = 0). \tag{2}$$

We can apply various search techniques (e.g., linear search, binary search, etc.) to locate the position of r in the cumulative distribution of P. The sampling time complexity is $O(n)$ or $O(log(n))$ while the memory complexity is $O(n)$.

Rejection method [28]. This method samples an element i from the target distribution P by instead sampling from a proposal distribution $Q = \{q_i\}$ and accepting the outcome from Q with probability $\frac{p_i}{Cq_i}$, repeating the draws from Q until an outcome from Q is accepted. Usually the proposal distribution Q is chosen by user and it is easier to sample than the target distribution P. In addition, C is called *bounding constant*, and satisfies $p_i \le Cq_i$ for all values of i. The average time complexity of rejection sampling is $O(C)$[1].

Figure 3(a) shows an example of using rejection method to sample a target distribution $P = \{0.2, 0.3, 0.4, 0.1\}$. We define the proposal distribution Q to be the uniform distribution, i.e., $Q = \{0.25, 0.25, 0.25, 0.25\}$, and set C to be 1.6. Then the acceptance probabilities of four elements $X1, X2, X3, X4$ are $0.5, 0.75, 1, 0.25$ respectively.

Alias method [37]. Alias method converts target distribution into a uniform distribution over (possibly degenerate) binary outcomes. Since the target distribution is nonuniform, it means that some elements have more than $\frac{1}{n}$ probability, and some have less. For each element having less than $\frac{1}{n}$, alias method constructs an alias-outcome by combining it with a small portion of some probability mass from one of the higher-probability elements, topping up the total probability to $\frac{1}{n}$. Internally, alias method maintains two tables, a probability table $U = \{u_i\}$ and an alias table $K = \{k_i\}$ (for $1 \le i \le n$), where u_i is the probability of choosing element at index i as an outcome and k_i is the alias-outcome with respect to i. To sample a value, it first generates a uniform [1, n] random variable x to determine an index into the two tables. Then it generates a [0, 1] random variable r, and on the basis of the probability u_x, the final outcome is $\begin{cases} x & r \le u_x, \\ k_x & otherwise. \end{cases}$

Figure 3(b) illustrates the probability table and alias table when using alias method to sample the target distribution $P = \{0.2, 0.3, 0.4, 0.1\}$. The algorithm typically entails $O(n)$ time complexity to build the two tables, after which random elements can be drawn from the distribution in $O(1)$ time complexity.

[1]http://www.columbia.edu/~ks20/4703-Sigman/4703-07-Notes-ARM.pdf

3 SECOND-ORDER RANDOM WALK WITH REJECTION

In this section, we introduce the rejection method for second-order random walk. Considering that the time cost of rejection method is related to the bounding constant, we also empirically and theoretically analyze the distribution of the bounding constants, and introduce a simple but effective solution for bounding constant estimation.

3.1 Rejection Method for Node2vec and Autoregressive Models

According to Section 2.2, rejection method consists of target distribution P, proposal distribution Q, and bounding constant C. In the context of second-order random walk, assume the previous and current nodes are u and v, then P is $\{\frac{w'_{vz}}{W'_v}\}$, Q is $\{\frac{w_{vz}}{W_v}\}$. According to the condition $p_i \le Cq_i$, we compute the bounding constant C_{uv} of an edge (u, v) as follows,

$$C_{uv} = max\{\frac{P}{Q}\} = max_{z \in N(v)}\{\frac{w'_{vz}}{w_{vz}}\frac{W_v}{W'_v}\}. \tag{3}$$

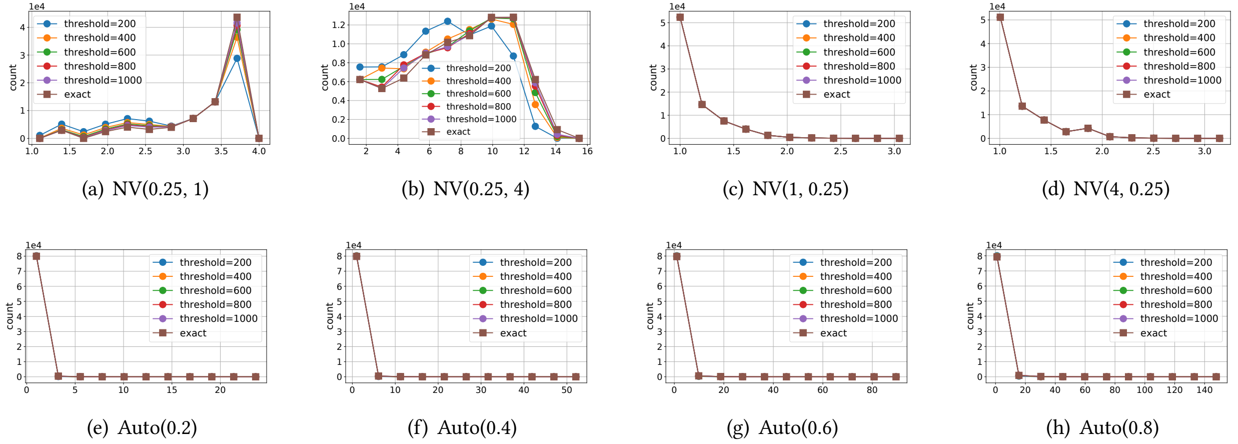
During the sampling, the acceptance ratio β_{uvz} is

$$\beta_{uvz} = \frac{1}{C_{uv}}\frac{w'_{vz}}{w_{vz}}\frac{W_v}{W'_v} = \frac{w'_{vz}}{w_{vz}} min_{t \in N(v)}\{\frac{w_{vt}}{w'_{vt}}\}. \tag{4}$$

Rejection for node2vec. According to the Equation 1, assume the previous node u and current node v have common neighbors, the ratio $\frac{w'_{vz}}{w_{vz}}$ belongs to $\{\frac{1}{a}, \frac{1}{b}, 1\}$. Then the bounding constant is $C_{uv} = \frac{W_v}{W'_v} max\{\frac{1}{a}, \frac{1}{b}, 1\}$. The acceptance ratio for a vertex z is $\beta_{uvz} = \frac{w'_{vz}}{w_{vz}} min\{1, a, b\}$.

Rejection for Autoregressive model. Similarly, C_{uv} is

The acceptance ratio β_{uvz} is $\frac{(1-\alpha)+\alpha\frac{p_{uz}}{p_{vz}}}{max_{t \in N(v)}\{(1-\alpha)+\alpha\frac{p_{ut}}{p_{vt}}\}}$.



(a) NV(0.25, 1) (b) NV(0.25, 4) (c) NV(1, 0.25) (d) NV(4, 0.25)

(e) Auto(0.2) (f) Auto(0.4) (g) Auto(0.6) (h) Auto(0.8)

Figure 4: The bounding constant distribution on Flickr. We visualize the distributions of exact bounding constants and estimated bounding constants with five different thresholds. The range of the bounding constants (x-axis) is uniformly divided into 10 buckets, i.e., $\frac{max-min}{10}$. NV(0.25, 1) means $a = 0.25$ and $b = 1$ in node2vec model, and Auto(0.2) means $\alpha = 0.2$ in autoregressive model.

3.2 The Distribution of Bounding Constants

The following theorem proves that the bounding constants for any edge in an unweighted graph is bounded and cannot be arbitrary large. The conclusion can be extended to the weighted graph with more complex analysis.

THEOREM 1. *Given an edge (u, v) in an unweighted graph, the bounding constant C_{uv} in node2vec and autoregressive model is bounded. Specifically, let θ_{uv} be the number of common neighbors of (u, v) ($\theta_{uv} \geq 1$), then the bound of C_{uv} of node2vec is*

$$C_{uv} \leq \begin{cases} \frac{d_v}{\theta_{uv}} & a \geq 1,\ b \geq 1, \\ d_v & 0 < a < 1,\ b \geq a, \\ \frac{d_v}{d_v-1-\theta_{uv}} & 0 < b < 1,\ a \geq b. \end{cases}$$

The one of autoregressive model is $C_{uv} \leq \frac{d_v}{\theta_{uv}}$.

PROOF. *1) The bound of C_{uv} in node2vec. In an unweighted graph, $W_v = d_v$ and the n2e transition probability is $\frac{1}{d_v}$. Then C_{uv} is $\frac{d_v}{\theta_{uv}+\frac{1}{a}+\frac{d_v-1-\theta_{uv}}{b}} max\{\frac{1}{a}, \frac{1}{b}, 1\}$.*

Next we give the bound with different a and b.

Case 1 ($a \geq 1$, $b \geq 1$): $C_{uv} = \frac{d_v}{\theta_{uv}+\frac{1}{a}+\frac{d_v-1-\theta_{uv}}{b}} \leq \frac{d_v}{\theta_{uv}}$.

Case 2 ($1 > a > 0$, $b \geq a$): $C_{uv} = \frac{d_v}{a\theta_{uv}+1+\frac{a(d_v-1-\theta_{uv})}{b}} \leq d_v$.

Case 3 ($1 > b > 0$, $a \geq b$): $C_{uv} = \frac{d_v}{b\theta_{uv}+\frac{b}{a}+(d_v-1-\theta_{uv})} \leq \frac{d_v}{d_v-1-\theta_{uv}}$.

2) The bound of C_{uv} in autoregessive model. Let $p_{vz} = \frac{1}{d_v}$ and $p_{uz} = \frac{1}{d_u}$, then C_{uv} is $C_{uv} = \frac{(1-\alpha)+\alpha\frac{d_v}{d_u}}{(1-\alpha)+\alpha\frac{\theta_{uv}}{d_u}} \leq \frac{d_v}{\theta_{uv}}$.

Discussion. There are two special cases (i.e., $\theta_{uv} = 0$ and $d_v - 1$)) for the bounds. When $\theta_{uv} = 0$, for the case 1 of node2vec, the maximum factor becomes $max\{\frac{1}{a}, \frac{1}{b}\}$, then C_{uv} is still bounded by d_v. For autoregressive model, $C_{uv} = 1$. When θ_{uv} is d_v-1, the maximum factor of node2vec becomes $max\{\frac{1}{a}, 1\}$ and it degenerates to case 1 or case 2. The autoregressive model is not affected.

According to Theorem 1, the bound of C_{uv} is proportional to the node degree in the worst case, and the time complexity of the rejection method will be linear to the degree. In real-world graphs with standard parameter settings, C_{uv} is much smaller than the degree actually. Let us denote the average bounding constant of v as C_v, i.e., $C_v = \frac{1}{d_v}\sum_{u\in N(v)} C_{uv}$. The brown line in Figure 4 shows the distribution of exact C_v on Flickr with standard parameter settings. For node2vec model, the most frequent values of C_v are below 10, and the autoregressive model has a similar trend. However, the maximal C_v of the autoregressive model is larger than the one of node2vec, this implies that the autoregressive model is more easily to be close to the bound defined in Theorem 1.

3.3 The Computation of Bounding Constants

As introduced in Section 2.2, the efficiency of rejection method is determined by the bounding constant. Therefore, we discuss how to compute (or estimate) the bounding constant in the context of second-order random walk. According to Equation 3, the exact C_v can be computed by enumeration which entails $O(d_v^2)$ time complexity for node v. For the whole graph, the total time cost has the same complexity as the one of triangle counting [15], which is expensive. We resort to computing approximate ones for improving the efficiency of bounding constants computation.

On the basis of time complexity $O(d_v^2)$, we notice that nodes with higher degree incur larger time cost. Therefore, we estimate the bounding constants via sampling when the node degree exceeds a predefined threshold D_{th}. Formally, we estimate the C_{uv} as $C_{uv} = max_{z \in SN(v)}\{\frac{w'_{vz}}{w_{vz}}\frac{W_v}{W'_v}\}$, where $SN(v)$ is a uniformly sampled neighborhood of v with size D_{th}. Then the estimation time cost is $O(d_v D_{th})$.

We empirically study the effectiveness of the above estimation. Figure 4 shows the distributions of estimated bounding constants on Flickr by setting $D_{th} = [200, 400, 600, 800, 1000]$. The results demonstrate that a relatively small threshold (e.g., 600) can achieve a good estimation. In addition, the above discussion assumes the sample size equals to the threshold. Theoretically, on the basis of the law of large numbers [8], the proposed method can achieve the exact estimation when the sample size is large enough.

4 COST ANALYSIS OF NODE SAMPLER

In this section, we analyze the time and memory cost of three node samplers and introduce a cost model in the context of second-order random walk.

Without loss of generality, we assume that a probability value is stored in b_f bytes and a node id is stored in b_i bytes. The unit of time cost is denoted by K. d_{max} is the maximum node degree in a graph. In addition, in second-order random walk models, the computation of biased weight is related to the existence of common neighbors, and we assume the cost of this operation is c, which is related to the node degree.

4.1 The Cost of Node Samplers

Naive Node Sampler. According to Equation 2, naive node sampler needs the cumulative distribution for sampling, and it builds the distribution on the basis of edge weights.

1) Memory cost: The sampler uses an array of length d_v to store probabilities for a node. Across the whole graph, we only need to allocate a single array of length d_{max}. In the view of a node, the average memory cost is $\frac{b_f d_{max}}{|V|}$ bytes.

2) Time cost: The time cost consists of the distribution building cost and sampling cost. Assume linear search is used to locate the index, the sampling cost of node v is $d_v K$. To build the distribution, for each neighborhood, we need to check the existence of the edge between the previous node and next node, therefore, the cost is $d_v cK$. In total, the time cost is $d_v(c+1)K$.

Rejection Node Sampler. Generally, the rejection node sampler follows Equations 3 and 4.

1) Memory cost: Considering that the start node of second-order random walk and successive sampled nodes for rejection are both sampled from the n2e distribution (proposal distribution Q), we use alias method which entails $(b_f + b_i) \times d_v$ bytes memory cost. Besides, to compute the acceptance ratio with regard to each edge (u, v), we need to store a factor $min_{t \in N(v)}\{\frac{w_{vz}}{w'_{vz}}\}$. There are total d_v different factors, the storage is $b_f \times d_v$. In total, for a node v, the memory cost is $d_v \times (b_f + b_i) + d_v \times b_f$ bytes.

2) Time cost: The sampling time complexity of rejection node sampler is the average bounding constant $O(C_v)$, and then the cost is $C_v K$. In addition, for each sample, we need to compute w' to obtain the acceptance ratio, which relies on checking the existence of the edge between the previous node and next node, resulting total C_v numbers of edge existence checking. Therefore, the total time cost is $C_v cK$.

Alias Node Sampler. As mentioned before, the alias method requires two tables for a single distribution. One stores probabilities and the other is store alias-outcomes (i.e., node ids).

1) Memory cost: Given a node v, there are d_v e2e distributions, thus entailing $d_v \times (b_f + b_i) \times d_v$ bytes memory cost. Furthermore, according to Algorithm 1, to sample the start node for a second-order random walk (i.e., $t = 0$), we need to sample it from the n2e distribution, then the alias method also incurs another two tables. Overall, the node v entails $(d_v + 1) \times d_v \times (b_f + b_i)$ bytes memory cost.

2) Time cost: Since the sampling performance of the alias method is $O(1)$, the alias node sampler can generate a sample in constant time, which is K.

Node Sampler	Memory Cost (M)	Time Cost (T)
Naive	$\frac{b_f d_{max}}{\|V\|}$	$d_v(c+1)K$
Rejection	$(2b_f + b_i)d_v$	$C_v cK$
Alias	$(b_f + b_i)(d_v^2 + d_v)$	K

Table 1: Cost comparison of a node sampler by using three different sampling methods for second-order random walk. Notice that C_v is the average bounding constant of node v. b_f and b_i are the bytes of storing a non-integer and integer value respectively.

4.2 Cost Model

We summarize the above cost analysis as the cost model listed in Table 1 for second-order random walk. We can instantiate a cost model by determining the data types and the complexity of checking the existence of common neighbors. For example, assuming that the probability is stored in *float* data type, the node id is stored in *integer* data type, and the binary search is used for common neighbor checking, then $b_f = 4$, $b_i = 4$ and $c = log(d_v)$.

From the table, we figure out that the more memory used the better time efficiency achieved. Concretely, we denote the memory costs of naive node sampler, rejection node sampler and alias node sampler by M_n, M_r and M_a respectively. Similarly, the time costs are T_n, T_r and T_a. According to Theorem 1, $1 \leq C_v \leq d_v$ holds. Then we have the following

two orders: $M_a > M_r > M_n$, and $T_a < T_r < T_n$. Therefore, in the ideal scenario where memory is unlimited or large enough, we simply use alias node sampler to achieve the best efficiency. However, as introduced in the Introduction, this incurs the memory explosion problem in practice. To smartly utilize the memory, we turn to assign different samplers for each node, so that maximize the efficiency without causing memory explosion. Finally, we propose a memory-aware second-order random walk framework.

5 MEMORY-AWARE SECOND-ORDER RANDOM WALK FRAMEWORK

Figure 2 illustrates the overview of the memory-aware second-order random walk framework. The core of the framework is a cost-based optimizer. The optimizer guarantees the high-efficiency of second-order random walk on various memory budgets. It not only greedily finds an efficient node sampler assignment from scratch, but also is able to fast update the assignment when the memory budget changes online.

During the execution, the framework 1) first initializes the cost model and computes the bounding constants for the rejection node sampler. 2) Then it executes the cost-based optimizer to generate an efficient node sampler assignment without violating the memory constraint. 3) On the basis of the assignment, it initializes each node sampler across the graph. 4) Finally, the framework is ready to do the second-order random walk. In addition, the framework provides flexible programming interfaces for users to define new samplers and application-oriented random walk models. Thus, the framework can be a middleware in existing second-order random walk based applications to improve the sampling efficiency.

5.1 Cost-based Optimizer

The responsibility of the cost-based optimizer is to find a node sampler assignment in which the sampling efficiency is maximized while the memory footprint is constrained by the given budget. To formally analyze the node sampler assignment problem, we define it as follows:

Definition 1. *(Node Sampler Assignment Problem). Given a graph $G = (V, E)$ and a set of node samplers $NS = \{ns_j\}$, $1 \le j \le S$, the cost model is $CM = \{(T_{ij}, M_{ij})\}$, where T_{ij} is the time cost of node i using sampler ns_j and M_{ij} is the corresponding memory cost, the goal of the problem is to assign a sampler to each node in graph G such that the total time cost is minimized and the memory cost does not exceed a predefined memory budget M. The formal objective is*

$$minimize \sum_{i=1}^{|V|} \sum_{j=1}^{j=S} T_{ij} x_{ij}, \tag{5}$$

$$subject\ to \sum_{i=1}^{|V|} \sum_{j=1}^{j=S} M_{ij} x_{ij} \le M, \tag{6}$$

$$\sum_{j=1}^{j=S} x_{ij} = 1, i = 1, 2, ..., |V|, \tag{7}$$

$$x_{ij} = \{0, 1\}, i = 1, 2, ..., |V|; j = 1, 2, ..., S. \tag{8}$$

The following theorem shows that the above problem is a 0-1 Multiple-Choice Knapsack Problem (MCKP).

Theorem 2. *The node sampler assignment problem is a 0-1 Multiple-Choice Knapsack Problem [33].*

Proof. *Let M_{max} be the maximum memory consumption among all M_{ij}, i.e., $M_{max} = max_{1 \le i \le |V|; 1 \le j \le S}\{M_{ij}\}$.*

We define $M^*_{ij} = M_{max} - M_{ij}$, then Equation 6 becomes $\sum_{i=1}^{|V|} \sum_{j=1}^{j=S} M^*_{ij} x_{ij} \ge |V| * M_{max} - M$.

According to the definition of 0-1 MCKP [33, 41], the assignment problem with the new memory constraint is an exact 0-1 MCKP.

In this paper, we have defined three node samplers, and they are alias node sampler, rejection node sampler, and naive node sampler. The cost model is listed in Table 1. Users can further extend the node sampler set by defining new samplers on the basis of our flexible programming interface which is introduced in Section 5.4.

5.2 Greedy Algorithms of Node Sampler Assignment

The original 0-1 MCKP is an NP-hard problem. It has pseudo-polynomial algorithm which is a dynamic programming solution [5], however, the dynamic programming method entails high time complexity $O(|V|M)$ and cannot process large graphs and large memory budgets. Here we focus on greedy algorithms which are often efficient for the big data scenarios.

5.2.1 LP Greedy Algorithm. To solve the 0-1 MCKP, a classical solution is to transform the original problem into linear MCKP (LMCKP) by relaxing the constraint (8) and allowing x_{ij} to be real value. The optimal solution of LMCKP is the lower bound for the 0-1 MCKP. Then the original problem is solved to optimality through enumeration [6], or finds the approximation by rounding [33]. Here we use the rounding technique to achieve an efficient node sampler assignment and theoretically analyze its approximation bound.

Properties of LMCKP. We restate two properties of the optimal solution for LMCKP in the context of graph.

Property 1. ***P-Domination**. For any node $v_i \in V$, if $T_{ij} \ge T_{ik}$ and $M_{ij} \ge M_{ik}$, then $x_{ij} = 0$ in the optimal solution.*

Algorithm 2 LP Greedy Algorithm

Input: Graph: $G(V, E)$, Cost Model: $CM = \{(T_{ij}, M_{ij})\}$, Memory Budget: M
Output: Assignment: *assignDict*, Trace: path

1: $usedMem = 0$
2: $path = []$
3: **for** each v_i in V **do**
4: eliminate P-Domination and LP-Domination of node samplers of v_i according to Property 1 and 2.
5: assign node sampler with smallest memory cost (NS_{i1}) to node v_i, i.e., $assignDict[v_i] = NS_{i1}$
6: $usedMem = usedMem + M_{i1}$
7: **end for**
8: compute the gradients $\frac{T_{ij+1}-T_{ij}}{M_{ij+1}-M_{ij}}$ for each node.
9: sort the gradient array q in ascending order.
10: **while** q.notEmpty() **do**
11: $NS_{ik} = (T_{ik}, M_{ik})$, i.e., select the minimal $\frac{T_{ij+1}-T_{ij}}{M_{ij+1}-M_{ij}}$ from q.
12: $(T_i, M_i) \leftarrow assignDict[v_i]$
13: **if** $usedMem - M_i + M_{ik} > M$ **then**
14: break;
15: **end if**
16: $assignDict[v_i] = NS_{ik}$
17: $usedMem = usedMem - M_i + M_{ik}$
18: $path.append(NS_{ik})$ //maintain the assignment trace for update.
19: **end while**

Proof. *Refer to the proof of Proposition 1 in the work [33].*

Property 2. ***LP-Domination**. For every node $v_i \in V$, if $T_{ir} \geq T_{is} \geq T_{it}$, $M_{ir} \leq M_{is} \leq M_{it}$, and $\frac{T_{is}-T_{ir}}{M_{is}-M_{ir}} > \frac{T_{it}-T_{is}}{M_{it}-M_{is}}$, then $x_{is} = 0$.*

Proof. *Refer to the proof of Proposition 2 in the work [33].*

On the basis of the two properties, we assume the cost model $CM = \{(T_{ij}, M_{ij})\}$ on every node are ordered as follows: $T_{i1} \geq T_{i2} \ldots \geq T_{iS}, M_{i1} \leq M_{i2} \ldots \leq M_{iS}$, and there is no LP-Domination. For any cost pair violates the above condition, we can set the corresponding x to be zero. Fortunately, the cost model in Table 1 satisfies the condition. In general cases, a set of user-defined node samplers may not always hold the condition, i.e., there are P-Domination or LP-Domination samplers. Notice that the samplers which are not dominated are the lower convex boundary among all the user-defined samplers [26], therefore, for each node, we can find these undominated samplers by ordering them in increasing memory costs and successively testing the sampler according to Property 1 and 2. Next we give the theorem about the optimality of LMCKP.

Theorem 3. *An optimal solution of LMCKP has at most two fractional variables, which are from the same node, i.e., two variables are x_{ij} and x_{il}. Furthermore, the variables are adjacent, i.e., $l = j + 1$.*

Proof. *Refer to the proof of Proposition 3 in the work [33].*

Solution. LP greedy algorithm uses LMCKP solver to get the optimal solution of LMCKP, then it rounds up the solution, i.e., assume x_{ij} and x_{ij+1} are two factional variables, it sets $x_{ij} = 1$ and $x_{ij+1} = 0$ for approximation. Algorithm 2 lists the main procedure. First for each node, it eliminates the P-Domination and LP-Domination in the input cost model, and assigns the node the most time expensive node sampler, e.g., NS_{i1}. Then it selects the minimal $\frac{T_{ij+1}-T_{ij}}{M_{ij+1}-M_{ij}}$ to update the assignment, which indicates the most profitable changes currently. When the memory budget runs up, the algorithm finishes and does the approximation implicitly.

More concretely, we present an example of using LP greedy algorithm on a toy undirected graph, which has 4 nodes and 4 edges. The node sampler set consists of naive, rejection and alias node samplers and the memory budget is 188 Bytes. Figure 5 visualizes the key intermediate statistics of the example including the corresponding cost model and the sorted gradients for each node. First, when LP greedy algorithm finishes the initialization (Lines 3-7 in Algorithm 2), each node is assigned naive node sampler and the *usedMem* is 12. Then it selects the minimal gradient -0.114, changes node sampler of node 3 from naive to rejection, and the *usedMem* becomes $12-3+24 = 33$. This greedy procedure (Lines 10-19) does not stop until the memory budget is reached. In this example, when node 2 is determined to use alias method by the algorithm, the *usedMem* reaches 144 and the remained memory budget $188 - 144 = 44$ cannot support any further node sampler update. Therefore, the final node sampler assignment is that nodes 0 and 1 use rejection node sampler and nodes 2 and 3 use alias sampler.

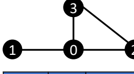
Analysis of approximation bounds. LP greedy algorithm does not guarantee to find the optimal solution of MCKP. But the following theorem shows that the LP greedy algorithm cannot be arbitrary worse with regard to the second-order random walk model.

Theorem 4. *Given a graph G, and the three node samplers are used, if the optimal solution of MCKP is denoted by OPT, the solution of LP greedy algorithm is A, then $OPT \leq A \leq max\{\frac{c+1}{c}, c\}d_{max}OPT$, where d_{max} is the maximum degree of G and c is the cost of checking common neighbors of an edge which is degree dependent.*

Proof. *First, it is obviously $OPT \leq A$ because of the approximation of A.*

Second, let OPT^{lp} be the optimal solution of LMCKP, since $OPT^{lp} \leq OPT$, we prove $A \leq max\{\frac{c+1}{c}, c\}d_{max}OPT^{lp}$ instead of $A \leq max\{\frac{c+1}{c}, c\}d_{max}OPT$.

Model: NV(0.25, 4)
Cost Model:
- $c = 1$
- $b_f = 4, b_i = 4$
Memory Budget: 188



Cost Model Table

vid	d_v	C_v	NS_1(Naïve)		NS_2 (Rejection)		NS_3(Alias)	
			M_{i1}	T_{i1}	M_{i2}	T_{i2}	M_{i3}	T_{i3}
0	3	2.41	3.0	6.0	36.0	2.41	96.0	1.0
1	1	1.00	3.0	2.0	12.0	1.00	16.0	1.0
2	2	1.60	3.0	4.0	24.0	1.60	48.0	1.0
3	2	1.60	3.0	4.0	24.0	1.60	48.0	1.0

Seq.	vid	Grad.	Sampler	Mem.	Seq.	vid	Grad.	Sampler	Mem.
0	3	-0.114	N->R	33	4	3	-0.025	R->A	120
1	2	-0.114	N->R	54	5	2	-0.025	R->A	144
2	1	-0.111	N->R	63	6	0	-0.024	R->A	204
3	0	-0.109	N->R	96	7	1	0	R->A	208

Sorted gradients and node sampler update sequence. N=Naïve, R=Rejection, A=Alias.

Figure 5: A concrete example of applying LP greedy algorithm on a toy graph. The memory budget is 188, and the final node sampler assignment is {0:R, 1:R, 2:A, 3:A}. The top table describes the cost model while the bottom table lists the sorted gradients where the rows in red color are the update log by LP greedy algorithm within the memory budget. Notice that four nodes are initialized with naive node samplers, the total used memory is $4 \times 3 = 12$ at the beginning.

Assume the node v_k has two fractional variables $x_{kj} = \lambda$ and $x_{kj+1} = 1 - \lambda$ in OPT^{lp}, then $OPT^{lp} = \sum_{i=1, i \neq k}^{|V|} \sum_{l=1}^{S} T_{il} + \lambda T_{kj} + (1-\lambda) T_{kj+1}$, and $A = \sum_{i=1, i \neq k}^{|V|} \sum_{l=1}^{S} T_{il} + T_{kj}$.

Let $D = \sum_{i=1, i \neq k}^{|V|} \sum_{l=1}^{S} T_{il}$, and $T_{kj} \geq T_{kj+1}$, we have

$$\frac{A}{OPT^{lp}} = \frac{D + T_{kj}}{D + \lambda T_{kj} + (1-\lambda) T_{kj+1}} \leq \frac{D + T_{kj}}{D + T_{kj+1}} \leq \frac{T_{kj}}{T_{kj+1}}$$
$$\leq max\{\frac{T_n}{T_r}, \frac{T_r}{T_a}\} = max\{\frac{d_{v_k}(c+1)}{C_{v_k} c}, C_{v_k} c\},$$

where T_n, T_r, T_a are the time cost of naive, rejection and alias node samplers respectively.

Finally, based on the Theorem 1 and $C_v \geq 1$, it is easy to figure out $max\{\frac{d_{v_k}(c+1)}{C_{v_k} c}, C_{v_k} c\} \leq max\{\frac{c+1}{c}, c\} d_{max}$. Therefore, the above theorem holds.

The above theorem gives the upper bound $max\{\frac{c+1}{c}, c\} d_{max}$ of approximation between LP greedy algorithm and the optimal one. The upper bound is not only related to the graph, but also influenced by the algorithm for common neighbor checking. When binary search is used for checking, c is $log(d_v)$, the upper bound is $d_{max} log(d_{max}) (d_{max} \geq 4)$; when hashset is used for checking, c is one, the upper bound becomes $2d_{max}$. Although the upper bound is loose and graph-related, in practice, our empirical studies in Section 6 showcase the high-efficiency of the assignment generated by the LP greedy algorithm for real-world graphs. Another intuitive explanation is that existing empirical studies [33] demonstrated that the search of P-domination in the class where fractional variables exist would lower the cost, while the search among other classes will not lower the cost. For the second-order random walk with the aforementioned three node samplers, there is actually no P-domination in the same class. Therefore, the output of the LP greedy algorithm is (close to) the optimal solution in high-probability.

5.2.2 Degree-based Greedy Algorithm. An intuition for solving the node sampler assignment problem is to use the alias node sampler as many as possible. From the cost model, we have two following observations: 1) The node with larger degree usually achieves more efficiency improvements when changing node sampler from rejection/naive to alias. This motivates us to assign nodes with large degree the alias node sampler first. 2) The node with smaller degree occupies less memory resulting more nodes can use alias node sampler within a memory budget. This motivates us to assign nodes with small degree the alias node sampler first.

On the basis of the above observations, we introduce a basic degree-based greedy algorithm, which works as follows: first, the nodes in the graph are sorted by degree in increasing (decreasing) order; then for each node, within the memory budget, it tries to assign node samplers in alias, rejection and naive order.

Compared with the LP greedy algorithm, the degree-based solution is much simpler. However, it does not consider the time cost explicitly and is hard to obtain an approximation bound. The experimental results will demonstrate that the degree-based algorithms only work well when the memory budget is large.

5.3 Adaptive Solution for Dynamic Memory Budget

In the real-world environment, e.g., cloud service, public clusters, the available memory is dynamic. This requires that the node sampler assignment can be adaptively adjusted when the memory budget changes. Due to the linear property of greedy algorithms, we can easily extend the proposed solution to be adaptive.

During the original greedy process, we maintain the trace of greedy options (Line 18 in Algorithm 2). When the memory budget changes, we execute the following two strategies to update the node sampler assignment.

Memory budget increase. When the available memory increases, the previous assignment is not affected by the new total budget, so we just restart the greedy algorithm from the last state in the trace.

Memory budget decrease. In this case, we need to revoke some node assignments to reduce the total memory. Since the original greedy algorithm does the greedy choice in increasing order in terms of the memory size, we execute the reverse order to reduce the used memory. Specially, we pop the greedy choices from the trace until the total memory satisfying the new budget.

```
class NodeSampler {
    virtual int sample(int nid) = 0;
    virtual float timeCost(int nid) = 0;
    virtual float memCost(int nid) = 0;
};
class SecondRandomWalker {
    virtual float biasedWeight(int u, int v, int z) = 0;
};
```

Figure 6: Programming interfaces of the memory-aware second-order random walk framework.

5.4 Framework Implementation

We carefully implemented the memory-aware framework in C++. The graph is organized as CSR format [29]. Both the cost-based optimizer and the random walk model run in a vertex-centric manner. We parallelize the computation at node level by using OpenMP library.

Programming interface. To enable users to easily benefit from the cost-based optimizer, we develop two programming interfaces for the framework – NodeSampler interface and SecondRandomWalker interface, which are illustrated in Figure 6. The NodeSampler defines the sampling procedure from a discrete distribution with corresponding time and memory cost. The SecondRandomWalker defines the logic of computing biased edge weight for an interested second-order random walk.

Optimization of the cost-based optimizer. During the initialization phase, the cost-based optimizer needs bounding constant C_v to compute the time cost of rejection method. Since the exact computation of C_v is expensive and the optimizer does not require the exact time cost, we use the bounding constant estimation (Section 3.3) to obtain approximate C_v for improving the efficiency of initialization.

6 EXPERIMENTS

6.1 Experimental Settings

Without clarification, the experiments are run on a cloud server, which equips with a 64-bit 24-core CPU, 96GB memory and 2TB hard disk. The operating system is Ubuntu 18.04 64bits. The default parallelism of the framework is set to 16.

Datasets. We use six public datasets: Blogcatalog, Flickr, Youtube, LiveJournal, Twitter, UK200705. All the graphs are processed into undirected. Table 2 lists the statistics of datasets. The memory size M_g of each graph is collected during the runtime from */proc/<pid>/statm* file.

Compared methods. The proposed memory-aware second-order random walk framework has four variants with different greedy algorithms and optimizations. *LP-std* is the solution with LP greedy algorithm. *LP-est* is the solution using the optimization of bounding constant estimation, and the default threshold is set to 600. *Deg-inc* and *Deg-dec* are two variants using degree-based greedy algorithm in increase and decrease order respectively. In addition, the memory-unaware second-order random walk with three basic node samplers are compared, and they are called naive, rejection, and alias for simplicity.

G	|V|	|E|	d_{avg}	M_g
Blogcatalog	10.3K	668K	64.8	13MB
Flickr	80.5K	11.8M	146.6	185MB
Youtube	1.1M	6.0M	5.3	108MB
LiveJournal	4.8M	86.2M	17.8	1,375MB
Twitter	41.6M	2.4B	39.1	~10GB
UK200705	105.9M	6.6B	62.6	~26GB

Table 2: Dataset statistics. M_g is the memory size of the graph.

Second-order random walk models. Node2vec and autoregressive models are used.

1. The node2vec model is controlled by two hyperparameters a and b, denoted by $NV(a, b)$. We follow the original work [9] and set $a, b \in [0.25, 1, 4]$.
2. The autoregressive model is controlled by one hyperparameter α, denoted by Auto(α). We set the values of α to be $[0, 0.2, 0.4, 0.6, 0.8]$.

Benchmarks. To evaluate the efficiency of the proposed solution, we choose two different benchmarks.

1. Node2vec Random Walk: Following the random walk sampling pattern in node2vec, every node in a graph samples a set of random walks with a fixed length. Here we use the same parameter settings from node2vec model [9], and sample 10 walks per node with walk length of 80.
2. Second-order PageRank query: Wu et al. [38] propose the second-order PageRank query. Given a query node v, it runs the second-order random walk with restart to estimate the pagerank. The decay factor is 0.85, the maximum length is 20, and the total sample size is 4|V|. In addition, we randomly choose 100 query nodes for each dataset.

The node2vec random walk task is run over the node2vec model, while the second-order PageRank query is run over the autoregressive model.

Evaluation Metrics. We compare the elapsed time (seconds) and memory footprint among different methods. The elapsed time is further decomposed into initialization time T_{init} and sampling time T_s. For the memory-aware second-order random walk framework, the T_{init} includes the cost of computing the bounding constants T_{C_v} and the cost of initializing node sampler related data structure T_{NS}. In other words, the initialization time is

$$T_{init} = \begin{cases} T_{C_v} + T_{NS} & LP-std,\ LP-est, \\ T_{NS} & otherwise. \end{cases} \tag{9}$$

For the sampling cost T_s, it is the average query time per node in autoregressive model, while it is the time cost of sampling 10 walks per node with walk length of 80 in node2vec. Each experiment is run five times, and the average results are reported.

6.2 Effectiveness of Greedy Algorithms

We show the efficiency of node sampler assignments generated by different greedy algorithms on various memory budgets. Four datasets are used. For Blogcatalog, Flickr, Youtube, we set the maximum memory budget to 3GB, 70GB, 25GB respectively, that ensures the alias method can run in memory. For LiveJournal, the ideal maximum memory budget (≈109GB) exceeds the physical total memory (96GB), so we set the maximum budget to 90GB. Then we vary memory budgets by setting the ratios of the given memory budget to the maximum budget to [0.1, 0.3, 0.5, 0.7, 0.9, 1.0].

We only report the results of four models – NV(4, 0.25), NV(0.25, 4), Auto(0.2), Auto(0.8). One reason is that the selected models are already representative since they have different ranges of the bounding constants (see Figure 4). Another reason is that other configurations have similar results, and the result space of all configurations is huge. In addition, Figure 7 only visualizes the sampling cost and initialization cost on Youtube and LiveJournal because of the limited space. Actually, the results on the other two datasets are consistent with the following analysis.

Sampling cost T_s analysis. In Figure 7 (a)-(d) and (i)-(l), we see that all the greedy algorithms achieve better efficiency by increasing the memory budget. The LP-std and LP-est always outperform the Deg-inc and Deg-dec. Especially, when the ratio is low, the gap of T_s can be up to 46×. Taking Figure 7(a) as an example, when running NV(0.25,4) on Youtube with 2.5 GB memory budget (i.e., ratio=0.1), the LP-std takes 48.115 seconds while Deg-inc needs 2211.71 seconds. This is because the degree is not linearly proportional to the cost of sampling, so the degree-based algorithms cannot reduce the cost quickly simply according to the degree order. The LP greedy algorithm models the cost saving via gradients, and it can improve the performance effectively even when the memory budget is low. On the other hand, when the memory budget is large enough (e.g., ratio=1.0), all the methods achieve similar efficiency. In addition, LP-std and LP-est have similar efficiency across different settings, this demonstrates that bounding constant estimation works well in practice.

Initialization cost T_{init} analysis. In Figure 7 (e)-(h) and (m)-(p), unlike the sampling cost T_s, the initialization cost T_{init} (Equation 9) gradually increases when the memory budget increases. This is because more alias node samplers are used, and they entail $O(d^2)$ initialization cost for each node. Another observation is that the initialization of LP-std and LP-est is slower than the one of Deg-inc and Deg-dec. One reason is that LP-std and LP-est need the bounding constants, whose computation entails T_{C_v} (yellow bar in Figure 7). Therefore, we decompose T_{init} of LP-std and LP-est into T_{C_v} and T_{NS}. We see that when the memory budgets are large, the T_{NS} of LP-std and LP-est is comparable to the one of Deg-inc and Deg-dec. When the memory budgets are small, the degree-based algorithms are more efficient with respect to the initialization cost. This is because the node sampler assignments generated by Deg-inc and Deg-dec are much simpler than the ones created by LP-std and LP-est.

Considering the overhead T_{C_v} of LP greedy algorithms, LP-est is proposed to reduce the initialization cost. Table 3 lists the T_{C_v} of LP-std and LP-est on four datasets. Since the computation of C_v is irrelevant to the memory budget, we showcase the average T_{C_v} of different memory budgets on the same second-order random walk model. Overall LP-est can reduce 8.76%-92.07% of T_{C_v} caused by LP-std, while it still remains the similar sampling efficiency.

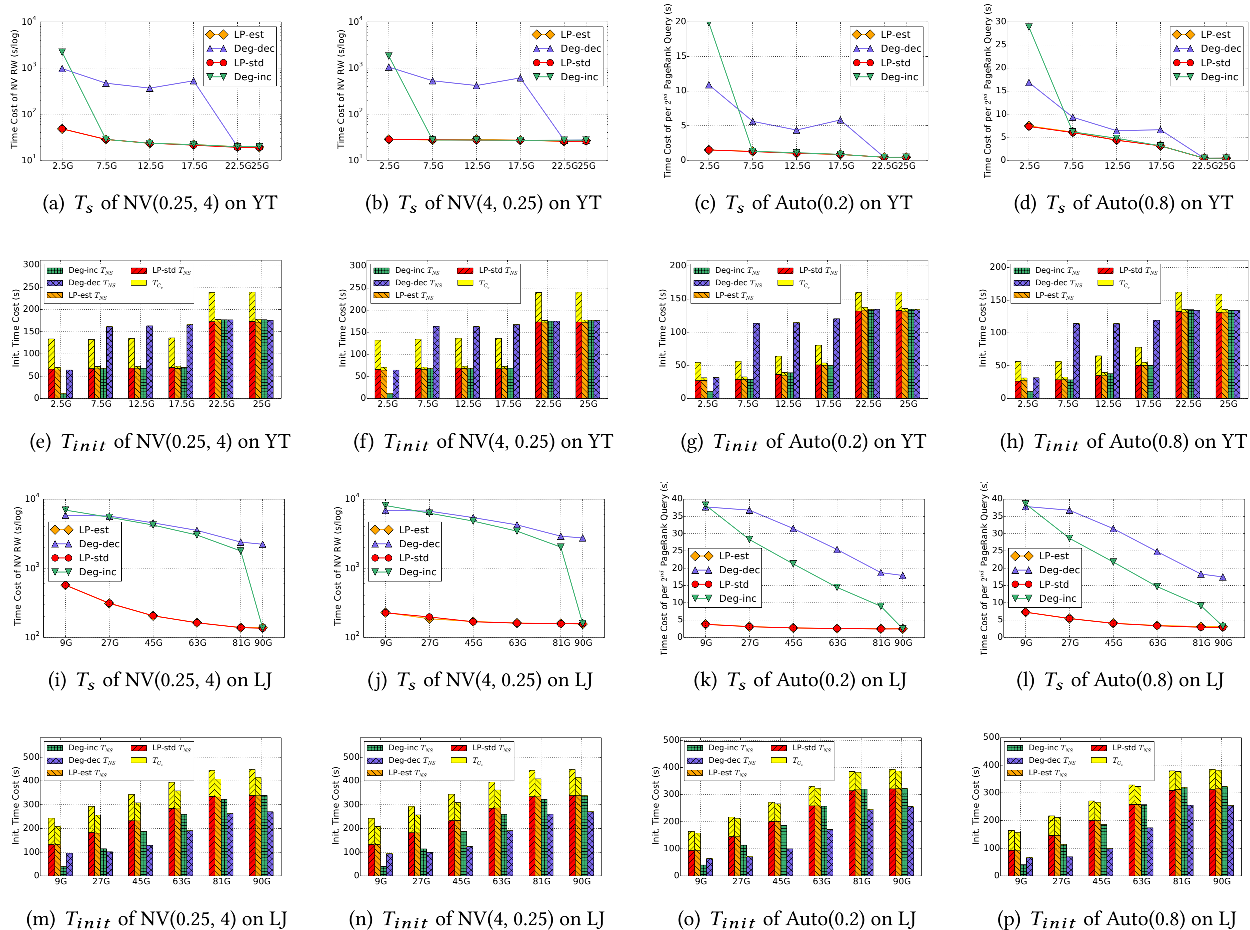
6.3 Efficiency of the Proposed Framework

We compare our proposed framework with three memory-unaware solutions on four datasets. Since our framework is memory-aware and the larger memory budget results in better performance, we choose memory budget ratios 0.1 and 1.0 as representative for comparison. Table 5 summarizes the sampling cost T_s and initialization cost T_{init}.

Memory footprint comparison. We present the memory consumption of memory-unaware solutions in Table 4. Consistent with our theoretical analysis in Section 4, the naive occupies the least memory, the rejection consumes the memory comparable to the graph size, and the alias incurs the most memory, which has the memory explosion problem.

Sampling cost T_s comparison. From Table 5, we see that when the memory budget is large enough for the dataset, alias method achieves the best performance, and LP-std (1.0) entails the similar time cost. The slight performance gap between alias method and LP-std (1.0) is caused by the following two reasons. One is that for nodes with degree one, LP-std uses the naive method on the basis of the cost model. The other is the overhead of the memory-aware second-order random walk framework itself. However, when the dataset becomes large (e.g., LiveJournal), the alias method suffers from Out-Of-Memory (OOM) error, but LP-std (1.0) still works well. This result demonstrates the robustness of our framework.

Another interesting observation is that LP-std (0.1) achieves comparable performance as well, and it is always faster than the rejection method. This implies even when the memory budget is small, our framework is able to find a good node sampler assignment for efficient random walk generation.



(a) T_s of NV(0.25, 4) on YT
(b) T_s of NV(4, 0.25) on YT
(c) T_s of Auto(0.2) on YT
(d) T_s of Auto(0.8) on YT
(e) T_{init} of NV(0.25, 4) on YT
(f) T_{init} of NV(4, 0.25) on YT
(g) T_{init} of Auto(0.2) on YT
(h) T_{init} of Auto(0.8) on YT
(i) T_s of NV(0.25, 4) on LJ
(j) T_s of NV(4, 0.25) on LJ
(k) T_s of Auto(0.2) on LJ
(l) T_s of Auto(0.8) on LJ
(m) T_{init} of NV(0.25, 4) on LJ
(n) T_{init} of NV(4, 0.25) on LJ
(o) T_{init} of Auto(0.2) on LJ
(p) T_{init} of Auto(0.8) on LJ

Figure 7: The comparison of T_s and T_{init} with respect to the greedy algorithms on Youtube (YT) and LiveJournal (LJ).

Dataset	NV(0.25, 4)			NV(4, 0.25)			Auto(0.2)			Auto(0.8)		
	LP-std T_{C_v}	LP-est T_{C_v}	Save cost	LP-std T_{C_v}	LP-est T_{C_v}	Save cost	LP-std T_{C_v}	LP-est T_{C_v}	Save cost	LP-std T_{C_v}	LP-est T_{C_v}	Save cost
Blogcatalog	2.830	1.670	**40.99%**	2.889	1.661	**42.51%**	1.867	1.470	**21.26%**	1.874	1.456	**22.31%**
Flickr	85.841	47.480	**44.69%**	85.981	47.477	**44.78%**	56.695	44.259	**21.93%**	56.734	44.322	**21.88%**
Youtube	66.453	5.333	**91.97%**	67.232	5.334	**92.07%**	28.128	4.139	**85.29%**	28.802	4.146	**85.61%**
LiveJournal	110.442	75.776	**31.39%**	109.949	75.697	**31.15%**	71.075	64.981	**8.57%**	71.040	64.816	**8.76%**

Table 3: The comparison of bounding constant computation cost T_{C_v} between LP-std and LP-est.

Graph	Naive	Rejection	Alias
Blogcatalog	0.3MB	8MB	2,848MB
Flickr	0.4MB	139MB	66,996MB
Youtube	6MB	174MB	22,949MB
LiveJournal	20MB	1,372MB	111,980MB*

Table 4: Memory footprint of memory-unaware solutions. * indicates the memory size is estimated.

Initialization cost T_{init} comparison. We compare the T_{init} of memory-unaware solutions and our proposed method. The naive method has the best initialization performance since it almost does not need to do any heavy preprocessing. The rejection method incurs the moderate initialization cost, and it only needs to compute the factor $min_{t \in N(v)} \frac{w_{vt}}{w'vt}$ in Equation 4. The alias method entails the heaviest initialization cost among the three memory-unaware solutions.

For the memory-aware solutions, the initialization cost consists of T_{C_v} and T_{NS}. When the memory budget is small, the initialization cost of our method is less than the one of alias method, since a few numbers of alias tables need to be created, but it is still greater than the rejection method because of T_{C_v}. When the memory budget is large, the initialization cost of the memory-aware framework is greater than the alias method due to T_{C_v} as well.

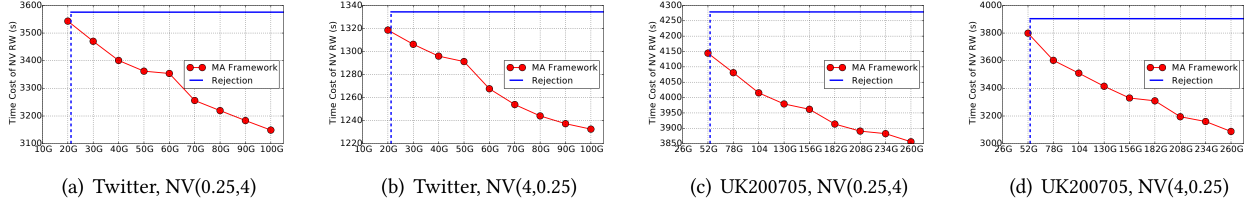
6.4 Efficiency over Billion-egde Graphs

To further demonstrate the efficiency of the memory-aware (MA) framework over very large graphs, we run node2vec random walk task on two billion-edge graphs, Twitter and UK200705. Assuming the graph size is M_g, we set memory

Model	Cost Type	Blogcatalog Naive	Blogcatalog Rejection	Blogcatalog Alias	Blogcatalog LP-std (0.1)	Blogcatalog LP-std (1.0)	Flickr Naive	Flickr Rejection	Flickr Alias	Flickr LP-std (0.1)	Flickr LP-std (1.0)
NV(0.25,4)	T_{init}	**0**	2.682	9.451	6.020	12.048	**0**	83.920	229.846	181.263	315.415
	T_s	40.894	1.247	**0.169**	0.881	0.173	723.564	22.230	**2.124**	15.052	2.205
NV(4,0.25)	T_{init}	**0**	2.692	9.329	6.151	12.176	**0**	83.855	229.846	182.058	315.709
	T_s	28.462	0.241	**0.174**	0.204	0.183	588.911	4.121	**2.258**	3.401	2.324
Auto(0.2)	T_{init}	**0**	1.852	9.115	4.119	10.834	**0**	55.433	220.835	123.821	276.978
	T_s	0.522	0.015	**0.004**	0.014	**0.004**	5.495	0.115	**0.040**	0.099	0.047
Auto(0.8)	T_{init}	**0**	1.807	9.207	4.156	11.058	**0**	55.662	220.735	123.596	277.582
	T_s	0.757	0.050	**0.004**	0.047	0.005	6.378	0.379	**0.044**	0.312	0.050

Model	Cost Type	Youtube Naive	Youtube Rejection	Youtube Alias	Youtube LP-std (0.1)	Youtube LP-std (1.0)	LiveJournal Naive	LiveJournal Rejection	LiveJournal Alias	LiveJournal LP-std (0.1)	LiveJournal LP-std (1.0)
NV(0.25, 4)	T_{init}	**0.003**	64.357	172.012	135.031	240.450	**0.013**	108.741	OOM	251.328	455.423
	T_s	2558.18	127.374	**18.188**	48.115	18.980	7472.88	803.293	OOM	569.428	**135.041**
NV(4, 0.25)	T_{init}	**0.003**	66.410	172.746	133.317	242.238	**0.013**	109.026	OOM	250.705	455.831
	T_s	2253.84	43.114	**24.403**	28.281	25.793	8815.27	289.735	OOM	226.582	**157.496**
Auto(0.2)	T_{init}	**0.003**	28.210	132.701	55.747	161.893	**0.013**	69.845	OOM	172.219	400.057
	T_s	24.202	1.780	**0.417**	1.486	0.427	44.077	4.411	OOM	3.743	**2.448**
Auto(0.8)	T_{init}	**0.003**	28.244	132.422	57.520	160.562	**0.013**	69.873	OOM	171.973	391.956
	T_s	33.631	8.282	**0.410**	7.318	0.449	44.118	8.828	OOM	7.265	**2.877**

Table 5: Efficiency comparison among memory-unaware approaches and the memory-aware framework. The bold style indicates the best performance and the underline style is the second best performance.



(a) Twitter, NV(0.25,4) (b) Twitter, NV(4,0.25) (c) UK200705, NV(0.25,4) (d) UK200705, NV(4,0.25)

Figure 8: Sampling efficiency of the memory-aware framework on Twitter and UK200705.

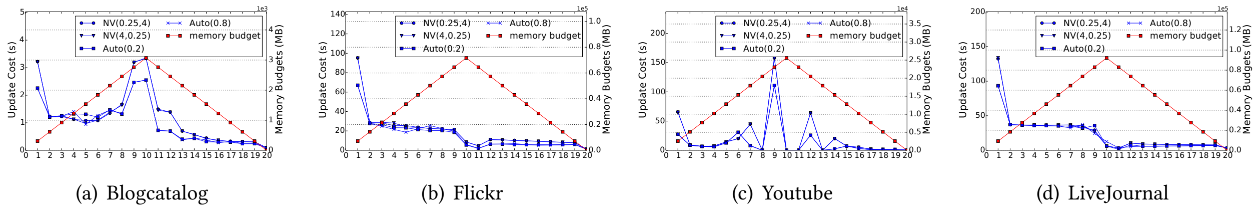
budgets by varying from M_g to $10M_g$. Table 2 gives the M_g of Twitter and UK200705. The experiments are conducted on a server equipping with 376GB memory. Among the three memory-unaware solutions, except the rejection method, naive method cannot finish the task in 4 hours, and alias method fails because of the Out-Of-Memory error. Therefore, we only compare the MA framework with the rejection method, which requires the minimal memory around 21G and 54G on Twitter and UK200705 respectively.

Figure 8 illustrates the experimental results. Notice that the performance of the MA framework with memory budget M_g is not listed because it cannot finish the execution in 4 hours. First, the MA framework can process the billion-edge graphs efficiently. On both Twitter and UK200705, it outperforms the rejection method even when the memory budget is smaller than the minimal memory requirement of the rejection method. This is because the framework assigns some nodes with small degree the naive method, thus saving memory for other nodes to use the alias method and improving the overall efficiency. Second, with the increase of memory budgets, the efficiency of the framework improves gradually. Taking $NV(4, 0.25)$ on UK200705 as an example, the improvement is about 19%, i.e., from 3799.02s to 3088.17s, when memory budget increases from $2M_g$ to $10M_g$. Compared with the results on previous median-size graphs, the performance improvement of MA framework on billion-edge graphs is not that significant. The reason is that even when the memory budget is $10M_g$, it is still no more than 0.06% of the memory required by alias method (e.g., Twitter and UK200705 need total 1796TB and 379TB for alias method respectively), and the framework cannot allocate many alias node samplers, especially for the nodes with large degree.

6.5 Evaluation with Dynamic Memory Budgets

Here we evaluate the time cost of node sampler assignment update with dynamic memory budgets. For each dataset, we synthetically generate a memory budget update trace. Specifically, given a maximal memory budget M_{max}, first the memory budget linearly increases with the step $\frac{M_{max}}{10}$. Once M_{max} is reached, the memory budget decreases linearly with the same step. The red line in Figure 9 illustrates the trace of dynamic budget. The blue lines visualize the corresponding assignment update cost under the different second-order random walk models. Notice that T_{C_v} is excluded since it is only computed once during the memory budget update.

From the figures, except the first point which initializes the node sampler assignment from scratch, we see that the assignment update cost is small. Taking LiveJournal on Auto(0.2)



(a) Blogcatalog (b) Flickr (c) Youtube (d) LiveJournal

Figure 9: The update cost of different models with dynamic memory budgets. x-axis is the timestamp of changing memory budget.

as an example, during the increase of memory budget, the update cost is around 35 seconds, while at the beginning it needs about 93 seconds. When the memory budget decreases, it only needs about 7 seconds. According to Section 5.3, the logic of memory budget decrease is much simpler than the one of memory budget increase.

In addition, for Blogcatalog and Youtube, when memory budget increases, we find that there are burst update costs. For NV(0.25, 4) on Youtube, when memory budget increases from 20480MB to 23040MB, it takes about 158.047 seconds for the update. After profiling the data, this is because the framework chooses the node with the maximal degree (28,754) to use alias method, which entails the heavy initialization cost.

7 RELATED WORK

7.1 Random Walk on Graphs

Most existing works about random walk focus on designing new models to improve the accuracy of the applications. For example, Li et al. [16] introduced rejection-controlled Metropolis hastings algorithm and generalized maximum-degree random walk algorithm to improve the accuracy of graph property estimation. Sengupta et al. [31] uses random walk to estimate reachability of nodes in a large graph. Besides the first-order random walk, the second-order random walk is widely applied to model higher-order dependencies in various domains, including user trails on the Web [40], global shipping traffic [40], online social networks [7] and e-mail communications [23]. Boldi et al. [2] proposed a triangular random model to unveil arc-community structure in the social network, which assigns different weights for the triangular successors and the non-triangular successors. Wu et al. [38] studied the effectiveness of various second-order random walk models on clickstream data.

With regard to the efficiency of random walk-based applications, a general approach is to reduce the number of samples to reach the same accurate estimation. Nazi et al. [24] introduced a walk-estimate framework, which starts with a much shorter random walk, and then proactively estimates the sampling probability to fast approximate the target distribution. Another approach is to reduce the time cost of a sample generation. Zhou et al. [44] introduced a distributed second-order random walk algorithm on a Pregel-like graph computation framework [21] to support large-scale node2vec. Our work also concentrates on efficiency optimization, but it emphasizes the performance of generating random walks in a single server. For the distributed solutions [44] that are built on Pregel-like graph computation frameworks, our proposed memory-aware framework can be applied to help improve the sampling efficiency for each worker. In addition, our framework can function as a middleware for other applications which use second-order random walk as a building block.

7.2 Cost-based Optimization

Cost-based optimization is a classical database technique for query optimization [4]. It looks at all of the possible ways in which a query can be executed and each way is assigned a "cost", which indicates how efficiently that query can be run. Then, we pick the execution plan that has the least cost. It has been extensively used in the relational database, NoSQL, and various big data processing systems [12, 20, 34].

With regard to the graph systems, cost-based optimization is also used to improve the efficiency of graph query [11, 43]. In addition, distributed pattern matching uses cost-based optimization to speed up the external storage access [13]. The memory-aware second-order random walk framework uses the idea of cost-based optimization to find an efficient node sampler assignment so that it can generate random walks fast.

8 CONCLUSION

Second-order random walk becomes an important tool for modeling higher-order dependencies in data. We studied the problem of how to efficiently support the second-order random walk with various memory budgets in this work. We proposed a memory-aware framework by developing a cost-based optimizer. The optimizer assigns different node samplers for each node in the graph and ensures the efficiency is maximized on a memory budget. We designed an effective greedy algorithm to generate such high-efficiency assignment. Finally, we empirically evaluated the framework on four real-world large graphs, and the results clearly demonstrated the advantages of our memory-aware framework. In addition, the complex relationships between the efficiency

and graph structural properties in the context of second-order random walk is still an open question, and we treat it as our future work.

REFERENCES

[1] Mansurul Bhuiyan and Mohammad Al Hasan. 2018. Representing Graphs as Bag of Vertices and Partitions for Graph Classification. *Data Science and Engineering* 3, 2 (Jun 2018), 150–165.

[2] Paolo Boldi and Marco Rosa. 2012. Arc-Community Detection via Triangular Random Walks. In *2012 Eighth Latin American Web Congress.* 48–56.

[3] Stephen Bonner, Ibad Kureshi, John Brennan, Georgios Theodoropoulos, Andrew Stephen McGough, and Boguslaw Obara. 2019. Exploring the Semantic Content of Unsupervised Graph Embeddings: An Empirical Study. *Data Science and Engineering* 4, 3 (Sep 2019), 269–289.

[4] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98).* 34–43.

[5] Krzysztof Dudzinski and Stanislaw Walukiewicz. 1987. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research* 28, 1 (1987), 3 – 21.

[6] Krzysztof Dudzinski and Stanislaw Walukiewicz. 1987. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research* 28, 1 (January 1987), 3–21.

[7] Shanshan Feng, Gao Cong, Arijit Khan, Xiucheng Li, Yong Liu, and Yeow Meng Chee. 2018. Inf2vec: Latent Representation Model for Social Influence Embedding. In *34th IEEE International Conference on Data Engineering (ICDE '18).* 941–952.

[8] Geoffrey Grimmett and David Stirzaker. 2001. *Probability and random processes.* Vol. 80. Oxford university press.

[9] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable Feature Learning for Networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16).* 855–864.

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17).* 1025–1035.

[11] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08).* 405–418.

[12] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *Proc. VLDB Endow.* 4, 11 (2011), 1111–1122.

[13] Jiewen Huang, Kartik Venkatraman, and Daniel J. Abadi. 2014. Query optimization of distributed pattern matching. In *2014 IEEE 30th International Conference on Data Engineering (ICDE '14).* 64–75.

[14] Amy N. Langville and Carl D. Meyer. 2011. *Google's PageRank and Beyond: The Science of Search Engine Rankings, chapter The mathematics guide.* Princeton University Press.

[15] Matthieu Latapy. 2008. Main-memory Triangle Computations for Very Large (Sparse (Power-law)) Graphs. *Theor. Comput. Sci.* 407, 1-3 (Nov. 2008), 458–473.

[16] Rong-Hua Li, Jeffrey Xu Yu, Lu Qin, Rui Mao, and Tan Jin. 2015. On random walk based graph sampling. In *2015 IEEE 31st International Conference on Data Engineering (ICDE '15).* 927–938.

[17] Xiaolin Li, Yuan Zhuang, Yanjie Fu, and Xiangdong He. 2019. A trust-aware random walk model for return propensity estimation and consumer anomaly scoring in online shopping. *Science China Information Sciences* 62, 5 (Mar 2019), 52101.

[18] David Liben-Nowell and Jon Kleinberg. 2003. The Link Prediction Problem for Social Networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management (CIKM '03).* 556–559.

[19] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. 2014. LinkSCAN*: Overlapping community detection using the link-space transformation. In *2014 IEEE 30th International Conference on Data Engineering (ICDE '14).* 292–303.

[20] Hai Liu, Dongqing Xiao, Pankaj Didwania, and Mohamed Y. Eltabakh. 2016. Exploiting Soft and Hard Correlations in Big Data Query Optimization. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1005–1016.

[21] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10).* 135–146.

[22] George Marsaglia. 1963. Generating Discrete Random Variables in a Computer. *Commun. ACM* 6, 1 (1963), 37–38.

[23] Rosvall Martin, Esquivel Alcides V., Andrea Lancichinetti, West Jevin D., and Lambiotte Renaud. 2014. Memory in network flows and its effects on spreading dynamics and community detection. *Nature Communications* 5, 4630 (2014).

[24] Azade Nazi, Zhuojie Zhou, Saravanan Thirumuruganathan, Nan Zhang, and Gautam Das. 2015. Walk, Not Wait: Faster Sampling over Online Social Networks. *Proc. VLDB Endow.* 8, 6 (Feb. 2015), 678–689.

[25] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14).* 701–710.

[26] David Pisinger. 1995. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research* 83, 2 (1995), 394 – 410. EURO Summer Institute Combinatorial Optimization.

[27] Adrian E. Raftery. 1985. A Model for High-Order Markov Chains. *Journal of the Royal Statistical Society. Series B (Methodological)* 47, 3 (1985), 528–539.

[28] Christian P. Robert and George Casella. 2010. *Monte Carlo Statistical Methods.* Springer Publishing Company, Incorporated.

[29] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[30] Vsevolod Salnikov, Michael T. Schaub, and Renaud Lambiotte. 2016. Using higher-order Markov models to reveal flow-based communities in networks. *Scientific reports* 5, 23194 (2016), 1–13.

[31] Neha Sengupta, Amitabha Bagchi, Maya Ramanath, and Srikanta Bedathur. 2019. ARROW: Approximating Reachability Using Random Walks Over Web-Scale Graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE '19).* 470–481.

[32] Yingxia Shao, Bin Cui, Lei Chen, Mingming Liu, and Xing Xie. 2015. An Efficient Similarity Search Framework for SimRank over Large Dynamic Graphs. *Proc. VLDB Endow.* 8, 8 (April 2015), 838–849.

[33] Prabhakant Sinha and Andris A. Zoltners. 1979. The Multiple-Choice Knapsack Problem. *Operations Research* 27, 3 (1979), 503–515.

[34] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1626–1629.

[35] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. 2018. VERSE: Versatile Graph Embeddings from Similarity Measures. In *Proceedings of the 2018 World Wide Web Conference (WWW '18).* 539–548.

[36] Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *Int J Data Warehousing and Mining* 2007 (2007), 1–13.

[37] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.* 3, 3 (1977), 253–256.

[38] Yubao Wu, Yuchen Bian, and Xiang Zhang. 2016. Remember Where You Came from: On the Second-order Random Walk Based Proximity Measures. *Proc. VLDB Endow.* 10, 1 (2016), 13–24.

[39] Yunpeng Xiao, Xixi Li, Yuanni Liu, Hong Liu, and Qian Li. 2018. Correlations multiplexing for link prediction in multidimensional network spaces. *Science China Information Sciences* 61, 11 (Jun 2018), 112103.

[40] Jian Xu, Thanuka Wickramarathne, and Nitesh V. Chawla. 2016. Representing higher-order dependencies in networks. In *Science Advances*.

[41] Eitan Zemel. 1980. The Linear Multiple Choice Knapsack Problem. *Operations Research* 28, 6 (1980), 1412–1423.

[42] Zhipeng Zhang, Yingxia Shao, Bin Cui, and Ce Zhang. 2017. An Experimental Evaluation of Simrank-Based Similarity Search Algorithms. *Proc. VLDB Endow.* 10, 5 (2017), 601–612.

[43] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 340–351.

[44] Dongyan Zhou, Songjie Niu, and Shimin Chen. 2018. Efficient Graph Computation for Node2Vec. *CoRR* abs/1805.00280 (2018). arXiv:1805.00280