Speedup Graph Processing by Graph Ordering

Hao Wei, Jeffrey Xu Yu, Can Lu
Chinese University of Hong Kong
Hong Kong, China
{hwei,yu,lucan}@se.cuhk.edu.hk

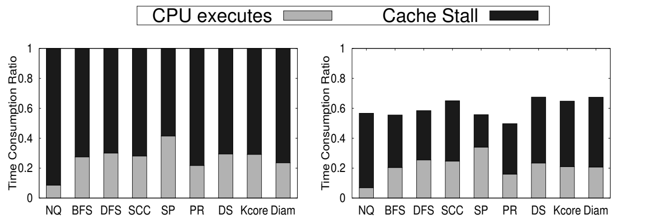
Xuemin Lin
The University of New South Wales
Sydney, Australia
lxue@cse.unsw.edu.au

ABSTRACT

The CPU cache performance is one of the key issues to efficiency in database systems. It is reported that cache miss latency takes a half of the execution time in database systems. To improve the CPU cache performance, there are studies to support searching including cache-oblivious, and cache-conscious trees. In this paper, we focus on CPU speedup for graph computing in general by reducing the CPU cache miss ratio for different graph algorithms. The approaches dealing with trees are not applicable to graphs which are complex in nature. In this paper, we explore a general approach to speed up CPU computing, in order to further enhance the efficiency of the graph algorithms without changing the graph algorithms (implementations) and the data structures used. That is, we aim at designing a general solution that is not for a specific graph algorithm, neither for a specific data structure. The approach studied in this work is graph ordering, which is to find the optimal permutation among all nodes in a given graph by keeping nodes that will be frequently accessed together locally, to minimize the CPU cache miss ratio. We prove the graph ordering problem is NP-hard, and give a basic algorithm with a bounded approximation. To improve the time complexity of the basic algorithm, we further propose a new algorithm to reduce the time complexity and improve the efficiency with new optimization techniques based on a new data structure. We conducted extensive experiments to evaluate our approach in comparison with other 9 possible graph orderings (such as the one obtained by METIS) using 8 large real graphs and 9 representative graph algorithms. We confirm that our approach can achieve high performance by reducing the CPU cache miss ratios.

1. INTRODUCTION

Graph processing has been extensively studied in the recent years given a large number of real applications developed over online social networks, location-based social networks, semantic web, biological networks, and road networks. In the literature, the reported studies have focused on new graph algorithms to process large graphs and has shown that they can achieve high efficiency. We list the graph algorithms that we investigate in this paper including Breadth-First Search (*BFS*) [20], Depth-First Search (*DFS*) [20], Strongly Connected Component (*SCC*) detection [48], Shortest Path (*SP*) by Bellman-Ford algorithm [20], PageRank (*PR*) [39], Dominating Set (*DS*) [19], graph decomposition (*Kcore*) [6] and graph diameter (*Diam*). Here, *BFS*, *DFS*, *SCC* and *SP* are primitive graph algorithms. On top of them, many graph algorithms are designed. *PR* is widely used in many applications. *DS* is a representative of the set cover problem in graph which is NP-hard, and there are greedy algorithms to solve it [17]. *Kcore* is a graph decomposition algorithm, which is to find every subgraph H of a graph G such that the degree of nodes in H is $\geq k$. *Diam* is one of the most important pieces of information about the graph structure. The graph algorithms listed have been extensively studied, and are efficient to process large graphs.



(a) The original order (b) Gorder

Figure 1: CPU execution and CPU Cache Stall over `sd1-arc`

In this paper, we focus on the following questions. Given the efficiency of the algorithms developed, can we make graph processing even faster? In addition, is there any general approach to enhance graph processing for all graph algorithms, rather than for some specific graph processing tasks only? To address these questions, we first show the potential to improve the efficiency significantly.

As observed in [2], a lot of time is wasted on CPU cache latency in database query processing. We also observe that graph algorithms waste a lot of time on CPU cache latency, and such time wasting cannot be handled by graph algorithms themselves. We show our finding in Fig. 1(a) by testing *NQ* (a primitive operation to access the neighbors of a node in a graph) as well as *BFS*, *DFS*, *SCC*, *SP*, *PR*, *DS*, *Kcore* and *Diam* using a hyperlink graph dataset sd1-arc with 94.94 millions of nodes and 1.9 billions of edges. The graph processing for the 9 tasks are different. To show the percentages of CPU cache stall (the CPU time waiting for the requested data due to the cache miss) and the CPU execution time while the CPU keeps computing, we normalize the entire graph processing time of each task to be 1. As shown in Fig. 1(a), CPU only works in the range of 10% and 45% (on average 30%) of the overall processing time, and wastes time in the range of 90% and 55% (on average 70% time) waiting for the desired data to be accessed in CPU cache in all algorithms tested.

The percentage of wasting CPU time is much higher than ex-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2882903.2915220

pected, which motivates us to study how to speedup CPU processing for graph computing. In this work, we explore how to reduce the time consumption on CPU cache stall by improving the CPU cache hit rate in graph computing, without changing the graph algorithms (implementations) and the data structures used. In other words, we do not aim at designing something that is for a specific graph algorithm, or for a specific data structure, or for a specific CPU cache mechanism. We seek a general solution for different graph algorithms. The main idea of our approach is to arrange nodes in graph in an order that nodes to be accessed is very likely to be in the CPU cache which is of considerably small size in comparison with the large graph size. We call our graph ordering Gorder. Fig. 1(b) shows the testing results by Gorder which rearranges all nodes in sd1-arc. Here, for comparison, the entire processing time by Gorder is normalized according to the entire processing time by the original order used. Take *NQ* as an example, The total processing time for *NQ* by Gorder is about 60% of the entire processing time in Fig. 1(a). The saving is the reduction of CPU cache stall, where the CPU execution time is almost the same, because it is done for the same algorithm over the same dataset. As can be seen in Fig. 1(b), the efficiency can be enhanced by Gorder, which significantly reduces the CPU cache stalls. More details about CPU cache are discussed in the appendix.

Related Works: The CPU cache performance has been studied in database systems. Ailamaki et al. show that cache miss latency takes half of the execution time in database systems [2], and Cieslewicz and Ross survey the CPU cache performance in [18]. There are many reported studies in literature that deal with trees [13, 15, 23, 35, 42, 43]. Rao et al. design cache sensitive search trees for searching in [42], and they also study cache sensitive B^+-trees for index structure in [43]. The cache performance is improved by fitting tree nodes into cache lines and removing most child pointers of tree nodes to reduce space. Lindstrom et al. generate cache-oblivious layout for binary search tree by optimizing a measure function with a locality measure to mimic the cache miss probability [35]. Ghoting et al. propose cache-conscious prefix tree to allocate the prefix tree in the memory [23]. Chen et al. propose Prefetching B^+-tree which uses the cache prefetch when loading the nodes of the B^+-tree, to reduce latency time [13], which can also be used to improve the hash join [12]. However, these works are designed to support searching for data that is with a total order, and they cannot be effectively used to support all different graph algorithms over complex graphs.

To deal with graphs, there are works on specific graph algorithms to improve CPU cache performance. Park et al. propose several strategies to optimize the cache performance of Floyd-Warshall, Bellman-Ford, Prim, and Bipartite matching algorithms in [40]. Then et al. optimize the multi-source breadth-first search (*BFS*) algorithm by sharing the common computation across concurrent *BFS*s to improve the cache hit rate [50]. However, these approaches are designed for some specific graph algorithms, and can not be used to support any graph algorithms in general. Some existing works improve the graph computing by graph ordering and node clustering. Banerjee et al. propose a node ordering scheme by the children-depth-first traversal method to improve the efficiency of the DAG traversal [4]. Auroux et al. propose to reorder the node set by BFS [3]. Mendelzon et al. propose a heuristic method for clustering nodes in the local area in the same disk page to minimize the number of I/Os needed to access the graph [38]. In addition, graph compression has been studied to reduce the number of bits per edge by placing nodes in a specific way. Kang et al. takes a graph decomposition approach to remove the hub nodes iteratively from the graph [29]. Boldi et al. study several ordering strategies for graph compression in [8], and use label propagation techniques to compute the clusters of the graph [7], where every node is assigned to the cluster having most neighbor nodes with it. Chierichetti et al. in [14] formulate graph compression as the variation of the graph bandwidth problem [16, 22], which finds the optimal permutation π of the node set minimizing $\max\{|\pi(u) - \pi(v)| : (u, v) \in E(G)\}$. The graph compression problem is closely related to the Minimum Linear Arrangement problem and its variants [27], which is NP-hard. Many heuristics have been proposed in literature [11, 32, 41, 44] but most existing approaches cannot process large graph successfully. The ideas presented here share the similarity with graph partitioning to minimize the number of edge-cuts, [30, 49, 51, 52, 53], which we discuss in detail later.

Major contribution: First, we explore a general approach to reduce the CPU cache miss ratio for graph algorithms with their implementation and data structures unchanged. This problem is challenging, because graphs are rather complex where there is no total order among data to follow and the graph algorithms designed are based on different techniques. It is worth noting that our general approach can be used together with other specialized optimizations for the specific algorithms. Second, we propose a solution by graph ordering, which is to find the optimal permutation ϕ among all nodes in a given graph G by keeping nodes that will be frequently accessed together in a window of size w, in order to minimize the CPU cache miss ratio. We use the ordering to renumber the node IDs as well as sort the nodes inside every adjacent lists. We will discuss in details that the same cannot be achieved by graph partitioning. Third, we prove the graph ordering problem to be studied is NP-hard, and give a basic algorithm with a bounded $\frac{1}{2w}$-approximation in time complexity of $O(w \cdot d_{max} \cdot n^2)$, where d_{max} is the maximum in-degree and n is the number of nodes in graph G. To further improve the performance, we propose a new priority queue (PQ) based algorithm, which is with the same $\frac{1}{2w}$-approximation but with $O(\sum_{u \in V}(d_O(u))^2)$ time complexity, where $d_O(u)$ is the out-degree of node u in G. We propose several optimization techniques including lazy-update to significantly reduce the cost of maintaining the priority queue. Fourth, we conducted extensive experimental studies to evaluate our approach in comparison with other 9 possible graph orderings (such as the one obtained by METIS) using 8 large real graphs and 9 representative graph algorithms. We confirm that our approach can achieve high performance by reducing the CPU cache miss ratio. In nearly all testings, our graph ordering outperforms other graph orderings for different graph algorithms over different datasets. The best speedup is > 2 comparing our graph ordering over others when testing graph algorithms over large real graphs.

The paper organization: The remainder of the paper is organized as follows. We discuss graph ordering in Section 2 in which we discuss the issues related to graph partitioning and graph ordering, give problem statement on graph ordering, and show the hardness of the problem. We give our algorithms in Section 3 in which we first give the approximation of our algorithm, and show we can significantly reduce the computational cost. We have conducted extensive testings, and report the experimental studies in Section 4. The paper is concluded in Section 5.

2. GRAPH ORDERING

In this paper, we model a graph as a directed graph $G = (V, E)$, where $V(G)$ represents the set of nodes and $E(G)$ represents the set of edges. The number of nodes and edges are denoted as $n = |V(G)|$ and $m = |E(G)|$, respectively. In addition, we use $N_O(u)$ and $N_I(u)$ to denote the out-neighbor set and in-neighbor set of u, such as $N_O(u) = \{v \mid (u, v) \in E(G)\}$ and $N_I(u) = \{v \mid (v, u) \in E(G)\}$, respectively. The in-degree, out-degree, and the degree of a node u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$. Two nodes, u and v, are sibling nodes if they share a common in-neighbor. It is worth mentioning that our approach to be discussed can deal with undirected graphs, since an undirected graph can be represented by a directed graph.

Our key idea for CPU speedup for graph computing is to find a way to represent a graph that can reduce the CPU cache miss ratio and is independent of any graph algorithms and the data structure they use. Here, reducing the CPU cache miss ratio is to reduce the number of times of copying data from main memory to cache, or from slower cache to faster cache. To achieve this goal is to keep nodes frequently accessed together stored closely in main memory so that they are more likely to be loaded into cache together by one single cache line transfer. A possible solution is to group nodes of the graph into blocks using graph partition algorithm which divides graph into partitions to minimize edge-cut which is the number of edges that cross different partitions. *METIS* [30] is a widely-used graph partition algorithm and is used to place nodes in blocks in the existing works [51, 52]. However, such graph partition algorithms can not serve the purpose of reducing CPU cache miss ratio effectively for the following reasons. First, a recent study [34] shows the real graphs do not have good edge-cut due to the power-law degree distribution and the existence of nodes with high degree. The neighbors of a node with high degree must be dispersed in different partitions because one single partition is not enough to accommodate all neighbors of such high degree nodes. Second, in dealing with large graphs in distributed or I/O contexts where graph partitioning works well [51, 52], the size of a partition is rather large, e.g., 64 kilobytes (64KB). In such cases, the algorithms can be designed to deal with the nodes/edges in a partition in a way to achieve efficiency. However, for CPU cache, a cache line is fixed to be 64 bytes (64B), which is very small, and it is very difficult for user programs to control the CPU cache. Third, there is an issue of deciding the number of partitions or the number of nodes to be kept in one graph partition, given the small CPU cache line of 64B. A simple way is to keep 16 nodes in one partition, assuming a node is represented by an integer of 4 bytes. However, it does not work for all graph algorithms. Consider *PR*, a node needs to be associated with its degree and a PageRank value, which may end up additional 4+8 bytes. In other words, a partition can only keep 5 nodes in a partition to feed the 64B size CPU cache line. Fourth, assuming we can keep all nodes in all 64B partitions one followed by another, it does not necessarily mean it can reduce the CPU cache miss ratio much, since CPU has its own mechanism to align the data into the 64B CPU cache lines in main memory, which may be different from the way of the partitions allocated. In other words, a partition may be allocated into more than one CPU cache line, though the partition size is equal to cache line size. Note that CPU alignment is difficult to be controlled by user programs.

In this work, we propose graph ordering to arrange all nodes in an order, aiming at reducing CPU cache miss ratio, for any graph algorithms with any data structure they use in general. The main idea behind graph ordering is to reflect the CPU access of graph in a sequential manner, since CPU accesses data in sequence no matter how complex the graph is, and at the same time to maintain locality of nodes that will be frequently accessed together closely in graph ordering, which is the focus of this work we will discuss in detail. It is important to note that the graph ordering only affects the way of arranging nodes in main memory, and is independent from graph algorithm, algorithm implementation, and the data structure used.

Example 2.1: A graph $G = (V, E)$ is shown in Fig. 2 as a running

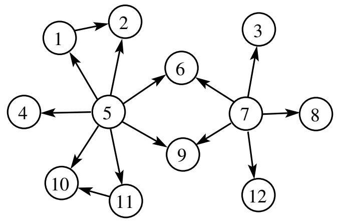


Figure 2: An Example

example. There are 12 nodes numbered from 1 to 12, which is considered as one possible graph ordering. Given such numbering, the graph can be represented by either adjacency-matrix or adjacency-list in Fig. 10, where $n = 12$ and the node numbered i will be placed at the i-th position in the corresponding array. □

As illustrated in the example, in graph computing, a node will be assigned with an ID. Two vertices will be stored closely if they have similar IDs in main memory. In the following, the node v_i is assumed to have an ID i, for simplicity.

Next, we discuss locality of a graph that needs to be maintained in graph ordering. To capture the locality, we look at the common statement that is used in graph algorithms below.

```
1: for each node $v \in N_O(u)$ do
2:     the program segment to compute/access $v$
```

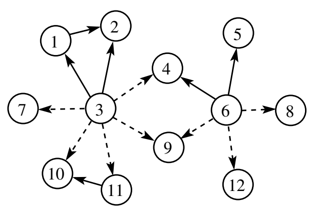
This pattern appears in *BFS*, *DFS*, *PR*, and *SP*, to name a few, when it needs to access nodes v in an out-neighbor set $N_O(u)$, and indicates that these nodes are the nodes that need to be maintained locally to reduce the CPU cache miss ratio. When looking at the statement carefully, there are two relationships, namely, neighbor relationship and sibling relationship. Supposed $v_1, \cdots, v_{d_O(u)}$ are in the out-neighbor set of u, u and v_i are in neighbor relationship for $1 \leq i \leq d_O(u)$, and any two v_i and v_j in $N_O(u)$ are in sibling relationship. Both neighbor relationship and sibling relationship play an important role in locality, since they will be accessed together. One fact is that the sibling relationship is a dominating factor since there are $d_O(u)$ neighbor relationships and $\binom{d_O(u)}{2}$ combinations of sibling relationships, and $\binom{d_O(u)}{2} \gg d_O(u)$ in general. In order to measure the closeness of two nodes, u and v, in terms of locality, we define a score function.

$$S(u, v) = S_s(u, v) + S_n(u, v) \tag{1}$$

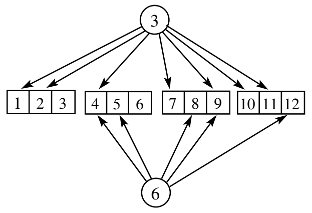
Here, $S_s(u, v)$ is the number of the times that u and v co-exist in sibling relationships, which is the number of their common in-neighbors, $|N_I(u) \cap N_I(v)|$. And $S_n(u, v)$ is the number of times that u and v in the neighbor relationship, which is either 0, 1, or 2, since both edge (u, v) and (v, u) may co-exist in a directed graph. One thing about the score function is that two nodes should be considered to be placed together, if they are in many sibling relationship, even though there is no edge between them.

Based on the score function, our problem is to maximize the locality of two nodes to be placed closely. And this is to find a permutation function $\phi(\cdot)$, to maximize the sum of the score, $S(\cdot)$, for close node pairs in G, where $\phi(u)$ assigns every node u with an unique number in $[1, n]$, assume there are n nodes in G.

Problem Statement: Find the optimal graph ordering (permutation), $\phi(\cdot)$, that maximizes Gscore (the sum of locality score), $F(\cdot)$,

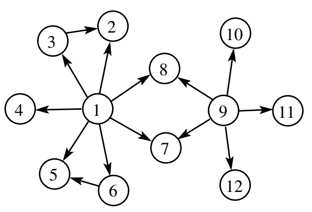


(a) Minimize Edge-Cuts

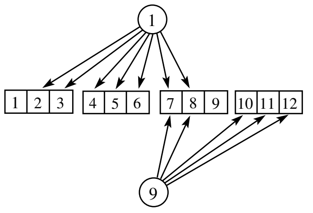


(b) Graph Partition

Figure 3: By Graph Partitioning



(a) Maximize $F(\phi)$



(b) Partition Representation

Figure 4: By Graph Ordering

based on a sliding window model with a window size w.

$$F(\phi) = \sum_{0<\phi(v)-\phi(u)\le w} S(u,v) \tag{2}$$

$$= \sum_{i=1}^{n} \sum_{j=\max\{1,i-w\}}^{i-1} S(v_i, v_j) \tag{3}$$

We discuss more on the problem formulation. We adopt a sliding window model with a window size w ($w > 0$). Assume two nodes, u and v, are assigned with IDs in the graph ordering, denoted as $\phi(u)$ and $\phi(v)$, and u appears before v ($\phi(v) > \phi(u)$). With the sliding window, for any node v in G, as a pivot, we maximize the score function $S(u, v)$ for the pair of node v and any other node u in G that appears before v in the sliding window of size w (Eq.(2)). In fact, implicitly by the sliding window, at the same time, we also maximize $S(u, v)$ for the pair of node v and any other node u in G that appears after v, where the role of u and v are reversed when u becomes a pivot. Suppose $\phi(v_i) = i$. Eq. (2) can be rewritten as Eq. (3). Let ϕ_w denotes a permutation ϕ maximizing $F(\phi)$ with a window size w. It is worth noting that $F(\phi_w)$ will be different for a different window size w and $F(\phi_w) > F(\phi_{w'})$ if $w > w'$. As confirmed in our testings, we use $w = 5$, which ensures high reduction of CPU cache miss ratio because two nodes within distance 5 is likely to be located in the same cache line.

The window used in graph ordering is different from the partition used in graph partitioning. The window we use is based on a sliding window, whereas the partition is a non-overlapping window. Consider graph partitioning. With the non-overlapping partitions, nodes within a partition can be placed randomly since they are considered locally and there is no difference where they are placed in the same partition. Furthermore, the last node in a partition can possibly be far away from the first node in its next partition, in terms of distance in graph. On the other hand, consider graph ordering, with the sliding window, the locality is considered for any node in a window before and after its appearance in a continuous way. Such continuity can significantly reduce the CPU catch miss ratio. Consider any two v_i and v_j, for $0 < j - i \le 2w$, where $\phi(v_i) = i$ and $\phi(v_j) = j$. The overlap range of windows between them is $2w + 1 - (j - i)$. Such high overlapping increases the chances to reduce the CPU cache miss ratio.

Below, we show the difference between graph partitioning and graph ordering using the graph G in Fig. 2. As shown in Fig. 2, there are 12 nodes and 14 edges in G, and G is a power-law graph, which has two nodes, v_5 and v_7, with high out-degree, 7 and 5, respectively. Note that there is no edge between v_6 and v_9. But v_6 and v_9 are siblings and share two in-neighbors, v_5 and v_7. Here, every node v_i in G is with an ID of i, which is a graph permutation number. However, such a permutation given in Fig. 2 does not preserve much locality. For example, v_2 is placed before v_3, which is supposed to be accessed/computed together, but v_2 is far away from v_3 in graph. Assumed one CPU cache line holds 3 nodes. Let partition size and window size be 3.

Consider graph partitioning. We need to partition the graph G in Fig. 2 into 4 partitions by minimizing the number of edges across partitions. Based on the graph partitioning, we can obtain a graph permutation by assigning two nodes in the same partition with close permutation numbers. The optimal graph partitioning for G in Fig. 2 is illustrated in Fig. 3(a), where the two graphs in Fig. 2 and Fig. 3(a) have the identical topological structure but are with different permutations. As shown in Fig. 3(a), the edges shown as dash-lines are the edges that cross two partitions to be cut. There are 4 partitions, $\{v_1, v_2, v_3\}$, $\{v_4, v_5, v_6\}$, $\{v_7, v_8, v_9\}$, $\{v_{10}, v_{11}, v_{12}\}$ in Fig. 3(a). Fig. 3(b) shows the 4 partitions of Fig. 3(a). From Fig. 3(b), we can see that if a graph algorithm accesses the out-neighbors of node v_3, $N_O(v_3) = \{v_1, v_2, v_4, v_7, v_9, v_{10}, v_{11}\}$, it needs to access all the 4 partitions in 4 CPU cache lines. If it needs to access the out-neighbors of node v_6, $N_O(v_6) = \{v_4, v_5, v_8, v_9, v_{12}\}$, it needs to access 3 partitions in 3 CPU cache lines.

Consider graph ordering. Fig. 4(a) shows the optimal permutation by graph ordering that maximizes $F(\phi)$, with the window size $w = 3$, for G shown in Fig. 2, where the two graphs in Fig. 2 and Fig. 4(a) have the identical topological structure but are with different permutations. Fig. 4(b) shows the 4 partitions of Fig. 4(a), as the result of graph ordering. From Fig. 4(b), we can see that if the graph algorithm accesses the out-neighbors of node v_1, $N_O(v_1) = \{v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$, it needs to access 3 CPU cache lines. If the graph algorithm needs to access the out-neighbors of node v_9, $N_O(v_9) = \{v_7, v_8, v_{10}, v_{11}, v_{12}\}$, it needs to access 2 CPU cache lines. The graph ordering outperforms graph partitioning as shown in this example.

An acute reader may find that the graph ordering problem we study in this work can be solved by graph partitioning. First, we show that it is possible provided the partition size is given. In graph ordering, the score function $S(\cdot)$ (Eq. (1)) takes both neighbor relations and sibling relations, whereas graph partitioning is to minimize the edge cut, which implies that the neighbor relationship, $S_n(u, v)$, is the only concern. For graph partitioning to deal with the sibling relationships, $S_s(u, v)$, it can convert G into a graph G' by adding all edges between every pair of v_i and v_j if they are siblings. In this way, graph partitioning by minimizing the number of edges across different partitions in G' can be considered as a way to determine a graph permutation. Second, we show that it is infeasible in practice, since the number of edges in G' can be huge. Supposed a node u has 100,000 out-neighbors in a graph. The number of additional edges in G' to represent the sibling relationships of u will be up to 10 billion, which is unacceptable for both the computation cost and the space cost.

The graph ordering problem of maximizing $F(\phi)$ is NP-hard.

Theorem 2.1: *Maximizing $F(\phi)$ to obtain an optimal permutation $\phi(\cdot)$ for a directed graph G is NP-hard.*

The proof sketch is given in Appendix.

3. THE ALGORITHMS

In this section, we focus on efficient algorithms and data structures for solving the graph ordering problem for large graphs with millions/billions of nodes and edges. We give approximate algorithms due to the NP-hard time complexity of the problem.

The optimal graph ordering, for the window size $w = 1$, by maximizing $F(\phi)$ is equivalent to the maximum traveling salesman problem, denoted as *maxTSP* for short. Serdyukov proposes a $\frac{3}{4}$-approximation algorithm for *maxTSP* in [46]. It needs $O(n^3)$ time complexity to compute the maximum cycle cover and maximum matching which is too costly for large graphs. Alternatively, Fisher et al. propose the best-neighbor heuristic, to achieve $\frac{1}{2}$-approximation for the *maxTSP* in [21]. It starts with a randomly chosen node, and inserts it into a queue. Then, it greedily selects a node that has the maximum edge weight with the node just inserted into the queue repeatedly.

In general, the problem of finding the optimal graph ordering by maximizing $F(\phi)$ is the problem of solving *maxTSP-w* with a window size w as a variant of the *maxTSP* problem. To solve the graph ordering problem as *maxTSP-w*, we can construct an edge-weighted complete undirected graph G_w from G. Here, $V(G_w) = V(G)$, and there is an edge $(v_i, v_j) \in E(G_w)$ for any pair of nodes in $V(G_w)$. The edge-weight for an edge (v_i, v_j) in G_w, denoted as s_{ij}, is $S(v_i, v_j)$ computed for the two nodes in the original graph G. Then the optimal *maxTSP-w* over G is the solution of *maxTSP* over G. Note that *maxTSP* only cares the weight between two adjacent nodes, whereas *maxTSP-w* cares the sum of weights within a window of size w.

In this work, instead of constructing the large complete graph G_w, we adapt the main idea of the best-neighbor heuristic that used to solve *maxTSP* to deal with graph ordering as *maxTSP-w* for a window size $w > 1$. We adapt the best-neighbor heuristic, because it can be used to deal with large graphs with millions/billions of nodes and edges, even though the best-neighbor heuristic is $\frac{1}{2}$-approximation, which is worse than $\frac{3}{4}$-approximation by Serdyukov's algorithm [46]. In fact, it is difficult to extend the Serdyukov's algorithm to deal with the case that $w > 1$ in general.

3.1 The *GO* Algorithm

We give an approximate algorithm to compute a graph ordering ϕ by maximizing $F(\phi)$. Maximizing $F(\phi)$ is done by greedily inserting the node v that has the largest score $\sum S(u, v)$ with nodes u that have recently inserted in the window with size w repeatedly, based on the idea of the best-neighbor heuristic. The algorithm is called *GO* (Graph Ordering) and is given in Algorithm 1. The algorithm takes three inputs, the graph G, the window size w, and the score function $S(\cdot, \cdot)$ (Eq.(1)). Given an array P of size n, the node v placed at $P[i]$ is assigned to a permutation number i. The algorithm first randomly selects a node v, and inserts it into $P[1]$ as the first node in the permutation (line 1). Let V_R represents the set of nodes for the remaining nodes that have not yet been inserted into the array of P. In the while statement line 3-10 when finding the node v_i that should be placed at $P[i]$, it computes the sum of score k_v for every $v \in V_R$

$$k_v = \sum_{j=\max\{1,i-w\}}^{i-1} S(v_j, v) \tag{4}$$

where v_j is a node in the window of size w that has been inserted before v_i, represented as $P[j]$ in the *GO* algorithm. The node v_{max} is the node with the largest k_v selected from V_R. The algorithm inserts the node v_{max} into $P[i]$ repeatedly, and terminates until all nodes in G are placed in P. The permutation ϕ is obtained such that $\phi(v) = i$, if v is placed at $P[i]$.

Algorithm 1 *GO* $(G, w, S(\cdot, \cdot))$

```
 1: select a node $v$ as the start node, $P[1] \leftarrow v$;
 2: $V_R \leftarrow V(G) \setminus \{v\}$, $i \leftarrow 2$;
 3: while $i \le n$ do
 4:     $v_{max} \leftarrow \emptyset$, $k_{max} \leftarrow -\infty$;
 5:     for each node $v \in V_R$ do
 6:         $k_v \leftarrow \sum_{j=\max\{1,i-w\}}^{i-1} S(P[j], v)$;
 7:         if $k_v > k_{max}$ then
 8:             $v_{max} \leftarrow v$, $k_{max} \leftarrow k_v$;
 9:     $P[i] \leftarrow v_{max}$, $i \leftarrow i + 1$;
10:     $V_R \leftarrow V_R \setminus \{v_{max}\}$;
```

We discuss the approximation of the algorithm *GO*, and give the approximation of *GO* in Theorem 3.1.

Theorem 3.1: *The algorithm GO gives $\frac{1}{2w}$-approximation for maximizing $F(\phi)$ to determine the optimal graph ordering.*

Proof Sketch: The optimal graph ordering with a window size w is the same with the optimal *maxTSP-w* problem, except that the former takes the permutation and the latter tries to find a circuit. Let F_w denotes the score of the optimal solution on G for *maxTSP-w* problem. And let F_{go} denotes the Gscore $F(\cdot)$ of the graph ordering by the *GO* algorithm. We give an upper bound $\overline{F}_w$ of F_w by formulating the below optimization problem.

$$\begin{aligned} \overline{F}_w = \quad & \text{maximize} && \sum_{i=1}^{n-1} \sum_{j>i} s_{ij} x_{ij} \\ & \text{subject to} && \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2w, i \in [1, n] \\ &&& 0 \le x_{ij} \le 1, \qquad i, j \in [1, n] \end{aligned} \tag{5}$$

Here, s_{ij} represents $S(v_i, v_j)$ between nodes v_i and v_j in G. On one hand, the optimal solution of *maxTSP-w* is a feasible solution of Eq. (5), which satisfies the two constraints by setting $x_{ij} = 1$ if v_i and v_j are located within the window of size w, and $x_{ij} = 0$ otherwise. On the other hand, Eq. (5) is a generalized form of *maxTSP-w* problem. This is because it only requires $\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2w$, for any $i \in [1, n]$. In other words, consider v_i and let W_i denotes the set of w nodes before v_i, plus the set of w nodes after v_i, and plus v_i itself, such that $|W_i| = 2w + 1$. Eq. (5) finds the max $\overline{F}_w$ where it is possible that $W_i \cap W_{i+1} = \emptyset$ for two adjacent nodes v_i and v_{i+1}. However, the *maxTSP-w* problem requires that $|W_i \cap W_{i+1}| = |W_i| - 1$. Besides, x_{ij} is not required to be either 0 or 1. Hence, we have $F_w \le \overline{F}_w$. We further give an upper bound of $\overline{F}_w$ by the Lagrangian duality.

$$\begin{aligned} \overline{F}_w &\le \max_{0 \le x_{ij} \le 1} \sum_{i=1}^{n-1} \sum_{j>i} s_{ij} x_{ij} + \sum_{i=1}^{n} \alpha_i (2w - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji}) \\ &= \max_{0 \le x_{ij} \le 1} \sum_{i=1}^{n-1} \sum_{j>i} (s_{ij} - \alpha_i - \alpha_j) x_{ij} + 2w \sum_{i=1}^{n} \alpha_i \end{aligned} \tag{6}$$

Here, α_i is the Lagrange multiplier and Eq. (6) is true for all $\alpha_i \in R$. Assume the graph ordering by *GO* is $v_1, v_2, \cdots, v_n$, where $\phi(v_i) = i$. In Eq. (6), α_i is given as the sum of the weights between v_{i+1} and the last up to w just before v_{i+1} by *GO* algorithm, such that $\alpha_i = \sum_{j=\max\{1,i-w+1\}}^{i} s_{j,i+1}$ for $i \in [1, n-1]$ and $\alpha_n = 0$.

We have $\alpha_i \geq 0$ and $\sum_{i=1}^{n} \alpha_i = F_{go}$. Since v_i is placed before v_j in the graph ordering by the algorithm *GO* for $1 \leq i < j \leq n$, we have $s_{ij} - \alpha_i \leq 0$ according to the best-neighbor heuristic, and hence $s_{ij} - \alpha_i - \alpha_j \leq 0$. Therefore, we have $F_w \leq \overline{F}_w \leq 2w \sum_{i=1}^{n} \alpha_i = 2w \cdot F_{go}$, which leads to the $\frac{1}{2w}$-approximation. □

In practice, we find that the lower bound of the approximation ratio of *GO* does not reflect the real performance of *GO*, and the Gscore F_{go} of the graph ordering by *GO* is very close to the optimal result. For the graph in Fig. 2, F_{go}, the optimal score F_w, and the upper bound of the optimal score $\overline{F}_w$ by Eq. (5) are 27, 32, and 35, respectively, with $w = 3$. We also compare F_{go} with the upper bound of the optimal score $\overline{F}_w$ using real datasets, since the optimal score F_w cannot be computed due to the NP-hard complexity. We analyze two small datasets: Facebook with 4,039 nodes and 88,234 edges from SNAP[1] and AirTrafficControl with 1,226 nodes and 2,615 edges from KONECT[2], given Eq. (5) needs $O(n^2)$ variables and $O(n)$ constraints. Table 1 shows F_{go} and $\overline{F}_w$ with different w. We can see that $F_{go}/\overline{F}_w$ is larger than 0.6, which indicates that the real performance of F_{go} is very close to the optimal score F_w in large graphs, given $F_{go} \leq F_w \leq \overline{F}_w$.

	$w = 3$		$w = 5$		$w = 7$	
	F_{go}	$\overline{F}_w$	F_{go}	$\overline{F}_w$	F_{go}	$\overline{F}_w$
Facebook	149,073	172,526	231,710	275,974	308,091	373,685
AirTraffic	2,420	3,468	2,993	4,697	3,465	5,545

Table 1: F_{go} **and** $\overline{F}_w$

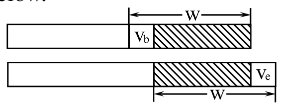
Theorem 3.2: *The GO Algorithm 1 is in $O(w \cdot d_{max} \cdot n^2)$, where d_{max} denotes the maximum in-degree of the graph G.*

The proof sketch is given in Appendix.

3.2 The Priority Queue based Algorithm

The computational cost of the *GO* algorithm is high as given in Theorem 3.2 in $O(w \cdot d_{max} \cdot n^2)$ where d_{max} is the maximum in-degree of G. The *GO* algorithm is impractical to be used to compute graph ordering for large graphs. The high computational cost is contributed by line 5-6 in Algorithm 1. We discuss it from two aspects. First, it repeatedly computes Eq. (4) w times for the same pair (v_j, v) while v_j is in the window of size w. Second, it scans every node v in the set of remaining nodes V_R in every iteration when computing Eq. (4). In other words, it scans every node $v \in V_R$ that has neighbor/sibling relationship with the nodes v_j in the window of size w, as well as the nodes v that has no neighbor/sibling relationship with the nodes v_j in the window of size w. Obviously, the latter is unnecessary. We give Example-A in the Appendix to discuss the two drawbacks in details.

Based on the observations, we propose a priority queue (PQ) based algorithm called *GO-PQ* (Algorithm 2). Like *GO*, we use an array of P to keep the graph ordering (permutation). The permutation ϕ is obtained such that $\phi(v) = i$, if v is placed at $P[i]$. Unlike *GO*, we efficiently compute line 5-6 in Algorithm 1, using a priority queue denoted as $\mathcal{Q}$. In the priority queue, $\mathcal{Q}[v]$ keeps a key which is k_v (Eq.(4)) for node v during computing, and the node v_{max} with the largest k_{max} will be popped up first from $\mathcal{Q}$ to be appended at the end of P. Consider two nodes, u and v in $\mathcal{Q}$, u will appear before v (or in other words pop before v), if $k_u > k_v$ regarding the current window of size w. We denote the key of v by $\mathsf{key}(v)$. In computing the graph ordering, to address the two drawbacks of *GO*, we incrementally update $\mathsf{key}(v)$ regarding the sliding windows shown below.



While the window is sliding, suppose v_b is the node to leave the window, and v_e is the node to join the window. We incrementally update $\mathsf{key}(v)$ for v in three ways.

Increase key: When v_e is newly placed in the permutation P, v in $\mathcal{Q}$ will increase its key value by 1 if v and v_e are considered local, based on the neighbor/sibling relationship. For the neighbor relationship, it means either there is an edge from v to v_e or there is an edge from v_e to v. For the sibling relationships, it means v and v_e share a common in-neighbor node. We denote the key increment by 1 for v in $\mathcal{Q}$ as $\mathcal{Q}.\mathsf{incKey}(v)$.

Decrease key: When v_b is about to leave the window while the window is sliding to the next position, v in $\mathcal{Q}$ will decrease its key value by 1 if v and v_b are considered local, based on the neighbor/sibling relationship as discussed in the case for increasing key. We denote the key decrement by 1 for v in $\mathcal{Q}$ as $\mathcal{Q}.\mathsf{decKey}(v)$.
The updating ensures that the node to be selected next from $\mathcal{Q}$ is the one v_{max} with the largest k_{max} among the nodes that have not been selected.

Find the max key: Finding the node v_{max} with largest k_v (Eq. (4)) from the set of remaining nodes is to pop up the top node from $\mathcal{Q}$. We denote it by $\mathcal{Q}.\mathsf{pop}()$.

The *GO-PQ* algorithm is given in Algorithm 2, which uses these three operations as shown in line 6-20 to replace line 5-6 of Algorithm 1 by *GO*. The *GO-PQ* algorithm gives the same result of *GO* with the same $\frac{1}{2w}$-approximation for a window of size w.

The *GO-PQ* algorithm takes the same three inputs, the graph G, the window size w, and the score function $S(\cdot, \cdot)$ (Eq.(1)). Given an array P of size n, the node v placed at $P[i]$ is assigned to a permutation number i. In *GO-PQ*, initially, it inserts all nodes of G into the priority queue $\mathcal{Q}$ where the key of each node is initialized to be zero. Then, it randomly selects a node v, and inserts it into $P[1]$ as the first node in the permutation before deleting the node v from $\mathcal{Q}$ (line 3). In the while loop (line 5-22), it updates the keys of the nodes only if they are related to two nodes, v_b and v_e, where v_b is the node to leave the sliding window, and v_e is the node to join the sliding window. In brief, first, in line 7-12, regarding v_e, if a node u is a neighbor of v_e, it increases its key by $\mathsf{incKey}(u)$; and if a node v is a sibling of v_e, it increases its key by $\mathsf{incKey}(v)$. Second, in line 13-20, regarding v_b, if a node u is a neighbor of v_b, it decreases its key by $\mathsf{decKey}(u)$; and if a node v is a sibling of v_b, it decreases its key by $\mathsf{decKey}(v)$. Finally, following the updates, in the while loop, it places the node v_{max} into the array P where v_{max} is selected by popping the top node from $\mathcal{Q}$ line 21. A example is given in Example-B in the Appendix.

Compared with *GO*, *GO-PQ* significantly reduces the computational cost. We give the time complexity of *GO-PQ* algorithm in Theorem 3.3.

Theorem 3.3: *The time complexity of the GO-PQ algorithm is $O(\mu \cdot \sum_{u \in V}(d_O(u))^2 + n \cdot \varrho)$, where μ denotes the time complexity for the updates* $(\mathsf{incKey}(\cdot)$ *and* $\mathsf{decKey}(\cdot))$ *and ϱ denotes the time complexity for finding the max node* $(\mathsf{pop}())$.

The proof sketch is given in Appendix.

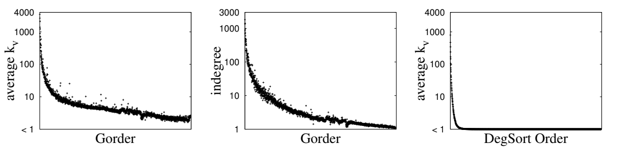
We will discuss the priority queue $\mathcal{Q}$ itself and its operations in Section 3.3. Below, we discuss the effectiveness of the window size w, selecting the first node, reducing the computational cost for updates, and dealing with huge nodes that have large outdegree.

[1] snap.stanford.edu/

[2] konect.uni-koblenz.de/networks/

Algorithm 2 *GO-PQ* $(G, w, S(\cdot,\cdot))$

```
1: for each node $v \in V(G)$ do
2:     insert $v$ into $\mathcal{Q}$ such that $\mathsf{key}(v) \leftarrow 0$;
3: select a node $v$ as the start node, $P[1] \leftarrow v$, delete $v$ from $\mathcal{Q}$;
4: $i \leftarrow 2$;
5: while $i \leq n$ do
6:     $v_e \leftarrow P[i-1]$;
7:     for each node $u \in N_O(v_e)$ do
8:         if $u \in \mathcal{Q}$ then $\mathcal{Q}.\mathsf{incKey}(u)$;
9:     for each node $u \in N_I(v_e)$ do
10:        if $u \in \mathcal{Q}$ then $\mathcal{Q}.\mathsf{incKey}(u)$;
11:        for each node $v \in N_O(u)$ do
12:            if $v \in \mathcal{Q}$ then $\mathcal{Q}.\mathsf{incKey}(v)$;
13:     if $i > w + 1$ then
14:        $v_b \leftarrow P[i-w-1]$;
15:        for each node $u \in N_O(v_b)$ do
16:            if $u \in \mathcal{Q}$ then $\mathcal{Q}.\mathsf{decKey}(u)$;
17:        for each node $u \in N_I(v_b)$ do
18:            if $u \in \mathcal{Q}$ then $\mathcal{Q}.\mathsf{decKey}(u)$;
19:            for each node $v \in N_O(u)$ do
20:                if $v \in \mathcal{Q}$ then $\mathcal{Q}.\mathsf{decKey}(v)$;
21:     $v_{max} \leftarrow \mathcal{Q}.\mathsf{pop}()$;
22:     $P[i] \leftarrow v_{max}, i \leftarrow i + 1$;
```



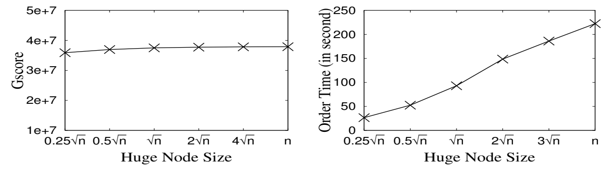
(a) $S()$ by *GO-PQ* (b) $d_I(u)$ Distri. (c) $S()$ by degsort

Figure 5: The Permutation and its In-degree Distri. on Flickr

Effectiveness of the window size w: It is important to note that the approximation of *GO* and *GO-PQ* is the same $\frac{1}{2w}$-approximation as proved. In *GO-PQ*, the window size w determines how the node v_b is defined which is related to k_v (Eq. (4)). However, in *GO-PQ*, the time complexity is not related to w. In other words, the window size w has no impacts on the time complexity.

Selecting the first node: The selection of the first node has impacts on the overall graph ordering. We select the node v_1 with the largest in-degree as the first node to start the ordering for the following two reasons. First, the node v_1 with the largest in-degree $d_I(v_1)$ (or largest $N_I(v_1)$) indicates that v_1 is highly likely to be a sibling node of another node v_i that shares common in-neighbor nodes in $N_I(v_1)$. Second, because $N_I(v_1)$ is large, v_1 will be frequently accessed together with its sibling nodes in the graph computing because there are a large number of in-coming edges pointing to v_1. The first node selected will lead the nodes selected in the following, and the nodes selected in the following iterations are more likely to be large in-degree nodes. Given the Flickr dataset with over 2.3M nodes, we compute the permutation by *GO-PQ*, denoted as Gorder, and show the in-degree distribution and the average scores of nodes comparing with the average scores of nodes by sorting nodes with decreasing in-degree in Fig. 5. In Fig. 5, a value in x-axis represents 1,000 nodes, and a value in y-axis represents the average k_v for the 1,000 nodes. Fig. 5(a) and Fig. 5(b) show the average score (Eq. (4)) and the average in-degree (d_I) of every 1,000 nodes. The nodes appear on the top of the permutation are with high score and high in-degree, and they are supposed to be locally stored in main memory following the permutation. Such high $F(\cdot)$ by summing all the score (Eq. (4)) cannot be achieved by simply sorting all nodes by in-degree, as shown in Fig. 5(c).

Reducing the computational cost: In *GO-PQ*, we have shown the main idea of the PQ-based algorithm. We can further reduce computational cost to make it efficient. Consider a node u that is a common in-neighbors of the two nodes, v_e and v_b, in the window before v_b leaves. In Algorithm 2, it first does $\mathsf{incKey}(v)$ w.r.t v_e followed by $\mathsf{decKey}(v)$ w.r.t v_b for every node in $N_O(u)$. Obviously, such pair of $\mathsf{incKey}(v)$ and $\mathsf{decKey}(v)$, will not change the key of such node v in $N_I(u)$, and such computing is unnecessary. To avoid such unnecessary computing, we can simply add the condition of "**if** $v_b \notin N_O(u)$ **then**" for line 11-12 to ensure that line 11-12 will only be executed when the condition is true. In a similar way, we add the condition of "**if** $v_e \notin N_O(u)$ **then**" for line 19-20 to ensure that line 19-20 will only be executed when the condition is true. In practice, since the window size w is rather small and v_e and v_b share many in-neighbors due to their close positions in the permutation, this can reduce cost significantly.



(a) Gscore (b) Graph Ordering Time

Figure 6: Huge Nodes on Flickr

Dealing with huge nodes: Real large graphs follow power-law degree distribution, with a small number of nodes having a large out-degree and a large number of nodes having a small out-degree. Supposed a node u has a large set of out-neighbors, $N_O(u)$. When one of its out-neighbors, $v \in N_O(u)$, is to join in/leave from the window, the key of every out-neighbor in $N_O(u)$ needs to update (line 11-12 and line 19-20 in Algorithm 2). This incurs high overhead for nodes that are with high out-degree $d_O(u)$, which is also indicated by the time complexity $O(\sum_u (d_O(u))^2)$. In this work, we take a high out-degree node as a **huge node** if its out-degree is greater than $\sqrt{n}$. For a large graph with millions of nodes, $\sqrt{n}$ is at least 1,000. To save such high computational cost, we do not compute the huge nodes when updating the key for the out-neighbors of these huge nodes. This is to add the condition of "**if** $d_O(u) \leq \sqrt{n}$ **then**" after line 9 and line 17 to ensure that line 10-12 and line 18-20 will only be executed when the condition is true. Such reduction of computing does not affect the quality of the chosen v_{max} to be placed into the array of P next, because the candidate nodes to be selected as v_{max} are likely to be the out-neighbors of huge nodes. Such reduction of computing those out-neighbors of huge nodes will reduce the similar amount of updates to all candidates that can possibly be the next v_{max}. Given two graph orderings, one is ϕ_1 by *GO-PQ* without special consideration to huge nodes, and the other is ϕ_2 by *GO-PQ* with special consideration to huge nodes discussed above. We measure the difference between the two using the Kendall tau coefficient [31]. Consider the two orderings ϕ_1 and ϕ_2, (v_i, v_j) is called the concordant pair if it is true that $\phi_1(v_i) > \phi_1(v_j)$ and $\phi_2(v_i) > \phi_2(v_j)$, or $\phi_1(v_i) < \phi_1(v_j)$ and $\phi_2(v_i) < \phi_2(v_j)$. Otherwise, (v_i, v_j) is called the discordant pair. Let C denotes the set of concordant pairs and D denotes the set of discordant pairs. The Kendall tau coefficient is

$$\tau(\phi_1, \phi_2) = \frac{2(|C| - |D|)}{n(n-1)} \tag{7}$$

where n is the size of permutation. The Kendall tau coefficient between ϕ_1 and ϕ_2 over Flickr is 0.7. This suggests that reduction of computing large out-degree nodes does not affect the quality of graph ordering obtained. In addition, on Flickr, we show the total score $F(\cdot)$ and the time to compute such $F(\cdot)$ for different huge

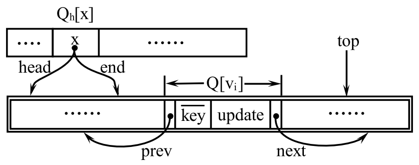


Figure 7: The Priority Queue: $\mathcal{Q}$, $\mathcal{Q}_h$, and top

node sizes in Fig. 6, where a huge node v is a node such that $d_O(v)$ is greater than either $\frac{1}{4}\sqrt{n}$, $\frac{1}{2}\sqrt{n}$, $\sqrt{n}$, $2\sqrt{n}$, $4\sqrt{n}$, or n. Note that the last one, $d_O(v) > n$, means not to reduce computing cost for huge nodes, since no nodes can be pruned. In Fig. 6, the x-axis is $\frac{1}{4}\sqrt{n}$, $\frac{1}{2}\sqrt{n}$, $\sqrt{n}$, $2\sqrt{n}$, $4\sqrt{n}$, and n. We take $F(\cdot)$ and the computing time regarding the one without pruning as the basis. Fig. 6(a) shows that $F(\cdot)$ computed for different huge node sizes change marginally. However, Fig. 6(b) shows that the reduction of computing time is significant. One question is whether dealing with huge nodes affects the approximation ratio given in Theorem 3.1. On one hand, Fig. 6(a) shows that dealing with the huge nodes has little influence to the quality of the output graph ordering in practice, even though Algorithm 2 with the special consideration of huge nodes may possibly fail to choose the real v_{max} during the iterations. On the other hand, the theoretical guarantee by Theorem 3.1 is just a lower bound. As shown in datasets Facebook and AirTrafficControl in Section 3.1, $F(\cdot)$ of the output graph ordering is much better than the given theoretical guarantee in practice. Given the limit influence of huge nodes, the graph ordering generated by pruning the huge nodes can still be better than the bound given by Theorem 3.1.

3.3 The Priority Queue and its Operations

We discuss the priority queue $\mathcal{Q}$ and its operations. There are three operations, namely, $\mathsf{incKey}(v)$, $\mathsf{decKey}(v)$, and $\mathsf{pop}()$. Behind the operations, there is an implicit operation that is to access the key of a node requested irrespective of its position in $\mathcal{Q}$. One way to design $\mathcal{Q}$ is to adapt the same data structure used in [10] as follows. First, to ensure accessing the key of a node v in constant time, the priority queue $\mathcal{Q}$ is designed as an array of the size n. Given a node v_i in the graph G, the key value is stored in $\mathcal{Q}[v_i]$. Second, in order to maintain the max value at the top, every node v_i may be re-positioned. To do so, $\mathcal{Q}$ is designed as a double-linked data structure. That is, in $\mathcal{Q}[v_i]$, where v_i is the original node ID, it has two pointers, namely, $\mathsf{prev}(v_i)$ and $\mathsf{next}(v_i)$, in addition to the key value of $\mathsf{key}(v_i)$. Third, to access nodes by a specific key value, on top of $\mathcal{Q}$, there is a head table, denoted as $\mathcal{Q}_h$, where $\mathcal{Q}_h[x]$ points to the first node in $\mathcal{Q}$ whose key is x. This data structure can be used to support $\mathsf{incKey}()$, $\mathsf{decKey}()$, and $\mathsf{pop}()$ in $O(1)$, $O(1)$, and $O(n)$, respectively. However, we observe that the number of $\mathsf{pop}()$ is much smaller than the number of $\mathsf{incKey}()$ and $\mathsf{decKey}()$. Take the data structure used in [10] as an example. Although it can achieve $O(1)$ for updates by $\mathsf{incKey}()$ and $\mathsf{decKey}()$, it needs to adjust the linked list for every $\mathsf{incKey}()$ and every $\mathsf{decKey}()$, and the number of possible updates is $O(\sum_u (d_O(u))^2)$, which is very high.

The issue is whether we can reduce the number of times of adjusting the linked list while keeping the time complexity for all the three operations the same. In this paper, we propose a lazy-updating strategy by which we reduce the number of adjusting linked list significantly.

Fig. 7 shows the $\mathcal{Q}$ data structure we use in this work. First, in every $\mathcal{Q}[v_i]$, in addition to the two components, $\mathsf{prev}(v_i)$ and $\mathsf{next}(v_i)$, we use $\overline{\mathsf{key}}(v_i)$ and $\mathsf{update}(v_i)$ together instead of maintaining $\mathsf{key}(v_i)$, such that $\mathsf{key}(v_i) = \overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i)$. The priority queue $\mathcal{Q}$ is represented as a doubly linked list by $\mathsf{prev}(\cdot)$ and $\mathsf{next}(\cdot)$ with decreasing $\overline{\mathsf{key}}(\cdot)$ values. The main idea here is that when we update key values, we keep the difference in the update where possible. By this, we only adjust the doubly linked list when $\overline{\mathsf{key}}(v_i)$ has to be changed, and do not adjust the doubly linked list for every updates. Second, in the head table $\mathcal{Q}_h$, we maintain two pointers in $\mathcal{Q}_h[x]$, namely, $\mathsf{head}(x)$ and $\mathsf{end}(x)$. Here, $\mathsf{head}(x)$ and $\mathsf{end}(x)$ point to the first and the last element in $\mathcal{Q}$ such that every node v that has the same key value ($\mathsf{key}(v) = x$) is positioned between $\mathsf{head}(x)$ and $\mathsf{end}(x)$ in the doubly linked list. Note that the length of $\mathcal{Q}_h$ can be as large as the maximum key value, which is far smaller than n in practice. Third, we add a new pointer top that always points to the node having the largest $\overline{\mathsf{key}}(\cdot)$.

With the above data structure, when popping v_{max}, we maintain the true $\mathsf{key}(v_i)$ for a node v_i by $\overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i)$ such that (a) for the top node, $\mathsf{key}(\mathsf{top}) = \overline{\mathsf{key}}(\mathsf{top})$, and (b) for any other node v_i, $\mathsf{key}(v_i) \leq \overline{\mathsf{key}}(v_i)$. Following the conditions below,

$$\begin{aligned} \mathsf{update}(\mathsf{top}) &= 0 \\ \mathsf{update}(v_i) &\leq 0 \quad \text{for } v_i \neq \mathsf{top} \\ \overline{\mathsf{key}}(\mathsf{top}) &\geq \overline{\mathsf{key}}(v_i) \\ \overline{\mathsf{key}}(\mathsf{top}) + \mathsf{update}(\mathsf{top}) &\geq \overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i) \end{aligned} \tag{8}$$

for any $v_i \neq \mathsf{top}$ in $\mathcal{Q}$, we guarantee that the top in $\mathcal{Q}$ is the one having the largest true key value $\mathsf{key}(\cdot)$. We do not need to adjust the doubly linked list when updating every time. There are only two cases we need to adjust the doubly linked list. First, when $\mathsf{update}(v_i) > 0$ after updating v_i, we have to make $\mathsf{update}(v_i) \leq 0$. In this process, we do $\overline{\mathsf{key}}(v_i) \leftarrow \overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i)$ and $\mathsf{update}(v_i) \leftarrow 0$. Since $\overline{\mathsf{key}}(v_i)$ is changed, we need to adjust the doubly linked list. Second, when selecting v_{max} to be popped, we make $\mathsf{update}(\mathsf{top}) = 0$ as given in Eq. (8), and therefore the adjustment of doubly linked list is needed.

Note that we increase the computational cost for pop marginally within the same time complexity of $O(n)$, and at the same time we significantly reduce the number of adjusting doubly linked list, since the number of updates is huge. We discuss the priority queue operations below.

First, for $\mathsf{decKey}(v_i)$, we only reduce the value in $\mathsf{update}(v_i)$ by 1 to maintain $\mathsf{key}(v_i) = \overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i)$. Here, since $\mathsf{update}(v_i) \leq 0$, there is no need to update $\overline{\mathsf{key}}(v_i)$.

Algorithm 3 decKey (v_i)

```
1: $\mathsf{update}(v_i) \leftarrow \mathsf{update}(v_i) - 1$;
```

Second, for $\mathsf{incKey}(v_i)$, we show it in Algorithm 4. We increase the update value by 1. We only adjust doubly linked list when $\mathsf{update}(v_i) > 0$. Since at this point $\mathsf{update}(v_i) = 1$, we increase $\overline{\mathsf{key}}(v_i)$ by 1 and make $\mathsf{update}(v_i) = 0$. This ensures $\mathsf{key}(v_i) = \overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i)$ and $\mathsf{update}(v_i) \leq 0$. Also, since $\overline{\mathsf{key}}(v_i)$ is changed, its position in $\mathcal{H}$ needs to be adjusted. We adjust the position by deleting/inserting it from/into $\mathcal{Q}$. Accordingly, we adjust the head table $\mathcal{Q}_h$ and adjust top if the new $\overline{\mathsf{key}}(v_i)$ is greater than $\overline{\mathsf{key}}(\mathsf{top})$, if needed.

Third, for $\mathsf{pop}()$ (Algorithm 5), recall that the true $\mathsf{key}(\mathsf{top})$ is $\overline{\mathsf{key}}(\mathsf{top})$ if $\mathsf{update}(\mathsf{top}) = 0$, because $\mathsf{key}(\mathsf{top}) = \overline{\mathsf{key}}(\mathsf{top}) + \mathsf{update}(\mathsf{top})$. When popping the node with the max true key value, we simply return the node pointed by top if $\mathsf{update}(\mathsf{top}) = 0$. However, when $\mathsf{update}(\mathsf{top}) < 0$, we need to check whether the node pointed by top is the node with the max true key value. (i) we make $\overline{\mathsf{key}}(v_i)$ as the true key value by $\overline{\mathsf{key}}(v_i) \leftarrow \overline{\mathsf{key}}(v_i) +$

Algorithm 4 incKey (v_i)

```
1: $\mathsf{update}(v_i) \leftarrow \mathsf{update}(v_i) + 1$;
2: **if** $\mathsf{update}(v_i) > 0$ **then**
3:     $\mathsf{update}(v_i) \leftarrow 0$, $x \leftarrow \overline{\mathsf{key}}(v_i)$, $\overline{\mathsf{key}}(v_i) \leftarrow \overline{\mathsf{key}}(v_i) + 1$;
4:     delete $v_i$ from $\mathcal{Q}$;
5:     insert $v_i$ into $\mathcal{Q}$ in the position just before $\mathsf{head}[x]$;
6:     update the head $\mathcal{Q}_h$ array accordingly;
7:     **if** $\overline{\mathsf{key}}(v_i) > \overline{\mathsf{key}}(\mathsf{top})$ **then**
8:         $\mathsf{top} \leftarrow v_i$;
```

Algorithm 5 pop ()

```
1: **while** $\mathsf{update}(\mathsf{top}) < 0$ **do**
2:     $v_t \leftarrow \mathsf{top}$;
3:     $\overline{\mathsf{key}}(v_t) \leftarrow \overline{\mathsf{key}}(v_t) + \mathsf{update}(v_t)$;
4:     $\mathsf{update}(v_t) \leftarrow 0$;
5:     **if** $\overline{\mathsf{key}}(\mathsf{top}) \leq \overline{\mathsf{key}}(\mathsf{next}(\mathsf{top}))$ **then**
6:         adjust the position of $v_t$ and insert $v_t$ just after $u$ in $\mathcal{Q}$, such that
           $\overline{\mathsf{key}}(u) \geq \overline{\mathsf{key}}(\mathsf{top})$ and $\overline{\mathsf{key}}(\mathsf{next}(u)) < \overline{\mathsf{key}}(\mathsf{top})$;
7:         $\mathsf{top} \leftarrow \mathsf{next}(\mathsf{top})$;
8:     update the head array;
9: $v_t \leftarrow \mathsf{top}$;
10: remove the node pointed by top from $\mathcal{Q}$ and update $\mathsf{top} \leftarrow \mathsf{next}(\mathsf{top})$;
11: **return** $v_t$;
```

$\mathsf{update}(v_i)$ followed by changing $\mathsf{update}(v_i)$ to be zero (line 3-4). (ii) we reposition the node v_t pointed by the current top to a new position in $\mathcal{Q}$ following the sorting order of the $\overline{\mathsf{key}}$ values, and update top pointer to point to the node having the largest $\overline{\mathsf{key}}$ value currently. We repeat it until $\mathsf{update}(\mathsf{top}) = 0$. The node returned by pop() is the node with the max true key value.

An example is given in Example-C in the Appendix.

Given the specific priority queue operations, we show the time complexity of *GO-PQ* below.

Theorem 3.4: *The time complexity of GO-PQ is $O(\sum_{u \in V}(d_O(u))^2)$.*

The proof sketch is given in Appendix.

The Constant factor: Comparing the data structure used in [10], the key difference is we shift the cost from updates (incKey and decKey) to pop while keeping all the operations in the same time complexity. The reason is that the number of updates ($\mathsf{decKey}(v_i)$ and $\mathsf{incKey}(v_i)$) is much larger than the number of pop(). The former is $O(\sum_{u \in V}(d_O(u))^2)$ whereas the latter is $O(n)$. Note that pop() (Algorithm 5) only takes a few iterations for the while loop to stop in practice, because the linked list is sorted in the decreasing order of $\overline{key}$, even though it is $O(n)$. We discuss how much we can save in updates. Since $\mathsf{decKey}(v_i)$ only needs to update $\mathsf{update}(v_i)$ in constant time $O(1)$, we focus on Algorithm 4 for $\mathsf{incKey}(v_i)$. There is a condition of $\mathsf{update} > 0$ in line 2. The adjustment of doubly linked list is done only when the condition is true. The question is how much we can save by this condition. To analyze the saving, we consider a key update sequence by $\mathsf{incKey}(v_i)$ and $\mathsf{decKey}(v_i)$ for a given v_i before v_i is adjusted by pop. During the period in $\mathcal{Q}$, the number of $\mathsf{incKey}(v_i)$ depends on the insertion of node v_e into the window, and the number of $\mathsf{decKey}(v_i)$ depends on the deletion of node v_b from the window, and v_e will become v_b after w iterations. Therefore, in the sequence, the number of $\mathsf{incKey}(v_i)$ is similar to the number of $\mathsf{decKey}(v_i)$. Based on this observation, we assume that the key updates on a given v_i has 0.5 probability to be incKey and 0.5 probability to be decKey. Suppose the total number of updates for a given node v_i is L, and the total number of updates that need to adjust the doubly linked list following the condition of $\mathsf{update}(v_i) > 0$ is $\lambda^L_{v_i}$. There are two extreme cases of the key update sequence with the equal numbers of incKey and decKey w.r.t. a node v_i. Let +1 and -1 represent incKey and decKey. The best case: incKey and decKey occur one by one alternatively in the sequence, (+1, -1, +1, -1, ...). Here, the first incKey triggers the adjustment of doubly linked list and sets $\mathsf{update}(v_i)$ to be 0. In the following sequence, for any update, $\mathsf{update}(v_i) \leq 0$ because the sum of its previous operations is ≤ 0, and therefore no adjustment is needed. The total number of doubly linked list needed is 1 not matter how large L is. The worst case: all incKey followed by all decKey. Here, we need to adjust the doubly linked list for every incKey in total of $\frac{L}{2}$ times. We can save a half of link adjustments in the worst case. Below we show the worst case hardly occurs, and give the expected number of adjustments $\lambda^L_{v_i}$ in Theorem 3.5.

Dataset	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	d_{avg}
Pokec	1.63M	30.62M	18.8
Flickr	2.30M	33.14M	14.4
LiveJournal	4.84M	68.47M	14.1
wikilink	11.19M	340.24M	30.4
Google+	28.94M	462.99M	16.0
pld-arc	42.88M	623.05M	14.5
twitter	61.57M	1,468.00M	23.8
sd1-arc	94.94M	1,937.00M	20.4

Table 2: Real Graphs

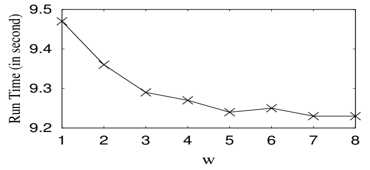


Figure 8: Window Size w

Theorem 3.5: *For a given node v_i over the random key update sequence of length L on v_i, and a sufficiently large L, the expected number of adjusting doubly linked list is as follows.*

$$E(\lambda^L_{v_i}) \sim \sqrt{\frac{2L}{\pi}} \tag{9}$$

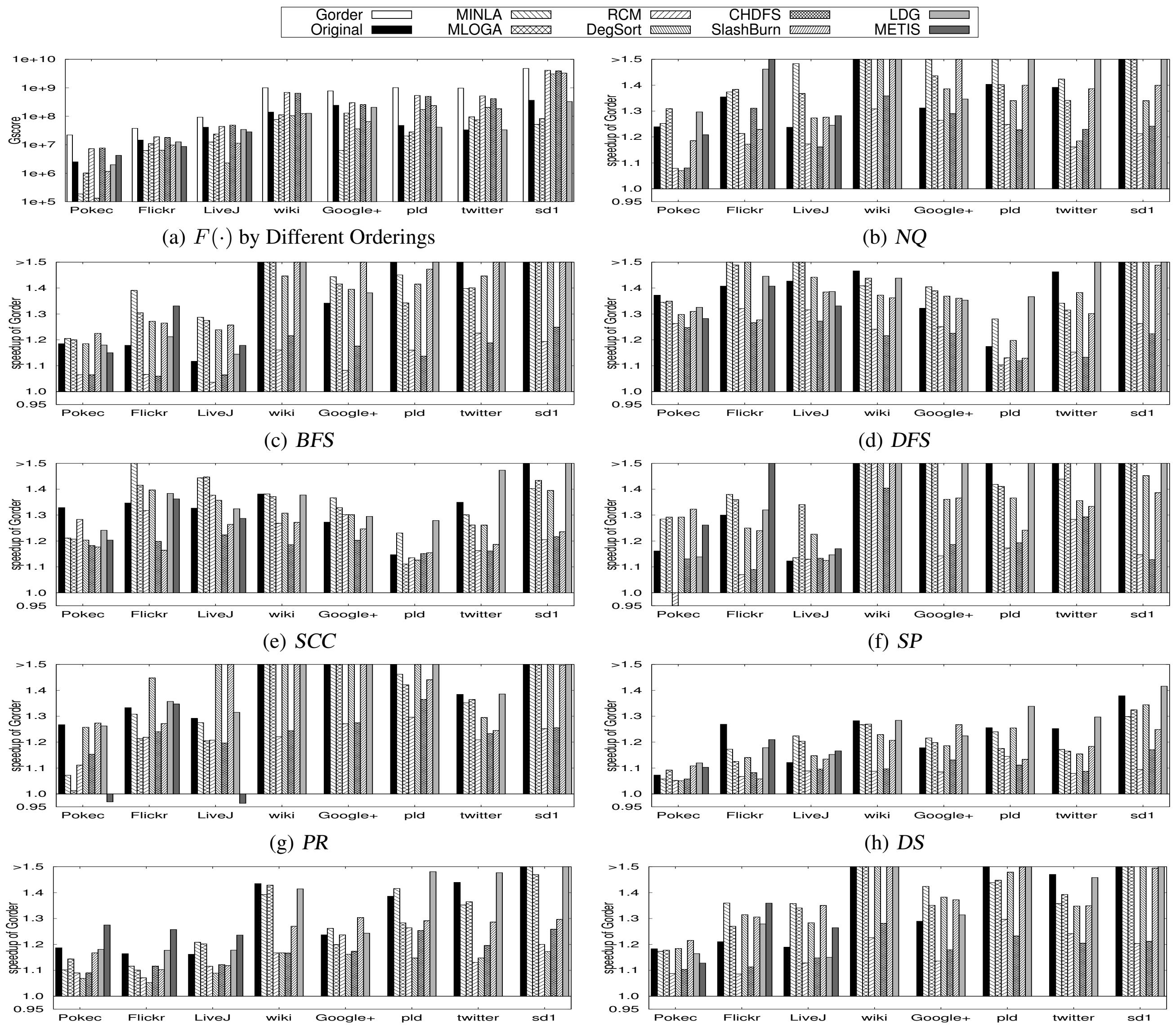
The proof sketch is given in Appendix.

We tested the performance of lazy-update strategy in Flickr. There are over 8 billion key update operations in total. By our lazy-update strategy, only 0.66 billion key updates really trigger the adjustments of doubly linked list, and about 40% computing time is saved as shown in Section 4. This indicates that the constant factor of key update operations by *GO-PQ* is very small, which allows us to handle large graphs with billions of nodes and edges.

Finally, the space complexity of *GO-PQ* is $O(n)$, because the size of the priority queue is linear with the number of nodes, and array P keeping the output permutation result is at most as large as the node set.

4. EVALUATION

We evaluate our Gorder by the *GO-PQ* algorithm, in comparison with other 9 ordering approaches, using 8 real large graphs and 9 graph algorithms. The graph data is stored in *compressed sparse row* (CSR) format [1], which is equivalent to adjacency list. All implementations for graph algorithms and graph orderings are done using C++ and compiled by G++ 4.9.2. The extensive experiments have been conducted on two machines, (a) a Linux server with Intel Core i7-4770@3.40GHz CPU and 32 GB memory, (b) a Linux server with Intel Xeon X5550@2.67GHz CPU and 24 GB memory. We report our findings on the configuration of (a), since all the results obtained in these two machines are similar. The L1, L2, L3 cache size of the machine (a) is 64KB, 256KB, 8MB, re-



(a) $F(\cdot)$ by Different Orderings (b) *NQ*

(c) *BFS* (d) *DFS*

(e) *SCC* (f) *SP*

(g) *PR* (h) *DS*

(i) *Kcore* (j) *Diam*

Figure 9: The Speedup of Gorder

Order	L1-ref	L1-mr	L3-ref	L3-r	Cache-mr
Original	11,109M	52.1%	2,195M	19.7%	5.1%
MINLA	11,110M	58.1%	2,121M	19.0%	4.5%
MLOGA	11,119M	53.1%	1,685M	15.1%	4.1%
RCM	11,102M	49.8%	1,834M	16.5%	4.1%
DegSort	11,121M	58.3%	2,597M	23.3%	5.3%
CHDFS	11,107M	49.9%	1,850M	16.7%	4.4%
SlashBurn	11,096M	55.0%	2,466M	22.2%	4.3%
LDG	11,112M	52.9%	2,256M	20.3%	5.4%
METIS	11,105M	50.3%	2,235M	20.1%	5.2%
Gorder	11,101M	**37.9%**	**1,280M**	**11.5%**	**3.4%**

Table 3: Cache Statistics by *PR* **over** Flickr **(M = Millions)**

Order	L1-ref	L1-mr	L3-ref	L3-r	Cache-mr
Original	623.9B	58.4%	180.0B	28.8%	18.6%
MINLA	628.8B	62.5%	196.6B	31.2%	14.8%
MLOGA	620.0B	62.1%	189.6B	30.5%	14.3%
RCM	628.9B	44.9%	103.8B	16.5%	10.2%
DegSort	632.2B	55.1%	149.5B	23.6%	15.9%
CHDFS	630.3B	38.0%	101.2B	16.1%	10.9%
SlashBurn	628.8B	44.5%	121.0B	19.3%	13.7%
LDG	637.9B	58.4%	186.2B	29.2%	18.6%
Gorder	620.3B	**31.5%**	**79.5B**	**12.8%**	**8.2%**

Table 4: Cache Statistics by *PR* **over** sd1-arc **(B = Billions)**

spectively. The efficiency is measured by the CPU time and the CPU cache miss ratio.

The eight real datasets we tested are with at least 1 million nodes and 30 million edges as shown in Table 2. They are mainly from SNAP (Pokec, LiveJournal) and KONECT (Flickr, wikilink). Here, Pokec, Flickr, LiveJournal, Google+[3][24] and twitter[4] are online social networks. wikilink is the hyperlink graph inside the English Wikipedia, and pld-arc and sd1-arc are two large hypelink graphs crawled in 2012[5]. The average degree is in the range from 14 to 30. twitter and sd1-arc are two billion-edge graphs.

[3]www.cs.berkeley.edu/~stevgong/dataset.html

[4]an.kaist.ac.kr/traces/WWW2010.html

[5]webdatacommons.org/hyperlinkgraph/

Order	*NQ*	*BFS*	*DFS*	*SCC*	*SP*	*PR*	*DS*	*Kcore*	*Diam*
Pokec	8.7	2.0	2.5	5.2	1.3	12.3	10.4	6.6	1,003
Flickr	5.1	1.5	1.8	3.7	1.0	9.1	8.6	5.3	620
LiveJ	19.4	4.9	5.9	12.1	4.6	26.4	24.0	16.8	2,556
wikilink	56.1	10.0	14.3	28.5	35.3	81.9	85.7	50.0	5,932
Google+	134	35.0	43.3	87.6	28.6	210	183	131	17,936
pld-arc	199	45.2	55.7	115	40.4	305	251	177	14,389
twitter	467	79.2	80.9	158	74.4	819	535	378	32,808
sd1-arc	492	83.7	104	218	120	665	587	430	30,202

Table 5: Running time by Gorder (in second)

The night graph algorithms: The graph algorithms being tested are Neighbors Query (*NQ*), Breadth-First Search (*BFS*) [20], Depth-First Search (*DFS*) [20], Strongly Connected Component (*SCC*) detection [48], Shortest Paths (*SP*) by the Bellman-Ford algorithm [20], PageRank (*PR*) [39], Dominating Set (*DS*) [19], graph decomposition (*Kcore*) [6] and graph diameter (*Diam*). All of them are fundamental for graph computing and provide the basis for the design of other advanced graph algorithms. For the accuracy of the collected information, except *NQ*, PageRank and *Diam*, we repeat these algorithms 10 times when running the experiments and report the total running time and the cache statistics. *NQ* randomly chooses $10n$ nodes for each dataset and accesses their out-neighbors. PageRank algorithm is stopped after 100 iterations in every dataset. The approximate graph diameter is obtained by finding the longest shortest distance of 5,000 randomly chosen nodes.

The night orderings: We compare our Gorder with the following 9 graph orderings: Original, MINLA, MLOGA, RCM, DegSort, CHDFS, SlashBurn, LDG, and METIS. Original uses the node IDs that come with the real graphs. MINLA is the node ordering π for the Minimum Linear Arrangement problem, which is to minimize $\sum_{(u,v)\in E(G)} \|\pi(u) - \pi(v)\|$. We have tried several existing solutions for this problem [41, 44], but none of them can be scalable to deal with large graphs effectively. Instead, we use simulated annealing technique to compute the result, which has good scalability and shows comparable performance with the state-of-the-art solutions. MLOGA is the node ordering for the Minimum Logarithmic Arrangement problem [45], that is to minimize $\sum_{(u,v)\in E(G)} \log_2(\|\pi(u) - \pi(v)\|)$. Similarly, we use simulated annealing to generate the node ordering. RCM is ordered by the reverse Cuthill-McKee algorithm [22], which is the well known solution for reducing graph bandwidth. DegSort is the node orders by sorting nodes in descending order of in-degree. CHDFS is obtained by children-depth-first traversal which is proposed in [4]. SlashBurn is the graph ordering used in graph compression algorithm [29]. LDG is the state-of-the-art streaming graph partitioner [49]. METIS is the de-facto standard offline graph partitioner [30] which is widely used in many applications. We set the partition size to be 64 for the two graph partitioners LDG and METIS to fit the size of cache line. Our Gorder is computed by *GO-PQ* with the window size $w = 5$. To determine the window size, we show the performance by different window sizes when testing *PR* in 100 iterations over Flickr in Fig. 8. The performance of *PR* improves when the window size becomes larger. But when $w > 5$, the improvement of running time is marginal. It is similar when we test other algorithms over different datasets. These orderings are used to renumber the node IDs in the datasets.

Exp-1 The Gscore and the CPU Cache Miss Ratio: We show the Gscore (Eq. (3)) by different orderings in Fig. 9(a), where the y-axis is in logarithmic. The Gscore of Gorder by *GO-PQ* is the highest in all the 8 real datasets, and it can achieve one order of magnitude higher than the Original ordering. RCM and CHDFS are second to Gorder but their Gscore are only a half of the Gscore by Gorder in most datasets. The higher Gscore means the better

Order	*NQ*	*BFS*	*DFS*	*SCC*	*SP*	*PR*	*DS*	*Kcore*	*Diam*
Original	50.8	15.3	5.4	7.8	21.5	52.1	21.9	20.8	14.9
MINLA	51.8	18.0	5.5	8.1	24.6	58.1	22.1	21.5	17.9
MLOGA	41.7	16.3	5.1	7.2	21.9	53.1	21.1	20.6	16.4
RCM	49.1	12.1	4.6	6.6	15.9	49.7	20.3	20.2	12.4
DegSort	45.7	16.7	4.8	7.0	24.9	58.3	21.4	18.6	17.0
CHDFS	42.1	12.3	4.1	5.8	18.5	49.9	21.1	20.6	12.9
SlashBurn	46.2	16.0	4.5	6.2	22.1	55.0	20.7	21.3	15.8
LDG	50.7	15.9	5.8	8.2	21.8	52.9	22.4	21.2	14.9
METIS	63.0	18.2	7.7	10.1	20.8	50.3	23.0	21.7	16.7
Gorder	**35.4**	**11.1**	**3.6**	**5.2**	**12.8**	**37.9**	**18.7**	**18.1**	**10.9**

Table 6: L1 Cache Miss Ratio on Flickr (in percentage %)

Order	*NQ*	*BFS*	*DFS*	*SCC*	*SP*	*PR*	*DS*	*Kcore*	*Diam*
Original	76.5	20.0	9.4	13.0	17.5	58.4	21.7	20.0	17.5
MINLA	76.0	22.7	10.2	12.8	20.7	62.5	21.8	20.5	18.3
MLOGA	76.0	21.7	9.4	12.3	19.8	62.1	21.8	20.6	18.5
RCM	61.6	14.4	7.5	8.7	**8.9**	44.9	18.2	17.5	11.7
DegSort	59.3	18.7	8.0	12.1	16.6	55.1	21.9	16.9	15.5
CHDFS	50.0	14.2	5.1	8.3	13.2	38.0	18.4	16.1	10.4
SlashBurn	56.6	16.8	6.7	9.3	10.2	44.5	18.9	16.8	13.5
LDG	74.7	22.7	10.0	13.6	18.7	58.4	22.0	20.3	17.9
Gorder	**40.0**	**12.1**	**4.6**	**7.2**	10.8	**31.5**	**16.9**	**14.5**	**9.5**

Table 7: L1 Cache Miss Ratio on sd1-arc (in percentage %)

graph locality which makes the graph accesses are more likely to be hit by the fast L1, L2 cache. Table 3 and Table 4 show the CPU cache statistics of running *PR* on two datasets, Flickr and sd1-arc, respectively, where M and B denote million and billion. Note that we do not show METIS in Table 4, because METIS fails to compute the graph partitions for the other 5 larger graph except Pokec, Flickr and LiveJournal, due to its excessive memory consumption. The cache statistics are collected by the perf tool.[6] Here, L1-ref denotes the number of L1 cache references, which is the total cache access number, because all cache accesses must be checked by the L1 cache firstly. L1-mr denotes the L1 cache miss ratio (the % of L1 cache miss numbers over L1-ref). L3-ref is the number of L3 cache reference. L3-r is the ratio of the cache reference checked in L3 cache, such that L3-r = L3-ref/L1-ref. A small L3-r means that most cache references are hit by the first two level fast cache, and can be effectively used to measure the performance of the first 2 level fast cache. Cache-mr denotes the percentage of cache references missed in all level of cache over the L1-ref.

In Table 3 and Table 4, L1-refs are similar by different orderings. This is because cache access numbers for the same algorithm to be tested are similar. The best results for the other measures are highlighted in **bold**. Gorder achieves the smallest cache miss numbers in all levels of cache, and reduces the cache miss ratio significantly, especially in the first two cache levels. Compared with Original, Gorder reduces over 30% cache miss numbers in every cache level for Flickr and nearly 50% cache miss numbers for sd1-arc. As indicated by Fig. 9(a) from the viewpoint of Gscore, RCM and CHDFS are second to Gorder. From the number of cache misses, RCM occurs 30% more cache miss numbers than Gorder in Flickr and CHDFS occurs 20% more cache miss numbers than Gorder in sd1-arc. The cache misses of MINLA, MLOGA and DegSort are nearly 2 times of Gorder, and METIS orderings occur 30% more cache misses than Gorder on average.

Exp-2 The running time speedup by *GO-PQ*: We show the speedup by Gorder comparing with the other orderings. As the basis of comparison, we first show the running time in second by Gorder for all graph algorithms over the large real graphs in Table 5. Fig. 9 shows the speedup of Gorder over the other 9 orderings, for every one of the 9 algorithms on the 8 large graphs. Here, the speedup

[6]perf.wiki.kernel.org/index.php

	Pokec	Flickr	LiveJ	wiki	G+	pld	twitter	sd1
T_w	13.4s	32.3s	39.5s	181s	719s	0.70h	1.48h	1.57h
T_{wo}	40.6s	140.8s	97.4s	536s	0.74h	1.84h	6.02h	4.24h
R_w	93%	92%	91%	82%	93%	97%	96%	97%

Table 8: Computing Time of *GO-PQ* (s: second, h: hour)

	Pokec	Flickr	LiveJ	wiki	G+	pld	twitter	sd1
MINLA	445	510	828	1,796	2,099	2,437	4,134	6,876
MLOGA	5,377	4,324	5,001	14,869	6,431	9,424	12,077	33,169
RCM	1.45	1.75	3.48	14.95	29.39	45.46	144	120
DegSort	0.14	0.17	0.44	1.01	2.86	4.35	5.92	9.72
CHDFS	0.53	0.45	1.34	3.52	10.60	13.47	27.68	36.82
SlashBurn	1,309	231	2,110	5,471	18,363	10,690	12,986	22,649
LDG	12.00	21.82	103	586	4,351	9,987	21,078	25,024
METIS	209	571	492	–	–	–	–	–
Gorder	13.43	32.30	39.54	181	719	2,502	5,339	5,665

Table 9: Graph Ordering Time (in second)

of Gorder over another ordering X is shown as the relative difference of $\frac{T(X)}{T(\text{Gorder})}$, where $T(X)$ indicates the running time of algorithm on the ordering X. Note that the running time of $T(\text{Gorder})$ is shown in Table 5 as the basis. In Fig. 9, for an ordering X, if its y-axis value $\beta > 1$, it means that Gorder outperforms the ordering X in β times. Otherwise, the ordering X outperforms Gorder. There are in total 72 (8×9) pairs of graph algorithms and graph datasets. From Fig. 9, we see that the graph algorithms run fastest using Gorder. Gorder can achieve the best with the positive speedup (> 1) compared to the other orderings over 69 out of 72 cases, in nearly every testing. The best speedup of Gorder over the Original ordering is larger than 2. There are only 3 out of 72 cases that Gorder fails to win the best (speedup < 1) but Gorder is very close to the best results (speedup ≥ 0.92). Besides, we have the following observations. (a) The speedup of the running time by Gorder becomes more significant while the graph size increases. Compared with Pokec, Flickr and LiveJournal, the other 5 datasets are much larger with at least 10 million nodes and 300 million edges. Given the small CPU cache, only limited graph information can be located within the CPU cache and hence the effect of the cache usage becomes more significant for large graphs. (b) Gorder shows different speedup performance for different graph algorithms. *DS* and *Kcore* algorithms use a heap-like data structure which is not cache friendly in nature and thus influences the speedup of Gorder. RCM is ordered by a BFS-like graph ordering method and so it has comparable performance with Gorder in *BFS* and *SP* algorithms. (c) The speedup results are consistent with the cache performance of different orderings shown in the Table 3 and Table 4. Furthermore, to give the details, we show the L1 CPU cache miss ratio of all the 10 orderings when testing the 9 graph algorithms over Flickr and sd1-arc in Table 6 and Table 7. These results confirm that the running time of graph algorithms is greatly influenced by the cache performance.

Exp-3: Computing Gorder **by** ***GO-PQ***: We show the computing time to compute Gorder based on $F(\cdot)$. The basic algorithm Algorithm 1 cannot process the large datasets within the limited time and the results are omitted. We use *GO-PQ* algorithm to compute the graph orderings and show the effectiveness of our new priority queue operations by lazy-update strategy. Let T_w and T_{wo} be the computing time for *GO-PQ* with and without the lazy-update strategy, and let R_w denote the ratio of key update operations pruned by the lazy-update strategy. Table 8 shows that over 90% operations are pruned in most cases. With the lazy-update, *GO-PQ* runs very fast for Pokec, Flickr and LiveJournal that have millions of nodes and edges. For the largest billion-edge graphs like twitter (61.57 millions of nodes and 1,468 millions of edges) and sd1-arc (94.94 millions of nodes and 1,937 millions of edges), the computing time T_w are about 1.5 hours. The computing time of the other orderings are listed in Table 9. DegSort is the fastest because it simply sorts the node set based on the degree information. RCM and CHDFS are also faster than Gorder because both of them are obtained by the variants of BFS traversal. However, it is important to note all graph orderings, including Gorder by *GO-PQ*, are computed offline. The cost saving by using Gorder for graph algorithm is huge, as it is the total number of times of running a graph algorithm multiplied by the cost saving for a single execution of the graph algorithm. For some graph algorithms which require long computation time, the time saving in a single execution is huge enough to cover the offline ordering time, for example, *Diam*. We test *Diam* and show the results in Table 10. For large datasets twitter and sd1-arc, Gorder can save more than 6,000 seconds computing time even comparing with RCM and CHDFS.

Order	Pokec	Flickr	LiveJ	wiki	G+	pld	twitter	sd
Original	1,187	750	3,040	10,503	23,128	21,961	48,238	50,784
MINLA	1,176	843	3,471	10,322	25,543	20,698	44,536	48,218
MLOGA	1,181	787	3,427	10,121	24,229	20,829	45,698	48,899
RCM	1,091	673	2,883	7,272	20,371	18,657	40,730	36,334
DegSort	1,188	815	3,281	9,500	24,799	21,278	44,228	47,723
CHDFS	1,107	690	2,934	7,600	21,150	17,732	39,517	36,585
SlashBurn	1,219	810	3,452	10,031	24,616	21,564	44,261	45,134
LDG	1,168	793	2,940	10,137	23,569	22,740	47,841	55,234
METIS	1,131	843	3,232	–	–	–	–	–
Gorder	**1,003**	**620**	**2,556**	**5,932**	**17,936**	**14,389**	**32,808**	**30,202**

Table 10: Running Time of *Diam* (in second)

Four additional experiments are given in Appendix. We demonstrate that Gorder can further accelerate a specialized BFS algorithm, which aims to optimize the cache performance during the graph traversal. Besides, we test the performance of Gorder under multicore environment and graph systems. We also run graph algorithms with profiling tools in datasets with Gorder. At the end, we study Gorder in dynamic scenario with graph updates.

5. CONCLUSION

In this paper, we explore a general approach to reduce the CPU cache miss ratio for graph algorithms with their implementations and data structures unchanged. This problem is challenging, given graphs are rather complex and the graph algorithms designed are based on different techniques. We study graph ordering, which is to find the optimal permutation ϕ among all nodes in a given graph G by keeping nodes that will be frequently accessed together locally in a window of size w, in order to minimize the CPU cache miss ratio. We prove the graph ordering problem is NP-hard, and give a new *GO-PQ* algorithm with a bounded $\frac{1}{2w}$-approximation in time complexity $O(\sum_{u \in V}(d_O(u))^2)$, where $d_O(u)$ is the out-degree of node u in G. Since the number of updates is huge, we further propose the lazy-update strategy to reduce the cost of maintaining the priority queue significantly. We conducted extensive experimental studies to evaluate our approach in comparison with other 9 possible graph orderings using 8 large real graphs and 9 representative graph algorithms. We confirm the efficiency of our graph ordering. In nearly all testings, our graph ordering outperforms others. The best speedup is > 2 in testing different graph algorithms over large real graphs.

ACKNOWLEDGEMENTS: We thank Zhigang Wang to help conducting the experiments in graph systems. This work was partially supported by the grant of the Research Grants Council of the Hong Kong SAR, China No. 14209314 and Xuemin Lin was partially supported by ARC DP150102728, DP140103578 and NSFC 61232006.

6. REFERENCES

[1] *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *Proc. of VLDB'99*, 1999.

[3] L. Auroux, M. Burelle, and R. Erra. Reordering very large graphs for fun & profit. In *International Symposium on Web AlGorithms*, 2015.

[4] J. Banerjee, W. Kim, S. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE Trans. Software Eng.*, 1988.

[5] A. I. Barvinok, D. S. Johnson, G. J. Woeginger, and R. Woodroofe. The maximum traveling salesman problem under polyhedral norms. In *Proc. of IPCO'98*, 1998.

[6] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[7] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proc. of WWW'11*, 2011.

[8] P. Boldi, M. Santini, and S. Vigna. Permuting web graphs. In *Proc. of WAW'09*, 2009.

[9] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. *Technology*, 2005.

[10] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proc. of SIGMOD'13*, 2013.

[11] M. Charikar, M. T. Hajiaghayi, H. Karloff, and S. Rao. l2 spreading metrics for vertex ordering problems. In *Proc. of SODA'06*, 2006.

[12] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proc. of ICDE'04*, 2004.

[13] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proc. of SIGMOD'01*, 2001.

[14] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. of KDD'09*, 2009.

[15] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of PLDI, Atlanta, Georgia, USA*, 1999.

[16] P. Z. Chinn, J. Chvatalova, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices - a survey. *Journal of Graph Theory*, 6(3), 1982.

[17] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3), 1979.

[18] J. Cieslewicz and K. Ross. Database optimizations for modern hardware. *Proc of the IEEE*, 96(5), 2008.

[19] E. Cockayne. Domination of undirected graphs - a survey. In *Theory and Applications of Graphs*, pages 141–147. Springer, 1978.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, 2 edition, 2001.

[21] M. Fisher, G. Nemhauser, and L. Wolsey. An analysis of approximations for finding a maximum weight hamiltonian circuit. *Operations Research*, 27(4), 1979.

[22] A. George and J. W. Liu. Computer solution of large sparse positive definite. 1981.

[23] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proc. of VLDB'05*, 2005.

[24] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song. Evolution of social-attribute networks: measurements, modeling, and implications using google+. In *Proc. of IMC'12*, 2012.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI Hollywood, CA, USA*, 2012.

[26] C. M. Grinstead and J. L. Snell. *Introduction to probability*. American Mathematical Soc., 2012.

[27] L. H. Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 1964.

[28] R. Hassin and S. Rubinstein. An approximation algorithm for the maximum traveling salesman problem. *Inf. Process. Lett.*, 67(3), 1998.

[29] U. Kang and C. Faloutsos. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *Proc. of ICDM'11*, 2011.

[30] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1), 1998.

[31] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 1938.

[32] Y. Koren and D. Harel. A multi-scale algorithm for the linear arrangement problem. In *Graph-Theoretic Concepts in Computer Science*. Springer, 2002.

[33] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *OSDI Hollywood, CA, USA*, 2012.

[34] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proc. of WWW'08*, 2008.

[35] P. Lindstrom and D. Rajan. Optimal hierarchical layouts for cache-oblivious search trees. In *Proc. of ICDE'14*, 2014.

[36] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. of ASPLOS'96*, 1996.

[37] The Apache Software Foundation. Giraph website. http://giraph.apache.org.

[38] A. O. Mendelzon and C. G. Mendioroz. Graph clustering and caching. In *Computer Science 2*. Springer, 1994.

[39] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[40] J. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel Distrib. Syst.*, 15(9), 2004.

[41] J. Petit. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics (JEA)*, 2003.

[42] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proc. of VLDB'99*, 1999.

[43] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *Proc. of SIGMOD'00*, 2000.

[44] I. Safro, D. Ron, and A. Brandt. Multilevel algorithms for linear ordering problems. *Journal of Experimental Algorithmics (JEA)*, 2009.

[45] I. Safro and B. Temkin. Multiscale approach for the network compression-friendly ordering. *Journal of Discrete Algorithms*, 2011.

[46] A. I. Serdyukov. An algorithm with an estimate for the traveling salesman problem of the maximum. *Upravlyaemye Sistemy*, 25:80–86, 1984.

[47] Y. Shao, B. Cui, and L. Ma. PAGE: A partition aware engine for parallel graph computation. *IEEE Trans. Knowl. Data Eng.*, 27(2), 2015.

[48] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1), 1981.

[49] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of KDD'12*, 2012.

[50] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *PVLDB*, 8(4), 2014.

[51] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3), 2013.

[52] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14), 2013.

[53] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14), 2014.

APPENDIX

CPU caches: In computer systems, the CPU cache plays a very important role to speedup computing by loading data from comparative low speed main memory to high speed CPU cache and keeping frequently accessed data in the CPU cache. In brief, most modern CPUs are equipped with three-level caches, namely, L1/L2/L3, where an individual core is associated with its L1 and L2 caches, and many cores share the same L3 cache. With the three-level CPU

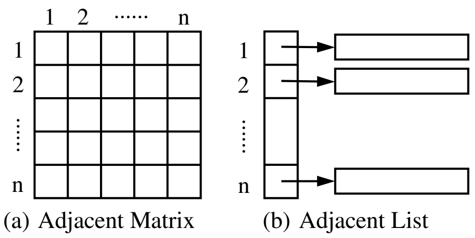


Figure 10: Graph Representations

cache, in computing, the CPU checks whether the requested data is in the fastest L1 cache, then L2 cache and the slowest L3 cache. If all CPU caches do not have the data, then the CPU copies a unit of data pieces containing the requested data from the main memory to the CPU cache. The unit of data to be transferred between the CPU cache and the main memory is a CPU cache line, which is small in size, e.g., 64 bytes (64B). In general, the L1 cache is smallest and fastest, the L2 cache is relative larger and comparative slower, and the L3 cache is the largest and is the slowest. Take Intel CPU since Core i3 as an example. The sizes of L1 and L2 are 64KB and 256KB, and the size of L3 is in the range of 3MB to 20MB. The latency of L1, L2, L3 cache and memory is about 1, 3, 10, and 60 nanoseconds, respectively, if the corresponding cache miss occurs.[7] The latency of memory is high due to the fact that the CPU needs to copy the cache line containing data from main memory to the CPU cache. Given the fact that there exists a wide speed gap between the CPU cache and the memory, reducing the cache miss ratio on all three level CPU caches can significantly improve the performance. On the other hand, the long latency of the memory access will exist in a long period of time due to the "memory wall" effect, which is caused by the fact that the slow improvement of memory access time in chip industry cannot follow the pace of the processor's increasing clock frequency [9].

Graph algorithm implementation: To implement graph algorithms, the well-known data structures to represent a graph $G = (V, E)$ are adjacency-matrix and adjacency-list [20]. The adjacency-matrix is not the first choice in practice since it represents a graph using a matrix which consumes large memory space, $n \times n$, where n is the number of nodes (Fig. 10(a)). The adjacency-list maintains a graph G using an array of n lists. First, there is an array Adj of size n to maintain all nodes. Second, for any node v in the array Adj, denoted as $Adj[v]$, there is a list of neighbor nodes u, if there is an edge (v, u) in E. Such a list of neighbor nodes can be represented by either an array or a linked list, and the array representation (Fig. 10(b)) is desirable to avoid the pointer-chasing problem which causes the CPU cache miss [36]. It is worth noting that, in addition to the list of neighbor nodes, graph algorithms may maintain some additional information in $Adj[v]$ to achieve efficiency. For example, for *PR*, in $Adj[v]$, it may maintain the node degree as well as the PageRank value for every node v to compute PraageRank iteratively, where the degree is represented by an integer type of 4 bytes and the PageRank value is represented by a double type of 8 bytes. In short, graph algorithms are implemented by the best data structure to achieve high efficiency in dealing with complex large graphs.

The proof sketch of Theorem 2.1: We prove it by showing that the special case when window size $w = 1$ is NP-hard. Consider the window size $w = 1$. To maximize $F(\phi)$ (Eq.(3)), for any v in G, we only consider the node u immediately appears before v in the permutation regarding $S(u, v)$. In order to do so, we construct an edge-weighted complete undirected graph G_w from G. Here, $V(G_w) = V(G)$, and there is an edge $(u, v) \in E(G_w)$ for any pair of nodes in $V(G_w)$. The edge-weight for an edge (u, v) in G_w is $S(u, v)$ computed for the two nodes in the original graph G. With G_w constructed, finding a graph ordering ϕ over G by maximizing $F(\phi)$ becomes a maximum traveling salesman problem without returning to the start node over G_w, which is to find the longest traveling salesman path over all nodes in G_w. The maximum traveling salesman problem is NP-hard [5, 28] and it does not change the computational complexity when it does not need to return to the start node. □

The proof sketch of Theorem 3.2: It requires $O(n)$ iterations to compute the permutation iteratively (line 3-10). In each iteration, it scans the set of the remaining nodes V_R (line 5-8) in $O(n)$, and computes the scores between the scanned node and the nodes in the window (line 6) in $O(w \cdot d_{max})$. The total time complexity of *GO* (Algorithm 1) is $O(w \cdot d_{max} \cdot n^2)$. □

The proof sketch of Theorem 3.3: Regarding time complexity, there are two terms. The first term is related to the updates (incKey and decKey). For every node $v \in G$, the updates occur only twice for v's neighbors and siblings, one for incKey and one for decKey, when v is to join/leave the window. Overall, computing neighboring relationships in G is $O(m)$, and computing sibling relationships in G is $O(\sum_v \sum_{u \in N_I(v)} d_O(u))$. Here, $O(\sum_v \sum_{u \in N_I(v)} d_O(u)) = O(\sum_{(u,v) \in E} d_O(u)) = O(\sum_u (d_O(u))^2)$. Thus, we have $O(m + \sum_v \sum_{u \in N_I(v)} d_O(u)) = O(m + \sum_u (d_O(u))^2) = O(\sum_u (d_O(u))^2)$, since $m < \sum_u (d_O(u))^2$. The second term is related to inserting v_{max} into P by popping it from $\mathcal{Q}$ in all n iterations. The time complexity for *GO-PQ* is $O(\mu \cdot \sum_{u \in V} (d_O(u))^2 + n \cdot \varrho)$. □

The proof sketch of Theorem 3.4: Let's see the time complexity got by Theorem 3.3. First, the time complexity of μ related to updates is $O(1)$, because both $\mathsf{decKey}(v_i)$ and $\mathsf{incKey}(v_i)$ are in $O(1)$, as Algorithm 3 and Algorithm 4. The $\mathsf{pop}()$ needs to adjust the linked list when $\mathsf{update(top)} < 0$. When $\mathsf{update(top)} < 0$, by $\overline{\mathsf{key}}(\mathsf{top}) \leftarrow \overline{\mathsf{key}}(\mathsf{top}) + \mathsf{update(top)}$, $\mathsf{update(top)}$ will be set to zero. Here the time complexity of $\mathsf{pop}()$ is $O(n)$ in the worst case. However, $\mathsf{pop}()$ is in the reverse of incKey which increases the key values. The total key value decreased by $\mathsf{pop}()$ cannot be larger than the total key values increased by incKey. In other words, the total time consumption on $\mathsf{pop}()$ for all nodes, related to $O(\varrho \cdot n)$, cannot be larger than the overall time consumption on incKey. Since the overall time consumption on incKey is at most as large as the number of neighboring and sibling relationship pairs, $O(m + \sum_{u \in V} (d_O(u))^2) = O(\sum_{u \in V} (d_O(u))^2)$, the time complexity of *GO-PQ* algorithm is in $O(\sum_{u \in V} (d_O(u))^2)$. □

The proof sketch of Theorem 3.5: We consider the sequence of key updates over v_i as a one-dimensional random walk, which starts at 0 and moves +1 ($\mathsf{incKey}(v_i)$) or -1 ($\mathsf{decKey}(v_i)$) with equal probability at each step. Let M_L be the max number that the random walk reaches after L steps. It is worth noting that M_L indicates the number of data structure adjustments in $\mathsf{incKey}(v_i)$ that satisfies the condition of $\mathsf{update}(v_i) > 0$. We have $M_L \geq 0$ and $\lambda_{v_i}^L \geq 0$. The step of moving +1 corresponds to incKey whereas the step of moving -1 corresponds to decKey. For every step of moving -1, it does not affect the max number of M_L. For every step of moving +1, M_L increases further only when the sum of the previous moving steps is larger than the current M_L, and adjusting the doubly linked list for v_i only occurs when M_L further

[7] software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

(a) The priority queue $\mathcal{Q}$

Node ID	prev	next	$\overline{\text{key}}$	update
v_1	$\emptyset$	$\emptyset$	3	0
v_2	$\emptyset$	$\emptyset$	4	0
v_3	v_7	v_8	2	-2
v_4	$\emptyset$	v_{10}	3	0
v_5	$\emptyset$	$\emptyset$	2	0
v_6	$\emptyset$	$\emptyset$	0	0
v_7	v_{11}	v_3	2	-2
v_8	v_3	v_{12}	2	-2
v_9	$\emptyset$	$\emptyset$	2	0
v_{10}	v_4	v_{11}	3	0
v_{11}	v_{10}	v_7	3	0
v_{12}	v_8	$\emptyset$	2	-2

(b) The head $\mathcal{Q}_h$

$\overline{\text{key}}$	head	end
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	v_7	v_{12}
3	v_4	v_{11}
4	$\emptyset$	$\emptyset$

Table 11: A snapshot of $\mathcal{Q}$

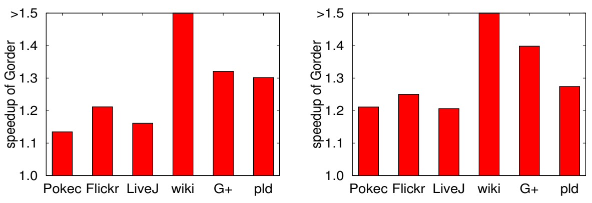
increases. Hence, we have $E(\lambda^L_{v_i}) = E(M_L)$. Based on the result in [26] (Page 481), we have $E(M_L) \sim \sqrt{\frac{2L}{\pi}}$ for a sufficiently large L, which completes the proof. □

Example-A: Consider the example G in Fig. 2. Supposed the *GO* algorithm has already inserted (v_6, v_9, v_5, v_1, v_2) into the array of P after 5 iterations and the window size $w = 3$. The current window consists of the last three nodes, (v_5, v_1, v_2). Here, v_2 is the newest node inserted into the window and will be removed from the window after 3 new nodes to be inserted in the following 3 iterations. In these 3 iterations in the *GO* algorithm, the locality score between v_2 and the remaining nodes that have not been inserted into P need to be repeatedly computed in line 5-6. On the other hand, the *GO* algorithm wastes time to compute it for v_3, v_7, v_8 and v_{11} in line 5-6, although they have no neighbor/sibling relationships with the nodes in the window.

Example-B: To reduce the cost for the repeated computation, the *GO-PQ* algorithm uses a data structure to record the locality scores between every remaining node and the nodes of the window, and only updates the locality scores of the necessary nodes. Since the data structure needs to efficiently find the node with the highest locality score in every iteration, it is implemented as a priority queue denoted as Q. Assume that *GO-PQ* has inserted (v_6, v_9, v_5, v_1, v_2) into the array P. In the next iteration in the *GO-PQ* algorithm, we insert v_4, which is the node having the largest score, into P, and perform incKey($\cdot$) operations for all neighbors and siblings of v_4 that have not been inserted yet, namely, v_{10} and v_{11}. Node v_5 is the oldest node in the current window. So the *GO-PQ* algorithm removes v_5 from the window and perform decKey($\cdot$) operations for all neighbors and siblings of v_5 that have not been inserted, namely, v_{10} and v_{11}. No computation is needed for the nodes v_3, v_7, v_8 and v_{11}. Finally, the resulting graph ordering is $\phi = (v_6, v_9, v_5, v_1, v_2, v_4, v_{10}, v_{11}, v_7, v_3, v_8, v_{12})$. □

Example-C: Supposed the *GO-PQ* algorithm has already inserted (v_6, v_9, v_5, v_1, v_2) into the array of P and the window of size $w = 3$ contains v_5, v_1, and v_2. Table 11 shows the snapshot of $\mathcal{Q}$, where the top points to v_4 which will be returned as the node v_{max} with the max true key value in this iteration. Since the update($\cdot$) for v_3, v_7, v_8, v_{12} is -2, at least, the next two incKey($\cdot$) for them do not need to adjust the doubly linked list. □

Additional Exp-A: Optimized *BFS* and multicore performance: The recent work [50] designs a specialized *BFS* algorithm aiming to improve the cache performance during graph traversal. Gorder can further improve the cache usage of the optimized graph traversal algorithms. Fig. 11(a) shows the speedup of Gorder when applied for the best optimized BFS algorithm of [50], BFS256. It utilizes SIMD instructions and 256-bit wide registers. Here, the



(a) BFS256 (b) BFS256 (4 threads)

Figure 11: Optimized BFS

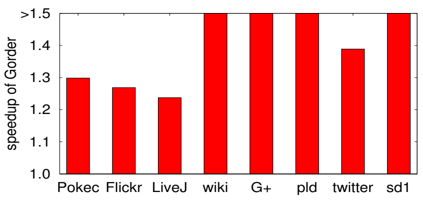


Figure 12: PageRank (4 threads)

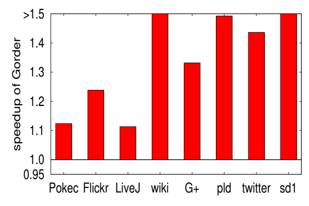
speedup of Gorder is the ratio of the running time by BFS256 using the original datasets over the running time by BFS256 in the datasets with Gorder. BFS256 cannot deal with the largest 2 dataset twitter and sd1-arc. The speedup of Gorder in BFS256 algorithm in Fig. 11(a) is close to that in basic *BFS* algorithm in Fig. 9(c). It shows that Gorder can be used to further support the algorithms like BFS256 to enhance the performance. Furthermore, we test the performance of Gorder under multicore environment and run the BFS256 and *PR* with 4 threads (the machine used has 4 physical core). Fig. 11(b) and Fig. 12 show the speedup results which are similar with those by the single thread versions. In Fig. 12, the speedup of Gorder in wikilink and sd1-arc is larger than 2.

The cache performance in a multicore environment is also the bottleneck of the graph computing, as shown in the single core graph computing in Fig. 1. This is because in a multicore environment, even though every core has its own L1/L2 cache, all cores share the same L3 cache which makes the cache resource even more limited to every core.

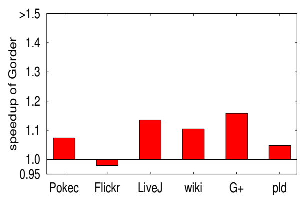
Additional Exp-B: Graph Systems: In real applications, graph data can be too large to be fit into one machine's memory. Many graph systems have been developed in recent years to support scalable graph computing. We conduct testing using two graph systems, GraphChi [33] and PowerGraph [25]. Here, GraphChi is a graph system developed on just a PC by storing the graph data in the external disk and loading the requested graph data into the memory when necessary. On the other hand, PowerGraph is the representatives of distributed graph system, which partitions the large graph into several partitions and store them separately in a cluster of machines. For testing GraphChi, we install GraphChi on the same Linux server to be used for all testing. For testing PowerGraph, we install them in a cluster of 32 machines which are equipped with Intel Core i3-2100 CPU and 8GB memory.

Fig. 13(a) shows the speedup of Gorder when running the PageRank algorithm in GraphChi. The speedup of Gorder in wikilink and sd1-arc is more than 1.8 times. Compared with Fig. 9(g), we can see that the improvement of cache performance by Gorder on GraphChi is significant. This is because GraphChi attempts to optimize the random access program and Gorder is able to reduce the random access problem greatly by storing the nodes with high locality.

Fig. 13(b) shows the speedup of Gorder when running the PageRank algorithm in PowerGraph. The speedup results for the two

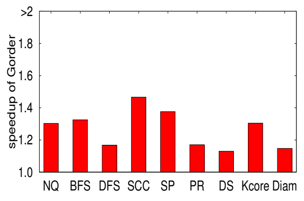


(a) GraphChi

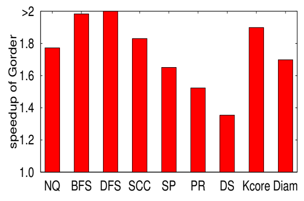


(b) PowerGraph

Figure 13: PageRank on Graph Systems



(a) Profiling in Flickr



(b) Profiling in sd1-arc

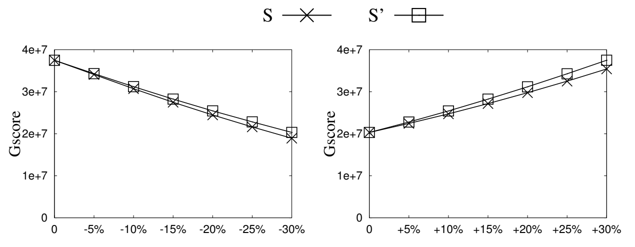
Figure 14: Profiling

largest graphs twitter and sd1-arc are missing, because the 32 machine cluster used cannot run the two datasets successfully due to the excessive memory usage on node replication. It is worth mentioning that PowerGraph has its own mechanism to partition the graph into the clusters of machines. This implies that nodes with close IDs, as done by Gorder, are not necessary to be allocated into the same machine. Nevertheless, Gorder provides a good initial ordering, which boosts the PowerGraph system to generate a good graph partitions and the graph algorithms running on the system can benefit from Gorder through the good partition. On the other hand, we find that the improvement by Gorder on Giraph [37] is less obvious. Here, Giraph is optimized for simple partition strategies and a high quality graph partition may not lead to the good performance. It is the implementation issues according to the empirical study of [47].

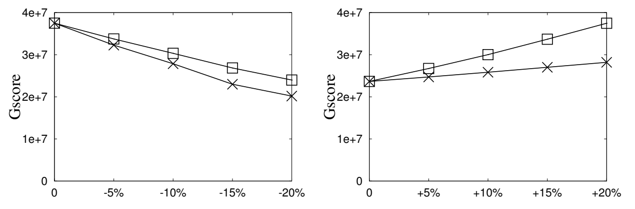
In general, the issue of applying Gorder to a distributed graph system is rather complicated. A key issue is whether the graph partitioning used in a distributed graph system can take the advantage of Gorder, among many factors that influence the performance of distributed graph system, such as load balancing. We plan to study it as our future work.

Additional Exp-C: Gorder **with Profiling:** Profiling tool can record the statistics information, including the frequency of function calls, the function workload and branch probability, during the previous runs of the graph algorithms. The compiler GCC can use these historical statistics information to optimize the code if the program is compiled with the GCC parameter -fprofile-generate and -fprofile-use.[8] We compile the program with the GCC profiling tools and run the graph algorithms in datasets with original order and Gorder to make the comparisons. The results are shown in Fig. 14. Profiling information does improve the graph computing time when running the optimized code in original datasets. Also profiling information can help graph algorithms running with our Gorder even faster. In Fig. 14, the speedup of Gorder for every graph algorithm is nearly the same comparing with the speedup result running without profiling information shown in Fig. 9. As the analysis of Exp-2 in Section 4, in Fig. 14, the speedup of Gorder in large dataset sd1-arc is much more significant than the results in small dataset Flickr because the cache can hold very limited information of large graph when running the graph algorithms.

Additional Exp-D: Graph Update: In practice, graphs need to be updated by insertion/deletion of nodes/edges. It is costly to recompute the Gorder for every update of the graph. Here, we show the performance of Gorder, without any recomputing, when nodes/edges are inserted/deleted. In other words, we do not update the Gorder for any edge insertion/deletion and node removal. The node IDs remain unchanged in these cases. For node insertion, we simply put the node to be inserted at the end of Gorder and assign it with a new node ID in order. For node deletion, we simply remove the nodes from the datasets. To test node-deletion and node-insertion, we do the following. We randomly delete nodes up to 20% of the total number of nodes in a graph dataset for node deletion, and then we insert the deleted nodes back to the graph incrementally for node insertion. To test edge-deletion and edge-insertion, we also randomly delete edges up to 30% of the total number of edges in a graph dataset for edge deletion, and then we insert the deleted edges back to the graph incrementally for edge insertion. In the figures, we show the difference with and without recomputing. Here, S denote the Gscore of the node ordering without recomputing, and S' denote the Gscore of the node ordering by recomputing after updates. Fig. 15 shows the results in Flickr. Fig. 15(a) shows the influence of Gscore after deleting 5%-30% edges randomly from the original graph. Fig. 15(b) shows the influence of Gscore after adding back 5%-30% edges randomly selected. Gorder is not sensitive to the edge insertion/deletion. Fig. 15(c) and Fig. 15(d) show the results of node insertion/deletion. Note that deleting $x\%$ nodes causes deleting about $2x\%$ adjacent edges. Inserting a large number the new nodes at the end has impacts on the performance. Based on the results, the recomputing timing can be determined.



(a) Delete Edges (b) Insert Edges



(c) Delete Nodes (d) Insert Nodes

Figure 15: Graph Updates in Flickr

[8]gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html