Fast Iterative Graph Computing with Updated Neighbor States

Yijie Zhou†, Shufeng Gong†, Feng Yao†, Hanzhang Chen†, Song Yu†, Pengxi Liu†,
Yanfeng Zhang✉†, Ge Yu†, Jeffrey Xu Yu‡
†*Northeastern University,* ‡*The Chinese University of Hong Kong*
{zhouyijie, yaofeng, chenhanzhang, yusong, liupengxi}@stumail.neu.edu.cn,
{gongsf, zhangyf, yuge}@mail.neu.edu.cn, yu@se.cuhk.edu.hk

***Abstract*—Enhancing the efficiency of iterative computation on graphs has garnered considerable attention in both industry and academia. Nonetheless, the majority of efforts focus on expediting iterative computation by minimizing the running time per iteration step, ignoring the optimization of the number of iteration rounds, which is a crucial aspect of iterative computation. We experimentally verified the correlation between the vertex processing order and the number of iterative rounds, thus making it possible to reduce the number of execution rounds for iterative computation. In this paper, we propose a graph reordering method, GoGraph, which can construct a well-formed vertex processing order effectively reducing the number of iteration rounds and, consequently, accelerating iterative computation. Before delving into GoGraph, a metric function is introduced to quantify the efficiency of vertex processing order in accelerating iterative computation. This metric reflects the quality of the processing order by counting the number of edges whose source precedes the destination. GoGraph employs a divide-and-conquer mindset to establish the vertex processing order by maximizing the value of the metric function. Our experimental results show that GoGraph outperforms current state-of-the-art reordering algorithms by 1.83× on average (up to 3.34×) in runtime. Compared with traditional synchronous computation, our method improves the iterative computations up to 6.30× in runtime.**

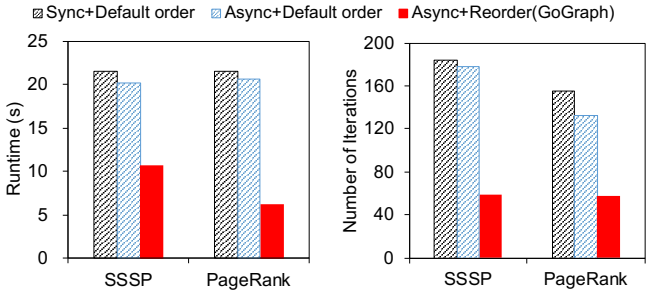
***Index Terms*—Graph reorder, Iterative computation, Asynchronous model, CPU Cache**

I. Introduction

Iterative computation is an important method for graph mining, such as PageRank [1], single source shortest path (SSSP), breadth first search (BFS), and so on. Since iterative computation involves traversing the entire graph multiple times to update vertex states until convergence, which is time-consuming. There have been many works that improve iterative computation from practice to theory [2]–[8].

Most of them [4], [6], [9], [10] accelerates iterative computations by reducing the runtime of each iteration round under the assumption that, for a given graph and iterative algorithm, the number of iteration rounds required to attain a specific convergence state is fixed. For example, in PageRank [11], it may take 20-30 iterations to converge; in SSSP, the number of iteration rounds is roughly the graph diameter [12]. However, in practice, the number of iteration rounds for a given graph and iterative algorithm may exhibit substantial variation depending on the vertex update method. We elaborate on this point in the following observation.

The first two authors contributed equally to this paper.
Yanfeng Zhang✉ is the corresponding author.

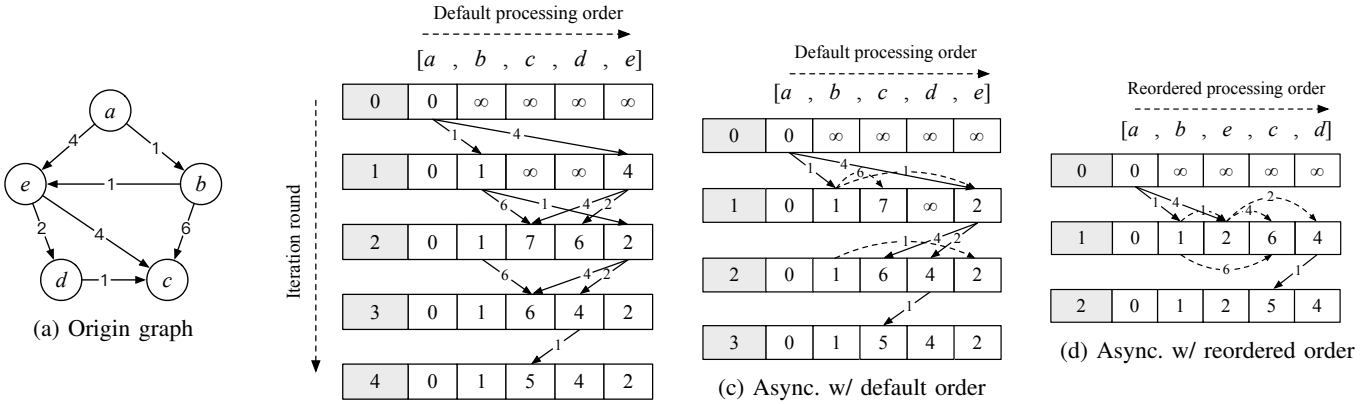


(a) Runtime (b) Iteration rounds

Fig. 1: Runtime & Number of iterations of SSSP and PageRank with different vertex updating modes (Sync. vs. Async.) and different vertex processing orders (default order vs. reordered with GoGraph) on wiki-2009 dataset

Observation. In general, traditional iterative algorithms are typically designed with a synchronous mode [6], [13], processing each vertex in a round-robin fashion. Specifically, to simplify the parallel semantics, in each round, every vertex is updated based on the state of its neighbors from the previous iteration. However, in some algorithms employing an asynchronous mode [14], [15], vertices can update their state using the neighbors' state from the current iteration rather than the previous one. This adjustment is beneficial because the updated neighbors' state is closer to convergence. Consequently, computations based on these updated state values yield results closer to convergence, thereby reducing the number of iteration rounds. Furthermore, we found that the vertex processing order further affects whether the latest state values can be utilized by its neighbors in the current iteration. It means that the vertex processing order is significant for iterative computation to reduce the number of iteration rounds.

Fig. 1 shows the runtime and number of iterations of SSSP and PageRank with different vertex updating modes and different vertex processing orders on wiki-2009 [16]. Both SSSP and PageRank have less runtime and fewer iterations in the asynchronous case. Furthermore, we reordered the vertices with GoGraph (our vertex reordering method), ensuring that as many vertices as possible can leverage the latest state values



(a) Origin graph

(b) Sync. w/ default order

(c) Async. w/ default order

(d) Async. w/ reordered order

Fig. 2: Iterative process and the number of iteration rounds generated by employing different iterative computation modes and vertex processing orders when running the SSSP algorithm, where the source vertex is a, the default processing order is alphabetical order based on vertex labels $[a, b, c, d, e]$, the reordered order is obtained by our method $[a, b, e, c, d]$

of their neighbors from the current round. It can be observed that, after reordering vertices, the advantage of asynchronous mode is significantly improved, and the number of iteration rounds and runtime are significantly reduced.

We use Fig. 2 as an example to illustrate the results in Fig. 1, where the iterative algorithm is SSSP and the source vertex is a. Fig. 2a gives the topology of the graph. In Fig. 2b, we employ synchronous iteration, where each vertex is updated using the values of its neighbors from the previous iteration. For example, in the first iteration, e's state is updated to 4 based on the initial state of a, while in the second iteration, e is updated to 2 based on the previous iteration state of b. It takes four iterations to achieve converged results. Fig. 2c employs an iterative asynchronous technique in which each vertex uses the updated states of its neighbors if they have been updated. For example, in the first iteration, e's state is updated to 2 based on the updated state of b from the current iteration. Compared to the synchronous mode in Fig. 2b, e changes and converges faster. Finally, the asynchronous mode achieves the same converged result in three rounds of iterations. We observed that the updates for c and d depend on the state value of e. In Fig. 2d, we rearrange the processing order by placing e before c and d. With this adjustment, each vertex ensures that all of its incoming neighbors are updated before it, leading to the convergence of the graph after two iterations.

Motivation. From Fig. 1 and Fig. 2, we can see that the processing order has a significant impact on the efficiency of asynchronous iterative computation. An effective vertex processing order can accelerate iterative computation. This motivates us to search for the optimal processing order to speed up iterative computation.

Our goal. As observed above, we can rearrange the vertices processing order so that when updating each vertex, its neighbors have already been processed and updated. Since the updated neighbor state from the current iteration make the vertex closer to the convergence state, the iterative computation will be accelerated. In this paper, we aim to investigate the design of a vertex reordering method capable of accelerating iterative computations while keeping other factors constant, such as the execution mode and task scheduling strategy of the graph processing systems, as well as result accuracy.

GoGraph. To achieve the above goal, we propose a graph reordering method, GoGraph, which can construct an efficient vertex processing order to accelerate the iterative computation. It has the following unique considerations and design.

Measure the quality of the processing order. Before formulating an effective reordering method, it is necessary to design a metric for quantifying the efficiency of the vertex processing order in accelerating iterative computation. While some intuitive metrics, such as runtime and the number of iteration rounds, can be regarded as benchmarks, they require completing iterative computations for meaningful results. Evaluating the quality of the vertex processing order becomes quite challenging in the absence of full iterative computation. Regarding the above problem, based on the theoretical guidance, we introduce a metric function that counts the number of edges whose sources are in front of destinations, which in turn reflects the quality of the processing order.

A divide-and-conquer method for vertex reordering. It is not trivial to find an optimal vertex processing order that enhances iterative computational efficiency. We demonstrate that finding the optimal processing order is an NP-hard problem. Therefore, directly establishing the optimal order for a given metric is impractical. GoGraph adopts the divide-and-conquer mindset. Initially, high-degree vertices are extracted from the graph to minimize their impact on the localization decisions of numerous low-degree vertices. Subsequently, the remaining graph is divided into smaller subgraphs to simplify the complexity of reordering, with a focus on intra- and inter-subgraph perspectives. The reordering process involves finding an optimal position for each vertex in the vertex processing order within the subgraph, and an optimal position for each subgraph in the subgraph processing order. This is achieved

by maximizing the value of the metric function. Finally, the high-degree vertices are taken into account to determine the complete vertex processing order. We theoretically prove the effectiveness and efficiency of GoGraph.

To summarize, we make the following contributions:

1) *Metric Function*. A novel metric function $\mathcal{M}(\cdot)$ is proposed to measure the efficiency of processing order in accelerating iterative computation. We theoretically prove the effectiveness of $\mathcal{M}(\cdot)$ (Section III).

2) *GoGraph*. An heuristic graph reordering method GoGraph is proposed, that can accelerate iterative computation significantly. We also provide an efficient implementation of GoGraph (Section IV).

3) *Evaluation*. We evaluate the effectiveness of GoGraph in accelerating iterative computation with a comprehensive experiment (Section V).

Furthermore, we first provide some preliminary foundational knowledge and some based definitions in Section II. And, finally, we list a comprehensive related work in Section VI and conclude this paper in Section VII.

II. Preliminaries

This section will introduce the fundamental knowledge related to our work.

Graphs. Consider a directed graph denoted as $G(V, E)$, where V and E are the set of vertices and edges. Given a vertex v, $IN(v) = \{u \mid (u, v) \in E\}$ and $OUT(v) = \{w \mid (v, w) \in E\}$ represent the set of incoming neighbors (abbr. *in-neighbors*) and outgoing neighbors (abbr. *out-neighbors*) of v, respectively, $|IN(v)|$ and $|OUT(v)|$ are the number of incoming neighbors (abbr. *in-degree*) and outgoing neighbors (abbr. *out-degree*), respectively.

Vertex processing order. The vertex processing order $O_V = [v_0, \cdots, v_{|V|-1}]$ is one permutation of the vertices V in G. There are $|V|!$ permutations for the graph with $|V|$ vertices. We aim to find one vertex permutation that accelerates the iterative computation as much as possible.

Ordinal number. We define the *ordinal number* of a vertex v as its position in the processing order, denoted as $p(v)$. The value of $p(v)$ ranges from 0 to $|V| - 1$. In each iteration, if a vertex u is processed before vertex v, then its ordinal number $p(u)$ is smaller than that of v, i.e., $p(u) < p(v)$, indicating that a lower ordinal number corresponds to an earlier processing order. For example, the ordinal numbers corresponding to the five vertices a, b, c, d, e in Fig. 2d are 0, 1, 2, 3, 4, with $p(e) = 4$. However, in the processing order shown in Fig. 2d, as the vertices are reordered, $p(e)$ is changed to 2.

Positive/Negative edge. An edge (u, v) is defined as a *positive edge* if the ordinal number of the source u is smaller than that of the destination v, i.e., $p(u) < p(v)$. Otherwise, edge (u, v) is a *negative edge*, i.e., $p(u) > p(v)$. In the case of the positive edges, the source has been updated when the destination is being processed. Using the updated states of the source to update the destination may speed up the convergence of the destination. On the contrary, in the case of the negative edges, when the destination is updated, the source has not been updated yet, so it has to use the state of the source from the previous iteration.

Iterative Computation. In each round of iterative computation, each vertex is updated using function $\mathcal{F}(\cdot)$, where the input of $\mathcal{F}(\cdot)$ is the set of the states of its incoming neighbors. In the k-th round of iteration, the update of the vertex v can be expressed as follows,

$$x_v^k = \mathcal{F}\big(\{x_u^{k-1} | u \in IN(v)\}\big), \tag{1}$$

where x_v^k is the state value of v after the k-th iteration. When processing vertex v, it accumulates the most recent state values of its in-neighbors u and subsequently applies the function $\mathcal{F}(\cdot)$ based on these collected values. In synchronous iteration, the state values of u are uniformly updated at the end of each round, so the state values of u that vertex v gathers in the k-th round originate from the $k-1$-th round. To illustrate this iterative process, we will employ the PageRank and SSSP algorithms as specific examples.

$$\texttt{PageRank:}\quad x_v^k = \sum_{u \in IN(v)} x_u^{k-1} \cdot d/OUT(u),$$
$$\texttt{s.t., } d \texttt{ is the damping factor.}$$

$$\texttt{SSSP:}\quad x_v^k = \min\{x_v^{k-1}, x_u^{k-1} + d(u, v) | u \in IN(v)\},$$
$$\texttt{s.t., } d(u, v) \texttt{ is the distance between } u \texttt{ and } v.$$

As discussed in Section I, using the updated state of the neighbors from the current iteration can accelerate iterative computation. In the case of neighbors that have not yet been processed, we continue to use their states from the previous iteration. The update function in Eq. 1 can be reformulated as Eq. 2.

$$\begin{aligned} x_v^k = \mathcal{F}\big(&\{x_{u_1}^k | u_1 \in IN(v), p(u_1) < p(v)\} \\ &\cup \{x_{u_2}^{k-1} | u_2 \in IN(v), p(u_2) > p(v)\}\big). \end{aligned} \tag{2}$$

In the k-th round of the Eq. 2, u_1 with ordinal numbers smaller than v are updated, while u_2 with ordinal numbers larger than v remain unchanged. The vertex state update examples for PageRank and SSSP can also be modified accordingly:

$$\begin{aligned} \texttt{PageRank:}\quad x_v^k = &\sum_{u_1 \in IN(v), p(u_1) < p(v)} x_{u_1}^k \cdot d/OUT(u_1) + \\ &\sum_{u_2 \in IN(v), p(u_2) > p(v)} x_{u_2}^{k-1} \cdot d/OUT(u_2), \\ &\texttt{s.t., } d \texttt{ is the damping factor} \end{aligned}$$

$$\begin{aligned} \texttt{SSSP:}\quad & x_v^k = \min\{x_v^{k-1}, x_{u_1}^k + d(u, v), x_{u_2}^{k-1} + d(u, v) | \\ & u_1, u_2 \in IN(v), p(u_1) < p(v), p(u_2) > p(v)\}, \\ & \texttt{s.t., } d(u, v) \texttt{ is the distance between } u \texttt{ and } v; \\ & u_1 \texttt{ is processed before } v \texttt{ and } u_2 \texttt{ is after } v. \end{aligned}$$

In practice, the vertex update method in Eq. 2 has been applied in many fields, such as Gauss-Seidel iteration [17] in linear algebra, Adsorption [18], Katz metric [19], SimRank [20], Belief propagation [21] and so on [22], [23].

III. Problem Statement

Before introducing the vertex reordering method, we first clarify the properties that function $\mathcal{F}(\cdot)$ required to have. Next, we formally define the vertex reordering problem and introduce a metric function designed to assess the efficiency of processing order in accelerating iterative computation. Finally, we propose the objective function of our paper based on the metric function.

In fact, if utilizing the updated state of neighbors to accelerate the iterative algorithm, the vertex states change monotonically increasing or decreasing, i.e., the update function $\mathcal{F}(\cdot)$ should be a monotonically increasing function.

Monotonic. In iterative computation, considering a monotonically increasing function $\mathcal{F}(\cdot)$, the vertex states progressively decrease (or increase), moving closer to a convergent state. During the update of each vertex, its state value diminishes (or grows), advancing towards convergence, especially when the state value of its neighbors is smaller (or larger). Then, we have the following inequality

$$\begin{aligned} &\check{x}_v = \mathcal{F}(x_{u_1}, \cdots \check{x}_{u_i}, \cdots x_{u_{|IN(v)|}}) \\ &\leq \hat{x}_v = \mathcal{F}(x_{u_1}, \cdots \hat{x}_{u_i}, \cdots x_{u_{|IN(v)|}}) \\ &if \quad \check{x}_{u_i} \leq \hat{x}_{u_i} \end{aligned} \tag{3}$$

where $\{u_1, \cdots u_i, \cdots u_{IN(v)}\}$ is the set of v's incoming neighbors, $\check{x}_v, \check{x}_{u_i}$ and $\hat{x}_v, \hat{x}_{u_i}$ are two state values of v and u_i respectively.

There is a broad range of iterative algorithms with monotonic vertex update functions, including SSSP , Connected Components (CC) [24], Single-Source Weighted Shortest Path (SSWP) [25], PageRank, Penalized Hitting Probability (PHP) [26], Adsorption [27], and more. Monotonic property provides a wider optimization space for iterative computation and has been emphasized in many graph analysis works [14], [22], [28], [29]. We also organize our vertex reordering method according to the monotonic function $\mathcal{F}(\cdot)$. The subsequent lemma is derived directly from the monotonicity of $\mathcal{F}(\cdot)$.

Lemma 1. *Given the monotonically increasing function $\mathcal{F}(\cdot)$ of an iterative algorithm, $\check{x}_v = \mathcal{F}(x_{u_1}, \cdots \check{x}_{u_i}, \cdots x_{u_{|IN(v)|}})$ and $\hat{x}_v = \mathcal{F}(x_{u_1}, \cdots \hat{x}_{u_i}, \cdots x_{u_{|IN(v)|}})$ are two state values of v, where $\{u_1, \cdots u_i, \cdots u_{|IN(v)|}\}$ are the incoming neighbor of v, $\check{x}_{u_i}$ and $\hat{x}_{u_i}$ are two state values of i-th incoming neighbors of v, x^*_v and $x^*_{u_i}$ are the converged states of v and u_i respectively. Then we have*

$$|x^*_v - \check{x}_v| \leq |x^*_v - \hat{x}_v| \quad if \quad |x^*_{u_i} - \check{x}_{u_i}| \leq |x^*_{u_i} - \hat{x}_{u_i}| \tag{4}$$

Proof. There are two cases in the proof, 1) the vertex state continues to increase, and 2) the vertex state continues to decrease.

When the state values of the vertices continue increasing during the iterative computation, we have $x^*_{u_i} \geq max\{\check{x}_{u_i}, \hat{x}_{u_i}\}$ and $x^*_v \geq max\{\check{x}_v, \hat{x}_v\}$. Since $|x^*_{u_i} - \check{x}_{u_i}| \leq |x^*_{u_i} - \hat{x}_{u_i}|$, then we have $\check{x}_{u_i} \geq \hat{x}_{u_i}$. According to inequation 3, we have $\check{x}_v \geq \hat{x}_v$. It means that $\check{x}_v$ is closer to the converged state, i.e., $|x^*_v - \check{x}_v| \leq |x^*_v - \hat{x}_v|$.

Similarly, when the state values of the vertices continue decreasing during the iterative computation, we have $x^*_{u_i} \leq min\{\check{x}_{u_i}, \hat{x}_{u_i}\}$ and $x^*_v \leq min\{\check{x}_v, \hat{x}_v\}$. Since $|x^*_{u_i} - \check{x}_{u_i}| \leq |x^*_{u_i} - \hat{x}_{u_i}|$, then we have $\check{x}_{u_i} \leq \hat{x}_{u_i}$. According to inequation 3, we have $\check{x}_v \leq \hat{x}_v$. It means that $\check{x}_v$ is closer to the converged state, i.e., $|x^*_v - \check{x}_v| \leq |x^*_v - \hat{x}_v|$. □

Based on the above lemma, we propose the following theorem to explain why using the updated state of incoming neighbors from the current iteration accelerates iterative computation.

Theorem 1. *Given the graph G, monotonically increasing update function $\mathcal{F}(\cdot)$ and two processing orders $O^1_V = [a, \cdots v, u, \cdots z]$, $O^2_V = [a, \cdots u, v, \cdots z]$. There is an edge (u, v) and no edge (v, u) in G. The difference between O^1_V and O^2_V is the ordinal number of u and v, i.e., $p(u) > p(v)$ in O^1_V and $p(u) < p(v)$ in O^2_V, which results in O^2_V having one more positive edge than O^1_V. Then, we have*

$$\begin{aligned} &\hat{x}^k_v = \mathcal{F}_{O^1_V}(x^{k-1}_u, \cdots) \\ &\check{x}^k_v = \mathcal{F}_{O^2_V}(x^k_u, \cdots) \end{aligned} \quad and \quad |x^*_v - \hat{x}^k_v| \geq |x^*_v - \check{x}^k_v| \tag{5}$$

where x^*_v is the converged state of v, $\check{x}^k_v$ and $\hat{x}^k_v$ are the state values of v after k-th iteration resulted from O^1_V and O^2_V respectively.

Proof. There is an edge (u, v) and no edge (v, u) in G. Therefore, compared with O^1_V, in O^2_V, the ordinal number of v is larger than u and u has been updated before updating v. Thus when updating v, we can use the updated state of u from the current iteration, i.e., $\hat{x}^k_v = \mathcal{F}_{O^1_V}(x^k_u, \cdots)$. While in O^1_V, we have to use the state of u from the previous iteration, i.e., $\check{x}^k_v = \mathcal{F}_{O^2_V}(x^{k-1}_u, \cdots)$.

Since $\mathcal{F}(\cdot)$ is a monotonically increasing function, the states of the vertices gradually progress toward the convergence states. The state of updated u is closer to converged state, i.e., $|x^*_u - x^{k-1}_v| \geq |x^*_u - x^k_v|$. According to the Lemma 1, we have $|x^*_v - \hat{x}^k_v| \geq |x^*_v - \check{x}^k_v|$. □

The theorem presented above highlights that leveraging the updated state of incoming neighbors from the current iteration accelerates iterative computation. The more updated incoming neighbors are involved, corresponding to a higher count of positive edges, results in a faster convergence of the vertex. This implies that the speed of vertex convergence is intricately linked to the order in which vertices are processed. Consequently, the promotion of iterative computation can be achieved through the adoption of an efficient processing order, ultimately reducing the required number of iteration rounds. In summary, the problem addressed in this paper can be formulated as follows.

Problem Formulation. Find an optimal graph processing order, denoted as $O_V = \mathcal{R}(G)$, that can minimize the number of iteration rounds k required by the iterative algorithm when performing $\mathcal{F}(\cdot)$ with O_V, i.e.,

$$O_V = \arg\min_k (|X^*_V \ominus \left(X^k_V = \mathcal{F}^k_{\mathcal{R}(G)}(X^0_V)\right)| \leq \epsilon). \tag{6}$$

where $X_V^* = \{x_v^* | v \in V\}$ is the state value set of converged vertex, $X_V^k = \mathcal{F}_{\mathcal{R}(G)}^k(X_V^0)$ is the set of vertex states after k iterations using function $\mathcal{F}(\cdot)$ with $O_V = \mathcal{R}(G)$ as the processing order, $\mathcal{R}(G)$ is the graph reordering function that can return a permutation of vertices, $\epsilon \geq 0$ is the maximum tolerable difference between the vertex convergence state and the real state, $\ominus$ computes the difference between X_V^* and X_V^k. In general, there are two implementations of $|X_V^* \ominus X_V^k|$, i) $\max(|x_v^* - x_v^k|, v \in V)$, e.g., in SSSP, ii) $\sum_{v \in V}(|x_v^* - x_v^k|)$, e.g., in PageRank.

However, before iterative computation, it is impossible to know the number of iterations for a given processing order O_V. Thus, we are not able to know which processing order O_V returned by $\mathcal{R}(G)$ is optimal. Therefore, it is required to design another metric to measure the quality of processing orders.

As observed in Section I and supported by Theorem 1, the processing of each vertex shows that more updated incoming neighbors corresponds to fewer iteration rounds and shorter runtime in iterative computation. Based on this insight and intuition, we propose the following measure function.

Metric Function. Given a graph processing order O_V, we use the following measure function to measure the efficiency of processing order O_V in accelerating iterative computation.

$$\begin{aligned} &\mathcal{M}(O_V) = \sum_{v \in V} \sum_{u \in IN(v)} \chi(u,v) = \sum_{(u,v) \in E} \chi(u,v) \\ &where \\ &\chi(u,v) = \begin{cases} 1 & if \quad p(u) < p(v), \\ 0 & if \quad p(u) > p(v). \end{cases} \end{aligned} \tag{7}$$

It can be seen that $\mathcal{M}(O_V)$ counts the sum of incoming neighbors with smaller ordinal numbers for each vertex. In other words, the value of $\mathcal{M}(O_V)$ is the number of edges where the source vertex has a smaller ordinal number than the destination vertex, indicating positive edges. The function $\mathcal{M}(O_V)$ achieves its maximum value when all vertices' incoming neighbors are in front of them, i.e., the incoming neighbors have smaller ordinal numbers. In this scenario, as all incoming neighbors are positioned in front of each vertex, updating each vertex ensures that its incoming neighbors have been updated and are closer to the convergence state. Leveraging the updated state of incoming neighbors propels the vertex closer to convergence, thereby accelerating the iterative computation.

Based on the measure function, we propose the following objective function.

Objective Function. Since the value of $\mathcal{M}(\cdot)$ serves as a metric to measure the efficiency of processing order in accelerating iterative computation, the objective of this paper is to identify the optimal graph processing reorder, denoted as $O_V = \mathcal{R}(G)$, that maximizes the $\mathcal{M}(\cdot)$ value.

$$O_V = \underset{\mathcal{R}(G)}{\arg\max}\ \mathcal{M}(\mathcal{R}(G)), \tag{8}$$

where $\mathcal{R}(G)$ returns a permutation of vertices in V.

NP-hard & NP-approximate. Derived from the objective function, our goal can be seen as identifying an optimal processing order that maximizes the number of positive edges. This is similar to the *topological sort*. The topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering. Therefore, if the graph is a directed acyclic graph, then we can use the topological sorting algorithm [30] to reorder the vertices. In this case, $\mathcal{M}(\mathcal{R}(G))$ will achieve the maximum value $|E|$.

A more viable approach involves first generating a directed acyclic subgraph from the cyclic graph by selectively removing edges, followed by the application of the topological sorting algorithm. Since the primary objective of reordering is to enhance the number of positive edges, preserving as many edges as possible in the generated directed acyclic subgraph is crucial. This problem is known as the Maximum Acyclic Subgraph (MAS) problem, and it has been demonstrated to be both NP-hard and NP-approximate [31]–[33]. Consequently, maximizing the value of $\mathcal{M}(\mathcal{R}(G))$ is inherently an NP-hard and NP-approximate problem.

In theory, [32] has proven that the lower bound of the number of edges that can be preserved in acyclic subgraphs is $|E|/2$. However, in practice, there have been many works [32], [34]–[37] that have demonstrated the ability to identify larger directed acyclic subgraphs. Specifically, [38]–[40] derive a directed acyclic subgraph by deleting vertices or edges from Conditional Preference Networks, and [37] identifies the maximum acyclic subgraph based on matrix. Despite these achievements, these methods are not suitable for our problem. They are either designed for specific scenarios or are impractical for large-scale graph data.

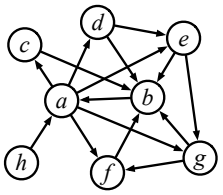
On the other hand, even if the maximum acyclic subgraph is obtained and topological sorting is employed to derive the processing order that maximizes $\mathcal{M}(\mathcal{R}(G))$, conventional topological sorting algorithms may overlook the neighbor relationships between vertices. This oversight can result in two connected vertices being positioned far apart from each other in the processing order. During iterative computation, frequent access to the neighbors of each vertex is common. The CPU cache hit ratio tends to decrease when a vertex is situated far away from its neighbors [41], which reduces computational efficiency. To overcome the above problems, in this paper, we introduce an efficient graph reordering method called GoGraph.

IV. Reordering Method

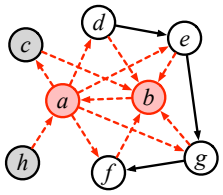
In this section, we introduce the vertex reordering algorithm, GoGraph, an efficient and effective vertex reordering method.

A. GoGraph

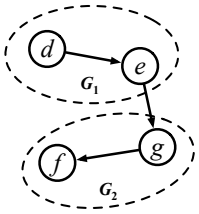
Due to the complex links between vertices in the graph, the change in the ordinal number of a vertex may result in some changes in edges, i.e., some positive edges become negative and some negative edges become positive, which result in an unpredictable $\mathcal{M}(\cdot)$. Therefore, it is difficult to reorder the



(a) The initial graph.



(b) Extract high-degree vertices.



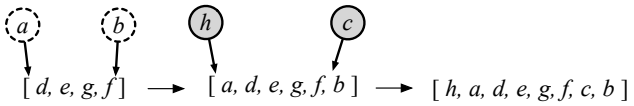
(c) Divide the remaining vertices

O_{V_1}=[d, e] O_{V_2}=[g, f]

(d) Reordering vertices intra-subgraphs.

[[d, e], [g, f]] → O_V= [d, e, g, f]

(e) Reordering vertices inter-subgraphs.



(f) Inserting high-degreed/isolated vertices into the processing order.

Fig. 3: An illustrative example of GoGraph

vertex from the perspective of the whole graph. In GoGraph, we adopt a divide-and-conquer method for vertex reordering. The method initially extracts the high-degree vertices from the graph, then divides the graph into subgraphs and reorders the vertices within and between these subgraphs, yielding the complete vertex processing order.

The overview of GoGraph is illustrated in Fig. 3. It comprises following steps, 1) *extract high-degree & isolated vertices*, 2) *divide other vertices*, 3) *reorder vertices within subgraphs*, 4) *reorder subgraphs* and 5) *insert high-degree & isolated vertices.*

Next, we will introduce the intuitions and details of each step.

Extract high-degree vertices. It is widely acknowledged that most real-world graphs exhibit a power-law property, where a very small number of vertices have extremely high degrees, while the majority of vertices have lower degrees. This property poses a challenge for the vertex reordering process. In the process of reordering, the placement of high-degree vertices in the reordering subsequence significantly influences the positioning decisions of numerous lower-order vertices. Thus we first remove the high-degree vertice V_{HD} and their edges E_{HD} from the graph.

We take Fig. 3a as an example to illustrate this point. If the high-degree vertices are not removed, we may first reorder a and b and assign a higher ordinal number to a than b to make edge (b, a) positive. Consequently, we obtain a temporary processing order $[b, a]$. Then we insert other vertices into $[b, a]$. In order to achieve the optimal vertex processing order, when inserting each vertex, we try to maximize $\mathcal{M}(\cdot)$ value. Ultimately, the optimal vertex processing order is $O_V^1 = [d, e, c, b, h, a, g, f]$. On the contrary, after removing a and b, we first reorder d, e, f, and g, where h and c become isolated and also be removed. It is evident that the optimal processing order is $[d, e, g, f]$. Finally, we insert a, b and c, h into $[d, e, g, f]$ and obtain the processing order $O_V^2 = [h, a, c, d, e, g, f, b]$. The $\mathcal{M}(\cdot)$ values of these two orders are $\mathcal{M}(O_V^1) = 10$ and $\mathcal{M}(O_V^2) = 14$. Based on the previous definition of $\mathcal{M}(\cdot)$, O_V^2 is a better processing order. Although the edge (b, a) is sacrificed in O_V^2 due to a's higher ordinal number than b, it brings more positive edges.

It is notable that after removing the high-degree vertices and their edges from the graph, there will appear some isolated vertices that have no edges with other vertices. As illustrated in Fig. 3b, vertex c and h become isolated vertices after removing a, b since they only connect with a and b. Since isolated vertices have no edges connecting them to other vertices, they do not influence the reordering of other vertices. Therefore, we eliminate isolated vertices, denoted as V_I.

Divide other vertices. Due to the complex interconnection between vertices, although the complexity of the graph structure is reduced after removing high-degree vertices and isolated vertices, it is still difficult to reorder vertices from the perspective of the whole graph. Therefore, to simplify reordering, we divide the remaining graph into smaller subgraphs. Then design the reordering method considering both intra- and inter-subgraphs perspectives. The graph dividing result requires that there are as many edges within the subgraph as possible and as few edges between the subgraphs as possible. The reason is explained as follows.

Firstly, when reordering vertices within a subgraph, if there are few edges between vertices intra-subgraph and many edges between vertices inter-subgraph, the better local reordering results may not bring a better global reordering result. Instead, the result of global reordering is determined by the reordering result of vertices between subgraphs.

Secondly, [41] points out that the locality of vertices in the processing order has a great impact on CPU cache performance. In most graph algorithms, accessing a vertex often necessitates accessing its neighbors. For example, in PageRank, updating a vertex requires accessing the states of its incoming neighbors. Sequential accesses to the neighbor states of vertices stored in more distant locations in physical memory may result in serious cache misses. Therefore, vertices connecting each other should be as close as possible in the processing order. Thus, we should group the vertices that are closely connected into the same subgraphs.

Therefore, when dividing the graph, there should be as many edges as possible within the subgraph and as few edges as possible between subgraphs. This is similar to the purpose of graph community detection or graph partitioning in distributed computing, so we can employ these methods such as Louvain [42] or Metis [43]. Finally, we obtain a set of subgraphs of the graph $\{G_1, \cdots, G_K\}$, where $G_i = \{V_i, E_i\}$.

Reorder vertices within subgraphs. After dividing the graph,

we first reorder the vertices inside each subgraph G_i. To begin, we can randomly select a vertex as the initial vertex of the processing order O_{V_i} formed by the vertices V_i in the subgraph. In practice, the initial vertex always has the smallest in-degree, since such vertex tends to rank at the front of the processing order. We then select a vertex from the remaining vertices to insert into the processing order. We prefer selecting the vertex v with BFS, so that v has better locality with the vertices in $O^c_{V_i}$.

For each selected vertex v, we search the optimal insertion position from the tail to the head of $O^c_{V_i}$ that maximizes the $\mathcal{M}(\mathcal{R}(G_i))$ value based on the position of the vertices that are in $O^c_{V_i}$. Intuitively, the essence of inserting vertices is to maximize the number of positive edges. It is worth mentioning that the overhead of such sequential attempts is not substantial, since we only need to search at locations near v's neighbors in $O^c_{V_i}$ (the details will be discussed in implementation, Section IV-C)

Reorder subgraphs. Next, we reorder the subgraphs to merge the order of vertices within subgraphs into a whole processing order. To make the vertices in the same subgraph continuous in the processing order, we treat the entire subgraph as a super vertex, which brings vertices within the same subgraph physically closer in memory, thereby reducing CPU cache misses. There will be edges between two super vertexes if there are edges between vertices in these two subgraphs, as shown in Fig. 3c, there is an edge between G_0 and G_1 since there is an edge (e, g). Then we reorder the super vertices formed by subgraphs using a similar method that reorders vertices within subgraphs. The difference is that there is a weight value on the edge between subgraphs, which is defined as the number of edges from one subgraph to another one, i.e., $w_{G_i,G_j} = |\{(u,v)|u \in G_i, v \in G_j\}|$. Then the objective function of ordering super vertices becomes $\mathcal{M}(O_P) = \sum_{G_i,G_j \in P} \chi_c(G_i, G_j)$, where $\chi_c(G_i, G_j) = w_{G_i,G_j}$ if $p(G_i) < p(G_j)$, else $\chi_c(G_i, G_j) = 0$, where $p(G_i)$ returns the ordinal number of super vertex formed by G_i.

After reordering subgraphs, the processing order of G' will be obtained by decompressing the super vertices formed by subgraphs.

Insert high-degree & isolated vertices. So far, we have reordered all vertices except for those with high degrees and isolated vertices. Since isolated vertices do not have connected edges with the vertices presently in the processing order, we prioritize inserting vertices with high degrees into the processing order, followed by the insertion of isolated vertices. We search for the optimal position to insert v from front to back, maximizing the increase in the $\mathcal{M}(\cdot)$ value. The insertion method is similar to the approach used when reordering vertices within a subgraph. For isolated vertices, we employ the same method as inserting high-degree vertices into the existing processing order.

Next, we analyze the effectiveness and efficiency of GoGraph theoretically.

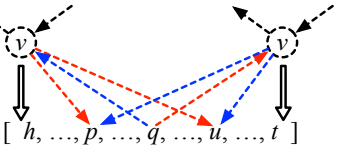


Fig. 4: Example of positive and negative edges when vertex v is inserted into the head or tail of O^c, where red dashed lines represent positive edges and blue dashed lines represent negative edges

B. Effectiveness

Randomly, the probability of each edge being positive edge is 1/2, and the $\mathcal{M}(\mathcal{R}(G))$ is $|E|/2$. According to the following analysis, it can be seen that the $\mathcal{M}(\mathcal{R}(G))$ value of the processing order obtained by GoGraph is no smaller than $|E|/2$, which is no worse than a random processing order.

Before the theoretical analysis, we first propose the following theorem.

Lemma 2. *Given a current processing order O^c, and a candidate vertex v to be inserted into O^c, after inserting v into the O^c using GoGraph, the value of $\mathcal{M}(O^c)$ will increase by at least $|E^c_v|/2$, where $|E^c_v|$ is the number of edges connecting v and vertices in O^c.*

Proof. There are two situations after v is inserted into O^c, i) v is inserted into the head or tail of O^c, and ii) v is inserted into another position of O^c.

We denote the neighbors of v that are in O^c are $N^c_v = \{IN(v) \cup OUT(v)\} \cap O^c$, then the edges between v and N^c_v are E^c_v, i.e.$|N^c_v| = |E^c_v|$.

For the first case, when v is inserted into the head (tail) of O^c, outgoing edges (incoming edges) of v in E^c_v are positive edges since v's ordinal number is smaller (larger) than its outgoing neighbors. Then, the incoming edges (outgoing edges) are negative. Thus, when v is inserted into head or tail of O^c, there are $max(|\{N^c_v \cap IN(v)\}|, |\{N^c_v \cap OUT(v)\}|)$ edges are positive edges, where $N^c_v \cap IN(v)$ and $N^c_v \cap OUT(v)$ are incoming neighbor and outgoing neighbors of v in O^c respectively. As we all know $max(|\{N^c_v \cap IN(v)\}|, |\{N^c_v \cap OUT(v)\}|) \geq (|\{N^c_v \cap IN(v)\}| + |\{N^c_v \cap OUT(v)\}|)/2 = |N^c_v|/2$. Therefore, the number of positive edges is no smaller than $|E^c_v|/2$.

For the second case, v is inserted into another position that maximizes the $\mathcal{M}(\cdot)$ value. It means that we find a better inserting position that makes the number of positive edges larger than inserting into the tail or head. Then the number of positive edges is larger than $|E^c_v|/2$.

In summary, after inserting v into O^c, at least $|E^c_v|/2$ edges are positive edges. □

For the example in Fig. 4, $(v,p), (q,v), (v,u)$ are edges of v that connect v and vertices in O^c. If v is inserted into the head of O^c, v's outgoing edges $\{(v,p), (v,u)\}$ are positive edges while incoming edge (q,v) is a negative edge. If v is inserted

into the tail, then (q, v) becomes a positive edge, while (v, p) and (v, u) become negative edges.

Based on the lemma, we have the following theorem.

Theorem 2. *After reordering vertices in graph $G(V, E)$, we obtain the processing order O_V. Then we have*

$$\mathcal{M}(O_V) \geq |E|/2 \tag{9}$$

Proof. During the reordering process, there are 4 types of edges, 1) the edges within each subgraph E_{V_i}, 2) the edges between subgraphs E_P, 3) the edges between high-degree vertices and the vertices in subgraphs E_{HD}, 4) the edges between isolated vertices and high-degree vertices E_{ISO}, i.e., $E = \bigcup_{i=1}^{N} E_{V_i} \cup E_P \cup E_{HD} \cup E_{ISO}$.

In each subgraph, when inserting each vertex v, $|E_v^c|$ only contains the edges connecting v and the vertices in the current local processing order $O_{V_i}^c$, not the edges connecting v and other vertices. Therefore, for each edge $(u, v) \in E_i$, $(u, v) \in E_u^c$ if v is inserted into $O_{V_i}^c$ before u otherwise $(u, v) \in E_v^c$. Thus we have $E_{V_i} = \bigcup_{v \in V_i} N_v^c$. According to Lemma 2, we have $\mathcal{M}(O_{V_i}) \geq \sum_{v \in V_i} |E_v^c|/2 = |E_i|/2$.

Similarly, for the other 3 types of edges, the number of positive edges between subgraphs is at least $|E_P|/2$, the number of positive edges between high-degree vertices and the vertices in subgraphs is at least $|E_{HD}|/2$, the number of positive edges between isolated vertices and high-degree vertices is at least $|E_{ISO}|/2$.

Finally, we have the number of positive edges is at least $\sum_{i=1}^{N} |E_i|/2 + |E_P|/2 + E_{HD}/2 + |E_{ISO}|/2 = |E|/2$, i.e., $\mathcal{M}(O_V) \geq |E|/2$. □

C. Implementation

GoGraph adopts a divide-and-conquer idea, and a sketch of its implementation is shown in Algorithm 1.

In terms of the core idea of GoGraph, the essence lies in finding an optimal position of the processing order for vertex v that maximizes $\mathcal{M}(\cdot)$, then inserting the candidate v into this position. However, before obtaining the global processing order, it is impractical to obtain the real ordinal number of each vertex, because with the vertex insertion, the real ordinal number value of the vertex will continue to change. Therefore, we use val to represent the ordinal number (line 1). The larger the val, the larger the oral number of the vertex in the processing order. Finally, we sort vertices according to val to derive the real processing order and the ordinal number of each vertex (line 36).

Before computing the val of vertices, we first extract the high-degree vertices (line 2). As a rule of thumb, we simply extract the top 0.2% vertices with the highest degree. Then we divide the graph (line 3) with the exiting graph partitioning method, such as Rabbit-Partition [44], Metis [43], and Louvain [42]. In our implementation, we use the graph partitioning method introduced in Rabbit. We proceed to compute the val of each vertex within G_i (line 4-8). It is notable that the vertex val is a local value within G_i. After computing the local vertex val, we treat each subgraph as a super-vertex (line 9-11), the

Algorithm 1: GoGraph Algorithm Sketch

Input: Graph $G(V, E)$
Output: Vertex processing order O_V

1 Init $v.val$ of each v with ∞; // val represents the value of the ordinal number
// ***Divide phase***
2 Extract high-degree vertices, isolated vertices and their edges G_{HD}, G_{ISO} from G, the remain vertices and edges form subgraph G';
3 Divided graph G' into K subgraphs $\{G_1, \cdots, G_K\}$ with graph partitioning or clustering method;
// ***Conquer phase***
4 **for** *each subgraph* $G_i(V_i, E_i)$ **do**
5 $O_{V_i} \leftarrow \emptyset$; // init the processing order of G_i
6 **for** *each vertex* v *in* V_i **do**
7 $v.val \leftarrow$GetOptVal(O_{V_i}, v)
8 $O_{V_i}.add(v)$;
// ***Combine phase: reorder subgraphs***
9 **for** *each subgraph* G_i **do**
10 create super vertex sv_i with V_i;
11 $sv_i.val \leftarrow \infty$; // $sv.val$ represents the value of the ordinal number of sv
12 **for** *each* sv_i **do**
13 **for** *each* sv_j **do**
 // compute the weight of edge
14 $w(sv_i, sv_j) \leftarrow |\{(u, v) | u \in G_i, v \in G_j\}|$
15 Construct a graph G_{sv} with vertices set $\{sv_i | 1 < i \leq K\}$ and edges set $\{(sv_i, sv_j) | 1 \leq i \leq K, 1 \leq j \leq K\}$;
16 $O_S \leftarrow \emptyset$;// init the processing order of G_{sv}
17 **for** *each super vertex* sv **do**
18 $sv.val \leftarrow$GetOptVal(O_S, sv)
19 $O_S.add(sv)$;
20 $O_V \leftarrow \emptyset$; // init the processing reorder of G
// ***decompose super vertices, update val***
21 Sort O_S in the ascending order of $vs.val$;
22 $mval_{pre} \leftarrow 0$;
23 **for** *each super vertex* $sv \in O_S[i]$ **do**
24 $mval_{cur} \leftarrow 0$;
25 **for** *each vertex* v *in* sv **do**
26 $v.val \leftarrow v.val + mval_{pre}$;
27 $mval_{cur} \leftarrow \max(mval_{cur}, v.val)$;
28 $O_V.add(v)$;
29 $mval_{pre} = mval_{cur}$;
// ***reorder high-degree & isolated vertices***
30 **for** *each vertex* v *in* V_{HD} **do**
31 $v.val \leftarrow$ GetOptVal(O_V, v);
32 $O_V.add(v)$;
33 **for** *each vertex* v *in* V_{ISO} **do**
34 $v.val \leftarrow$ GetOptVal(O_V, v);
35 $O_V.add(v)$;
36 Sort O_V in the ascending order of $v.val$;// obtain the final processing order

```
1  function GetOptVal(order O, vertex v)
2      N_v ← {IN(v) ∪ OUT(v)} ∩ O;
3      Sort N_v in the ascending order of v.val;
4      pe_v = |OUT(v) ∩ O|; // init the number of
           positive edges
5      val ← 0; // init the val
6      max_pe_v ← −∞; // the max number of postive
           edges
7      for each vertex N_v[i] in N_v do
8          if N_v[i] ∈ OUT(v) then
9              if v is super vertex then
10                 pe_v ← pe_v − w(v, N_v[i]);
11             else
12                 pe_v ← pe_v − 1;
13         else
14             if v is super vertex then
15                 pe_v ← pe_v + w(v, N_v[i]);
16             else
17                 pe_v ← pe_v + 1;
18         if max_pe_v < pe_v then
19             max_pe_v ← pe_v;
20             val ← (N_v[i].val + N_v[i + 1].val)/2;
                   // insert v between N_v[i] and
                   N_v[i + 1]
21     return val;
```

edges between super-vertices have weights that are equal to the number of edges between the subgraphs (line 14). Then we compute the val of each super vertex (line 17-19). After that, we sort the subgraphs ascending with val of the super vertex (line 21). Then, we unzip the super vertices and obtain the global val of each vertex in G', which is done by adding the maximum val of vertices in the previous subgraph to the val of each vertex for G_i (line 22-29).

The val is computed by finding the current optimal position in the current processing order, and taking the average of the val of the predecessor and successor to indicate a value in between. During the search for the optimal position, it is unnecessary to recompute the value of $\mathcal{M}(\cdot)$ for every potential insertion position of vertex v. This is due to the fact that for a vertex not connected to v, the count of positive and negative edges remains unchanged when v is inserted in front of or behind it. Consequently, the value of $\mathcal{M}(\cdot)$ also remains unchanged. As shown in Fig. 4, inserting v into the front or behind of h, the value of $\mathcal{M}(O^c)$ remains unchanged since (v, p) and (v, u) are always positive. Therefore, to find the best position in the current order, it is enough to count the number of positive edges pe_v when v is inserted into the front or behind each neighbor, which can be implemented as follows (shown in GetOptVal function).

Firstly, we extract v's neighbors that are in the processing order and form a neighbor sequence (line 2), then we sort the neighbors of vertex v in ascending order according to their val and form a sorted neighbor sequence (line 3). Then, we traverse each position in the neighbor sequence from the head or tail to find the optimal position one by one, and update the number of positive edges pe_v incrementally (lines 7-20), instead of recounting. If v is inserted into the head of the neighbor sequence, the initial value of $pe_v = |OUT(v)| \cap O_V$ (line 4). After moving the v to the back of the next neighbor, pe_v is updated. If the next neighbor is incoming neighbor, then $pe_v = pe_v + 1$ (lines 8-12). Otherwise, the $pe_v = pe_v - 1$ (lines 13-17). Note that if v is a super vertex, then the update granularity of pe_v is the weight of connected edges (line 10 and line 15). When v moves to the tail of the neighbor sequence, the value of $pe_v = |IN(v)| \cap O_V$. For the example in Fig. 4, we assume there is a neighbor sequence $[p, q, u]$. With v as the head, initially the $pe_v = 2$. After moving v behind p, $pe_v = 2 - 1 = 1$, since p is an outgoing neighbor. After v moves behind q, $pe_v = 1 + 1 = 2$, since q is an incoming neighbor. Once the optimal position is determined, we compute the val of v with its predecessor and successor (line 20).

V. Experiments

A. Experimental Setup

In default, the experiments are performed on a Linux server with Intel Xeon Gold 6248R 3.00GHz CPU, 98 GB memory, and it runs on 64-bit Ubuntu 22.04 with compiler GCC 7.5.

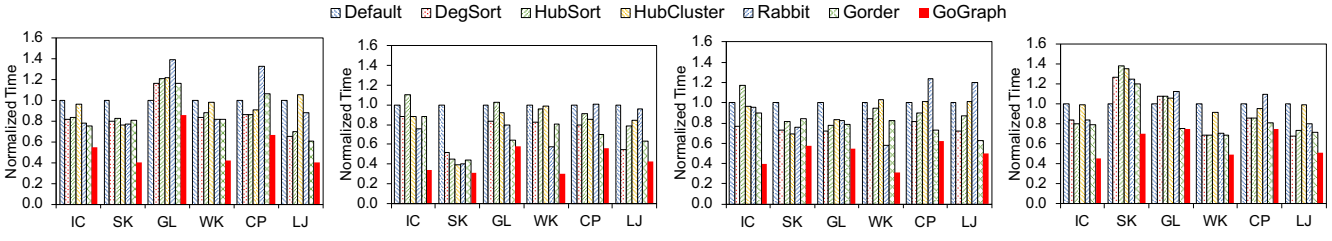
Graph Workloads. We use four typical graph analysis algorithms in our experiments, including PageRank [1], Single Source Shortest Path (SSSP), Breadth First Search (BFS), and Penalized Hitting Probability (PHP) [45]. When the difference between vertex state value in two consecutive iterations is less than 10^{-6}, PageRank and PHP are considered to have achieved convergence. For SSSP and BFS, the algorithms are considered to have reached convergence when all vertex states are no longer changing.

TABLE I: Datasets

Dataset	Vertices	Edges	Abbreviation
Indochina [16]	11,358	49,138	IC
SK-2005 [16]	121,422	36,7579	SK
Google [46]	875,713	5,241,298	GL
Wiki-2009 [16]	1,864,433	4,652,358	WK
Cit-Patents [47]	3,774,768	18,204,371	CP
LiveJournal [16]	4,033,137	27,972,078	LJ

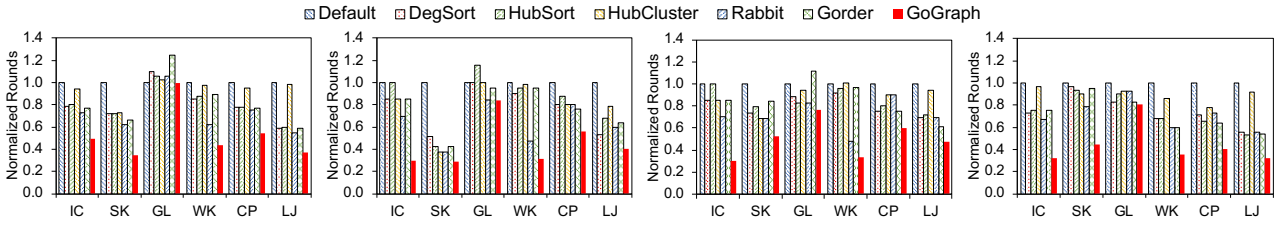
Datasets. Six real-world datasets are used in our experiments, including indochina-2004 [16], sk-2005 [16], Google [46], wikipedia-2009 [16], cit-Patents [47] and soc-livejournal [16]. The details of each dataset are outlined in Table I.

Competitors. We compare GoGraph with the 6 graph ordering methods listed, Default, Degree Sorting, Hub Sorting [48], Hub Clustering [49], Rabbit [44], and Gorder [41]. The default order employs the original IDs as the processing order.



(a) PageRank (b) SSSP (c) BFS (d) PHP

Fig. 5: The comparison of runtime



(a) PageRank (b) SSSP (c) BFS (d) PHP

Fig. 6: The comparison of iteration rounds

B. Overall Performance

We first compare GoGraph with the competitors in terms of the runtime and number of iteration rounds for each graph algorithm executed on graphs in Table I. The *Normalized* runtime and iteration rounds results are reported in Fig. 5 and Fig. 6, respectively, where the result of Default order is treated as the baseline, i.e., the Default finishes in unit time 1. It can be seen from Fig. 5, GoGraph outperforms others in all cases. Specifically, GoGraph achieves 2.10× speedup on average (up to 3.33× speedup) over Default, 1.66× speedup on average (up to 2.75× speedup) over Degree Sorting, 1.85× speedup on average (up to 3.24× speedup) over Hub Sorting, 1.93× speedup on average (up to 3.34× speedup) over Hub Clustering, 1.80× speedup on average (up to 2.42× speedup) over Rabbit, and 1.62× speedup on average (up to 2.68× speedup) over Gorder. On the other hand, in the measurement of the number of iterative rounds as shown in Fig. 6, GoGraph incurs the least number of iteration rounds on most tested conditions. Specifically, GoGraph reduces the number of iteration rounds on average by 52% (up to 71%) compared with Default, 39% (up to 65%) compared with Degree Sorting, 40% (up to 70%) compared with Hub Sorting, 45% (up to 68%) compared with Hub Clustering, 32% (up to 57%) compared with Rabbit, and 39% (up to 67%) compared with Gorder. It is worth noting that GoGraph does not appreciably reduce the number of iteration rounds in the PageRank result for the GL graph, demonstrating that the default order of GL graph is naturally a well-defined processing order. The gain of the corresponding GoGraph on the GL graph comes from the reduction of cache miss. Another observation is that Orignal, Degree Sorting, Hub Sorting, and Hub Clustering exhibit similar trends in runtime and number of iteration rounds, whereas Rabbit and Gorder do not. This distinction is because the former focuses on the effect of vertex processing order, while the latter concentrates on CPU cache optimization. GoGraph takes both into account in a comprehensive way.

C. Convergence comparison

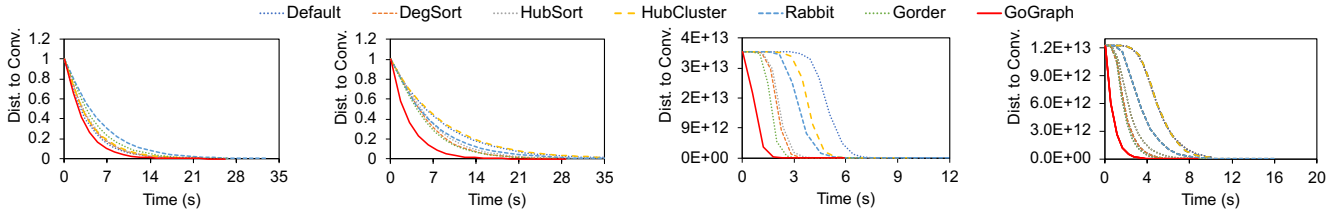
To evaluate the effect of GoGraph's reordering, we compared the convergence rates of GoGraph and its competitors. Our evaluation consists of running the PageRank and SSSP algorithms on CP and LJ graphs, both of which use different reordering algorithms. We use the absolute difference between the sum of vertex state values at convergence and the sum of all vertex state values at time t to represent the distance to convergence, mathematically expressed as: $dist_t = \left|\sum_{v \in V} x^* - \sum_{v \in V} x_t\right|$. Its trend over time is shown in Fig. 7. We can see that GoGraph achieves convergence faster in all cases. In achieving the same converged state, the GoGraph algorithm consumes only 59% of the average time used by other algorithms (with a minimum requirement of 37%). This efficiency is attributed to the advantages gained in each iteration of the vertex processing order constructed by GoGraph.

D. Impact of processing order in improving Async mode

To verify the impact of the asynchronous update mode and the processing orders on accelerating iterative computation, we compare the runtime of PageRank and SSSP on different graphs using synchronous update mode and default processing order (Sync + Def.), asynchronous update mode and default processing order (Async + Def.), asynchronous update mode and the processing order generated by GoGraph (Async + GoGraph). Fig. 8 shows the normalized results. It is shown that asynchronous updating mode can accelerate iterative computation compared with synchronous updating mode. And GoGraph achieves significant improvements, obtaining 1.56×-6.30× (3.04× on average) speedups.

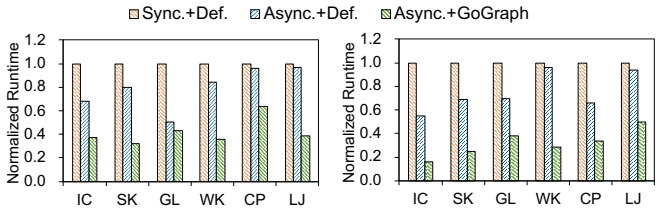
E. CPU Cache Miss

To figure out the effect of GoGraph on reducing cache miss, we ran PageRank algorithm on all graphs and recorded the cache misses. The results are shown in Fig. 9. Compared to its competitors, GoGraph can reduce the cache miss by 30%



(a) PageRank on CP (b) PageRank on LJ (c) SSSP on CP (d) SSSP on LJ

Fig. 7: The comparison of convergence speed



(a) PageRank (b) SSSP

Fig. 8: Impact of processing order in improving Async mode

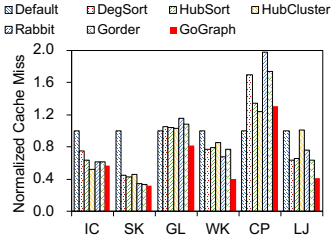


Fig. 9: Cache miss comparison

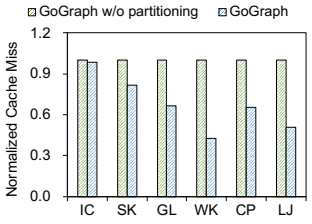


Fig. 10: The impact of partition on cache miss

on average, which means that we incur less I/O overhead when reading graphs from memory. This is attributed to our consideration of localization when dividing the graphs.

To evaluate the impact of graph partitioning on reducing cache misses, we recorded the cache misses of PageRank on different processing orders obtained by GoGraph with and without partitioning. The results are depicted in Fig. 10. It illustrates that the graph partition contributes to reducing 33% (up to 58%) cache misses.

F. Efficiency of Metric Function

The design of GoGraph revolves around maximizing the value of $\mathcal{M}(\cdot)$, in that the higher the value of $\mathcal{M}(\cdot)$, the fewer the number of iteration rounds. To verify this claim, we record $\mathcal{M}(\cdot)$ values of processing orders of CP graph after applying different reorder methods and the number of iteration rounds of various algorithms on these orders. The results are shown in Table II.

It can be seen that i) the larger the $\mathcal{M}(\cdot)$ value of the processing order is, the fewer iteration rounds the algorithm requires, which is in line with our claims and expectations. ii) The $\mathcal{M}(\cdot)$ value of processing order produced by GoGraph is the largest and the number of iteration rounds always are smallest. It means that the metric function $\mathcal{M}(\cdot)$ is effective

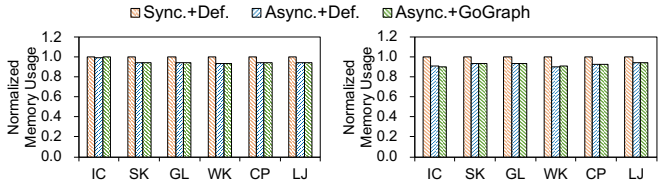
TABLE II: Metrics and Iteration Rounds of Various Algorithms After Applying Different Reorder Methods on CP.

Reorder method	$\mathcal{M}(\cdot)$	$\frac{\mathcal{M}(\cdot)}{\lvert E\rvert}$	Number of iteration rounds			
			PageRank	SSSP	BFS	PHP
Default	1,302,313	0.07	99	25	36	67
HubCluster	2,303,977	0.13	94	20	34	52
DegSort	3,623,082	0.20	77	20	25	48
HubSort	3,691,804	0.20	77	22	26	44
Gorder	5,875,924	0.32	76	19	22	43
Rabbit	8,883,616	0.49	75	20	25	49
GoGraph	13,871,315	0.76	54	14	17	27

for measuring the efficiency of processing order in accelerating iteration computations.

G. Memory Usage

Our method boosts iterative computation performance by reducing iteration counts without theoretically increasing memory overhead, compared to the baseline (synchronous update with default graph order). To evaluate memory overhead, we examined memory usage in three scenarios: synchronous update with default order (Sync. + Def.), asynchronous update with default order (Async. + Def), and asynchronous update reordered by GoGraph (Async. + GoGraph), shown in Fig. 11. Memory usage across these methods is similar. GoGraph improves iterative computation efficiency by reordering processing without extra data structures. However, the synchronous approach slightly increases memory overhead due to recording vertices' current and previous states.



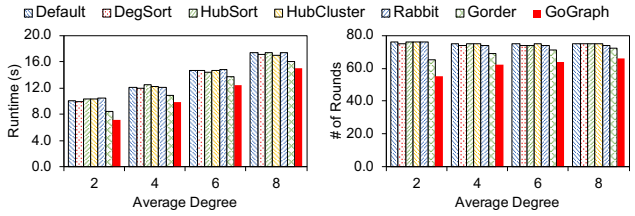
(a) PageRank (b) SSSP

Fig. 11: Memory usage of different iterative computations

H. Average Degrees

To evaluate the impact of average graph degrees on GoGraph's performance, we generated a series of graphs with different average degrees (2, 4, 6 and 8) using the Barabasi-Albert model [50] in NetworkX, each with 1,000,000 vertices. We applied different reordering methods and ran PageRank

on these graphs. Fig. 12 shows PageRank's runtime and iteration counts across datasets, showing GoGraph's superior performance in both aspects with varying average degrees.



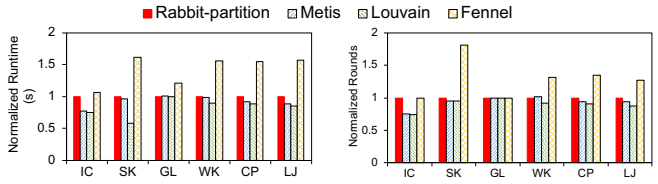
(a) Runtime (b) Number of Iteration Rounds

Fig. 12: The impact of different average degrees

Increased average degrees led to longer PageRank runtimes due to larger graph sizes, though iteration counts stayed similar. However, reordering methods were less effective on these synthetic graphs compared to real ones, as the default order of generated graphs is already more optimal than that of real graphs, diminishing the enhancements from GoGraph and others.

I. Partition Methods

In GoGraph, we use the graph partition method introduced in Rabbit (Rabbit-Partition) by default. To evaluate the effectiveness of various graph partitioning methods in GoGraph, we employed Metis [43], Louvain [42], and Fennel [51] for graph partitioning. Fig. 13 presents the normalized runtime and iteration counts of PageRank on graphs reordered by GoGraph using these methods, with Rabbit-Partition as the baseline.



(a) Normalized Runtime (b) Normalized Rounds

Fig. 13: The impact of different partition methods

The results indicate that different partitioning methods impact the runtime and iteration counts for graph algorithms on reordered graphs. Rabbit-Partition, Metis, and Louvain showed similar performance, while Fennel underperformed due to its stream-based approach, which makes decisions with partial graph knowledge. Thus, a superior partitioning method enhances GoGraph's performance.

VI. Related Work

Optimizations on synchronous iteration. There have been efforts to accelerate iterative computing by exploring asynchronous computing modes. Priter [52] and Maiter [14] preferentially select vertices in each iteration, aiming to expedite algorithm convergence by performing iterative computations on specific vertices instead of all vertices., thereby trying to avoid some inefficient computation. However, this method requires user-defined vertex selection strategies, which is not a trivial work. FBSGraph [53] employs a forward and backward sweeping execution framework for vertex processing, addressing the issue of slow propagation of vertex states when their outgoing neighbors precede them in the processing order. PathGraph [54] further optimizes the path of vertex state propagation, enhancing the efficiency of state propagation. GoGraph tries to maximize the advantages of asynchronous computations by rearranging graph processing order based on revealing the reason why asynchronous updating mode accelerates iterative computations.

Graph reordering. Vertex reordering has been a focal point in graph data preprocessing to enhance memory access efficiency through increased vertex locality. To improve the temporal and spatial locality, Gorder [41] uses a slide window to compute the score between the ordered vertices and the unordered vertices. The larger the score, the more frequently the unordered vertices will be accessed after the ordered vertices, from which an ordering algorithm can be deduced. Rabbit [44] maps the more frequently accessed vertices close to the L1 cache, thus reducing the overhead of swapping cache lines. Hub Clustering [49] assigns a contiguous range of subscripts to hub vertices whose degrees are larger than the average degree at the front of the graph data array. Since the neighbors of the high-degree vertices are likely to overlap, storing them together in memory decreases the frequency of cache line swapping. Hub Sorting (also known as frequency based clustering) [48] is a lightweight reordering method that extracts hub vertices, arranges them in descending order, and then swaps them with the vertices with continuous subscripts at the front of the data array. Thus, the subscripts of non-hub vertices can be preserved as much as possible in order to reduce the cost of reordering operations and improve the locality of the power law graph. They reorder vertices to enhance graph locality, boost cache hit rates, and speed up iterations. GoGraph optimizes vertex processing order, allowing more vertices to refresh their states using neighbors' latest states within the same iteration.

VII. Conclusion

In this paper, we propose GoGraph, a graph reordering algorithm that establishes a well-formed graph processing order, resulting in a reduction of the number of iteration rounds and acceleration of iterative computation. Specifically, we introduce a metric to evaluate the quality of the processing order, based on the count of positive edges. GoGraph employs a divide-and-conquer strategy to optimize this metric, with experimental results validating its effectiveness in reorganizing the vertex processing order.

Acknowledgment

This work is supported by The National Key R&D Program of China (2018YFB1003400), The National Natural Science Foundation of China (U2241212, 62072082, 62202088, 62072083, and 62372097), Joint Funds of Natural Science Foundation of Liaoning Province (2023-MSBA-078), Research Grants Council of Hong Kong, China, No.14205520, and Fundamental Research Funds for the Central Universities (N2216012 and N232405-17).

REFERENCES

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.

[2] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *PVLDB*, vol. 5, no. 8, 2012.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI '12*, 2012, pp. 17–30.

[4] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *OSDI '16*, 2016, pp. 301–316.

[5] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *SOSP '13*, 2013, pp. 456–471.

[6] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang, "Grape: Parallelizing sequential graph computations," *PVLDB*, vol. 10, no. 12, pp. 1889–1892, 2017.

[7] P. Yi, J. Li, B. Choi, S. S. Bhowmick, and J. Xu, "Flag: towards graph query autocompletion for large graphs," *DSE*, vol. 7, no. 2, pp. 175–191, 2022.

[8] H. Wu, C. Song, Y. Ge, and T. Ge, "Link prediction on complex networks: An experimental survey," *DSE*, vol. 7, no. 3, pp. 253–278, 2022.

[9] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *ICDE '15*. IEEE, 2015, pp. 363–374.

[10] C.-Y. Liu, W. Choi, S. Khadirsharbiyani, and M. Kandemir, "Mbfgraph: An ssd-based external graph system for evolving graphs," in *SC '23*, 2023, pp. 1–13.

[11] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *COMNET*, vol. 30, no. 1-7, pp. 107–117, 1998.

[12] M. Ghaffari and J. Li, "Improved distributed algorithms for exact shortest paths," in *SIGACT '18*, 2018, pp. 431–444.

[13] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: time to fuse for distributed graph-parallel computation," in *PPoPP '15*, 2015, pp. 194–204.

[14] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *TPDS*, vol. 25, no. 8, pp. 2091–2100, 2013.

[15] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu, "Adaptive asynchronous parallelization of graph algorithms," in *SIGMOD '18*, 2018, pp. 1141–1156.

[16] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI '15*, vol. 29, no. 1, 2015.

[17] D. P. Koester, S. Ranka, and G. C. Fox, "A parallel gauss-seidel algorithm for sparse power system matrices," in *SC '94*, 1994, pp. 184–193.

[18] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *WWW '17*, 2008, pp. 895–904.

[19] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[20] G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," in *KDD '02*, 2002, pp. 538–543.

[21] J. Pearl, "Reverend bayes on inference engines: A distributed hierarchical approach," in *AAAI '82*, 1982, pp. 133–136.

[22] Q. Wang, Y. Zhang, H. Wang, L. Geng, R. Lee, X. Zhang, and G. Yu, "Automating incremental and asynchronous evaluation for recursive aggregate data processing," in *SIGMOD '20*, 2020, pp. 2439–2454.

[23] F. Yao, Q. Tao, W. Yu, Y. Zhang, S. Gong, Q. Wang, G. Yu, and J. Zhou, "Ragraph: A region-aware framework for geo-distributed graph processing," *PVLDB*, vol. 17, no. 3, pp. 264–277, 2023.

[24] T.-S. Hsu, V. Ramachandran, and N. Dean, "Parallel implementation of algorithms for finding connected components in graphs." in *Parallel Algorithms*, 1994, pp. 23–41.

[25] Y. Yang, M. Jiang, and W. Li, "2d path planning by lion swarm optimization," in *ICRAS '20*. IEEE, 2020, pp. 117–121.

[26] Y. Wu, R. Jin, and X. Zhang, "Fast and unified local search for random walk based k-nearest-neighbor query in large graphs," in *SIGMOD '14*, 2014, pp. 1139–1150.

[27] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *WWW '08*, 2008, pp. 895–904.

[28] G. Feng, Z. Ma, D. Li, S. Chen, X. Zhu, W. Han, and W. Chen, "Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s," in *SIGMOD '21*, 2021, pp. 513–527.

[29] K. Vora, R. Gupta, and G. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *ASPLOS '17*, 2017, pp. 237–251.

[30] P. E. Black, "Topological sort," https://www.nist.gov/dads/HTML/topologicalSort.html.

[31] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 2010.

[32] V. Guruswami, R. Manokaran, and P. Raghavendra, "Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph," in *FOCS '08*, 2008, pp. 573–582.

[33] C. Lu, J. X. Yu, R.-H. Li, and H. Wei, "Exploring hierarchies in online social networks," *TKDE*, vol. 28, no. 8, pp. 2086–2100, 2016.

[34] R. Hassin and S. Rubinstein, "Approximations for the maximum acyclic subgraph problem," *IPL*, vol. 51, no. 3, pp. 133–140, 1994.

[35] B. Berger and P. W. Shor, "Tight bounds for the maximum acyclic subgraph problem," *JALG*, vol. 25, no. 1, pp. 1–18, 1997.

[36] M. Charikar, K. Makarychev, and Y. Makarychev, "On the advantage over random for maximum acyclic subgraph," in *FOCS '07*, 2007, pp. 625–633.

[37] A. Cvetkovic and V. Y. Protasov, "Maximal acyclic subgraphs and closest stable matrices," *SIMAX*, vol. 41, no. 3, pp. 1167–1182, 2020.

[38] J. Goldsmith, J. Lang, M. Truszczynski, and N. Wilson, "The computational complexity of dominance and consistency in cp-nets," *JAIR*, vol. 33, pp. 403–432, 2008.

[39] T. Cormen, "Introduction to algorithms th cormen, ce leiserson, rl rivest, and c. stein, eds," 2001.

[40] Z. Liu, K. Li, and X. He, "Cutting cycles of conditional preference networks with feedback set approach," *Computational Intelligence and Neuroscience*, vol. 2018, 2018.

[41] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *SIGMOD '16*, 2016, pp. 1813–1828.

[42] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *JSTAT*, vol. 2008, no. 10, p. P10008, 2008.

[43] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SISC*, vol. 20, no. 1, pp. 359–392, 1998.

[44] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *IPDPS '16*. IEEE, 2016, pp. 22–31.

[45] Z. Guan, J. Wu, Q. Zhang, A. Singh, and X. Yan, "Assessing and ranking structural correlations in graphs," in *SIGMOD '11*, 2011, pp. 937–948.

[46] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *IM*, vol. 6, no. 1, pp. 29–123, 2009.

[47] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *SIGKDD '05*, 2005, pp. 177–187.

[48] Z. Yunming, K. Vladimir, M. Charith, Z. Matei, and S. P. Amarasinghe, "Optimizing cache performance for graph analytics," *CoRR*, vol. abs/1608.01362, 2016.

[49] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *IISWC '18*. IEEE Computer Society, 2018, pp. 203–214.

[50] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[51] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *WSDM '14*, 2014, pp. 333–342.

[52] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritized iterative computations," in *SoCC '11*, 2011, pp. 1–14.

[53] Y. Zhang, X. Liao, H. Jin, L. Gu, and B. B. Zhou, "Fbsgraph: Accelerating asynchronous graph processing via forward and backward sweeping," *TKDE*, vol. 30, no. 5, pp. 895–907, 2018.

[54] P. Yuan, C. Xie, L. Liu, and H. Jin, "Pathgraph: A path centric graph processing system," *TPDS*, vol. 27, no. 10, pp. 2998–3012, 2016.