I/O efficient: computing SCCs in massive graphs

Conference Paper *in* The VLDB Journal · June 2013

DOI: 10.1145/2463676.2463703

5 authors, including:

Zhiwei Zhang
Tianjin University
44 PUBLICATIONS **856** CITATIONS
SEE PROFILE

Jeffrey Xu Yu
The Chinese University of Hong Kong
391 PUBLICATIONS **12,625** CITATIONS
SEE PROFILE

Lu Qin
University of Technology Sydney
258 PUBLICATIONS **8,098** CITATIONS
SEE PROFILE

Lijun Chang
The University of Sydney
103 PUBLICATIONS **3,158** CITATIONS
SEE PROFILE

I/O Efficient: Computing SCCs in Massive Graphs

Zhiwei Zhang[†], Jeffrey Xu Yu[†], Lu Qin[†], Lijun Chang[‡], Xuemin Lin[‡]

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

[‡]The University of New South Wales, Australia, {ljchang,lxue}@cse.unsw.edu.au

ABSTRACT

A strongly connected component (SCC) is a maximal subgraph of a directed graph G in which every pair of nodes are reachable from each other in the SCC. With such a property, a general directed graph can be represented by a directed acyclic graph (*DAG*) by contracting an SCC of G to a node in *DAG*. In many real applications that need graph pattern matching, topological sorting, or reachability query processing, the best way to deal with a general directed graph is to deal with its *DAG* representation. Therefore, finding all SCCs in a directed graph G is a critical operation. The existing in-memory algorithms based on depth first search (*DFS*) can find all SCCs in linear time w.r.t. the size of a graph. However, when a graph cannot resident entirely in the main memory, the existing external or semi-external algorithms to find all SCCs have limitation to achieve high I/O efficiency. In this paper, we study new I/O efficient semi-external algorithms to find all SCCs for a massive directed graph G that cannot reside in main memory entirely. To overcome the deficiency of the existing *DFS* based semi-external algorithm that heavily relies on a total order, we explore a weak order based on which we investigate new algorithms. We propose a new two phase algorithm, namely, tree construction and tree search. In the tree construction phase, a spanning tree of G can be constructed in bounded sequential scans of G. In the tree search phase, it needs to sequentially scan the graph once to find all SCCs. In addition, we propose a new single phase algorithm, which combines the tree construction and tree search phases into a single phase, with three new optimization techniques. They are early acceptance, early rejection, and batch processing. By the single phase algorithm with the new optimization techniques, we can significantly reduce the number of I/Os and CPU cost. We conduct extensive experimental studies using 4 real datasets including a massive real dataset, and several synthetic datasets to confirm the I/O efficiency of our approaches.

Categories and Subject Descriptors

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

Keywords

graph algorithm; I/O efficient; SCC computing

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

1. INTRODUCTION

Finding all Strongly Connected Components (SCCs) is an important operation for directed graphs. Here, an SCC is a maximal subgraph S for a given directed graph G, such that for every pair of nodes u and v in S, there is a directed path from u to v in S and there is also a directed path from v to u in S. Many real applications that deal with a directed graph need to find all SCCs first.

Topological Sort: Topological sort is widely used in many applications especially in planning and scheduling. In a topological sort, nodes in a directed graph are ranked according to a partial order specified by the edges. If there are cycles in the graph, all nodes in a cycle are considered as equal rank and merged into one. This is done by finding all SCCs in the graph. In [13], Hellings et al. propose an algorithm for external bisimulation on graphs where all nodes are assumed to be in the reverse topological order and stored on disk. This needs to find all SCCs in a preprocessing step.

Reachability Query Processing: Reachability query is a widely studied query to ask whether a node u can reach another node v through a directed path in a directed graph. There are many applications in social networks, biological networks, software analysis, and semantic web. Because two nodes in an SCC are reachable from each other, in the literature, almost all algorithms to process reachability queries over a general directed graph G first convert G into a directed acyclic graph (*DAG*) by contracting an SCC into a node, which needs to find all SCCs in a prepossessing step. As an example, for handling reachability queries over a massive directed graph, the GRAIL index [22] needs to be built on *DAG*. The key point is that it must compute all SCCs before constructing an index for a general directed graph.

Graph Pattern Matching: Pattern matching in *XML* data and graph data has been widely studied. Computing SCCs is an optimization technique to compress a large graph for processing pattern matching queries [12].

In the literature, many efficient in-memory algorithms for finding all SCCs have been studied. The Kosaraju-Sharir algorithm [3] can find all SCCs for a directed graph in linear time w.r.t. the size of the graph, by depth first searching the graph twice in memory. However, due to the fact that the sizes of many real graphs are increasing rapidly, a graph cannot reside entirely in the main memory. As an example, there are more than 901 million active nodes and more than 125 billion edges.[1] As a small part of the entire web, WEBSPAM-UK2007[2] contains 105,896,555 pages in 114,529 hosts in the .UK domain in May 2007. This graph contains nearly 4 billion edges.

[1]http://newsroom.fb.com/

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

For handling a large directed graph $G(V, E)$, a naive way to externalize the in-memory *DFS* based algorithm requires $O(|E|)$ I/Os. Chiang et al [8] propose an algorithm with I/O complexity $O(|V| + \frac{|V|}{M} \cdot scan(|E|) + sort(|E|))$. Later, Kumar and Schwabe [14] and Buchsbaum et al. [7] improve the I/O complexity to $O((|V| + \frac{|E|}{B}) \log_2 \frac{|V|}{B} + sort(|E|))$ by maintaining the list of edges that should not be traversed using tournament trees [14] and buffered repository trees [7], respectively. Here, B is the disk block size. Despite their theoretical guarantees, these algorithms are considered impractical for general directed graphs that encountered in real applications. Cosgaya-Lozano et al. [10] study a heuristic external algorithm to compute all SCCs for a directed algorithm based on contraction used by Chiang et al [8] which is for undirected graphs. But, for directed graphs, the algorithm may end up an infinite loop and cannot compute all SCCs. Sibeyn et al. [20] study a semi-external algorithm, which constructs two Depth First Search (*DFS*) trees on the directed graph, based on the Kosaraju-Sharir algorithm, to find all SCCs. But, the algorithm generates a large number of I/Os, and is hard to be optimized.

In this paper, we propose a new I/O efficient semi-external algorithm to compute all SCCs for a directed graph G. The main contributions of this work are summarized below. First, we observe that the total order used in Kosaraju-Sharir algorithm [3] to handle an in-memory graph is too strong for an I/O efficient algorithm to handle a graph on disk. We propose a weaker order which is based on the depth of a node in a graph. By the weaker order, we can significantly reduce the number of I/Os. Second, we analyze the main costs incurring in the algorithm by Sibeyn et al. [20], and propose new techniques to reduce such costs using a single tree. Third, based on the weak order, we propose a new two phase algorithm. In the first phase it constructs a tree, and in the second phase it searches the tree constructed to find all SCCs. In the tree construction phase, it needs $\mathsf{depth}(G) \cdot |E|/B$ I/Os in the worst case, where $\mathsf{depth}(G)$ is the longest simple path in G, and B is the disk block size. In the tree search phase, it only needs to use $|E|/B$ I/Os. As a comparison, the algorithm in [20] finds all SCCs by two *DFS* over a graph G. One *DFS* needs $\mathsf{depth}(G) \cdot |E|/B$ I/Os. Fourth, we further propose a new single phase algorithm, which combines the tree construction and tree search phases into one phase. We propose new optimization techniques, namely, early acceptance, early rejection, and batch processing. By early acceptance, we contract a partial SCC into a node in an early stage while constructing a tree. By early rejection, we remove nodes that will not participate in any SCCs while constructing the tree in an early stage. In addition, we use batch processing techniques to reduce the CPU cost. By the single phase algorithm, we can significantly reduce the number of I/Os and CPU cost. Finally, we conduct extensive experimental studies using 4 real datasets including a massive real dataset, and several synthetic datasets to confirm the I/O efficiency of our approach.

The remainder of this paper is organized as follows. In Section 3, we discuss the preliminaries and give the problem statement for computing SCCs. In Section 4, we discuss two existing solutions. In Section 5, we outline our approaches. We discuss a new two phase single tree approach with a tree construction phase and a tree search phase in Section 6 and a new single phase single tree approach with new optimization techniques in Section 7. In Section 2, we discuss the related work. We report our experimental results in Section 8 and conclude the paper in Section 9.

2. RELATED WORK

Finding strongly connected components of a directed graph G is a primitive operation in directed graph exploration, which has been studied for both internal memory model and external memory model. In the internal memory model, where the whole graph resides in main memory, strongly connected components of a directed graph can be computed in $O(n + m)$ where $n = |V(G)|$ and $m = |E(G)|$ based on Depth-First Search (*DFS*) [9].

The difference between accessing an element in main memory and fetching an element on disk is captured by the external memory model (or the I/O model) [2], which counts disk accesses incurred by an algorithm using the following parameters: M is the memory size and B is the block size. In the I/O model [2], scanning n elements requires $scan(n) = \Theta(\frac{n}{B})$ I/Os, and sorting n elements requires $sort(n) = \Theta(\frac{n}{B} \cdot \log_{\frac{M}{B}} \frac{n}{B})$ I/Os.

A naive way to externalize the internal *DFS* algorithm requires $O(|E|)$ I/Os. Chiang et al [8] propose an algorithm with I/O complexity $O(|V| + \frac{|V|}{M} \cdot scan(|E|) + sort(|E|))$. Later, Kumar and Schwabe [14] and Buchsbaum et al. [7] improve the I/O complexity to $O((|V| + \frac{|E|}{B}) \log_2 \frac{|V|}{B} + sort(|E|))$ by maintaining the list of edges that should not be traversed using tournament trees [14] and buffered repository trees [7], respectively. Despite their theoretical guarantees, these algorithms are considered impractical for general directed graphs that encountered in real applications. As an example, for the WEBSPAM-UK2007 graph with 105,895,555 nodes used in our experiment, the algorithm by Buchsbaum et al. [7], needs about 1,566,000,000 I/Os for a certain *DFS*, and our algorithm uses about 4,000,000 I/Os to find all SCCs in the large graph.

Two approaches are proposed recently targeting at designing efficient algorithms with good performance in real applications [20, 10]. Sibeyn et al. [20] propose an implementation of semi-external *DFS*, i.e., assuming that the main memory can hold the node set but not the edge set of the graph. Cosgaya-Lozano and Zeh [10] present a contraction-based algorithm which does not follow the line of *DFS*-based algorithms. Their algorithm identifies and contracts SCCs repeatedly until the graph fits in memory, then an internal memory algorithm is run to find the final SCCs.

Other than the problem of finding SCCs or *DFS* tree on external directed graphs, several problems in the external memory model are studied in the literature. Dementiev et al. [11] provide an implementation of an external memory minimum spanning tree algorithm based on the ideas of [19], which performs extremely well in practice, even though theoretically inferior to the algorithms of [1, 8]. Ajwani et al. [4, 6] propose implementations of external undirected breadth-first search algorithm with the idea from [15]. Ulrich Meyer et al. [17, 18, 16] design and implement practical I/O-efficient single source shortest paths algorithm on general undirected sparse graphs. Surveys about designing I/O efficient algorithms for massive graphs can be found at [21, 5].

3. PROBLEM DEFINITION

We model a directed graph as $G(V, E)$, where $V(G)$ represents the set of nodes and $E(G)$ represents the set of directed edges in G, and denote the number of nodes and the number of edges of graph G by n and m, i.e., $n = |V(G)|$ and $m = |E(G)|$. For simplicity, we also use V and E to denote $V(G)$ and $E(G)$ of G respectively, when it is obvious. Given a graph G, a path $p = (v_1, v_2, \cdots, v_k)$ is a sequence of k nodes in V such that, for each $v_i (1 \leq i < k)$, $(v_i, v_{i+1}) \in E$. A node v_i can reach a node v_j in G, denoted $v_i \rightarrow v_j$, iff there exists a path from v_i to v_j in G. The length of a path p, denoted as $\mathsf{len}(p)$, is the number of edges in p. A simple path is a path $(v_1, v_2, \cdots, v_k)$ with k distinct nodes. A longest simple path p is a simple path in G with the largest $\mathsf{len}(p)$. We use $\mathsf{depth}(G)$ to denote the length of the longest simple path in G. A cycle is a path where a same node appears more than once, and a

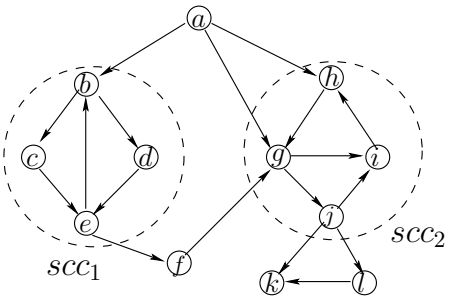


Figure 1: A Graph G with 2 SCCs

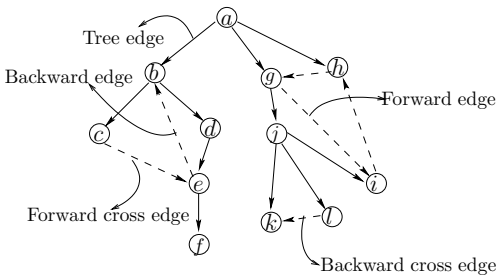


Figure 2: A *DFS* Tree T with Different Edge Types

simple cycle is a path $(v_1, v_2, \cdots, v_{k-1}, v_k)$ where the first $k-1$ nodes are distinct but $v_k = v_1$.

Given a directed graph G, node v_i is strongly connected to node v_j, denoted as $v_i \leftrightarrow v_j$, iff $v_i \rightarrow v_j$ and $v_j \rightarrow v_i$ in G. Here, $\leftrightarrow$ is an equivalence relation, which is reflexive, symmetric, and transitive. We use $v_i \nleftrightarrow v_j$ to denote that node v_i is not strongly connected to node v_j. A strongly connected component (SCC) of G is the maximal set of nodes V_c ($\subseteq V$) such that, for every pair of nodes v_i and v_j in V_c, $v_i \leftrightarrow v_j$, and for every pair of nodes v_i and v_k $v_i \in V_c$ and $v_k \notin V_c$, $v_i \nleftrightarrow v_k$. Fig. 1 shows a graph G with 12 nodes and 18 edges. There are 2 SCCs, SCC_1 and SCC_2.

Finding SCCs in a directed graph G can be done with a spanning tree of G. A spanning tree $T(V, E)$ of $G(V, E)$ is a connected directed tree that consists of all nodes ($V(T) = V(G)$) for a connected graph G. For a directed graph G that has several connected components, a spanning tree can be constructed with a virtual node v_0 which links to every spanning tree constructed for a connected component in G. The depth of a node v in a directed tree T is denoted by $\mathsf{depth}(v, T)$, where $\mathsf{depth}(v_0, T) = 0$.

Problem Statement: compute all SCCs for a large directed graph $G(V, E)$ with limited memory M. Here $c|V| \leq M \ll \|G\|$ where c is a small constant number for example $c = 2$, $|V|$ is the number of nodes, and $\|G\|$ is the space for the entire graph G.

4. EXISTING SOLUTIONS

In the literature, there are an external algorithm and a semi-external algorithm to find all SCCs of a directed graph G. The external algorithm is based on contraction [10], and the semi-external algorithm is based on depth first search (*DFS*) [20].

Contraction Based EM-SCC: Cosgaya-Lozano et al. [10] provide a heuristic algorithm, called EM-SCC, to compute all SCCs for a large directed graph, by taking the idea of the contraction-based algorithm for undirected graphs by Chiang et al. in [8]. Given limited memory M, EM-SCC compresses a graph iteratively by contraction until a graph can fit in M, In brief, it processes G in iterations, $G = G_0, G_1, G_2, \cdots, G_f$. In the i-th iteration, EM-SCC contracts some partial SCCs into a node and compresses G_i to be a smaller G_{i+1} for next iteration. The last G_f must fit in M. In the i-th iteration ($i < f$), G_i cannot fit in M. EM-SCC partitions G_i into smaller partitions, $G_{i_1}, G_{i_2}, \cdots$ where G_{i_j} can fit in M. EM-SCC computes SCCs using an in-memory algorithm for G_{i_j}, and compresses G_i by contracting an SCC in G_{i_j} into a node, which maintains the connectivity between nodes in the SCC with other nodes in G.

Unlike the case for undirected graphs where Chiang et al. [8] ensure an undirected graph G can fit into main memory in a log number of iterations, EM-SCC has its limitation to use the same idea for directed graphs. EM-SCC cannot stop in a finite number of iterations in the following cases to compute all SCCs. (Case-1) An SCC of G_i appears across a number of partitions, and the partitions cannot be further compressed by contraction. (Case-2) G_i is a directed acyclic graph, but cannot fit in M. When G contains a large SCC or a large number of small/mid sized SCCs, the probability of Case-1 to happen is high. When G is a *DAG*-liked graph, the probability of Case-2 to happen is high. Due to its problem of finding all SCCs in a finite number of iterations, we will mainly discuss our approach with the semi-external algorithm based on depth first search.

Depth First Search Based DFS-SCC: Sibeyn et al. [20] propose a semi-external algorithm, which finds all SCCs by depth first searching G twice. With a depth first search of G, it constructs a *DFS* tree T which is a spanning tree of G. Given a spanning tree T, every edge (u, v) in G is categorized into tree-edge and non-tree-edge. A tree-edge (u, v) is an edge in T. There are 4 types of non-tree-edges: (1) forward-edge: u is an ancestor of v in T, (2) backward-edge: u is a descendant of v in T, (3) forward-cross-edge: u and v do not have ancestor/descendant relationship and u appears before v by the preorder of T for G, and (4) backward-cross-edge: u and v do not have ancestor/descendant relationship and u appears after v by the preorder of T for G. A spanning tree is a *DFS* tree if and only if there is no forward-cross-edges in the graph. In order to find such tree, the idea is to reduce the number of forward-cross-edges until none can be found. Fig. 2 shows such a tree T rooted at a with solid edges for the graph G in Fig. 1. Given T, (a, b) is a tree-edge because $(a, b) \in E(T)$, (e, b) is a backward-edge because b is an ancestor of e in T, (g, i) is a forward-edge because g is an ancestor of i in T, (c, e) is a forward-cross-edge, because c appears before e by the preorder of T, and (l, k) is a backward-cross-edge because l appears after k by the preorder of T.

The semi-external depth first search algorithm is shown in Algorithm 1, denoted as DFS-Tree(G) [20]. Algorithm 1 constructs a *DFS* tree. Among the 4 types of non-tree-edges, backward-edge, forward-edge, and backward-cross-edge are excluded by the ancestor/descendant relationship and preorder. Algorithm 1 repeatedly reconstructs a *DFS* tree by reducing the number of forward-cross-edges (lines 8-10) in every iteration (lines 7-12). In an iteration, Algorithm 1 accesses all edges in $E(G)$ on disk in sequential order, and attempts to reduce the number of forward-cross-edges as many as possible. The iteration stops when there are no forward-cross-edges ($updated$ = false), and the number of iterations is $\mathsf{depth}(G)$ in the worst case. There are theoretical approaches in the literature to reduce the I/O complexity but are impractical [8, 14, 7], as discussed in the related work.

The DFS-SCC algorithm (Algorithm 2) finds all SCCs by depth first searching G twice, or in other words, by calling Algorithm 1 twice. The first time is to obtain a decreasing postorder of nodes over the *DFS* tree obtained by DFS-Tree(G) (lines 1-2). It then constructs a graph $\overline{G}$ by reversing every edge in G (line 3). Because $V(\overline{G}) = V(G)$, with the same decreasing postorder, it calls DFS-Tree($\overline{G}$) again to obtain a new *DFS* tree T_S (line 4). Here, all nodes in a subtree rooted at a child of the virtual node v_0 of T_S

Algorithm 1 DFS-Tree(G)

1: **for all** nodes $v \in V(G)$ in the order as specified **do**
2: add edge (v_0, v) in T where v_0 is a virtual node;
3: construct an initial *DFS* tree T and assign all nodes in $V(T)$ with preorder;
4: $updated \leftarrow$ true;
5: **while** $updated =$ true **do**
6: $updated \leftarrow$ false;
7: **for all** edges $(u, v) \in E(G)$ **do**
8: **if** (u, v) is a forward-cross-edge **then**
9: $w \leftarrow$ the parent of v in T;
10: delete the edge (w, v) from T and add the edge (u, v) in T;
11: $updated \leftarrow$ true;
12: reassign all nodes in $V(T)$ with preorder by *DFS* of T if $updated =$ true;
13: **return** T;

Algorithm 2 DFS-SCC(G)

1: $T \leftarrow$ DFS-Tree(G);
2: sort $V(G)$ in decreasing postorder by traversing T;
3: construct a graph $\overline{G}$ from G by reversing every edge in G, where $V(\overline{G}) = V(G)$ (with the same postorder);
4: $T_S \leftarrow$ DFS-Tree($\overline{G}$);
5: output that all nodes in a subtree of the virtual node v_0 of T_S forms an SCC;

are in the same SCC. The main idea of Algorithm 2 is to develop a semi-external algorithm based on the Kosaraju-Sharir algorithm which finds all SCCs in main memory [3].

Consider G in Fig. 1. The first *DFS* traverses G in $abcefgjklihd$, and its decreasing postorder is $abdcefgjihlk$. With the decreasing postorder, in the second *DFS*, the root v_0 of the *DFS* tree computed has 6 subtrees representing 6 SCCs $\{a\}$, $\{b, c, d, e\}$, $\{f\}$, $\{g, h, i, j\}$, $\{k\}$, and $\{l\}$.

There are 3 main costs in DFS-SCC. (Cost-1) DFS-SCC needs to traverse G twice using *DFS* to compute all SCCs. In each *DFS*, it needs the number of depth$(G) \cdot |E(V)|/B$ I/Os in the worst case where B is the block size. In the literature, there are no efficient external *DFS* algorithms due to a large number of random I/O accesses produced in the in-memory algorithm designed for *DFS*. (Cost-2) Even though there are partial SCCs which can be possibly contracted to save space while *DFS*, or in other words, while constructing a *DFS* tree, it cannot be done. We explain the reasons. As implied by the Kosaraju-Sharir algorithm [3], DFS-SCC uses a total order of nodes (decreasing postorder), computed in the first *DFS*, in the second *DFS*. Therefore, the nodes cannot be contracted in the first *DFS*. In the second *DFS*, we can possibly contract some partial SCCs into a node, but we have to maintain which node belongs to which SCC while reading edges $E(G)$ sequentially from disk to construct the second *DFS* tree. In other words, we cannot save space for free. All SCCs have to be identified at the same time when the entire second *DFS* tree is constructed. (Cost-3) In *DFS* (Algorithm 1), to reduce the number of forward-cross-edges, when forward-cross-edge, (u, v), is found, it needs to reshape the tree by updating the preorder of nodes in all subtrees rooted on the path from u to the root of the *DFS* tree constructed. Fig. 3 shows an example. Suppose a node, u, in subtree B to node g is a forward-cross-edge. The edge (a, g) will be deleted, and the edge (u, g) will be added. Also, the preorder of all nodes in the subtrees rooted at nodes, e, f, and g must be redone. Such process can be done in main memory but consumes high CPU cost.

In this paper, we propose a new algorithm in order to reduce the large CPU/IO cost produced by the DFS-SCC algorithm.

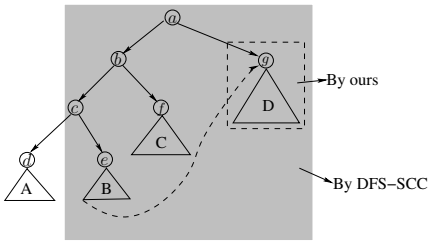


Figure 3: CPU Cost for Updating an Edge

5. THE NEW APPROACHES

In this section, we outline our new approaches to compute all SCCs for a directed graph G. Our key observation is that the Kosaraju-Sharir algorithm [3] is very efficient based on the total order over $V(G)$ when G can be held in main memory entirely, because all memory accesses can be done in constant time. In designing I/O efficient algorithms, such a total order is too strong in the sense that it cannot be used to reduce the number of I/Os and/or save space, and there is no obvious relationship between the total order and SCCs per se.

Instead of the decreasing postorder used in DFS-SCC, in this paper, we use a rather weak order. The idea behind the weak order is as follows. For an SCC, there must exist at least one cycle. While constructing a spanning tree T for G, a cycle (a part of SCC) will appear as to contain at least one edge (u, v) that links to a higher level node in T. A necessary condition can be specified as depth$(u) >$ depth(v) in T, or in other words, node u at a level, depth(u), links to a node v at a higher level, depth(v). There are two cases when depth$(u) >$ depth(v) holds. First, v is an ancestor of u in T. In such a case, (u, v) is a backward-edge and the cycle can be identified by combining tree-edges on the path from v to u plus the edge (u, v). Second, v is not an ancestor of u on T. In this paper, we call such an edge an up-edge. For the second case, we can locally reshape T in a similar way as done in Algorithm 1 when dealing with forward-cross-edges. In brief, let w be the parent of v for an up-edge (u, v), we delete (w, v) from T and add (u, v) in T. By doing so, we reduce an up-edge.

To reduce Cost-1 in the DFS-SCC algorithm, we create a BR-Tree, to compute all SCCs. Our algorithm can compute all SCCs efficiently by traversing the BR-Tree constructed only once. To reduce Cost-2, we propose new optimization techniques namely, early acceptance and early rejection. By early acceptance, we contract a partial SCC into a node in an early stage while constructing BR-Tree. By early rejection, we identify necessary conditions to remove nodes that will not participate in any SCCs while constructing BR-Tree. It is worth noting that such optimizations can not be easily introduced into the DFS-SCC algorithm as discussed before. To show the effectiveness of our approach, consider a tree T in Fig. 4. The 3 nodes e, f, and g can be contracted into a node e' by early acceptance, and node a as well as all nodes on the subtrees C and D can be removed by early rejection if the conditions are met. A large number of nodes/edges can be removed in every iteration in our algorithm. To reduce Cost-3, we show that when an up-edge (u, v) is found while constructing a BR-Tree, we only need to update the depths of nodes on the subtree rooted at node u. Reconsider Fig. 3, when eliminating an up-edge, (u, g) where u is in B, in the DFS-SCC algorithm, the order of all nodes under subtrees denoted by B, C and D need to be updated. We only need to update the nodes in the subtree D.

In the following we discuss our approach step-by-step based on a spanning tree called BR-Tree for G. A BR-Tree T can be constructed in any traversal order with which all non-tree edges in G

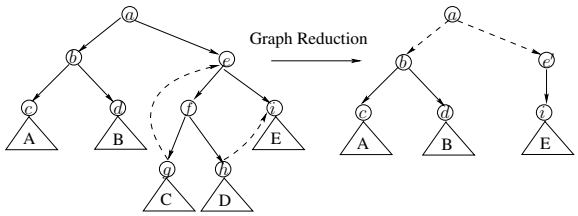


Figure 4: Early Acceptance and Early Rejection

are either backward-edge, up-edge, or down-edge. An edge (u, v) is a backward-edge if v is an ancestor of u. An edge (u, v) is an up-edge w.r.t. T iff $\mathsf{depth}(u, T) \geq \mathsf{depth}(v, T)$. An edge (u, v) is a down-edge w.r.t. T iff u is not an ancestor of v, v is not an ancestor of u in T, and $\mathsf{depth}(u, T) < \mathsf{depth}(v, T)$. The size of BR-Tree is $2|V(G)|$, because a node only needs to maintain its parent node in T.

Single Tree: We discuss a single BR-Tree T to compute all SCCs with a naive approach as follows.

construct a BR-Tree T; no-up-edge $\leftarrow$ true;
while no-up-edge = true **do**
  compute partial SCCs by contracting nodes in T with sequential scanning $E(G)$;
  if there is a new up-edge found in T **then** construct a BR-Tree T **else** no-up-edge $\leftarrow$ false;

Here, we can construct a BR-Tree T in a similar manner as shown in Algorithm 1 using the notion of up-edge[3] instead with I/O complexity $\mathsf{depth}(G) \cdot |E(G)|/B$ where B is a block size. Assume N up-edges are found in the while loop statement while contracting nodes, it needs to reconstruct BR-Tree T N times. Due to the high I/O complexity, this naive single tree approach is infeasible.

A Two Phase Construct and Search Approach: To significantly reduce the I/O complexity with a single tree for computing all SCCs, we give a new two phase approach, named 2P-SCC (Algorithm 3). In the first phase we construct a tree, and in the second phase we compute all SCCs by searching the tree constructed. The I/O complexity for the first construction phase is $\mathsf{depth}(G) \cdot |E(G)|/B$, and the I/O for the second search phase is only one scan of $E(G)$. We show that we can do the one scan of $E(G)$ in the second phase using a special spanning tree, BR$^+$-Tree, where a node u maintains its parent node in T plus one of its ancestors v by a backward-edge (u, v) in G. Here the size of BR$^+$-Tree is $3|V(G)|$. We will discuss the two phase approach using the special spanning BR$^+$-Tree, and then we show that we can do the same using BR-Tree instead of BR$^+$-Tree.

The one sequential scan of $E(G)$ is achieved based on a refined "up-edge". We give its new definition we use in this work which is based on two functions: $\mathsf{drank}(u, T)$ and $\mathsf{dlink}(u, T)$.

$$\mathsf{drank}(u, T) = \min\{\mathsf{depth}(v, T) | v \in \mathsf{Rset}(u, G, T)\}$$
$$\mathsf{dlink}(u, T) = \mathrm{argmin}_v\{\mathsf{depth}(v, T) | v \in \mathsf{Rset}(u, G, T)\}$$

Here, $\mathsf{Rset}(u, G, T)$ denotes the set of nodes including u and nodes that v can reach by a BR$^+$-Tree T of G. The function $\mathsf{drank}(u, T)$ gives the minimum depth of v that u can reach in BR$^+$-Tree T, and $\mathsf{dlink}(u, T)$ specifies the unique node v that is with $\mathsf{drank}(u, T)$. Given BR$^+$-Tree T of G, we redefine an up-edge below.

Definition 5.1: (**Up-Edge**) Given a BR$^+$-Tree T of graph G, an edge $e = (u, v) \in E(G)$ is an up-edge w.r.t. T iff the following two conditions are satisfied: (1) u is not an ancestor of v and v is not an ancestor of u in T, and (2) $\mathsf{drank}(u, T) \geq \mathsf{drank}(v, T)$. □

[3]The up-edge will be used instead forward-cross-edge in the condition (line 8, Algorithm 1).

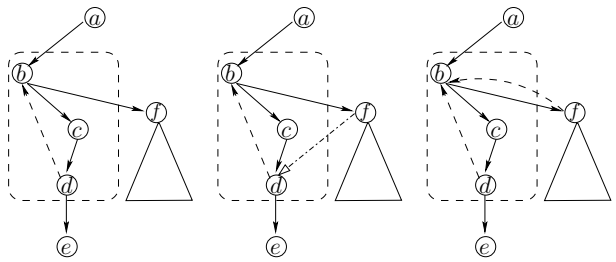


Figure 5: The Refined up-edge

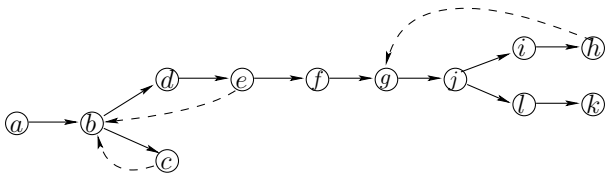


Figure 6: A BR-Tree T of G in Fig. 1

We explain the new up-edge using an example. Fig. 5(a) shows a BR$^+$-Tree T. The spanning tree part is shown by the solid edges and the additional backward-edges are shown by dotted edges. By the initial definition of up-edge, the edge (f, d) is not an up-edge over the spanning tree. By the new definition of up-edge, (f, d) becomes an up-edge (Fig. 5(b)). We handle this up-edge of (f, d) by indicating that f is in a cycle in G. This is done by deleting (f, d) from T, but adding a backward-edge of (f, b) to BR$^+$-Tree. As a result, all SCCs of G induce a connected subtree of BR$^+$-Tree T of G in the tree search phase by a single sequential scan of $E(G)$. As an example, Fig. 6 shows a BR$^+$-Tree T for G in Fig. 1. There are no up-edges in T. The SCC with nodes $\{b, c, d, e\}$ induces a subtree rooted at b in T. The SCC with nodes $\{g, h, i, j\}$ induces a subtree rooted at g in T. We will discuss the two-phase approach in Section 6.

A Single Phase Approach: To further improve the I/O efficiency with a single tree for computing all SCCs, we propose a new single phase approach, named 1P-SCC (Algorithm 6). 1P-SCC uses a single BR-Tree T with $2|V(G)|$ space instead of BR$^+$-Tree, which is achieved by two optimization techniques, early acceptance and early rejection. On the other hand, the two optimization techniques can further reduce the space to keep $V(G)$ in iterations. The space reduced can be used as a buffer to reduce the I/O cost for reading $E(G)$. We will discuss it in Section 7.

6. TWO PHASE SINGLE TREE

In this section, we discuss the two phases of the 2P-SCC algorithm (Algorithm 3), namely, Tree-Construction (Algorithm 4) and Tree-Search (Algorithm 5).

6.1 Tree Construction

We show an algorithm for tree construction, and the algorithm is named Tree-Construction as shown in Algorithm 4. Our Algorithm 4 shares similarity with Algorithm 1 but is with several main differences. Like Algorithm 1, Algorithm 4 traverses G and constructs a tree by reshaping the tree iteratively. Unlike Algorithm 1, Algorithm 4 does not request to use *DFS* to construct a spanning tree. The algorithm is to construct a tree by eliminating up-edges, lines 7-12, which can reduce CPU cost as indicated in Fig. 3. The elimination of an up-edge is done in one of two ways. One is explained in Fig. 5. The other is to reshape the tree with a pushdown operation.

Algorithm 3 2P-SCC(G)

Input: A directed graph G.
Output: All the SCCs in G.

1: $T \leftarrow$ Tree-Construction(G);
2: **return** Tree-Search(G, T);

Algorithm 4 Tree-Construction(G)

Input: A directed graph G.
Output: A spanning tree T of G.

1: $T \leftarrow$ a spanning tree of G;
2: update-drank(G, T);
3: $updated \leftarrow$ true;
4: **while** $updated =$ true **do**
5: $updated \leftarrow$ false;
6: **for all** edge $(u, v) \in E(G)$ in sequential order on disk **do**
7: **if** (u, v) is an up-edge **then**
8: **if** dlink(T, v) is an ancestor of u **then**
9: delete the edge (u, v) from T;
10: add the edge $(u,$ dlink$(T, v))$ in T;
11: **else**
12: $T \leftarrow T \Downarrow (u, v)$;
13: $updated \leftarrow$ true;
14: update-drank(G, T);
15: **return** T;

The operation pushdown ($\Downarrow$): Given a spanning tree T of G, for an up-edge $(u, v) \in E(G)$ w.r.t. T, pushdown operation $T \Downarrow (u, v)$ modifies T to be T' as follows. Let $T' = T$. (1) Cut the subtree T_v rooted at v from T'. (2) Paste the subtree T_v as a new subtree of u in T'. (3) Update depth(v) for every node in T_v w.r.t. T'.

Since u and v do not have ancestor/descendant relationship by the definition of up-edge, the operation $\Downarrow$ will result in another spanning tree T' of G, but reduce an up-edge. The cost of changing node depth (the step (3) in pushdown) is done locally. On the other hand, when *DFS* tree reshapes, Algorithm 1 needs to change the preorder globally. Algorithm 4 is more efficient than Algorithm 1, as indicated in Fig. 3.

In Algorithm 4, the while loop statement (lines 4-14) repeats by sequentially scanning $E(G)$ until no up-edges are found. In Algorithm 4, the procedure update-drank(G, T) updates the drank of all nodes in T in one scan of all edges of $E(G)$, where for each node $v \in V(T)$ we keep a backward-edge (v, u) with the smallest depth(u, T) if there is any. The procedure of update-drank does not incur additional I/Os because it can be done at the same time when accessing $E(V)$ (line 6).

Algorithm 4 needs at most $3 \times n \times b$ bytes of memory, where $n = |V(G)|$ and b is the number of bytes to store a node in G.

Lemma 6.1: *Algorithm 4 stops in at most* depth(G) *iterations.* □

Example 6.1: We show how to build a BR$^+$-Tree T for G (Fig. 1). An initial spanning tree T is shown in Fig. 7(1). First, for the up-edge (f, g), we cut the edge (a, g) and add a new edge (f, g). The result is shown in Fig. 7(2). For the up-edge (i, h), we cut the edge (a, h) and add a new edge (i, h). The result is shown in Fig. 7(3). Then, we find the up-edge (l, k), we cut the edge (i, k) and add another edge (l, k). For the up-edge (c, e), because dlink$(e, T) = b$ is an ancestor of c, thus we cut the edge (c, e) and add an edge (c, b). The final BR$^+$-Tree is shown in Fig. 7(4) without up-edges. □

6.2 Tree Search

Given a directed graph G, one way to compute all SCCs of G is to contract all nodes in an SCC into a node. We take this idea

Algorithm 5 Tree-Search(G, T)

Input: A directed graph G, and a BR$^+$-Tree T.
Output: All SCCs of G.

1: **for all** edge $(u, v) \in E(G)$ in sequential order on disk **do**
2: **if** (u, v) is a backward edge and has not been merged **then**
3: $p \leftarrow$ the path on T from v to u;
4: contract all nodes in p into one node;
5: **for all** node $u \in V(T)$ **do**
6: **output** all nodes v represents as an SCC;

using a BR$^+$-Tree T constructed by sequential scan of $E(G)$ once. First, recall that a BR$^+$-Tree T is a spanning tree where a node u maintains its parent node in T plus one of its ancestors v by a backward-edge (u, v) in G. And there are no up-edges in G w.r.t T by Definition 5.1. We reemphasize that by definition an edge $(u, v) \in E(G)$ is an up-edge iff drank$(u, T) \geq$ drank(v, T) and there is no ancestor/descendant relationship between u and v in T. It is important to note that drank(u, T) gives the minimum depth of v that u can reach in BR$^+$-Tree T. Second, a partial SCC is represented by a cycle in BR$^+$-Tree which is a path with tree-edges from u to v plus a backward-edge from v to u. We denote it as $p = (v_1, v_2, \cdots, v_{k-1}, v_k)$ where $u = v_1 = v_k$ and $v = v_{k-1}$. All nodes on the path p belong to an SCC, and can be contracted into a node u with drank(v, T), due to the backward-edge (v, u). Importantly, there will not be any new up-edges when contracting such a path into a node in G w.r.t. the T contracted. We show it in two cases after all nodes in p are contracted in u. In the first case, let (v_i, w) be an edge from a node v_i on the path p to a node that does not appear in the path p. The edge (v_i, w) cannot be a new up-edge by definition. In the second case, let (w, v_i) be an edge from a node that does not appear in the path p to a node v_i on the path p. The edge (w, v_i) cannot be a new up-edge by definition. In addition, all other non-up-edges in G w.r.t. T cannot be up-edges because they are not affected by the contraction of nodes in p. This ensures that one scan of $E(G)$ can compute all SCCs.

We give the tree search algorithm in Algorithm 5. While scanning all edges in $E(G)$, if an edge is a non-tree edge w.r.t. BR$^+$-Tree, the edge can be either a backward-edge or a down-edge. A backward-edge will lead to contract more nodes into a node which may represent a partial SCC already. The contraction will not introduce new up-edges as discussed above, and it only needs one sequential scan of $E(G)$.

Example 6.2: We show how Algorithm 5 works using an example. Suppose we have constructed the BR$^+$-Tree as shown in Fig. 8(1). There are three backward edges, (c, b), (e, b) and (h, g). First, we contract nodes using the backward-edge (c, b), and obtain a partial SCC bc as shown in Fig. 8(2). Second, we contract nodes using the backward-edge (e, b). The nodes on the path (bc, d, e) are contacted into an SCC $bcde$ as shown in Fig. 8(3). Finally, we can contract nodes using the backward-edge (h, g) in a similar way. The 6 SCCs are represented by the 6 nodes in the tree in Fig. 8(4). □

7. SINGLE PHASE SINGLE TREE

The two-phase approach (Algorithm 4) uses a BR$^+$-Tree which consumes $3|V(G)|$ space, and requires at most depth(G) sequential scans of $E(G)$ in the construction phase and 1 sequential scan of $E(G)$ in the tree search phase. In this section, we discuss a single phase approach. We can reduce the space from $3|V(G)|$ to $2|V(G)|$ using BR-Tree based on the notion of BR$^+$-Tree. We can significantly reduce the number of sequential I/Os by further

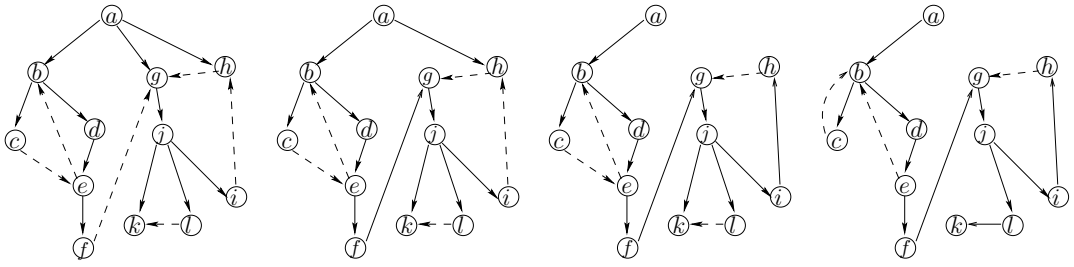


Figure 7: Tree Construction Example for G in Fig. 1

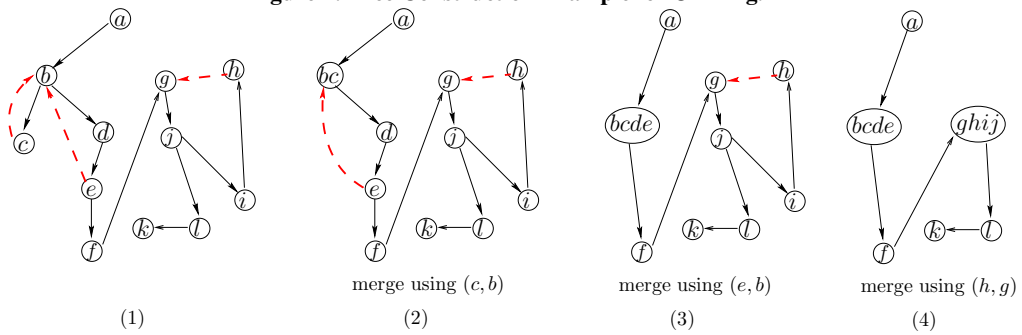


Figure 8: Tree Search Example of T in Fig. 7

Algorithm 6 1P-SCC(G)

Input: A directed graph G.
Output: All the SCCs in G.

```
1: G' ← G; T ← a spanning tree of G'; updated ← true;
2: while updated = true do
3:     updated ← false;
4:     for all edge (u, v) ∈ E(G') in sequential order on disk do
5:         if (u, v) is a backward edge then
6:             p ← the path on T from v to u;
7:             contract all nodes in p into a node;
8:             updated ← true;
9:         if (u, v) is an up-edge then
10:            T ← T ⇓ (u, v);
11:            updated ← true;
12:    early-acceptance(G', T) if the size of a large SCC is ≥ τ;
13:    early-rejection(G', T); {early-rejection and early-acceptance
       can be done together.}
14: for all node u ∈ V(G') do
15:    output all nodes that contract to a single u as an SCC;
```

reducing graph G to be processed in every iteration. The graph reduction is achieved by our optimization techniques, namely, early acceptance, early rejection, and batch edge reduction. Early acceptance is to contract cycles as soon as a backward-edge is found in an iteration. Early rejection is to remove those nodes that will not be used to form any new SCCs. Batch edge reduction is to process as many edges as possible together to reduce CPU cost by making full use of the main memory.

7.1 Early Acceptance

By early acceptance, we mean to contract cycles while constructing a BR-Tree. In this sense, all backward edges are eliminated while constructing a BR-Tree, therefore there is no need to maintain the backward-edges in the first tree construction phase for the later contraction in the second tree search phase. Because the contraction is done in tree construction, $\mathsf{drank}(u,T)$ can be easily computed as $\mathsf{drank}(u,T) = \mathsf{depth}(u,T)$.

The single phase algorithm is shown in Algorithm 6, which combines Tree-Construction and Tree-Search into one phase. Like Tree-Construction, it processes iteratively. In an iteration, for an up-edge (u,v), it conducts in a similar way as it is done in Tree-Construction (lines 11-13). The single phase algorithm needs to reshape the tree using pushdown ($\Downarrow$), but does not need to replace an up-edge to a backward-edge as done in Tree-Construction, because the same is replaced by the contraction. For a backward-edge (u,v), we find the tree path from v to u in T, and contract all nodes in the path (lines 7-10) as done in Tree-Search. The single phase algorithm stops when no up-edges and no backward-edges are found. For such a BR-Tree, a node in T represents an SCC in G.

The early-acceptance (line 12) reduces a graph G' into a smaller graph G' to be used in the next iteration if the algorithm finds that there are large SCCs whose size is greater than a given threshold τ. It is done as follows. Let T be the current spanning tree for G'. (1) For a node u in T, which represents a set of nodes V_u contracted, we exclude all induced edges in V_u from G', and use u to represent all nodes in V_u. (2) For all edges (v_i,v_j) where $v_i \in V_u$ and $v_j \notin V_u$, the edge (v_i,v_j) is replaced with (u,v_j). (3) For all edges (v_i,v_j) where $v_j \in V_u$ and $v_i \notin V_u$, the edge (v_i,v_j) is replaced with (v_i,u). As we will show later, a significant number of nodes and edges will be eliminated and a smaller graph will be used in the following iterations.

Example 7.1: We show an example of early acceptance. Suppose initially we construct a spanning tree T as shown in Fig. 9(1). We find that edge (e,b) is a backward edge. At this time, instead of eliminating up-edges, we merge nodes using edge (e,b). We merge nodes in the tree path (b,d,e) and create a new partial SCC node bde. The result is shown in Fig. 9(2). We find the up-edge (f,g), and thus we cut the edge (a,g) from T and add a new edge (f,g). The result after eliminating up-edge (f,g) is shown in Fig. 9 (3). Similarly, we eliminate up-edge (i,h) by cutting edge (a,h) and adding edge (i,h), and we eliminate up-edge (l,k) by cutting edge (j,k) and adding edge (l,k), and we get another tree as shown in Fig. 9 (4). We find the backward edge (h,g) and we merge nodes

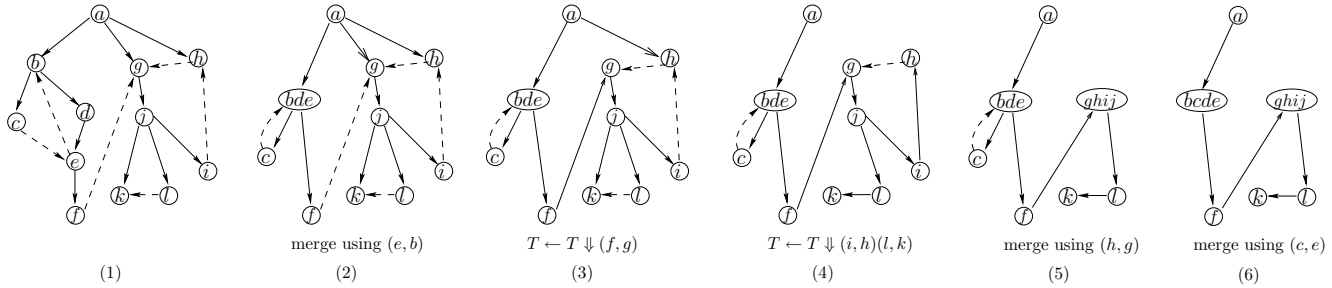


Figure 9: Early Acceptance Example for G in Fig. 1

Algorithm 7 early-rejection(G, T)

1: **for all** node u in G **do**
2: **if** $\mathsf{drank}(u) < \mathsf{drank}_{min}$ **or** $\mathsf{drank}(u) > \mathsf{drank}_{max}$ **then**
3: **output** the nodes u represents as an SCC;
4: remove u from T;

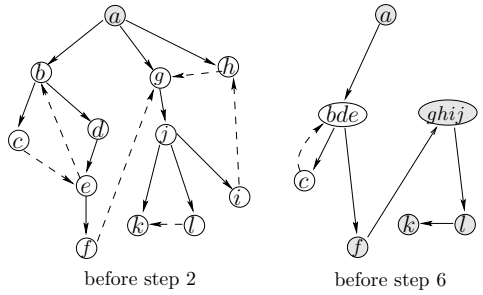


Figure 10: Early Rejection Example

in the path from g to h on T. We created an SCC node $ghij$, which is shown in Fig. 9 (5). Finally, we find backward edge (c, e), merge nodes in the path from e to c on T, and end up with the graph as shown in Fig. 9(6) with 2 SCCs, $bcde$ and $ghij$. □

We will discuss early-rejection (line 13) which removes the nodes that will not be involved in a new SCC, in the next subsection. Note that both early-acceptance and early-rejection can be done at the same time to reduce cost.

7.2 Early Rejection

We discuss how to remove those nodes that will not be involved in a new SCC in an early stage by early-rejection(G, T). The condition is specified by a min drank (drank_{min}) and a max drank (drank_{max}) for the graph G, w.r.t. its spanning tree T.

$$\mathsf{drank}_{min} = \min\{\mathsf{drank}(v), \text{ for all}(u, v) \text{ with } \mathsf{drank}(u) \geq \mathsf{drank}(v)\}$$
$$\mathsf{drank}_{max} = \max\{\mathsf{drank}(u), \text{ for all}(u, v) \text{ with } \mathsf{drank}(u) \geq \mathsf{drank}(v)\}$$

Given a spanning tree T of a graph G, for an edge (u, v), $\mathsf{drank}(u) \geq \mathsf{drank}(v)$ suggests that u and v are possibly in an SCC. drank_{min} and drank_{max} specify the upper and lower bound, respectively, over all such edges in a graph G. Here, drank_{min} suggests that any node u with $\mathsf{drank}(u)$ ($< \mathsf{drank}_{min}$) cannot be in a larger SCC than what SCC u may be involved in already, and drank_{max} suggests that any node u with $\mathsf{drank}(u)$ ($> \mathsf{drank}_{max}$) cannot be in a larger SCC than what SCC u may be involved in already. The condition holds because in the single phase approach, nodes are contracted during construction and $\mathsf{drank}(u) = \mathsf{depth}(u)$.

The early-rejection algorithm is shown in Algorithm 7. When the condition is held on a node u, all the nodes, represented by u as an SCC in T for G, are output, and u will be removed from T. As a result, all the edges connected to/from u can be excluded in the further iteration.

Algorithm 8 1PB-SCC(G)

Input: A directed graph G.
Output: All the SCCs in G.

1: $G' \leftarrow G$; $T \leftarrow$ a spanning tree of G'; $updated \leftarrow$ true;
2: **while** $updated =$ true **do**
3: $updated \leftarrow$ false;
4: divide $E(G')$ into batches $B = \{B_1, B_2, \cdots\}$ where $B_i \in B$ can fit in main memory;
5: **for all** $B_i \in B$ **do**
6: $G" \leftarrow T \cup B_i$;
7: compute all SCCs in $G"$ using an in-memory algorithm, and construct a *DAG* by contracting nodes in an SCC into one node;
8: let topological order for all nodes in $|V(G")|$ be $\{v_1, v_2, \cdots, v_{|V(G")|}\}$;
9: **for** $j = 1$ **to** $|V(G")|$ by the topological order **do**
10: **for all** $(v_i, v_j) \in E(G")$ **do**
11: **if** $\mathsf{drank}(v_i) \geq \mathsf{drank}(v_j)$ **then**
12: $T \leftarrow T \Downarrow (v_i, v_j)$; $update \leftarrow$ true;
13: early-acceptance(G', T) if the size of a large SCC is $\geq \tau$;
14: early-rejection(G', T); {early-rejection and early-acceptance can be done together.}
15: **for all** node $u \in V(G')$ **do**
16: **output** all nodes that contract to a single u as an SCC;

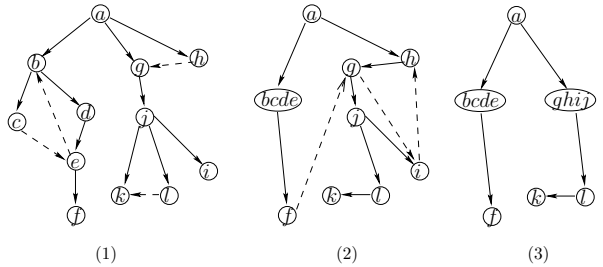


Figure 11: 1PB-SCC Construction Example for G in Fig. 1

Example 7.2: We show an example on early rejection. Consider the spanning tree T shown in Fig. 9(1) for G in Fig. 1. Here, $\mathsf{drank}_{min} = 1$ and $\mathsf{drank}_{max} = 4$. Node a can be early rejected because $\mathsf{drank}(a) = 0$. We can remove a from T. And the edges (a, b), (a, g), (a, h) do not need to be considered in the following iterations. The result after the early rejection is shown in Fig. 10 (before step 2). Consider the spanning tree T shown in Fig. 9(5). Here, $\mathsf{drank}_{min} = \mathsf{drank}_{max} = 1$. All nodes, f, k, l, and $ghij$ can be removed from the graph, as shown in Fig. 10 (before step 6). All related edges can be removed in further processing. □

7.3 Batch Edge Reduction

In the 1P-SCC algorithm (Algorithm 6), CPU cost is rather high. We explain it below. For an edge (u, v) (line 4), in order to determine whether (u, v) is a backward-edge/up-edge, by Definition 5.1, it needs to check whether there is an ancestor/descendant relationship between u and v over the spanning tree T. Assume that we use $O(|V(G')|)$ memory to maintain a spanning tree T for G'. T

is possibly updated frequently, because it will be updated when an up-edge is found. When T is updated, the existing techniques for ancestor/descendant relationship detection in constant time cannot be used. To find whether v is an ancestor of u, it needs to check whether v is on the path from u towards the root of T in main memory. Such CPU cost depends on the average length of a path, or in other words, the average depth in T. In Algorithm 6, when BR-Tree is reshaped by pushdown ($\Downarrow$) for an up-edge (u, v), the depth of all nodes under the subtree rooted at v increase by at least one. When more $\Downarrow$ operations are conducted on T, the average depth of nodes in T will be larger. Therefore, the CPU cost to check ancestor/descendant relationship increases. We discuss how to reduce the cost of computing ancestor/descendant relationships as well as the cost to maintain depth of nodes when finding up-edges on T below.

Our algorithm is shown in Algorithm 8 which enhances Algorithm 6 by batch processing to reduce CPU cost. Given memory M_B which is available to edges in main memory. First, we read all edges we can read from disk into main memory as a batch, and denote it as B_i (line 5). Second, we construct a graph G" in memory by the current T plus the edges in B_i, use an in-memory algorithm to compute all SCCs (Kosaraju-Sharir algorithm [3]), and construct a *DAG* of G" (lines 7-8). By the *DAG* of G", all SCCs are eliminated and therefore no backward edges exist in the *DAG*. The CPU cost for computing ancestor/descendant relationships for edges in G" are shared in the process of constructing *DAG*. Third, we construct a new BR-Tree T for the *DAG*, which is also for G" in memory (lines 9-12). In order to reduce the CPU cost for maintaining the depth of nodes when eliminating up-edges, we sort all nodes in G" in topological order which can be done in constructing *DAG* without extra cost. We process nodes in the topological order, and we can guarantee that all incoming nodes of v in G" have been processed. The depth of v in the new T can be simply computed as $\mathsf{drank}(v) = \max_{(u,v)\in E(G")}\{\mathsf{drank}(u) + 1\}$ using dynamic programming. In such a way, when eliminating (u, v) using $\Downarrow$ (line 12), we do not update the drank of all nodes in the subtree rooted at v, but only update the $\mathsf{drank}(v)$ by setting $\mathsf{drank}(v) \leftarrow \mathsf{drank}(u) + 1$. Fourth, like Algorithm 6, we conduct early-acceptance and early-rejection (line 13-14), and output all SCCs (lines 15-16).

CPU Cost Saving: Consider G' to be processed, where $m = |E(G')|$ and $n = |V(G')|$. By the 1P-SCC algorithm (lines 4-11), all nodes are in memory. Assume all m edges are processed one by one, and there are m' up-edges. Suppose the average cost for updating the drank values of nodes in a subtree is C_T and the average cost for up-edges checking is C_V. The CPU cost for the 1P-SCC algorithm is $C_{\text{1P-SCC}} = O(m' \cdot C_T + m \cdot C_V)$. It is important to note that C_T is to update all nodes in a subtree and C_V is to check a path from a node towards the root. Both are costly. On the other hand, by the 1PB-SCC algorithm, there are β batches to process (lines 4-12). The graph G' with n nodes and $n-1+\frac{m}{\beta}$ edges are constructed in each batch, where $n-1$ is the number of edges in the spanning tree in main memory, and $\frac{m}{\beta}$ is the number of edges can be loaded into main memory in a batch. In one batch, finding all SCCs using the Kosaraju-Sharir algorithm [3] in such a graph G' in memory will cost $O(2n - 1 + \frac{m}{\beta}) = O(\frac{m}{\beta} + n)$ time, where n is the number of nodes, and $n - 1 + \frac{m}{\beta}$ is the number of edges. And there are β batches, the total becomes $O(m + \beta n)$. When β is small, where a large number of edges can be processed in a batch, the 1PB-SCC algorithm is efficient. When $\beta = \frac{m}{n}$, which means n edges are processed in each batch, the CPU cost for the 1PB-SCC algorithm is $C_{\text{1PB-SCC}} = O(m)$ which is obviously better than the CPU cost for the 1P-SCC algorithm $C_{\text{1P-SCC}} = O(m' \cdot C_T + m \cdot C_V)$.

Example 7.3: We show an example of finding SCCs of the graph shown in Fig. 1 using Algorithm 8. The procedure is shown in Fig. 11. Suppose the initial BR-Tree is shown as the solid edges in Fig. 11(1). The G' is shown in Fig. 11(1) by adding four more edges (dotted lines) into memory. After finding SCCs in G', nodes b, c, d, and e are merged into one node $bcde$ and after topological sorting and performing dynamic programming by finding the longest paths on G', the new BR-Tree T is shown as the solid edges in Fig. 11(2). We then form a new G' as shown in Fig. 11(2) by adding three more edges (dotted lines) into memory. After finding SCCs and topological sorting on G', we find a new SCC $ghij$ by merging nodes g, h, i and j, and the final BR-Tree is shown in Fig. 11(3). □

7.4 I/O Efficiency

It is worth noting that the total number of I/O accesses depends on the number of iterations, where in an iteration it needs to access batches of edges of $E(G)$ for G sequentially. Reducing the effective number of iterations becomes a key point. Assume we can find all strongly connected components in L iterations, and assume on average P nodes are merged/rejected in the i-th iteration. These P nodes and Q edges between any pair of these P nodes will be eliminated in the following j-th iteration for $i < j \leq L$. The saving in the i-th iteration is $(P + 2Q)(L - i)b/B$, where b is the number of bytes to store a node, and B is the block size. In total, the number of I/Os we can save is

$$\sum_{i=1}^{L}(P + 2Q)(L - i)b/B = (P + 2Q)\frac{(L - 1)L}{2B}b$$

In addition, more space saved, the better to make batch edge reduction more effective, because $P/2$ more edges can be loaded in the next (i+1)-th iteration when P nodes are eliminated on average in i-th iteration. In the total L iterations,

$$\sum_{i=1}^{L}(P/2)(i - 1) = P(L - 1)L/4$$

more edges can be loaded in main memory. Our techniques significantly reduce the number of iterations and therefore the I/O accesses.

We show how effective our single phase algorithm is using a large real dataset as an example. WEBSPAM-UK2007 consists 105,896,555 web-pages 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. There exist 193,670 SCCs including 84,498,517 nodes. The biggest strongly connected component contains 68,582,555 nodes and the second biggest one contains 235,228 nodes. The smallest strongly connected component contains 2 nodes. Our 1PB-SCC algorithm with early acceptance and early rejection occurs 21 iterations to find all SCC. Without early acceptance and early rejection, we need more than 50 iterations. After the last iteration, the graph to be processed only consists of 35,769,598 edges. More than 99% of edges are pruned in the first 20 iterations. Table 1 shows information for the first 5 iterations. In the first 5 iterations, 29.5M nodes and 646M edges have been pruned. Among all the nodes pruned in the first 5 iterations, 9,757,730 nodes are pruned by early rejection and 22,591,413 nodes are pruned by early acceptance. Early acceptance outperforms early rejection because 80% nodes belong to an SCC and only 20% nodes do not belong to any strongly connected component. In the first iteration, by reducing 7.6M nodes, we can load additional 3.8M edges into the memory to process in a batch in the following iter-

Iteration	1	2	3	4	5
# of Nodes Reduced	7.6M	6.8M	5.4M	5.3M	4.4M
# of Edges Reduced	113M	145M	109M	142M	137M
% of Nodes Reduced	8.61%	7.66%	5.86%	5.98%	4.77%
% of Edges Reduced	3.02%	3.87%	2.93%	3.79%	3.66%

Table 1: Nodes and Edges Reduced in the First 5 Iterations

ations, and the more space we saved by pruning nodes, the more I/Os we can reduce.

8. PERFORMANCE STUDIES

In this section, we show our experimental results by comparing our algorithms with the external contraction based algorithm EM-SCC [10] and the semi-external *DFS* based algorithm DFS-SCC (Algorithm 2). Our algorithms are all based on BR-Tree: the basic 2P-SCC algorithm (Algorithm 3), the 1P-SCC algorithm which is with both early-acceptance and early-rejection, and 1PB-SCC (Algorithm 8) which is 1P-SCC plus batch processing. In 1P-SCC and 1PB-SCC, the parameter τ for early-acceptance is 0.5% of $|V(G)|$. That is, when an SCC is greater than 0.5% of $|V(G)|$, the algorithm will do early-acceptance the graph. The early rejection is processed in every 5 iterations. All the algorithms are implemented using Visual C++ 2005 and tested on a PC with Intel Core2 Quard 2.66GHz CPU and 3.43GB memory running Windows XP. The disk block size is 64KB. The default memory size M is the memory to hold $3|V(G)|$ plus one disk block, that is $M = 4 \times (3|V(G)|) + 64K$ where 4 is the number of bytes to keep a node. We set the max time cost to be 5 hours. If a test does not stop in the time limit, we will denote it using INF. In our experiments, we do not show the result of EM-SCC since it is much slower than all other algorithms and cannot stop in most cases.

Datasets: In our experiments, we use 4 large real datasets including a massive dataset, and several synthetic datasets.

The 4 real large datasets are: cit-patent, go-uniprot, citeseerx, and WEBSPAM-UK2007. The cit-patent[4] includes all citations in patents granted in the US between 1975 and 1999. The cit-patents contains 3,774,768 nodes and 16,518,947 edges, with average degree 4.37 per node. Go-uniprot includes the gene ontology and annotations from the UniProt dataset[5]. The graph of go-uniprot contains 6,967,956 nodes and 34,770,235 edges, with average degree 4.99 per node. The citeseerx is the complete citation graph as of March 2010[6]. The graph of citeseerx contains 6,540,399 nodes and 15,011,259 edges, with average degree 2.3 per node. Among the datasets, go-uniprot and cit-patents are dense graphs and citeseerx is a sparse graph. Because these real graphs always contain small SCCs, we add random edges to make it contain more and larger SCCs. For every graph, we add 10% more edges. WEBSPAM-UK2007[7] 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.

We also generate 3 different kinds of synthetic graph datasets, in order to test the scalability in all situations. All the graphs contain nodes from 30M to 70M with average degree varying from 3 to 7. A synthetic graph is generated as follows. We construct a graph G by randomly selecting all nodes in SCCs first. Then we add edges among the nodes in an SCC until all nodes form an SCC. Finally, additional random nodes and edges are added to the graph. For the Massive-SCC case, a graph contains at least a massive SCC with 200,000 to 600,000 nodes. For the Large-SCC case, a graph contains at least a large SCC with 4,000 to 12,000 nodes. For the Small-SCC case, the largest SCC in a graph is less than 100 nodes. The parameters and their default values are shown in Table 2.

[4]snap.stanford.edu/data

[5]www.uniprot.org

[6]citeseerx.ist.psu.edu

[7]barcelona.research.yahoo.net/webspam/datasets/uk2007/links/

Parameter	Range	Default
Size of $\|V\|$	30M,40M,50M,60M,70M	30M
Average Degree D	3,4,5,6,7	5
Size of Massive-SCC	200K,300K,400K,500K,600K	400K
Size of Large-SCC	4K,6K,8K,10K,12K	8K
Size of Small-SCC	20,30,40,50,60	40
Number of Massive-SCCs	1	1
Number of Large-SCCs	30,40,50,60,70	50
Number of Small-SCCs	6K,8K,10K,12K,14K	10K

Table 2: Range and Default Value for Parameters

Name	1PB-SCC	1P-SCC	2P-SCC	DFS-SCC
cit-patents (T)	24s	22s	701s	840s
go-uniprot (T)	22s	21s	301s	856s
citeseerx (T)	10s	8s	517s	669s
cit-patents (I/O)	16,031	13,331	133,467	667,530
go-uniprot (I/O)	26,034	47,947	471,540	619,969
citeseerx (I/O)	15,472	13,482	104,611	392,659

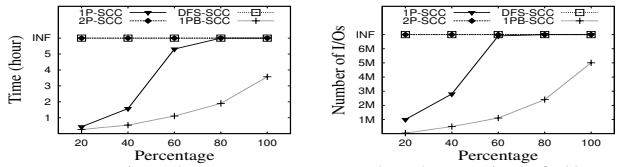
Table 3: Real Large Datasets (T: time, I/O: # of I/Os)

Exp-1 (Performance on Real Large Datasets): Table 3 shows the results for the 3 real large datasets: cit-patents, go-uniprot, and citeseerx. 1P-SCC runs fastest and DFS-SCC is the slowest. 1PB-SCC has similar performance with 1P-SCC. The processing time for 1PB-SCC is slightly larger than 1P-SCC in all the three datasets. This is because in all the three datasets, the number of nodes that are involved in at least one SCC is small. As a result, even if many edges are processed together in a batch by 1PB-SCC, the number of up-edges in each batch is small, and thus 1PB-SCC has to spend more time on topological sort all the other edges. Such cost makes 1PB-SCC slower than 1P-SCC. For the number of I/Os, 1PB-SCC consumes more I/Os than 1P-SCC in the cit-patents and citeseerx datasets, but less I/Os than 1P-SCC in the go-uniprot dataset. This is because the average size of SCCs in the go-uniprot dataset is smaller than the other two datasets. In such a situation, the probability for an SCC to be merged in a certain batch is high. As a result, although the CPU cost cannot be reduced, the number of iterations for 1PB-SCC is smaller than 1P-SCC in the go-uniprot dataset. Thus the I/O cost for 1PB-SCC is small.

Exp-2 (Performance on the Massive WEBSPAM-UK2007): We test the massive WEBSPAM-UK2007 which contains 105,895,908 nodes and 3,738,733,568 edges. To store all edges on disk, we need at least 30GB. In our testing, the default memory size is set as $M = 1.18G$. We conduct two testings. The first is to test by varying the size of its graph. We extract a subgraph of WEBSPAM-UK2007 G which is an induced subgraph over a subset of nodes selected in G, ranging from 20% to 100%. The second is to test by varying the internal memory from 1GB to 3GB. The results are shown in Fig. 12 and Fig. 13 respectively.

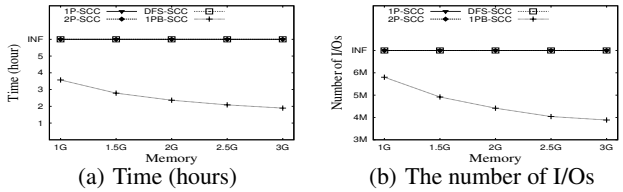
Fig. 12(a) shows the time cost of algorithms when varying the graph size, and Fig. 12(b) shows the corresponding I/O cost. 1PB-SCC can find all SCCs for any sizes of the graph. 1P-SCC can find all SCCs when the size of graph is below 60% of the original graph. 1PB-SCC performs well to handle a massive SCC which contains more than 60 million nodes by our optimization techniques. Fig. 13 shows the effectiveness of our algorithms when there is additional memory space available. Even with a large main memory, DFS-SCC, 2P-SCC, and 1P-SCC cannot compute all SCCs. 1PB-SCC can make full use of the additional memory with additional batch processing.

Exp-3 (Vary Node Size $|V|$ in Synthetic Data): We vary the num-



(a) Time (hours) (b) The number of I/Os

Figure 12: WEBSPAM-UK2007: Varying Node Size



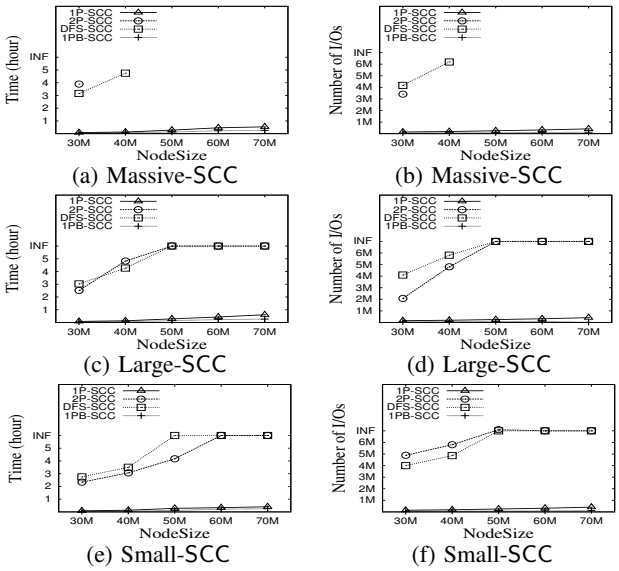
(a) Time (hours) (b) The number of I/Os

Figure 13: WEBSPAM-UK2007: Vary Memory

ber of nodes $|V(G)|$ in the graph G from 30,000,000 to 70,000,000 and we have three sets of graphs. Fig. 14(a) and Fig. 14(b) show the processing time and number of I/Os for Massive-SCC graphs. Fig. 14(c) and Fig. 14(d) show the performance for Large-SCC graphs, Fig. 14(e) and Fig. 14(f) show the performance for Small-SCC graphs. The average degree for each graph is 5. When the graph size increases, the processing time and the number of I/Os for all tests increase. For Massive-SCC, DFS-SCC increases sharply, and costs much more time and I/O. 2P-SCC cannot stop within the time limit if the size of graph is more than 40M. This is because when the graph size increases, the depth and the width of BR-Tree will also increase and the cost to calculate the dlink of edges increases very fast. The cost of 1P-SCC is much smaller. This is because the graph size decreases significantly by early-acceptance and early-rejection. 1PB-SCC performs best. This is because when the memory can hold a large number of edges, in addition to early-rejection and early-acceptance, 1PB-SCC can build the desired BR-Tree without checking each edge one by one. The topological sort reduces the cost significantly, especially when the size of an SCC is large. Although 1PB-SCC has smallest time cost, the gap of I/O between 1PB-SCC and 1P-SCC is very small, because 1PB-SCC aims to reduce the CPU cost of checking up-edges other than the I/O cost. The results for Large-SCC and Small-SCC are shown in Fig. 14(c), Fig. 14(d), Fig. 14(e), and Fig. 14(f). They share the similarity with Massive-SCC.

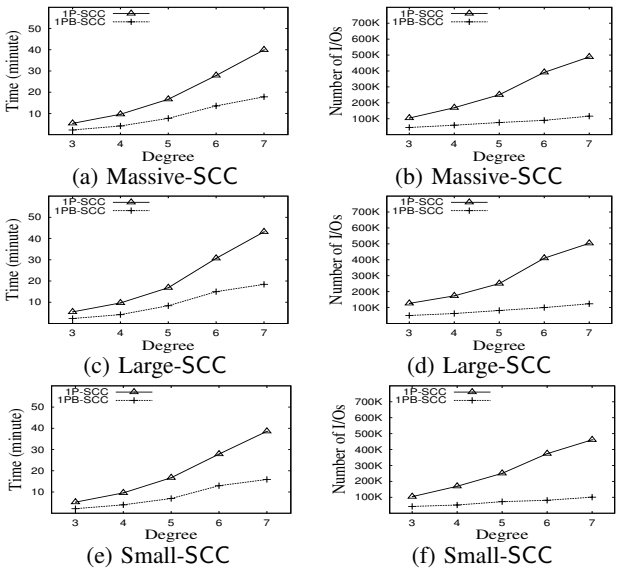
Exp-4 (Vary Average Degree in Synthetic Data): We vary the average degree D of nodes in a graph from 3 to 7. We also test on three sets of graphs with Massive-SCC, Large-SCC, and Small-SCC respectively. In each test, the default number of SCCs and the default average size are used. When the average degree of graph increases, the processing time and I/O cost increase for all algorithms. Among all algorithms, 1PB-SCC performs best for both time cost and I/O cost. The increasing rate is very slow. This is because that even higher degree means larger search space and more I/Os, it also indicates that more edges will stay in one SCC. 1PB-SCC takes advantage of this by merging SCCs according to the partial graph in the internal memory. The results are shown in Fig. 15.

Because the gap between DFS-SCC/2P-SCC and the others is big, we omit DFS-SCC and 2P-SCC in Fig. 15. DFS-SCC and 2P-SCC can only handle with degree 3 and 4. The DFS-SCC and 2P-SCC increase sharply, because when the average degree increases, the total number of edges that need to be checked increases, and more edges will introduce more search space and more iterations that needed to find out the *DFS* tree or BR-Tree. When



(a) Massive-SCC (b) Massive-SCC
(c) Large-SCC (d) Large-SCC
(e) Small-SCC (f) Small-SCC

Figure 14: Synthetic Data: Vary Node Size



(a) Massive-SCC (b) Massive-SCC
(c) Large-SCC (d) Large-SCC
(e) Small-SCC (f) Small-SCC

Figure 15: Synthetic Data: Vary Degree

the size of SCC is small, 2P-SCC outperforms DFS-SCC, because each SCC contains a small number of nodes. These nodes are easy to be put together when searching BR-Tree. When the size of SCC becomes very large, 2P-SCC does perform well. But, when the size of SCC is large but not enough, DFS-SCC may outperform 2P-SCC. Consider degree 4. In Massive-SCC, DFS-SCC and 2P-SCC take 4.45 and 4.18 hours. In Large-SCC, DFS-SCC and 2P-SCC take 3.56 and 4.18 hours. In Small-SCC, DFS-SCC and 2P-SCC take 3.05 and 2.81 hours.

Exp-5 (Vary SCC Size in Synthetic Data): We vary the size of each SCC in each set of the graphs. For Massive-SCC graphs, we vary the size of SCC from 200,000 to 600,000. The results are shown in Fig. 16(a) and Fig. 16(b). Only 1P-SCC and 1PB-SCC can find all SCCs within the time limit. All the other algorithms cannot. 1PB-SCC performs best. The results for Large-SCCs are shown in Fig. 16(c) and Fig. 16(d). The results for Small-SCC are shown in Fig. 16(e) and Fig. 16(f). 2P-SCC cannot compute all SCCs for Massive-SCC and Large-SCC cases. For Small-SCC, 2P-

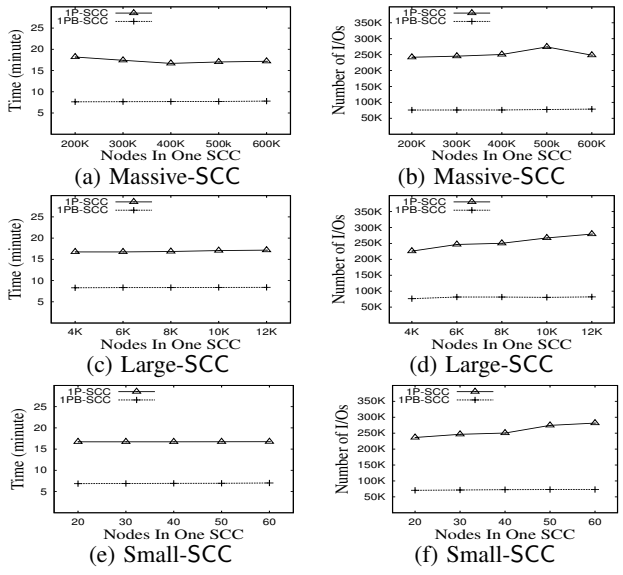


Figure 16: Synthetic Data: Vary SCC Size

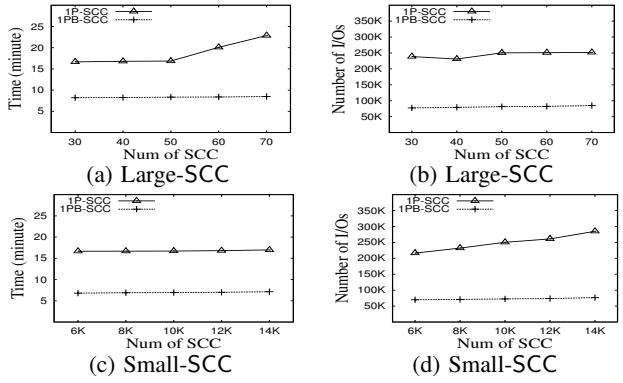


Figure 17: Synthetic Data: Vary the Number of SCCs

SCC takes 3.51, 4.07, and 4.17 hours when the numbers of SCCs in an SCC are, 20, 30, and 40, respectively. DFS-SCC cannot process any cases.

Exp-6 (Vary the Number of SCCs in Synthetic Data): We also test different number of SCCs for Large-SCC and Small-SCC, because we only create one SCC for Massive-SCC. The results are shown in Fig. 17. For Large-SCC, we vary the number of SCCs from 30 to 70 and each SCC contains 8,000 nodes. For Small-SCC, we vary the number of SCCs from 6K to 14K. For all tests, both 1PB-SCC and 1P-SCC can find all SCCs, and 1PB-SCC outperforms 1P-SCC. 2P-SCC cannot handle Large-SCC graphs, and takes hours for Small-SCC graphs. DFS-SCC cannot process any cases.

9. CONCLUSIONS

In this paper, we study new I/O efficient semi-external algorithms to find all SCCs for a large directed graph. We propose a new two phase algorithm with a tree construction followed by a tree search to find all SCCs. The two phase algorithm uses a bounded number of sequential scans of the graph. In order to further improve its I/O efficiency and reduce CPU cost, we propose a new single phase algorithm which combines the two phases into one phase. The three new optimization techniques we proposed along with the single phase algorithm are early acceptance, early rejection, and batch processing. By early acceptance, we contract a partial SCC into a node in an early stage while constructing a tree. By early rejection, we remove nodes that will not participate in any SCCs while constructing the tree in an early stage. The two techniques reduce the number of I/Os by reducing the graph to deal with as a smaller graph in iterations. In addition, we use batch processing technique to reduce the CPU cost. Our extensive performance studies confirm that we can significantly reduce the number of I/Os and CPU cost, and compute all SCCs that cannot be done by others in limited time.

10. ACKNOWLEDGEMENTS

The work was partially supported by grant of the Research Grants Council of the Hong Kong SAR, China No. 418512 and the fifth author was partially supported by NSFC61232006, NSFC61021004, ARCDP110102937 and ARCDP120104168.

11. REFERENCES

[1] J. Abello, A. L. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3), 2002.

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

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[4] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory bfs algorithms. In *Proc. of SODA'06*, 2006.

[5] 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.

[6] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory bfs implementation. In *Proc. of ALENEX'07*, 2007.

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

[8] 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.

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

[10] A. Cosgaya-Lozano and N. Zeh. A heuristic strong connectivity algorithm for large graphs. In *Proc. of SEA'09*, 2009.

[11] R. Dementiev, P. Sanders, D. Schultes, and J. F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, 2004.

[12] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1), 2010.

[13] J. Hellings, G. H. Fletcher, and H. Haverkort. Efficient external-memory bisimulation on dags. In *Proc. of SIGMOD'12*, 2012.

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

[15] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear i/o. In *Proc. of ESA'02*, 2002.

[16] U. Meyer and V. Osipov. Design and implementation of a practical i/o-efficient shortest paths algorithm. In *Proc. of ALENEX'09*, 2009.

[17] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proc. of ESA'03*, 2003.

[18] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In *Proc. of ESA'06*, 2006.

[19] J. F. Sibeyn. External connected components. In *Proc. of SWAT'04*, 2004.

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

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

[22] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1), 2010.