Contract & Expand: I/O Efficient SCCs Computing

Zhiwei Zhang[1], Lu Qin[3], Jeffrey Xu Yu[1,2]

[1] *The Chinese University of Hong Kong, Hong Kong, China*

[2] *Key Laboratory of High Confidence Software Technologies Ministry of Education (CUHK Sub-Lab)*

[1] *{zwzhang,yu}@se.cuhk.edu.hk*

[3] *Centre for Quantum Computation and Intelligent Systems, University of Technology, Sydney, Australia*

Lu.Qin@uts.edu.au

***Abstract*—As an important branch of big data processing, big graph processing is becoming increasingly popular in recent years. Strongly connected component (SCC) computation is a fundamental graph operation on directed graphs, where an SCC is a maximal subgraph S of a directed graph G in which every pair of nodes is reachable from each other in S. By contracting each SCC into a node, a large general directed graph can be represented by a small directed acyclic graph (*DAG*). In the literature, there are I/O efficient semi-external algorithms to compute all SCCs of a graph G, by assuming that all nodes of a graph G can fit in the main memory. However, many real graphs are large and even the nodes cannot reside entirely in the main memory. In this paper, we study new I/O efficient external algorithms to find all SCCs for a directed graph G whose nodes cannot fit entirely in the main memory. To overcome the deficiency of the existing external graph contraction based approach that usually cannot stop in finite iterations, and the external *DFS* based approach that will generate a large number of random I/Os, we explore a new contraction-expansion based approach. In the graph contraction phase, instead of contracting the whole graph as the contraction based approach, we only contract the nodes of a graph, which are much more selective. The contraction phase stops when all nodes of the graph can fit in the main memory, such that the semi-external algorithm can be used in SCC computation. In the graph expansion phase, as the graph is expanded in the reverse order as it is contracted, the SCCs of all nodes in the graph are computed. Both graph contraction phase and graph expansion phase use only I/O efficient sequential scans and external sorts of nodes/edges in the graph. Our algorithm leverages the efficiency of the semi-external SCC computation algorithm and usually stops in a small number of iterations. We further optimize our approach by reducing the size of nodes and edges of the contracted graph in each iteration. We conduct extensive experimental studies using both real and synthetic web-scale graphs to confirm the I/O efficiency of our approaches.**

I. Introduction

Graph is an important data structure to model complex relationships among entities. A road network, a social network, and the entire WWW can be modelled as graphs, and all such graphs are huge. In this paper, we study the problem of strongly connected component (SCC) computation, which is a fundamental graph operation on 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.

Computing SCCs on large graphs is highly demanded by many real applications that need topological sort, reachability query processing, and graph pattern matching in graph processing. (1) 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 are merged into one node. This is done by finding all SCCs in the graph. In [16], Hellings et al. propose an efficient 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. (2) 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, such as [25]. (3) 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 [15].

In the literature, there are efficient in-memory and I/O efficient semi-external algorithms to compute all SCCs of a directed graph G. An in-memory algorithm requires G to reside entirely in memory and a semi-external algorithm requires all nodes of G to reside entirely in memory. For the in-memory algorithm, 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. For the semi-external algorithm, Zhang et al. [26] propose an I/O efficient algorithm to compute all SCCs by constructing a special in-memory spanning tree of G using sequential scans of the graph on disk. However, due to the fact that the sizes of many real large graphs keep growing rapidly, even the nodes of a graph cannot reside entirely in the main memory. For example, the social network graph in Facebook contains 1.11 billion active nodes and more than 150 billion edges.[1] As a small part of the entire web graph, WEBSPAM-

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

UK2007[2] contains 105,896,555 pages and 4 billion edges in 114,529 hosts in the .UK domain in May 2007.

In order to handle a graph $G(V, E)$ where V cannot fit entirely in memory M, a naive way to externalize the in-memory *DFS* requires $O(|E|)$ I/Os. Chiang et al. [10] propose an algorithm with I/O complexity $O(|V| + \frac{|V|}{M} \cdot scan(|E|) + sort(|E|))$. Later, Kumar and Schwabe [17] and Buchsbaum et al. [8] improve the I/O complexity to $O((|V|+\frac{|E|}{B})\cdot\log_2\frac{|V|}{B} + sort(|E|))$ by maintaining the nodes that should not be traversed using tournament trees [17] and buffered repository trees [8] respectively, where B is the disk block size. Despite their theoretical guarantees, these algorithms are considered impractical for general directed graphs that encountered in real applications, due to the large number of random I/Os generated. Cosgaya-Lozano et al. [13] study a heuristic external algorithm to compute all SCCs for a directed algorithm based on contraction used by Chiang et al. [10] which is for undirected graphs. But, for directed graphs, the algorithm may end up an infinite loop and cannot compute all SCCs.

In this paper, we study external algorithms for SCC computation. The main contributions of this work are summarized below. Firstly, we analyze the deficiency of the existing *DFS* based algorithm [8] that consumes a large number of random I/Os and the contraction based algorithm [13] that needs large number of iterations and may end up an infinite loop. Secondly, we propose a new two-phase algorithm with graph contraction followed by graph expansion. In graph contraction, we only contract the number of nodes of the graph with bounded number of new edges generated. We stop when all nodes can fit in the main memory and process the contracted graph using an I/O efficient semi-external algorithm. Using nodes contraction, the number of iterations can be significantly reduced comparing to [13]. In graph expansion, the removed nodes are put back into the graph in a reverse order of their removal, while the SCCs of all nodes are computed. We analyze the I/O cost of our approach and show that our algorithm can significantly reduce the number of random I/Os comparing to [8]. Thirdly, we introduce techniques to further reduce the I/O cost of our algorithm by reducing the number of nodes and edges generated in each iteration of graph contraction. Finally, we conduct extensive experimental studies using both real and synthetic web-scale graphs to confirm the I/O efficiency of our approaches.

The remainder of this paper is organized as follows. In Section II, we discuss the preliminaries and give the problem statement for computing SCCs. In Section III, we discuss existing solutions on both external and semi-external algorithms for computing SCCs. In Section IV, we analyze the deficiency of existing external approaches and outline our two-phase approach with graph contraction followed by graph expansion. We introduce the graph contraction phase in Section V and discuss the graph expansion phase in Section VI. We study optimization techniques to further reduce the I/O cost of our algorithm in Section VII. In Section VIII, we report our experimental results. We discuss the related work in Section IX and conclude the paper in Section X.

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

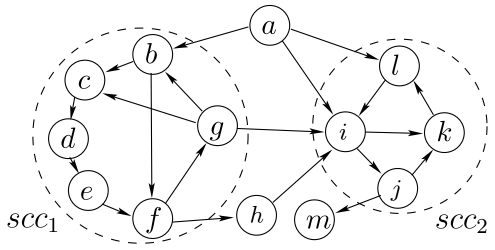


Fig. 1. A Graph G with 2 SCCs

II. 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. For simplicity, we also use V and E to denote $V(G)$ and $E(G)$ respectively, when it is obvious. Each node $v \in V(G)$ has a unique identity in G, denoted by $\mathsf{id}(v)$, which specifies a unique total order among all nodes in G.

For each node $v \in V(G)$, we use $\mathsf{nbr}_{in}(v, G)$ to denote the set of in-neighbors of v in G, and we use $\mathsf{nbr}_{out}(v, G)$ to denote the set of out-neighbors of v in G. We use $\mathsf{nbr}(v, G)$ to denote the set of in-neighbors and out-neighbors of v in graph G. We have $\mathsf{nbr}_{in}(v, G) = \{u|(u, v) \in E(G)\}$, $\mathsf{nbr}_{out}(v, G) = \{u|(v, u) \in E(G)\}$ and $\mathsf{nbr}(v, G) = \mathsf{nbr}_{in}(v, G) \cup \mathsf{nbr}_{out}(v, G)$. We use $\mathsf{deg}_{in}(v, G)$, $\mathsf{deg}_{out}(v, G)$, and $\mathsf{deg}(v, G)$ to denote the in-degree, out-degree and total degree of a node $v \in V(G)$ respectively. We have $\mathsf{deg}_{in}(v, G) = |\mathsf{nbr}_{in}(v, G)|$, $\mathsf{deg}_{out}(v, G) = |\mathsf{nbr}_{out}(v, G)|$ and $\mathsf{deg}(v, G) = |\mathsf{nbr}(v, G)|$.

Given a graph $G(V, E)$, a vertex cover of G is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then either $u \in V'$ or $v \in V'$. In other words, each vertex covers its incident edges and a vertex cover of G is a set of vertices that covers all $E(G)$ [12]. It is trivial that any superset for a vertex cover is still a vertex cover. A minimum vertex cover is a vertex that has the minimum set cardinality among all the vertex covers of G.

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 $v_i \to v_j$, iff there exists a path from v_i to v_j in G. A node v_i cannot reach a node v_j in G, denoted $v_i \not\to v_j$, iff there exists no path from v_i to v_j in G. A node v_i is strongly connected to a node v_j, denoted as $v_i \leftrightarrow v_j$, iff $v_i \to v_j$ and $v_j \to v_i$ in G. Here, $\leftrightarrow$ is an equivalence relation, which is reflexive, symmetric, and transitive. We use $v_i \not\leftrightarrow v_j$ 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_s $(\subseteq V)$ such that, for every pair of nodes v_i and v_j in V_s, $v_i \leftrightarrow v_j$, and for every pair of nodes v_i and v_k where $v_i \in V_s$ and $v_k \notin V_s$, $v_i \not\leftrightarrow v_k$. Each SCC of G has a unique identity. For each node $v \in V(G)$, we use $\mathsf{SCC}(v, G)$ to denote the SCC that v belongs to. For any two nodes $u \in V(G)$ and $v \in V(G)$, $u \leftrightarrow v \Leftrightarrow \mathsf{SCC}(u, G) = \mathsf{SCC}(v, G)$. For any set of nodes $S \subseteq V(G)$, we use $\mathsf{SCC}(S, G)$ to denote the set of SCCs that

Algorithm 1 DFS-SCC(G)

```
Input: a directed graph $G(V, E)$.
Output: all the SCCs in $G$.

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;
```

nodes in S belong to, i.e., $\mathsf{SCC}(S, G) = \{\mathsf{SCC}(v, G) | v \in S\}$.

Example 2.1: Fig. 1 shows a graph G with 13 nodes and 20 edges. For nodes b and e, $b \leftrightarrow e$ since $b \rightarrow e$ through path (b, c, d, e) and $e \rightarrow b$ through path (e, f, g, b). There are 2 SCCs, SCC_1 and SCC_2, where $SCC_1 = \{b, c, d, e, f, g\}$ and $SCC_2 = \{i, j, k, l\}$. □

In this paper, we assume that the graph G cannot reside entirely in the main memory. For all I/O operations, we follow the standard I/O model in [2]. We use M to denote the size of the main memory and use B to denote the size of each block on disk. All data are read/written in blocks. We assume $M \geq 2 \times B$. We use $sort(m)$ to denote the I/O cost of external sorting m elements on disk and we use $scan(m)$ to denote the I/O cost of sequentially scanning all m elements once on disk. In the I/O model [2], we have $scan(m) = \Theta(\frac{m}{B})$, and $sort(m) = \Theta(\frac{m}{B} \cdot \log_{\frac{M}{B}} \frac{m}{B})$.

Problem Statement: compute all strongly connected components (SCCs) for a large directed graph $G(V, E)$ with limited memory M. Here $2 \times B \leq M < \|G\|$ where B is the block size and $\|G\|$ is the space for the entire graph G.

III. Existing Solutions

In the literature, there are two external algorithms and two semi-external algorithms to compute all SCCs for a directed graph G. A semi-external algorithm assumes that all nodes of the graph can reside entirely in the main memory, i.e., $M \geq c \times |V|$ for a constant c, and an external algorithm only assumes that at least two disk blocks can fit in the main memory, i.e., $M \geq 2 \times B$. The two external algorithms are based on contraction [13] and external depth first search (*DFS*) [8] respectively, and the two semi-external algorithms are based on semi-external *DFS* [23] and a special spanning tree BR-Tree of G respectively [26].

Contraction Based EM-SCC: Cosgaya-Lozano et al. [13] provide a heuristic algorithm, called EM-SCC, to compute all SCCs for a large directed graph. The idea is taken from the contraction-based algorithm to compute all connected components for an undirected graph [10]. Given limited memory M, EM-SCC compresses a graph iteratively by contraction until the 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}. 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.

Unlike the algorithm [10] which ensures an undirected graph G can fit into main memory in a log number of iterations, EM-SCC cannot stop in a finite number of iterations for directed graphs 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. Either case happens frequently in real world graphs.

Depth First Search Based DFS-SCC: DFS-SCC computes all SCCs by simulating the in-memory Kosaraju-Sharir algorithm [3] which traverses the graph G twice using depth first search. The framework of DFS-SCC is shown in Algorihtm 1. In the first time traversal, it obtains 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 are in the same SCC. The key operation of DFS-SCC is DFS-Tree which constructs a *DFS* tree for a graph G.[3] Thus, we introduce existing external *DFS* algorithms below.

In the literature, external *DFS* algorithms are proposed by Kumar et al. [17] and Buchsbaum et al. [8]. [8] is an improvement for [17]. The basic idea of [8] is to simulate the internal memory *DFS* algorithm by maintaining the visited nodes using an augmented external (2,4)-tree, called buffer repository tree (BRT). The I/O complexity for *DFS* using the algorithm in [8] is $O((|V| + \frac{|E|}{B}) \log_2 \frac{|V|}{B} + sort(|E|))$. Comparing to the trivial external memory *DFS* which consumes $O(|E|)$ I/Os, the algorithm in [8] does not improve significantly especially on sparse graphs [5], and a large number of random I/Os are generated by the algorithm in [8]. Thus, computing SCCs based on external *DFS* is not I/O efficient.

Example 3.1: Consider G in Fig. 1. The first *DFS* traverses G in $abcdefgijklmh$, and its decreasing postorder is $abcdefhgijmkl$. With the decreasing postorder, in the second *DFS*, the root v_0 of the *DFS* tree has 5 subtrees representing 5 SCCs $\{a\}$,$\{b, c, d, e, f, g\}$,$\{h\}$,$\{i, j, k, l\}$, and $\{m\}$. □

Semi-External Approach Semi-SCC: Since there is no efficient external algorithm to compute all SCCs for a graph G in the literature, some papers focus on semi-external algorithms by relaxing the condition $2 \times B \leq M < \|G\|$ to $c \times |V| \leq M < \|G\|$, where c is a constant. Following the framework in Algorithm 1, a semi-external algorithm can be designed by applying semi-external *DFS* [23] when generating the *DFS* tree (line 1 and line 4). In [23], given a graph G, a spanning tree T of G is maintained in memory using $O(|V|)$ space. The algorithm iteratively scans the edges of G on disk and updates T until T becomes a *DFS* tree. The semi-external

[3]We do not need to construct a *DFS* tree explicitly. We only need to obtain the *DFS* order of all nodes in the graph.

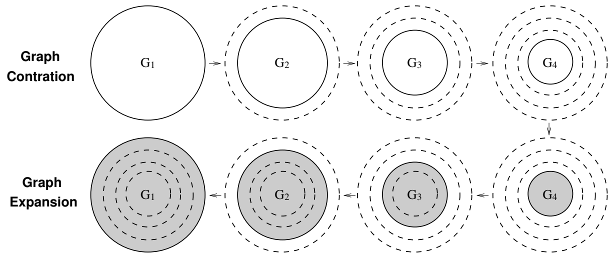


Fig. 2. A Solution Overview

DFS algorithm in [23] is much more efficient than the external *DFS* algorithm in [8]. However, it is not optimized for SCC computation using the Kosaraju-Sharir algorithm (Algorithm 1). This is because Algorithm 1 needs to maintain a total order (decreasing postorder) of nodes in the first *DFS* (line 1-2), to be used in the second *DFS* (line 4). As a result, in the first *DFS*, nodes cannot be contracted or removed even if partial SCCs have been found.

In order to improve the semi-external *DFS* based approach using Algorithm 1, Zhang et al. [26] develop a more efficient semi-external algorithm to compute all SCCs, by defining a weaker order based on the depth of a node in a spanning tree (BR-Tree) of G. Using the weaker order, only one BR-Tree needs to be constructed in memory. The algorithm iteratively scans the edges of G on disk and updates the BR-Tree until no new SCC can be found. When updating the BR-Tree, each partial SCC can be contracted into one node, and nodes that will not contribute to any new SCCs can be removed from G. The I/O cost to compute all SCCs is largely reduced.

Although Semi-SCC is much more efficient than external *DFS* and external contraction based approaches, it assumes that $M \geq c \times |V|$. In this paper, we aim to design an efficient external algorithm to compute SCCs when $M < c \times |V|$.

IV. A New Contraction-Expansion Approach

Given that the contraction based EM-SCC may end up infinite iterations, and the *DFS* based DFS-SCC can generate a large number of random I/Os. In this paper, we propose a novel contraction-expansion based external algorithm to compute all SCCs for a graph G. Instead of accessing each node one by one in EM-SCC, our algorithm computes the SCCs of all nodes in batches in order to reduce the number of random I/Os used to access each node. Our algorithm is processed in two phases, namely, graph contraction and graph expansion.

In the graph contraction phase, a list of graphs G_1, G_2, $\cdots$, G_l are generated, where $G_1 = G$, and for each $1 \leq i < l$, G_{i+1} is generated by removing a batch of nodes from G_i, i.e., $V(G_{i+1}) \subset V(G_i)$. It stops until all nodes of the graph can fit in memory, i.e., $c \times |V(G_l)| \leq M$ for a constant c. In other words, SCCs of G_l can be computed using Semi-SCC.

In the graph expansion phase, after computing all SCCs of G_l using Semi-SCC, the removed nodes are added back to the graph in the reverse order of their removal in the graph contraction phase, i.e., the lastly removed nodes in the graph contraction phase are firstly added back in the graph expansion phase. When a node is added back, the SCC it belongs to is computed. More specifically, given that all

Algorithm 2 Ext-SCC(G)

Input: a directed graph $G(V, E)$.
Output: all the SCCs in G.

```
1: $G_1(V_1, E_1) \leftarrow G(V, E)$; $i \leftarrow 1$;
2: **while** $V_i$ can not fit in memory **do**
3:     contract $G_i(V_i, E_i)$ to $G_{i+1}(V_{i+1}, E_{i+1})$ with $V_{i+1} \subset V_i$;
4:     $i \leftarrow i + 1$;
5: compute all SCCs in $G_i(V_i, E_i)$ using Semi-SCC;
6: **while** $i > 1$ **do**
7:     $i \leftarrow i - 1$;
8:     compute SCCs for all nodes in $V_i - V_{i+1}$;
9:     combine SCCs in $V_i - V_{i+1}$ with SCCs in $V_{i+1}$;
10: output all SCCs in $V_1$;
```

SCCs of G_l are computed using Semi-SCC, we compute the SCCs for nodes $V(G_{l-1}) - V(G_l)$, $V(G_{l-2}) - V(G_{l-1})$, $\cdots$, $V(G_1) - V(G_2)$ in order. Since $V(G) = V(G_1) = V(G_l) \cup (V(G_{l-1}) - V(G_l)) \cup (V(G_{l-2}) - V(G_{l-1})) \cdots \cup (V(G_1) - V(G_2))$, the SCCs for all nodes are computed after graph expansion.

The two phases are illustrated in Fig. 2 where gray parts represent the nodes whose SCCs are computed. Our algorithm Ext-SCC to compute all SCCs in a graph G is shown in Algorithm 2. Initially, $G_1 = G$ and $i = 1$ (line 1). The graph contraction phase is shown in line 2-4. It iteratively generates a new graph G_{i+1} based on G_i with a smaller number of nodes and stops when the nodes of the graph can fit in the main memory. The graph expansion phase is shown in line 5-9. It first computes all SCCs of $V(G_i)$ using the semi-external algorithm Semi-SCC (line 5). Then it iteratively computes all SCCs for nodes $V(G_i) - V(G_{i+1})$ with decreasing order of i (line 6-9). Finally, all SCCs of $V(G_1)$ are output since $G_1 = G$ (line 10).

Below, we analyze the deficiencies of the existing external *DFS* based DFS-SCC algorithm and external contraction based EM-SCC algorithm, comparing to our Ext-SCC algorithm.

For DFS-SCC, as introduced in Section III, the I/O complexity of the algorithm is $O((|V| + \frac{|E|}{B}) \log_2 \frac{|V|}{B} + sort(|E|))$, which is higher than $O(|V|)$. The high I/O cost is produced by the large number of random accesses when visiting nodes one by one on disk, and it is costly to access each node using one random disk access. In our algorithm, instead of processing nodes one by one, we process the nodes of G in batches. As shown in Algorithm 2, we contract the graph G_i by removing a batch of nodes $V(G_i) - V(G_{i+1})$ with increasing i in the graph contraction phase, and add the removed nodes back to compute the SCCs for each batch of removed nodes in a reverse order in the graph expansion phase. As we will show later, both the graph contraction phase and the graph expansion phase are processed using only sequential scans and external sorts. In such a way, all nodes/edges of the graph can be processed in blocks and the number of random accesses is minimized.

For EM-SCC, it is a contraction based algorithm with some critical problems as introduced in Section III. Our contraction-expansion based algorithm Ext-SCC can solve the problems. Firstly, EM-SCC may not be able to terminate. In our Ext-SCC algorithm, when generating G_{i+1} from G_i, we make sure that $V(G_{i+1})$ is a proper subset of $V(G_i)$ by removing a batch of nodes from $V(G_i)$. Thus, our algorithm can always terminate.

Furthermore, our stop condition only requires that all nodes $V(G_i)$ can fit in the main memory, which is usually much smaller than the size of the whole graph G_i. Our stop condition is easier to be satisfied comparing to EM-SCC which requires the whole graph G_i to fit in the main memory. Secondly, even if EM-SCC can terminate in a finite number of iterations, the contraction is unstable since it relies largely on the order of edges stored on disk. It is possible that only a small number of nodes are contracted in each iteration. In our Ext-SCC algorithm, the selection of nodes to be removed in each iteration does not rely on the order of edges stored on disk, and thus our algorithm is much more stable than EM-SCC. As confirmed in our experiments, our algorithm can usually terminate in a small number of iterations.

In the following, we discuss the two phases, and propose techniques to further minimize the I/O cost.

V. Graph Contraction

In this section, we introduce how to contract a graph G_{i+1} from graph G_i. As introduced before, when constructing G_{i+1}, we should make sure that the nodes of G_{i+1} is a proper subset of G_i by removing at least one node from G_i, i.e., $V(G_{i+1}) \subset V(G_i)$. In order to reduce the number of iterations, the number of nodes to be removed in each iteration should be as large as possible. However, the nodes to be removed in each iteration i cannot be arbitrarily selected. In order to make sure that all SCCs can be computed correctly, for each graph G_i, the newly contracted graph G_{i+1} should satisfy the following three properties.

- (Contractible): The number of nodes in the contracted graph G_{i+1} should be smaller than the number of nodes in G_i, i.e., $V(G_{i+1}) \subset V(G_i)$.
- (SCC-preservable): For any two nodes $u \in V(G_{i+1})$ and $v \in V(G_{i+1})$, u and v are in the same SCC in G_{i+1} iff u and v are in the same SCC in G_i, i.e., $\mathsf{SCC}(u, G_{i+1}) = \mathsf{SCC}(v, G_{i+1}) \Leftrightarrow \mathsf{SCC}(u, G_i) = \mathsf{SCC}(v, G_i)$.
- (Recoverable): If a node $v \in V(G_i)$ is removed when constructing G_{i+1} from G_i, the connectivity of v to other nodes in G_{i+1} can be recovered using only v's neighbors in G_{i+1}. In other words, if v is absent in $V(G_{i+1})$, all v's neighbors should be in $V(G_{i+1})$, i.e., $v \in V(G_i) - V(G_{i+1}) \Rightarrow \mathsf{nbr}(v, G_i) \subseteq V(G_{i+1})$.

In the following, given a graph $G_i(V_i, E_i)$, we introduce how to construct the nodes V_{i+1} of G_{i+1} and edges E_{i+1} of G_{i+1} respectively. We will show that by constructing V_{i+1}, the contractible and recoverable properties are satisfied, and by constructing E_{i+1}, the SCC-preservable property is satisfied.

To construct V_{i+1}: Given $G_i(V_i, E_i)$, we first investigate the properties of V_{i+1}. We start from the recoverable property, i.e., for any $v \in V_i - V_{i+1}$, $\mathsf{nbr}(v, G_i) \subseteq V_{i+1}$. Consider an arbitrary edge $(u, v) \in E_i$, if $v \notin V_{i+1}$, i.e., $v \in V_i - V_{i+1}$, then $u \in V_{i+1}$, since $u \in \mathsf{nbr}(v, G_i)$. Similarity, if $u \notin V_{i+1}$ then $v \in V_{i+1}$. In other words, for any edge $(u, v) \in E_i$, either $v \in V_{i+1}$ or $u \in V_{i+1}$. We have the following lemma.

Lemma 5.1: *G_{i+1} is recoverable if and only if V_{i+1} is a vertex cover of G_i.* □

Algorithm 3 Get-V(G_i)

Input: a directed graph $G_i(V_i, E_i)$ to be contracted.
Output: the nodes of the contracted graph V_{i+1} sorted by node ids.

```
1: $V_{i+1} \leftarrow \emptyset$;
2: $E_{in} \leftarrow$ edges $(u, v) \in E_i$ order by $(\mathsf{id}(v), \mathsf{id}(u))$;
3: $E_{out} \leftarrow$ edges $(u, v) \in E_i$ order by $(\mathsf{id}(u), \mathsf{id}(v))$;
4: $V_d \leftarrow$ nodes $(v, \mathsf{deg}(v, G_i))$ for all $v \in V_i$ order by $\mathsf{id}(v)$ by $E_{in} \bowtie E_{out}$;
5: $E_d \leftarrow$ edges $(u, \mathsf{deg}(u, G_i), v)$ order by $\mathsf{id}(u)$ by $E_{out} \bowtie V_d$;
6: $E_d \leftarrow$ edges $(u, \mathsf{deg}(u, G_i), v)$ order by $\mathsf{id}(v)$ by sorting $E_d$;
7: $E_d \leftarrow$ edges $(u, \mathsf{deg}(u, G_i), v, \mathsf{deg}(v, G_i))$ order by $\mathsf{id}(v)$ by $E_d \bowtie V_d$;
8: for all $(u, \mathsf{deg}(u, G_i), v, \mathsf{deg}(v, G_i)) \in E_d$ do
9:     $V_{i+1} \leftarrow V_{i+1} \cup \{u > v ? u : v\}$
10: sort nodes in $V_{i+1}$ and eliminate duplicate nodes;
11: return $V_{i+1}$;
```

Proof Sketch: The lemma can be derived easily from the above discussion. □

In order to reduce the number of iterations in the Ext-SCC algorithm, the cardinality of V_{i+1} should be as small as possible. This leads to the minimum vertex cover problem which is NP-hard [3]. In the literature, a lot of approximate algorithms have been developed to find the vertex cover of a graph G. When the graph G cannot fit in the main memory, there are semi-external and external algorithms to find an approximate minimum vertex cover for a graph G. For the semi-external algorithm, in [9], James et al. develop a streaming algorithm to find a vertex cover with the approximation ratio 2, by maintaining an in-memory hash table H. However, since $|H| = O(|V(G)|)$ in the worst case, the algorithm cannot be used directly in constructing V_{i+1}. For the external algorithm, in [7], an algorithm is introduced to find a vertex cover with an approximation ratio $\frac{\sqrt{\Delta(G)}}{2} + \frac{3}{2}$, where $\Delta(G)$ is the maximum degree of nodes in G, i.e., $\Delta(G) = \max\{\mathsf{deg}(v) | v \in V(G)\}$. In the algorithm, an operator $>$ is defined among all nodes in G as follows.

Definition 5.1: (Operator $>$): For any $u \in V(G)$ and $v \in V(G)$, $u > v$ iff either of the following two conditions holds. (1) $\mathsf{deg}(u, G) > \mathsf{deg}(v, G)$. (2) $\mathsf{deg}(u, G) = \mathsf{deg}(v, G)$ and $\mathsf{id}(u) > \mathsf{id}(v)$. The $>$ operator specifies a unique total order among all nodes in the graph G. □

Using the $>$ operator, given a graph G, the algorithm in [7] scans all edges of G on disk sequentially. For each edge (u, v) scanned, if $u > v$, then u is added to the vertex cover, otherwise, v is added to the vertex cover.

In this paper, given $G_i(V_i, E_i)$, we adapt the external algorithm in [7] to construct V_{i+1}. We will further reduce the size of V_{i+1} in Section VII. The basic algorithm to compute V_{i+1} is shown in Algorithm 3. After initializing V_{i+1} to be $\emptyset$ (line 1), two edge lists are created on disk, E_{in} and E_{out}, by grouping incoming edges and out-going edges for each node in G_i respectively using external sort (line 2-3). Since all edges are sorted in E_{in} and E_{out}, the degrees of all nodes in G_i can be computed in V_d by joining E_{in} and E_{out} using a single sequential scan of E_{in} and E_{out} (line 4). Next, we create another edge list E_d with degree information of both nodes augmented on each edge, for the ease of comparison of nodes using the $>$ operator. E_d can be created in three steps. Firstly, by joining E_{out} and V_d using a sequential scan, the degree of node u can be augmented into each edge (u, v) in E_d

(line 5). Secondly, E_d is sorted by the non-augmented node of each edge (line 6). Thirdly, by joining E_d and V_d using a sequential scan, the degree of node v can be augmented into each edge (u, v) in E_d (line 7). After creating E_d, we only need to scan edges in E_d sequentially once, and for each edge scanned, add the larger node compared by the $>$ operator into V_{i+1} (line 8-9). Since V_{i+1} may contain duplicate nodes, we sort V_{i+1} by node ids and eliminate duplicate nodes by scanning V_{i+1} once sequentially (line 10).

Lemma 5.2: *The set V_{i+1} computed in Algorithm 3 is recoverable and contractible.* □

Proof Sketch: Since Algorithm 3 computes a vertex cover V_{i+1} of G_i, from Lemma 5.1, V_{i+1} computed in Algorithm 3 is recoverable. Next, we prove that V_{i+1} is contractible. We only need to prove that there exists a node v such that $v \in V_i$ and $v \notin V_{i+1}$. By Definition 5.1, the operator $>$ specifies a unique total order among all nodes in G_i. Let v be the smallest node in the total order defined by the operator $>$, v cannot be added into V_{i+1}, because there does not exits an edge (u, v) or an edge (v, u) with $u > v$. Thus the lemma holds. □

Theorem 5.1: *The I/O complexity of Algorithm 3 is $O(sort(|E_i|) + sort(|V_i|))$.* □

Proof Sketch: Omitted due to lack of space. □

To construct E_{i+1}: Given a graph $G_i(V_i, E_i)$ and the node set V_{i+1} of the contracted graph G_{i+1}, we construct the edges E_{i+1} of the contracted graph G_{i+1}. Since V_{i+1} is constructed in a way such that the contractible and recoverable properties are satisfied, E_{i+1} needs to be constructed that maintains the SCC-preservable property. In other words, after constructing E_{i+1}, for any two nodes $u \in V_{i+1}$ and $v \in V_{i+1}$, if u and v are in the same SCC in G_i, then u and v should be in the same SCC in G_{i+1}, and vice versa. Note that by removing a node v from G_i, for any two nodes u and w in V_{i+1}, if u can only reach w through v in G_i, then the connectivity of u and w is destroyed after the removal of node v. In order to maintain such connectivity in G_{i+1}, new edges need to be added after the removal of v. Suppose u can reach w through a path $(u \cdots v_{in}, v, v_{out}, \cdots w)$ in G_i, where $v_{in} \in \mathsf{nbr}_{in}(v, G_i)$ and $v_{out} \in \mathsf{nbr}_{out}(v, G_i)$, after the removal of v, we can add a new edge (v_{in}, v_{out}) in E_{i+1}, such that u can still reach w through a path $(u \cdots v_{in}, v_{out}, \cdots w)$ in G_{i+1}. In order to do this, we need to make sure that both v_{in} and v_{out} are in G_{i+1}. It is true because V_{i+1} maintains the recoverable property such that for each removed node v, i.e., $v \in V_i - V_{i+1}$, all its neighbors in G_i are in V_{i+1}, i.e., $\mathsf{nbr}(v, G_i) \subseteq V_{i+1}$.

An algorithm to construct E_{i+1} can be designed as follows. For each node $v \in V_i - V_{i+1}$, and each pair of nodes $v_{in} \in \mathsf{nbr}_{in}(v, G_i)$ and $v_{out} \in \mathsf{nbr}_{out}(v, G_i)$, remove the edge (v_{in}, v) and (v, v_{out}), and add a new edge (v_{in}, v_{out}) in E_{i+1}. By doing this, for any pair of nodes $u \in V_{i+1}$ and $w \in V_{i+1}$, if u can reach w in G_i, u can still reach w in G_{i+1}. The edges are constructed to ensure that no connectivity of node pairs will be destroyed. We will prove it that the construction will introduce no new connectivity information among all nodes in V_{i+1}.

Algorithm 4 Get-E(G_i, V_{i+1})

Input: a directed graph $G_i(V_i, E_i)$ to be contracted,
the nodes of the contracted graph V_{i+1} sorted by node ids.
Output: the edges of the contracted graph E_{i+1}.

```
 1: $E_{in} \leftarrow$ edges $(u, v) \in E_i$ order by $(\mathsf{id}(v), \mathsf{id}(u))$;
 2: $E_{out} \leftarrow$ edges $(u, v) \in E_i$ order by $(\mathsf{id}(u), \mathsf{id}(v))$;
 3: $E_{del} \leftarrow$ edges $(u, v) \in E_i$ for $v \in V_i - V_{i+1}$ order by $\mathsf{id}(v)$ by $V_{i+1} \bowtie E_{in}$;
 4: $E_{del} \leftarrow$ edges $(u, v, \mathsf{nbr}_{out}(v, G_i))$ for $v \in V_i - V_{i+1}$ ordered by $\mathsf{id}(v)$ by
    $E_{del} \bowtie E_{out}$;
 5: $E_{add} \leftarrow \emptyset$;
 6: for all edge $(u, v) \in E_{del}$ do
 7:     for all $w \in \mathsf{nbr}_{out}(v, G_i)$ by sequential scan of $E_{del}$ do
 8:         $E_{add} \leftarrow E_{add} \cup (u, w)$;
 9: $E_{pre} \leftarrow$ edges $(u, v) \in E_i$ for $u \in V_{i+1}$ order by $\mathsf{id}(u)$ by $V_{i+1} \bowtie E_{out}$;
10: $E_{pre} \leftarrow$ edges $(u, v) \in E_i$ for $u \in V_{i+1}$ order by $\mathsf{id}(v)$ by sorting $E_{pre}$;
11: $E_{pre} \leftarrow$ edges $(u, v) \in E_i$ for $u \in V_{i+1}$ and $v \in V_{i+1}$ order by $\mathsf{id}(v)$ by
    $V_{i+1} \bowtie E_{pre}$;
12: $E_{i+1} \leftarrow E_{pre} \cup E_{add}$;
13: return $E_{i+1}$;
```

Algorithm 4 shows how to construct E_{i+1} externally, given $G_i(V_i, E_i)$ and V_{i+1}. As discussed above, E_{i+1} consists of two parts, namely, the preserved edges in G_i with both ends in V_{i+1}, denoted E_{pre}, and the newly added edges by removing nodes from G_i, denoted E_{add}. Let E_{in} and E_{out} be the edges of G_i by grouping incoming and out-going edges for each node in G_i respectively, which are the same as those used in Algorithm 3 (line 1-2), Algorithm 4 constructs E_{add} in line 3-8 and constructs E_{pre} in line 9-11, and union E_{add} and E_{pre} to construct E_{i+1} (line 12-13).

In order to construct E_{add}, the algorithm first identifies the set of incoming edges to be removed, denoted E_{del}, by joining V_{i+1} and E_{in} using a single sequential scan of V_{i+1} and E_{in} on disk (line 3). When scanning V_{i+1} and E_{in}, for each edge $(u, v) \in E_{in}$, if $v \notin V_{i+1}$, then (u, v) is added to E_{del}. After constructing the removed incoming edges, we augment the out-neighbors of v into each incoming edge $(u, v) \in E_{del}$. This can be done using a single sequential scan of E_{del} and E_{out} (line 4). Line 5-8 construct E_{add} using a single sequential scan of all edges in E_{del}. In E_{del}, for each node v that is removed from G_i, each of its in-neighbors u in G_i is stored as an edge (u, v), and its out-neighbors $\mathsf{nbr}_{out}(v, G_i)$ is also augmented in the edge (u, v) with form $(u, v, \mathsf{nbr}_{out}(v, G_i))$. When accessing each removed incoming edge (u, v) of v (line 6), the removed out-going edge (v, w) of v can be accessed in the same sequential scan of E_{del} (line 7), and a new edge (u, w) is added into E_{add} (line 8).

The preserved edges E_{pre} can be constructed in three steps. Firstly, by joining V_{i+1} and E_{out} using a sequential scan, all edges (u, v) with $u \in V_{i+1}$ can be preserved in E_{pre} (line 9). Secondly, we sort E_{pre} such that all edges $(u, v) \in E_{pre}$ are sorted by $\mathsf{id}(v)$ (line 10). Thirdly, by joining V_{i+1} and E_{pre} using a sequential scan, all edges (u, v) with $u \in V_{i+1}$ and $v \in V_{i+1}$ can be preserved in E_{pre} (line 11).

Lemma 5.3: *The edge set E_{i+1} constructed by Algorithm 4 is* SCC*-preservable.* □

Proof Sketch: We only need to prove for any nodes $u \in V_{i+1}$ and $w \in V_{i+1}$, $u \rightarrow w$ in $G_i \Leftrightarrow u \rightarrow w$ in G_{i+1}.

To prove $\Rightarrow$, for any path p from u to w in G_i, we can find a path from u to w in G_{i+1} as follows. For each node v on

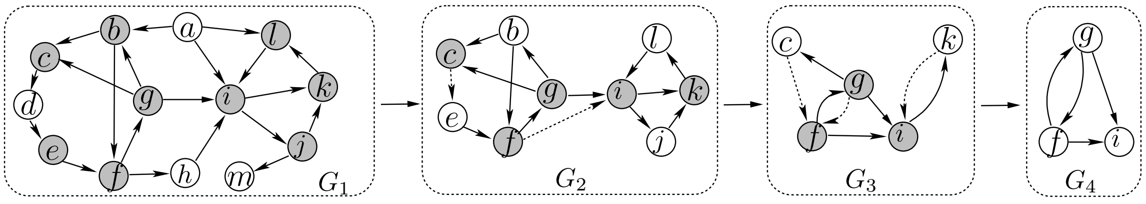


Fig. 4. Graph Contraction Example

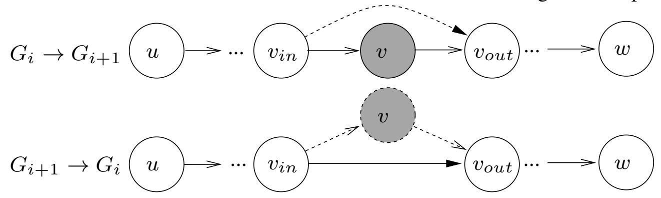


Fig. 3. Paths in G_i and G_{i+1}

path p with $v \notin V_{i+1}$, let v_{in} and v_{out} be the predecessor and successor of v on path p, remove edge (v_{in}, v) and (v, v_{out}) from path p and add a new edge (v_{in}, v_{out}) on p. From the construction of E_{i+1}, we have $(v_{in}, v_{out}) \in E_{i+1}$. The new path p is a path on G_{i+1}. The construction process is illustrated in the upper part of Fig. 3.

To prove $\Leftarrow$, for any path p from u to w in G_{i+1}, we can find a path from u to w in G_i as follows. For each edge (v_{in}, v_{out}) on path p, if (v_{in}, v_{out}) is a newly added edge, from the construction of E_{i+1}, there exists a node $v \in V_i$ such that $(v_{in}, v) \in E_i$ and $(v, v_{out}) \in E_i$. We remove (v_{in}, v_{out}) from p and add two new edges (v_{in}, v) and (v, v_{out}) on p. The new path p is a path on G_i. The construction process is illustrated in the lower part of Fig. 3. □

Theorem 5.2: *The I/O complexity of Algorithm 4 is* $O(sort(|E_i|) + scan(V_{i+1}) + scan(|E_{i+1}|))$. □

Proof Sketch: Omitted due to lack of space. □

Bounding Edge Size $|E_{i+1}|$: From the above discussion, when a node v is removed from G_i when constructing G_{i+1}, $\deg_{in}(v, G_i) \times \deg_{out}(v, G_i)$ new edges are added into E_{i+1}. When $\mathsf{deg}(v, G_i)$ is large, the number of newly added edges can be very large. However, from the construction of V_{i+1}, a node v is removed only if for all $u \in \mathsf{nbr}(v, G_i)$, $\mathsf{deg}(u, G_i) \geq \mathsf{deg}(v, G_i)$. Thus, the degree of any removed node cannot be too large. The following two theorems give an upper bound of the degree for any removed node v, and an upper bound of the number of new edges in E_{i+1} respectively.

Theorem 5.3: $\forall v \in V_i - V_{i+1}$, $\mathsf{deg}(v, G_i) \leq \sqrt{2 \times |E_i|}$. □

Proof Sketch: For any $v \in V_i - V_{i+1}$ and $u \in \mathsf{nbr}(v, G_i)$, we have $\mathsf{deg}(u, G_i) \geq \mathsf{deg}(v, G_i)$. The total degrees of all nodes in $\mathsf{nbr}(v, G_i)$ is $\sum_{u \in \mathsf{nbr}(v, G_i)} \mathsf{deg}(u, G_i) \leq 2 \times |E_i|$, since each edge in E_i is counted at most twice in the summation. We also have $\sum_{u \in \mathsf{nbr}(v, G_i)} \mathsf{deg}(u, G_i) \geq \sum_{u \in \mathsf{nbr}(v, G_i)} \mathsf{deg}(v, G_i) = \mathsf{deg}(v, G_i)^2$. Thus $\mathsf{deg}(v, G_i)^2 \leq 2 \times |E_i|$. Theorem 5.3 holds. □

Theorem 5.4: *The number of new edges in E_{i+1} is bounded by $\alpha_i \times |E_i|$, where α_i is the arboricity [11] of graph G_i.* □

Proof Sketch: By the construction of E_{i+1}, the number of new edges in E_{i+1} is

$$\begin{aligned}
&\Sigma_{v \in V_i - V_{i+1}} \mathsf{deg}_{in}(v, G_i) \times \mathsf{deg}_{out}(v, G_i) \\
&\leq \Sigma_{v \in V_i - V_{i+1}} \mathsf{deg}_{in}(v, G_i) \times \mathsf{deg}(v, G_i) \\
&= \Sigma_{v \in V_i - V_{i+1}} \Sigma_{u \in \mathsf{nbr}_{in}(v, G_i)} \mathsf{deg}(v, G_i) \\
&= \Sigma_{v \in V_i - V_{i+1}} \Sigma_{u \in \mathsf{nbr}_{in}(v, G_i)} \min\{\mathsf{deg}(v, G_i), \mathsf{deg}(u, G_i)\} \\
&\leq \Sigma_{v \in V_i} \Sigma_{u \in \mathsf{nbr}_{in}(v, G_i)} \min\{\mathsf{deg}(v, G_i), \mathsf{deg}(u, G_i)\} \\
&= \Sigma_{(u,v) \in E_i} \min\{\mathsf{deg}(v, G_i), \mathsf{deg}(u, G_i)\}.
\end{aligned} \tag{1}$$

According to [11], $\sum_{(u,v) \in E_i} \min\{\mathsf{deg}(v, G_i), \mathsf{deg}(u, G_i)\} \leq \alpha_i \times |E_i|$ for any graph G_i. □

As proved in [11], $\alpha_i \leq \min\{\lceil \sqrt{|E_i|} \rceil, \mathsf{deg}_{max}\}$ for any graph G_i, where deg_{max} is the maximum degree for all nodes in G_i. In practice, α_i is usually small. For example, $\alpha_i = O(1)$ if G_i is a planar graph. Note that in Theorem 5.4, the upper bound $\alpha_i \times |E_i|$ is very loose, since it amplifies $\Sigma_{v \in V_i - V_{i+1}} \Sigma_{u \in \mathsf{nbr}_{in}(v, G_i)} \min\{\mathsf{deg}(v, G_i), \mathsf{deg}(u, G_i)\}$ to $\Sigma_{v \in V_i} \Sigma_{u \in \mathsf{nbr}_{in}(v, G_i)} \min\{\mathsf{deg}(v, G_i), \mathsf{deg}(u, G_i)\}$ in the second $\leq$ of Eq. (1), and as analyzed in Theorem 5.3, the set $V_i - V_{i+1}$ only contains nodes with small degrees in G_i. In section VII, we propose techniques to further reduce $|E_{i+1}|$, and in our experiments, it is even possible that $|E_{i+1}| < |E_i|$ for a certain graph G_i.

Example 5.1: Fig. 4 shows the graph contraction phase for G in Fig. 1. The grey nodes in each G_i are the set of nodes preserved in G_{i+1}, and the dashed edges in each G_i are the newly added edges when constructing G_i from G_{i-1}. In G_1 ($= G$), node b is preserved in V_2, since there is an edge (a, b) with $\mathsf{deg}(b, G_1) > \mathsf{deg}(a, G_1)$, thus $b > a$ by operator $>$ defined in Definition 5.1. $d \in V_1 - V_2$ because for its two neighbors c and e, $c > d$ and $e > d$. After removing d, a new edge (c, e) is added in G_2 since $c \in \mathsf{nbr}_{in}(d, G_1)$ and $e \in \mathsf{nbr}_{out}(d, G_1)$. Using such a way, the newly constructed G_2 has 9 nodes and 14 edges by removing parallel edges and self circles. In a similar way, G_3 can be constructed from G_2 with $|V_3| = 5$ and $|E_3| = 8$. G_4 can be constructed from G_3 with $|V_4| = 3$ and $|E_4| = 4$. Suppose the main memory can only keep three nodes. The graph contraction phase stops after G_4 is constructed, since G_4 can be processed using Semi-SCC. □

VI. Graph Expansion

The graph contraction phase stops when all nodes of the contracted graph G_{i+1} can fit in memory, such that the SCCs of G_{i+1} can be computed using the semi-external algorithm Semi-SCC. The graph expansion phase computes all SCCs of G_i using the information computed in G_{i+1} with decreasing i iteratively. Given graph $G_i(V_i, E_i)$ and $G_{i+1}(V_{i+1}, E_{i+1})$, suppose all SCCs of nodes in V_{i+1} are computed, denoted as SCC_{i+1}, we discuss computing SCC_i, the set of all $\mathsf{SCC}s$ of nodes in V_i. According to the SCC-preservable property of G_{i+1} in Lemma 5.3, we only need to compute $\mathsf{SCC}(v, G_i)$ for each $v \in V_i - V_{i+1}$, since $\mathsf{SCC}(v, G_i) = \mathsf{SCC}(v, G_{i+1})$ for each $v \in V_{i+1}$ according to Lemma 5.3. We start with the following lemma.

Lemma 6.1: *Given any two nodes $u \in V_{i+1}$ and $w \in V_{i+1}$ (possibly $u = w$) with $\mathsf{SCC}(u, G_i) = \mathsf{SCC}(w, G_i)$ in graph G_i, for any node $v \in V_i - V_{i+1}$, $\mathsf{SCC}(u, G_i) = \mathsf{SCC}(w, G_i) = \mathsf{SCC}(v, G_i) \Leftrightarrow u \to v$ and $v \to w$ in G_i.* $\square$

Proof Sketch: $\Rightarrow$ is trivial. We prove $\Leftarrow$. Because $\mathsf{SCC}(u, G_i) = \mathsf{SCC}(w, G_i)$, we have $u \leftrightarrow w$ in G_i. Since $u \to v$ and $v \to w$ in G_i, we can derive that $v \to w \to u \to v$ in G_i. Thus $\mathsf{SCC}(u, G_i) = \mathsf{SCC}(w, G_i) = \mathsf{SCC}(v, G_i)$. $\square$

Lemma 6.1 suggests a way to compute $\mathsf{SCC}(v, G_i)$ for a node $v \in V_i - V_{i+1}$: to find two nodes u and w (possibly the same) in V_{i+1} with $\mathsf{SCC}(u, G_i) = \mathsf{SCC}(w, G_i)$ and $u \to v$ and $v \to w$ in G_i. However, u and w are not easy to find, and $u \to v$ and $v \to w$ are not easy to compute. In the following, we show that it is enough to find u from $\mathsf{nbr}_{in}(v, G_i)$ and find w from $\mathsf{nbr}_{out}(v, G_i)$. We first investigate some properties of $\mathsf{SCC}s$ in $\mathsf{nbr}_{in}(v, G_i)$ and $\mathsf{nbr}_{out}(v, G_i)$ in Lemma 6.2 and Lemma 6.3.

Lemma 6.2: *For any node $v \in V_i - V_{i+1}$, if $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i) \cap \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i) \neq \emptyset$, then $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i) \cap \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i) = \{\mathsf{SCC}(v, G_i)\}$.* $\square$

Proof Sketch: Suppose there exists $v_{in} \in \mathsf{nbr}_{in}(v, G_i)$ and $v_{out} \in \mathsf{nbr}_{out}(v, G_i)$ with $\mathsf{SCC}(v_{in}, G_i) = \mathsf{SCC}(v_{out}, G_i)$, since $v_{in} \to v$ and $v \to v_{out}$, from Lemma 6.1, we have $\mathsf{SCC}(v_{in}, G_i) = \mathsf{SCC}(v_{out}, G_i) = \mathsf{SCC}(v, G_i)$ in graph G_i. As a result, $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i) \cap \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i) = \{\mathsf{SCC}(v, G_i)\}$ $\square$

Lemma 6.3: *For any node $v \in V_i - V_{i+1}$, if there exists another node $u \in V_i$ with $\mathsf{SCC}(v, G_i) = \mathsf{SCC}(u, G_i)$, then $\mathsf{SCC}(v, G_i) \in \mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i)$ and $\mathsf{SCC}(v, G_i) \in \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i)$.* $\square$

Proof Sketch: Since $u \neq v$ and $\mathsf{SCC}(v, G_i) = \mathsf{SCC}(u, G_i)$, there exists v_{in} and v_{out}, such that $u \to v$ through a path $(u, \cdots, v_{in}, v)$ with $v_{in} \in \mathsf{nbr}_{in}(v, G_i)$, and $v \to u$ through a path $(v, v_{out}, \cdots, u)$ with $v_{out} \in \mathsf{nbr}_{out}(v, G_i)$. Thus $\mathsf{SCC}(v, G_i) = \mathsf{SCC}(v_{in}, G_i) = \mathsf{SCC}(v_{out}, G_i)$. As a result, $\mathsf{SCC}(v, G_i) \in \mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i)$ and $\mathsf{SCC}(v, G_i) \in \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i)$. $\square$

According to the recoverable property of G_{i+1} in Lemma 5.2, for any node $v \in V_i - V_{i+1}$, $\mathsf{nbr}_{in}(v, G_i) \subseteq V_{i+1}$ and $\mathsf{nbr}_{out}(v, G_i) \subseteq V_{i+1}$. As a result, both $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i)$ and $\mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i)$ are computed in G_{i+1}. The following lemma shows that $\mathsf{SCC}(v)$ can be computed using $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i)$ and $\mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i)$ only.

Lemma 6.4: *For any node $v \in V_i - V_{i+1}$, $\mathsf{SCC}(v, G_i)$ can be computed using $\mathsf{nbr}_{in}(v, G_i)$ and $\mathsf{nbr}_{out}(v, G_i)$ only.* $\square$

Proof Sketch: There are two situations: 1) If $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i) \cap \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i) \neq \emptyset$, according to Lemma 6.2, $\mathsf{SCC}(v, G_i)$ can be calculated using $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i) \cap \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i)$. 2) If $\mathsf{SCC}(\mathsf{nbr}_{in}(v, G_i), G_i) \cap \mathsf{SCC}(\mathsf{nbr}_{out}(v, G_i), G_i) = \emptyset$, according to Lemma 6.3, there does not exist another node $u \in G_i$ with $\mathsf{SCC}(v, G_i) = \mathsf{SCC}(u, G_i)$. As a result, v is an SCC with a single node in G_i. $\square$

Algorithm 5 $\mathsf{Expansion}(G_i, G_{i+1}, \mathsf{SCC}_{i+1})$

```
Input:  graph $G_i(V_i, E_i)$ and its extracted graph $G_{i+1}(V_{i+1}, E_{i+1})$,
        the SCCs of all nodes in $G_{i+1}$ sorted by node ids $\mathsf{SCC}_{i+1}$.
Output: the SCCs of all nodes in $G_i$ sorted by node ids $\mathsf{SCC}_i$.

1:  $\overline{E}_i \leftarrow$ reverse all edges in $E_i$;
2:  $E'_{in} \leftarrow \mathsf{augment}(E_i)$;
3:  $E'_{out} \leftarrow \mathsf{augment}(\overline{E}_i)$;
4:  $\mathsf{SCC}_{del} \leftarrow$ nodes $(v, \mathsf{SCC}(v, G_i))$ for $v \in V_i - V_{i+1}$ by $E'_{in} \bowtie E'_{out}$;
5:  $\mathsf{SCC}_i \leftarrow \mathsf{SCC}_{i+1} \cup \mathsf{SCC}_{del}$;
6:  sort $\mathsf{SCC}_i$ by node ids;
7:  return $\mathsf{SCC}_i$;

8:  Procedure $\mathsf{augment}(E)$
9:  $E \leftarrow$ edges $(u, v) \in E$ order by $(\mathsf{id}(v), \mathsf{id}(u))$;
10: $E' \leftarrow$ edges $(u, v) \in E$ for $v \in V_i - V_{i+1}$ order by $\mathsf{id}(v)$ by $V_{i+1} \bowtie E$;
11: $E' \leftarrow$ edges $(u, v) \in E$ for $v \in V_i - V_{i+1}$ order by $\mathsf{id}(u)$ by sorting $E'$;
12: $E' \leftarrow$ edges $(u, v, \mathsf{SCC}(u, G_{i+1}))$ for $v \in V_i - V_{i+1}$ order by $\mathsf{id}(u)$ by
    $E' \bowtie \mathsf{SCC}_{i+1}$;
13: $E' \leftarrow$ edges $(u, v, \mathsf{SCC}(u, G_{i+1}))$ for $v \in V_i - V_{i+1}$ order by $(\mathsf{id}(v), \mathsf{SCC}(u,$
    $G_{i+1}), \mathsf{id}(u))$ by sorting $E'$;
14: return $E'$
```

Given $G_i(V_i, E_i)$, $G_{i+1}(V_{i+1}, E_{i+1})$, and SCC_{i+1}, our external algorithm to compute SCC_i is shown in Algorithm 5. In order to join $\mathsf{nbr}_{in}(v, G_i)$ and $\mathsf{nbr}_{out}(v, G_i)$ for each removed node v, the algorithm augments the SCC information into the in-neighbors of the removed nodes using $\mathsf{augment}(E_i)$ (line 2), and augments the SCC information into the out-neighbors of the removed nodes using $\mathsf{augment}(\overline{E}_i)$ (line 3), where $\overline{E}_i$ is generated by reversing every edge in E_i. Note that in-neighbors in $\overline{E}_i$ become out-neighbors in E_i. The procedure to augment the SCC into the in-neighbors of $E_i/\overline{E}_i$ of each removed node is shown in line 8-14. In the procedure, a new edge set E' that keeps the incoming edges of only the removed nodes $V_i - V_{i+1}$ is created by joining E and V_{i+1} (line 10). In line 11, edges (u, v) in E' are sorted by $\mathsf{id}(u)$ in order to augment $\mathsf{SCC}(u)$. In line 12, $\mathsf{SCC}(u)$ is augmented in each edge (u, v) in E' by sequential scan of E' and SCC_{i+1}. In line 13, edges $(u, v, \mathsf{SCC}(u))$ in E' are sorted by $(\mathsf{id}(v), \mathsf{SCC}(u), \mathsf{id}(u))$ to put the in-neighbors of each node in $V_i - V_{i+1}$ together in order to compute $\mathsf{nbr}_{in}(v, G_i) \cap \mathsf{nbr}_{out}(v, G_i)$ efficiently. After augmenting the SCCs into the in-neighbors and out-neighbors of the removed nodes, in line 4, the $\mathsf{SCC}s$ of all nodes $v \in V_i - V_{i+1}$ can be computed using $\mathsf{nbr}_{in}(v, G_i) \cap \mathsf{nbr}_{out}(v, G_i)$ by a sequential scan of E'_{in} and E'_{out}. In line 5, by combining SCCs in $V_i - V_{i+1}$ and SCCs in V_{i+1} computed in G_{i+1}, the SCCs for all nodes in G_i can be computed as SCC_i. Finally, all nodes in SCC_i are sorted (line 6) and SCC_i is returned (line 7).

Theorem 6.1: *The I/O complexity of Algorithm 5 is $O(scan(|V_{i+1}|) + sort(|E_i|) + sort(|V_i|))$.* $\square$

Proof Sketch: Omitted due to lack of space. $\square$

Theorem 6.2: *Algorithm 5 computes all $\mathsf{SCC}s$ of G_i.* $\square$

Proof Sketch: For node $v \in V_{i+1}$, $\mathsf{SCC}(v, G_i) = \mathsf{SCC}(v, G_{i+1})$ according to the SCC-preservable property in Lemma 5.3. For node $v \in V_i - V_{i+1}$, $\mathsf{SCC}(v, G_i)$ is correctly

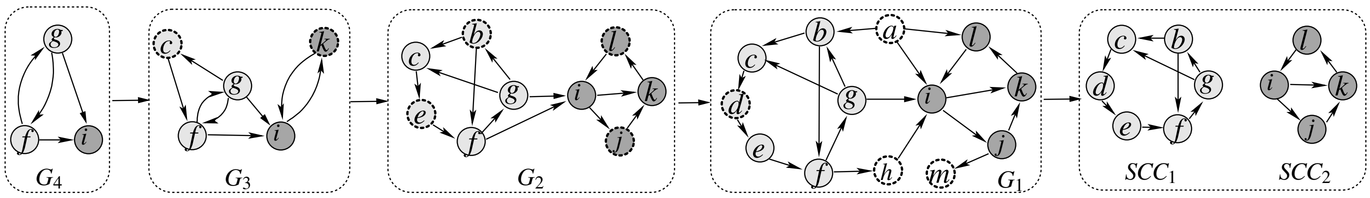


Fig. 5. Graph Expansion Example

computed according to Lemma 6.4. □

Example 6.1: Fig. 5 shows the process of the graph expansion phase, to expand the graphs in order of G_4, G_3, G_2, and G_1 generated in the graph contraction phase in Fig. 4. The dashed circles in each graph G_i are the removed nodes when constructing G_{i+1} from G_i. By applying the Semi-SCC algorithm on G_4, two SCCs are computed, namely, SCC_1 denoted by light gray nodes, and SCC_2 denoted by dark gray nodes. In G_3, for node $c \in V_3 - V_4$, we have $g \in \mathsf{nbr}_{in}(c, G_3)$ and $f \in \mathsf{nbr}_{out}(c, G_3)$ with $\mathsf{SCC}(g, G_3) = \mathsf{SCC}(f, G_3) = SCC_1$. Thus $\mathsf{SCC}(c, G_3) = SCC_1$. Similarly, the SCCs of node k in G_3, and nodes b, e, l, j in G_2 can be computed. In G_1, for node h, $\mathsf{SCC}(\mathsf{nbr}_{in}(h, G_1), G_1) = \{SCC_1\}$ and $\mathsf{SCC}(\mathsf{nbr}_{out}(h, G_1), G_1) = \{SCC_2\}$, $\mathsf{SCC}(\mathsf{nbr}_{in}(h, G_1), G_1) \cap \mathsf{SCC}(\mathsf{nbr}_{out}(h, G_1), G_1) = \emptyset$. Thus node h in G_1 is an SCC with a single node. Finally, there are two SCCs SCC_1 and SCC_2 with 6 and 4 nodes respectively. □

VII. I/O Cost Minimization

In this section, we show how to optimize our contraction-expansion based Ext-SCC approach by further reducing the I/O cost. The I/O cost can be reduced in two ways: (1) to reduce the number of graphs constructed in the graph contraction phase, and (2) to reduce the number of nodes and edges for each graph G_i constructed in the i-th iteration. According to the stop condition of graph contraction, reducing the number of graphs is equivalent to reducing the number of nodes $|V_i|$ in each graph G_i. Thus, the key point to reduce the I/O cost of the Ext-SCC algorithm is to reduce the number of nodes $|V_i|$ and number of edges $|E_i|$ in each graph G_i. In the following, we introduce techniques to reduce the nodes and edges when constructing G_{i+1} from G_i.

Node Reduction: Given graph G_i, when constructing G_{i+1}, suppose V_{i+1} has already been computed using Algorithm 3, the following two types of nodes can be removed from V_{i+1}.

- (Type-1): For a node $v \in V_{i+1}$, if there does not exit another node $u \in V_i$ such that $\mathsf{SCC}(v, G_i) = \mathsf{SCC}(u, G_i)$, v can be removed from V_{i+1}.
- (Type-2): For a node $v \in V_{i+1}$, if $\mathsf{nbr}(v, G_i) \subseteq V_{i+1}$, v can be removed from V_{i+1}.

A Type-1 node v can be removed because v itself is an SCC with a single node in G_i, and there is no need to include v in later iterations as v cannot be combined with other nodes to form new SCCs in later iterations. A Type-2 node v can be removed because when $\mathsf{nbr}(v, G_i) \subseteq V_{i+1}$, $V_{i+1} - \{v\}$ is still a vertex cover of G_i. According to the node selection condition in Lemma 5.1, $V_{i+1} - \{v\}$ is a valid node set of G_{i+1}. Type-2 nodes are order sensitive, i.e., if node u and v are Type-2 nodes, after u is removed, v may not be a Type-2 node. Note that our aim is to reduce nodes in V_{i+1} without introducing new I/O cost, thus we only reduce those Type-1 and Type-2 nodes that are easy to be identified. The following lemma can be used to reduce Type-1 nodes.

Lemma 7.1: *Any node v with $\mathsf{deg}_{in}(v, G_i) = 0$ or $\mathsf{deg}_{out}(v, G_i) = 0$ is a Type-1 node.* □

Proof Sketch: Omitted due to lack of space. □

To remove Type-1 node v from V_{i+1} with $\mathsf{deg}_{in}(v, G_i) = 0$ or $\mathsf{deg}_{out}(v, G_i) = 0$, when generating V_d in line 4 of Algorithm 3, by joining E_{in} and E_{out}, we only keep the nodes with both $\mathsf{deg}_{in}(v, G_i) > 0$ and $\mathsf{deg}_{out}(v, G_i) > 0$ in V_d. Since V_{i+1} is generated from E_d which is computed using V_d, all nodes v with $\mathsf{deg}_{in}(v, G_i) = 0$ or $\mathsf{deg}_{out}(v, G_i) = 0$ will be removed in V_{i+1} in Algorithm 3. Such an operation does not generate any extra I/O cost in Algorithm 3.

In order to reduce Type-2 nodes, when scanning all edges in E_d in line 8-9 of Algorithm 3, for each edge (u, v) scanned, suppose $v > u$, before adding v into V_{i+1}, we check whether u has been added into V_{i+1}. If so, edge (u, v) has been covered by node u and there is no need to add v into V_{i+1} to cover the edge (u, v) again. The situation for $u > v$ can be handled similarly. In such a way, Type-2 nodes can be effectively reduced in V_{i+1}. However, such a solution needs to check whether $u \in V_{i+1}$ using a dictionary T which many not reside entirely in the main memory. Suppose T can only hold s nodes in memory, since a node with higher degree are less possible to be removed from V_{i+1}, when adding nodes into T, we only maintain the top s smallest nodes using operator $>$ in T, to make sure that T can reside entirely in the main memory. By doing so, we can reduce the number of Type-2 nodes in V_{i+1} without generating any extra I/O cost in Algorithm 3.

Edge Reduction: Given $G_i(V_i, E_i)$, we introduce two ways to reduce the number of edges when generating G_{i+1} in the i-th graph contraction phase without increasing the I/O complexity. Although $|V_i| < |V_{i-1}|$, it is possible that $|E_i| > |E_{i-1}|$. We develop two methods to reduce the edge size in order to reduce the intermediate results. We will discuss the efficiency in our performance studies.

Firstly, for parallel edges with the same form (u, v), only one of them needs to be kept in E_{i+1}. Such edges can be reduced in a lazy way, when generating E_{in} in the next iteration of Get-E (line 2 of Algorithm 3). A sequential scan of E_{in} needs to be added after line 2 of Algorithm 3 to eliminate parallel edges in E_{in}. In addition, it is straightforward that each edge (u, w) with $u = w$ can be removed from E_{i+1}. This can be done in line 8 of Algorithm 4, by checking whether

Parameter	Range	Default
Size of $\lvert V\rvert$	25M,50M,100M,150M,200M	100M
Average Degree D	2,3,4,5,6	4
Memory Size M	200M,300M,400M,500M,600M	400M
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 I
RANGE AND DEFAULT VALUE FOR PARAMETERS

$u = w$ before adding (u, w) to E_{add}.

Secondly, using operator $>$, according to Theorem 5.3, nodes with small degrees are removed when constructing V_{i+1}, and for each removed node $v \in V_i - V_{i+1}$, $\text{deg}_{in}(v, G_i) \times \text{deg}_{out}(v, G_i)$ new edges are added into E_{i+1}. By considering $\text{deg}_{in}(v, G_i) \times \text{deg}_{out}(v, G_i)$ in the operator $>$, $|E_{i+1}|$ can be further reduced. We redefine the operator $>$ as follows.

Definition 7.1: (Operator $>$): For any $u \in V(G)$ and $v \in V(G)$, $u > v$ iff one of the following three conditions holds. (1) $\text{deg}(u, G) > \text{deg}(v, G)$. (2) $\text{deg}(u, G) = \text{deg}(v, G)$ and $\text{deg}_{in}(u, G) \times \text{deg}_{out}(u, G) > \text{deg}_{in}(v, G) \times \text{deg}_{out}(v, G)$. (3) $\text{deg}(u, G) = \text{deg}(v, G)$ and $\text{deg}_{in}(u, G) \times \text{deg}_{out}(u, G) = \text{deg}_{in}(v, G) \times \text{deg}_{out}(v, G)$ and $\text{id}(u) > \text{id}(v)$. The $>$ operator specifies a unique total order among all nodes in the graph G. □

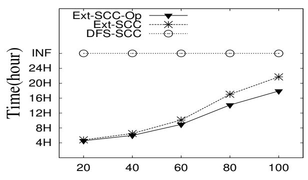
In Algorithm 3, in order to make use of the new $>$ operator in line 9, when generating V_d in line 4, both $\text{deg}(v, G_i)$ and $\text{deg}_{in}(v, G_i) \times \text{deg}_{out}(v, G_i)$ need to be computed in V_d in line 4, and augmented in all nodes in E_d in line 5-7.

VIII. PERFORMANCE STUDIES

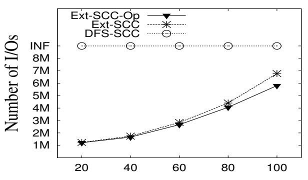
In this section, we conduct experimental studies by comparing four external algorithms for SCC computation, namely, the external contraction based EM-SCC [13], the external *DFS* based DFS-SCC [8], our external contraction-expansion based algorithm Ext-SCC (Algorithm 2), and our algorithm Ext-SCC-Op by applying the optimization techniques introduced in Section VII in Ext-SCC. All the algorithms are implemented using Visual C++ 2005 and tested on a PC with Intel Core2 Quar 2.66GHz CPU and 3.5GB memory running Windows XP. The disk block size is $256KB$. The default memory size is $400M$. For the semi-external algorithm Semi-SCC used in Ext-SCC, we apply the algorithm 1PB-SCC introduced in [26], which is currently the most I/O efficient semi-external algorithm for SCC computation. The 1PB-SCC algorithm needs to hold $2 \times |V(G)|$ plus one disk block in the main memory, that is $M = 4 \times (2 \times |V(G)|) + 256K$ where 4 is the number of bytes to keep a node in memory. We set the max time cost to be 24 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 results of EM-SCC since it cannot stop in all cases.

Datasets: In our experiments, we use a real large web graph and several synthetic datasets. The real web graph is WEBSPAM-UK2007[4], which consists of 105,896,555 web-

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

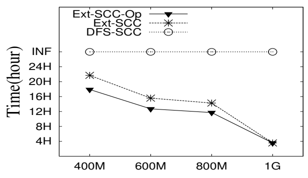


(a) Time (Vary Graph Size)

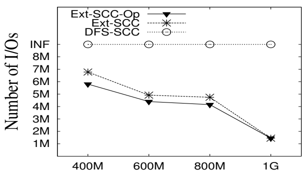


(b) I/Os (Vary Graph Size)

Fig. 6. WEBSPAM-UK2007: Varying Graph Size (Percent)



(a) Time (Vary Memory)



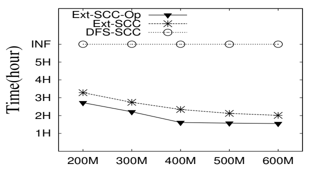
(b) I/Os (Vary Memory)

Fig. 7. WEBSPAM-UK2007: Varying Memory Size

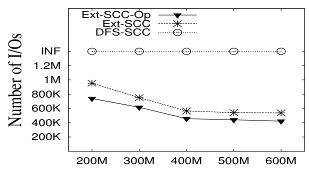
pages in 114,529 hosts in the .UK domain. The graph contains 105,895,908 nodes and 3,738,733,568 edges, with the average degree 35 per node. For synthetic data, we generate 3 different kinds of datasets, denoted Massive-SCC, Large-SCC, and Small-SCC, containing different sizes of SCCs. The graphs contain nodes from 25M to 200M with average degree varying from 2 to 6. 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. The parameters for synthetic datasets and their default values are shown in Table I.

Exp-1 (Performance on WEBSPAM-UK2007): Fig. 6(a) and Fig. 6(b) show the time and I/O costs when varying the number of edges of WEBSPAM-UK2007 from 20% to 100% respectively. DFS-SCC cannot stop in the time limit even if the graph contains only 20% of the edges. When $|E|$ increases, the time and I/O consumptions for both Ext-SCC and Ext-SCC-Op increase. The reasons are twofold. Firstly, when $|E|$ increases, the number of iterations in graph contraction increases. This is because when number of edges $|E|$ increases, according to the node selection scheme to construct V_{i+1} in Algorithm 3, more nodes will be selected in V_{i+1}, thus more iterations are needed according to the stop condition of graph contraction in Ext-SCC. Secondly, when $|E|$ increases, the cost to sort and scan edges in each iteration increases, thus more time and I/Os are consumed in each iteration. Ext-SCC-Op outperforms Ext-SCC in all cases since more nodes/edges are removed in each iteration in Ext-SCC-Op.

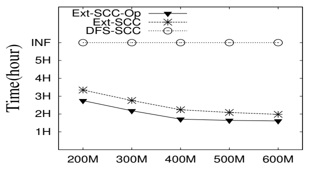
We vary the memory size from $400M$ to $1G$. The results are shown in Fig. 7(a) and Fig. 7(b) for time and I/O costs respectively. When the memory size increases, the time and I/O costs for both Ext-SCC and Ext-SCC-Op decrease. There are two reasons. Firstly, when the memory size increases, the stop condition for graph contraction is easier to be satisfied since more nodes can fit in memory. Secondly, when the memory size increases, the costs of the external sorts in both graph contraction and graph expansion phases decrease. Ext-SCC-Op



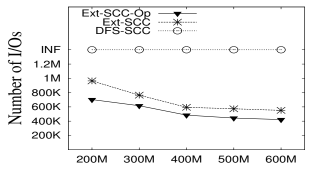
(a) Time (Massive-SCC)



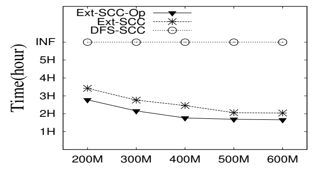
(b) I/Os (Massive-SCC)



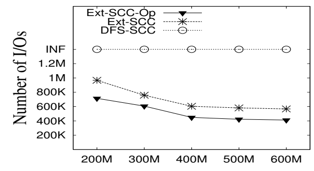
(c) Time (Large-SCC)



(d) I/Os (Large-SCC)

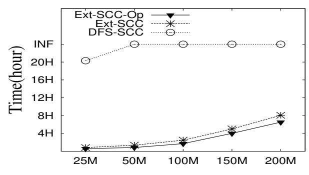


(e) Time (Small-SCC)

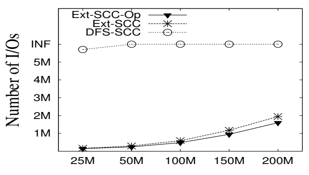


(f) I/Os (Small-SCC)

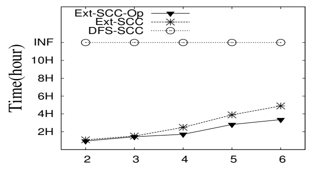
Fig. 8. Synthetic Data: Vary Memory Size



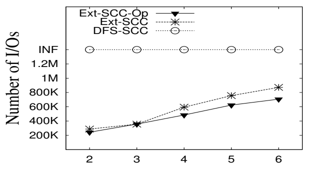
(a) Time (Vary $|V|$)



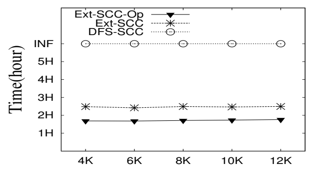
(b) I/Os (Vary $|V|$)



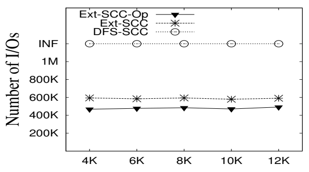
(c) Time (Vary Degree)



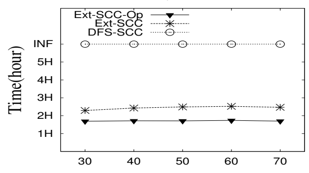
(d) I/Os (Vary Degree)



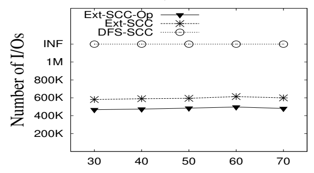
(e) Time (Vary SCC Size)



(f) I/Os (Vary SCC Size)



(g) Time (Vary SCC Num)



(h) I/Os (Vary SCC Num)

Fig. 9. Synthetic Data (Large-SCC)

outperforms Ext-SCC in all cases. When the memory increases from $800M$ to $1G$, the costs for both Ext-SCC and Ext-SCC-Op decrease sharply. The reason is that, in order to process the graph using Semi-SCC, $105,895,908 \times 8 + 256K = 847.4M$ memory is needed, thus when the memory size is $1G$, no iteration is needed and Semi-SCC can be directly applied on the original graph to output all SCCs.

Exp-2 (Vary Memory Size M in Synthetic Data): To test the synthetic data, we vary the memory size M from $200M$ to $600M$. The time and I/O costs on Massive-SCC dataset are shown in Fig. 8(a) and Fig. 8(b) respectively. DFS-SCC cannot stop in limited time in all cases. Similar to the results on the real dataset in Fig. 7, when M increases, the time and I/O costs for both Ext-SCC and Ext-SCC-Op decrease. When M is smaller, the decrease rate is larger. This is because when M is smaller, more iterations are needed for both Ext-SCC and Ext-SCC-Op, and in the graph contraction phase, the contraction rate decreases when the number of iterations increases, since the graph becomes denser with larger number of iterations. Ext-SCC-Op outperforms Ext-SCC by 20% on average for both time and I/O consumptions. Fig. 8(c) and Fig. 8(d) show the results on Large-SCC dataset, and Fig. 8(e) and Fig. 8(f) show the results on Small-SCC dataset. The results for both Large-SCC and Small-SCC datasets are similar to those in the Massive-SCC dataset, and this is true for all the remaining test cases when varying other parameters in synthetic data. In the following, due to the lack of space, we only show the test results on the Large-SCC dataset.

Exp-3 (Vary Node Size $|V|$ in Synthetic Data): We vary the node size $|V|$ from $25M$ to $200M$, and the time and I/O costs are shown in Fig. 9(a) and Fig. 9(b) respectively. When $|V|$ increases, the time and I/O consumptions for both Ext-SCC and Ext-SCC-Op increase. This is because the stop condition for graph contraction is harder to be satisfied when $|V|$ is larger, and the cost on each iteration to scan and sort nodes/edges is larger when $|V|$ is larger. Ext-SCC-Op outperforms Ext-SCC in all test cases. DFS-SCC cannot stop within the time limit when $|V| \geq 50M$. When $|V| = 25M$, DFS-SCC consumes more than 20 hours while both Ext-SCC and Ext-SCC-Op consume less than 1 hour.

Exp-4 (Vary Average Degree in Synthetic Data): We vary the average degree D of nodes from 2 to 6. The time and I/O costs on Large-SCC are shown in Fig. 9(c) and Fig. 9(d) respectively. When D increases, the time and I/O consumptions for both Ext-SCC and Ext-SCC-Op increase. This is because when D increases, the number of edges increases. As a result, more iterations are needed and larger cost is consumed in each iteration as analyzed in Exp-1 when varying the graph size. Ext-SCC-Op outperforms Ext-SCC, and when D is larger, the gap between Ext-SCC-Op and Ext-SCC is larger. This is because when number of edges is larger, more edges can be pruned by the edge reduction techniques used in Ext-SCC-Op.

Exp-5 (Vary SCC Size and SCC Number in Synthetic Data): Fig. 9(e) and Fig. 9(f) show the time and I/O costs when varying the average SCC size from $4K$ to $12K$ respectively. Fig. 9(g) and Fig. 9(h) show the time and I/O costs when varying the number of SCCs from 30 to 70. When either the average SCC size increases, or the number of SCCs increases, the time and I/O costs for both Ext-SCC and Ext-SCC-Op are not influenced much. As analyzed in Section VII, the key factors that influence the cost of Ext-SCC are the number of nodes and the number of edges of the graph. As a result, the size of SCCs and the number of SCCs do not have significant impact on the efficiency of our algorithms as long as $|E(G)|$

and $|V(G)|$ are fixed. This also explains why the results in the three datasets Massive-SCC, Large-SCC, and Small-SCC are similar as stated in Exp-2.

IX. 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, strongly connected components of a directed graph can be computed in $O(|V(G)| + |E(G)|)$ time based on *DFS* [12].

A naive way to externalize the internal *DFS* algorithm requires $O(|E|)$ I/Os. Chiang et al. [10] propose an algorithm with I/O complexity $O(|V| + \frac{|V|}{M} \cdot scan(|E|) + sort(|E|))$. Later, Kumar and Schwabe [17] and Buchsbaum et al. [8] improve the I/O complexity to $O((|V|+\frac{|E|}{B}) \log_2 \frac{|V|}{B}+sort(|E|))$ by maintaining the list of nodes that should not be traversed using tournament trees [17] and buffered repository trees [8], respectively. Despite their theoretical guarantees, these algorithms are considered impractical for general directed graphs that encountered in real applications. Cosgaya-Lozano and Zeh [13] present a contraction based algorithm which contracts SCCs repeatedly until the graph fits in memory, then an internal memory algorithm is used to find the final SCCs. Such an algorithm may end up an infinite loop and cannot compute all SCCs. Both *DFS* based algorithm [8] and contraction based algorithm [13] are introduced in details in Section III.

In addition to external algorithms, there are semi-external algorithms for SCC computation which assume that all nodes of the graph can fit in the main memory. Sibeyn et al. [23] propose a semi-external *DFS*, which can be used to find all SCCs of a graph. Zhang et al. [26] improve such an algorithm by constructing and maintaining a special in-memory spanning tree of the graph. The semi-external algorithms [23] and [26] are introduced in details in Section III.

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. [14] provide an implementation of an external memory minimum spanning tree algorithm based on the ideas of [22], which performs extremely well in practice, even though theoretically inferior to the algorithms of [1], [10]. Ajwani et al. [4], [6] propose implementations of external undirected breadth-first search algorithm with the idea from [18]. Ulrich Meyer et al. [20], [21], [19] 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 [24], [5].

X. Conclusions

In this paper, we study I/O efficient algorithms to find all SCCs for a directed graph, with the assumption that the nodes of the graph cannot reside entirely in memory. We overcome the deficiencies of the existing external SCC computation algorithms, and propose a new two-phase algorithm with graph contraction followed by graph expansion. We analyze the I/O cost of our approach and show that our algorithm can significantly reduce the number of random I/Os. We propose techniques to further reduce the I/O cost of our algorithm and confirm the I/O efficiency of our approaches using extensive experiments on both real and synthetic web scale graphs.

ACKNOWLEDGEMENTS: The work was supported by grant of the Research Grants Council of the Hong Kong SAR, China No. 418512.

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] E. Angel, R. Campigotto, and C. Laforest. Analysis and comparison of three algorithms for the vertex cover problem on large graphs with low memory capacities. *Algorithmic Operations Research*, 6(1):56–67, 2011.

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

[9] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *Proc. of SIGMOD'12*, pages 457–468, 2012.

[10] 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.

[11] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] 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.