Accelerating Depth-First Traversal by Graph Ordering

Qiuyi Lyu
qiuyilv@gmail.com
Shandong University

Mo Sha
sham@comp.nus.edu.sg
National University of Singapore

Bin Gong
gb@sdu.edu.cn
Shandong University

Kuangda Lyu
kdlv@stu.xidian.edu.cn
Xidian University

ABSTRACT

Cache efficiency is an important factor in the performance of graph processing due to the irregular memory access patterns caused by the sparse nature of graphs. To increase the cache hit rate, prior studies proposed a variety of preprocessing approaches based on the reordering, which permutes the vertexes' labels to improve the locality of graph structures. However, the locality enhancement of existing reordering approaches does not bring much performance benefit in depth-first traversal, which is widely adopted in a majority of graph processing applications. Furthermore, the state-of-the-art reordering approach suffers from an obvious overhead on preprocessing which will greatly limit the application of their approach. In this paper, we propose SeqDFS, a depth-first graph traversal method that optimizes the cache efficiency by adjusting the order of vertexes visited and can be further extended to dynamic scenarios. We conduct extensive experiments on 16 real-world datasets and 3 representative depth-first graph applications, of which the results show that our proposal achieves a significant speed-up on both directed and undirected graphs.

CCS CONCEPTS

• **Theory of computation** → *Graph algorithms analysis.*

KEYWORDS

depth-first search, DFS, DFS tree, graph ordering

ACM Reference Format:
Qiuyi Lyu, Mo Sha, Bin Gong, and Kuangda Lyu. 2021. Accelerating Depth-First Traversal by Graph Ordering. In *33rd International Conference on Scientific and Statistical Database Management (SSDBM 2021), July 6–7, 2021, Tampa, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468791.3468796

1 INTRODUCTION

Graph processing is one of the most effective tools to model complex problems in a structured manner and mining corresponding underlying knowledge. Generally, graph processing is data-intensive, indicating that the memory access efficiency is a key factor of the overall performance. However, during graph processing, the locality of the memory access patterns tends to be low. This means that, the data adjacently accessed is usually not physically closely located in the memory, which significantly degrades the cache hit rate. Prior studies [10] point out that the cache miss latency takes a large part of the execution time in graph processing. Hence, improving the locality of the memory access patterns during graph processing will effectively improve the computational efficiency.

For most cases, graph processing is conducted by visiting vertexes under some certain patterns dependent on the applications. Among them, depth-first traversal is an important access pattern, which is a fundamental building block that is adopted in many graph applications, e.g., bridge detection [20], topological sorting [19], subgraph isomorphisim[7], reachability query [17, 23, 25, 28], strongly connected component detection [9, 16, 18], cycle detection [21], and biconnected component detection [11], etc.

For this reason, there is a kind of preprocessing approach based on the vertex reordering [2, 3, 10, 13, 27], which relabels graph vertexes based on different assumptions. For example, HubCluster [3] makes hubs, i.e., vertexes with a higher degree, to be close with each other in terms of their labels to increase the reuse probability of the cache page; Gorder [10] calculates a permutation based on the Gscore, which maximizes the number of edges of which the label difference between in-degree and out-degree does not exceed a predefined window distance, in order to reduce cache replacement when traversing along edges.

However, the permutations of the existing reordering approaches are generated by their optimization targets which increase the locality of vertex labels based on the proximity of the graph topology, rather than based on memory access behaviors. When graph processing is conducted in a depth-first manner, as the depth increases, vertex visits quickly cross different neighboring areas of the graph structure, and hence, the optimization assumptions of existing approaches may not lead to an effective improvement of the cache hit rate in such scenarios. Furthermore, on one hand, preprocessing-based reordering approaches are time-consuming. On the other hand, it is difficult to maintain the reordering permutations when the graph is modified.

This raises a research gap, i.e., how to propose an effective approach to optimize the label proximity of the vertexes visited during the depth-first traversal in graph processing, which can be also effectively maintained in dynamic graphs. To mitigate this, we propose SeqDFS, a depth-first traversal method based on the Depth-First Search (DFS) sequence, which leads a higher cache hit rate in graph processing. We first generate the DFS sequence, according to the

SSDBM 2021, July 6–7, 2021, Tampa, FL, USA

ACM ISBN 978-1-4503-8413-1/21/07…$15.00
https://doi.org/10.1145/3468791.3468796

pop-up order of the DFS traversal tree, and reorder the vertexes of the graph based on the sequence. Therefore, such a DFS sequence can be regarded as a chain partitioning solution of the DFS tree, and each tree chain indicates a continuous subinterval of the DFS sequence. Based on this, edges of the graph are divided into tree edges and non-tree edges, and the traversal along the tree edges guarantees that the labels of the vertexes visited are continuous. In SeqDFS, when traversing the graph in a DFS manner, we determine the order of vertex traversal according to this DFS sequence, thereby improving the locality of the memory access and achieving a boosted cache hit rate. For dynamic graphs, we propose an efficient algorithm to maintain the DFS sequence, so that we can still effectively determine subsequent vertexes for DFS graph traversal based on this, in order to optimize the cache efficiency of graph processing. To the best of our knowledge, SeqDFS is the first to investigate how to directly take into consideration the vertex visit order that improves the cache efficiency of depth-first graph processing.

The contributions of this paper are summarized as follows:

- We propose to reorder the graph by DFS traversal. Compared with the state-of-the-art method[10], SeqDFS could be orders of magnitude faster in graph order construction. In addition, our method could be adapted to dynamic graphs, to the best of our knowledge, SeqDFS is the only method that could maintain the graph order in dynamic graphs.
- We propose a new DFS traversal method, SeqDFS, to be conducted along the vertex ordering sequence. SeqDFS optimizes the vertex visit order such that we could reduce the cache misses as well as branch prediction misses. SeqDFS is readily applicable and easy to implement, which can be widely adopted by the depth-first graph applications on both directed and undirected graphs.
- We evaluate our proposed SeqDFS on 16 large real-world datasets and 3 graph algorithms. The experimental evaluation results show that SeqDFS can outperform the state-of-the-art graph vertex reordering approaches among all evaluated datasets. For DFS traversal, SeqDFS could get a speedup of 1.51x. For strongly connected component detection, the speedup is 1.78x. For bridge detection, the speedup is 2.53x.

The rest of this paper is organized as follows. Section 2 presents the preliminaries. Section 3 reviews the related studies. Section 4 elaborates on the methodology of SeqDFS. We evaluted the proposed SeqDFS in Section 5. Section 6 concludes the paper.

2 PRELIMINARIES

In this paper, given a directed graph $G = (V, E)$, V is the vertex set and E is the edge set. The number of vertexes in G is denoted as $n = |V|$ and the number of edges in G is denoted as $m = |E|$. Given a vertex u, we denote the direct predecessor (resp. successor) of u as $N_{in}(u)$ (resp. $N_{out}(u)$). Then the in-degree and out-degree of vertex u are denoted as $d_{in}(u)$=$|N_{in}(u)|$ and $d_{out}(u)$=$|N_{out}(u)|$. For a given vertex u, the child vertexes connected by tree edges are denoted as $C_t(u)$, the other vertexes connected by non-tree edges are denoted as $C_{nt}(u)$. For an edge from vertex u to vertex v, we denote it as (u, v).

Definition 2.1 (DFS). Given a graph G, DFS traverses G in a particular order by picking an unvisited vertex v from the out-neighbors of the most recently visited vertex u to search, and backtracks to the vertex from where it comes, when u has explored all possible ways to search further [24].

Definition 2.2 (DFS Tree). Given a graph G, a DFS-Tree of G denoted by T_G, is an ordered spanning tree constructed by the process of DFS.[24]

Definition 2.3 (DFS Sequence). Given a graph G, a DFS sequence of G is generated by DFS traversal of the graph. Every vertex will be pushed into sequence when it is visited for the first time and labelled with the time when the vertex is popped out of the DFS stack.

3 RELATED WORK

As discussed, cache miss is the bottleneck that limits the performance of DFS. Hence, how to decrease the cache miss number is a main concern in this line of research. Some studies [2, 3, 10, 13, 27] propose to minimize the cache miss ratio by graph ordering.

For instance, Zhang et al. propose Frequency-Based Clustering [27]. Their approach is based on the observation that certain vertexes are much more likely to be accessed than others in power-law distributed graphs. Such vertexes are usually attached with a large number of edges. Thus, they reorganize the graph order according to the out-degrees so that the vertexes with large out-degrees will be clustered together. As Frequency-Based Clustering computes the graph ordering only based on the out-degrees, it achieves high efficiency in graph order construction, however, it may not bring large performance improvement for graph applications. Balaji et al. present Hub Clustering [3] which is a variation of Frequency-Based Clustering. Hub Clustering can achieve lower reordering overheads than Frequency-Based Clustering, but it leads to a reduced speed-up compared with Frequency-Based Clustering.

Lakhotia et al. [13] propose the Block Reordering approach. They first present a new metric *Profit* that quantifies cache data reuse. Then they conduct a joint optimization that maximizes both cache data reuse and cache line utilization. Their experiments show that they can achieve up to 2.3x speed-up over the original graph order. However, Block Reordering is designed for clustering sibling vertexes. They are not well optimized for DFS-based applications.

Arai et al. present Rabbit [2] which is a just-in-time paralleled graph ordering approach. They focus on the locality derived from hierarchical community structures in real-world graphs. Their approach is based on an observation that a community has dense inner-edges; thus, a vertex in a community is more likely to be visited from the other vertexes in the same community. Therefore, they try to improve the locality by co-locating vertexes in each community. By paralleling the graph ordering procedure, Rabbit could achieve end-to-end performance improvement. However, their approach tends to be effective only in graphs with community structures.

Wei et al. present Gorder [10], a general approach that tries to improve graph memory access by graph ordering. They try to find the optimal permutation among all vertexes in a given graph by keeping vertexes that are frequently accessed together locally. To be specific, they define a score function to measure the closeness

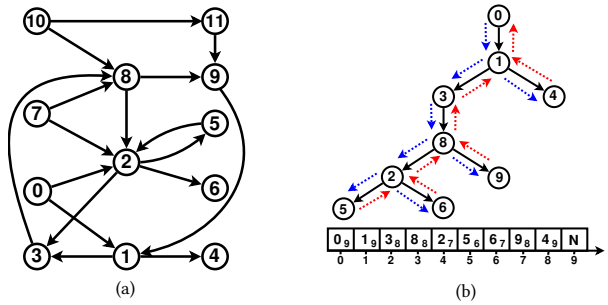


Figure 1: (a) is an example graph G. (b) shows the corresponding DFS tree rooted at vertex 0 and the DFS sequence.

of two vertexes, as higher scores are awarded if two vertexes are neighbors or siblings. Then, they maximize the sum of the scores for the graph vertexes in a sliding window such that the vertexes in the local area are more likely to be accessed together. In the best case, Gorder can achieve a speed-up of 2x. However, Gorder suffers from large overheads in graph order construction for some real-world graphs, e.g., hundreds or even thousands of seconds are needed [10] which is unacceptable for some applications.

Although the graph ordering approach can achieve a significant performance improvement, it has some disadvantages. As discussed in Section 1, it ignores the importance of the actual vertex visit order, as well as will suffer from performance degradation in dynamic graphs. Different from the previous studies, we reorder the graph to achieve a better vertex visit order. Most of the performance improvement comes from the optimization of the vertex visit order that is much more stable in dynamic graphs compared with existing work. In a nutshell, SeqDFS can achieve better performance in dynamic graphs even without maintaining the graph order, which indicates that SeqDFS is effective in both static and dynamic graphs.

4 SEQDFS: DEPTH-FIRST GRAPH TRAVERSAL ON DFS SEQUENCE

In this section, we present the sequence construction method and our proposed DFS algorithm. Based on these, we first show how our SeqDFS method is applied to trees, and then we demonstrate how SeqDFS is readily adapted to general graphs. Finally, we discuss how SeqDFS works on dynamic graphs.

4.1 SeqDFS on Trees

As discussed, the vertex sequence is constructed corresponding to the DFS tree. We show an example in Figure 1. Figure 1(a) is an example graph G that we will use throughout the paper. Figure 1(b) shows the DFS tree and the corresponding DFS sequence constructed from vertex 0. Vertexes are recorded when they are visited, i.e., pushed in, and the subscripts denote the timestamps when they are popped out. Therefore, the subscript of a vertex u refers to the index (in the DFS sequence) of the vertex v that should be visited after this vertex u is popped out. For example, vertex 2's subscript is 7, which means that the vertex in the 7th position (which is vertex 9) should be investigated after vertex 2 is popped out. At the end of the sequence, we add a flag vertex N (in position 9) to define the end of the sequence. Thus, vertexes 0, 1, and 4 will not visit any vertexes after they are popped out. We note that the non-tree edges are not covered by the DFS sequence, i.e., the sequence is a representation of the DFS tree constructed from vertex 0.

As the sequence maintains vertexes in the DFS visit order, it is more cache-friendly to the DFS procedure. Consider the DFS sequence starting at vertex 0, the next vertex is vertex 1 that is a child vertex of vertex 0. At the same time, vertex 3 is a child vertex of vertex 1, and vertex 8 is a child vertex of vertex 3. For all the vertexes in this sequence, 5 out of 9 vertexes are the child vertex of the vertex before it. Hence, if we conduct DFS along the sequence, the DFS procedure will be quite efficient as it acquires a higher data locality. However, the "next vertex" rule does not always hold. For example, although vertex 6 is in the next position of vertex 5, it is not a child vertex of vertex 5. If we only focus on the sequence, we couldn't prune vertex 6 from the visit. Therefore, pop-out time is needed.

Further, the sequence is a reflection of the DFS stack operation. For a vertex in the sequence, its position is the time when it is visited. The pop-out time implies the structure of the DFS tree used to exclude the vertexes that should not be visited. For instance, vertex 9's pop-out time is 8, which is the position where vertex 4 is located. This means vertex 4 is the next vertex that should be investigated after vertex 9 is popped out. However, vertex 4's pop-out position is larger than that of vertex 9, which indicates that vertex 4 is not a child vertex of vertex 9. Thus, vertex 4 should not be visited from vertex 9.

Combining the position of the vertex and the pop-out time, we can guarantee the correctness of the traversing graphs by the generated DFS sequences.

Lemma 4.1. *For vertex x in the sequence, vertex y is the next vertex in the sequence. Then vertex y is: 1) a child vertex of vertex x or, 2)a sibling vertex of x or, 3) a sibling vertex of x's ancestor vertexes.*

Proof. As mentioned, the vertexes in the sequence are in the DFS visit order. For vertex x that is currently visited, if there are unvisited child vertexes, one of them will be visited next. Thus, the next vertex in the sequence is the child vertex. Otherwise, vertex x will be popped out and DFS will backtrack to find the next vertex to visit. As backtracking is triggered, the next vertex must be a sibling vertex of x or a sibling vertex of x's ancestor vertexes. □

Lemma 4.2. *Given two vertexes u and v, if vertex u is in front of vertex v in the sequence and the pop-out position of vertex u is larger than (or equal to) that of vertex v, then vertex v is in the subtree rooted at vertex u.*

Proof. As vertex u is in front of vertex v in the sequence, u is visited earlier than v by DFS. If the pop-out position of vertex u is larger than (or equal to) v's pop-out position, u will be popped out after v is popped out. Hence, v must be in the subtree rooted at vertex u. □

Lemma 4.3. *Given two vertexes u and v, if vertex v is the next vertex of vertex u in the sequence and vertex u's pop-out position is*

Algorithm 1 SeqDFS on Trees

```
1: next = null //next is a global variable.
2: Procedure DFS(vertex cur)
3: next = the next vertex of cur in the sequence
4: while next.pop ≤ cur.pop do
5:     DFS(next)
6: end while
7: EndProcedure
```

larger than (or equal to) vertex v's pop-out position, then vertex v is a child vertex of vertex u.

Proof. Vertex u's pop-out position is larger than vertex v's pop-out position, this means that vertex u will be popped out later than v. From Lemma 4.2, we can infer that vertex v must in the subtree rooted at vertex u. According to Lemma 4.1, vertex v must be a child vertex of vertex u. □

Lemma 4.4. *Given two vertexes u and v, if vertex v is the next vertex of vertex u in the sequence, and vertex u's pop-out position is smaller than that of vertex v, then vertex v is a sibling vertex of vertex u or u's ancestor vertexes.*

Proof. From Lemma 4.1, we can infer that vertex v is either a child vertex of vertex u or a sibling vertex of vertex u(or u's ancestor vertexes). If vertex u's pop-out position is smaller than vertex v's pop-out position, this means that vertex v is not a child vertex of vertex u. Thus, it must be a sibling vertex of u or u's ancestor vertexes. □

We can infer from Lemma 4.3 and 4.4 that, if the current vertex's pop-out position is larger than(or equal to) that of the next vertex in the sequence, we can guarantee that the next vertex is a child vertex of the vertex visited currently. Therefore, we can directly visit the next vertex in this scenario. Otherwise, backtracking is invoked. We show how to conduct DFS along the sequence in Algorithm 1. We will check whether the next sequence vertex is a child of the currently visited vertex(line 4). If true, the next vertex will be visited; otherwise, we will backtrack to the ancestor vertex and try to visit the vertex *next*. As the sequence is constructed in DFS visit order, if the father vertex of *next* is in the DFS stack, *next* will be visited by its father vertex (Lemma 4.5). If the father vertex is not in the stack, meaning that the next vertex is out of the visit range, DFS will be terminated.

Lemma 4.5. *In Algorithm 1, given two vertexes u and v, if vertex v is visited from vertex u after backtracking, then vertex v is a child vertex of vertex u.*

Proof. From Lemma 4.3, we can infer that vertex v is not a child vertex of its previous vertex. Hence, backtracking will be triggered. The backtracking will be continued until the condition in line 4 is met. As the sequence is transformed from a DFS tree, the backtracking will find the first vertex that includes vertex v in its subtree. Thus, vertex u must be the father vertex of vertex v in the tree. □

Example 4.6. We take Figure 2 as an example to illustrate how SeqDFS works in the tree of Figure 1(b). The vertex *next* is shown

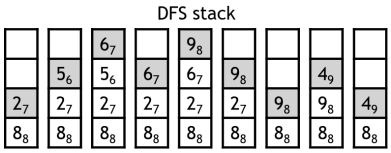


Figure 2: The DFS stack for the DFS procedure conducted from vertex 8.

at the top of the stack and colored in gray. Assume that we want to conduct DFS from vertex 8. The next vertex is 2. The subscript of vertex 2 is 7 which is smaller than that of vertex 8. Therefore, vertex 2 will be visited. Then vertex 5 is in the next position. Similarly, its subscript is smaller than that of vertex 2. Vertex 5 is visited. Then we will try to visit vertex 6. Its subscript is 7 which is larger than that of vertex 5; thus, vertex 5 will be popped out of the stack. We will backtrack to vertex 2. As *next* is a global variable, it still points to vertex 6. Vertex 2's subscript is equal to that of vertex 6, as a result, vertex 6 will be visited. After that, vertex 9 and 4 will be investigated. It is similar to the aforementioned procedure and thus, the corresponding details are omitted.

We can observe from this example that, for a DFS sequence constructed from a tree, as there are no non-tree edges, all the vertexes will be visited sequentially. In this scenario, SeqDFS will sequentially access the next vertex in the sequence and the smallest number of cache line replacement is needed. However, the sequence here can only be used to speed up the DFS procedure based on trees, which renders the application limited. Therefore, we extend SeqDFS for general graphs in the next subsection.

4.2 SeqDFS on General Graphs

We demonstrate how SeqDFS functions on trees in Section 4.1. Nevertheless, the real-world graphs are much more complicated in that there are a large number of non-tree edges in the graph. In this subsection, we discuss how to construct the DFS sequence on general graphs and perform DFS on them.

4.2.1 Sequence Construction. It is generally difficult to find the graph ordering that achieves the optimal graph processing performance, e.g., Gorder [10] proves that it is NP-hard to obtain the graph ordering with the highest Gscore. In this paper, we propose a heuristic method to determine the graph ordering. As mentioned in the previous subsection, higher data locality can be achieved when we conduct DFS along the sequence. For a sequence constructed from a graph, the more edges covered by the sequence, the better performance can be achieved. Therefore, we adopt a greedy strategy to construct the sequence, aiming to cover as many edges as possible.

Lemma 4.7. *Given a graph G and the corresponding DFS trees, the number of the DFS trees is t. Then, the number of edges covered by the trees is $n - t$.*

Lemma 4.8. *Given a graph G in which all the vertexes are in the same weakly connected component, then the smallest number of DFS trees needed to cover the graph is equal to the number of vertexes whose $d_{in}(\cdot)$ is zero.*

Algorithm 2 Sequence Construction

```
1:  seqIndex = 0
2:  Procedure Con_DFS(vertex cur, sequence seq)
3:  set cur as visited
4:  seqVertex = cur
5:  seq.push_back(seqVertex)
6:  seqIndex = seqIndex+1;
7:  for vertex x in N_out(cur) do
8:    if x not visited then
9:      Con_DFS(x, seq)
10:   else
11:     add edge x to seqVertex
12:   end if
13: end for
14: seqVertex.pop = seqIndex
15: EndProcedure

16: Procedure Seq_Con(G(V, E))
17: seq = new sequence
18: for vertex x in V do
19:   if d_in(x) = 0 then
20:     Con_DFS(x, seq)
21:   end if
22: end for
23: for vertex x in V do
24:   if x not visited then
25:     Con_DFS(x, seq)
26:   end if
27: end for
28: EndProcedure
```

The proof of Lemma 4.7 and 4.8 is obvious, and thus is omitted. We can conclude from Lemma 4.7 that, the fewer trees we use to cover the graph, the more edges will be covered by the sequence. Therefore, to cover more edges, we should try to use as few trees as possible. Inspired by Lemma 4.8, we first try to construct the sequence from the vertexes whose $d_{in}(\cdot)$ is zero. For some isolated strongly connected components (SCC) in the graph, there will be no vertexes with a zero in-degree; thus, we will construct the sequence from the remaining unvisited vertexes. It needs to be noted that, conduct the process from any vertexes in the isolated SCC would cover all the vertexes in the same SCC. After that, all the vertexes are covered by our DFS sequence.

We show the sequence construction algorithm in Algorithm 2. From line 18 to 22, we try to traverse the graph from vertexes whose in-degree is zero. If all the vertexes are in a weakly connected component, the number of DFS trees will be equal to the number of zero in-degree vertexes. Then, we construct the sequence from the unvisited vertexes to cover the vertexes in the isolated SCC. The *Con_DFS* procedure shows how our sequence is constructed. From line 4 to 5, we initialize a new sequence vertex and push the vertex into the sequence. Then, we increase the sequence index (*seqIndex*) in line 6. We note that the *seqIndex* is a global variable that is initialized in line 1. After that, we check the child vertexes. If the vertexes are not visited, we will recursively conduct DFS along the edges (line 9). Otherwise, we will add the edge to the sequence vertex. (line 11). Finally, when the vertex needs to be popped out, we record its pop-out time (line 14). For the example graph in Figure 1(a), we show the corresponding DFS sequence in Figure 3. In addition to the sequence, every vertex has an adjacent list that points to the child vertexes connected by non-tree edges.

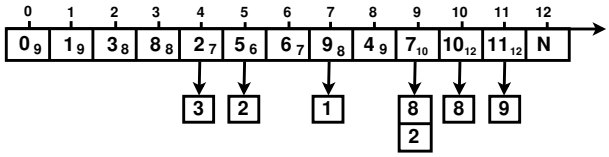


Figure 3: DFS sequence for graph G with non-tree edges.

In our method, we only consider the relationship between two neighbouring vertexes. In fact, the graph order can be further improved by taking more adjacency information into consideration. The most straightforward method is to apply Gorder's approach. That is, when we perform DFS to construct the sequence, we always try to choose a child vertex that is more connected with the sequence vertexes in front of it. For example, in Figure 3, after vertex 2 is pushed in, we can either push vertex 5 or 6 into the sequence. However, there is another edge $(5, 2)$ in the graph which means vertex 5 is better connected with vertex 2; thus, vertex 5 should be pushed into the next position. In this example, we can also consider more adjacency information, e.g. the edge connections among vertex 8, 3 and 1. Nevertheless, the optimization will incur great overheads. As we want to implement a lightweight graph ordering method, the overheads are unacceptable for SeqDFS.

4.2.2 DFS traversal. As discussed, the edges in the graph fall into two categories: 1) tree edges that are covered by the sequence and 2) non-tree edges that are kept in the adjacency list. Therefore, two different methods can be used to traverse the graph. For a vertex that is visited currently, when we try to visit its child vertexes, we can either visit the tree edges first or visit the non-tree edges first. For the method that visits non-tree edges first, we call it AF, meaning adjacency list first. The other one is called SF, meaning sequence first. We note that we cannot control the memory access pattern when we visit the non-tree edges. In fact, from the view of memory access, there are no differences between the AF method and the classic adjacency list based DFS except that the graph is reordered. Thus, the performance of the AF method is a reflection of the effectiveness of our graph ordering method. Compared with the AF method, the SF method optimizes the vertex visit order. It always tries to visit the next memory address first. As a result, higher data locality can be achieved if the child vertex is in the next position. To evaluate how much performance improvement can be achieved from the graph ordering and the vertex visit order optimization, we discuss both methods in this subsection.

We illustrate the DFS algorithm with the SF method in Algorithm 3. When we try to visit a vertex, we will first mark it as visited. Then, we will check the next vertex in the sequence in line 4. If the next vertex is an unvisited child vertex, we will visit it in line 6. No matter whether the current vertex is visited or not, the vertex *next* will point to the vertex in the current vertex's pop-out position (line 8), which is the next vertex we should try to visit. It will be

Algorithm 3 Sequence First DFS Algorithm

```
1:  Procedure SF(vertex cur)
2:  cur.vis = true
3:  next = the next vertex of cur
4:  while next.pop ≤ cur.pop do
5:      if next.vis ≠ true then
6:          SF(next)
7:      end if
8:      next = the vertex in next's pop position
9:  end while
10: for x ∈ C_nt(cur) do
11:     if x.vis ≠ true then
12:         SF(x)
13:     end if
14: end for
15: EndProcedure
```

Algorithm 4 Adjacent list First DFS Algorithm

```
1:  vertex next //next is a global variable
2:  Procedure AF(vertex cur)
3:  cur.vis = true
4:  for x ∈ C_nt(cur) do
5:      if x.vis ≠ true then
6:          AF(x)
7:      end if
8:  end for
9:  next = the next vertex of cur
10: while next.pop ≤ cur.pop do
11:     if next.vis = true then
12:         next = the vertex in next's pop position
13:     else
14:         AF(next)
15:     end if
16: end while
17: EndProcedure
```

visited in the next run if it is a child vertex of the vertex visited currently. The *while* loop from line 4 to 8 guarantees that all the child vertexes connected by tree edges will be visited. After this, the child vertexes connected by non-tree edges will be visited (line 10 to 14).

There are some scenarios the DFS procedure needs to visit the child vertexes in a specific order. For example, some applications will perform the DFS according to the weight of the edges. In this scenario, we may need to add some extra control flows to adapt to this scenario. For example, add an if condition to determine which visit order we should follow in this run. It needs to be noted that, even if we need to visit the vertexes connected by non-tree edges first, our method(AF method) could still achieve comparable performance with the state-of-the-art methods. We will elaborate on the details in Section 5.3.

Example 4.9. We show an example of the SF method in Figure 4. Assume that we need to traverse from vertex 1. Vertex 3's subscript is 8 which is smaller than that of vertex 1. Thus, vertex 3 will be pushed into the stack. Similarly, vertexes 8, 2 and 5 will be pushed into stack sequentially. Vertex 6's subscript is larger than that of vertex 5. Hence, vertex 5 tries to visit the non-tree edges. However, vertex 2 is visited before, so vertex 5 is popped out and backtracks to vertex 2. Vertex *next* now points to vertex 6 and visits it. Then, *next* points to vertex 9 whose subscript is larger than that of vertex 6 and vertex 2. As both vertexes have no non-tree edges, they will be popped out. Vertex 9 will be visited from vertex 8. After that, vertexes 4 and 7 will be investigated. The details are omitted as they are similar to the aforementioned procedure.

From this example, we can deduce that the SF method can achieve a higher cache utilization. For instance, in Example 4.9, vertexes 1, 3, 8, 2 and 5 are sequentially accessed from memory. Compared with the traditional DFS approach that accesses memory in a random manner, the visit order of the SF method can greatly lower the cache misses. In fact, the SF method can also achieve higher cache efficiency than the AF method. Assume that we want to conduct DFS from vertex 8, if we visit non-tree edges first, we will only sequentially visit vertex 2 after we visit vertex 8. Then, vertex 3 will be visited through non-tree edges by random memory access. In this case, although vertex 5 is also a child vertex of vertex 2 and it is just in the next memory position, the AF method will not visit it in the next run. However, for the SF method, vertex 8, 2 and 5 will be visited sequentially, the optimized vertex visit order can achieve a higher cache efficiency.

In addition to the higher cache efficiency achieved, in practice, the SF method can also lower branch prediction misses. This is because the SF method handles edges in different categories in different manners. It can be observed from Figure 3, most of the non-tree edges will connect vertexes in front of the current vertex. On the contrary, all the tree edges will connect vertexes behind the current vertex. When we visit vertexes connected by tree edges, we always try to visit vertexes backward that are more likely to be unvisited. When we visit the vertexes connected by non-tree edges, we always try to visit the vertex in the front position, which is very likely to be already "visited". For instance, in Example 4.9, we start our traversal from vertex 1, when we visit vertexes 1, 3, 8, 2, 5, 6, 9, and 4, as all of them are first visited by the SF access, the "visited" status checking in line 5 will always be true. On the contrary, when we need to visit the non-tree edges $(2, 3)$, $(5, 2)$, and $(9, 1)$, as the edges try to visit vertexes backward, the status checking in line 11 will always be false. The high degree of consistency will greatly improve the branch prediction accuracy.

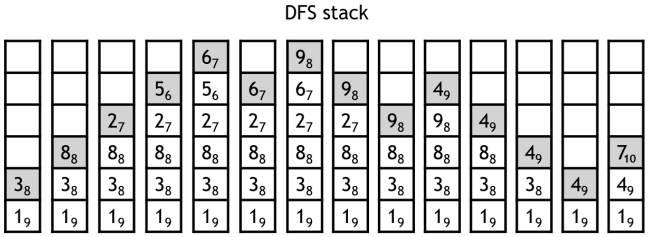


Figure 4: Sequence first DFS traverse from vertex 1.

Table 1: Percentage of condition being true in line 5 and 11.

Dataset	line 5	line 11	Dataset	line 5	line 11
LJ	90.79	0.63	Reddit	88.86	0.49
Web	96.77	0.62	Amazon	91.98	1.08
Email	97.18	1.23	Wiki-t	95.11	2.42
Wiki	96.32	3.31	Youtube	95.29	1.41
Pokec	89.75	0.53	CS	92.53	2.15
BerkStan	99.11	0.07	DB	98.01	0.71
Twitter	91.35	4.25	Trec	97.45	0.47
Arabic	99.36	0.02	Uk	98.99	0.06

To validate this, we conduct an experiment to compute the percentage of the condition being true in line 5 and 11 (Algorithm 3), respectively. The result is shown in Table 1. We can observe from the table that, in most cases, more than 90 percent of the "if" condition will be true in line 5. However, for the "if" condition in line 11, most will be false. In fact, the percentage of the condition being true is correlated to the number of cycles in the graph. As stated in the previous paragraph, for a cycle in the graph, there must be a backward edge in our DFS sequence which tends to incur a false condition in line 11. At the same time, as it will continue to traverse along the sequence, it will meet a visited vertex to finish the cycle. Thus, it is bound to incur a false condition in line 5. For some social network graphs such as Lj, Pokec and Reddit, as there are a large number of cycles, they are more likely to get a false condition in both line 5 and 11.

In fact, we can achieve a higher branch prediction accuracy when we need to do a whole graph traversal. It frequently happens in some applications. For instance, the Tarjan algorithm will traverse all the graph vertexes when we need to find the SCCs or generate a DAG (Directed Acyclic Graph). We want to highlight that in static graphs, the Tarjan algorithms only need to be executed once to generate the DAGs. The preprocessing time for graph ordering may be hard to amortize. However, for dynamic graphs whose edges can be inserted/deleted, there is no perfect approach to maintain the DAGs [26]. Especially when edges are deleted, although there are some techniques [26] that can speed up the DAG computation, DFS traversal is still inevitable. As a consequence, it is essential to accelerate the Tarjan algorithms.

For the Tarjan algorithms, the source vertex from which we start the traversal does not matter. We can traverse the graph in the same order that we construct the sequence. Thus, in Figure 3, we can start the Tarjan algorithms from vertex 0. In this case, all the vertexes connected by tree edges will be "unvisited", and the "if" condition in line 5 will always be true. On the contrary, the status of the vertex connected by non-tree edges will be "visited", and the "if" condition in line 11 will always be false. Therefore, **there will be no branch predictions for the "visited" status checking.**

The corresponding pseudocode is shown in Algorithm 5. Our SCC detection procedure visits vertexes along the sequence (line 30). The visit order is the same as the sequence construction order. All the tree edges are handled in the "while" loop from line 9 to 13. As the vertexes connected by the tree edges are unvisited, there is thus no need to check the visit state of the vertex. We recursively traverse the child vertex (line 10). Similarly, the vertexes connected by the non-tree edges are handled from line 14 to 18. All the vertexes are visited before, and thus, we only need to check whether they are still in the stack (line 15). Obviously, the elimination of the branch predictions will greatly speedup the performance.

Algorithm 5 Strongly Connected Component Detection

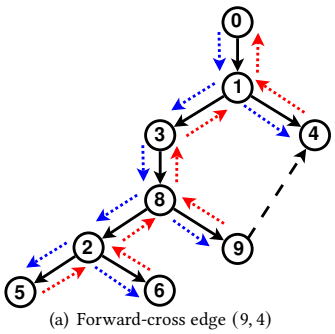
```
1:  dfn=0
2:  scc=0
3:  Procedure Tarjan_SeqDFS(vertex now, stack tjstack)
4:  now.dfn = dfn++
5:  now.low = now.dfn
6:  now.vis = true
7:  tjstack.push_back(add(now))
8:  vertex *ptr = add(now)+1//next sequence vertex
9:  while ptr.pop ≤ now.pop do
10:    Tarjan_SeqDFS(*ptr, tstack)
11:    now.low = min(now.low, ptr.low)
12:    ptr = address of seq(ptr.pop)
13: end while
14: for x ∈ C_nt(now) do
15:    if x ∈ tjstack then
16:       now.low = min(now.low, x.dfn)
17:    end if
18: end for
19: if now.low = now.dfn then
20:    vertex *nxt = tjstack.pop_back()
21:    nxt.scc = scc
22:    while nxt ≠ add(now) do
23:       nxt = tjstack.pop_back()
24:       nxt.scc = scc
25:    end while
26:    scc++
27: end if
28: EndProcedure

29: Procedure Scc_Detection
30: for i=0; i<n; i++ do
31:    if seq[i] hasn't been visited then
32:       Tarjan_SeqDFS(seq[i], stack tjstack)
33:    end if
34: end for
35: EndProcedure
```

4.3 SeqDFS on Dynamic Graphs

We have thus far discussed SeqDFS on static graphs. Nevertheless, the real-world graphs may be evolving. Once the graph is modified, the corresponding graph ordering desires to be updated to achieve optimal performance on the updated graph. However, for most graph reordering approaches, they only focus on the static graphs. Hence, an entire reconstruction is needed after every update, which suffers from great performance degradation in dynamic scenarios and generally impractical. To solve this problem, we discuss how SeqDFS deals with dynamic graphs in this subsection.

4.3.1 Maintaining the Graph Order. Different from the existing approaches, SeqDFS provides a maintainable graph ordering. As discussed, the DFS sequence corresponds to a DFS tree. Therefore, the problem can be transformed into maintaining the DFS tree which has been extensively studied [4, 6, 24]. Yang et al. [24]



(a) Forward-cross edge $(9, 4)$

v	Time Interval
0	$(0, 17)$
1	$(1, 16)$
2	$(4, 9)$
3	$(2, 13)$
4	$(14, 15)$
5	$(5, 6)$
6	$(7, 8)$
8	$(3, 12)$
9	$(10, 11)$

(b) The time interval label

Figure 5: The newly inserted forward-cross edge $(9, 4)$ and the time interval label.

propose an efficient DFS tree maintaining approach that can be adapted to our scenario. Fortunately, no extra structure/information is needed in the maintaining process. We will discuss their approach in this section.

Generally, DFS tree maintaining is to eliminate the edges that will break the DFS tree structure, which are defined as forward-cross edge. A newly inserted edge (u, v) is a forward-cross edge, if it does not have an ancestor/descendant relationship, and u is visited before v in the construction of the DFS tree [24]. We show an example in Figure 5(a). During the construction of the DFS tree, vertexes are visited along the blue arrows and backtracked along the red arrows. After the DFS tree is constructed, a new edge $(9, 4)$ is inserted. However, vertex 9 is visited before vertex 4 during the construction of the DFS tree and they do not have an ancestor/descendant relationship in the original DFS tree. Thus, the edge $(9, 4)$ is a forward-cross edge.

The core operation in the DFS tree maintaining approach is to find out the forward-cross edges. To identify the forward-cross edges, Yang el al. propose to label the graph vertexes with the time interval that is defined in Definition 4.10. We show the corresponding time interval of Figure 5(a) in Figure 5(b).

Definition 4.10 (Time Interval). Given a DFS Tree $\mathcal{T}$ and a vertex u, the time interval of u is denoted as $\mathcal{I}_{\mathcal{T}}(u) = [x, y]$, where $x = \mathcal{I}_{\mathcal{T}}(u).left$ is the visited timestamp of u in the DFS traversal, and $y = \mathcal{I}_{\mathcal{T}}(u).right$ is the pop-out timestamp of u when its out-neighbors are visited completely in the DFS traversal [8].

An inserted edge (s, t) will be a forward-cross edge if $\mathcal{I}_{\mathcal{T}}(s).right < \mathcal{I}_{\mathcal{T}}(t).left$. This means, in the original DFS tree, vertex s is popped out of stack before vertex t is visited. For example, in Figure 5(b), vertex 9's pop-out time is 11 and vertex 4's visit time is 14. Vertex 9 is popped out before vertex 4 is visited. Thus, it is a forward-cross edge. We note that we can deduce the time interval information from our DFS sequence. In fact, as we construct the DFS sequence by the DFS procedure, the visited time of the vertex can be inferred from its index in the sequence. Vertex u is visited earlier than vertex v if it is in front of vertex v in the sequence. Thus, for SeqDFS, $\mathcal{I}_{\mathcal{T}}(\cdot).left$ can be replaced by the vertex's index. The corresponding $\mathcal{I}_{\mathcal{T}}(\cdot).right$ can be replaced by the vertex's pop-out time used in SeqDFS. Therefore, we can directly identify the forward-cross edges using our DFS sequence.

Another core operation for DFS tree maintaining is to reconstruct the local tree structure. Yang et al. [24] try to locate a minimum area that would be affected by the edge update. In the process, the LCA (lowest common ancestor) method is used to find out the root vertex where the reconstruction process starts. Finally, a DFS procedure is executed to modify the tree structure and the time interval.

Although our graph ordering is maintainable, the maintaining is impractical for real-world graphs. First, graph ordering maintaining tends to cost high overheads. As the graph is sequentially kept in memory, modifying the vertex position means moving and copying large memory blocks, which is very expensive. Second, the DFS tree maintaining method is time-consuming. For example, when the edge insertion updates the tree, it will cost nearly 0.5 second to modify the tree structure in LiveJournal, even worse, the update for deletion will cost more than 1 second [24]. Nevertheless, for SeqDFS, reconstructing the whole graph will take only 3.28 seconds. This means that for large number of graph updates, periodically reconstructing the graph ordering is a practical strategy.

4.3.2 Maintaining the Sequence on Dynamic Graphs. SeqDFS achieves most of its performance improvement from the optimization of the vertex visit order. It means that we can still achieve a satisfactory performance improvement even if we do not maintain the graph ordering. However, due to our proposed optimization in vertex visit order, further operations are needed to handle the dynamic graphs. We hence discuss the dynamic graphs in this subsection.

In SeqDFS, we consider each newly inserted edge as a non-tree edge. Thus, when we insert an edge (u, v), we only need to add the edge to the adjacency list of vertex u. Compared with edge insertions, edge deletions are much more complicated. If the edge to be deleted is a non-tree edge, we will directly delete it from the vertex's adjacency list. If the edge to be deleted is a tree edge, as it is implicitly covered by the sequence, we need to traverse along the sequence to find out the edge and mark the target vertex with a flag $tedge$. If $tedge$ is set to false, it means the tree edge that connects to the vertex is deleted. The process is very similar to the aforementioned DFS traversal procedure, thus we omit the algorithm here.

Vertex insertions/deletions rely on edge insertions/deletions. When we need to insert a new vertex, we only need to add it to the end of our sequence and insert all the related edges as non-tree edges. The pop-out time of the newly inserted vertex will be the prior vertex's pop-out time plus one. Hence, it will never be visited by sequential access. Vertex deletions are simple. We only need to delete all the related edges.

5 EVALUATION

In this section, we conduct experiments by comparing SeqDFS with the state-of-the-art approaches. In Section 5.1, we introduce the experimental setup. In Section 5.2, we demonstrate the graph ordering construction overhead in different reordering approaches. Then we conduct comparison in terms of DFS running time and the corresponding effectiveness of cache/branch prediction in Section 5.3 and 5.4 respectively. Finally, we evaluate the performance of SeqDFS in SCC and bridge detection.

5.1 Experimental Setup

Environment: Our experiments are conducted 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.

Baselines: We compare several existing approaches with our proposed SeqDFS. For all the baseline methods, we acquire the code from their authors.

- Original: The original graph ordering.
- Gorder [10]: The state-of-the-art approach that focuses on graph ordering, which maximize the locality metric, Gscore, by a proposed greed strategy.
- Rabbit [2]: A lightweight graph ordering approach that focuses on end-to-end performance improvement.
- HC [3]: HubCluster, a variation of [27], which is a lightweight Frequency-Based graph reordering approach that could achieve the best performance in graph ordering construction.
- AF: SeqDFS with the AF method proposed in this paper, with sequence first optimization disabled.
- SF: SeqDFS with the SF method proposed in this paper, with optimized vertex visit order by sequence first optimization.

We note that for Original, Gorder, Rabbit and HC, the classic DFS algorithm [8] is used to evaluate their performance.

Datasets: We conduct experiments on 16 real-world datasets. The datasets are shown in Table 2. We collect the datasets from SNAP [14], KONECT [12], and LAW [1]. All the datasets are stored in the compressed sparse row (CSR) format.

- LJ, Youtube, and Pokec are social network datasets, which are power-law graphs in nature.
- CS is a citation network extracted from the CiteSeer digital library.
- BerkStan and Web are web graph datasets in which vertexes represent web pages and directed edges represent hyperlinks between them.
- Wiki, Wiki-t, and Email are communication networks.
- Amazon is an Amazon product co-purchasing network.
- DB is a complete DBpedia network. Vertexes are entities included in DBpedia and each edge represents one triple.
- Trec is a web graph collected from the TREC conference.
- Reddit and Twitter are two social network datasets obtained from [22].
- Uk and Arabic are two web datasets crawled by [5].

5.2 Construction of the Graph Ordering

We show the graph reordering time for all the approaches in Table 2. HC achieves the best performance in graph ordering construction. As HC is a lightweight approach that sorts the graph vertexes only according to their out-degree numbers, its time complexity depends on the sorting approach they used. On the contrary, Gorder brings the highest overhead in graph ordering construction. Gorder applies an approximate approach of travelling salesman problem (TSP) to compute the graph ordering, and hence, the process is very time-consuming. After the optimization of priority queue, its time complexity is $O(\sum_{u \in V}(d_{out}(u))^2)$. Rabbit is a community-based graph ordering method. The exact time complexity is not shown in their paper; however, it is claimed that Rabbit has a time complexity roughly in proportion to m. SeqDFS can achieve the suboptimal performance in graph ordering construction. This is because SeqDFS is a traversal-based approach. Most of the overhead is caused by the traversal which dominates our graph ordering construction costs. Thus, the time complexity is $O(n+m)$. We want to highlight that, as all the existing graph ordering methods could not be adapted to dynamic graphs, periodically reconstruct the graph order is the only way to maintain the graph order. Thus, for online scenarios, the methods with large order construction overheads could hardly be applicable.

Table 2: Dataset statistics and graph ordering time(s).

Dataset	$\|V\|$(M)	$\|E\|$(M)	d_{avg}	Gorder	Rabbit	HC	SeqDFS
LJ	4.84	68.99	14.23	56.91	10.17	0.13	3.28
Web	0.87	5.11	5.83	0.98	0.51	0.03	0.42
Email	0.26	0.42	1.58	0.09	0.06	0.01	0.03
Wiki	2.39	5.02	2.09	3.92	0.91	0.04	0.26
Pokec	1.63	30.62	18.75	19.98	4.06	0.05	1.64
BerkStan	0.68	7.60	11.09	0.92	0.35	0.02	0.16
Twitter	2.88	6.43	2.23	5.39	0.92	0.08	0.47
Reddit	2.63	57.49	21.86	94.08	8.63	0.09	2.32
Amazon	0.40	3.39	16.79	0.56	0.25	0.01	0.19
Wiki-t	1.14	7.83	6.87	3.24	0.51	0.03	0.14
Youtube	1.14	4.94	8.68	2.6	0.69	0.05	0.25
CS	0.38	1.75	4.06	0.54	0.15	0.02	0.12
DB	3.97	13.82	6.96	3.66	1.87	0.09	0.88
Trec	1.60	8.06	10.06	0.9	0.57	0.04	0.26
Arabic	22.74	640	28.14	89.71	30.28	0.80	9.05
Uk	18.52	298.11	16.09	39.31	17.81	0.52	5.50

5.3 DFS Running Time

The performance of all evaluated approaches in terms of DFS running time is illustrated in Figure 6. Every time, we will randomly select 100 source vertexes and conduct DFS from them, the total DFS time is recorded. We will report the average time of 10 repeats.

HC exhibits the worst performance in the experiment. For all the 16 datasets, HC can only speed up 4 of them. In most cases, HC even degrades the DFS performance. Compared with HC, Rabbit can achieve better performance. Nevertheless, on average, it cannot achieve any performance improvement than the Original. This is because in some of the datasets, the original graph ordering may already have a certain degree of locality in nature.

For all the baselines, Gorder is the only approach that can reduce the DFS running time among all the datasets. Therefore, in the remainder of this paper, we only compare our proposal with Gorder.

LJ, Pokec, and Reddit are social network datasets with a large average out-degree. For Gorder, they introduce high overheads (56.91s, 19.98s and 94.08s, respectively) to compute the graph ordering. Compared with the original graph, on average, Gorder can save 34% of the DFS running time. However, SF can reduce 47% of the DFS running time while the graph ordering overheads are much smaller (3.28s, 1.64s and 2.32s, respectively). We note that although the AF method can also achieve a better performance than the original graph, it is slower than Gorder in these three datasets.

Most datasets (Web, BerkStan, Amazon, Youtube, DB and Trec) take several seconds to conduct DFS. Compared with Original, the

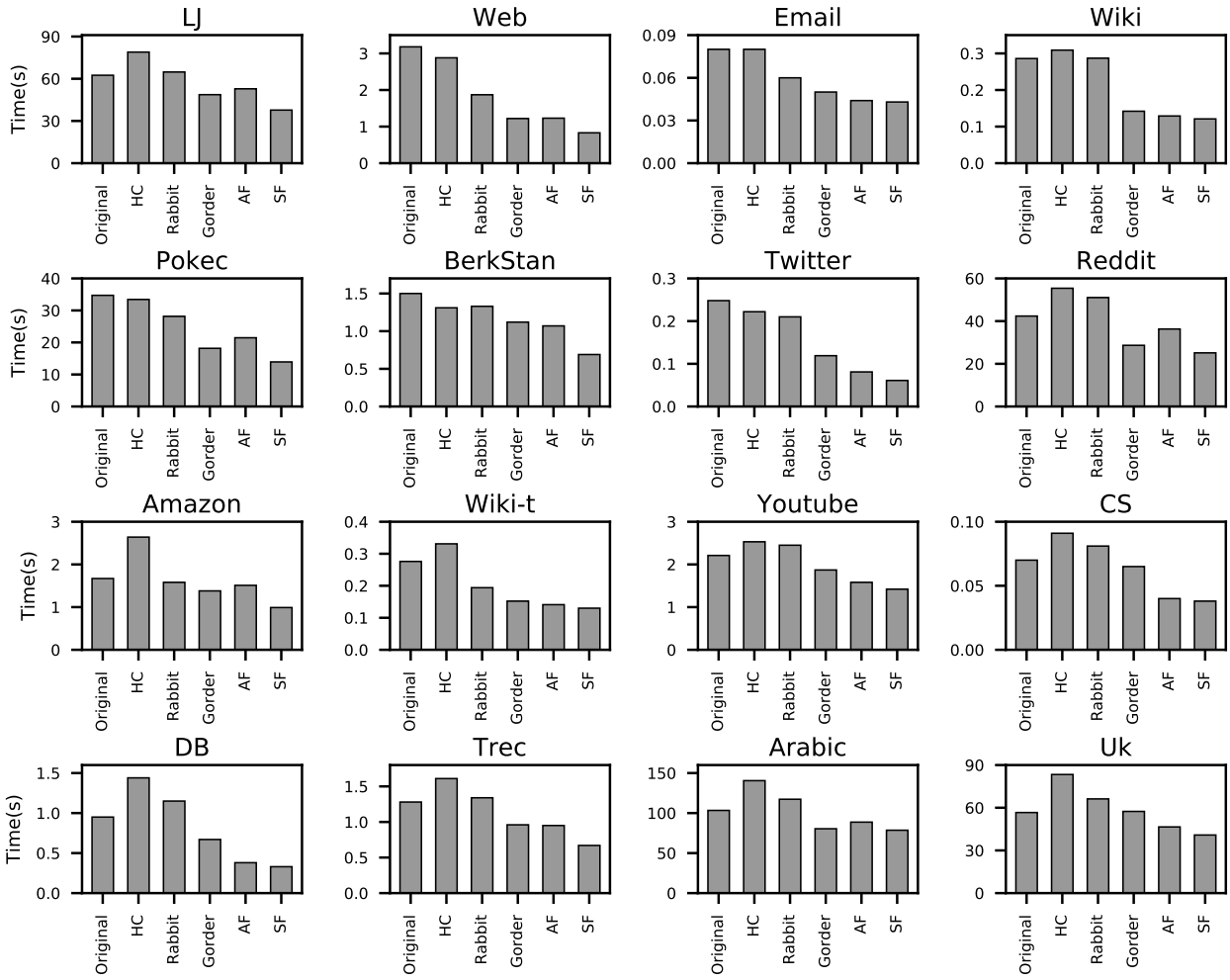


Figure 6: DFS running time (in seconds)

largest performance improvement can be achieved in Web. For the SF method, 73% of the DFS running can be reduced. For Gorder, the number is 61%. Nevertheless, the largest performance gap lies in DB. Compared with Gorder, SF can reduce more than 50% of the DFS running time. On average, for these 6 datasets, SF can save 33% of the DFS running time.

Arabic and Uk are two large datasets with several hundred millions of edges. SF achieves the best performance in both datasets. On average, SF can save 16% of the DFS running time than Gorder. Compared with the Original, the number is 26%.

The DFS running time in the 5 other datasets is less than one second. The largest performance improvement is achieved in Twitter. Gorder can reduce 50% of the DFS running time while SF can save more than 75% of the DFS running time. Similar performance can be achieved in Wiki and Wiki-t. In both datasets, SF can reduce more than 50% of the DFS running time. For the 5 datasets (Email, Wiki, Wiki-t, Twitter, CS), compared with Gorder, SF can reduce 26% of the DFS running time on average.

SF consistently outperforms Gorder in all the datasets. In fact, even AF can outperform Gorder in most of the datasets except LJ, Pokec, Reddit, Arabic, and Amazon. As discussed, SeqDFS accelerates DFS by optimizing the vertex visit order. For the DFS traversal along the tree edges, we can access the vertexes sequentially. However, the vertexes connected by non-tree edges will still incur random memory access. Therefore, the fewer the non-tree edges are, the better performance we can achieve. Nevertheless, all of these datasets are dense graphs with an average out-degree larger than 14. There are a lot of non-tree edges that will incur random memory access. Thus, the speed-up is limited.

This experiment also proves that optimizing the vertex visit order is essential. The SF method outperforms the AF method in all the datasets. In the best cases, in Web and BerkStan, SF can reduce 34% of the DFS running time than the AF method. However, they achieve similar performance in Email and CS. This is because these graphs are small and sparse. When the graph is small, few vertexes will be visited during DFS, the performance gap will hence be small. If the graph is sparse with few non-tree edges, both the AF and the

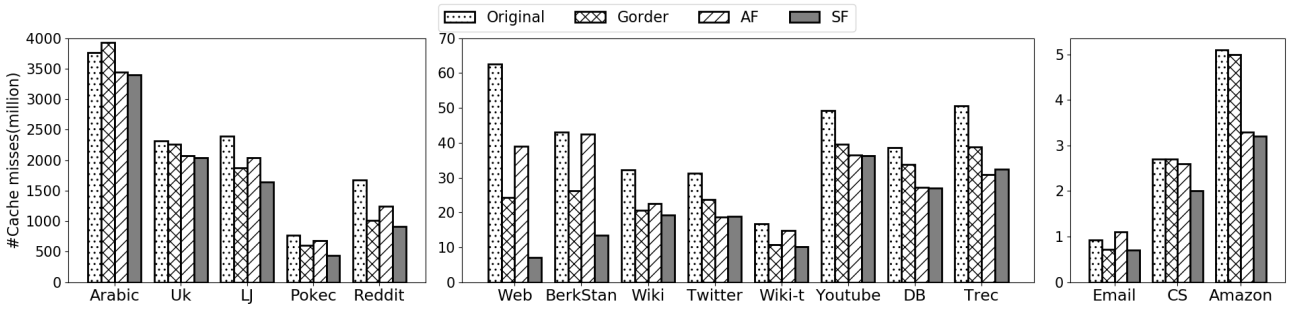


Figure 7: Number of last level cache misses (million)

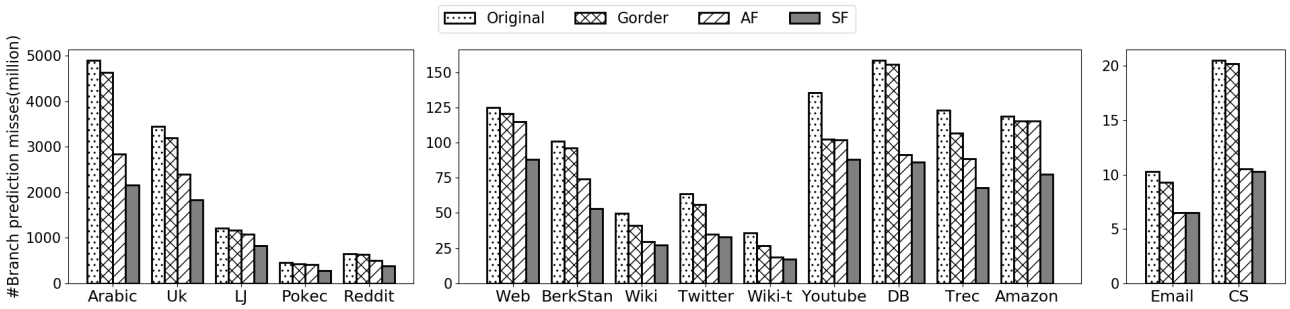


Figure 8: Number of branch prediction misses(million)

SF method will conduct DFS along the sequence. Thus, they will achieve a similar performance.

5.4 Cache/Branch Prediction Effectiveness

We use Valgind[15] to simulate the cache and branch prediction procedure. The last level cache miss number is shown in Figure 7. For all the 16 datasets, the SF method achieves the smallest cache miss number in 14 of them. In Twitter, the cache miss number of the SF method is a little larger than that of the AF method, but the gap is small. However, in Trec, AF performs better than SF, about 4% of the cache miss number can be reduced.

For the large social network datasets LJ, Pokec and Reddit, compared with Gorder, the AF method will cause more cache misses; however, SF can reduce an average of 16% cache miss number. Similar to Figure 6, the largest performance gap lies in Web. In our experiment, when we execute DFS on Web, SF will cause a little more than 7 million cache misses. For Gorder, this number is 24 million. Gorder and SF achieve similar performance in Wiki and Wiki-t. Nevertheless, SF is always preferable. For the other datasets, compared with Gorder, SF can achieve an average of 17.3% cache miss number reduction.

For all the 16 datasets, Gorder outperforms AF in 8 of them. This means if we apply the graph ordering approach without the optimization on the vertex visit order, AF can only achieve a similar cache efficiency with Gorder. In practice, compared with the Original, AF can still achieve an average of 17% cache miss reduction.

We show the branch prediction miss number in Figure 8. Generally, Gorder's branch prediction is more accurate than the Original. However, the improvement is limited. Both the AF and the SF method can achieve a more significant improvement in branch prediction than Gorder. For LJ, pokec and Reddit, compared with Gorder, the AF method can reduce an average of 11% branch prediction miss number. The SF method performs even better, with 37% reduction achieved. For BerkStan and Web, this tendency stays the same. Compared with Gorder, SF can reduce an average of 35% of the branch prediction misses. The largest performance gap lies in Arabic, in which SF can save more than 56% of the branch prediction miss number than Gorder.

In most cases, SF outperforms AF. However, they achieve similar performance in Wiki, Twitter, Wiki-t, Email and CS. This is because these graphs are sparse. As mentioned above, both AF and SF tend to visit vertexes along the sequence when the graph is sparse. The similar memory access pattern will hence, lead to similar performance in branch prediction. We highlight that, in these datasets, the SF method can still achieve a much better branch prediction performance than Gorder, with an average of 37.8% branch prediction misses reduced.

5.5 Performance of Tarjan Algorithms

We compare the performance of SCC detection(Tarjan algorithm) with the optimization of Gorder and SeqDFS, respectively. The results are shown in Table 3. SeqDFS outperforms Gorder on all

Table 3: The processing time(ms) for SCC detection and bridge detection(in the bracket).

Dataset	Original	Gorder	SeqDFS
LJ	1835(2068)	1517(1988)	1140(1102)
Web	159(178)	73(120)	57(52)
Email	10(16)	7(8)	5(4)
Wiki	177(211)	92(145)	85(65)
Pokec	739(891)	632(838)	501(493)
BerkStan	58(88)	52(85)	38(51)
Twitter	273(340)	155(217)	144(98)
Reddit	1620(1725)	1140(1419)	1071(907)
Amazon	88(99)	58(88)	36(38)
Wiki-t	84(111)	56(75)	44(40)
Youtube	138(160)	115(145)	72(62)
CS	33(68)	24(60)	22(22)
DB	273(564)	225(504)	165(209)
Trec	110(169)	94(154)	59(71)
Arabic	4063(7039)	3812(6760)	3515(4605)
Uk	2486(3280)	2121(3195)	1936(2319)

the datasets. On average, SeqDFS can save 20% of the overhead. Compared with the Original, 41% of the overhead can be saved.

We also adapt SeqDFS to the bridge detection problem that can also be solved by Tarjan algorithms. As it is quite similar to Algorithm 5, we omit the corresponding pseudocode. We transform all the datasets into undirected graphs and conduct experiments on them. The results are shown in Table 3, where the numbers in the bracket are the time needed to conduct bridge detection. For Original and Gorder, bridge detection costs more overhead than SCC detection in all the datasets. However, for our SeqDFS method, bridge detection can be a little faster than SCC detection in some datasets. As Gorder relies on connections to construct the graph order, the undirected graph with a higher connectivity makes it difficult for Gorder to cluster vertexes. However, SeqDFS can benefit from the undirected graphs. Due to the higher connectivity in undirected graphs, SeqDFS can construct a larger tree structure, which means more visits will be executed along the sequence. Thus, an even better performance than in directed graphs can be achieved. For the original graphs, SeqDFS can reduce 57% of the overhead. Compared with Gorder, the number is 48%. This proves that SeqDFS can be readily applicable to other graph applications that depend on depth-first traversal with satisfactory performance.

6 CONCLUSION

In this paper, we present SeqDFS to speed up the DFS performance. First, we reorder the graph vertexes into a vertex sequence that is more consistent with the DFS visit order. Second, we propose a new DFS method in which part of the DFS visits can access the memory sequentially. Both methods can achieve fewer cache misses and branch prediction misses, and thus, improve the DFS performance. We conduct the experiment on 16 real-world graphs. Our experiments show that, compared with the state-of-the-art approaches, SeqDFS can achieve an average of 27% DFS running time reduction while the graph ordering overhead is only 1/15. Our experiments also demonstrate that our method could be used to speedup other DFS-based algorithms, e.g., SCC and bridge detection.

7 ACKNOWLEDGEMENTS

Mo Sha's work is partially supported by a MoE Tier 2 grant (MOE2017-T2-1-141) in Singapore.

REFERENCES

[1] [n.d.]. LAW: The Laboratory for Web Algorithmics. http://law.di.unimi.it.
[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.
[3] Vignesh Balaji and Brandon Lucia. 2018. 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)*.
[4] Surender Baswana and Keerti Choudhary. 2015. On dynamic DFS tree in directed graphs. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 102–114.
[5] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.
[6] Lijie Chen, Ran Duan, Ruosong Wang, and Hanrui Zhang. 2016. Improved algorithms for maintaining DFS tree in undirected graphs. *CoRR, abs/1607.04913* (2016).
[7] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.
[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
[9] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Etats-Unis Informaticien, and Edsger Wybe Dijkstra. 1976. *A discipline of programming*. Vol. 1. prentice-hall Englewood Cliffs.
[10] Wei Hao, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *the 2016 International Conference on Manage of Data*.
[11] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378.
[12] Jérôme Kunegis. 2017. The Koblenz Network Collection. http://konect.uni-koblenz.de.
[13] Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, and Viktor Prasanna. 2017. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 273–282.
[14] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[15] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746
[16] Micha Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72.
[17] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2017. Reachability querying: can it be even faster? *TKDE* 1 (2017), 1–1.
[18] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.
[19] Robert Endre Tarjan. 1976. Edge-disjoint spanning trees and depth-first search. *Acta Informatica* 6, 2 (1976), 171–185.
[20] Robert Endre Tarjan and TARJAN RE. 1974. A Note on Finding the Bridges of a Graph. (1974).
[21] Alan Tucker. 2006. Chapter 2: covering circuits and graph colorings. *Applied Combinatorics* 49 (2006).
[22] Yanhao Wang, Qi Fan, Yuchen Li, and Kian-Lee Tan. 2017. Real-time influence maximization on dynamic social streams. *Proceedings of the VLDB Endowment* 10, 7 (2017), 805–816.
[23] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2018. Reachability querying: an independent permutation labeling approach. *VLDBJ* 27, 1 (2018), 1–26.
[24] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. 2019. Fully dynamic depth-first search in directed graphs. *Proceedings of the VLDB Endowment* 13, 2 (2019), 142–154.
[25] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. 2010. Grail: Scalable reachability index for large graphs. *PVLDB* 3, 1-2 (2010), 276–284.
[26] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. 2013. Dagger: A scalable index for reachability queries in large dynamic graphs. *arXiv preprint arXiv:1301.0977* (2013).
[27] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.
[28] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*. ACM, 1323–1334.