SOWalker: An I/O-Optimized Out-of-Core Graph Processing System for Second-Order Random Walks

Yutong Wu, Zhan Shi, Shicai Huang, Zhipeng Tian, Pengwei Zuo, Peng Fang, Fang Wang, and Dan Feng, *Wuhan National Laboratory for Optoelectronics Huazhong University of Science and Technology*

https://www.usenix.org/conference/atc23/presentation/wu

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

SOWalker: An I/O-Optimized Out-of-Core Graph Processing System for Second-Order Random Walks

Yutong Wu, Zhan Shi, Shicai Huang, Zhipeng Tian, Pengwei Zuo,
Peng Fang, Fang Wang, Dan Feng
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Corresponding Author: Zhan Shi (zshi@hust.edu.cn)

Abstract

Random walks serve as a powerful tool for extracting information that exists in a wide variety of real-world scenarios. Different from the traditional first-order random walk, the second-order random walk considers recent walk history in selecting the next stop, which facilitates to model higher-order structures in real-world data. To meet the scalability of random walks, researchers have developed many out-of-core graph processing systems based on a single machine. However, the main focus of out-of-core graph processing systems is to support first-order random walks, which no longer perform well for second-order random walks.

In this paper, we propose an I/O-optimized out-of-core graph processing system for second-order random walks, called SOWalker. First, we propose a walk matrix to avoid loading non-updatable walks and eliminate useless walk I/Os. Second, we develop a benefit-aware I/O model to load multiple blocks with the maximum accumulated updatable walks, so as to improve the I/O utilization. Finally, we adopt a block set-oriented walk updating scheme, which allows each walk to move as many steps as possible in the loaded block set, thus significantly boosting the walk updating rate. Compared with two state-of-the-art random walk systems, GraphWalker and GraSorw, SOWalker yields significant performance speedups (up to 10.2×).

1 Introduction

Random walks on graphs have received significant attention for their ability to extract meaningful insights in graph data analysis and machine learning [10–14]. Most existing random walk implementations are based on the first-order Markov model [15, 16], which assumes the transition probability only depends on the current vertex and is independent of the previous information. Although many encouraging results have been obtained under this assumption, the high-order information such as second-order properties is ignored, and thus some recent works have revealed the necessity of second-order random walks [17, 18]. Node2vec [14], one of the most popular network embedding methods, uses the second-order random walk to capture neighborhood information of vertices, which significantly outperforms the first-order methods like DeepWalk [13]. Similar findings have been found in graph proximity measurements. Wu et al. [19] developed the second-order PageRank and SimRank, and Liao et al. [20] proposed the second-order CoSimRank, which can explore cluster structures in the graph and better model real-world applications. In addition, the random walk has been widely used in social physics. Rosvall et al. [21] showed how the second-order random walk model constraints on dynamics influence community detection, ranking, and information spreading, while it is difficult for the first-order random walk in these scenarios.

Graphs with billions of edges are becoming more prevalent in many domains, and performing some tasks often requires tens of TB to several PB spaces. Although some vendors such as Amazon (AWS) [6], Oracle [7], and Microsoft [8] provide graph database services, striking a balance between low cost and high quality remains challenging. For example, when processing our largest graph, CrawlWeb (see Section 4.1), on Amazon Neptune [9], we use an Amazon Elastic Compute Cloud (EC2) instance, db.r5.4xlarge (8 cores, 16 virtual cores, 128 GB memory), and 3 TB of storage space. The monthly cost for this instance is up to $3, 000. As the size of the graph continues to grow, the storage requirements and computational resources needed would also increase. This would likely lead to higher costs in terms of storage space, computing resources, and potentially data transfer. In contrast, out-of-core graph processing systems are cheaper and easier, as they utilize external storage for processing large graphs [22–25]. These systems divide a large graph into several blocks (i.e., subgraphs) and store them on disks. During the graph processing, a block is loaded into memory and application-specific vertex or edge values in this block can be updated immediately. As expected, the significant performance bottleneck of out-of-core graph processing systems is the I/O between memory and disks, and developers can ill afford to ignore it. Recently, numerous works have been devoted to designing I/O-efficient graph processing systems for random walks.

DrunkardMob [26] is the first large-scale out-of-core graph processing system that allows massive random walks to be performed in parallel. The states of all walks are represented compactly in memory to minimize the memory footprint of each walk. GraphWalker [27] adopts a state-aware I/O model and an asynchronous walk updating scheme to further improve the I/O performance. Besides, it proposes a lightweight block-centric indexing scheme to reduce the memory requirement for storing walk states. However, these proposed solutions cannot be efficiently compatible with second-order random walks.

In this paper, we propose an I/O-optimized out-of-core graph processing system for second-order random walks, called SOWalker. First, to eliminate useless walk I/Os, we propose a *walk matrix* to represent the walks, so as to prevent loading non-updatable walks whose previous vertices are not in memory. Along with the proposed walk matrix, we design a succinct and compact data structure to provide an efficient representation of a walk. Second, to improve the I/O utilization, we develop a *benefit-aware I/O model*. Specifically, we load multiple blocks with the maximum accumulated updatable walks and only load walks whose previous and current vertices are both in the loaded blocks. For this purpose, we map the block scheduling problem into the maximum edge weight clique problem, and adopt a heuristic algorithm to provide comparable I/O performance but significantly reduce the computation time. Finally, to boost the walk updating rate, we adopt a *block set-oriented walk updating scheme*, which allows each walk can be updated as much as possible in the loaded block set, so as to accelerate the progress of random walks. To summarize, we make the following contributions.

- We propose a *walk matrix*, which prevents loading non-updatable walks, so as to eliminate useless walk I/Os.
- We develop a *benefit-aware I/O model*, which loads multiple blocks with the maximum accumulated updatable walks, so as to maximize the I/O utilization.
- We adopt a *block set-oriented walk updating scheme*, which allows each walk to move as many steps as possible in the loaded block set, so as to boost the walk updating rate.
- We conduct detailed experiments on a variety of real-world and synthetic graphs to evaluate SOWalker. Extensive evaluation results show that SOWalker can substantially reduce the I/O cost, achieving up to 10.2× speedup.

The rest of this paper is organized as follows. Section 2 presents the background and motivation. Section 3 describes the detailed system designs of SOWalker. Section 4 evaluates the system and compares it with two state-of-the-art systems. Section 5 gives an overview of related work, and finally, Section 6 concludes this paper.

2 Background and Motivation

Given a graph $G = (V, E)$, where V and E are the set of vertices and edges, respectively. Each edge $e \in E$ is an ordered pair $e = (u, v)$ and is associated with a weight w_{uv}. For each $u \in V$, the neighbor set of a vertex u is $N(u)$. For easy reference, we illustrate the frequently used notations in Table 1.

Notation	Description
$G = (V, E)$	graph G with vertex set V and edge set E
$e = (u, v)$	edge from u to v
w_{uv}	weight between vertex u and v
$N(u)$	set of neighbor vertices of vertex u
B	block set
$\|B\|$	number of blocks in B
m	maximum number of blocks cached in memory
B_L	loaded block set in a batch
$W(i, j)$	number of walks crossing between block i and j
AUW	accumulated updatable walks
CDG	complete directed graph
k	actual number of blocks to be loaded
β	a bitmap to represent whether the block is cached in memory
T_0, T_s	initial temperature and end temperature
γ	cooling coefficient of temperature
$iter_{max}$	maximum number of iterations

Table 1: Notations.

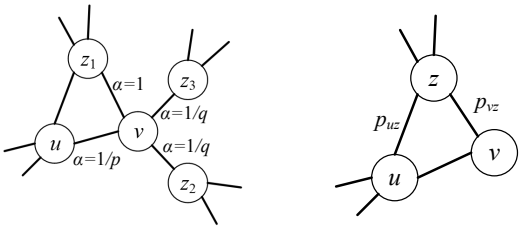
2.1 Second-order Random Walk

First-order random walk. Suppose a walk is visiting vertex v, in the next step, the walk will move to a neighbor of v with transition probability $p_{vz} = P(z|v) = w_{vz}/W_v$, where $W_v = \sum_{t \in N(v)} w_{vt}$.

Second-order random walk. Given that the walk is visiting vertex v at the current step and vertex u at the previous step, the second-order transition probability that moving to vertex z at the next step is $p_{uvz} = p(z|uv)$. Such transition probability can be interpreted as the edge-to-edge transition probability: let $\alpha = (u, v)$ be the edge from vertex u to v, and $\beta = (v, z)$ be the edge from vertex v to z, that is, $p_{uvz} = p_{\alpha\beta}$.

Below are two representative examples of second-order random walk-based algorithms.

Node2vec. Node2vec [14] is a popular network embedding method that introduces the second-order random walk. In order to combine Depth First Search (DFS) and Breadth First Search (BFS), two parameters p and q control the random walk strategy. Parameter p controls the probability of repeated access to the just visited vertex. Parameter q controls whether a walk moves inward or outward. Given that vertex u was visited at the previous step, the unnormalized transition probability p_{uvz} from the current vertex v to the



(a) Node2vec (b) Second-order PageRank

Figure 1: The transition probability computation in second-order random walk algorithms.

next vertex z depends on the edge weight w_{vz} and $\alpha_{pq}(u,v,z)$, i.e., $p_{uvz} = \alpha_{pq}(u,v,z) \cdot w_{vz}$. α_{pq} is calculated in the following formula (shown in Figure 1(a)):

$$\alpha_{pq}(u,v,z) = \begin{cases} \frac{1}{p}, & d_{uz} = 0 \\ 1, & d_{uz} = 1 \\ \frac{1}{q}, & d_{uz} = 2 \end{cases}$$

where d_{uz} denotes the shortest path distance between vertices u and z, and $d_{uz} \in \{0,1,2\}$.

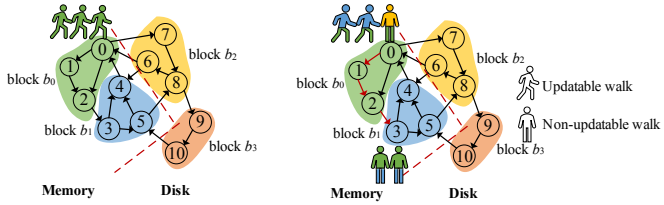
Second-order PageRank. Wu et al. [19] proposed a second-order PageRank and used an autoregressive model to compute the second-order influence probability, which is described as follows:

$$p_{uvz} = \frac{p'_{uvz}}{\sum_{t \in N(v)} p'_{uvt}}$$

where $p'_{uvz} = (1-\alpha)p_{vz} + \alpha p_{uz}$ (shown in Figure 1(b)). The parameter $\alpha \in [0,1)$ is a constant (e.g., 0.2) to control the strength of effect from the previous step.

2.2 Motivation

The out-of-core random walk systems divide a graph into several blocks and cache some of them in memory, the remainder blocks reside in disks temporarily. The total number of cached blocks is limited by the available memory and block size. During the random walk procedure, a block is loaded from disk into memory according to a scheduling model, which is called the *current block*. Walks that reside on the current block are also loaded into memory and can be updated as much as possible until they reach the boundary of the loaded block. However, the main focus of out-of-core random walk systems is to support first-order random walks, which are no longer effective for second-order random walks. In the following, we will discuss three major challenges that lead to poor I/O performance on existing systems.



(a) First-order random walks (b) Second-order random walks

Figure 2: Differences in updatable walks between (a) first-order random walks and (b) second-order random walks. All three first-order random walks are updatable as opposed to only two second-order random walks. View in color for optimal visualization.

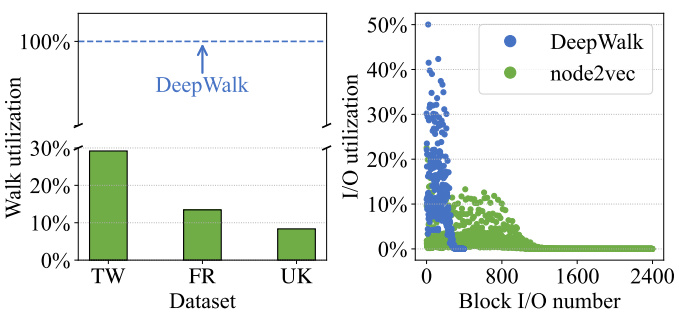
Non-updatable walks result in useless walk I/Os. The first-order random walk's transition probability only depends on the current vertex. As long as the current vertex is in memory, the walk can be immediately updated. Unlike the first-order random walk, the second-order random walk considers recent walk history in selecting the next stop. However, the previous vertex might belong to other blocks on slow disks. Consequently, due to the lack of previous vertex information, some loaded walks cannot be updated directly, resulting in *non-updatable walks*. As a result, these non-updatable walks lead to useless walk I/Os.

As an example, Figure 2 illustrates the differences in updatable walks between first-order and second-order random walks for a specific iteration, where block b_0 and b_1 are in memory, and block b_0 is the current block. Suppose that there are three walks residing on vertex 0. For first-order walks in Figure 2(a), all three walks are updatable. As a result, the walk utilization of first-order random walks is always 100%, which is defined as the ratio of updatable walks to the total loaded walks. For second-order walks in Figure 2(b), the color of the walk represents its state, with the upper and lower colors indicating the block that the previous and current vertex belongs to, respectively. Out of the three walks, one has its upper half-colored yellow, indicating that its previous vertex belongs to block b_2, which is not in memory. As a result, this walk is non-updatable, resulting in the walk utilization of only 2/3. In order to further quantitatively evaluate the walk utilization of node2vec on real-world graphs, we conducted experiments on three datasets (introduced in Section 4.1). As shown in Figure 3(a), the walk utilization of node2vec is less than 30%, and it decreases as the size of the graph increases.

In SOWalker, we propose a walk matrix, which prevents loading non-updatable walks to eliminate useless walk I/Os.

The non-optimal block scheduling model results in low I/O utilization. To update non-updatable walks, the existing block scheduling model [27, 28] iteratively loads ancillary blocks where previous vertices belong to, resulting in a large number of additional block I/Os. On the other hand, due to the irregular structure of graphs and the randomness inherent in random walks, previously visited vertices are unevenly scattered in different blocks.

To quantify the effect of the non-optimal block scheduling model on I/O utilization, we run DeepWalk (i.e., first-order) and node2vec (i.e., second-order) on a state-of-the-art system, GraphWalker [27]. The I/O utilization is defined as the num-



(a) Walk utilization (b) I/O utilization

Figure 3: Walk utilization and I/O utilization.

ber of walk steps divided by the number of edges in a loaded block. For Deepwalk, we use GraphWalker's state-aware I/O model to preferentially load the block with the most walks into memory. For node2vec, we also load a block with the most walks as the current block and iteratively load another block into memory as the ancillary block. Figure 3(b) shows the I/O utilization of DeepWalk and node2vec. We can see that the I/O utilization is significantly low in the second-order random walk application. Besides, running DeepWalk requires fewer than 400 block I/Os, while running node2vec requires over 2400 block I/O, which severely slows down the processing of random walks.

In SOWalker, we develop a benefit-aware I/O model, which loads multiple blocks with the maximum accumulated updatable walks to maximize the I/O utilization.

The block-oriented walk updating scheme brings low walk updating rate. Existing systems [27] manage walks at a block granularity and restrict walk updating to a block, which is called *block-oriented walk updating*. However, this hinders the walk updating and walks fail to utilize the vertex information in other blocks residing in memory, resulting in low walk updating rate. For example, in Figure 2(b), suppose that two updatable walks move along the red path toward vertex 3. Under the block-oriented walk updating scheme, block b_1, where vertex 3 belongs to, is not the current block, so the walks cannot continue to move, which leads to low walk updating rate. In fact, if a walk moves to any vertex belonging to the block in memory, it can further be updated, since the previous and current vertex information are both available.

In SOWalker, we adopt a block set-oriented walk updating scheme, which allows each walk to move as many steps as possible in the loaded block set to boost the walk updating rate.

3 Design of SOWalker

In this section, we first present the system overview of SOWalker. Then, we introduce the detailed designs including the walk matrix, benefit-aware I/O model, and block set-oriented walk updating scheme.

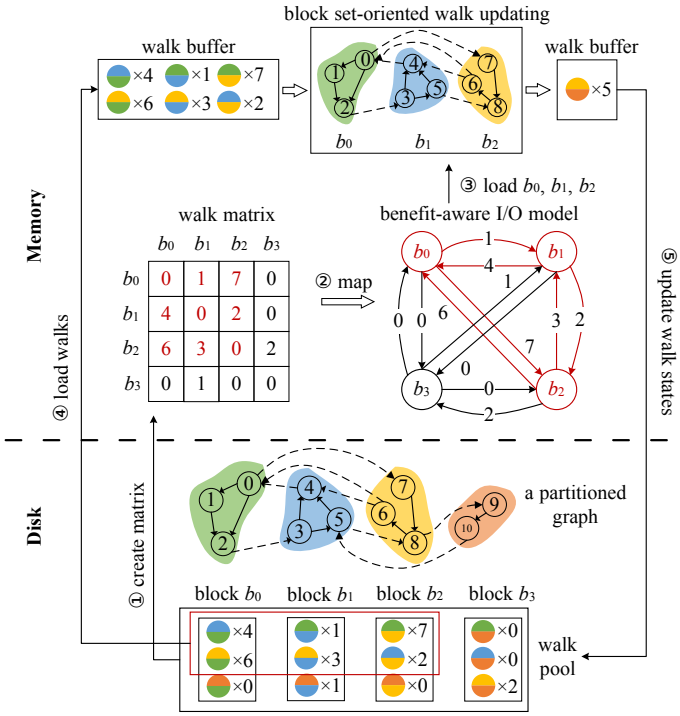


Figure 4: Overall design of SOWalker. View in color for optimal visualization.

3.1 System Overview

Figure 4 shows the overall design of SOWalker. To represent large graphs in the external memory setting, the graph is often partitioned into several blocks and stored on disks, and each block is associated with a walk pool storing the temporarily non-updatable walk states. A semi-circle in the walk pool is a vertex in a walk, with the color of the upper and lower semi-circle indicating the block that the previous and current vertices belong to, respectively. '$\times n$' means there are n walks of such state. Considering that the previous and current vertices may belong to two blocks, we propose a *walk matrix* to intuitively show the number of walks crossing between blocks. The values in the walk matrix are created and updated based on the walk states in the walk pool (①).

During the random walk procedure, we first load m blocks simultaneously to fit into memory, where m denotes the maximum number of blocks cached in memory. Suppose that $m = 3$ in Figure 4. We define loading a block as one *block I/O*, and scheduling and loading m blocks at the same time as a *batch*. To maximize accumulated updatable walks in a batch, we develop a *benefit-aware I/O model*. Specifically, relationships between blocks (i.e., the values in the walk matrix) can be mapped as a directed complete graph (②), and the block scheduling can be modeled into the maximum edge weight clique problem. Nodes in the clique are the blocks to be loaded. As an example in the figure, there are 23 updatable walks in block b_0, b_1, and b_2, which is the maximum

among all candidate block sets. Therefore, we load the three blocks into memory (③, shown as red nodes), and only load the walks whose previous and current vertices are both in the loaded blocks (④, walks in red box).

During the updating phase, we adopt a *block set-oriented walk updating scheme* that allows walks to access vertices in all loaded blocks since there are some edges connecting blocks (shown as arrows with dashed lines). That is to say, if the next vertex is still in memory, the walk can keep moving until it reaches a termination condition or visits a vertex belonging to a disk block. Here, 18 walks finish and the remaining 5 walks move to block b_3. When there are no more walks in memory, update the walk states in the walk pool (⑤). Repeat the process of block loading and walk updating until all random walks are finished.

3.2 Walk Matrix Representation

In order to skip loading non-updatable walks and eliminate useless walk I/Os, we use a walk matrix to represent the walks. The dimensionality of the matrix is the number of blocks. Each element (i, j) in this matrix represents the walks whose previous vertex belongs to block i, and the current vertex belongs to block j. The sum of all elements is the number of unfinished walks. The walk matrix is created and updated according to the walk states in the walk pool. In each batch, m blocks are selected based on the number of walks in the walk matrix, which will be discussed in Section 3.3. When all the walks in memory have been finished or have reached the boundary of the loaded block set, SOWalker checks the walk states to obtain the block IDs of the previous and current vertices. If the IDs are different, it means the walk is crossing blocks. Count the number of such walks that cross blocks and update the corresponding element of the walk matrix. Based on the walk matrix, SOWalker can readily check whether a walk can be updated, judging that both the previous vertex and the current vertex are in memory, thus skipping loading non-updatable walks and eliminating useless walk I/Os.

Figure 5 shows the detail of walk matrix W. The elements in W represent the number of walks crossing blocks at the current time. Suppose a graph is divided into 8 blocks, and the maximum number of blocks cached in memory is 3. If SOWalker selects blocks b_0, b_1, and b_2 to load into memory, then only the updatable walks, which are in the red box need to be loaded. Other walks do not need to be loaded because the blocks containing the previous or previous vertices are not in memory. Note that the number of walks in $W(i,i)$ is always 0 because the walk whose previous and current vertices are in the same block can be updated without additional block I/Os. Without the walk matrix, walks whose current vertices are in blocks b_0, b_1, and b_2 will be loaded (in the green box). However, the walks in the set difference of the green box and the red box are non-updatable walks.

In order to organize the walk data more compactly, we adopt a succinct data structure to encode each walk with 128 bits as shown in Figure 5 (on the right). *source*, *previous* and *current* is encoded in 29 bits, which represents the start vertex, previous vertex, and current vertex of a walk respectively. In this way, SOWalker can support starting random walks from 2^{29} source vertices simultaneously. Commonly, random walks are fed into downstream tasks, so it is necessary to save walk paths easily. In order to identify each walk quickly, we encode *walk ID* in 34 bits, which supports a maximum of 32 (i.e., $2^{34}/2^{29}$) walks starting from each vertex. Besides, *hop* indicates the number of steps the walk has already moved.

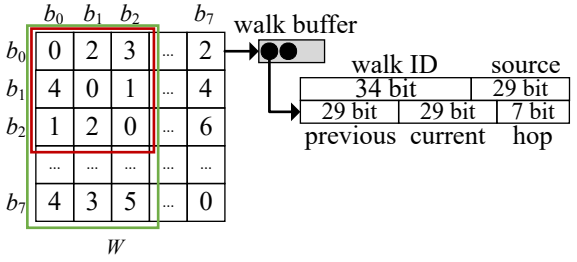


Figure 5: Walk matrix representation.

3.3 Benefit-Aware I/O Model

This section discusses how to efficiently schedule blocks. Since the previous and current vertices of a second-order random walk may belong to different blocks, we need to consider dependencies between blocks. Thus, we simultaneously schedule multiple blocks, instead of one block. Specifically, to improve the I/O utilization, we propose a benefit-aware I/O model to load multiple blocks with the maximum accumulated updatable walks. For this purpose, we formulate the block scheduling problem as the maximum edge weight clique problem. We also adopt an efficient heuristic algorithm to provide comparable I/O performance with a fraction of the cost.

Problem definition. Suppose the block set is B, $|B|$ is the number of blocks, and m is the maximum number of blocks cached in memory. $W(i, j)$ denotes the number of walks crossing between block i and j, that is, the values in the walk matrix. B_L is the loaded block set in a batch. The goal of block scheduling is to load multiple blocks with the maximum accumulated updatable walks, so the benefit is measured in terms of the *accumulated updatable walks* (*AUW*), defined as:

$$AUW(B_L) = \sum_{i \in B_L} \sum_{j \in B_L} W(i, j) \tag{1}$$

Problem mapping. Here, the relationship between blocks can be illustrated with a complete directed graph. The edge weight between two neighborhood blocks denotes the number of walks crossing between two blocks. For the block set B, the complete directed graph (*CDG*) is defined as:

$$CDG = (B, E) \tag{2}$$

$$E = \{e_{ij} = W(i,j) | i, j \in B\} \tag{3}$$

To maximize the accumulated updatable walks, we convert the block scheduling problem into the maximum edge weight clique problem, i.e., maximizing the sum of edge weights from all the feasible candidates.

Definition 1 (*Maximum Weighted Scheduling*, *MWS*): Given the complete directed graph $CDG = (B, E)$ along with the memory capacity requirements, MWS produces the loaded blocks, which satisfies that (1) the block scheduling problem is feasible according to the memory capacity requirements, i.e., the maximum number of blocks cached in memory; (2) the block scheduling problem produces the maximum sum of edge weights that maximizes the number of accumulated updatable walks in a batch.

Taking into account both the memory capacity and maximizing the sum of edge weights, the MWS is formulated as:

$$\max \sum_{i} \sum_{j} e_{ij} y_{ij} \tag{4}$$

$$\text{s.t.} \sum_{i=0}^{|B|-1} x_i = m \tag{5}$$

$$y_{ij} \leq x_i \tag{6}$$

$$y_{ij} \leq x_j \tag{7}$$

$$x_i, y_{ij} \in \{0, 1\} \tag{8}$$

where the variable x_i equals one if block i is loaded in memory. Constraint 5 guarantees that only m blocks can be chosen. By Constraints 6 and 7, for any edge (i, j), a binary variable $y_{ij} = 1$ if and only if both $x_i = 1$ and $x_j = 1$.

To maximizes $AUW(B_L)$, MWS must select m blocks for scheduling, but it ignores the contribution to block I/O reduction. In fact, the candidate block cached in memory does not yield block I/O but the walks in that block can be updated. Therefore, we aim to maximize the number of accumulated updatable walks in a block I/O. The objective function is redefined as:

$$\max_{B_L} S = \frac{AUW(B_L)}{k} \tag{9}$$

where k is the actual number of blocks to be loaded.

We use a bitmap β to record whether the block is cached in memory. If block i is in memory, the bit of block i is set to 1, i.e., $\beta_i = 1$. Otherwise, the bit is set to 0. By identifying which blocks are in memory, we can fully utilize the blocks in memory. Based on this consideration, we try to add the following constraint to the formulation:

$$\sum_{i=0}^{|B|-1} \beta_i \cdot x_i = m - k \tag{10}$$

where $m - k$ chosen blocks are already in memory. Constraint 10 ensures that the chosen block i does not need to be loaded, if and only if $\beta_i = 1$ and $x_i = 1$. We iterate over all $k \in [1, m]$ to find the optimal solution.

Algorithm 1: SA-based benefit-aware I/O model

Input: $CDG = (B, E)$, B_0: initial block set
Output: the loaded block set B_L

```
Function SelectBlocks(CDG=(B,E), B_0):
    B_L ← B_0                    // initial block set
    t ← T_0                      // initial temperature
    i ← 0                        // iteration counter
    while t ≥ T_s and i ≤ iter_max do
        B_c ← CHOOSENEWBLOCK(CDG, B_L)
        ΔS = S(B_c)-S(B_L)
        if ΔS>0 or e^(ΔS/t) > random(0,1) then
            B_L ← B_c
        t ← γt
        i ← i+1
    return B_L
```

The above linear programming method can guarantee the optimality of the solution obtained. However, the complexity of this problem is in order of 2^n [29]. As the scale of the problem is increasing, the complexity also soars. To settle the problem in a reasonable time, we adopt a heuristic algorithm to provide sub-optimal solutions, which is possible to solve large-scale problems within an acceptable time [30].

Solutions via heuristic algorithm. Since the maximum edge weight clique problem is NP-hard [31], many heuristic algorithms have been proposed to achieve a reasonable trade-off between computation time and solution quality. Simulated annealing (SA) is a local search procedure to find an efficient and feasible solution. In order to escape from local optima, a worse solution is accepted as the new solution with a probability that decreases as the computation proceeds. Despite its simpler structure and fewer parameters, SA has shown competitiveness in searching for optimal or near-optimal solutions and has been widely used to solve the maximum edge weight clique problem [32–35].

Inspired by Ernst et al. [35], we also use SA to select a loaded block set to maximize the number of accumulated updatable walks in a block I/O. The establishment of the objective function is described according to Equation 9. The detailed procedures of the SA-based benefit-aware I/O model is given as follows. Algorithm 1 illustrates the pseudo-code of this model.

Step 1: Initialize. Set initial temperature T_0, end temperature T_s, cooling coefficient γ of temperature, and the maximum number of iterations $iter_{max}$, where $iter_{max} = C^m_{|B|}$. Previous work has shown that a good initial solution results in faster convergence and improves the quality of the solution [36, 37]. In order to find a reasonably good initial block set, blocks appear in descending order of the number of walks in it. We

choose the top-m block as the initial block set B_0, meaning the block with more walks is more likely to be loaded into memory.

Step 2: Accept or reject the new solution. SA works iteratively by successively replacing the current solution with a random solution. In each iteration, randomly remove a selected block from the current block set B_L. Then, one of the remaining blocks is chosen randomly, and getting a new candidate block set B_c. Compute the difference between the new candidate block set B_c and the current block set B_L, i.e., the increment of the objective function $\Delta S = S(B_c) - S(B_L)$. If B_c is better, i.e., $\Delta S \geq 0$, then it will be accepted. Otherwise, it will be accepted with probability $p = e^{\Delta S/t}$, where t denotes the current temperature. Generate a random number ζ, where $\zeta \in [0,1]$. If $p > \zeta$, then B_c will be accepted. Otherwise, it will be rejected.

Step 3: Continue or end. Compute current temperature $t = \gamma t$ and the number of iterations $i = i + 1$. If $t < T_s$ or $i > iter_{max}$, end the algorithm. Otherwise, go back to step 2.

Compared to the exact but complicated linear programming method, SA provides an approximate solution but is much simpler, which yields orders of magnitude speedup and the computation time is only a small fraction of the total execution time (see Section 4.4.1).

3.4 Block Set-Oriented Walk Updating

Existing random walk systems partition a graph into several blocks. The block loading and walk management are at a block granularity, resulting in the walk updating being limited to a single block, called *block-oriented walk updating*. Once a walk reaches the boundary of the current block, the updating of a walk will be stopped. However, such a strategy hinders the walk updating and potentially increases the number of block I/Os.

To see this problem more concretely, we will consider a partitioned graph in Figure 4. The graph is partitioned into four blocks. Suppose two blocks are cached in memory and all walks start at vertex 0. We use the state-aware I/O model in GraphWalker [27] to load the block containing the largest number of walks as the current block, and iteratively load another block as the ancillary block. We skip loading the blocks without containing any previous vertex information. Figure 6(a) shows the process of block-oriented walk updating. (i, j) means the blocks cached in memory, where i is the current block, and j is the ancillary block. Only the walk in the current block can be updated. '+' means the lasted loaded block. As a result, 10 block I/Os are required and the walk steps per block I/O is 2.4.

We argue that although a graph is partitioned into several blocks, walks can move across blocks via the cut edges between these blocks. Thus, if a walk moves to any vertex belonging to the block in memory, it can further be updated, until it reaches the boundary of the block set in memory and moves to the block in disk. Such walk updating is called *block set-oriented walk updating*, which is illustrated in the example in Figure 6(b). In the first batch, blocks b_0 and b_1 are loaded into memory. Walk w_1 can be finished directly without extra block I/Os. While it needs 4 block I/Os in the block-oriented walk updating scheme. In the third batch, both walk w_0 and w_2 can be updated. In contrast, in the block-oriented walk updating scheme, only the walk in the current block can be updated. As a result, the number of block I/Os is reduced to 6 and the walk steps per block I/O increases to 4. The reason is that the block set-oriented walk updating allows walks to repeatedly visit the block in memory, which boosts the walk updating rate and accelerates the random walk process. Recently, GraSorw [28] also allows walks to be updated across the blocks. However, it limits the number of blocks in memory to 2, which is less flexible for different-scale graphs and random walks.

(i, j)	Walk paths
$(+b_0, +b_1)$	w_0: 0→7 w_1: 0→1→2→3 w_2: 0→2→3
(b_1, b_0)	w_1: 3→4→0 w_2: 3→5→8
$(+b_2, b_0)$	w_0: 7→8→6→4
$(b_2, +b_1)$	w_2: 8→6→0
$(+b_0, b_1)$	w_1: 0→2→3
$(b_0, +b_2)$	w_2: 0→1→2 (end)
$(+b_1, b_0)$	w_1: 3→4 (end)
$(b_1, +b_2)$	w_0: 4→0
$(+b_0, b_1)$	w_0: 0→7
$(+b_2, b_0)$	w_0: 7→8→9 (end)

(a) The process of block-oriented walk updating

(i, j)	Walk paths
$(+b_0, +b_1)$	w_0: 0→7 w_1: 0→1→2→3→4→ 0→2→3→4 (end) w_2: 0→2→3→5→8
$(b_0, +b_2)$	w_0: 7→8→6→4
$(+b_1, b_2)$	w_0: 4→0 w_2: 8→6→0
$(+b_0, b_1)$	w_0: 0→7
$(b_0, +b_2)$	w_0: 7→8→9 (end) w_2: 0→1→2 (end)

(b) The process of block set-oriented walk updating

Figure 6: Block- vs. Block set-oriented walk updating.

4 Evaluation

In this section, we evaluate the effectiveness of SOWalker. First, we introduce our experimental setup. Then, we compare SOWalker with two state-of-the-art random walk systems, GraphWalker [27] and GraSorw [28], in terms of overall performance and I/O efficiency. Third, we evaluate the effect of different block scheduling models. Finally, we analyze the impact of block size.

4.1 Setup

Environment. The hardware platform used in our experiments is a commodity server equipped with a 32-core 2.10 GHz Intel Xeon CPU E5-2620 with 128GB main memory and a 3TB HDD, running Ubuntu 20.04 LTS. SOWalker is implemented in around 4, 000 lines of C++ code and compiled by g++ 9.4.0 with an optimization flag as -O3. We use OpenMP for parallel random walks, and the number of threads is set to 32 unless explicitly specified. The reported results were averaged over 5 runs, and the error bars have been omitted as the variance was negligible.

Datasets. Table 2 describes the statistics of our evaluated graphs. RANDOM (RND) is a synthetic graph where each vertex is connected to five randomly selected neighbors. The probability of two vertices being connected is inversely proportional to the difference in their IDs. RMAT-27 (RM27), RMAT-28 (RM28), and Kron30 (K30) are synthetic graphs generated with the Graph500 generator [5], exhibiting a power-law degree distribution. Twitter (TW) [1] and Friendster (FR) [2] are social graphs that show the relationship between users within each online social network. UK-Union (UK) [3] and CrawlWeb (CW) [4] are web graphs that consist of hyperlink relationships between web pages. *Graph Size* is the amount of data stored in text format as an edge list. *CSR Size* is the storage cost to store graphs in CSR format. Since systems are executed in an out-of-core environment, the memory limit is set to 2GB (for RND and RM27), 4GB (for RM28, Twitter, Friendster, and UK-Union), or 32GB (for Kron30 and CrawlWeb). *Block Size* is heuristically set to 1/4 of the memory size according to Section 4.4.3. $|B|$ is the number of blocks that a graph is partitioned into according to the block size.

Dataset	$\|V\|$	$\|E\|$	Graph Size	CSR Size	Block Size	$\|B\|$
RM27	134.2M	1.1B	18GB	4GB	512MB	9
RND	268.4M	1.4B	24.7GB	5.2GB	512MB	11
TW	61.5M	1.5B	24.4GB	5.5GB	1GB	6
RM28	268.4M	2.1B	34.9GB	8GB	1GB	9
FR	65.6M	3.6B	58GB	13.5GB	1GB	14
UK	133.6M	5.5B	94.6GB	20.4GB	1GB	21
K30	1.1B	33.8B	628.3GB	120GB	8GB	16
CW	3.6B	126B	2.6TB	470GB	8GB	59

Table 2: Statistics of datasets.

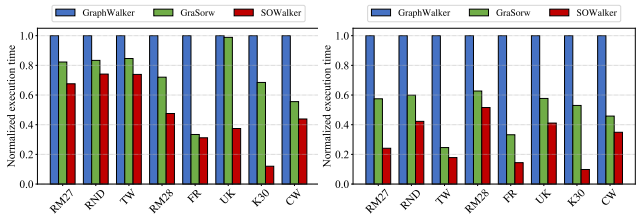
Graph algorithms. We evaluate SOWalker with two second-order random walk-based applications discussed in Section 2, i.e., node2vec and the second-order PageRank. For node2vec, we set the parameter $p = 0.5$, $q = 2$. Each vertex samples 10 walks with a fixed walk length of 80. For the second-order PageRank, the maximum walk length is 20, and we simulate 2, 000 random walks starting at each query source vertex.

Systems for comparison. We perform a comprehensive analysis of SOWalker's performance and compare it with two state-of-the-art random walk systems.

- GraphWalker [27], an I/O-efficient system for first-order random walks. When executing second-order random walks, we adopt the state-aware I/O model to load a block with the maximum number of walks as the current block and iteratively load another block into memory as the ancillary block. Both GraphWalker and SOWalker use the same parameter configuration.
- GraSorw [28] is the first out-of-core graph processing system designed for second-order random walks. It iteratively selects a block as the current block and uses a learning-based block loading model. However, this model consists of three stages: getting the running logs under the full-load mode, training, and running with the trained thresholds. Both the first and third stages involve second-order random walks, rendering it deficient in real-world applications. Therefore, we only use the full-load mode to load the ancillary block. On the other hand, GraSorw fixes the number of blocks in memory to 2, so we set the block size to half the memory size.

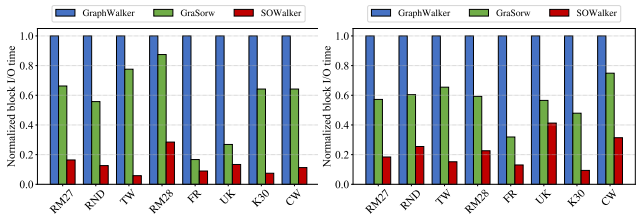
4.2 Overall Performance

We first compare the execution time of the chosen algorithms on different graphs and systems. Figure 7 shows the execution time normalized w.r.t. GraphWalker. We can see that SOWalker is faster than both GraphWalker and GraSorw in all cases. Specifically, SOWalker achieves 1.4-8.3× and 2.4-10.2× speedups over GraphWalker on node2vec and the second-order PageRank, respectively. As for GraSorw, SoWalker achieves 1.2-5.7× and 1.4-5.4× speedups over it on node2vec and the second-order PageRank, respectively. The main reason for the speedup in SOWalker is twofold. First, SOWalker loads multiple blocks with the maximum accumulated updatable walks, which improves the I/O utilization and the walk updating rate, so it requires much fewer blocks I/Os to run second-order random walks. GraSorw, on the other hand, is unaware of the walk states, just iteratively selects a block as the current block and loads an ancillary block into memory. Although GraphWalker loads a block with the maximum number of walks as the current block, it is unaware of the number of walks that can be updated. Therefore, both of them suffer poor performance. Second, SOWalker adopts the block set-oriented walk updating scheme, which allows



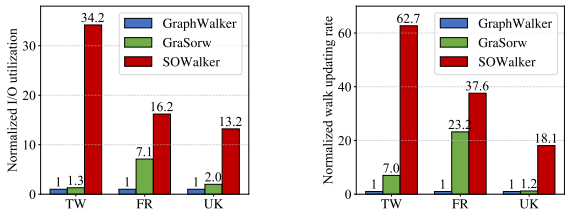
(a) Node2vec (b) Second-order PageRank

Figure 7: Execution time comparison.



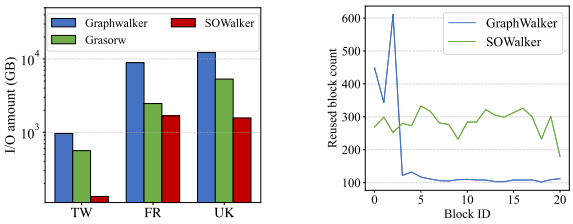
(a) Node2vec (b) Second-order PageRank

Figure 8: Block I/O comparison.



(a) I/O utilization (b) Walk updating rate

Figure 9: I/O utilization and walk updating rate.



(a) I/O amount (b) Reused block count

Figure 10: I/O amount and reused block count.

the loaded walks can be updated as much as possible in the loaded block set, so as to further accelerate the random walk process and reduce the block I/O costs.

4.3 I/O-efficiency Evaluation

Block I/O comparison. To justify the above argument, we compare the block I/O time on SOWalker and the other systems. Block I/O time is the time cost of loading blocks. Figure 8 shows the block I/O time normalized w.r.t. GraphWalker. In all cases, SOWalker outperforms both GraphWalker and GraSorw. Specifically, the block I/O time in SOWalker is only 5.8-41.3% of that in GraphWalker, and 7.5-72.9% of that in GraSorw, respectively. This is mainly attributed to SOWalker's benefit-aware I/O model that loads multiple blocks with the maximum accumulated updatable walks, so as to accelerate the random walk process and significantly reduce the block I/O number. On the other hand, GraphWalker and GraSorw load blocks iteratively, which incur great I/O cost.

I/O utilization. To verify that SOWalker can improve the I/O utilization, Figure 9(a) shows the average I/O utilization for node2vec on Twitter, Friendster, and UK-Union, normalized w.r.t. GraphWalker. As we can see, for all the graphs, SOWalker shows the highest average I/O utilization. Compared to GraphWalker and GraSorw, the I/O utilization of SOWalker is improved by 13.2-34.2× and 2.3-26.4×, respectively. This is mainly attributed to the benefit-aware I/O model, which maximizes the number of walks that can be updated, thereby improving I/O utilization. Besides, according to our walk matrix, we only loads walks whose previous and current vertices are both in memory. This guarantees that all walks can be updated and provides 100% walk utilization. While GraphWalker and GraSorw are unaware of the number of walks that can be updated, resulting in low I/O utilization.

Walk updating rate. Figure 9(b) reports the average walk updating rate normalized w.r.t. GraphWalker. The average walk updating rate in SOWalker significantly outperforms GraphWalker by up to 62.7×. Benefiting from our block set-oriented walk updating scheme, which allows each walk to move as many steps as possible in the loaded block set, SOWalker achieves more walk steps. In contrast, in GraphWalker, once a walk reaches the boundary of the block, the updating of a walk will stop, so the walk steps are limited. Although GraSorw also allows walks to be updated across the two blocks in memory, SOWalker still achieves up to 1.6-15.6× average walk updating rate stemming from the fact that it maximizes the I/O utilization based on the benefit-aware I/O model.

I/O amount. As mentioned earlier, the I/O amount can be divided into two parts: edge data on blocks and walk data. Let N be the total number of block I/Os, M be the block size, and W be the total number of loaded walks, with each walk encoded with 128 bits. The total I/O amount $A = N * M + W * 128$. Since block size is pre-defined, the I/O amount is proportional to the number of block I/Os and loaded walks. Figure 10(a) shows the I/O amount that each system runs node2vec on Twitter, Friendster, and UK-Union. We can see that there is a significant reduction in I/O amount. Compared to GraphWalker, SOWalker achieves an I/O reduction of over 80% on these graphs. Furthermore, the I/O amount in SOWalker

is only 23.7-67.9% of that in GraSorw. This reduction can be attributed to the accelerated random walk process, which enables the walks to be completed faster and significantly reduces the number of block I/Os.

Reused block counts. Blocks in memory can be reused as they do not yield block I/O but walks in these blocks can be updated. Figure 10(b) shows the reused counts for each block on UK-Union. The reused block counts in SOWalker are usually much higher than those in GraphWalker. The reason is that we consider the contribution of loaded blocks in the benefit-aware I/O model. In contrast, in GraphWalker, only a few blocks are highly reused, because many walks stay in these blocks. This makes them more likely to be selected as the current block and cached in memory for a long time. On the other hand, other ancillary blocks are iteratively loaded, resulting in frequent swapping between memory and disk. Consequently, the reused counts of these blocks are low.

4.4 Design Choices

In this section, we conduct experiments to validate some of our critical design choices that are essential to achieve optimal performance for SOWalker.

4.4.1 Comparisons of Scheduling Models

We now evaluate the effectiveness of the block scheduling models by comparing the following models:

- Random: randomly chooses m blocks to load into memory, which is used as the baseline.
- Max-m: chooses top-m blocks based on the number of walks in a block.
- Exact: the exact benefit-aware I/O model according to the linear programming method.
- Benefit-aware I/O model (BA): the benefit-aware I/O model according to the simulated annealing algorithm.

We run node2vec on UK-Union. A similar trend can also be observed on the other graphs and algorithms; their results are omitted due to space limitations. Table 3 presents the execution time, block I/O time, and computation time for the above models. Note that the computation time of the Random model and Max-m model is very short by a negligible amount. There are two observations that can be found. First, both the Random and Max-m models yield relatively higher execution times than BA model. This is expected since the Random model is an arbitrary order without any optimization. The Max-m model also suffers poor performance as it only focuses on the maximum number of walks. Some loaded walks cannot be updated due to the lack of previous vertex information, which wastes precious disk bandwidth and slows down the processing of random walks. Second, BA model achieves both the best performance and the near-optimal block scheduling model. To verify the correctness of BA, we compare it with the Exact model. The results exhibit that BA gives rise to a similar but slightly higher block I/O cost over the Exact model. More importantly, BA provides a speedup of 6.7× of the Exact model, and the computation time of the simulated annealing algorithm is only 10 seconds. While the computation time of the Exact model is nearly 3.5 hours, which constitutes 85% of the total execution time, and we cannot afford such a level of slowdown. In summary, our BA model can achieve faster runtime and better I/O performance.

Model	Execution time (s)	Block I/O time (s)	Block I/O number	Computation time (s)
Random	4970	3234	9868	-
Max-m	3871	2162	6391	-
Exact	14311	548	1484	12097
BA	2133	575	1537	10

Table 3: The comparison with different block scheduling models. '-' means that the computation time is negligible.

4.4.2 Comparisons of Walk Updating Schemes

Next, we evaluate the effectiveness of SOWalker's block set-oriented walk updating through a comparison experiment with block-oriented walk updating.

Block-oriented walk updating cannot be directly used in SOWalker, since our benefit-aware I/O model requires the help of block set-oriented walk updating. Therefore, we design a baseline system, which loads a block with the maximum number of walks as the current block and iteratively loads another block into memory as the ancillary block. We incrementally add the block set-oriented or block-oriented scheme to the baseline system and evaluate the performance impact of our contribution.

Figure 11 exhibits the performance of node2vec running on Twitter, Friendster, and UK-Union. The block set-oriented scheme outperforms the block-oriented scheme for all graphs.

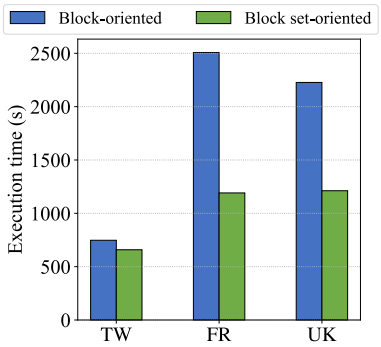


Figure 11: Block- vs. Block set-oriented walk updating.

The performance improvement is especially significant for Friendster, which yields up to 2.1× speedups. The reason behind this is that under the block set-oriented scheme, the walk can move across the entire loaded block set in memory. This leads to a longer walk steps in a block I/O. In contrast, the block-oriented scheme restricts walk updating to only one block, hindering the walk updating process and resulting in a large number of block I/Os.

4.4.3 Impact of Block Size

We also evaluate the impact on performance with different block sizes. Memory is limited to 4GB to illustrate the applicability. To demonstrate, we run node2vec on three representative graphs, Twitter, Friendster, and UK-Union. Figure 12 shows the execution time. The results on GraSorw are omitted since it fixes the number of blocks in memory to 2 and the block size is fixed to half the memory size. SOWalker presents superior performance over GraphWalker across all cases. This improvement is especially significant for small block sizes. This is because with the block-oriented walk updating scheme, GraphWalker restricts walk updating to a block, which severely wastes the vertex information in other blocks residing in memory. While our block set-oriented walk updating scheme allows walks to move across blocks in memory, so as to best utilize resources. Even when the block size is set to 2GB, i.e., 2 blocks in memory, the block-oriented walk updating scheme degrades into the block set-oriented walk updating scheme, the results are still encouraging. The reason is that our benefit-aware I/O model loads multiple blocks with the maximum accumulated updatable walks, so as to accelerate the random walk process. Besides, we observe that the block size should be neither too small nor too large to achieve a good performance in SOWalker. For the smaller block size, the walk updating rate is low. While for the larger block size, the I/O utilization is low. Therefore, according to our experiences, heuristically setting the block size to 1/4 of the memory size can produce the best performance.

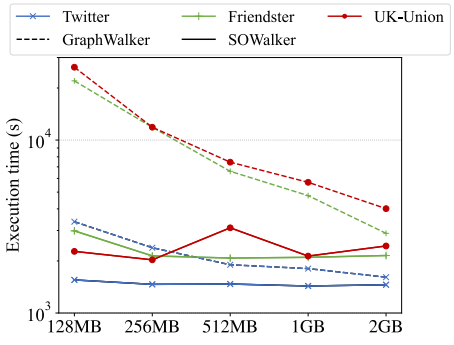
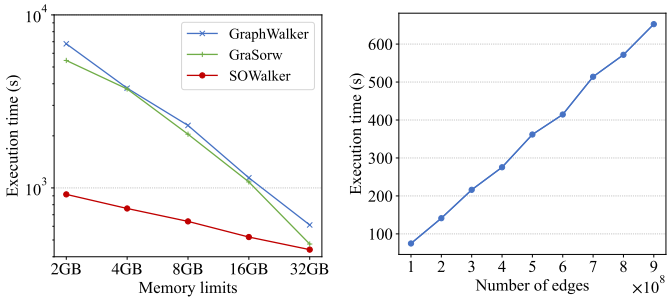


Figure 12: Impact of block size.



(a) Scalability with memory (b) Scalability with graph size

Figure 13: Scalability with memory and graph size.

4.5 Scalability

We evaluate the scalability of SOWalker by examining the performance improvements achieved with increased memory limits. Figure 13(a) illustrates the performance variance of node2vec on UK-Union as the memory increases. All three systems demonstrate good scalability when more powerful memory resources can be utilized. Although as memory increases, the performance disparities between systems tend to diminish, it is worth noting that SOWalker outperforms other systems utilizing 16GB memory, even when it is equipped with only 2GB memory. When the memory is increased to 32GB, the whole graph can fit into memory, and SOWalker still outperforms the other two systems, yielding 1.4× speedups.

We also evaluate the scalability of SOWalker with respect to graph size. We conduct experiments on random graphs and vary the number of edges from 1×10^8 to 9×10^8, with an average degree of 10. As shown in Figure 13(b), as the graph size increases, the implementation exhibits good scalability. However, due to the variability in the structures of different random graphs, some noise could be generated.

5 Related Work

Many graph systems have been proposed to process large graphs. In the past, numerous systems have emphasized the ability to run in a distributed environment, which use a cluster of machines to process large graphs [43–48]. However, distributed graph systems are still bugged by load imbalance problems and significant communication overheads.

Since out-of-core graph processing systems can represent large graphs in the external memory setting, they serve as a promising alternative to distributed solutions. GraphChi [22] is a pioneer in this category, which utilizes the Parallel Sliding Window (PSW) technique to reduce random I/O accesses from storage. X-Stream [49] provides a two-phase Scatter-Gather programming model that makes tradeoffs between random memory access and sequential access from streaming

data. GridGraph [23] presents a 2-level hierarchical partitioning scheme to improve the locality and reduce the number of I/Os. DynamicShards [24] uses dynamic shards to reduce disk I/Os. CLIP [25] and LUMOS [50] make full use of the loaded blocks to reduce disk I/O operations. However, these existing works were not originally designed for random walks and thus give sub-optimal performance.

With the increasing interest in the performance optimization of random walks, a large number of systems have been designed to handle random walks. DrunkardMob [26] is the first random walk system, which enables the simulation of billions of random walks on massive graphs, on just a single computer. However, it adopts the iteration-based model, which limits the efficiency and scalability of random walks. GraphWalker [27] develops a state-aware I/O model and an asynchronous random walk updating schedule to improve the I/O utilization. As it is designed for first-order random walks, it still incurs excessive disk I/Os when executing second-order random walks. GraSorw [28] is designed specifically for second-order random walks. It develops a bi-block execution engine and a learning-based block loading model to improve the I/O efficiency. However, its bi-block execution engine limits the number of blocks in memory to 2, which is less flexible for different-scale graphs and random walks. Moreover, the learning-based block loading model has to run the second-order random walk task twice to get the runtime statistics, rendering it deficient in real-world applications. SOWalker differs from all these systems in the walk representation, block scheduling model, and walk updating scheme. It designs a walk matrix to avoid loading non-updatable walks, proposes a benefit-aware I/O model to improve the I/O utilization, and adopts a block set-oriented walk updating scheme to boost the walk updating rate.

Meanwhile, memory optimizations and the increased number of cores make it possible to process large graphs more efficiently on a single machine. For example, ThunderRW [38] employs the step interleaving technique to hide memory access latency by switching the executions of different random walk queries. FlashMob [39] tries to harvest spatial and temporal locality underneath the apparently random nature of random walks. Besides, Shao et al. [40] proposed a memory-aware framework for second-order random walks, which automatically assigns a suitable sampling method for each node to minimize the time cost within a memory budget. While SOWalker focuses on I/O optimizations, some of these techniques can be implemented to further enhance the in-memory performance.

6 Conclusion

In this paper, we propose an I/O-optimized out-of-core graph processing system for second-order random walks, called SOWalker. To eliminate useless walk I/Os, we propose a walk matrix to prevent loading non-updatable walks. To improve the I/O utilization, we develop a benefit-aware I/O model to load multiple blocks with the maximum accumulated updatable walks. To boost the walk updating rate, we adopt a block set-oriented walk updating scheme to allow each walk to move as many steps as possible in the loaded block set. Our optimizations yield significant performance benefits compared to the state-of-the-art random walk systems and greatly reduce the I/O cost.

In the future, we would like to explore promising directions of second-order random walks. The current graph partitioning is quite simple, so we plan to design carefully graph partitions in order that random walkers should be trapped for long times in good partitions. Besides, we note that cache stall is also a performance bottleneck. The in-memory optimization of the second-order random walk is another attractive study to follow.

Acknowledgments

This work was supported in part by the Key Program of the National Natural Science Foundation of China (Grant No. 61832020), the Major Program of the National Natural Science Foundation of China (Grant No. 82090044), the Joint Funds of the National Natural Science Foundation of China (Grant No. U22A2027), and the Science Fund for Creative Research Groups of the National Natural Science Foundation of China (Grant No. 61821003). We are grateful to our shepherd, Călin Iorgulescu, and the anonymous reviewers for their constructive comments and suggestions.

References

[1] Twitter. http://an.kaist.ac.kr/traces/WWW2010.html.

[2] Friendster. https://snap.stanford.edu/data/com-Friendster.html.

[3] UK-Union. https://law.di.unimi.it/webdata/uk-union-2006-06-2007-05.

[4] CrawlWeb. http://webdatacommons.org.

[5] Graph500. https://graph500.org.

[6] Brad Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan Thompson, Divij Vaidya, and Shawn Wang. Amazon Neptune: Graph Data Management in the Cloud. *ISWC Posters & Demonstrations, Industry and Blue Sky Ideas Tracks*, 2018.

[7] Spatial and Graph Analytics with Oracle Database 19c. Technical report, Oracle, 2019.

[8] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[9] Amazon Neptune. https://aws.amazon.com/cn/neptune.

[10] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

[11] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast RandomWalk with Restart and Its Applications. In *Sixth international conference on data mining (ICDM)*, pages 613–622, 2006.

[12] Glen Jeh and Jennifer Widom. SimRank: A Measure of Structural-Context Similarity*. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 538–543, 2002.

[13] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 701–710, 2014.

[14] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 855–864, 2016.

[15] Amy N. Langville and Carl D. Meyer. Google's PageRank and Beyond: The Science of Search Engine Rankings. *Princeton university press*, 2011.

[16] Paul A.Gagniuc. Markov Chains: From Theory to Implementation and Experimentation. *John Wiley & Sons*, 2017.

[17] Denis R. Newman-Griffis and Eric Fosler-Lussier. Second-Order Word Embeddings from Nearest Neighbor Topological Features. *arXiv preprint arXiv: 1705.08488*, 2017.

[18] Wenyi Tang, Guangchun Luo, Yubao Wu, Ling Tian, Xu Zheng, and Zhipeng Cai. A Second-Order Diffusion Model for Influence Maximization in Social Networks. *IEEE Transactions on Computational Social Systems*, 6(4):702–714, 2019.

[19] Yubao Wu, Yuchen Bian, and Xiang Zhang. Remember Where You Came From: On The Second-Order Random Walk Based Proximity Measures. *Proceedings of the VLDB Endowment*, 10(1):13–24, 2016.

[20] Xueting Liao, Yubao Wu and Xiaojun Cao. Second-Order CoSimRank for Similarity Measures in Social Networks. In *2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[21] Martin Rosvall, Alcides V. Esquivel, Andrea Lancichinetti, Jevin D. West, and Renaud Lambiotte. Memory in Network Flows and its Effects on Spreading Dynamics and Community Detection. *Nature communications*, 5(1):1–13, 2014.

[22] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[23] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 376–386, 2015.

[24] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 507–522, 2016.

[25] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen and Weimin Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX annual technical conference (USENIX ATC)*, pages 125–137, 2017.

[26] Aapo Kyrola. DrunkardMob: Billions of Random Walks on Just a PC. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 257–264, 2013.

[27] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 559–571, 2020.

[28] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. An I/O-Efficient Disk-based Graph System for Scalable Second-Order RandomWalk of Large Graphs. *Proceedings of the VLDB Endowment*, 15(8):1619–1631, 2022.

[29] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[30] Ruizhi Li, Xiaoli Wu, Huan Liu, Jun Wu, and Minghao Yin. An Efficient Local Search for the Maximum Edge Weighted Clique Problem. *IEEE Access*, 6:10743–10753, 2018.

[31] Uriel Feige. Approximating Maximum Clique by Removing Subgraphs. *SIAM Journal on Discrete Mathematics*, 18(2):219–225, 2004.

[32] Sarab Almuhaideb, Najwa Altwaijry, Shahad AlMansour, Ashwaq AlMklaf, AlBandery Khalid AlMojel, Bushra AlQahtani, and Moshail AlHarran. Clique Finder: A Self-Adaptive Simulated Annealing Algorithm for the Maximum Clique Problem. *International Journal of Applied Metaheuristic Computing (IJAMC)*, 13(2):1–22, 2022.

[33] Xiutang Geng, Jin Xu, Jianhua Xiao and Linqiang Pan. A Simple Simulated Annealing Algorithm for the Maximum Clique Problem. *Information Sciences*, 177(22):5064–5071, 2007.

[34] Andrea Grosso, Marco Locatelli, and F Della Croce. Combining Swaps and Node Weights in an Adaptive Greedy Approach for the Maximum Clique Problem. *Journal of Heuristics*, 10(2):135–152, 2004.

[35] Ernst Althaus, Markus Blumenstock, Alexej Disterhoft, Andreas Hildebrandt, and Markus Krupp. Algorithms for the Maximum Weight Connected k-Induced Subgraph Problem. In *International Conference on Combinatorial Optimization and Applications*, pages 268–282, 2014.

[36] D. Janaki Ram, T. H. Sreenivas, and K. Ganapathy Subranmaniam. Parallel Simulated Annealing Algorithms. *Journal of parallel and distributed computing*, 37(2):207–212, 1996.

[37] Jun Gu and Xiaofei Huang. Efficient Local Search With Search Space Smoothing: A Case Study of the Traveling Salesman Problem (TSP). *IEEE Transactions on Systems, Man, and Cybernetics*, 24(5):728–735, 1994.

[38] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. ThunderRW: An In-Memory Graph Random Walk Engine. *Proceedings of the VLDB Endowment*, 14(11):1992–2005, 2021.

[39] Ke Yang, Xiaosong Ma, and Saravanan. Random Walks on Huge Graphs at Cache Efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 311–326, 2021.

[40] Yingxia Shao, Shiyue Huang, Xupeng Miao, Bin Cui, and Lei Chen. Memory-Aware Framework for Efficient Second-Order RandomWalk on Large Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1797–1812m 2020.

[41] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[42] Lijun Chang. Efficient Maximum Clique Computation over Large Sparse. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 529–538, 2019.

[43] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[44] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Danny Bickson. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI)*, pages 17–30, 2012.

[45] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX symposium on operating systems design and implementation (OSDI)*, pages 599–613, 2014.

[46] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

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

[48] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2016.

[49] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, 2013.

[50] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 429–442, 2019.