Divide & Conquer: I/O Efficient Depth-First Search

Zhiwei Zhang[†], Jeffrey Xu Yu[†], Lu Qin[‡], Zechao Shang[†]

[†]The Chinese University of Hong Kong, China, {zwzhang,yu,zcshang}@se.cuhk.edu.hk

[‡]Centre for QCIS, FEIT, University of Technology, Sydney, Australia, {Lu.Qin}@uts.edu.au

ABSTRACT

Depth-First Search (*DFS*), which traverses a graph in the depth-first order, is one of the fundamental graph operations, and the result of *DFS* over all nodes in G is a spanning tree known as a DFS-Tree. There are many graph algorithms that need *DFS* such as connected component computation, topological sort, community detection, eulerian path computation, graph bipartiteness testing, planar graph testing, etc, because the in-memory *DFS* algorithm shows it can be done in linear time w.r.t. the size of G. However, given the fact that real-world graphs grow rapidly in the big data era, the in-memory *DFS* algorithm cannot be used to handle a large graph that cannot be entirely held in main memory. In this paper, we focus on I/O efficiency and study semi-external algorithms to *DFS* a graph G which is on disk. Here, like the existing semi-external algorithms, we assume that a spanning tree of G can be held in main memory and the remaining edges of G are kept on disk, and compute the DFS-Tree in main memory with which *DFS* can be identified. We propose novel divide & conquer algorithms to *DFS* over a graph G on disk. In brief, we divide a graph into several subgraphs, compute the DFS-Tree for each subgraph independently, and then merge them together to compute the DFS-Tree for the whole graph. With the global DFS-Tree computed we identify *DFS*. We discuss the valid division, that can lead to the correct *DFS*, and the challenges to do so. We propose two division algorithms, named Divide-Star and Divide-TD, and a merge algorithm. We conduct extensive experimental studies using four real massive datasets and several synthetic datasets to confirm the I/O efficiency of our approach.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms

Keywords

Graph Algorithm; I/O Efficient; Depth-first Search

1. INTRODUCTION

Depth-first search (*DFS*) is one of the fundamental graph operations to access all nodes in a graph G by traversing it in the depth-first order, and the result of *DFS* over all nodes in G is a spanning tree known as a DFS-Tree. The importance of *DFS* comes from the fact that a large number of graph algorithms need to access nodes in an order, and there are a large number of applications that need to traverse a graph in the depth-first order. In brief, the graph algorithms that need *DFS* include topological sort, reachability query, finding connected components, planar graph testing, graph bipartiteness testing, and even frequent subgraphs mining.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723740.

Motivation: In the literature, the in-memory *DFS* algorithm has been well studied, and is shown to be done in linear time, w.r.t. the graph size [16]. However, due to the fact of rapid graph growth in the big data era, the size of many graphs grows rapidly so that they cannot entirely reside in main memory. For example, there are more than 1.32 billion nodes with average degree more than 132 in facebook[1]. As a small part of the entire web, uk-2007-05[2] contains 105,896,555 nodes and 3,738,733,648 edges. Motivated by this, in this paper, we study new I/O efficient algorithms to *DFS* a graph G when G cannot be entirely held in main memory.

Related Work: In the external memory (EM) model [1], it assumes that the main memory can only keep M elements while the remaining will be kept in blocks on disk, where one block contains B elements. Suppose one I/O access will read/write B elements (1 block) from/into disk into/from main memory. The I/O complexity to scan N elements, denoted as $scan()$, is $\Theta(\frac{N}{B})$ I/Os, and the I/O complexity to sort N elements, denoted as $sort()$, is $O(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$ I/Os [1].

Given the external memory model, Chiang et al. [5] propose a *DFS* algorithm for a graph $G(V, E)$ with I/O complexity $O(|V| + \frac{|V|}{M} \cdot scan(|E|) + sort(|E|))$. Later, Kumar and Schwabe [10] and Buchsbaum et al. [4] improve to $O((|V| + \frac{|E|}{B}) \log_2 \frac{|V|}{B} + sort(|E|))$. The main idea of the approaches is to reduce the I/O cost to find the next unvisited node. They reduce I/O complexity to $O(N)$, from $O(E)$ by a naive approach, using either tournament trees [10] or buffered repository trees [4]. However, they still need a large number of I/Os, because $O(N)$ I/Os for every step in *DFS* is very high, and is impractical to be used.

The most up-to-date semi-external algorithm for *DFS* is by Sibeyn et al. [14]. They assume that main memory can hold a spanning tree of G but not all the edges of G, and convert the *DFS* problem to the problem of finding a DFS-Tree with which *DFS* can be computed. It is worth noting that Zhang et al. [18] study an I/O efficient semi-external algorithm to find all strongly connected components in a graph using a weak order instead of the total order that *DFS* implies. Their approach cannot be used for *DFS*, because the weak order cannot be used to find the DFS-Tree.

There are reported studies on some specific graphs. Her et al. [9] propose an external memory method to *DFS* in general grid graphs. Arge et al. [3] propose a method for undirected embedded planar

[1]http://newsroom.fb.com/company-info/

[2]http://law.di.unimi.it/datasets.php

graphs. However, these approaches cannot be used to handle a general directed graph. Surveys about designing I/O efficient algorithms for massive graphs can be found in [17, 2].

There are also distributed/parallel algorithms. Makki et al. [11] and Sharma et al. [13] propose distributed *DFS* algorithms with bound message length and transmitting times. Makki et al. [12] propose a distributed algorithm to find the DFS-Tree in a distributed environment, which uses a stack-type structure to enable better dynamic backtracking. However, simulating the content in the messages and transmitting will result in a large number of random access. Freeman [8] examines a parallel *DFS* algorithm in sub-linear time, which only works on undirected or planar graphs.

The Main Contributions: In this paper, we discuss the inefficiency of the existing semi-external *DFS* algorithms, and propose a novel divide & conquer semi-external algorithm that significantly reduces the I/O cost for *DFS*, where a semi-external algorithm assumes that only a spanning tree of the graph can be held in memory. In our divide & conquer algorithm, we divide a graph into several subgraphs, compute the DFS-Tree for each subgraph independently, and then merge them together to compute the DFS-Tree for the whole graph. With the global DFS-Tree computed we identify *DFS*. It is important to note that a subgraph can be further divided recursively if it is still too large to fit in memory, and for each subgraph divided we can reduce the I/O cost because we do not need to scan the entire graph to handle a subgraph. We discuss the motivation of designing a divide & conquer *DFS* algorithm, the valid division that can lead to the correct *DFS*, and the challenges to do so. We design two division algorithms, named Divide-Star and Divide-TD, and a merge algorithm. By Divide-Star, the division is made according to the children of the root. By Divide-TD, the division is made according to a cut-tree. We conduct extensive experimental studies using four real massive datasets and several synthetic datasets to confirm the I/O efficiency of our approach.

Paper Organization: The remainder of this paper is organized as follows. We give the problem statement in Section 2, and discuss the existing solutions in Section 3. In Section 4, we show the benefit and challenges of divide & conquer approaches and outline the framework of our approaches. We discuss the properties of a valid division in Section 5, and propose two divide algorithms in Section 6 and one merge algorithm in Section 7. We report our experimental results in Section 8. We conclude this work in Section 9.

2. THE PROBLEM STATEMENT

We model a directed graph as $G(V, E)$, where V represents the set of nodes and E represents the set of edges (ordered pairs of nodes). We denote the number of nodes and edges by n and m, i.e., $n = |V|$ and $m = |E|$, respectively. In the following, we may use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges of a graph G, when necessary. In addition, we use $N_O(u)$ and $N_I(u)$ to denote the out-neighbors and in-neighbors of u, i.e., $N_O(u) = \{v \mid (u,v) \in G\}$ and $N_I(u) = \{v \mid (v,u) \in G\}$, respectively. The in-degree and out-degree of node u are denoted as $d_I(u) = |N_I(u)|$ and $d_O(u) = |N_O(u)|$ respectively.

Depth-First Search (*DFS*): Given a graph $G(V, E)$, depth-first search (*DFS*) is to search G following the depth-first order. Specifically, it searches a graph G in an order by picking up an unvisited node v from the out-neighbors of the most recently visited u, $N_O(u)$, to search, and backtracks to the node from where it comes when a node u has explored all possible ways to search further [6]. In the following discussion, for simplicity and without loss of generality, we assume that a graph is connected in the sense that its underneath undirected graph is connected, and there is a node in G whose in-degree is zero to start *DFS*. When there are more than one node whose in-degree is zero, we can add a virtual root node γ which has an edge to every node v in G that has zero in-degree

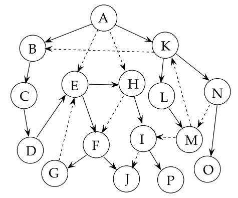
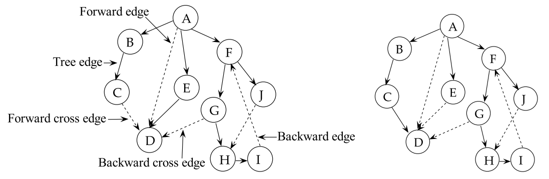


Figure 1: A Graph G and its DFS-Tree



(a) A spanning tree T with edge types (b) A DFS-Tree

Figure 2: An Example

($d_I(v) = 0$). By *DFS* starting from γ, it can *DFS* every node in G. When there is no node whose in-degree is zero, a node in G will be randomly selected as the first node for *DFS*. The similar steps can be performed to handle a graph that is not connected.

The DFS-Tree: A *DFS* of G results in a total order over all nodes in G. It is worth mentioning that *DFS* is not unique and the total order is not unique. The result of *DFS* forms an ordered spanning tree called a DFS-Tree. By *DFS* the graph in a different order, it may result in a valid but different DFS-Tree. The DFS-Tree can be used to represent the *DFS* or the total order. In other words, *DFS* the graph G is equivalent to finding the DFS-Tree of G.

Example 2.1: Fig. 1 shows a graph G. Starting *DFS* from node A, a result of *DFS* shows a total order over all nodes in G, which can be A, B, C, D, E, F, G, J, H, I, P, K, L, M, N and O. The corresponding DFS-Tree is marked by solid lines in Fig. 1. □

The need to find a DFS-Tree of G first followed by finding *DFS* over G stems from the fact that a graph can be very large such that it cannot be entirely held in memory. In such a situation, the application of any in-memory *DFS* algorithm will result in a huge number of random I/O accesses, which makes it impractical.

Edge Types: In order to discuss how DFS-Tree can be used in finding *DFS* over G when G cannot be held in memory, we classify edges of G into different types under an ordered spanning tree of G [6, 14]. Let T be an ordered spanning tree of G, the edges of G that appear in T are called tree-edges. The remaining edges, (u, v), are non-tree edges, and are categorized into four types: (1) forward-edge if u is an ancestor of v in T, (2) backward-edge if u is a descendant of v in T, (3) forward-cross edge if u and v do not have ancestor/descendant relationship and u is visited before v by the preorder of T for G, and (4) backward-cross edge if u and v do not have ancestor/descendant relationship and u is visited after v by the preorder of T for G. We use cross-edge to denote an edge that is either a forward-cross edge or a backward-cross edge. By definition, as shown in [14], a DFS-Tree is an ordered spanning tree T that does not have any forward-cross edges. As a result, the problem of *DFS* in G can be solved by finding a DFS-Tree of G, and the condition of ensuring that a spanning tree is a DFS-Tree can be done by checking forward-cross edges. Based on this, a semi-external *DFS* algorithm is designed to hold a spanning tree in memory when the graph G cannot be held in memory.

Example 2.2: A graph G is shown in Fig. 2(a). Here, the ordered spanning tree is represented by the nodes connected by the solid edges where the nodes are visited in an order of A, B, C, E, D, F, G, H, I, and J. Such a spanning tree is not a DFS-Tree, because (C, D) is a forward-cross edge. On the other hand, the spanning tree shown in Fig. 2(b) is a DFS-Tree, since no forward-cross edges exist in the spanning tree. □

Problem Statement: To *DFS* over a directed graph G that cannot be held entirely in memory, in this paper, we study a semi-external algorithm that aims at computing a DFS-Tree (by which *DFS* can be obtained) with the limited given memory M such that $k \cdot |V| \leq M \leq |G|$. Here, k is a small constant number (e.g., $k = 3$), and $|G| = |V| + |E|$.

Discussion: In this paper, as the first step, we focus on designing a semi-external *DFS* algorithm by assuming that $k \cdot |V| \leq M$. This restricts the algorithm to only handle a graph G whose spanning tree can be held in main memory, and the focus becomes which edges we should keep in main memory to find a DFS-Tree in an I/O efficient manner. Such an assumption is made due to the following reasons. For most social networks and web graphs, the number of edges is much larger than the number of nodes. In SNAP[3] among 79 real-world graphs, the largest graph contains 65 M nodes and 1.8 G edges. In KONET[4] among 230 real-world graphs, the largest graph contains 68 M nodes and 2.6 G edges. Second, designing an I/O efficient semi-external *DFS* algorithm is non-trivial. Third, it has been well recognized that external *DFS* is very hard to be solved efficiently [10, 5, 4]. In the literature, the best external *DFS* algorithm requires $O((|V| + \frac{|E|}{B}) \log_2 \frac{|V|}{B} + sort(|E|))$ I/Os [10, 4] which is obviously too high to handle real-world large graphs.

3. EXISTING SOLUTIONS

Algorithm EdgeByEdge**:** Sibeyn et al. [14] propose a semi-external algorithm to find the DFS-Tree for a graph G. The basic idea is to repeatedly restructure the spanning tree T held in memory by replacing any forward-cross edge found in T when reading edges of G from disk until there do not exist any forward-cross edges in G.

Example 3.1: Reconsider Fig. 2 where we omit the virtual node γ. Assume that a spanning tree T is in memory, which includes the solid edges as shown in Fig. 2(a). The dotted edges in Fig. 2(a) represent the non-tree edges. Among the five non-tree edges, (A, D), (C, D), (G, D), (J, H) and (I, F), the edge (C, D) is the only forward-cross edge w.r.t. T in memory. When edge (C, D) is read from G on disk, by restructure, EdgeByEdge will delete the edge of (E, D) from T and add (C, D) into T. This results in a spanning tree T shown in Fig. 2(b). Assume that the visiting order by *DFS* is A, B, C, D, E, F, G, H, I, and J over T in Fig. 2(b). EdgeByEdge finds that there is no more forward-cross edges in the next iteration when scanning G. As a result, the spanning tree in Fig. 2(b) is a DFS-Tree to be computed. □

Algorithm EdgeByBatch**:** By EdgeByEdge, the number of I/O accesses required to find the DFS-Tree may be large since it may need to scan the entire graph G for n times in the worst-case. This is because it may reduce only one forward-cross edge in scanning G in every iteration. In order to reduce the number of I/O accesses by making use of the memory whenever possible, Sibeyn et al. in [14] propose a batch processing algorithm, denoted as EdgeByBatch, which is illustrated in Algorithm 1. The algorithm first constructs an initial spanning tree T in memory (line 1-2). Then, it restructures the spanning tree T iteratively by invoking the procedure Restructure(G, T, M) until no forward-cross edges exist in G w.r.t. T. The procedure Restructure is shown in line 7-16 of Algorithm 1. It loads edges of G on disk into the memory in a batch manner. In every batch, it will construct a graph G_M by adding as many edges as possible into the spanning tree T held in memory under the constraint of $|G_M| \leq M$ (line 11). Here, $E(G_M) \subseteq E(G)$. Given G_M held in memory, it constructs a new DFS-Tree T for G_M in memory (line 14) if there exists a forward-cross edge in G_M w.r.t. T (line 12), and removes those non-tree edges regarding T from the memory (line 15). A variable $update$ is used to check whether there exists at least one forward-cross edge in G w.r.t. T (line 13), and it is used as a termination condition for the EdgeByBatch algorithm. After all edges of G are processed in a batch manner, the spanning tree T along with the variable $update$ is returned (line 16).

[3]http://snap.stanford.edu/data/

[4]http://konect.uni-koblenz.de/networks/

Algorithm 1 EdgeByBatch(graph G, memory size M)

```
1: for all nodes $v \in V(G)$ do
2:     add edge $(\gamma, v)$ in $T$ where $\gamma$ is a virtual node;
3: $update \leftarrow$ true;
4: while $update =$ true do
5:     $(T, update) \leftarrow$ Restructure$(G, T, M)$;
6: return $T$;

7: Procedure Restructure (graph $G$, spanning tree $T$, memory size $M$)
8: $update \leftarrow$ false;
9: while there exist unprocessed edges of $G$ on disk do
10:     $G_M \leftarrow T$;
11:     load unprocessed edges of $G$ from disk to enlarge $G_M$ in memory where
        possible under the constraint of $|G_M| \leq M$;
12:     if there exists a forward-cross edge in $G_M$ w.r.t. $T$ then
13:         $update \leftarrow$ true
14:         construct a new $T$ by conducting DFS over $G_M$;
15:     remove non-tree edges of $G_M$ and only keep $T$ in memory;
16: return $(T, update)$;
```

In Algorithm 1, whenever there is more memory available, it will load as many edges as possible and apply the *DFS* procedure to the graph in memory. Note that *DFS* should visit the nodes which stay in memory before newly loaded ones. Algorithm 1 performs well when a large part of the graph can be loaded into memory. However, since the graph size cannot be reduced during the *DFS* procedure, the batch processing has to scan the whole graph even if only one forward-cross edge exists in G, which is inefficient. In the worst case, the algorithm still needs to scan all the edges of the graph on disk for n times.

4. A NEW DIVIDE & CONQUER METHOD

In this section, we analysis the problem and outline our new approach to compute the DFS-Tree.

4.1 Problem Analysis

We give the motivation to design new divide & conquer algorithms by analyzing the drawbacks of the existing semi-external *DFS* algorithms, and we show the challenges of designing a good divided & conquer algorithm.

Drawbacks of Existing Solutions: When the graph G cannot entirely fit in the main memory, the existing algorithms, EdgeByEdge and EdgeByBatch, have several drawbacks which make them inefficient to compute the DFS-Tree.

(1) A total order in $V(G)$ needs to be maintained in the whole process of DFS. Recall that when the graph G can fit in the main memory, the in-memory based *DFS* algorithm in [6] can perform *DFS* efficiently based on a total order of all nodes in G. This is because the algorithm can find the next node to be visited one by one in the total order by checking unvisited nodes and each node can be checked in constant time. In the semi-external algorithms EdgeByEdge and EdgeByBatch introduced in [14] to compute the DFS-Tree, the total order is also adopted to restructure the in-memory spanning tree. The total order of nodes in $V(G)$ needs to be maintained in every iteration of both algorithms, since whenever the spanning tree is restructured both algorithms need the new total order of all nodes to compute and update the type of every edge in the graph G. Maintaining such a total order may result in high computational cost because once the order of a certain node is changed, all the nodes with larger order may need to update their positions in the total order accordingly.

(2) A large number of I/Os are produced. Both EdgeByEdge and EdgeByBatch have to scan all the edges of G in order to eliminate even only one forward-cross edge found in an iteration.

(3) A large number of iterations are needed due to low locality. Both EdgeByEdge and EdgeByBatch need a large number of iterations to terminate. This is because they assume that edges on disk are stored in an arbitrary order without considering data locality, i.e., they do not consider the possibility to group together the edges that are near each other in the visiting sequence when *DFS* the graph. As a result, even if the batch processing technique is adopted, it is possible that, in one iteration, the elimination of a forward-cross edge e_1 in a certain batch will produce a new forward-cross edge e_2 in another batch that has to be eliminated in the next iteration. In other words, the locality-unaware approach adopted in both EdgeByEdge and EdgeByBatch will result in a chain effect in the process of forward-cross edge elimination, which in turn requires a large number of iterations.

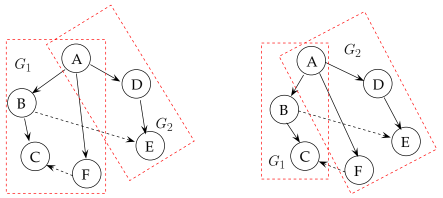
Why Divide & Conquer? The above drawbacks motivate us to find more efficient semi-external algorithms to compute the DFS-Tree. In this paper, we propose a new solution based on divide & conquer. We aim at dividing the graph G into several subgraphs $G_0, G_1, \cdots, G_p$ with possible overlaps with each other, such that the DFS-Tree of G can be obtained by combining the DFS-Tree for each G_i ($0 \leq i \leq p$) together. We show such a division always exists. Let T be the DFS-Tree of G. Suppose there are p subtrees $T_1, T_2, \cdots, T_p$ of T together with a tree T_0. Here, the root of T_0 is the root of T and all the p leaves of T_0 are the root of p subtrees. Obviously, $G_0, G_1, \cdots, G_p$ is a valid division for divide and conquer, if every G_i ($0 \leq i \leq p$) is a subgraph induced by nodes in T_i. The divide & conquer approach can overcome the above drawbacks for EdgeByEdge and EdgeByBatch in the following three ways.

1) After dividing G into $G_0, G_1, \cdots, G_p$, instead of maintaining a total order for all nodes in $V(G)$, we only need to maintain a total order for nodes in each $V(G_i)$ for $0 \leq i \leq p$. As a result, when the order of a node in a certain $V(G_i)$ is changed, only the order of the nodes in G_i needs to be updated.

2) Given the divided subgraphs $G_0, G_1, \cdots, G_p$ for the original graph G, the DFS-Tree for G can be computed using the DFS-Tree for each G_i ($0 \leq i \leq p$). In such a way, when a certain forward-cross edge exists in a certain graph G_i, we only need to scan the edges in G_i to eliminate such a forward-cross edge without wasting I/Os to scan the edges of the whole graph G on disk. Therefore, the number of I/Os is largely reduced.

3) With the divided subgraphs $G_0, \cdots, G_p$ for G, the locality for edges w.r.t. *DFS* is largely increased. This is because the *DFS* process can be performed for each G_i ($0 \leq i \leq p$) independently. In such a way, when processing G_i using EdgeByBatch, a less number of batches are needed in every iteration when scanning edges. This can largely reduce the number of iterations caused by the chain effect. In the case that G_i can fit entirely in the main memory, only one batch is needed for EdgeByBatch to terminate in one iteration.

Based on the above discussion, to achieve a good division G_0, $G_1, \cdots, G_p$ of the graph G, each subgraph G_i should be as small as possible. In order to achieve this, we need to (1) maximize the number of divided subgraphs, and (2) minimize the size difference between any two divided subgraphs at the same time. Here, to maximize the number of subgraphs is to enlarge the chances of finding a division that can divide a large graph into small subgraphs. The maximization is interrelated to the minimization of differences between two divided subgraphs, because a division is not cost effective, if it always divides a large graph into a large subgraph along with a large number of small subgraphs.



(a) Valid Division (b) Invalid Division

Figure 3: Two Divisions for Graph G

Challenges: Although a valid division of a graph always exists for *DFS*, there are still a lot of difficulties to find a good division.

(Challenge 1): It is not straightforward to check whether a division is valid. In order to guarantee that a division is valid, four properties should be satisfied, namely, node-coverage, contractible, independence, and *DFS*-preservable. By node-coverage, we mean that the divided subgraphs should jointly cover all nodes in the graph. By contractible, we mean that each divided subgraph should be smaller than the original graph. By independence, we mean that the result of the *DFS* on a certain subgraph does not affect that of another. By *DFS*-preservable, we mean the DFS-Tree for a graph can be constructed based on the DFS-Tree for each divided subgraph. However, checking the properties, especially the *DFS*-preservable property, is not straightforward. This is because after dividing G into $G_0, G_1, \cdots, G_p$, and computing the corresponding DFS-Trees $T_0, T_1, \cdots, T_p$, by *DFS*-preservable, we need to check whether there is a way to organize all the DFS-Trees for the divided subgraphs to form the DFS-Tree for G with no forward-cross edges w.r.t. G, and the possible ways to organize all such DFS-Trees for the divided subgraphs can be exponential. The details of the four properties will be introduced in Section 5.

(Challenge 2): Finding a good division method is non-trivial. A straightforward division method is to divide a graph into subgraphs such that there is no cross-division edges, i.e., edges that connect different subgraphs. However, such a division may not always exist. Recall that a good division should satisfy two conditions. First, it should generate as many subgraphs as possible. Second, the subgraph sizes should not vary too much. However, neither of the conditions is easy to be satisfied. For the first condition, even dividing the graph into two subgraphs to satisfy the properties of a valid division is non-trivial, since the cross-division edges can have various types. The situation becomes more complex when we want to divide the graph into multiple subgraphs. For the second condition, it makes the problem more difficult since dividing the graph evenly usually results in a larger number of cross-division edges, making the properties for a valid division harder to be satisfied.

(Challenge 3): A merge procedure needs to be carefully designed to ensure that the result tree is a DFS-Tree*.* Recall that a DFS-Tree is an ordered tree. Even if a good division of the graph is calculated to satisfy the properties of a valid division, we cannot simply combine them together to form the final DFS-Tree, since this may produce forward-cross edges w.r.t. the cross-division edges. Therefore, we need to carefully design a merge procedure that can combine the DFS-Trees for the divided subgraphs in a certain order such that no new forward-cross edges w.r.t. the cross-division edges are produced after the combination of all such DFS-Trees.

Example 4.1: For example, in Fig. 3, for the graph G, and the same spanning tree T shown in solid lines, there are two different division results. The first division divides the graphs into G_1 and G_2 as shown in Fig. 3(a), and the corresponding DFS-Trees are T_1 and T_2, which are two subtrees of T. Simply combining T_1 and T_2 will result in T, which is not a DFS-Tree since edge (B, E) is a forward-cross edge. However, if we exchange the positions of T_1 and T_2 in T, it becomes a valid DFS-Tree since no forward-cross edge exists in this case. Thus, the division in Fig. 3(a) is a valid division. The second division divides the graphs into another G_1 and G_2 as shown in Fig. 3(b), and the corresponding DFS-Trees are T_1 and T_2, which are two subtrees of T. For this division, no matter how T_1 and T_2 are ordered to form a new tree, the merged tree will not be a DFS-Tree for G since either edge (B, E) or (F, C) will be the forward-cross edge. Now, suppose we restructure the graph G by adding the forward cross edge (B, E) into the spanning tree T and removing the edge (D, E) from T, then no matter how the graph G is divided according to the subtrees of T, it will result in a valid division since the whole tree T is a DFS-Tree. □

The above example shows that, in order to find a valid division, how the divided subgraphs are ordered such that their DFS-Trees can be combined to form the DFS-Tree of the whole graph is important, and it can determine the order of merging. Based on this observation, we need to consider the relationships among all subgraphs in both the division procedure and merge procedure, and find a way to capture such relationships when designing an effective divide & conquer algorithm.

4.2 Algorithm Framework

Our divide & conquer algorithm is designed to overcome the three challenges introduced in the previous subsection as follows.

1) To address *challenge 1*, after a division, in addition to compute the divided subgraphs G_0, G_1, $\cdots$, G_p, we also compute a light-weight summary graph (S-Graph) denoted as Σ. Σ is used to capture the relationships among all the divided subgraphs. Given such a S-Graph Σ, we can check whether a division is valid by searching the S-Graph instead of searching the whole graph G. This can also save large computational cost since the S-Graph Σ is much smaller than the original graph G and thus can be kept in the main memory.

2) To address *challenge 2*, first, we design the divide & conquer algorithm in a way that can divide the graph as early as possible when the in-memory spanning tree is restructured iteratively, in order to avoid scanning the whole graph. Second, according to the first condition of a good division, we find two division algorithms, namely, Divide-Star and Divide-TD with the aim of dividing the graph G into as most subgraphs as possible. The details of the two division algorithms will be introduced in Section 6. Third, according to the second condition of a good division, we need to avoid generating very large subgraphs in the division. In order to achieve this, after a division, for a subgraph that is still very large, we recursively divide such a subgraph using the same divide & conquer process until it is restructured to form a DFS-Tree or the whole divided subgraph can fit in the main memory such that the DFS-Tree can be computed directly in memory.

3) To address *challenge 3*, given a division G_0, G_1, $\cdots$, G_p of graph G, suppose the corresponding DFS-Trees are T_0, T_1, $\cdots$, T_p respectively, and the S-Graph is Σ, we design an efficient merge algorithm that can compute the DFS-Tree of graph G based on only T_0, T_1, $\cdots$, T_p and the S-Graph Σ without scanning the edges of the original graph on disk. The details of the merge algorithm are introduced in Section 7. Note that since the division process is conducted recursively, the merge process should also be conducted recursively in a reverse manner.

The Framework: We outline the framework of our divide & conquer algorithm, which is denoted as DivideConquerDFS (Algorithm 2). It first creates an initial spanning tree T by adding a virtual node γ and connecting it to all nodes in G, which is the same initial spanning tree created in EdgeByEdge and EdgeByBatch. Then it invokes an algorithm DivideConquer to recursively construct the DFS-Tree of the graph G in a divide & conquer manner.

DivideConquer is shown in line 4-18 of Algorithm 2. It takes a graph G, a spanning tree T of G, and the memory size M as input, and outputs a tree T which is a DFS-Tree of G, where G is stored on disk, and T is kept in memory. The algorithm first checks

Algorithm 2 DivideConquerDFS(graph G, memory size M)

```
1: for all nodes $v \in V(G)$ do
2:     add edge $(\gamma, v)$ in $T$ where $\gamma$ is a virtual node;
3: return DivideConquer($G, T, M$);

4: Procedure DivideConquer(graph $G$, tree $T$, memory size $M$)
5: if $|G| \leq M$ then
6:     return DFS-Tree $T$ of $G$ using an in-memory algorithm;
7: $dividable \leftarrow$ false;
8: while $dividable =$ false do
9:     $(T, update) \leftarrow$ Restructure($G, T, M$);
10:    if $update =$ false then
11:        return $T$;
12:    $(G_0, G_1, \cdots, G_p; T_0, T_1, \cdots, T_p; \Sigma) \leftarrow$ Divide($G, T, M$);
13:    if $p > 1$ then
14:        $dividable \leftarrow$ true;
15: for $i = 1$ to $p$ do
16:     $T_i \leftarrow$ DivideConquer($G_i, T_i, M$);
17: $T \leftarrow$ Merge($T_0, T_1, T_2, \cdots, T_p; \Sigma$);
18: return $T$;
```

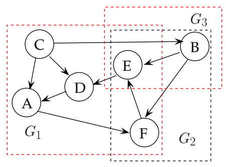
whether the graph G can fit in memory (line 5), if so, it simply loads the graph into memory, computes the DFS-Tree T of G using the in-memory algorithm, and returns T as the final result (line 6). Otherwise, the algorithm needs to compute the DFS-Tree of G by restructuring T or further divide G to construct the DFS-Tree in a divide & conquer manner. In order to do so, the algorithm uses a variable *dividable* to record whether a valid division of G can be found, which is initialized to be false (line 7). Then in a while loop (line 8-14), the algorithm tries to restructure the current spanning tree T w.r.t. G by invoking the same procedure Restructure used in EdgeByBatch (line 9) until T is a DFS-Tree of G (line 10-11) or a valid division of G is found based on the current spanning tree T (line 12-14). Here, the division is processed by invoking a Divide procedure to produce a graph division G_0, G_1, $\cdots$, G_p of G with corresponding spanning trees T_0, T_1, $\cdots$, T_p. The division procedure also computes the summary graph (S-Graph) Σ to be used in the merge procedure (line 12). The division is valid only if $p > 1$ (line 13-14). After a valid graph division is computed, in line 15-16, a DFS-Tree T_i is computed for each divided subgraph G_i by invoking the same procedure DivideConquer recursively. After that, the DFS-Tree T can be computed by merging all DFS-Trees T_i for the divided subgraphs G_i according to the S-Graph Σ (line 17), and returns T as the DFS-Tree of G.

Next, in Section 5, we discuss the properties for a valid graph division. In Section 6, we propose two graph division algorithms, namely, Divide-Star and Divide-TD, to be used in line 12 of Algorithm 2. In Section 7, we introduce the merge algorithm, to be used in line 17 of Algorithm 2.

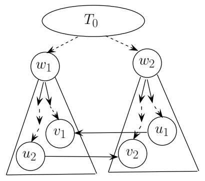
5. DIVISION PROPERTIES

We introduce the properties that a valid division of graph G should satisfy. As discussed above, a valid division should satisfy four properties, namely, node-coverage, contractible, Independence, and *DFS*-preservable. Let a valid division be G_0, G_1, $\cdots$, G_p of G, we define the four properties below.

- (*Node-Coverage*): The divided subgraphs jointly cover all nodes of G, i.e., $\bigcup_{0 \leq i \leq p} V(G_i) = V(G)$. This is to ensure that no node is missed after the division.
- (*Contractible*): The number of nodes in each subgraph G_i should be smaller than that of G, i.e., $|V(G_i)| < |V(G)|$ for any $0 \leq i \leq p$. This is to ensure that the division is meaningful.
- (*Independence*) Given two DFS-Trees T_i and T_j for G_i and G_j respectively $(1 \leq i < j \leq p)$, for any two nodes $u, v \in V(T_i) \cap V(T_j)$, u is an ancestor (descendant) of v in T_i iff u is an ancestor (descendant) of v in T_j. This requires the relationships of any pair of nodes in $V(T_i) \cap V(T_j)$ are consistent in T_i and T_j to ensure that each tree can be dealt independently.



(a) Independence



(b) DFS-Preservable

Figure 4: Illustration for Division Properties

- (*DFS-Preservable*) Given the DFS-Tree T_i of G_i for any $0 \le i \le p$, there exists a DFS-Tree T for graph G such that $V(T) = \bigcup_{0\le i\le p} V(T_i)$ and $E(T) = \bigcup_{0\le i\le p} E(T_i)$. This is to ensure the DFS-Tree for graph G can be constructed using the DFS-Trees of graphs G_i for $0 \le i \le p$.

Among the four properties for a valid division, the node-coverage property and the contractible property are trivial, thus, in the following, we assume that the node-coverage property and the contractible property are satisfied and then focus on the discussion of the independence property and DFS-preservable property.

5.1 Independence Property

The importance of the independence property is to ensure that the *DFS* in G_i does not affect the *DFS* in G_j for any $0 \le i < j \le p$. A trivial case for a division of G has the independence property is $V(G_i) \cap V(G_j) = \emptyset$ for any $0 \le i < j \le p$. However, when $V(G_i) \cap V(G_j) \neq \emptyset$, the independence property may not hold. For instance, node u is an ancestor of node v in the DFS-Tree of G_i, but is a sibling of node v in the DFS-Tree of G_j. We give a theorem to ensure the independence property of a division.

Theorem 5.1: *Given a division G_0, G_1, $\cdots$, G_p of a graph G, the independence property is satisfied if and only if, for any two subgraphs G_i and G_j ($0 \le i < j \le p$), $E(G_i) \cap E(G_j) = \emptyset$.* □

The proof sketch is given in Appendix.

Example 5.1: Fig. 4(a) shows a graph G. One division of G results in two graphs G_1 and G_2 as marked. We have $E(G_1) \cap E(G_2) = \{(E, F)\}$. In such a case, there exists a DFS-Tree T_1 of G_1 with *DFS* order C, A, F, E, D in which F is an ancestor of E, and there exists a DFS-Tree T_2 of G_2 with *DFS* order B, E, F in which F is a sibling node of E. The relationships of E and F in T_1 and T_2 are inconsistent. Therefore, such a division does not have independence property. However, for another division G_1 and G_3, it has the independence property since there exists only one common node E in G_1 and G_3, and thus the *DFS* on G_1 and G_3 will not affect each other. □

5.2 DFS-Preservable Property

We discuss the DFS-Preservable property, which ensures that if the DFS-Trees for all subgraphs have been found, the DFS-Tree for the whole graph can be computed just according to the existing DFS-Trees. In other words, after computing the DFS-Tree for each subgraph, it does not need to *DFS* the graph any more to find the DFS-Tree for the whole graph. In order to ensure the DFS-Preservable property, we need to guarantee that the union of the DFS-Trees for the divided subgraphs is a spanning tree/forest of G, which leads to the following lemma.

Lemma 5.1: *Given a division G_0, G_1, $\cdots$, G_p of G that satisfies the independence property, suppose the corresponding DFS-Trees for G_0, G_1, $\cdots$, G_p are T_0, T_1, $\cdots$, T_p respectively, then the union of T_0, T_1, $\cdots$, T_p is a spanning tree/forest of G only if, for any $0 \le i < j \le p$, one of the following conditions is satisfied:*

C_1: $V(T_i) \cap V(T_j) = \emptyset$;

C_2: $V(T_i) \cap V(T_j)$ is the root of T_i; and

C_3: $V(T_i) \cap V(T_j)$ is the root of T_j; □

The proof sketch is given in Appendix.

5.3 Root Based Valid Divisions

Together with all the four properties, we take a **root based division** approach in this work, as implied by the the DFS-Preservable property (Lemma 5.1). We explain the root based division. Given a graph G and a division $G_0, G_1, \cdots, G_p$ of G, let $T_0, T_1, \cdots, T_p$ be the corresponding DFS-Trees with roots $r_0, r_1, \cdots, r_p$, respectively, the root based division is a division if $V(G_0) \cap V(G_i) = \{r_i\}$ and r_i is a leaf node of T_0 for any $1 \le i \le p$, and $V(G_i) \cap V(G_j) = \emptyset$ for any $1 \le i < j \le p$.

It is important to note that there are two things to find a DFS-Tree of a graph G. First, there is DFS-Tree, and second, the DFS-Tree represents a total order following a *DFS*. In our root based division approach, we first construct a spanning tree with the same edge set as a DFS-Tree which we call DFS*-Tree. Then, we determine the exact order of nodes in the DFS*-Tree to make it as a DFS-Tree. It is worth noting that not every spanning tree of G is a DFS*-Tree and a DFS-Tree can be determined from a DFS*-Tree. We explain it using an example.

Example 5.2: For the graph G in Fig. 2(a), the spanning tree, represented by the solid lines, is not a DFS-Tree, since there exists a forward-cross edge (C, D) in G. However, it is a DFS*-Tree because if we visit the graph in the order A, E, D, B, C, F, G, H, I, and J, it will result in a DFS-Tree with the same edge set as the tree in Fig. 2(a). On the other hand, if we set the same order to nodes in the DFS*-Tree in Fig. 2(a) by swapping the subtrees rooted at E and B, then we will have the same DFS-Tree. □

Next, we show that with the properties, given all DFS-Trees T_i for subgraph G_i, for $0 \le i \le p$, the final DFS-Tree T must be a DFS*-Tree.

Lemma 5.2: *Given a graph G and a division G_0, G_1, $\cdots$, G_p of G, let T_0, T_1, $\cdots$, T_p be the DFS-Trees of G_0, G_1, $\cdots$, G_p respectively, and T be a graph with $V(T) = \bigcup_{0\le i\le p} V(T_i)$ and $E(T) = \bigcup_{0\le i\le p} E(T_i)$. Suppose the independence property of the division is satisfied, then the DFS-preservable property of the division is satisfied if and only if T is a DFS*-Tree of G.* □

Proof Sketch: The lemma can be easily proved according to the above discussion. □

Below, we further show how to check whether a spanning tree of a graph is a DFS*-Tree. We illustrate a simple case that cannot be a DFS*-Tree in Fig. 4(b). In Fig. 4(b), G contains a spanning tree T with two additional edges (u_1, v_1) and (u_2, v_2). Here, w_i represents a root of a spanning tree T_i for G_i for $i = 1, 2$. Given (u_1, v_1) and (u_2, v_2), T_1 and T_2 cannot satisfy the DFS-preservable property, if we treat them independently by division. We prove the general case in Lemma 5.3.

Lemma 5.3: *Given a graph G and a spanning tree T^* of G. T^* is a DFS*-Tree of G if and only if there do not exist edges (u_1, v_1), (u_2, v_2), $\cdots$, (u_k, v_k) and nodes w_1, w_2, $\cdots$, w_k such that for every $1 \le i \le k$, (u_i, v_i) is a cross edge, w_i is an ancestor of both v_i and u_{i+1} in T^*, and w_i and w_{i+1} do not have ancestor/descendant relationship in T^* (suppose $u_{k+1} = u_1$, and $w_{k+1} = w_1$, and a node is an ancestor/descendant of itself).* □

The proof sketch is given in Appendix.

Based on Lemma 5.3 and Lemma 5.2, we propose two division algorithms in Section 6. In the division algorithms, we make sure that we can find DFS-Tree T_i for subgraph G_i, for $(0 \le i \le q)$. By merging all DFS-Trees T_i, we will have a DFS*-Tree. The DFS*-Tree has the same edge set as a DFS-Tree for G, and we will discuss how to get DFS-Tree from DFS*-Tree in our merging algorithm in Section 7.

6. DIVISION ALGORITHMS

Given graph G, and a division $G_0, G_1, \cdots, G_p$, a key issue is to check whether the division is valid efficiently. In order to do so, we introduce a S-Graph to avoid checking a valid division by scanning the entire graph G. Then, we discuss how to construct T_0 based on which a valid division is identified. It is worth mentioning that T_0 is the tree that connects to all other trees T_i for $1 \leq i \leq p$.

6.1 S-Graph

Recall that checking whether a division is DFS-preservable can be done by checking the cross-edges in G (Lemma 5.3). However, it is inefficient if it needs to consider all cross-edges in the graph G which may not fit entirely in the main memory. In order to make it efficient, we define a small graph called S-Graph to capture the relationship of different subgraphs in the division. S-Graph is much smaller than the original graph G and can be used to efficiently check the DFS-preservable property of a root based division. Below, in order to define S-Graph, we first discuss S*-Graph.

Definition 6.1: (S*-Graph) Given a root based division G_0, G_1, $\cdots$, G_p of G, let r_i be the root of the corresponding DFS-Tree for G_i, for $0 \leq i \leq p$. A S*-Graph, denoted as Σ^*, is a graph by contracting each G_i ($1 \leq i \leq p$) to r_i. Here, $V(\Sigma^*) = V(G_0)$ in which a node can be a contracted node representing a subgraph G_i. There are 4 cases for all edges (u, v) in G. 1) If both u and v in G_0, (u, v) appears in Σ^*. 2) If $u \in G_0$ and $v \in G_i$ for $i \neq 0$, there is an edge (u, r_i) in Σ^*. 3) If $u \in G_i$ for $i \neq 0$ and $v \in G_0$. there is an edge (r_i, v) in Σ^*. 4) If $u \in G_i$ and $v \in G_j$, for $i \neq 0$, $j \neq 0$, and $i \neq j$, there is an edge (r_i, r_j) in Σ^*. □

Example 6.1: Fig. 5(a) shows a graph G and a spanning tree T (in solid lines). G is divided into 5 subgraphs: G_0, G_1, G_2, G_3, and G_4. Its S*-Graph Σ^* is shown in Fig. 5(b) where the node B, E, H, and K represents G_1, G_2, G_3, and G_4, respectively, and an S-edge is represented by a dotted line. For example, the edge (B, E) in Fig. 5(b) indicates there is at least one edge from G_1 to G_2. □

The following lemma shows that we can check whether a root based division is DFS-preservable using the S*-Graph only.

Lemma 6.1: *Given a graph G and a root based division G_0, G_1, $\cdots$, G_p of G, let T_0 be a spanning tree of G_0 and Σ^* be the* S*-Graph *w.r.t. the division, then the division is DFS-preservable w.r.t. T_0 if and only if T_0 is a* DFS*-Tree *of* Σ^*. □

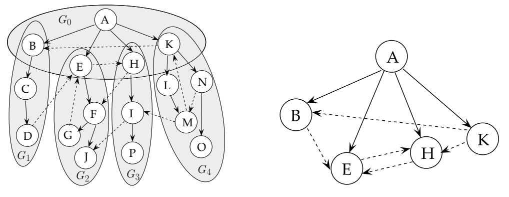
The proof sketch is given in Appendix.

The S*-Graph can largely reduce the search space for checking whether a root based division is DFS-preservable, because we do not need to scan the entire graph. However, checking the conditions in Lemma 5.3 is still costly based on S*-Graph. Next, we introduce S-Graph, based on which an efficient algorithm can be devised to check the validity of a division.

In order to define S-Graph, we do the following. First, we define a new operation pushup to push up the relationship implied by a cross-edge (u, v) to one of its parent in a spanning tree T, for example, (w, v), where w is the parent of u in T. Second, we show that the DFS-preservable property remains unchanged by pushing up a cross-edge. Third, we show that we can push up cross-edges as high as possible. Here, we use S-edge(u, v) to denote the resulting edge by pushing up edge (u, v) repeatedly until it cannot be pushed up any higher along T. And the S-Graph is the resulting graph by replacing all cross-edges with their S-edges.

Definition 6.2: (The operation pushup**)** Given a graph G and a spanning tree T of G, for an arbitrary cross-edge (u, v) w.r.t. T in G, let w be the parent node of u in T, if w is not an ancestor of v in T, then the pushup operation on u results in a new edge (w, v), denoted as pushup$((u, v), T, u) = (w, v)$, otherwise, pushup$((u, v), T, u) = (u, v)$. The operator pushup$((u, v), T, v)$ can be defined similarly. □

With the pushup operation, we show the resulting tree is still a DFS*-Tree.



(a) Graph G (b) S*-Graph (S-Graph)

Figure 5: A graph G and its S*-Graph (S-Graph)

Lemma 6.2: *Given a spanning tree T of G, for an arbitrary cross-edge (u, v) w.r.t. T in G, let G' be a graph by replacing (u, v) in G with either* pushup$((u, v), T, v)$ *or* pushup$((u, v), T, u)$*, then T is a* DFS*-Tree *of G if and only if T is a* DFS*-Tree *of G'.* □

The proof sketch is given in Appendix.

With Lemma 6.2 and Lemma 5.2, given a spanning tree T of G, replacing an arbitrary cross-edge (u, v) with either pushup$((u, v)$, $T, u)$ or pushup$((u, v), T, v)$ will keep the DFS-preservable property of a division unchanged. Thus we can iteratively push a cross-edge (u, v) with either pushup$((u, v), T, u)$ or pushup$((u, v), T, v)$ until we cannot push up any higher. We define it as S-edge(u, v).

Definition 6.3: (S-edge**):** Given a spanning tree T of G, for an arbitrary cross-edge (u, v) w.r.t. T in G, if we iteratively replace (u, v) with either pushup$((u, v), T, u)$ or pushup$((u, v), T, v)$ until pushup$((u, v), T, u) = (u, v)$ and pushup$((u, v), T, v) = (u, v)$, then the resulted edge is an S-edge(u, v). □

Consider Fig. 5(a), there is a cross-edge (H, F). By pushup$((H, F), T, F)$, a new S-edge (H, E) is generated in Fig. 5(b). The edge (H, E) is S-edge(H, F), because it cannot be pushed up any higher. With S-edges, we define S-Graph Σ below.

Definition 6.4: (S-Graph Σ**):** Given a root based division G_0, G_1, $\cdots$, G_p of graph G, let T_0 be the DFS-Tree of G_0 and Σ^* be the S*-Graph w.r.t. the division, the S-Graph Σ is a graph by removing all backward and forward edges w.r.t. T_0 in Σ^*, and replacing all cross-edges, (u, v), w.r.t. T_0 in Σ^* with their corresponding S-edges, S-edge(u, v). □

For the S*-Graph in Fig. 5(b), its S-Graph is the same as its S*-Graph. We have the following Lemma.

Lemma 6.3: *Given a spanning tree T of G, let G' be graph G by removing all backward and forward edges w.r.t. T and replacing all cross-edges w.r.t. T in G with their* S-edges*, then T is a* DFS*-Tree *if and only if G' is an directed acyclic graph (DAG).* □

The proof sketch is given in Appendix.

It is important to note that checking whether a tree T is a DFS*-Tree of G is very complicated using Lemma 5.3 because it requires checking an exponential number of combinations of cross-edges and ancestor nodes. With Lemma 6.3, we can check it by checking whether the S-Graph is a DAG. We show a theorem below.

Theorem 6.1: *Given a root based division G_0, G_1, $\cdots$, G_p of graph G, let T_0 be a spanning tree of G_0 and Σ be the* S-Graph *w.r.t. the division and T_0, then the division is DFS-preservable w.r.t. T_0 if and only if Σ is a DAG.* □

The theorem can be derived directly from Lemma 6.3 and Lemma 6.1.

Theorem 6.1 provides an efficient way to check whether a division is DFS-preservable, which can be done by checking whether the S-Graph is a DAG. Based on Theorem 6.1, we discuss two division algorithms in the next two subsections.

6.2 Division Algorithm Divide-Star

In this subsection, we discuss our division algorithm. Theorem 6.1 indicates a way to check whether a division is valid by checking whether the S-Graph Σ w.r.t. the division is a DAG. However, there are two problems to be solved in order to find a valid root based

Algorithm 3 Divide-Star(graph G, tree T, memory size M)

```
1: $r_0 \leftarrow$ the root of $T$;
2: $\{r'_1, r'_2, \cdots, r'_{p'}\} \leftarrow$ the child nodes of $r_0$ in $T$;
3: create a graph $\Sigma$ with $V(\Sigma) = \{r_0, r'_1, \cdots, r'_{p'}\}$ and $E(\Sigma) = \{(r_0, r'_1), (r_0, r'_2), \cdots, (r_0, r'_{p'})\}$;
4: for all edge $(u, v) \in E(G)$ in sequential order on disk do
5:     $w \leftarrow$ the LCA of $u$ and $v$ in $T$;
6:     if $(u, v)$ is a cross-edge and $w = r_0$ then
7:         $(r'_i, r'_j) \leftarrow$ S-edge$(u, v)$;
8:         $E(\Sigma) \leftarrow E(\Sigma) \cup \{(r'_i, r'_j)\}$;
9: if $\Sigma$ is not a DAG then
10:     for all SCC $S$ in $\Sigma$ do
11:         if $|S| > 1$ then
12:             modify $T$ and $\Sigma$ using the node contraction operation w.r.t. $S$;
13: $T_0 \leftarrow$ a star with root $r_0$ and child nodes $\{r_1, r_2, \cdots, r_p\}$ which are the same with the child nodes of $r_0$ in $T$;
14: for $i = 1$ to $p$ do
15:     $T_i \leftarrow$ the subtree rooted at $r_i$ in $T$;
16: for $i = 0$ to $p$ do
17:     $G_i \leftarrow$ the subgraph induced by nodes in $T_i$;
18: return $(G_0, G_1, \cdots, G_p; T_0, T_1, \cdots, T_p; \Sigma)$;
```

division G_0, G_1, $\cdots$, G_p: (1) How to find a division such that the corresponding S-Graph is a DAG? and (2) If the S-Graph is not a DAG, can we still produce a new valid division based on the current division and the S-Graph Σ? In order to solve the above two problems, we first analyze a simple case: the spanning tree T_0 of G_0 in theorem 6.1 is a star (a tree of depth 1). We denote such a division method as Divide-Star.

Recall that in the algorithm framework (Algorithm 2), the division algorithm takes the current in-memory spanning tree T of G as input and outputs a division G_0, G_1, $\cdots$, G_p, their corresponding subtrees in T, as well as the S-Graph Σ (see line 12 of Algorithm 2). Suppose the root of T is r'_0 and r'_0 has p' child nodes $r'_1, r'_2, \cdots, r'_{p'}$ in T, a straightforward division can be designed as follows: Let T'_0 be a star rooted at r'_0 with p' child nodes $r'_1, r'_2, \cdots, r'_{p'}$, and T'_i ($1 \le i \le p'$) be the subtree of T rooted at r'_i. G'_i ($0 \le i \le p'$) is a subgraph induced by nodes in T'_i.

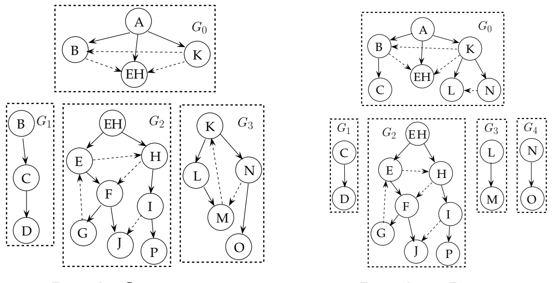
Given such a division, the corresponding S-Graph Σ can be constructed as follows: Initially, Σ contains nodes $r'_0, r'_1, \cdots, r'_{p'}$, and edges from r'_0 to each r'_i ($1 \le i \le p'$). Then for each cross-edge (u, v) w.r.t. T, if the LCA of u and v in T is r'_0, we can find nodes r'_i and r'_j in Σ such that S-edge$(u, v) = (r'_i, r'_j)$, and we add the edge (r'_i, r'_j) in Σ.

After constructing Σ, if the graph Σ is a DAG, then we find a valid graph division G'_0, G'_1, $\cdots$, $G'_{p'}$ according to theorem 6.1, and Divide-Star is finished. However, when Σ is not a DAG, according to theorem 6.1, the division $G'_0, G'_1, \cdots, G'_{p'}$ is not DFS-preservable. In such a situation, we need to find a way to eliminate the cycles in Σ to make it a DAG. In order to do this, we modify the tree T, S-Graph Σ, and the division based on SCC-aware graph division which is introduced as follows.

SCC-Aware Graph Division: Given a spanning tree T of G, a division of G and the corresponding S-Graph Σ, if Σ is not a DAG, then *SCC-aware graph division* modifies T, Σ, and the current division as follows: We compute all strongly connected components (SCCs) in Σ, and for each SCC S in Σ, if it contains multiple nodes $S = \{r'_{i_1}, r'_{i_2}, \cdots, r'_{i_k}\}$, then we apply the *node contraction operation* w.r.t. S to modify T, Σ, and the division as follows:

- *Modifying T:* We add a virtual node s, an edge (r'_0, s), and edges (s, r'_{i_1}) $(s, r'_{i_2}), \cdots, (s, r'_{i_k})$ in T, and remove the edges (r'_0, r'_{i_1}) $(r'_0, r'_{i_2}), \cdots, (r'_0, r'_{i_k})$ from T.

- *Modifying Σ:* In the S-Graph Σ, we add a virtual node s. For any edge (u, v) in Σ where $u \in S$ and $v \notin S$, we add a new edge (s, v) in Σ, and for any edge (u, v) in Σ where $u \notin S$ and $v \in S$, we add a new edge (u, s) in Σ, and we remove all nodes in S and their corresponding edges from Σ.



(a) Divide-Star Example (b) Divide-TD Example

Figure 6: Examples for Divide-Star and Divide-TD

- *Modifying Division:* For the division, we create a new tree T' rooted at the virtual node s by connecting s to the roots of the trees T'_{i_1}, T'_{i_2}, $\cdots$, T'_{i_k}. Then we remove subgraphs G'_{i_1}, G'_{i_2}, $\cdots$, G'_{i_k} from the division, and add a new subgraph which is the subgraph induced by nodes in T' in the division.

By applying the above node contraction operations for all SCCs in Σ, SCC-aware graph division will result in a new spanning tree T, a new S-Graph Σ, and a new division G_0, G_1, $\cdots$, G_p. Obviously, such a division is a valid division since its corresponding S-Graph Σ is a DAG. However, the spanning tree T is not a spanning tree of the original graph G since some virtual nodes are added in T. In Section 7, we will discuss how to handle such virtual nodes in the merge algorithm in order to compute the DFS-Tree of the original graph G according to a SCC-aware graph division.

Example 6.2: For the graph G shown in Fig. 5 (a), a spanning tree T is shown in solid edges. A straightforward division divides graph G into five subgraphs, G'_0, G'_1, $\cdots$, G'_4 where G'_0 is the subgraph induced by the star with five nodes A, B, E, H, and K, and G'_i ($1 \le i \le 4$) are the subgraphs induced by the nodes in the subtrees rooted at B, E, H, and K respectively. The corresponding S-Graph Σ is shown in Fig. 5 (b). Obviously, Σ is not a DAG since it contains an SCC $\{E, H\}$, thus such a division is not DFS-preservable. However, if we contract the two nodes E and H by adding a virtual node EH, the new division is shown in Fig. 6 (a) which contains only four subgraphs, G_0, G_1, G_2, and G_3, where G_2 is the subgraph induced by the union of nodes in the above subgraphs G'_2 and G'_3. The new Σ is the same as G_0 in Fig. 6 (a). Obviously, the new division is DFS-preservable. □

The Divide-Star Algorithm: According to the above discussion, our algorithm Divide-Star is shown in Algorithm 3. Initially, we get the root r_0 of T, as well as its child nodes, and initialize the S-Graph Σ as a star rooted at r_0 (line 1-3). Then we construct Σ by scanning all edges of G on disk (line 4). For each scanned edge (u, v), we compute the LCA w of u and v. If w is r_0 and (u, v) is a cross-edge w.r.t. T, then the S-edge of (u, v) should be included in the S-Graph Σ (line 5-8). Next, we check whether Σ is a DAG, if not, we need to modify T and Σ using the node contraction operations introduced above for each SCC S in graph Σ (line 9-12). Then, we start to divide the graph. We first construct T_0 which is a star with root be r_0 and leaves be the child nodes of r_0 in T (line 13). After constructing T_0, each T_i ($1 \le i \le p$) is a subtree rooted at a leaf node of T_0 (line 14-15). Finally, we compute the induced subgraph of nodes in T_i ($0 \le i \le p$) as the subgraph G_i in the division, and return the divided subgraphs, the corresponding subtrees, as well as the S-Graph Σ as the result.

Example 6.3: A graph G is shown in Fig. 5 (a) with a spanning tree T marked in solid lines. By applying the Divide-Star algorithm (Algorithm 3), the division is shown in Fig. 6 (a) with four subgraphs G_0, G_1, G_2, G_3. For each G_i, the corresponding T_i which is a subtree of T is shown in solid lines. For simplicity, the graph G_0 is shown the same as the S-Graph Σ , which is a DAG by contracting the SCC $\{E, H\}$ into a virtual node EH. Three S-edges in Σ are (B, EH), (K, EH), and (K, B). □

6.3 Division Algorithm Divide-TD

Recall that the aim of the division algorithm is to maximize the number of divided subgraphs. In the Divide-Star algorithm, given a graph G and a spanning tree T, the division is computed based on a star structure T_0 with the same root as T. Such a division algorithm has two drawbacks. First, the S-Graph Σ is computed on top of T_0 which is constructed based on only one level of nodes in T. When the root r_0 of T has a small number of child nodes, or the relationship of the subgraphs induced by the subtrees rooted at the child nodes of r_0 is complex, after computing the division by contracting all SCCs, it is possible that only few divided subgraphs are left. Second, the S-Graph Σ is created by scanning the graph G on disk once and compute the set of S-edges (r'_i, r'_j) with both r'_i and r'_j be the child nodes of r_0 in T, however, when the number of child nodes of r_0 is small, the number of such S-edges is small comparing to the number of all possible S-edges existing in the graph. Therefore, a large number S-edges are computed but not used when scanning the edges. This is obviously a waste of I/Os.

According to the above discussion, the key issue to overcome the above drawbacks is to enlarge the size of T_0 and the corresponding S-Graph Σ as long as they can fit in the main memory. In order to do so, instead of considering only one level of nodes in T, we can consider several levels of nodes in T to construct T_0 and the corresponding S-Graph Σ. This motivates us to explore a multi-level subtree of T denoted as a cut-tree which is defined as follows.

Definition 6.5: (Cut-Tree) Given a tree T with root t_0, a *cut-tree* T_c is a subtree of T which satisfies two conditions: (1) the root of T_c is t_0, and (2) for any node v, suppose the child nodes of v in T are $v_1, v_2, \cdots, v_k$, if $v \in V(T_c)$, then v is either a leaf node in T_c, or a node in T_c with child nodes $v_1, v_2, \cdots, v_k$. □

The reason for condition (2) in the above definition is that, for any S-edge (w_u, w_v), we need to make sure either both w_u and w_v belong to $V(T_c)$ or none of w_u and w_v belong to $V(T_c)$, such that the corresponding S-Graph can capture the relationship of divided subgraphs connected using S-edges.

Given a spanning tree T, and a cut-tree T_c, we can construct T_0 and the S-Graph Σ based on T_c. The question is, given the memory budget M', how to construct T_c such that the size of T_c is maximized and the corresponding S-Graph Σ can fit in memory. In the following, we discuss this issue.

Cut-Tree Construction: Given a tree T with root r_0 and the memory budget M', the cut-tree T_c can be constructed as follows. Initially, T_c contains one node r_0. Then, we iteratively pick a leaf node v from T_c and all child nodes of v in T as the child nodes of v in T_c. Note that the corresponding S-Graph w.r.t. T_c contains at most $|V(T_c)|^2$ edges. Therefore, the process stops when after adding the next node, $|V(T_c)|^2 > M$. Here, for simplicity, we assume that each edge consumes one unit of memory.

Note that after constructing T_c, we can compute a S-Graph Σ with $V(\Sigma) = V(T_c)$, however, when Σ is not a DAG, we still need to use the SCC contraction technique used in the SCC-aware graph division (see Section 6.2) to make it a DAG. However, in such a case, after we construct T_0 based on the new S-Graph Σ and T_c, we need to make sure that each virtual node in T_0 is a leaf node of T_0, because each virtual node represents a SCC with multiple nodes in the original Σ, and the nodes in an SCC cannot be further divided. Based on such an idea, we construct T_0 in a top-down manner based on T_c, and design our algorithm Divide-TD which is introduced below.

The Divide-TD **Algorithm:** The Divide-TD algorithm is shown in Algorithm 4. It first computes a cut-tree T'_0 of T using the above discussed method, and initializes Σ to be T'_0 (line 1-2). Then it scans all edges (u, v) in G on disk and add $(w_u, w_v) = $ S-edge(u, v) into Σ if both w_u and w_v belong to $V(T'_0)$ (line 3-7). In line 8-11, the algorithm contracts all SCCs in Σ using the same method used in Algorithm 3. The top-down process to construct T_0 is shown in line 12-21. After initilizing T_0 and a FIFO queue Q (line 12-13), it first pushes the root r_0 of T into Q (line 14). Then the algorithm iteratively adds edges into T_0 until Q becomes $\emptyset$. In each iteration, it first gets the top node u in Q (line 16), and adds all child nodes v of u into T_0 if u is not a virtual node and v is in the tree T'_0 (line 17-20). For each such v added into T_0, it is pushed into Q for further expansion (line 21). After computing T_0, the S-Graph Σ is updated by removing all nodes that are not in $V(T_0)$ from Σ. Finally, the subtrees $T_1, T_2, \cdots, T_p$ and the divided subgraphs $G_0, G_1, \cdots, G_p$ are computed similarly to those computed in Algorithm 3.

Algorithm 4 Divide-TD(graph G, tree T, memory size M)

```
1:  $T'_0 \leftarrow$ a cut-tree of $T$;
2:  $\Sigma \leftarrow T'_0$;
3:  for all edge $(u, v) \in E(G)$ in sequential order on disk do
4:      $w \leftarrow$ the LCA of $u$ and $v$ in $T$;
5:      if $(u, v)$ is a cross-edge and $w$ is a non-leaf node in $V(T'_0)$ then
6:          $(w_u, w_v) \leftarrow$ S-edge$(u, v)$;
7:          $E(\Sigma) \leftarrow E(\Sigma) \cup \{(w_u, w_v)\}$;
8:  if $\Sigma$ is not a DAG then
9:      for all SCC $S$ in $\Sigma$ do
10:         if $|S| > 1$ then
11:             modify $T$ and $\Sigma$ using the node contraction operation w.r.t. $S$;
12: $r_0 \leftarrow$ the root of $T$;
13: $T_0 \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
14: $Q.push(r_0)$;
15: while $Q \neq \emptyset$ do
16:     $u \leftarrow Q.pop()$;
17:     if $u$ is not a virtual node then
18:         for all child node $v$ of $u$ in $T$ do
19:             if $v \in V(T'_0)$ then
20:                 add edge $(u, v)$ into $T_0$;
21:                 $Q.push(v)$;
22: remove nodes that are not in $V(T_0)$ and their corresponding edges from $\Sigma$;
23: $\{r_1, r_2, \cdots, r_p\} \leftarrow$ the leaf nodes of $r_0$ in $T_0$;
24: for $i = 1$ to $p$ do
25:     $T_i \leftarrow$ the subtree rooted at $r_i$ in $T$;
26: for $i = 0$ to $p$ do
27:     $G_i \leftarrow$ the subgraph induced by nodes in $T_i$;
28: return $(G_0, G_1, \cdots, G_p; T_0, T_1, \cdots, T_p; \Sigma)$;
```

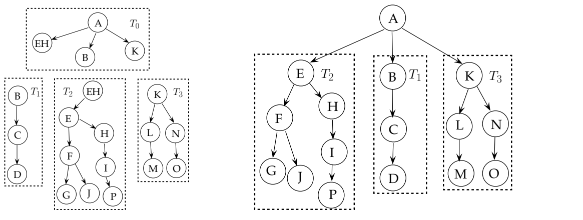
Remark 6.1: *For both* Divide-Star *and* Divide-TD*, the input graph G can be a general directed graph. According to Theorem 6.1, in order to compute a valid division for G, we need to make the* S-Graph Σ *to be a DAG using the contraction operations by computing all* SCC*s of* Σ. *Recall that* Σ *is a light-weight memory-resident graph, therefore, we can apply any in-memory* SCC *computation algorithm to compute the DAG.* □

Example 6.4: A graph G is shown in Fig. 5 (a) with a spanning tree T marked in solid lines. Suppose a cut-tree T'_0 of T consists of all nodes in T with depth no larger than 2 (suppose the depth of root A is 0). Obviously, when constructing the S-Graph Σ based on T'_0, nodes E and H belong to the same SCC. Consequently, when we construct T_0 based on T'_0 and S-Graph, the decedent nodes of E and H in T'_0 will not be contained in T_0. The new S-Graph Σ is the graph G_0 shown in Fig. 6 (b), and the corresponding tree T_0 is shown in the solid lines in graph G_0. Based on T_0 and Σ, the division results in 5 subgraphs as shown in Fig. 6 (b). Comparing to the division in Fig. 6 (a), the subgraph G_3 is further divided into two smaller subgraphs G_3 and G_4 in Fig. 6 (b). □

Discussion: Divide-TD is an improvement of Divide-Star by considering multiple levels of nodes in the spanning tree T for each round of scanning of the graph G. Divide-TD is a generalization of Divide-Star by making full use of the memory whenever possible. Therefore, Divide-TD outperforms Divide-Star, as confirmed by our experiments in Section 8.

Algorithm 5 Merge(subtrees $T_0, T_1, \cdots, T_p$, S-Graph Σ)

```
1: $T \leftarrow T_0$;
2: topological sort all nodes in $\Sigma$;
3: reorder all nodes in $T$ according to the reverse topological order of the corresponding nodes in $\Sigma$;
4: for $i = 1$ to $p$ do
5:     combine $T_i$ into $T$;
6: for all virtual node $v \in V(T)$ do
7:     $u \leftarrow$ parent node of $v$ in $T$;
8:     remove $(u, v)$ from $T$;
9:     for all child node $w$ of $v$ in $T$ do
10:        remove $(v, w)$ from $T$; add $(u, w)$ into $T$;
11: return $T$;
```



(a) Merge Example (b) Merge Result

Figure 7: Example for Merge

7. MERGE ALGORITHM

In this section, we discuss the merge algorithm. Recall that the Merge procedure in Algorithm 2 takes the divided DFS-Trees T_0, $T_1, \cdots, T_p$, and the corresponding S-Graph Σ as input and outputs a DFS-Tree of the graph G. Two problems need to be solved in Merge. (1) How to organize the DFS-Trees $T_0, T_1, \cdots, T_p$ in the merged tree T such that T is a DFS-Tree of graph G? and (2) How to handle virtual nodes in the DFS-Trees $T_0, T_1, \cdots, T_p$?

(1) In order to address the first problem, we need to use the information contained in the S-Graph Σ. Recall that Σ is a graph that maintains the topology of the edges across different divided subgraphs, and in our division algorithm, we make sure that Σ is a DAG and $V(\Sigma) = V(T_0)$. Therefore, we can simply topological sort all nodes in Σ and reorder nodes in T_0 according to the reverse topological order of the corresponding nodes in Σ, and then combine all T_i $(1 \leq i \leq p)$ with T_0 to get the DFS-Tree of G. Here we use the reverse topological order because we need to avoid forward-cross edges w.r.t. the DFS-Tree.

(2) In order to handle the second problem, suppose by combining all DFS-Trees $T_0, T_1, \cdots, T_p$, we get a tree T. For each virtual node $v \in V(T)$, suppose the parent node of v in T is u, then we remove edge (u, v) from T, and for each child node w of v in T, we remove edge (v, w) from T and add a new edge (u, w) into T. It is easy to verify that the resulted tree T is a DFS-Tree of G.

Our Merge algorithm is shown in 5 and is self explained according to the above discussion.

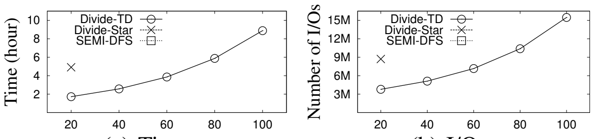
Example 7.1: Suppose we divide the graph G shown in Fig. 5 (a) using the Divide-Star algorithm (Algorithm 3), the four divided subgraphs are shown in Fig. 6 (a), where the S-Graph Σ is the same as G_0. By topological sorting Σ, the reverse topological order for the three leaf nodes are EH, B, and K. After reordering nodes in T_0 accordingly, the subtrees T_0, T_1, T_2 and T_3 are shown in Fig. 7 (a). After combing them into T, a virtual node EH exists in T with parent node A and a child node E. Thus, we remove edges (A, EH) and (EH, E) from T and add a new edge (A, E) into T. The final T is shown in Fig. 7 (b), which is obviously a DFS-Tree of the original graph G. □

8. PERFORMANCE STUDIES

In this section, we show the experimental results by comparing our results with the SEMI-DFS approach [14]. Our algorithms are all based on divide & conquer. The algorithm Divide-Star is to find the DFS-Tree by dividing the graph according to the children of the root. Divide-TD is to divide the graph into smaller subgraphs in a top-down manner. All the algorithms are implemented using Visual C++ 2010 and tested on a PC with Intel Core2 Quard 2.66GHz CPU and 3.50GB memory running Windows 7 Enterprise. The disk block size is 64KB. In all experiments, we report the processing time and the number of I/Os, which include the cost spent on all operations to compute the DFS-Tree from the original graph. We set the max time cost to be 8 hours. If a test does not stop in the time limit, the result will not appear in the corresponding figure.

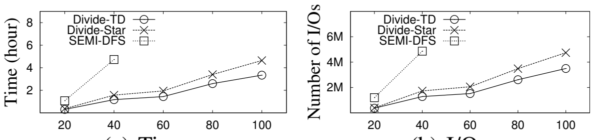
Parameter	Range	Default
Size of $\|V\|$	30M, 40M, 50M, 60M, 70M	50M
Average Degree D	3, 4, 5, 6, 7	5
Power-Law-Ness $\|A\|/D$	0.25, 0.5, 1, 2, 4	1
Memory Size M(GB)	0.5, 0.75, 1, 1,25, 1,5	1

Table 1: Range and Default Value for Parameters



(a) Time (b) I/Os

Figure 8: Webgraph Results (Varying Percentage)



(a) Time (b) I/Os

Figure 9: Twitter-2010 Results (Varying Percentage)

Datasets: We use 4 real massive datasets, and several synthetic datasets. The real datasets are: wikilink, arabic-2005, twitter-2010, and webspam-uk2007. The wikilinks [15] is a large-scale cross-document coreference corpus labeled via links to wikipedia[5]. It contains 25,942,246 nodes and 601,038,301 edges, with average degree 23.16 per node. Arabic-2005 is a 2005 crawl for websites that contain pages written in Arabic[6]. The graph of arabic-2005 contains 22,744,080 nodes and 639,999,458 edges, with average degree 28.14 per node. Twitter-2010 remembers the direction of tweets transmission[7]. Nodes are users and there is an edge from u to v if v is a follower of u. It contains 41,652,230 nodes and 1,468,365,182 edges. Comparing to wikipedia and arabic-2005, the twitter-2010 is a hard-to-compress dataset, and the dataset contains a huge SCC with 33,479,734 nodes, which is 80.4% to the size of the whole graph. For webspam-uk2007[8], it consists of 105,896,555 webpages in 114,529 hosts in the .UK domain. The graph contains 105,895,908 nodes and 3,738,733,568 edges, with the average degree 35 per node. For the four real datasets used in our experiments, namely, wikilink, arabic-2005, twitter-2010, and webspam-uk2007, they consume 4.6 GB, 4.8 GB, 11.4 GB, and 29.2 GB space on disk respectively to store only the topological structure of the graphs. For these datasets, we set the memory as $2GB$.

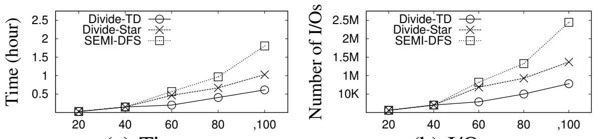
We also generate synthetic graph datasets in order to test the scalability of our approaches. Given node size $|V|$ and average degree D, the first is random graph. We randomly generate a node pair and add to the graph until the number of edges is $D \cdot |V|$. The second is power-law graph. We generate it using a preference attachment method in [7]. Parameter A (the "power-law-ness") is used in the graph generation with $|V|$ and D. The smaller the $\frac{|A|}{D}$ is, the smaller the fraction of high-degree nodes is in all nodes. The default memory size M is $1GB$ for these datasets. The range and default value of all parameters are shown in Table 1.

[5] http://www.iesl.cs.umass.edu/data/wiki-links

[6] http://law.di.unimi.it/webdata/arabic-2005/

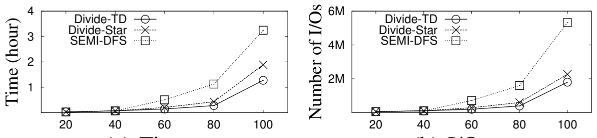
[7] http://law.di.unimi.it/webdata/twitter-2010/

[8] http://barcelona.research.yahoo.net/webspam/datasets/uk2007/links/



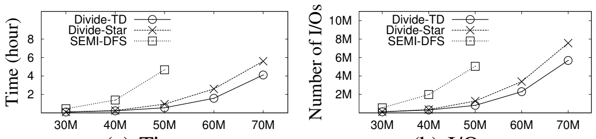
(a) Time (b) I/Os

Figure 10: Wikilink Results (Varying Percentage)



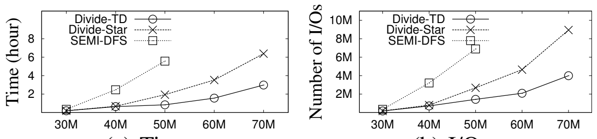
(a) Time (b) I/Os

Figure 11: Arabic Results (Varying Percentage)



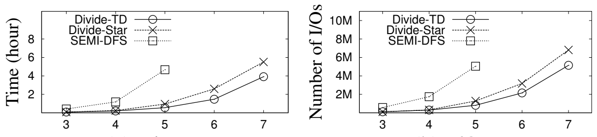
(a) Time (b) I/Os

Figure 12: Power-Law Graph (Varying Node Size)



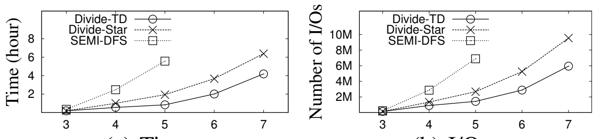
(a) Time (b) I/Os

Figure 13: Random Graph (Varying Node Size)



(a) Time (b) I/Os

Figure 14: Power-Law Graph (Varying Degree)



(a) Time (b) I/Os

Figure 15: Random Graph (Varying Degree)

Exp-1 (Performance on Real Large Datasets): We test our approaches on 4 real datasets, webspam-uk2007, twitter-2010, wikilinks and arabic-2005. For each dataset, we randomly select edges from E and vary the size of edges $|E|$ from 20% to 100% for all graphs as shown in x-axis. The results of time and I/O cost for the datasets webspam-uk2007, twitter-2010, wikilinks, and arabic-2005 are shown in Fig. 8, Fig. 9, Fig. 10, and Fig. 11 respectively. Among all the approaches, Divide-TD performs best in all cases. Divide-Star has better performance than SEMI-DFS, but will cost more than Divide-TD.

While increasing $|E|$, the cost for all the approaches increases. Divide-TD has the slowest increasing rate followed by Divide-Star. The reason is, when $|E|$ becomes larger, the cost will increase as it needs to scan and restructure more edges in each iteration. However, Divide-TD can result in more subgraphs after division. Once the graph is divided, each subgraph will be smaller and more edges can be loaded in. Then the cost to find the DFS-Tree for each subgraph will be smaller compared with the original graph. Thus Divide-TD has the best performance, but SEMI-DFS costs most.

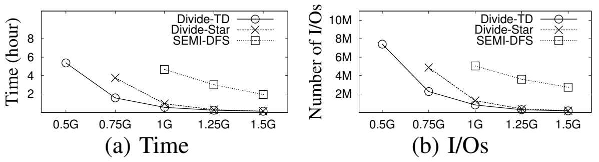
For the graph webspam-uk2007, SEMI-DFS cannot find the DFS-Tree even when the graph size is 20% of the original graph. However, Divide-TD can find the DFS-Tree in limited time even the graph size is 100% of the original graph. For the graph of twitter-2010, Divide-TD performs best and SEMI-DFS can only work out in limited time when the graph size is at most 40% of the original graph. For the graph of wikilink, Divide-TD also performs best. When the graph size is less than 40% percentage, all approaches have the same cost because the whole graph can reside in the main memory. The main cost is to load the graph from disk into main memory and the external-memory stack used in the *DFS* procedure. For the graph of arabic-2005, its cost is small when the graph size is less than 80% percentage, but increases sharply when it reaches 100% of the original graph. The reason is that, if $|E|$ is less than 80% of the original graph, most or all edges can reside in the memory. When 20% more edges are added, these edges are all kept on disk. Divide-TD has a marginal increasing rate because it divides the graph into subgraphs and for each subgraph the number of edges is smaller compared with other approaches. SEMI-DFS has the largest increasing rate as it takes the original graph as a whole and needs more iterations to find the DFS-Tree.

Exp-2 (Vary Node Size $|V|$ in Synthetic Data): We vary the number of nodes $|V|$ from 30M to 70M. The results of time and I/O cost for power-law graphs are shown in Fig. 12(a) and Fig. 12(b) respectively. Similarly, the results of random graphs are shown in Fig. 13(a) and Fig. 13(b). While increasing $|V|$, the time and I/O cost of all algorithms increase. SEMI-DFS cannot find the DFS-Tree in limited time when $|V|$ is larger than 50M. Among all the algorithms, Divide-TD has the lowest increasing rate and SEMI-DFS has the highest increasing rate. Divide-Star has higher increasing rate than Divide-TD, but still performs better than SEMI-DFS, because Divide-Star and Divide-TD are based on divide & conquer approach, and although $|V|$ increases, after division, our algorithms deal with smaller graphs. When the node size is small, Divide-Star and Divide-TD have similar performance. This is because that the memory can hold more than half of the original graph. Even dividing the graph according to the children of root by Divide-Star, after division, the memory can hold a large part of each subgraph and the procedure to find the DFS-Tree for each subgraph is fast. However, without division, SEMI-DFS still needs much more time even when the memory can keep a large part of the graph.

When $|V|$ becomes larger, Divide-Star has a higher increasing rate compared with Divide-TD. The reason is that, it is divided according to the children of the root. In this scenario, it will result in a smaller number of subgraphs and the size for each subgraph is larger. Compared with random graphs and power-law graphs, Divide-Star will have a higher increasing rate in random graph because in random graph, the distribution of edges is even. It will result in several large subgraphs. The largest subgraph after division is still quite large which makes the cost for Divide-Star high.

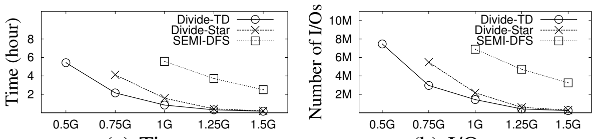
Exp-3 (Vary Average Degree in Synthetic Data): We vary the average degree of graphs from 3 to 7 and we test on power-law and random graphs. The default node size is 50,000,000. The results of time and I/O cost for power-law graphs are shown in Fig. 14(a) and Fig. 14(b) respectively and the results for random graphs are shown in Fig. 15(a) and Fig. 15(b). When the average degree is larger than 5, SEMI-DFS cannot find the DFS-Tree in limited time. With larger degree, the time and I/O cost for all these algorithms increase. SEMI-DFS has the largest increasing rate and Divide-TD has the smallest. With more edges, divide & conquer approaches need more iterations. However, Divide-Star and Divide-TD outperform SEMI-DFS. This is because after division, all these approaches only need to deal with smaller subgraphs. The divide & conquer approaches are stable when the number of edges increases.

Exp-4 (Vary Memory Size in Synthetic Data): We vary the memory size from 0.5GB to 1.5GB and we test on power-law and random graphs. The graph tested in this experiment has 50,000,000



(a) Time (b) I/Os

Figure 16: Power-Law Graph (Varying Memory Size)



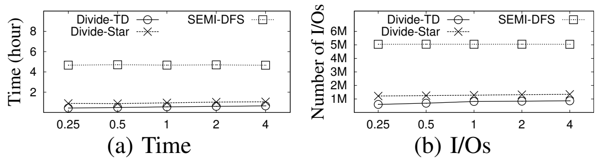
(a) Time (b) I/Os

Figure 17: Random Graph (Varying Memory Size)

nodes and 250,000,000 edges. The results of time and I/O cost for power-law graphs are shown in Fig. 16(a) and Fig. 16(b) respectively and the results for random graphs are shown in Fig. 17(a) and Fig. 17(b). When the memory size is less than 1GB, SEMI-DFS cannot find the DFS-Tree in limited time for both graphs. With more memory, the cost for Divide-TD decreases sharply. The reason is that with more memory, the corresponding S-Graph has more nodes and edges and the graph will be divided into more subgraphs according to the larger S-Graph. For Divide-Star approaches, its decreasing rate will be slower than Divide-TD. The reason is that the size for S-Graph cannot be tuned according to the memory size in these two approaches, so the number of subgraphs will not increase a lot with larger memory. All divide & conquer approaches perform much faster than SEMI-DFS. The gap between SEMI-DFS and our approaches becomes larger with smaller memory. This is because when the memory size is not large, the number of edges that can be loaded in addition to the spanning tree is small. In this scenario, if the graph can be divided, the spanning tree that needs to be kept in the memory can be smaller. Then less iteration and less cost are needed to find the corresponding DFS-Tree.

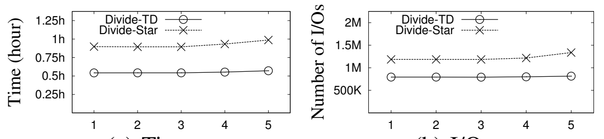
Exp-5 (Vary Power-Law-Ness A in Power-Law Graphs): We vary the parameter A in power-law graphs. The more the $\frac{|A|}{D}$ is, the more the fraction of high-degree nodes is among all nodes. We vary A from $0.25D$ to $4D$. D is average degree and it is 5 in this testing. The graph being tested has 50,000,000 nodes and the time and I/O cost are shown in Fig. 18(a) and Fig. 18(b) respectively. With larger A value, divide & conquer approaches will increase slightly. The reason is that larger A indicates that more nodes have high degree. In this scenario, dividing the nodes with high degree will cost more than dividing those with small degree. So the cost will increase. However, the increasing rate is quite slow. For the SEMI-DFS approach, it increases with larger A. The reason is that more nodes with high degree indicates larger intermediate results in *DFS* procedure. Such intermediate results are kept on disk and results in an increase of I/O cost.

Exp-6 (Vary Start Node): In this experiment, we test the effect of the start node to the performance of Divide-Star and Divide-TD. We use the synthetic power-law graph by setting all parameters to their default values. We divide the nodes evenly into 5 partitions according to their degrees, such that nodes in partition i ($1 \leq i \leq 4$) have degrees no larger than nodes in partition $i + 1$. In each partition, we randomly select one node as the start node, and we repeat each test for 10 times and take the average. The results for processing time and I/O cost are shown in Fig. 19(a) and Fig. 19(b) respectively. When the degree of start node increase, the processing time and I/O cost for Divide-Star increase very slightly because when the degree of the start node is larger, computing the S-Graph is more costly, but cannot be the dominant cost. The performance of Divide-TD is not sensitive to the degree of the start node because it considers multiple levels of nodes in each round whose size is dependent on the size of the memory.



(a) Time (b) I/Os

Figure 18: Power-Law Graph (Varying $|A|/D$)



(a) Time (b) I/Os

Figure 19: Power-Law Graph (Varying Start Node)

9. CONCLUSION

In this paper, we study the problem of I/O efficient *DFS* in a large graph that cannot reside entirely in the main memory, which is widely used in many real applications. We analyze the drawbacks of the existing semi-external *DFS* algorithms, and observe that the I/O efficiency of the *DFS* algorithm can be largely improved when we conduct *DFS* in a divide & conquer manner. We discuss the challenges of divide & conquer, and discuss four properties in order to find a valid graph division. Based on the properties, we design two novel graph division algorithms and a graph merge algorithm to significantly reduce the I/O cost of *DFS*. We conduct extensive performance studies on both real and synthetic web-scale datasets, one of which contains more than 3.7 billion edges. The experimental results show that our divide & conquer based *DFS* algorithms outperform the existing semi-external *DFS* algorithms significantly.

Acknowledgements. The work is supported by Research Grants Council of the Hong Kong SAR, China No. 14209314 and 418512. The third author is supported by ARC DE140100999.

10. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9), 1988.

[2] D. Ajwani and U. Meyer. *Algorithmics of Large and Complex Networks*, chapter 1: Design and Engineering of External Memory Traversal Algorithms for General Graphs. Springer, 2009.

[3] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *J. Graph Algorithms Appl.*, 7(2):105–129, 2003.

[4] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proc. of SODA'00*, 2000.

[5] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. of SODA'95*, 1995.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2001.

[7] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Structure of growing networks with preferential linking. *Phys. Rev. Lett.*, 85, 2000.

[8] J. Freeman. Parallel algorithms for depth-first search. Technical Report MS-CIS-91-71, 1991.

[9] J.-H. Her and R. S. Ramakrishna. An external-memory depth-first search algorithm for general grid graphs. *Theor. Comput. Sci.*, 374(1-3):170–180, 2007.

[10] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc of SPDP'96*, 1996.

[11] S. A. M. Makki and G. Havas. Distributed algorithms for depth-first search. *Inf. Process. Lett.*, 60(1):7–12, Oct. 1996.

[12] S. A. M. Makki and G. Havas. An efficient method for constructing a distributed depth-first search tree. In *Proc. of PDPTA'97*, 1997.

[13] M. B. Sharma and S. S. Iyengar. An efficient distributed depth-first-search algorithm. *Inf. Process. Lett.*, 32(4), 1989.

[14] J. F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proc. of SPAA'02*, 2002.

[15] S. Singh, A. Subramanya, F. Pereira, and A. McCallum. Wikilinks: A large-scale cross-document coreference corpus labeled via links to Wikipedia. Technical Report UM-CS-2012-015, 2012.

[16] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[17] J. S. Vitter. External memory algorithms and data structures. *ACM Comput. Surv.*, 33(2), 2001.

[18] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin. I/o efficient: Computing sccs in massive graphs. In *Proc. of SIGMOD'13*, 2013.

APPENDIX

Proof of Theorem 5.1:

(1) We prove $\Rightarrow$: This is trivial, since $E(G_i) \cap E(G_j) = \emptyset$ indicates that there is no edge connection between two graphs. So the DFS-Trees are independent.

(2) We prove $\Leftarrow$: For any two graphs G_i and G_j ($1 \leq i < j \leq p$), the result of DFS-Tree in G_i will not affect that in G_j indicates that for nodes in $V(G_i) \cap V(G_j)$, there will be only one visiting order. If $E(G_i) \cap E(G_j) \neq \emptyset$, there exist at least one edge (u, v) such that $(u, v) \in E_i$ and $(u, v) \in E_j$. Suppose there is a virtual node γ_i that is connected to all nodes in G_i as a root node to *DFS* G_i, and a virtual node γ_j that is connected to all nodes in G_j as a root node to *DFS* G_j, note that although the order of visiting a certain node v in a *DFS* is dependent on the set of nodes that visited before v, the first node of the *DFS* can be arbitrarily selected since there are no proceeding visited nodes before it in the *DFS*. In such a situation, we can construct two DFS-Trees as follows:

- In the first DFS-Tree T_i, we visit u as the first node in the *DFS* (except for the root node γ_i) and v will be visited after u. In such a way, u is an ancestor or sibling of v in the corresponding DFS-Tree T_i of G_i.
- In the second DFS-Tree T_j, we visit v as the first node in the *DFS* (except for the root node γ_j). In such a case, no matter how v is visited in the *DFS*, u cannot be an ancestor of v in the corresponding DFS-Tree T_j of G_j since node v has only one ancestor which is γ_j. If u and v are siblings, v is visited before u.

In other words, we can get DFS-Trees T_i and T_j for G_i and G_j respectively, such that the relationships of u and v in T_i and T_j are inconsistent. Thus the independence property is violated.

According to (1) and (2), the theorem is proved. □

Proof of Lemma 5.1:

Suppose the union of $T_0, T_1, \cdots, T_p$ is a graph T, i.e., $V(T) = \bigcup_{0 \leq i \leq p} V(T_i)$, and $E(T) = \bigcup_{0 \leq i \leq p} E(T_i)$, we prove the lemma by contradiction. Suppose T is a spanning tree/forest of G, and there exists a certain pair of trees T_i and T_j ($0 \leq i < j \leq p$), such that none of the conditions C_1, C_2, and C_3 are satisfied. That means there exists a node $v \in V(T_1)$ and $v \in V(T_2)$, where v is neither a root of T_1 nor a root of T_2. Suppose the parent node of v in T_1 is u_1, and the parent node of v in T_2 is u_2, according to the independence property, $u_1 \notin V(G_2)$, which indicates that $u_1 \neq u_2$. In such a situation, in T, the node v has two parent nodes u_1 and u_2. This contradicts with the assumption that T is a spanning tree/forest of G. Thus, the lemma holds. □

Proof of Lemma 5.3:

(1) We prove $\Rightarrow$: We prove this by contradiction. First, T^* is a DFS*-Tree of G and the corresponding DFS-Tree is T, then for every edge $(u, v) \in E(T^*)$, (u, v) is a cross edge in T^* if and only if it is a cross edge in T. (However, a forward-cross (backward-cross) edge in T^* may become a backward-cross (forward-cross) edge in T.) Two nodes u, v have ancestor or descendant relationship in T if and only if they have ancestor or descendant relationship in T^*. Next, suppose there exist edges (u_1, v_1), (u_2, v_2), $\cdots$, (u_k, v_k) and nodes w_1, w_2, $\cdots$, w_k such that for every $1 \leq i \leq k$, (u_i, v_i) is a cross edge, w_i is an ancestor of both v_i and u_{i+1} in T^*, and w_i and w_{i+1} do not have ancestor/descendent relationship in T^*, then according to the above discussion, for every $1 \leq i \leq k$, (u_i, v_i) is also a cross edge w.r.t. T, w_i is also an ancestor of both v_i and u_{i+1} in T, and w_i and w_{i+1} do not have ancestor/descendent relationship in T. Since T is a DFS-Tree, (u_i, v_i) can only be a backward-cross edge w.r.t. T. Considering that w_i is an ancestor of both v_i and u_{i+1} in T, and w_i and w_{i+1} do not have ancestor/descendent relationship in T, we can get that the *DFS* order of u_{i+1} is smaller than that of u_i for any $1 \leq i \leq k$. Thus we can derive that the *DFS* order of u_{k+1} is smaller than that of u_1, this contradicts with that $u_{k+1} = u_1$. Thus, $\Rightarrow$ holds.

(2) We prove $\Leftarrow$: We prove this by constructing a DFS-Tree from T^*. Given a tree T^* such that there do not exist edges (u_1, v_1), (u_2, v_2), $\cdots$, (u_k, v_k) and nodes w_1, w_2, $\cdots$, w_k such that for every $1 \leq i \leq k$, (u_i, v_i) is a cross edge, w_i is an ancestor of both v_i and u_{i+1} in T^*, and w_i and w_{i+1} do not have ancestor/descendent relationship in T^*, then for every forward-cross edge (u, v) w.r.t. T^*, we eliminate the forward-cross edge by reconstructing T^* using the following operation:

Forward-cross edge elimination operation: Suppose w is the lowest common ancestor of u and v in T^*, and u lies in the subtree rooted at w_u, and v lies in the subtree rooted at w_v, where w_u and w_v are two child nodes of w in T^*, we simply swap the positions of the two subtrees rooted at w_u and w_v in T^*.

By applying the above forward-cross edge elimination operation, we claim that no new forward-cross edge is produced. We prove this claim by contradiction. Suppose after eliminating forward-cross edge (u, v), a new forward-cross edge (u', v') is produced. Then (u', v') is a backward-cross edge in T^* before the elimination operation. Consequently, we find two edges (u, v), (u', v'), and two nodes w_v and w_u w.r.t. T^* such that both (u, v) and (u', v') are cross edges, w_v is an ancestor of both v and u' and w_u is an ancestor of both u and v', and w_u and w_v do not have ancestor or descendant relationship. This contradicts with the assumption. Thus $\Leftarrow$ holds.

According to (1) and (2), the lemma holds. □

Proof of Lemma 6.1:

(1) We prove $\Rightarrow$: we prove this by contradiction. Suppose T_0 is not a DFS*-Tree of Σ^*, then according to lemma 5.3, we can find edges (u_1, v_1), (u_2, v_2), $\cdots$, (u_k, v_k) and nodes w_1, w_2, $\cdots$, w_k in Σ^*, such that for every $1 \leq i \leq k$, (u_i, v_i) is a cross edge, w_i is an ancestor of both v_i and u_{i+1} in T_0, and w_i and w_{i+1} do not have ancestor/descendent relationship in T_0. Let the corresponding edges of (u_1, v_1), (u_2, v_2), $\cdots$, (u_k, v_k) in the original graph G be (u'_1, v'_1), (u'_2, v'_2), $\cdots$, (u'_k, v'_k), then obviously, we find edges (u'_1, v'_1), (u'_2, v'_2), $\cdots$, (u'_k, v'_k) and nodes w_1, w_2, $\cdots$, w_k in G, such that for every $1 \leq i \leq k$, (u'_i, v'_i) is a cross edge, w_i is an ancestor of both v'_i and u'_{i+1} in T_0, and w_i and w_{i+1} do not have ancestor/descendent relationship in T_0. According to lemma 5.3 and lemma 5.2, the division is not DFS-preservable. Thus $\Rightarrow$ holds.

(2) We prove $\Leftarrow$: let T_1, T_2, $\cdots$, T_p be the corresponding DFS-Trees of G_1, G_2, $\cdots$, G_p respectively, and T be the union of T_0, T_1, $\cdots$, T_p, we show that if T_0 is a DFS*-Tree of Σ^* then T is a DFS*-Tree of G, thus according to lemma 5.2, the division is DFS-preservable. Since T_0 is a DFS*-Tree of Σ^*, we can reorder nodes in T_0 such that T_0 is a DFS-Tree of Σ^*, suppose we reorder nodes in T accordingly based on the order of the root nodes r_1, r_2, $\cdots$, r_p in the new T_0, for any cross edge (u', v') in T, we consider two cases as follows:

- Case 1, u' and v' belong to the same tree T_i: since T_i is a DFS-Tree, (u', v') should be a backward-cross edge.
- Case 2, u' and v' belong to two different trees T_i and T_j: let the (u, v) in G be the corresponding edge of (u', v') in Σ^*, since T_0 is a DFS-Tree of Σ^*, (u', v') should be a backward-cross edge in T_0 w.r.t. Σ^*, as a result (u, v) should be a backward-cross edge in T w.r.t. G, since the *DFS* orders of nodes are consistent in T_0 and T.

Based on case 1 and case 2, we conclude that there are no forward-cross edges in G w.r.t. T, thus T is a DFS-Tree of G. As a result, $\Leftarrow$ is proved.

According to (1) and (2), the lemma holds. $\square$

Proof of Lemma 6.2:

We prove the case when G' is a graph by replacing (u, v) in G with $\mathsf{pushup}((u, v), T, v)$, the case by replacing (u, v) with $\mathsf{pushup}((u, v), T, u)$ can be proved similarly.

(1) We prove $\Rightarrow$: Since T is a DFS^*-Tree of G, we can reorder T such that only backward-cross edges exist in G w.r.t. T, now we show that only backward-cross edges exist in G' w.r.t. T. We only need to show that the edge $\mathsf{pushup}((u, v), T, u)$ is a backward-cross edge. If $\mathsf{pushup}((u, v), T, u) = (u, v)$, the case is trivial. We only need to prove the case for $\mathsf{pushup}((u, v), T, u) = (w, v)$ where w is the parent node of u in T. Since (u, v) is a backward-cross edge w.r.t. T, w is the parent node of u, but not the parent node of v, we can get that the *DFS* order of w is smaller than that of v, thus, (w, v) is a backward-cross edge w.r.t. T in graph G. As a result $\Rightarrow$ is proved.

(2) We prove $\Leftarrow$: Since T is a DFS^*-Tree of G', we can reorder T such that only backward-cross edges exist in G' w.r.t. T, now we show that only backward-cross edges exist in G w.r.t. T. Given that $\mathsf{pushup}((u, v), T, u)$ is a backward-cross edge in G' w.r.t. T, we only need to show that the edge (u, v) is a backward-cross edge in G w.r.t. T. If $\mathsf{pushup}((u, v), T, u) = (u, v)$, the case is trivial. We only need to prove the case for $\mathsf{pushup}((u, v), T, u) = (w, v)$ where w is the parent node of u in T. Since (w, v) is a backward-cross edge w.r.t. T, the *DFS* order of w is larger than that of v. Since w is the parent node of u, the *DFS* order of u is larger than that of w. We can get that the *DFS* order of u is larger than that of v, thus, (u, v) is a backward-cross edge w.r.t. T. As a result $\Leftarrow$ is proved.

According to (1) and (2), the lemma is proved. $\square$

Proof of Lemma 6.3:

Given a spanning tree T of G, in order to check whether T is a DFS^*-Tree, according to Lemma 5.3, we need to find whether there exist edges (u_1, v_1), (u_2, v_2), $\cdots$, (u_k, v_k) and nodes $w_1, w_2, \cdots, w_k$ such that for every $1 \leq i \leq k$, (u_i, v_i) is a cross edge, w_i is an ancestor of both v_i and u_{i+1} in T, and w_i and w_{i+1} do not have ancestor/descendent relationship in T. Let c_i be the LCA of u_i and v_i, since w_{i-1} is an ancestor of u_i and w_i is an ancestor of v_i, and w_{i-1} and w_i do not have ancestor/descendent relationship, we can get that c_i is also the LCA of w_{i-1} and w_i. Now suppose that in the graph G, all cross edges has been replaced by their corresponding S-edges, and there still exist edges (u_1, v_1), (u_2, v_2), $\cdots$, (u_k, v_k) and nodes $w_1, w_2, \cdots, w_k$ such that for every $1 \leq i \leq k$, (u_i, v_i) is a cross edge, w_i is an ancestor of both v_i and u_{i+1} in T, and w_i and w_{i+1} do not have ancestor/descendent relationship in T. In this situation, we can get that if c_i is the LCA of u_i and v_i, then c_i is the parent node of u_i and v_i. We also have that c_i is the LCA of w_{i-1} and w_i, and w_{i-1} is an ancestor of u_i and w_i is an ancestor of v_i. We can easily derive that $v_i = w_i$ and $u_i = w_{i-1}$, thus we have $v_i = w_i = u_{i+1}$ for any $1 \leq i \leq k$. In other words, we only need to check whether there are cycles formed by cross edges in the new graph G. Therefore, the lemma is proved. $\square$