International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:15, No:7, 2021

Speedup Breadth-First Search by Graph Ordering

Qiuyi Lyu, Bin Gong

Abstract—Breadth-First Search (BFS) is a core graph algorithm that is widely used for graph analysis. As it is frequently used in many graph applications, improving the BFS performance is essential. In this paper, we present a graph ordering method that could reorder the graph nodes to achieve better data locality, thus, improving the BFS performance. Our method is based on an observation that the sibling relationships will dominate the cache access pattern during the BFS traversal. Therefore, we propose a frequency-based model to construct the graph order. First, we optimize the graph order according to the nodes' visit frequency. Nodes with high visit frequency will be processed in priority. Second, we try to maximize the child nodes' overlap layer by layer. As it is proved to be NP-hard, we propose a heuristic method that could greatly reduce the preprocessing overheads. We conduct extensive experiments on 16 real-world datasets. The result shows that our method could achieve comparable performance with the state-of-the-art methods while the graph ordering overheads are only about 1/15.

Keywords—Breadth-first search, BFS, graph ordering, graph algorithm.

I. Introduction

GRAPH representation is widely used for various scenarios [15], [26], [28], [29], [10], [9] such as social networks, citation networks, and computer networks. As graphs are frequently used in the real world, graph analysis is becoming more and more important. Among all the graph analysis methods, the Breadth-first search is one of the core methods that draw much attention.

A plethora of works have focused on improving the BFS performance. The most popular methods [2], [5], [11], [19] try to parallel the BFS procedure to speed up the traversal performance. However, existing works [3], [30] have shown that the cache stall costs great overheads in the BFS process. Only engage more CPUs in the computation could hardly improve the BFS performance. Thus, how to reduce the cache misses is the core problem.

An intuitive idea to reduce the cache misses is to optimize the adjacent list. We show an example in Fig. 1. Fig. 1(a) is an example graph that we will use throughout the paper and Fig. 1(b) is the corresponding adjacent list. We optimize the adjacent list by sorting the edge arrays and show it in Fig. 1(c). In this scenario, all the edge arrays are sorted according to their destinations. As the graph nodes are kept in memory according to their index, the sorting achieves better data locality. For instance, if we perform a BFS from node 5, for the adjacent list in Fig. 1(b), we will visit nodes 4, 8, and 3 one by one. Although node 4 and node 3 are close to each other, they show a terrible locality in data access. However, if we use the adjacent list in Fig. 1(c) in which the edge arrays are sorted according to the node index, node 5's child nodes will be visited in the order of 3, 4, and 8. In this scenario, the node 4 will be visited after node 3. As they are very likely to be in the same cache line, better data locality could be achieved. Nevertheless, Malicevic et al. [27] prove that only optimizing the adjacent list could not reduce the cache misses. The real-world graph is huge that the destination nodes in the adjacent list are very likely to be kept far away from each other. No matter how we order the adjacent list, the destination nodes could hardly be located in the same cache line. Thus, graph ordering methods are proposed.

There have been extensive researches [30], [22], [8], [25], [23] focusing on developing graph ordering techniques. However, although they could significantly speed up the graph algorithms, they are either of low efficiency or incur extremely high overheads. For example, Gorder [30] tries to cluster the nodes that are frequently accessed together in the local area to achieve better cache utilization. Nevertheless, for large graphs with tens of millions of edges like LiveJournal, their preprocessing time could be 39.54 seconds, while the average BFS time is only about 0.49s. Compared with the performance gains, the large preprocessing overheads are unacceptable. Therefore, lightweight graph ordering methods are needed.

Several lightweight graph ordering methods [31], [4], [6] have been proposed recently. Different from Gorder, they concentrate on end-to-end performance improvements. That is, their preprocessing overheads could be easily amortized even if the graph application is just executed a few times. However, [6] proves that the BFS procedure could hardly benefit from the existing lightweight graph ordering methods. This is because [6] BFS will process a small fraction of edges per iteration, leading to limited reuse of cache line.

We present the Maximum Overlap (MO) method in this paper. Our method is based on an observation that the sibling nodes' access dominates the cache access pattern in the BFS procedure. Thus, we try to reorder the graph nodes according to the sibling relationships. The most straightforward method is to order the graph in the BFS visit order. An example is shown in Fig. 1(d). We assume that we want to perform a BFS from node 1. As node 4 is next to node 5, node 5 and node 4 will be sequentially visited. After that, node 5 try to visit its child nodes in the order 4, 3, and 8. Again, as they are in the adjacent memory position, the number of cache misses will be significantly reduced. However, the method is inefficient as it ignores the node visit frequency and the child nodes overlap. Thus, we present our hierarchical graph order method that could reorder the graph nodes layer by layer. First, we propose a frequency-based model that could optimize the graph order according to the node visit frequency. Then, we try to maximize the child nodes overlap to improve the data locality further. The experiments prove that our method is very efficient in graph order construction and could achieve

Qiuyi Lyu is with Shandong University, China (e-mail: anna_shmaglit@yahoo.com).

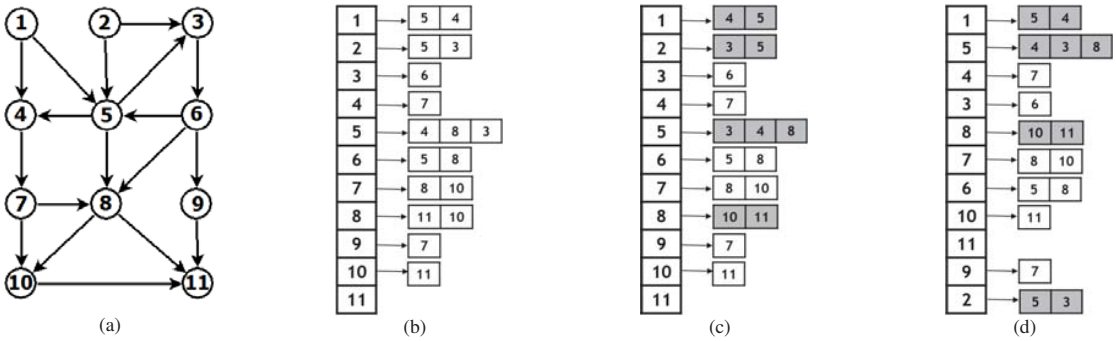

Fig. 1 Fig. 1(a) is an example graph that will be used all over the paper. Fig. 1(b) is the corresponding adjacent list. Fig. 1(c) is the optimized adjacent list. Fig. 1(d) is the adjacent list optimized by the graph ordering method

comparable performance with state-of-the-art methods.

Hereby, we summarize the contributions as the following:

- First, we propose a frequency graph ordering model that focuses on the speedup of BFS traversal. Different from the existing methods, we consider the node visit frequency.
- Second, we propose a hierarchical node cluster method that could maximize the child node overlap layer by layer. Moreover, the method is readily comprehensible and easy to implement.
- Finally, we compare our method against the state-of-the-art approach, using 16 large real-world datasets. We confirm that our method could make a comparable performance against the state-of-the-art methods while the preprocessing overheads are only about 1/15.

The rest of the paper is organized as follows: Section II shows the related works. Section V proposes the MO method. The experiment is reported in Section V. Section VI concludes the paper.

II. Related Work

Graph ordering has been studied for decades. At the very beginning, graph ordering [14], [18] was proposed for reducing the bandwidth in matrix computation. Cuthill-McKee [14] and Reverse CuthillMcKee (RCM) [18] methods reorder the graph nodes into BFS visit order, thus the matrix fill will be reduced which in turn reducing the space and time required to perform the matrix computation. As RCM shows a great importance in paralleled computing, the paralleled RCM [16], [20] are also studied.

Recent years, graph ordering [31], [6], [22], [4], [30] is popular for speedup graph analysis. Wei et al. present Gorder [30], a general approach that tries to improve graph memory access by reordering the graph nodes. They optimize the cache utilization by greedy strategy. When deciding the node in the next position, they always try to find out the node with the most relations with the front nodes. Here, the relation means the neighbor relationships and the sibling relationships. However, as we aforementioned, Gorder will incur significant overheads in the graph ordering procedure. The processing time is unacceptable for large graphs. Thus, lightweight graph ordering methods are studied.

Zhang et al. propose Frequency-Based Clustering [31] which focuses on the graph ordering in power-law graphs. They find that the hub nodes with numerous edges are much more likely to be accessed than the other nodes. Thus, they reorder the graph nodes according to the nodes' degree such that the nodes with large degrees will be clustered together. It is generally a sorting process. Therefore, it shows high efficiency in graph order construction. Hub Clustering [6] is a variation of Frequency-Based Clustering. Unlike Frequency-Based Clustering, it does not guarantee that the nodes are ordered in descending order of degree. Thus, it achieves a reduced speedup compared with Frequency-Based Clustering. As both the methods are lightweight graph ordering methods, they could achieve end-to-end speedup for graph analysis. Nevertheless, their performance lift may be very limited in some scenarios.

Community based graph ordering methods [17], [25], [12], [8], [4] are also studied. However, most of them [25], [12], [8] focus on the compression of social networks. Arai et al. [4] present Rabbit, which is a community-based graph ordering method that tries to speed up the graph analysis. Their approach is based on an observation that a community has dense inner-edges, and hence, during graph analysis, a node in a community involves frequent accesses to other nodes in the community [4]. They propose a hierarchical community-based ordering method that tries to improve locality by co-locating nodes in each community and within each subordinate inner community recursively [4]. Nevertheless, their approach tends to be effective only in graphs with community structures.

Lakhotia et al. [22] propose the Block Reordering approach. In addition to the spatial locality, they consider temporal locality. They define a notion of dynamically matching memory access patterns with cache contents in a way that jointly maximizes both cache data reuse and cache line utilization [22]. They prove that Block Reordering can achieve up to 2.3x speedup over the original graph order.

III. Maximum Overlap Method

Given a directed graph $G(V, E)$, V is the nodes set, and E is the edges set. The number of nodes and edges in G is

denoted as $n = |V|$ and $m = |E|$. Given a node u, we denote the in-neighbor (resp. out-neighbour) of u as $N_{in}(u)$ (resp. $N_{out}(u)$). The in-degree and out-degree of node u are denoted as $d_{in}(u)$=$|N_{in}(u)|$ and $d_{out}(u)$=$|N_{out}(u)|$. For an edge from node u to node v, we denote it as (u, v). Two nodes are defined as sibling nodes if they share a common in-neighbor.

The key idea for speedup the BFS traversal is to access the sibling node as much as possible sequentially. That means, in addition to the techniques we proposed in this paper, we will also sort the adjacent list like the example shown in Fig. 1(c). In order to measure the benefit that we could achieve from the cluster of sibling nodes, we define a score function.

$$Score(v_i, v_{i+1}, u) = \begin{cases} 1 & \text{if } v_i \text{ and } v_{i+1} \text{ share a common father node } u \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Here, v_i and v_{i+1} are two graph nodes. The subscript i indicates the nodes' index in the new graph order. For a specific node u in the graph, the score it could acquire is:

$$F(u) = \sum_{i=0}^{n} Score(v_i, v_{i+1}, u) \quad (2)$$

Thus, for the new graph order, the total score we could achieve is:

$$F(\pi(V)) = \sum_{u \in V} \sum_{i=0}^{n} Score(v_i, v_{i+1}, u) \quad (3)$$

Here, $\pi(V)$ is a permutation function that maps every node in V to a natural number. Equation (3) will measure how well the sibling nodes are clustered in the new graph order.

The model is readily comprehensible. However, the node visit frequency is ignored. It is well known that nodes in the graph have different visit frequencies. Some nodes are frequently visited, and some other nodes are seldom visited. For example, in Fig. 1(a), on the one hand, node 4 and node 5 are the child nodes of node 1. on the other hand, node 5, 8 and 9 are the child nodes of node 6. We could either put node 5 together with node 4 or together with nodes 8 and 9. If we consider the nodes' visit frequency, we could find that nodes 4 and 5 will be accessed together only when node 1 is visited. However, the visit of nodes 6 and 3 will both incur the co-access of nodes 5, 8, and 9. Thus, co-locate node 5, 8 and 9 are reasonable. To achieve better graph order performance, we define a new score function that considers the nodes' visit frequency. For node u in the graph, the score it could achieve is:

$$F(u) = \rho_u * \sum_{i=0}^{n} Score(v_i, v_{i+1}, u) \quad (4)$$

where ρ_u is the frequency that node u is visited or, in other words, the frequency that node u's child nodes are co-accessed.

Obviously, the more the sequential memory accesses are executed, the better cache locality we could achieve. Our problem is to find an optimal permutation of the graph nodes that could maximize the sum of all the nodes' scores. We show the problem definition here.

Problem 1. Given a graph $G(V, E)$, we find an optimal permutation $\pi : V \rightarrow N$ that maximizing the score of $F(\pi(V))$,

$$F(\pi(V)) = \sum_{u \in V} \rho_u * \sum_{i=0}^{n} Score(v_i, v_{i+1}, u) \quad (5)$$

Theorem 1. Maximizing $F(\pi(V))$ to achieve the optimal permutation $\pi(V)$ for directed graph $G(V, E)$ is NP-hard.

Proof: First, we assume that all the nodes will be visited with the same frequency. To simplify the proof, we assume $\rho = 1$ for all the nodes in the graph. Thus, the score $F(\pi(V))$ will only correlate to how many sequential accesses could be executed with the specific permutation. In some cases, some graph nodes may share a bunch of child nodes. Co-accessing the shared child nodes will result in a higher score as they could provide more sequential node access for the common father nodes. However, in our proof, we assume that such sharing does not exist in the graph. Thus, the sequential access of the specific nodes will only benefit their common father node. In this scenario, as there are only n nodes in the graph, at most $n-1$ sequential memory accesses could be executed. For node u in the graph, there are $d_{out}(u)$ child nodes and could perform at most $d_{out}(u)-1$ sequential memory access. The fewer father nodes we used to cover the $n-1$ sequential access, the more sequential accesses could be performed. Thus, we should try to use the least number of father nodes to cover the $n-1$ sequential access. Our problem could be reduced to the minimum set cover problem, which is NP-hard. ■

As we have proved, Problem 1 is NP-hard. It is difficult to find the optimal solutions. Thus, we propose to use a heuristic method to maximize the score function. Our method is based on two observations: 1) The hub nodes will make a big difference in the total score. We could infer from (4) that, for a given node u, the node visit frequency ρ_u and the corresponding sequence score $\sum_{i=0}^{n} Score(v_i, v_{i+1}, u)$ will greatly affect the total score. On the one hand, as the hub nodes usually get a high in-degree, they are more likely to be visited. That means their visit frequency is high. On the other hand, the hub nodes usually get a large out-degree which tends to achieve a high score in $\sum_{i=0}^{n} Score(v_i, v_{i+1}, u)$. Consider the large difference in nodes' visit frequency; the hub nodes may get a huge $F(\cdot)$ score, which will overwhelm the nodes with a few neighbors. Thus, prioritize the hub nodes is reasonable. 2) Maximum the child nodes overlap will improve the score. Until now, we only consider the scenario that there are no child nodes overlap in the graph. That is, the sibling nodes will only share one father node. However, in practice, it is quite common that the sibling nodes share several father nodes. For example, if we insert an edge $(6, 3)$ into the graph, nodes 3 and 8 will share two common father nodes: node 5 and 6. In this scenario, if we co-locate node 3 and 8 together, we could achieve a higher score from both $F(5)$ and $F(6)$. Thus, maximum the child nodes overlap is another problem we try to solve.

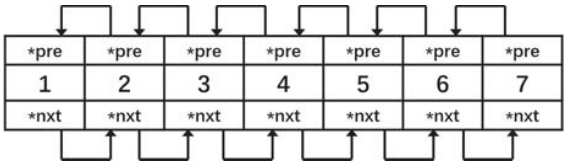

Fig. 2 The data structure used in our algorithm

To achieve a good performance, our method will apply a greedy strategy. We will always try to process the nodes with high visit frequency and large $F(\cdot)$ score. However, as the node visit frequency is hard to acquire, we will use the nodes' in-degree to approximate the value. Thus, (5) will be transformed into an in-degree based equation:

$$F(\pi(V)) = \sum_{u \in V} d_{in}(u) * \sum_{i=0}^{n} Score(v_i, v_{i+1}, u) \qquad (6)$$

It needs to be noted that, due to the power-law property, the nodes' in-degree maybe have a more significant difference than the node visit frequency in some social networks. Equation (6) tends to stress more on the hub nodes.

IV. Implementation

We have proposed the model in the previous section. However, due to its NP-hard nature, it is not easy to find out the optimal solution. Thus, in this section, we focus on an efficient heuristic algorithm.

As we aforementioned, we adopt a greedy strategy in our method. Our method aims to modify the memory location of the graph nodes to maximize the child nodes' overlap. However, frequently moving the memory block is costly. Thus, we use a link list to simulate the data moving process. The data structure is shown in Fig. 2. The graph is stored in an adjacent list. Every node will use two pointers to locate its position in the link list. During the processing, our method will continuously re-link the link list. Finally, we will get a new link list in which the index indicates the new graph order.

The method is shown in Algorithm 1. We call it MO (maximum overlap) algorithm because it tries to overlap the children nodes as many as possible. We will process the hub nodes in priority as they show great importance in our model. Thus, we first sort the nodes according to the nodes' in-degree and out-degree product in descending order in line 5. From line 7 to line 10 we process the first hub nodes with the most significant product of the in-degree and out-degree. Its child nodes will be added to the link list and be marked as visited. Then, we set their group(gp) property as the index of its father node.

The other nodes will be handled from line 11 to 30. For every unvisited node, we will inspect its child nodes one by one. If the child node is unvisited, that means the node has not been added to the link list. We will add it to the end of the link list and set its visit/group(gp) property (from line 13 to 19). Otherwise, the node has been visited before, which means it is an overlapped child node that has been visited from other nodes. We will push it into the "overlap" array (line 19).

Algorithm 1 Maximum Overlap Algorithm

```
1:  procedure MO(G(V, E))
2:      //sort all the nodes in descending order
3:      //according to the product of nodes's
4:      //in-degree and out-degree
5:      ord = sort(G(V, E))
6:      groupCnt = 1
7:      for node x ∈ N_out(ord[0]) do
8:          add x to the end of the link list
9:          x.visit = true
10:         x.gp = ord[0]
11:     for int i from 1 to n do
12:         for node y ∈ N_out(ord[i]) do
13:             if y.visit = false then
14:                 y.visit = true
15:                 y.gp = ord[i]
16:                 add y to the end of the link list
17:             else
18:                 //add y into the array overlap
19:                 overlap.add(y)
20:         //sort the visited according to the gp property
21:         sort(overlap)
22:         if overlap.size > threshold then
23:             for int j from 0 to overlap.size do
24:                 if overlap[j].gp=overlap[j+1].gp then
25:                     re-link the link list
26:                     link overlap[j+1] next to overlap[j]
27:                     overlap[j].gp = |v|+groupCnt
28:                     overlap[j+1].gp = |v|+groupCnt
29:                 else
30:                     groupCnt++
31:     //add the unvisited nodes into the link list
32:     for node z ∈ V do
33:         if z.visit=false then
34:             add z to the end of the link list
```

The most important technique we used in our method is sorting the "overlap" array according to nodes' gp property (line 21). In this way, nodes in the same group will be clustered together. The next step is to re-link the link list. However, we use a predefined variable "threshold" to restrict the process (line 22). If there are few nodes in the array overlap, we will not execute the re-link process as there may be few child nodes overlapped.

From lines 23 to 30 we re-link the link list according to the "overlap" array. For the nodes in the array, we will check its group(gp) property (line 24). If the currently visited node is in the same group as the next node, they will be linked in an adjacent position. After that, they will be allocated with a new group number (lines 27 and line 28). Finally, we will add the unvisited nodes to the end of the link list (line 32 to 34).

It needs to be noted that our algorithm could continuously overlap the already visited child nodes. However, it will not break the sibling relationship that has been constructed in adjacent positions. We show an example in Fig. 3. Assume that node v_1 is the largest hub node. Thus, the nodes in the graph

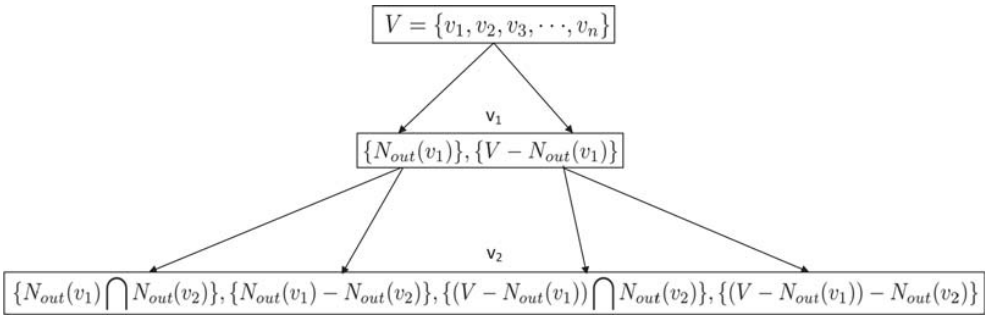

Fig. 3 Graph nodes divided by node v_1 and v_2

will be divided by node v_1 at first. They are split into two set: $\{N_{out}(v_1)\}$ and $\{V - N_{out}(v_1)\}$. Nodes in $\{N_{out}(v_1)\}$ will be added to the link list and marked as visited. Assume node v_2 is the next hub node that needs to be processed. We try to find out the child nodes that have been added to the link list before. After sorting the "overlap" array (line 21), the nodes with group number v_1 will be linked together, which is the set $\{N_{out}(v_1) \bigcap N_{out}(v_2)\}$. The other child nodes of v_1 will be located just behind them, which is $\{N_{out}(v_1) - N_{out}(v_2)\}$. Our adjustment in the link list will not break the sequential access of node v_1's child nodes.

Discussion: As we have stated, our method tries to maximize the child nodes overlap. That means our method tends to be more efficient in dense graphs or graphs with community structures. For social networks, nodes in the same community are more likely to share common child nodes. For dense graphs, as there are more edges in the graph, the nodes in the graph tend to share more child nodes. On the contrary, in sparse graphs, the performance of our method will degrade. On the one hand, in the sparse graphs, every node may have only a few edges, the nodes are less likely to share common child nodes. On the other hand, our method is a hierarchical technique that the child nodes will be split layer by layer. In sparse graphs, the hierarchical technique will be inefficient.

V. Experiment

In this section, we compare the MO method with the state-of-the-art approaches. First, we will introduce the experimental setup in Section V-A. Second, we show the graph ordering overheads in Section V-B. Third, we conduct a comparison in terms of BFS running time in Section V-C.

A. Experimental Setup

We conduct our experiments on a server with an Intel Xeon CPU E5-2680 v3 2.50GHz, 64GB RAM, 32KB Level 1 cache, 256KB Level 2 cache, and 30MB Level 3 cache. We compare our maximum overlap(MO) method with several state-of-the-art approaches:

- Original: This is the original graph that we acquire from the web.
- Gorder [30]: The state-of-the-art approach proposes a locality metric(Gscore) to measure the graph ordering efficiency. It maximizes the Gscore by a proposed greed strategy.
- Rabbit [4]: A lightweight graph ordering approach that focuses on the graph's community property. Nodes in the same community tend to be clustered together.
- HC [6]: The HubCluster method that will reorder the nodes according to the nodes' in-degree and out-degree.
- MO: The maximum overlap method proposed in this paper.

We will use the classic BFS algorithm [13] to evaluate the performance.

The experiments are conducted on 16 real-world datasets. We show the statistics of the dataset in Table I. All the datasets are collected from SNAP [24], KONECT [21], and LAW [1]. In our experiment, they are transferred into the compressed sparse row (CSR) format.

TABLE I
Dataset Statistics

Dataset	$\|V\|$	$\|E\|$	d_{avg}
LJ	4,847,571	68,993,773	14.23
Web	875,713	5,105,039	5.83
Email	265,214	420,045	1.58
Wiki	2,394,385	5,021,410	2.09
Pokec	1,632,803	30,622,564	18.75
BS	685,231	7,600,595	11.09
Twitter	2,881,151	6,439,178	2.23
Reddit	2,628,904	57,493,332	21.86
Amazon	400,727	3,200,440	16.79
Wiki-t	1,140,149	7,833,140	6.87
Youtube	1,138,499	4,942,297	8.68
CS	384,413	1,751,463	4.06
DB	3,966,924	13,820,853	6.96
Trec	1,601,787	8,063,026	10.06
Arabic	22,744,080	639,999,458	28.14
Uk	18,520,486	298,113,762	16.09

UK and Arabic [7] are two large web datasets with hundreds of million edges. LJ, Youtube, Reddit, and Pokec are power-law graphs collected from social networking sites. CS is a citation network that each edge indicate that an author has (co-)published a given publication. It is collected from the CiteSeer digital library. Wiki and Wiki-t are collected from Wikipedia, and Wiki-t is a wiki talk communication network. BS is a web graph of Berkeley and Stanford. Web is a web graph dataset from Google. Every node in the graph represents a web page, and the directed edges represent hyperlinks between them. Trec is a web graph collected from the TREC conference, and Amazon is an Amazon product co-purchasing network from June 1, 2003. DB is a complete

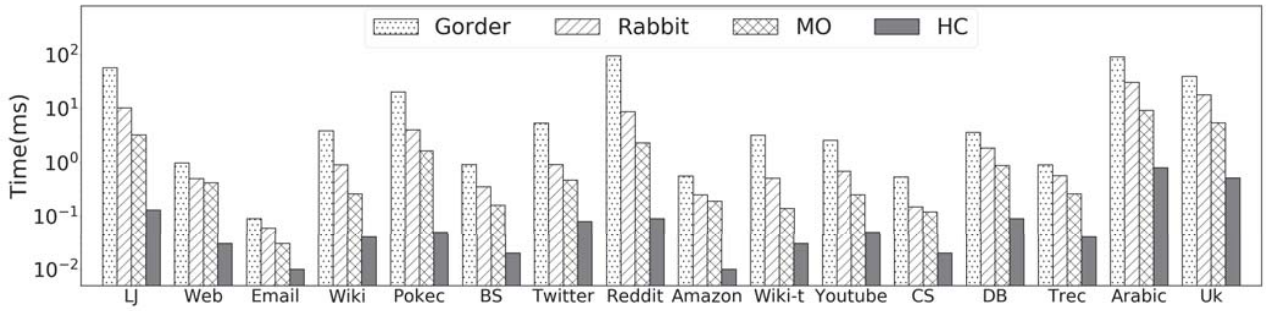

Fig. 4 Graph order construction time

DBpedia network. The data is extracted from version 3.6 of DBpedia.

B. Order Construction

We show the graph order construction time for all the methods in Fig. 4. Gorder will cost the highest overheads in graph order construction. As we have stated, Gorder is based on a greedy strategy, and it approximates the traveling salesman problem(TSP). Their overheads are very high due to their high time complexity. After they optimize the construction process by priority queue, their time complexity could be $O(\sum_{u \in V}(d_{out}(u))^2)$. Among all the methods, HC could achieve the most minor graph order construction overheads. According to their method, HC will only sort the graph nodes based on their out-degree numbers. The time complexity depends on the sorting approach they used. Benefit from their simple processing method, HC could construct all the graph orders in less than 1 second. Rabbit is the only graph ordering method that could be paralleled. The author claim that Rabbit's time complexity is roughly in proportion to m. For a fair comparison, we use one thread to construct the graph order. For most of the datasets, they could construct the graph order in less than 10 seconds. MO could achieve a suboptimal performance. As we also need to sort the nodes, the time complexity is related to the sorting algorithm we use. However, we want to highlight that MO could be orders of magnitude faster than Gorder and Rabbit in graph order construction.

C. BFS Time

For all the approaches, we evaluate the BFS running time in Fig. 5. We perform 100 BFS traversals from randomly generated source nodes, and report the average running time of 10 repeats.

HC shows the worst performance in our experiment. For all the 16 datasets, HC can only surpass the original in 6 of them. In most cases, HC will degrade the BFS performance. This is because the original graph obtained from the web may have a certain degree of locality in nature. The reorder procedure may break its original locality. The experiment proves that, although HC method could efficiently construct the graph order, its performance is not satisfactory.

Compared with HC, Rabbit achieves a better performance. It could speed up 8 datasets, and the improvement is more significant than HC. Nevertheless, as we aforementioned, it could only speed up the datasets that have community structure. For example, in LJ, Email, Pokec, Twitter, and Wiki-t, Rabbit shows notable improvement. All these datasets are social networks or communication networks that have community structures. On the contrary, when applied to other scenarios with no communities, Rabbit could not improve the BFS performance. For example, in BS, CS, Amazon, and DBpedia, Rabbit could achieve no improvement.

For all the baselines, Gorder acquire the best overall performance. Therefore, in the remainder of this paper, we will only compare our method with Gorder.

LJ, Youtube, Reddit, and Pokec are social networks. However, Gorder and MO methods show different performances in these datasets. In LJ, Reddit, and Pokec, both Gorder and MO could speed up the BFS procedure. On average, Gorder could achieve a speedup of 16%, while MO could achieve 21%. In Youtube, Gorder and MO could not achieve any improvement. As Youtube's average degree is 8.68, it is much sparser than the other three datasets, which means nodes in the graph are less likely to share common child nodes. Thus, maximizing the child nodes overlap is invalid. We want to highlight that, for Gorder, construct the graph order for LJ, Reddit and Pokec will incur significant overheads (56.91s, 19.98s and 94.08s, respectively). Our method can reduce 95% of the graph order construction time and the BFS performance could be even better.

Email and Wiki-t are communication networks. As we aforementioned, our method is good at handling graphs with community structures. Thus, compared with the original graph, we could achieve an average improvement of 33%. Nonetheless, due to the sparse nature of Wiki-t, our method could only achieve a suboptimal performance. Similar tendency could also be achieved in Web, BS, Twitter, DB and Arabic. In all these datasets, Gorder continuously outperform MO.

Arabic and UK are two large datasets with several hundred million edges. We notice that all the methods could hardly speed up their BFS performance. Gorder could speedup Arabic by about 11%. However, it would worsen the BFS performance in UK by about 3%. MO could acquire a very similar

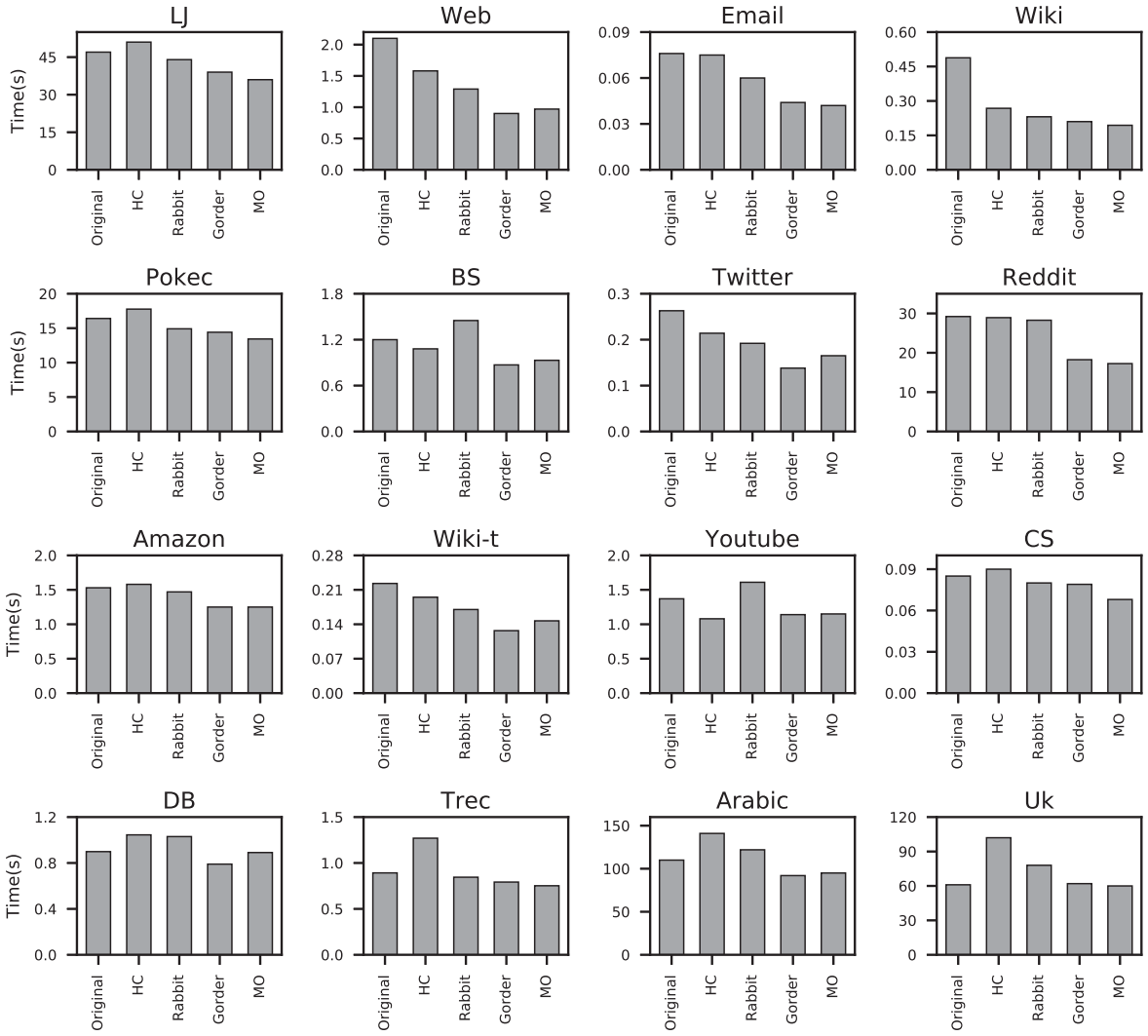

Fig. 5 BFS traversal time

performance, and the performance lift is also very limited. It is because the graph is huge, and Arabic has the highest average degree. Therefore, the BFS visit will traversal a large number of nodes that the cache could not contain. Although we optimize the graph order, the traversal will still incur a large number of random memory access.

Our experiments show that MO method could achieve comparable performance with the state-of-the-art method. For all the 16 datasets, MO outperforms Gorder in 8 of them. Generally, MO method could acquire the best performance in social networks and communication networks like LJ, Pokec, Reddit and Email. However, in sparse graphs like DB, Wiki-t, and Twitter, Gorder is better than MO method. As the graphs are sparse, every node will only connect a few child nodes. Thus it is hard to overlap their child nodes. Our MO method will be degraded in this scenario. It needs to be noted that, even in these datasets, MO could still acquire a suboptimal performance. Although Gorder could outperform MO method in some scenarios, MO will significantly reduce the graph order construction overheads. Compared with Gorder, MO's graph order construction overheads are only about 1/15.

VI. Conclusion

In this paper, we present MO method to speed up the BFS performance. Different from the existing works, our method focuses on sibling relationships. First, we proposed a frequency-based model. Then, we proved that maximize the score function to achieve the optimal graph order is NP-hard. Thus, we proposed a heuristic method based on a greedy strategy. Generally, we first clustered the sibling nodes in adjacent memory addresses. Then, we maximized the child nodes' overlap layer by layer. We conducted experiments on 16 real-world graphs. Our experiments show that MO method could achieve comparable performance with the

state-of-the-art approaches while the graph order construction speed is orders of magnitude faster. Our future work will focus on speeding up the DFS-based methods and the graph order maintaining in dynamic graphs.

REFERENCES

[1] LAW: The laboratory for web algorithmics. http://http://law.di.unimi.it.
[2] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.
[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number CONF, pages 266–277, 1999.
[4] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2016.
[5] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 523–530. IEEE, 2006.
[6] V. Balaji and B. Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.
[7] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.
[8] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, pages 587–596, 2011.
[9] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: who is in your small world. *arXiv preprint arXiv:1208.0090*, 2012.
[10] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996.
[11] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 378–389. IEEE, 2012.
[12] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, 2009.
[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
[14] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172, 1969.
[15] S. Dolev, Y. Elovici, and R. Puzis. Routing betweenness centrality. *Journal of the ACM (JACM)*, 57(4):1–27, 2010.
[16] M. Frasca, K. Madduri, and P. Raghavan. Numa-aware graph mining techniques for performance and energy efficiency. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.
[17] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.
[18] J. A. George. Computer implementation of the finite element method. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1971.
[19] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 78–88. IEEE, 2011.
[20] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzaran, and K. Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932. IEEE, 2014.
[21] J. Kunegis. The koblenz network collection. http://konect.uni-koblenz.de, 2017.
[22] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 273–282. IEEE, 2017.
[23] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236. IEEE, 2013.
[24] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[25] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, 2014.
[26] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.
[27] J. Malicevic, B. Lepers, and W. Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 631–643, 2017.
[28] A. McLaughlin and D. A. Bader. Scalable and high performance betweenness centrality on the gpu. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 572–583. IEEE, 2014.
[29] V. Ufimtsev and S. Bhowmick. Application of group testing in identifying high betweenness centrality vertices in complex networks. In *Eleventh Workshop on Mining and Learning with Graphs*, pages 1–8, 2013.
[30] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828, 2016.
[31] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.