External Topological Sorting in Large Graphs

Zhu Qing[1], Long Yuan[2](✉), Fan Zhang[2], Lu Qin[3], Xuemin Lin[2], and Wenjie Zhang[2]

[1] East China Normal University, Shanghai, China
Skullpirate.qing@gmail.com
[2] The University of New South Wales, Sydney, Australia
{longyuan,lxue,zhangw}@cse.unsw.edu.au, fan.zhang3@unsw.edu.au
[3] Centre for Artificial Intelligence, University of Technology Sydney, Sydney, Australia
lu.qin@uts.edu.au

Abstract. Topological sorting is a fundamental problem in graph analysis. Given the fact that real world graphs grow rapidly so that they cannot entirely reside in main memory, in this paper, we study external memory algorithms for the topological sorting problem. We propose a contraction-expansion paradigm and devise an external memory algorithm based on the paradigm for the topological sorting problem. Our new algorithm is efficient due to the introduction of the new paradigm and can be implemented easily by using the fundamental external memory primitives. We conduct extensive experiments on real and synthesis graphs and the results demonstrate the efficiency of our proposed algorithm.

1 Introduction

Graphs have been widely used to represent the relationships of entities in a large spectrum of applications such as social networks, web search, collaboration networks, and biology. With the proliferation of graph applications, research efforts have been devoted to many problems in managing and analyzing graph data. Among them, topological sorting is a fundamental one. Given a directed graph G, topological sorting aims to compute a node numbering in which each node in G is assigned with a non-negative number such that if G contains an edge (u, v), then the assigned number of u is smaller than that of v.

Applications. Topological sorting can be used in many real-world applications:

(1) *Graph structure analysis in social network.* Topological sorting can be used to compute the node importance when analyzing the graph structure in social network [5].
(2) *Job planning and scheduling.* In job planning, the jobs are represented by nodes, and there is an edge from u to v if job u must be completed before job v can be started. Then, a topological sorting gives an order in which to perform the jobs [9].

J. Pei et al. (Eds.): DASFAA 2018, LNCS 10827, pp. 203–220, 2018.
https://doi.org/10.1007/978-3-319-91452-7_14

(3) *A key step to solve other graph problems.* Topological sorting is also an important building block for other graph algorithms, such as separator partitions of planar graphs [11], contour tree simplification [4], multi-objective shortest path computation [12].

Motivation. In the literature, there are efficient in-memory algorithm and semi-external algorithm to compute topological sorting [2,9]. An in-memory algorithm assumes that the graph is resident in main memory while a semi-external algorithm assumes that all nodes of G are kept in main memory. Nevertheless, as the sizes of many real graphs keep growing rapidly, even the nodes of a graph cannot reside entirely in main memory. For example, the Facebook social network contains 1.32 billion nodes and 140 billion edges[1]; and a sub-domain of the web graph of the EU countries contains 1.07 billion nodes and 91 billion edges[2]. In [2], the authors propose an external memory topological sorting algorithm in which the nodes of G cannot fit entirely in memory. It iteratively computes a partial topological sorting of G until the final topological sorting of G is obtained. However, the algorithm involves other complex external memory subroutines which makes it hard to implement and it cannot effectively utilize the available memory to further improve its performance, even when the memory is abundant.

Our Approach. In order to address the drawbacks of the existing solutions, we propose a new paradigm for topological sorting in this paper. Our paradigm contains two phases, namely, graph contraction phase and graph expansion phase. In graph contraction phase, we contract the nodes of the graph iteratively until all nodes of the contracted graph can fit in main memory. Then, we compute the topological sorting for the contracted graph using the efficient semi-external algorithm. In graph expansion phase, the removed nodes are added back into the graph in a reverse order of their removal and the topological sorting for the graph with the new added nodes is computed. Our new approach just uses the fundamental external memory primitives and can effectively utilize the available memory due to the contraction of the input graph and the exploiting of semi-external topological sorting algorithm.

Contributions. In this paper, we make the following contributions:

(1) *A new paradigm for topological sorting.* We investigate the drawbacks of existing semi-external and external topological sorting algorithms and propose a new contraction-expansion paradigm for the topological sorting problem, which can overcome the drawbacks of existing solutions.
(2) *A new external memory topological sorting algorithm.* Following the contraction-expansion paradigm, we devise a new external memory topological sorting algorithm. Our new algorithm just uses the fundamental external memory primitives and exploits the high efficiency of semi-external topological sorting algorithm. Besides, we also analyze the correctness and I/O complexity of our approach.

[1] http://newsroom.fb.com/company-info.
[2] http://law.di.unimi.it/datasets.php.

(3) *Extensive performance studies on large real and synthetic datasets.* We conduct extensive performance studies using large real and synthetic graphs. The experimental results demonstrate the efficiency of our proposed algorithm.

2 Preliminaries

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. We denote the number of nodes and the number of edges of G by n and m, respectively, i.e., $n = |V(G)|$ and $m = |E(G)|$. We use $|G|$ to denote the sum of n and m, i.e., $|G| = |V(G)|+|E(G)|$. Each node $u \in V(G)$ has a unique identity, denoted by $\mathsf{id}(u)$. If there is a directed edge (u, v) in G, we say u is the tail and v is head, u is an in-neighbor of v and v is an out-neighbor of u. For each node $u \in G$, we use $\mathsf{nbr}^-(u, G)$ and $\mathsf{nbr}^+(u, G)$ to denote the set of u's in-neighbors and out-neighbors in G, respectively. For a node u, the in-degree of u, denoted by $\mathsf{deg}^-(u, G)$, is the number of u's in-neighbors and the out-degree of u, denoted by $\mathsf{deg}^+(u, G)$, is the number of u's out-neighbors. And the degree of u, denoted by $\mathsf{deg}(u, G)$, is the sum of u's in-degree and out-degree. For simplicity, we omit G from the notations if the context is self-evident. Given a directed graph G, a path $p = (v_1, v_2, \cdots, v_k)$ is a sequence of k nodes in $V(G)$ such that, for each $v_i (1 \leq i < k)$, $(v_i, v_{i+1}) \in E(G)$. In a directed graph, a path $(v_1, v_2, \cdots, v_k)$ forms a directed cycle if $v_1 = v_k$.

Definition 1 ***(Directed Acyclic Graph).*** *Given a directed graph G, G is a directed acyclic graph (DAG) if and only if there exist no directed cycles in G.*

Definition 2 ***(Topological Sorting).*** *Given a DAG G, a topological sorting of G is a node numbering in which each node is assigned with a non-negative number such that if G contains an edge (u, v), then the assigned number of u is smaller than that of v.*

We use Ω to denote an arbitrary node numbering and $\Omega(u)$ to denote the number assigned to a node u by Ω. For a graph G, we use Ω_G to denote the topological sorting of G and $\Omega_G(u)$ to denote the number assigned to a node u by Ω_G. We also call $\Omega_G(u)$ the topological sorting number of u.

Problem Statement. In this paper, we study the problem of computing the topological sorting for a given DAG G in an external memory model, namely, not only G but also $V(G)$ cannot reside entirely in main memory. We use the standard external memory model proposed in [1]. It assumes that the main memory can only keep M elements while the remaining are kept in blocks on disk, where one block contains B elements. Suppose one I/O access will read/write B elements (one block) from/into disk into/from main memory. External memory model contains two fundamental primitives: scanning N elements ($\mathsf{scan}(N)$) and sorting N elements ($\mathsf{sort}(N)$). The I/O complexity of $\mathsf{scan}(N)$ is $\Theta(\frac{N}{B})$ I/Os, and the I/O complexity of $\mathsf{sort}(N)$ is $O(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$ I/Os.

3 Existing Solutions

3.1 Semi-external Topological Sorting Algorithm

In this section, we first introduce the semi-external topological sorting algorithm SemiTS [2], which assumes that the nodes of the graph can reside in main memory. It is based on the following property: given a graph G, a post-order traversal on a depth first search tree of G visits nodes in the reverse order of a topological sorting of G. Therefore, SemiTS computes the topological sorting in a semi-external manner by using the existing semi-external depth first search algorithm. SemiTS is shown in Algorithm 1.

Algorithm 1. SemiTS(Graph G)

```
1: compute a DFS tree $T$ of $G$ with root $r$ using the semi-external DFS algorithm
   [18];
2: $\Omega_G \leftarrow \emptyset$; $i \leftarrow 1$; Stack $S \leftarrow \emptyset$;
3: PostOrder($r, T$);
4: **while** $S \neq \emptyset$ **do**
5:     $u \leftarrow S$.pop(); $\Omega_G(u) \leftarrow i$; $i \leftarrow i + 1$;
6: **return** $\Omega_G$;
7: **Procedure** PostOrder(TreeNode $t$, Tree $T$)
8: **if** $t$ = NULL **then return**;
9: **for each** child $t_c$ of $t$ in the corresponding DFS order **do**
10:    PostOrder ($t_c, T$);
11: $S$.push($t$);
```

SemiTS first computes a DFS tree T of G by using the state-of-the-art semi-external depth first search algorithm [18] (line 1). To facilitate the reversing of the post-order, SemiTS utilizes a stack S initializing with $\emptyset$ (line 2). Then, it conducts a post-order traversal on T (line 3). When the traversal finishes, SemiTS pops the nodes in S and assigns $\Omega_G(u)$ to u (line 4–6). Procedure PostOrder performs a post-order traversal on a DFS tree T starting from t. For a given tree node t, it first visits all its children in the corresponding DFS order (line 9–10) and then pushes t into the stack S (line 11).

3.2 External Memory Topological Sorting Algorithm

The state-of-the-art external memory algorithm, IterTS, is proposed in [2]. The main idea of IterTS is that if we have an arbitrary node numbering Ω, we can obtain the topological sorting Ω_G based on Ω by repeatedly adjusting the assigned numbers of such nodes u and v that $(u, v) \in E(G)$ but $\Omega(u) > \Omega(v)$.

For an arbitrary node numbering Ω, Algorithm 2 calls an edge $(u, v) \in E(G)$ topological sorted edge if $\Omega(u) < \Omega(v)$ and use $|\Omega|$ to denote the number of topological sorted edges for Ω. Algorithm 2 starts with an initial node numbering

Algorithm 2. IterTS(Graph G)

```
1: $\Omega \leftarrow \mathsf{InitOrder}(G)$;
2: while $|\Omega| < m$ do
3:     $\Omega \leftarrow \mathsf{ImproveOrder}(G, \Omega)$;
4: return $\Omega$;
5: Procedure InitOrder($G$)
6:   compute an out-tree $T$ of $G$ from $s$;   // $s$ is the only node in $G$ with $\mathsf{deg}^-(s) = 0$.
7:   compute two orderings $\Omega_l$ and $\Omega_r$ by Euler Tour [8] on $T$;
8:   if $|\Omega_l| < |\Omega_r|$ then return $\Omega_r$;
9:   else return $\Omega_l$;
10: Procedure ImproveOrder($G, \Omega$)
11:   compute an out-tree $T$ of $G$ from $s$ based on $\Omega$;  // $s$ is the only node in $G$ with $\mathsf{deg}^-(s) = 0$.
12:   for each edge $(u, v)$ following the pre-order traverse of $T$ do
13:       $\Omega'(v) \rightarrow \max\{\Omega(v), \Omega'(u) + 1\}$;
14:   for each topological sorted edge $(u, v)$ w.r.t $\Omega$ in increasing order according to $\Omega(v)$ do
15:       $\Omega''(v) \rightarrow \max\{\Omega'(v), \Omega''(u) + 1\}$;
16:   compute $\Omega_{\mathsf{new}}$ by sorting nodes based on $\Omega''$;
17:   return $\Omega_{\mathsf{new}}$;
```

by invoking InitOrder (line 1). After obtaining the initial Ω, it iteratively improves Ω by invoking ImproveOrder until $|\Omega|$ is m (line 2–4). For brevity, Algorithm 2 assumes G has only a single node s with $\mathsf{deg}^-(s) = 0$. Procedure InitOrder aims to compute a node numbering Ω with a big $|\Omega|$ heuristically. It computes an out-tree T rooted at s of G (line 6), computes two node numbering Ω_l and Ω_r through Euler Tour [8] on T (line 7) and returns the node numbering with a bigger number of topological sorted edges (line 8–9).

Procedure ImproveOrder intends to compute a new Ω_{new} based on Ω such that $|\Omega_{\mathsf{new}}| > |\Omega|$. It first computes an out-tree T rooted at s such that for each node u, its parent in T has the maximum assigned number among all its in-neighbors in G (line 11). After that, it adjusts the assigned number for each node based on the tree edges in T and the topological sorted edges and generates two temporary node numberings Ω' and Ω'', respectively. For each tree edge (u, v) in T, it iterates them following the pre-order traverse of T and adjusts $\Omega'(v)$ by setting it as $\max\{\Omega(v), \Omega'(u)+1\}$ (line 12–13). For each topological sorted edge (u, v) w.r.t Ω, it iterates them in the increasing order according to their $\Omega(v)$ and adjusts the assigned number of v by setting it as $\max\{\Omega'(v), \Omega''(u) + 1\}$ (line 14–15). In this way, the edges in T and the topological sorted edges w.r.t Ω are still topological sorted edge w.r.t Ω''. Then, it computes Ω_{new} by sorting the nodes based on Ω'' and returns Ω_{new} (line 16–17). Note that Algorithm 2 just shows the main idea of IterTS and omits the details about I/O issues. Given a graph G, Algorithm 2 finishes the topological sorting of G in $O(n \cdot \mathsf{sort}(|G|))$ I/Os.

3.3 Drawbacks of Existing Solutions

Regarding SemiTS, although it is significantly efficient in terms of computing topological sorting compared with IterTS [2]. It assumes that the nodes of the graph can fit in main memory, which does not satisfy our requirements. Regarding IterTS, although IterTS addresses the topological sorting problem upon the external memory model, it has the following two drawbacks: (1) Underutilization of available memory. As shown in Algorithm 2, IterTS does not take the available memory into consideration when computing Ω_G, which renders the algorithm unable to benefit from the availability of extra memory to further improve its performance, even when the memory is abundant. (2) Involvement of other complex external memory subroutines. Besides the two fundamental primitives scan and sort in external memory model, IterTS involves other complex external memory subroutines, such as, external list ranking, external Euler tour and external priority queue. All these subroutines are complex and hard to implement.

4 A New Approach

4.1 Contraction-Expansion Paradigm

As discussed in Sect. 3, SemiTS is efficient and can fully utilize the available memory, but it cannot handle the scenario that the nodes of the graph cannot be loaded in main memory. On the other hand, IterTS is an external memory solution. However, it cannot use the available memory effectively, limiting the improvement of its performance from the availability of extra memory. Motivated by this, we propose a new paradigm combining the merits of these two existing solutions. Our paradigm has two phases, namely, graph contraction phase and graph expansion phase.

In the *graph contraction phase*, we generate a list of graph $G_1, G_2, \cdots, G_k$, where $G_1 = G$, and for each $1 \leq i < k$, G_{i+1} is generated by removing a batch of nodes from G_i. The contraction phase stops when the nodes of the newly generated graph can fit in memory. In the *graph expansion phase*, after computing the topological sorting of G_k using SemiTS, the removed nodes are added back to the graph in the reverse order of their removal during 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 batch of nodes are added back, we compute the topological sorting for the graph with the new added nodes. More specifically, given that the topological sorting of G_k is computed using SemiTS, we compute the topological sorting of $G_{k-1}, G_{k-2}, \cdots, G_1$ in order and the topological sorting of G_i is computed based on the topological sorting of G_{i+1}. Since $G_1 = G$, the topological sorting of the original graph is obtained when the expansion phase finishes.

Advantages of Our Paradigm. Compared with the existing solutions, the advantages of our approach are twofold: (1) Regarding SemiTS, due to the introduction of contraction and expansion phases, our approach can handle scenario that the memory can not fit the nodes of the given graph. Therefore, our

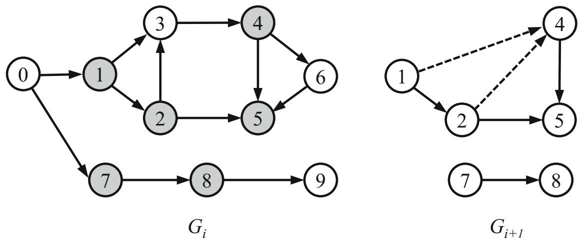


Fig. 1. Contraction phase

approach is a genuine external memory solution. (2) In contrast to IterTS, our approach can utilize the available memory effectively. In our approach, we stop the contraction phase once the available memory can fit the nodes of the contracted graph and directly use the SemiTS to compute the topological sorting of the contracted graph. In other words, the more available memory we have, the less contraction we would conduct. As a result, our approach is sensitive to the memory size and can significantly benefit from the extra memory considering the high performance of SemiTS. Besides, our contraction-expansion paradigm can be achieved by just using `scan` and `sort`, which makes our approach easy to implement.

Due to the adoption of contraction-expansion paradigm, our approach can obtain the merits of existing solutions. However, to make our paradigm practically applicable, the following issues should be addressed when designing our algorithm:

- *Contractility:* For a given G_i, the number of nodes in the contracted graph G_{i+1} should be smaller than that in G_i, i.e., $V(G_{i+1}) \subset V(G_i)$.
- *Expandability:* For a given G_{i+1} and $\Omega_{G_{i+1}}$, we should be able to compute Ω_{G_i} based on $\Omega_{G_{i+1}}$.
- *External-Memory Feasibility:* All the algorithms should be able to be implemented in an external-memory manner.

In the following, we will introduce how to address these issues one by one.

4.2 Contraction Phase

In the contraction phase, we focus on solving the contractility issue. It is trivial if we just consider the contractility alone. For example, we can arbitrarily select a batch of nodes and generate a new graph by removing those un-selected nodes. However, this approach makes satisfying the expandability requirement hard in the expansion phase. Taking the contractility and expandability into consideration simultaneously, we propose a vertex cover based contraction method, which is based on the following definition:

Definition 3 ***(Vertex Cover).*** *Given a graph G, a vertex cover of graph is a subset V' of $V(G)$ such that each edge of the graph is incident to at least one node of the set, i.e., $(u, v) \in E(G) \Rightarrow u \in V' \vee v \in V'$.*

Example 1. Consider the graph G_i in Fig. 1, the node set $\{v_1, v_2, v_4, v_5, v_7, v_8\}$ is a vertex cover of G_i, which are shown with grey shadow. And each edge of G_i is incident to at least one node in the vertex cover, for example, edge (v_0, v_1) is incident to v_1.

In the contraction phase, for an input graph G_i, we compute a vertex cover V' of G_i and consider the node in V' as $V(G_{i+1})$. The benefits of choosing a vertex cover V' of G_i as $V(G_{i+1})$ are twofold: (1) Based on Definition 3, we can easily get a vertex cover V' such that $V' \subset V(G_i)$. Therefore, the contractility is satisfied. (2) As we choose V' as $V(G_{i+1})$, for any edge $(u, v) \in E(G_i)$, either $u \in V(G_{i+1})$ or $v \in V(G_{i+1})$. Therefore, if we know $\Omega_{G_{i+1}}$, then, for a node $w \in V(G_i) \setminus V(G_{i+1})$, the relative numerical magnitudes of its in-neighbors and out-neighbors in Ω_{G_i} can be determined. As a result, we can obtain Ω_{G_i} by just considering w together with its neighbours, which satisfies the expandability requirements (the details will be discussed in Sect. 4.3).

We can obtain $V(G_{i+1})$ by computing a vertex cover V' of G_i. However, just taking the edge of G_i induced by V' as $E(G_{i+1})$ is not sufficient for the topological sorting problem. For example, in Fig. 1, if we just keep the edges induced by $\{v_1, v_2, v_4, v_5, v_7, v_8\}$ as $E(G_{i+1})$, the information that $\Omega_{G_{i+1}}(v_1)$ should be smaller than $\Omega_{G_{i+1}}(v_4)$ is lost. Therefore, we include two types of edges in $E(G_{i+1})$:

Definition 4 ***(Induced Edge).*** *Given a graph G_i and its vertex cover V', let $u, v \in V'$. If $(u, v) \in E(G_i)$, then (u, v) is an induced edge for G_{i+1}.*

Definition 5 ***(Contracted Edge).*** *Given a graph G_i and its vertex cover V', let u, v, w be three nodes in $V(G_i)$ and $u, v \in V' \wedge w \notin V'$. If $(u, w) \in E(G_i) \wedge (w, v) \in E(G_i) \wedge (u, v) \notin E(G_i)$, then (u, v) is a contracted edge for G_{i+1}.*

Example 2. Consider G_i in Fig. 1 again, the contracted graph G_{i+1} is shown on the right of Fig. 1. $V(G_{i+1})$ is the vertex cover of G_i. $E(G_{i+1})$ consists of two types of edges: the induced edges which are shown with solid line, such as (v_1, v_2), and the contracted edges which are shown with dotted line, such as (v_1, v_4).

Algorithm Design. Our contraction algorithm, Contract, is shown in Algorithm 3. It first computes $V(G_{i+1})$ by computing a vertex cover in an external-memory manner (line 1), then it computes $E(G_{i+1})$ by invoking conEdge (line 2).

Regarding computing the vertex cover, in order to reduce the number of iterations in the contraction phase, $|V_{i+1}|$ should be as small as possible. This leads to the minimum vertex cover problem which is NP-hard [9]. In the literature, [3] proposes an external memory algorithm to find a vertex cover V' with an approximation ratio $\sqrt{\Delta(G)}/2 + 3/2$, where $\Delta(G)$ is the maximum degree of G. It defines a total order $\prec$ for all nodes in the graph based on the node's degree. For each edge (u, v) in $E(G_i)$, if $u \prec v$, then u is added to V', otherwise, v is added to V'. Its I/O complexity is $O(\mathsf{sort}(G_i))$. Since we focus on external-memory topological sorting in this paper, we just use the algorithm in [3] directly to compute the vertex cover.

Algorithm 3. Contract(Graph G_i)

```
1: $V(G_{i+1}) \leftarrow$ compute a vertex cover of $G_i$ in an external-memory manner by [3];
2: $E(G_{i+1}) \leftarrow \mathsf{conEdge}(G_i, V(G_{i+1}))$;

3: **Procedure** $\mathsf{conEdge}(G_i, V(G_{i+1}))$
4: $E^- \leftarrow \mathsf{sort}\ (u,v) \in E(G_i)$ by $(\mathsf{id}(v), \mathsf{id}(u))$;
5: $E^+ \leftarrow \mathsf{sort}\ (u,v) \in E(G_i)$ by $(\mathsf{id}(u), \mathsf{id}(v))$;
6: $E_{\mathsf{ind}} \leftarrow (u,v) \in E(G_i)$ with $u \in V(G_{i+1})$ by sequential scan $V(G_{i+1})$ and $E^+$;
7: $E_{\mathsf{ind}} \leftarrow \mathsf{sort}\ (u,v) \in E_{\mathsf{ind}}$ by $(\mathsf{id}(v), \mathsf{id}(u))$;
8: $E_{\mathsf{ind}} \leftarrow (u,v) \in E(G_i)$ with $u, v \in V(G_{i+1})$ by sequential scan $V(G_{i+1})$ and $E_{\mathsf{ind}}$;
9: $E_{\mathsf{rem}} \leftarrow (u,v) \in E(G_i)$ with $v \notin V(G_{i+1})$ by sequential scan $V(G_{i+1})$ and $E^-$;
10: $E_{\mathsf{rem}} \leftarrow (u, v, \mathsf{nbr}^+(v, G_i))$ with $v \notin V(G_{i+1})$ by sequential scan $E_{\mathsf{rem}}$ and $E^+$;
11: **for each** edge $(u,v) \in E_{\mathsf{rem}}$ **do**
12:     **for each** $w \in \mathsf{nbr}^+(v, G_i)$ by sequential scan of $E_{\mathsf{rem}}$ **do**
13:         $E_{\mathsf{con}} \leftarrow E_{\mathsf{con}} \cup (u,w)$;
14: $E(G_{i+1}) \leftarrow E_{\mathsf{ind}} \cup E_{\mathsf{con}}$;
15: **return** $E(G_{i+1})$;
```

Procedure conEdge computes $E(G_{i+1})$ externally for a given G_i and $V(G_{i+1})$. As discussed above, $E(G_{i+1})$ consists of two types of edges, namely, the induced edges and the contracted edges. We denote them as E_{ind} and E_{con}, respectively. $E^-$and E^+ be the edges of G_i by grouping in-coming and out-going edges for each node in G_i, which can be obtained by external sorting based on $(\mathsf{id}(v), \mathsf{id}(u))$ and $(\mathsf{id}(u), \mathsf{id}(v))$, respectively (line 4–5). Here, sorting based on $(\mathsf{id}(v), \mathsf{id}(u))$ means when sorting edges (u, v), we sort them based on $\mathsf{id}(v)$. If two edges have the same $\mathsf{id}(v)$, we sort them based on $\mathsf{id}(u)$. conEdge first constructs E_{ind} (line 6–8) and E_{con} (line 9–13), and then unions E_{con} and E_{ind} to construct $E(G_{i+1})$ (line 14–15).

To construct E_{ind}, conEdge first computes the edges (u, v) with $u \in V(G_{i+1})$ by a sequential scan of $V(G_{i+1})$ and E^+ simultaneously. When scanning $V(G_{i+1})$ and E^+, for each edge $(u, v) \in E^+$, if $u \in V(G_{i+1})$, then (u, v) is added to E_{ind} (line 6). Then, it sorts all edges $(u, v) \in E_{\mathsf{ind}}$ based on $(\mathsf{id}(v), \mathsf{id}(u))$ (line 7). At last, it computes the edges (u, v) with $u, v \in V(G_{i+1})$ by a sequential scan of $V(G_{i+1})$ and E_{ind} simultaneously (line 8). conEdge constructs E_{con} based on the set of edges that will be removed from $E(G_i)$. It first computes the edges E_{rem} in which each edge (u, v) with $v \notin V(G_{i+1})$. E_{rem} can be computed by a single sequential scan of $V(G_{i+1})$ and E^- on disk (line 9). When scanning $V(G_{i+1})$ and E^-, for each edge $(u, v) \in E^-$, if $v \notin V(G_{i+1})$, then (u, v) is added to E_{rem}. After constructing E_{rem}, for each edge $(u, v) \in E_{\mathsf{rem}}$, we compute the out-neighbors of v, which can be obtained by a single sequential scan of E_{rem} and E^+ (line 10). Then, conEdge constructs E_{con} using a single sequential scan of all edges in E_{rem} (line 11–13). In E_{rem}, for each node v removed from G_i, each of its in-neighbors u in G_i and its out-neighbors $\mathsf{nbr}^+(v, G_i)$ are stored together in the form $(u, v, \mathsf{nbr}^+(v, G_i))$. When scanning each removed in-coming edge (u, v) of v, the removed out-going edge (v, w) of v can be obtained in the same sequential scan of E_{rem}. Then, conEdge adds a contracted edge (u, w) into E_{ind}.

Lemma 1. *Let G_i be the input graph and G_{i+1} be the contracted graph for Algorithm 3, respectively, then, $V(G_{i+1}) \subset V(G_i)$.*

Proof. Let v be the node in G_i such that for any other node $u \in V(G_i)$, we have $u \prec v$. Based on the method in [3] to compute the vertex cover, v cannot be added into $V(G_{i+1})$, because there does not exits an edge (u, v) or an edge (v, u) with $v \prec u$. Thus, the lemma holds. □

Lemma 2. *Let G_i be the input graph and G_{i+1} be the contracted graph for Algorithm 3, respectively, assume u and v be two nodes in $V(G_{i+1})$, then $\Omega_{G_{i+1}}(u) < \Omega_{G_{i+1}}(v)$ if and only if $\Omega_{G_i}(u) < \Omega_{G_i}(v)$.*

Proof. We can prove the lemma based on the procedure of Algorithm 3 directly. □

Lemma 3. *Let G_i be the input graph and G_{i+1} be the contracted graph, the I/O complexity of Algorithm 3 is $O(\mathsf{sort}(|G_i|) + \mathsf{scan}(|G_{i+1}|))$.*

Proof. This lemma can be proved directly based on the procedure of Algorithm 3. □

4.3 Expansion Phase

In expansion phase, we aim to obtain Ω_{G_i} through the computed $\Omega_{G_{i+1}}$. Based on Definition 3, we can divided the nodes in $V(G_i)$ into three types:

- Type-I: the nodes in $V(G_i) \setminus V(G_{i+1})$ without in-neighbors in G_i, i.e., $u \in V(G_i) \setminus V(G_{i+1}) \wedge \mathsf{deg}^-(u, G_i) = 0$.
- Type-II: the nodes in $V(G_i) \setminus V(G_{i+1})$ with in-neighbors in G_{i+1} and the nodes in $V(G_{i+1})$ with out-neighbors in G_i, i.e., $\{u \in V(G_i) \setminus V(G_{i+1}) \wedge \mathsf{deg}^-(u, G_i) > 0\} \cup \{u \in V(G_{i+1}) \wedge \mathsf{deg}^+(u, G_i) > 0\}$.
- Type-III: the nodes in $V(G_{i+1})$ without out-neighbors in G_i, i.e., $u \in V(G_{i+1}) \wedge \mathsf{deg}^+(u, G_i) = 0$.

For the Type-I nodes, since they have no in-neighbors in G_i, we can obtain their topological sorting numbers in G_i by assigning their numbers first. For the Type-II nodes, we consider two types of edges: (1) $(u, v) \in E(G_i) \wedge u \in V(G_{i+1}) \wedge v \in V(G_i) \setminus V(G_{i+1})$ (2) $(u, v) \in E(G_i) \wedge u \in V(G_{i+1}) \wedge v \in V(G_{i+1})$. We use E_{exp} to denote these edges. By sorting the edges in E_{exp} based on $\Omega_{G_{i+1}}(u)$, we can observe:

Observation 1. *For the Type-II nodes of $V(G_i)$ which are also in $V(G_{i+1})$, they appear in the tail position of edges in E_{exp} and their appearing order in the tail position of edges in E_{exp} is consistent with their numerical magnitude in the topological sorting.*

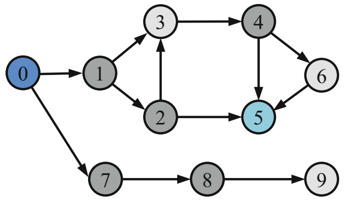
Algorithm 4. Expand (Graph G_i, Graph G_{i+1}, TopologicalSorting $\Omega_{G_{i+1}}$)

```
1: $\mathsf{num} \leftarrow 1$; $v_{\mathsf{pos}} \leftarrow 1$; $k \leftarrow 1$; $\mathsf{curr} \leftarrow \emptyset$;
2: compute $E^-$ and $E^+$ as line 4-5 of Algorithm 3;
3: $E_{\mathsf{rem}} \leftarrow (u,v) \in E(G_i)$ with $v \in V(G_i) \setminus V(G_{i+1})$ by sequential scan $V(G_{i+1})$ and $E^-$;
4: $V^-_{\mathsf{zero}} \leftarrow v \in V(G_i) \setminus V(G_{i+1})$ with $\mathsf{deg}^-(v, G_i) = 0$ by sequential scan $V(G_i) \setminus V(G_{i+1})$ and $E_{\mathsf{rem}}$;
5: for each $u \in V^-_{\mathsf{zero}}$ do
6:     $\Omega_{G_i}(u) = \mathsf{num}$; $\mathsf{num} \leftarrow \mathsf{num} + 1$;
7: $E_{\mathsf{ind}} \leftarrow$ compute as line 6-8 of Algorithm 3;
8: $E_{\mathsf{rem}} \leftarrow (u, v, \mathsf{REM})$ for each $(u,v) \in E_{\mathsf{rem}}$; $E_{\mathsf{ind}} \leftarrow (u, v, \mathsf{IND})$ for each $(u,v) \in E_{\mathsf{ind}}$;
9: $E_{\mathsf{exp}} \leftarrow E_{\mathsf{rem}} \cup E_{\mathsf{ind}}$;
10: $E_{\mathsf{exp}} \leftarrow$ sort $(u, v, \mathsf{FLAG}) \in E_{\mathsf{exp}}$ based on $(\Omega_{G_{i+1}}(u), \mathsf{id}(v))$;
11: for each $(u, v, \mathsf{FLAG}) \in E_{\mathsf{exp}}$ do
12:     if $\mathsf{FLAG} = \mathsf{REM}$ then
13:         $E_{v_{\mathsf{pos}}} \leftarrow (v, v_{\mathsf{pos}})$;
14:     $v_{\mathsf{pos}} \leftarrow v_{\mathsf{pos}} + 1$;
15: $E_{v_{\mathsf{pos}}} \leftarrow$ sort $(v, v_{\mathsf{pos}}) \in E_{v_{\mathsf{pos}}}$ by $(\mathsf{id}(v), v_{\mathsf{pos}})$;
16: $E_{v_{\mathsf{pos}}} \leftarrow (v, v'_{\mathsf{pos}}) \in E_{v_{\mathsf{pos}}}$ with $v'_{\mathsf{pos}} = \max\{v_{\mathsf{pos}}\}$ for $v$ by sequential scan $E_{v_{\mathsf{pos}}}$;
17: $E_{\mathsf{exp}} \leftarrow$ sort $(u, v, \mathsf{FLAG}) \in E_{\mathsf{exp}}$ based on $(\mathsf{id}(v), \mathsf{id}(u))$;
18: $E_{\mathsf{exp}} \leftarrow (u, v, \mathsf{FLAG}, v'_{\mathsf{pos}})$ by sequential scan $E_{v_{\mathsf{pos}}}$ and $E_{\mathsf{exp}}$;
19: $E_{\mathsf{exp}} \leftarrow$ sort $(u, v, \mathsf{FLAG}, v'_{\mathsf{pos}}) \in E_{\mathsf{exp}}$ based on $(\Omega_{G_{i+1}}(u), \mathsf{id}(v))$;
20: for each $(u, v, \mathsf{FLAG}, v'_{\mathsf{pos}}) \in E_{\mathsf{exp}}$ do
21:     if $\mathsf{curr} \neq u$ then
22:         $\Omega_{G_i}(u) = \mathsf{num}$; $\mathsf{num} \leftarrow \mathsf{num} + 1$; $\mathsf{curr} \leftarrow u$;
23:     if $k = v'_{\mathsf{pos}}$ then
24:         $\Omega_{G_i}(v) = \mathsf{num}$; $\mathsf{num} \leftarrow \mathsf{num} + 1$;
25:     $k \leftarrow k + 1$;
26: for each $u$ not assigned topological sorting number by sequential scan $V(G_i)$ and $\Omega_{G_i}$ do
27:     $\Omega_{G_i}(u) = \mathsf{num}$; $\mathsf{num} \leftarrow \mathsf{num} + 1$;
```

Observation 2. *For the Type-II nodes of $V(G_i)$ which are in $V(G_i) \setminus V(G_{i+1})$, they only appear in the head position of edges in E_{exp} and for a specific node in this case, its last appearance in E_{exp} is after the first appearances of its in-neighbors and before the first appearances of its out-neighbors in the tail position of edges in E_{exp}.*

Following these two observations, we can obtain the topological sorting of Type-II nodes in G_i based on their appearing orders in E_{exp}. For the Type-III nodes, since they have no out-neighbors in G_i, we can obtain their topological sorting numbers in G_i by handling them at last. By combining the above three cases together, we can obtain Ω_{G_i}.

Algorithm Design. Our expansion algorithm, Expand, is shown in Algorithm 4. Expand first computes the topological number for the Type-I nodes. It first computes E^- and E^+ similarly as Algorithm 3 (line 2). By a simultaneous sequential scan of $V(G_{i+1})$ and E^-, it obtains the edges $(u, v) \in E(G_i)$ with



G_i

Node	$\Omega_{G_{i+1}}$(u)
v_1	1
v_2	2
v_4	3
v_5	4
v_7	5
v_8	6

$\Omega_{G_{i+1}}$

u	v	FLAG	v'$_{pos}$
v_1	v_2	IND	-
v_1	v_3	REM	3
v_2	v_3	REM	3
v_2	v_5	IND	-
v_4	v_5	IND	-
v_4	v_6	REM	6
v_7	v_8	IND	-
v_8	v_9	REM	8

E_{exp}

Node	Ω_{G_i}(u)
v_0	1
v_1	2
v_2	3
v_3	4
v_4	5
v_6	6
v_7	7
v_8	8
v_9	9
v_5	10

Ω_{G_i}

Fig. 2. Expansion phase (Color figure online)

$v \in V(G_i) \setminus V(G_{i+1})$ and stores them as E_{rem} (line 3). After that, Expand computes the nodes without in-neighbors in G_i by a simultaneous sequential scan of $V(G_i) \setminus V(G_{i+1})$ and E_{rem} and assigns topological sorting number to them (line 4–6).

Then, Expand handles the topological sorting number assignment for the Type-II nodes. To obtain their topological sorting numbers, Expand first computes the two types of edges discussed above. The edges $(u, v) \in E(G_i) \wedge u \in V(G_{i+1}) \wedge v \in V(G_i) \setminus V(G_{i+1})$ has been computed in line 3. For the edges $(u, v) \in E(G_i) \wedge u \in V(G_{i+1}) \wedge v \in V(G_{i+1})$, they can be computed similarly as Algorithm 3 (line 7). After that, Expand aims to compute the last appearance for the Type-II nodes in $V(G_i) \setminus V(G_{i+1})$ in the edge sets E_{exp} sorted by $\Omega_{G_{i+1}}(u)$. To achieve this goal, Expand first arguments the edges in E_{rem} and E_{ind} with an indicator FLAG, which contains two values: REM and IND, and combines E_{rem} and E_{ind} together in E_{exp} (line 8–9). After that, Expand sorts the edges $(u, v, \mathsf{FLAG}) \in E_{\mathsf{exp}}$ based on $(\Omega_{G_{i+1}}(u), \mathsf{id}(v))$ and computes the last appearance position for the Type-II nodes in $V(G_i) \setminus V(G_{i+1})$ in E_{exp} in line 11–18. Note that for the edges with FLAG value IND, we only set their v'_{pos} as null in line 18 and sorting by $(\Omega_{G_{i+1}}(u), \mathsf{id}(v))$ can be achieved by first augmenting the edges with $\Omega_{G_{i+1}}(u)$, we omit the details for brevity. After sorting edges $(u, v, \mathsf{FLAG}, v'_{\mathsf{pos}}) \in E_{\mathsf{exp}}$ based on $(\Omega_{G_{i+1}}(u), \mathsf{id}(v))$ (line 19), Expand sequentially scans edge $(u, v, \mathsf{FLAG}, v'_{\mathsf{pos}}) \in E_{\mathsf{exp}}$ (line 20). If u is the first time to scan, it assigns the topological sorting number for u (line 21–22). If v reaches its last position, it assigns the topological sorting number for v (line 23–25).

For the Type-III nodes, Expand just conducts a simultaneous sequential scan on $V(G_i)$ and $\Omega(G_i)$ and assigns their topological sorting numbers to them (line 26–27).

Example 3. Figure 2 shows an example of expansion phase for G_i. The nodes in $V(G_i)$ are divided into three types: v_0 is the Type-I node, which is shown in blue. v_5 is the Type-III node, which is shown in aqua. The remaining nodes

Algorithm 5. CoExTS(Graph G)

```
1: $G_1 \leftarrow G$; $i \leftarrow 1$;
2: while $M < c \times |V(G_i)|$ do
3:     $G_{i+1} \leftarrow \mathsf{Contract}(G_i)$; $i \leftarrow i + 1$;
4: $\Omega_{G_i} \leftarrow \mathsf{SemiTS}\ (G_i)$;
5: while $i > 1$ do
6:     $i \leftarrow i - 1$; $\Omega_{G_i} \leftarrow \mathsf{Expand}\ (G_i, G_{i+1}, \Omega_{G_{i+1}})$;
7: output topological sorting $\Omega_G$ of $G$;
```

are Type-II nodes. The Type-II nodes in $V(G_{i+1})$, such as v_1, are shown in dark grey. The Type-II node in $V(G_i) \setminus V(G_{i+1})$, such as v_3, are shown in light grey. $\Omega_{G_{i+1}}$ shows the computed topological sorting for G_{i+1}. E_{exp} consists of two types of edges: (1) the edges (u, v) with $u \in V(G_{i+1}) \wedge v \in V(G_{i+1})$, such as (v_1, v_2), which are shown with an FLAG value IND. (2) the edges (u, v) with $u \in V(G_{i+1}) \wedge v \in V(G_i) \setminus V(G_{i+1})$, such as (v_1, v_3), which are shown with an FLAG value REM. The edges in E_{exp} are sorted by $\Omega_{G_{i+1}}(u)$. The v'_{pos} value for an IND edge is null, which is shown as $-$. The v'_{pos} value for an REM edge (u, v) is the last appearance position of v. For example, the v'_{pos} value for (v_1, v_3) is 3 as the last appearing position of v_3 in E_{exp} is 3. When computing Ω_{G_i}, we first compute the topological sorting number for Type-I nodes and v_0 is assigned with topological sorting number 1. When computing the topological number for Type-II nodes, we scan $(u, v, \mathsf{FLAG}, v'_{pos}) \in E_{\mathsf{exp}}$. If current u is different from last u, we assign the topological sorting number for u. And if current position is v'_{pos}, we assign the topological sorting number for v. At last, we assign the topological sorting number of Type-III nodes. Ω_{G_i} is shown on the right of Fig. 2.

Lemma 4. *Algorithm 4 computes Ω_{G_i} correctly.*

Proof. The correctness of Algorithm 4 can be directly derived from above analysis. □

Lemma 5. *The I/O complexity of Algorithm 4 is $O(\mathsf{sort}(|G_i|) + \mathsf{scan}(|G_{i+1}|))$.*

Proof. This lemma can be proved directly based on the procedure of Algorithm 4. □

4.4 Our Approach

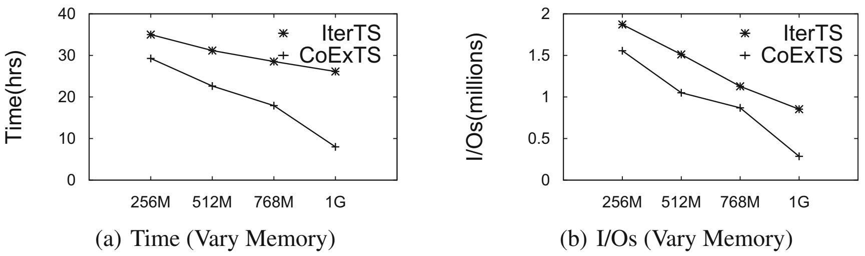
Our complete algorithm CoExTS is shown in Algorithm 5. It follows the contraction-expansion paradigm as discussed above. For a given graph G, it iteratively contracts G until the nodes of the contracted graph G_i can be loaded in memory (line 2–3). In line 2, M represents the size of available memory size and c is a constant factor. After computing Ω_{G_i} with the semi-external algorithm SemiTS (line 4), CoExTS conducts the expansion phase in the reverse order for the graphs generated in the contraction phase (line 5–6). At last, it outputs Ω_G of the original input graph G (line 7).

Theorem 1. *Given a graph G, Algorithm 5 computes Ω_G correctly.*

Proof. The correctness of Algorithm 5 can be directly derived from above analysis. □

Theorem 2. *Given a graph G, let $G_1, G_2, \cdots, G_k$ be the graphs generated in the contraction phase and $\mathsf{IO}_{\mathsf{semi}}$ is the I/O complexity of* SemiTS*, the I/O complexity of Algorithm 5 is $O((\sum_{i=1}^{k} \mathsf{sort}(|G_i|)) + \mathsf{IO}_{\mathsf{semi}})$.*

Proof. This theorem can be proved according to Lemmas 3 and 5 directly. □



(a) Time (Vary Memory) (b) I/Os (Vary Memory)

Fig. 3. UK-2007

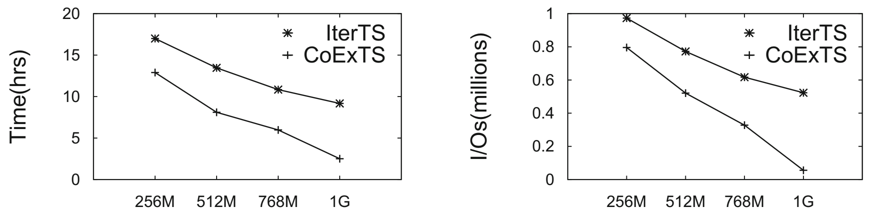
5 Performance Studies

In this section, we conduct experimental studies by comparing two external memory algorithms for topological sorting, namely, IterTS (Algorithm 2) and CoExTS (Algorithm 5). All algorithms are implemented using Visual C++ and STXXL[3] and tested on a Laptop with Intel Core i7 2.8 GHz CPU and 3.5 GB memory running Windows 7.

Dataset. In our experiments, we use a real world graph and two synthetic graphs. The real world graph is UK-2007[4], which contains the webpages and their hyperlinks information in the .UK domain. The original UK-2007 consists of 105,895,908 nodes and 3,738,733,568 edges. Since the original UK-2007 contains cycles, we collapse the cycles in UK-2007 and the generated DAG contains 105,895,908 nodes and 1,238,766,251 edges. For synthetic graphs, we generate a graph by fixing the number of its nodes first and randomly add edges that cannot form cycles in the graph. We generate two synthetic graphs Random1 and Random2. Random1 contains 100M nodes with average degree 10 and Random2 contains 100M nodes with average degree 20.

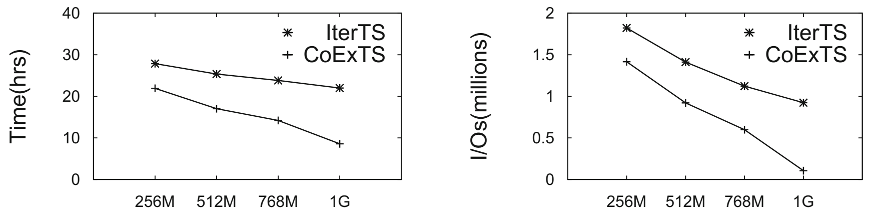
Exp-1: Performance on Real Graph UK-2007. In this experiment, we compare the running time and number of I/Os of IterTS and CoExTS on UK-2007

[3] http://stxxl.org/.
[4] http://chato.cl/webspam/datasets/uk2007/links/.



(a) Time (Vary Memory) (b) I/Os (Vary Memory)

Fig. 4. Random1



(a) Time (Vary Memory) (b) I/Os (Vary Memory)

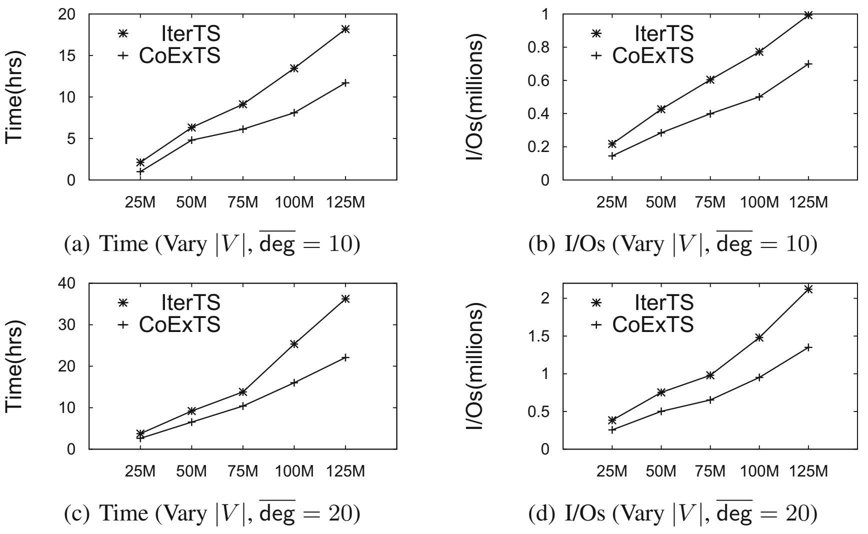
Fig. 5. Random2

when we vary the size of available memory from 256M to 1G. The results are shown in Fig. 3.

Figure 3(a) shows that (1) The running time of IterTS and CoExTS decrease as the size of available memory increases and CoExTS outperforms IterTS in all cases. This is because CoExTS adopts the contraction-expansion paradigm and uses SemiTS as a subroutine and SemiTS is very efficient for topological sorting compared with CoExTS. The performance improvement by introducing SemiTS surpasses the time consumption of graph contraction and expansion. (2) the performance gap between IterTS and CoExTS increases as the size of available memory increases. This is because as the size of available memory increases, the iterations in contraction phase and expansion phase in CoExTS decrease and we can further exploit the efficiency of SemiTS. On the other hand, IterTS does not take available memory into consideration during processing. Combining these two factors together, the performance gap increases as the size of available memory increases. For the same reasons, Fig. 3(b) shows similar trends on the number of I/Os when we vary the size of available memory.

Exp-2: Performance on Synthetic Graphs. In this experiment, we compare the running time and the number of I/Os of IterTS and CoExTS on Random1 and Random2 when we vary the size of available memory from 256M to 1G. The results are shown in Figs. 4 and 5, respectively.

For the synthetic graphs, we have similar conclusions from the performance study upon UK-2007. CoExTS also outperforms IterTS in all test cases. When available memory increases, the running time and I/Os for both IterTS and



(a) Time (Vary $|V|$, $\overline{\mathsf{deg}} = 10$) (b) I/Os (Vary $|V|$, $\overline{\mathsf{deg}} = 10$)

(c) Time (Vary $|V|$, $\overline{\mathsf{deg}} = 20$) (d) I/Os (Vary $|V|$, $\overline{\mathsf{deg}} = 20$)

Fig. 6. Scalability

CoExTS decrease. As the size of available memory increases, it is clear that the performance gap between IterTS and CoExTS increases as well. The reason is similar as that presented Exp-1.

Exp-3: Scalability. In this experiment, we evaluate the scalability of IterTS and CoExTS. To test the scalability, we vary the number of nodes from 25M to 125M in synthetic graphs and record the running time and number of I/Os of IterTS and CoExTS, respectively. In the first test, the average degree of the synthetic graphs is 10 and the results are shown in Fig. 6(a) and (b). In the second test, the average degree of the synthetic graphs is 20 and the results are shown in Fig. 6(c) and (d). The available memory in this experiment is 512M.

As shown in Fig. 6, the running time and I/Os of both two algorithms increase as $|V|$ increases. This is because, as $|V|$ increases, IterTS needs more iterations to adjust the assigned numbers for the nodes incident to the edges which are not topological sorted edges. In the meantime, the iterations in the contraction phase and expansion phase of CoExTS also increase as $|V|$ increases. CoExTS outperforms IterTS for all cases due to the introduction of contraction-expansion paradigm and the efficiency of SemiTS. From Fig. 6, it is clear that CoExTS has a better scalability than IterTS.

6 Related Work

Topological sorting is a fundamental problem in graph analysis and has been extensively studied in the literature. For the in-memory topological sorting algorithms, [9] describes an algorithm by iteratively removing the nodes with in-degree number 0. [13] proposes a depth first search based algorithm whose idea is used in the semi-external topological algorithm. The state-of-the-art external topological sorting algorithm is proposed in [2]. We introduce it in Sect. 3 and use it as our baseline in the experiments.

Several other graph algorithms focusing on I/O efficiency are proposed in the literature. [6] describes an I/O efficient algorithm for the core decomposition problem. I/O efficient algorithm for the maximal clique enumeration problem is proposed in [7]. [15] proposes an I/O efficient algorithm for the diversified top-k clique search problem in large graphs. [3] studies three I/O efficient algorithms for vertex cover. I/O efficient algorithms for the triangle enumeration problem are presented in [10]. The I/O efficient algorithm for the k-truss problem is investigated in [14]. A semi-external algorithm for the depth first search is proposed in [18]. [16] presents three I/O efficient algorithms for edge connectivity decomposition problem. [17] studies the Steiner Maximum-Connected Components search problem in semi-external memory model.

7 Conclusion

In this paper, we study the external topological sorting algorithm in large graphs. We propose a new contraction-expansion paradigm and devise an external-memory algorithm CoExTS for the topological sorting problem. The experimental results demonstrate the efficiency of our proposed algorithm.

Acknowledgements. Long Yuan is supported by Huawei YBN2017100007. Fan Zhang is supported by Huawei YBN2017100007. Lu Qin is supported by ARC DP160101513. Xuemin Lin is supported by NSFC 61672235, ARC DP170101628, DP180103096 and Huawei YBN2017100007. Wenjie Zhang is supported by ARC DP180103096 and Huawei YBN2017100007.

References

1. Aggarwal, A., Vitter, J., et al.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
2. Ajwani, D., Cosgaya-Lozano, A., Zeh, N.: A topological sorting algorithm for large graphs. J. Exp. Algorithmics **17**, Article No. 3.2 (2012)
3. Angel, E., Campigotto, R., Laforest, C.: Analysis and comparison of three algorithms for the vertex cover problem on large graphs with low memory capacities. Algorithmic Oper. Res. **6**(1), 56–67 (2011)
4. Arge, L., Revsbæk, M.: I/O-efficient contour tree simplification. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1155–1165. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10631-6_116
5. Buccafurri, F., Lax, G., Nocera, A., Ursino, D.: Moving from social networks to social internetworking scenarios: the crawling perspective. Inf. Sci. **256**, 126–137 (2014)
6. Cheng, J., Ke, Y., Chu, S., Özsu, M.T.: Efficient core decomposition in massive networks. In: Proceedings of ICDE, pp. 51–62 (2011)
7. Cheng, J., Ke, Y., Fu, A.W.-C., Yu, J.X., Zhu, L.: Finding maximal cliques in massive networks. TODS **36**(4), 21 (2011)

8. Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E.: External-memory graph algorithms. In: SODA, vol. 95, pp. 139–149 (1995)
9. Cormen, T.H.: Introduction to Algorithms. MIT Press, Cambridge (2009)
10. Hu, X., Tao, Y., Chung, C.-W.: Massive graph triangulation. In: Proceedings of SIGMOD, pp. 325–336 (2013)
11. Maheshwari, A., Zeh, N.: I/O-efficient planar separators. SIAM J. Comput. **38**(3), 767–801 (2008)
12. Raith, A., Ehrgott, M.: A comparison of solution strategies for biobjective shortest path problems. Comput. Oper. Res. **36**(4), 1299–1331 (2009)
13. Tarjan, R.E.: Edge-disjoint spanning trees and depth-first search. Acta Inform. **6**(2), 171–185 (1976)
14. Wang, J., Cheng, J.: Truss decomposition in massive networks. Proc. VLDB Endow. **5**(9), 812–823 (2012)
15. Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: Diversified top-k clique search. VLDB J. **25**(2), 171–196 (2016)
16. Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: I/O efficient ECC graph decomposition via graph reduction. PVLDB **9**(7), 516–527 (2016)
17. Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: I/O efficient ECC graph decomposition via graph reduction. VLDB J. **26**(2), 275–300 (2017)
18. Zhang, Z., Yu, J.X., Qin, L., Shang, Z.: Divide & conquer: I/O efficient depth-first search. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 445–458 (2015)