An Experimental Study of Algorithms for Packing Arborescences

Loukas Georgiadis
Department of Computer Science & Engineering, University of Ioannina, Greece

Dionysios Kefallinos
Department of Computer Science & Engineering, University of Ioannina, Greece

Anna Mpanti
Department of Computer Science & Engineering, University of Ioannina, Greece

Stavros D. Nikolopoulos
Department of Computer Science & Engineering, University of Ioannina, Greece

Abstract

A classic result of Edmonds states that the maximum number of edge-disjoint arborescences of a directed graph G, rooted at a designated vertex s, equals the minimum cardinality $c_G(s)$ of an s-cut of G. This concept is related to the *edge connectivity* $\lambda(G)$ of a strongly connected directed graph G, defined as the minimum number of edges whose deletion leaves a graph that is not strongly connected. In this paper, we address the question of how efficiently we can compute a maximum packing of edge-disjoint arborescences in practice, compared to the time required to determine the edge connectivity of a graph. To that end, we explore the design space of efficient algorithms for packing arborescences of a directed graph in practice and conduct a thorough empirical study to highlight the merits and weaknesses of each technique. In particular, we present an efficient implementation of Gabow's arborescence packing algorithm and provide a simple but efficient heuristic that significantly improves its running time in practice.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Mathematics of computing → Graph algorithms

Keywords and phrases Arborescences, Edge Connectivity, Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.14

Supplementary Material *Software (Source Code and Sample Input Instances)*:
`https://github.com/sakiskef/PackingArborescencesAlgorithms`
archived at `swh:1:dir:de7880aa38dd66bdb3661030d905a224bd23c8b9`

Funding Research supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant", Project FANTA (eFficient Algorithms for NeTwork Analysis), number HFRI-FM17-431.

Acknowledgements We would like to thank the anonymous referees for several useful comments.

1 Introduction

Let $G = (V, E)$ be a directed graph (digraph), with m edges and n vertices. Digraph G is *strongly connected* if there is a directed path from each vertex to every other vertex. Throughout the paper we let s be a fixed but arbitrary start vertex of G. If G is strongly connected, then all vertices are reachable from s and reach s. The *edge connectivity* $\lambda(G)$ of G is the minimum number of edges whose deletion leaves a digraph that is not strongly connected. Computing the edge connectivity of a graph is a classical subject in graph theory, as it is an important notion in several application areas, such as in the reliability

© Loukas Georgiadis, Dionysios Kefallinos, Anna Mpanti, and Stavros D. Nikolopoulos;
licensed under Creative Commons License CC-BY 4.0

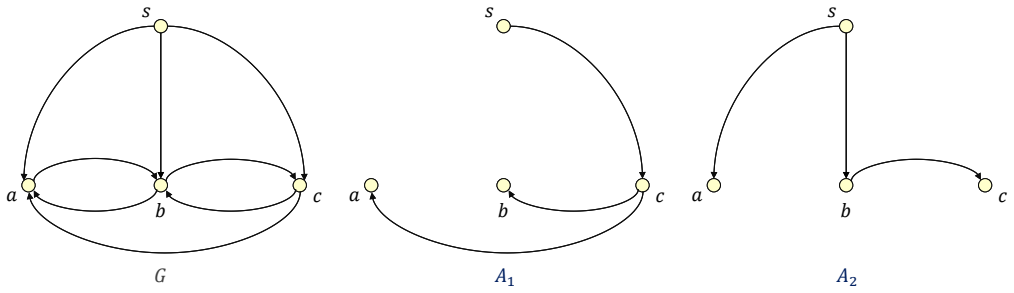

Figure 1 A directed graph G with start vertex s and minimum s-cut value $c_G(s) = 2$. Subgraphs A_1 and A_2 are two edge-disjoint s-arborescences of G.

of transportation and communication networks, and in production, scheduling, and power engineering [23]. The *reverse digraph* of G, denoted by $G^R = (V, E^R)$, is the digraph that results from G by reversing the direction of all edges. An *s-cut* is the set of edges directed from S to $V \setminus S$, where S is any vertex set that contains s such that $S \subset V$. We let $c_G(s)$ denote the minimum cardinality of an s-cut of G. Then, the edge connectivity of G is the minimum cardinality of an s-cut of G or G^R, i.e., $\lambda(G) = \min\{c_G(s), c_{G^R}(s)\}$. This observation also holds for undirected graphs, since the edge-connectivity of an undirected graph is equal to the edge-connectivity of the corresponding directed graph where each edge is oriented in both directions.

A *spanning tree* of an undirected connected graph G is a connected acyclic spanning subgraph of G. We extend this definition to directed graphs by ignoring the edge orientations. Let T be a *spanning tree* of a directed graph G rooted at s; T is an *s-arborescence* of G if s has in-degree zero and every other vertex has in-degree one. (Thus, any s-arborescence is a spanning tree of G but not vice versa.) The *arborescence packing problem* for vertex s is to construct the greatest possible number of edge-disjoint s-arborescences. See Figure 1. These concepts are useful in applications such as modeling broadcasting and evacuation [10].

Currently, the state of the art algorithm for computing the edge-connectivity λ of a digraph is the algorithm of Gabow [11] which runs in $O(\lambda m \log n^2/m)$ time. Gabow's algorithm is inspired by matroid intersection and is based on the idea of packing spanning trees. Moreover, in [11], Gabow shows how to extend his edge-connectivity algorithm so that it also computes a maximum packing of edge-disjoint s-arborescences of G in $O(k^2n^2)$ time, where $k = c_G(s)$. The edge connectivity of a simple undirected graph can be computed in $\tilde{O}(m)$ time[1], randomized [14, 17] or deterministic [15, 18]. In particular, the deterministic algorithm of Kawarabayashi and Thorup [18], as well as its improvement by Henzinger et al. [15], apply Gabow's algorithm on a contracted graph, for which the latter runs in $\tilde{O}(m)$ time.

In this paper we consider the arborescence packing problem from a practical perspective. Our starting point is the following fundamental theorem of Edmonds:

▶ **Theorem 1** (Edmonds [7]). *The maximum number of edge-disjoint s-arborescences of G equals the minimum cardinality of an s-cut.*

Edmonds gave an algorithmic proof, but the algorithm is complicated and seems to require exponential time in the worst-case [11]. Later, Lovasz [21] gave an elegant proof of Edmonds' theorem. Tarjan [25] presented an $O(k^2m^2)$-time algorithm to compute a maximum packing

[1] The notation $\tilde{O}(\cdot)$ hides poly-logarithmic factors.

of edge-disjoint s-arborescences, where $k = c_G(s)$. Schiloach [24], presented later an $O(k^2mn)$-time algorithm. The same bound was achieved by Tong and Lowler [26], who also claimed that Schiloach's algorithm is flawed. See also [9, 16] for recent interesting generalizations of Edmonds' theorem.

Currently, the best bound for the arborescence packing problem is $O(mk \log n + nk^4 \log^2 n)$, which was achieved by Bhalgat et al. [2] using the concept of edge splitting [1, 12]. Let G be a digraph with start vertex s. Define the s-v *edge-connectivity* $c_G(s, v)$, for any vertex $v \neq s$, as the cardinality of the minimum s-v cut. We say that a vertex v is *eligible* if *indegree*$(v) \geq$ *outdegree*(v). For such a vertex v, we can assume that *indegree*$(v) =$ *outdegree*(v) since we can add multiple edges from v to s without affecting the s-w edge-connectivity $c_G(s, w)$ of any $w \neq s$. Splitting off two edges (x, y) and (y, z) means deleting these two edges, and adding a new edge (x, z). This operation can be done so that the s-v edge-connectivity $c_G(s, v)$ is preserved for any $v \neq y$. Splitting off of an eligible vertex v means to split off pairs of edges incident on v so that each pair consists of an edge entering v and an edge leaving v, without affecting the connectivity of the remaining graph until v has no outgoing edge. Now, v can be removed from the graph without affecting the connectivity of the remaining graph [2]. Bhalgat et al. describe a procedure that removes any specified set S of eligible vertices while maintaining the s-v edge-connectivity of all $v \notin S$. Their algorithm extends the edge-connectivity algorithm of Gabow so that it can compute a splitting of all vertices in S. Then, it recursively computes a maximum arborescence packing of the resulting graph, which can then be used to recover a maximum arborescence packing of the original graph by putting back the vertices in S.

Here we explore the design space of efficient algorithms that compute a maximum packing of edge-disjoint arborescences. In particular, we present an efficient implementation of Gabow's arborescence packing algorithm, and provide a simple but efficient heuristic that significantly improves its running time in practice. Then, we conduct a thorough empirical study to highlight the merits and weaknesses of each technique. To the best of our knowledge, we present the first efficient implementations of algorithms for packing arborescences. Hence, we also complement the work of Georgiadis et al. [13] that studies the practical efficiency of algorithms for computing the edge-connectivity.

2 Preliminaries

Let G be a directed graph, which may have multiple edges, with a distinguished start vertex s. We denote the vertex and edge sets of G by $V(G)$ and $E(G)$, respectively. A vertex v is *reachable* in G if there is a path from s to v; v is *unreachable* if no such path exists. An *s-cut* is the set of edges directed from S to $V(G) \setminus S$, where S is any vertex set that contains s such that $S \subset V(G)$. We let $c_G(s)$ denote the minimum cardinality of an s-cut of G. Then, $c_G(s) > 0$ if and only if all vertices are reachable.

Let $S \subseteq V(G)$. The out-degree (resp., in-degree) of S, denoted by $\delta^+_G(S)$ (resp., $\delta^-_G(S)$), is the number of edges directed from S to $V(G) \setminus S$ (resp., from $V(G) \setminus S$ to S). For a vertex $v \in V(G)$, $\delta^+_G(v)$ (resp., $\delta^-_G(v)$) denotes its out-degree (resp., in-degree) in G. We let $\delta_G = \min_{v \in V(G)}\{\delta^+_G(v), \delta^-_G(v)\}$ denote the minimum degree of the graph. We let $E^+_G(v) = \{(v, w) \in E(G)\}$ (resp., $E^-_G(v) = \{(u, v) \in E(G)\}$), i.e., the set of edges directed from v (resp., to v), and refer to $E_G(v) = E^+_G(v) \cup E^-_G(v)$ as the set of edges adjacent to v. We omit the subscript G if the graph is clear from the context.

Let T be a spanning tree of G and let e be an edge that is not contained in T. The *fundamental cycle of e in T*, denoted by $C(e, T)$, is the cycle that is formed by adding e into T.

An *s-arborescence* A is a directed graph such that all vertices in $V(A)$ are reachable from s, $\delta^-_A(s) = 0$ and $\delta^-_A(v) = 1$ for all $v \in V(G) - s$. I.e., there is exactly one directed path in A from s to any other vertex. We say that A is an *s-arborescence of* G if A is a spanning subgraph of G, i.e., $V(A) = V(G)$. If $V(A) \subset V(G)$, then we say that A is a *partial s-arborescence of* G.

A *k-intersection* of G is a collection T of k spanning forests $T_1, \ldots, T_k$ of G that contains at most k edges directed to each vertex $v \in V(G) - s$, and none to s, i.e., $\delta^-_T(s) = 0$ and $\delta^-_T(v) \leq k$ for $v \in V(G) - s$. A k-intersection $T = \{T_1, \ldots, T_k\}$ is *complete* if each T_j is a spanning tree, so that $\delta^-_T(v) = k$ for all $v \neq s$. Edmonds [6] also proved the following *Matroid Characterization* of s-cuts:

▶ **Theorem 2** (Edmonds [6]). *The edges of a directed graph can be partitioned into k s-arborescences if and only if they can be partitioned into k spanning trees and every vertex except s has in-degree k.*

Referring to this result as the Matroid Characterization of minimum-cut is justified by the fact that the spanning trees with the above property are formed by the intersection of two matroids.

For a graph G and a subgraph H of G, we let $G - H$ denote the subgraph of G with vertex set $V(G)$ and edge set $E(G) - E(H)$. Also, for an edge e, we let $G \cup \{e\}$ denote the graph after adding e to G, and let $G - e$ denote the graph after deleting e from G.

3 Algorithms

In this section we provide an overview of the algorithms that we consider in our experimental study. Let G be the input digraph with n vertices, m edges, and start vertex s. Throughout this section, we let $k = c_G(s)$. We assume that all vertices are reachable (from s) in G, so $k > 0$, since otherwise there is nothing to do.

Let A be a partial s-arborescence of G. We say that A is *good* if $c_{G-A}(s) \geq k - 1$. All the algorithms we consider try to enlarge a partial s-arborescence A of G by adding one edge at a time.

We note that we can combine any packing arborescences algorithm with Gabow's edge connectivity algorithm as follows. First, we run Gabow's edge connectivity algorithm on G, and compute a complete k-intersection T of G in $O(km \log n^2/m)$ time. Then, we keep in G only the edges of $E(T)$ and run the packing arborescences algorithm on the reduced graph. (This is valid by the Matroid Characterization of s-cuts.) If the packing algorithm runs in $O(f(m, n))$ time on the original graph, then the combined algorithm runs in $O(km \log n^2/m + f(nk, n))$ total time.

3.1 The algorithm of Tarjan

Tarjan [25] computes a maximum packing of arborescences $\mathcal{A} = \{A_1, \ldots, A_k\}$ by executing k iterations of the following procedure. During the j-th iteration (for $j = 1, 2, \ldots, k$), it computes an arborescence A_j of G such that A_j is pairwise edge-disjoint with the arborescences in $\mathcal{A}^{(j-1)} = \{A_1, \ldots, A_{j-1}\}$, and $G^{(j)} = G \setminus \mathcal{A}^{(j)}$ has $c_{G^{(j)}}(s) = k - j$. (I.e., A_j is good for $G^{(j-1)}$.) To compute A_j we work as follows. We initialize a vertex set $S = \{s\}$, the set of edges $E(A_j) = \emptyset$ and mark all edges of $G^{(j-1)}$ as usable. Then, we perform the following step until $S = V(G)$. We find a usable edge $e = (u, v)$ such that $u \in S$ and $v \in V(G) \setminus S$, mark e as unusable and compute $c_{G'}(S)$, where $G' = G^{(j-1)} \setminus (A_j \cup \{e\})$. If $c_{G'}(S) \geq k - j$ then we add v to S, and add e to A_j.

To test if $c_{G'}(S) \geq k - j$, it suffices to determine if we can send at least $k - j$ units of flow from S to v (where we assign unit edge capacities). This can be done in $O(km)$ time by executing at most $k - j$ iterations of the Ford-Fulkerson method [8]. Since we perform k iterations, and in each iteration we test at most m edges, the total running time is $O(k^2m^2)$. This bound is reduced to $O(km \log n^2/m + k^4n^2) = O(k^4n^2)$ if we first compute a complete k-intersection of G by Gabow's edge connectivity algorithm, and then run Tarjan's algorithm on the reduced graph that contains only the edges of $E(T)$.

Implementation details

In our experiments, we noticed that the order in which we examine the usable edges may affect the running time of the algorithm significantly. Hence, we considered two versions of the algorithm: In the first version, we maintain the vertices of S in a stack, and examine the usable edges $e = (u, v)$ starting from the most recently added vertices $u \in S$. In the second version, we maintain S in a FIFO queue, and examine the usable edges $e = (u, v)$ starting from the least recently added vertices $u \in S$. As in turns out, in our experiments the stack version performed significantly better on most instances. (See Appendix A.)

3.2 The algorithm of Tong and Lawler

The algorithm of Tong and Lawler [26] applies a divide and conquer approach. Similarly to Tarjan's algorithm, it grows a partial arborescence A of G by trying to add one edge at a time. Initially $V(A) = \{s\}$ and $E(A) = \emptyset$. A candidate edge $e = (u, v)$, such that $u \in V(A)$ and $v \in V(G) \setminus V(A)$, is selected according to the following rule: $u \neq s$, unless there is no other candidate edge. Then, we compute $c_{G'}(s)$, where $G' = G - (A \cup \{e\})$, and consider the following two cases. (a) If $c_{G'}(s) \geq k - 1$ then the examination of e is *successful*. In this case, we add v to $V(A)$, and add e to $E(A)$. If A is now an arborescence, that is $V(A) = V(G)$, then we set $G = G - A$ and recursively compute $k - 1$ edge-disjoint arborescences in G. (b) Otherwise, we have $c_{G'}(S) = k - 2$, and the examination of e is *unsuccessful*. In this case, the algorithm of Tong and Lawler applies divide and conquer as follows. Let $(S, V(G) \setminus S)$ be a minimum s-cut of G, where $s \in S$. It is easy to observe that $e \in E(S, V(G) \setminus S)$, hence $v \in V(G) \setminus S$, and moreover, that there must be an edge $e' \in E(A)$ such that $e' \in E(S, V(G) \setminus S)$. Next, we split G into two auxiliary graphs G_1 and G_2, with corresponding partial arborescences A_1 and A_2 as follows. To construct G_1, we contract the vertices of S into s and delete all edges directed to s. Then, A_1 consists of the edges of A that were not deleted and is a partial arborescence of G_1 (but it may not be a full arborescence yet). Similarly, we construct G_2 by contracting the vertices of $V(G) \setminus S$ into v and delete self-loops. To form a corresponding partial arborescence A_2 of A in G_2, we delete all the edges of A that are directed from S to $V(G) \setminus S$ except for the first edge (x, y) on a path from S to $V(G) \setminus S$ in A. That is, x is reachable from s through a path that contains only vertices in $S \cap V(A)$. So, A_2 consists of the edges of A that were not deleted and is a partial arborescence of G_2 (but it may not be a full arborescence yet). We recursively compute k edge-disjoint arborescences $A_1^1, A_2^1, \ldots, A_k^1$ of G_1 and $A_1^2, A_2^2, \ldots, A_k^2$ of G_2, where $E(A_1^1) \subseteq E(A_1)$ and $E(A_1^2) \subseteq E(A_2)$. Then, we combine these arborescences to form k edge-disjoint arborescences of G. This combination is easy to perform because each arborescence of G_1 is edge-disjoint from exactly $k - 1$ arborescences of G_2 and vice versa. Hence, to form the desired arborescences of G, we combine each pair of non-disjoint arborescences.

As in Tarjan's algorithm, we can test if an edge e can be included in the partial arborescence A in $O(km)$ time by executing k iterations of the Ford-Fulkerson method. Since at most kn edges are added in the packing $\mathcal{A}$, the total time spent for successful edge additions is $O(k^2mn)$. On the other hand, since we can split G at most n times, there are at most n unsuccessful edge examinations. Thus, the total time spent for unsuccessful edge examinations is $O(kmn)$, which results to a total running time of $O(k^2mn)$.

Tong and Lawler also observed that the running time of their algorithm can be improved to $O(kmn + k^3n^2)$ after some preprocessing. The preprocessing phase computes a flow of value k from s to each other vertex $t \neq s$. After we have computed an s-t flow, we delete the edges entering t with zero flow. By repeating this process for all vertices $t \neq s$, after $O(kmn)$ time we are left with a subgraph H of G with $O(kn)$ edges and $c_H(s) = k$. Thus, we can compute k edge-disjoint arborescences of H in $O(k^3n^2)$ time. If we use Gabow's edge connectivity algorithm to compute a complete k-intersection of G instead of H, then we obtain an $O(km \log n^2/m + k^3n^2) = O(k^3n^2)$ time bound.

Implementation details

As in our implementation of Tarjan's algorithm, we examine candidate edges (to be included in the partial arborescence A) using a stack or a FIFO queue. As with Tarjan's algorithm, the running time of the Tong-Lawler algorithm depends on the order in which we examine candidate edges. In our experiments the stack version of the Tong-Lawler algorithm outperformed the queue version, but not consistently. (See Appendix A.) When we split G, we create two new graph instances for G_1 and G_2, and assign new vertex ids so that they are in the ranges $[1, |V(G_1)|]$ and $[1, |V(G_2)|]$ respectively. To restore the original vertex ids, we maintain mappings $h_i : V(G_i) \mapsto V(G)$, $i = 1, 2$. Moreover, for each edge in G_1 and G_2 that has exactly one endpoint in a contracted part of the graph (i.e., for each edge (x, y) such that $x \in S$ and $y \in V(G) \setminus S$, or vice versa), we associate it with the corresponding original edge of G. This information suffices to combine the arborescences in G_1 and G_2 and form the arborescences of G. Thus, after a split, we no longer need to keep the initial graph in memory.

3.3 The algorithm of Gabow

Gabow [11] presented an $O(k^2n^2)$-time algorithm to compute a maximum arborescence packing. First, it computes a complete k-intersection T of the input digraph for s. We let G be the subgraph with edges $E(T)$. Then, it repeats the following procedure, until $k = 0$:

1. Compute a complete $(k-1)$-intersection T of G.
2. Find a good s-arborescence A of G, using the algorithm described below in Section 3.3.2.
3. Decrease k by one and repeat the procedure on $G - A$.

Similarly to the algorithms of Tarjan, and of Tong and Lawler, in Step 2 Gabow's algorithm tries to enlarge a partial arborescence by adding one edge at a time. Unlike the these other algorithms, however, Gabow's algorithm does not perform flow computations, but relies on the framework of his edge-connectivity algorithm. Hence, we first provide an overview of how Gabow computes the value $k = c_s(G)$ of a minimum s-cut, together with a complete k-intersection T of G, in $O(km \log n^2/m)$ time.

3.3.1 Computing a complete k-intersection T of G

Recall that a complete k-intersection T of G is a collection of k edge-disjoint spanning trees $T_1, \ldots, T_k$, such that each vertex $v \neq s$ has in-degree k and s has in-degree zero. Gabow's algorithm computes T in k iterations, where in the k'-th iteration ($k' = 1, \ldots, k$), it begins

with a complete $(k'-1)$-intersection and tries to enlarge it so that it becomes a complete k'-intersection. To that end, it executes a *round robin* algorithm that maintains a forest $T_{k'}$ and tries to locate "joining" edges that will make $T_{k'}$ a spanning tree of G, while satisfying the invariant that $\delta^-_T(s) = 0$ and $\delta^-_T(v) \leq k'$ for all $v \neq s$. In the following, we call a vertex v *deficient* if $\delta^-_T(v) < k'$.

In more detail, during the k'-th iteration $T_{k'}$ is a forest of rooted trees, referred to as *f-trees*, where each f-tree F_z is rooted at its unique deficient vertex z. For $z \neq s$, $\delta^-_T(z) = k' - 1$. An edge $e = (x, y)$ is *joining* if x and y are in different f-trees of $T_{k'}$. The round robin algorithm looks to enlarge $T_{k'}$ by one edge at a time, and simultaneously to increase the in-degree of a deficient vertex $z \neq s$ by one. To that end, it examines an edge e_1 in $E^-(z) \setminus E(T)$. If e_1 is joining then we are done. Otherwise, it adds e_1 in some T_i and looks for a joining edge in the fundamental cycle $C(e, T_i)$. To break the cycle, we can remove from T_i an edge $e_2 \in C(e, T_i)$. Then, we can add $e_2 = (u, v)$ to some other tree of T, or replace e_2 with a edge in $E^-(v) \setminus E(T)$. This pattern continues until a joining edge is found. The sequence of edges that leads to a joining edge is called an *augmenting path*. We define this notion formally below.

An ordered pair of edges e, f is called a *swap* if $f \in C(e, T_i)$ for some $T_i \in T$. To *execute the swap* is to replace f in T_i by e. A *partial augmenting path P from z* is a sequence of distinct edges $e_1, \ldots, e_l$, such that:

1. $e_1 \in E^-(z) \setminus E(T)$.
2. For each $i < l$ either
   a. $e_{i+1} \in C(e_i, T_j)$, where T_j contains e_{i+1} but not e_i, or
   b. $e_i, e_{i+1} \in E^-(v)$, for some vertex v, where T contains e_i but not e_{i+1}.
3. Executing all swaps of P (i.e., the pairs e_i, e_{i+1} of (2a)) gives a new collection of forests.

An *augmenting path P from z* is a partial augmenting path from z whose last edge e_l is joining for z. To *augment T along P* is to execute each swap of P and add e_l to $T_{k'}$. This increases the in-degree of z by one (so z is no longer deficient), while no other in-degree changes.

Each iteration is organized as a sequence of at most $\lceil \log n \rceil$ rounds. At the start of each round all f-trees are active except F_s. Then, we repeatedly choose an active f-tree F_z and search for an augmenting path from z. If no such path is found, then the algorithm terminates and reports an s-cut of cardinality $k'-1$. Otherwise, we have found an augmenting path P from z and we augment T along P. Thus, we enlarge T by one edge and F_z is joined to another f-tree F_w. The resulting f-tree of $T_{k'}$ is rooted at w (since z is no longer deficient), and becomes inactive for the rest of the current round. Gabow showed that with an appropriate implementation, each round runs in $O(m)$ time. Furthermore, he showed that by organizing the search for augmenting paths carefully, all augmentations can be executed at the end of a round.

3.3.2 Computing a good s-arborescence

We now give an overview of how Gabow computes a good s-arborescence A of G in Step 2 of his arborescence packing algorithm. The algorithm maintains the following subgraphs of G: a partial s-arborescence A of G, a working graph $H = G - A$, and a complete $(k-1)$-intersection T for s on G. It uses the following key concept. An *enlarging path* consists of an edge $e \in E^+(A)$, and if $e \in T$, an augmenting path P for the $(k-1)$-intersection $T - e$ on $H - e$. Gabow shows that if $V(A) \neq V(G)$, then there is always an enlarging path.

The algorithm marks the edges that are known to belong in any complete $(k-1)$-intersection contained in H. It also maintains a set X of vertices such that each $u \in X$ has all edges of $E^+(u) \cap E^+(V(A))$ marked. The algorithm is divided into "periods", where each period enlarges either A or X. Initially, A contains only the start vertex s, X is empty, and all edges are unmarked. Then, we repeatedly apply the following procedure that locates an enlarging path, until it halts:

Period Step. If A is an arborescence, that is $V(A) = V(G)$ then halt. Otherwise, choose a vertex $u \in V(A) \setminus X$ and execute the *Edge Step*.

Edge Step. If all edges in $E^+(u) \cap E^+(V(A))$ are marked then add u to X and continue with the next *Period Step*. Otherwise, choose an unmarked edge $e \in E^+(u) \cap E^+(V(A))$. If $e \notin E(T)$, then add e to A and continue with the next *Period Step*.

Search Step. At this point e belongs in T. Search for an augmenting path for the $(k-1)$-intersection $T - e$ in $H - e$. If the search is successful, then use the augmenting path to enlarge A and continue with the next *Period Step*. If the search is unsuccessful then mark e and go to *Edge Step*.

The correctness of this procedure is based on the following facts. Suppose $e \in E^+(u) \cap E^+(V(A))$ has no enlarging path, i.e., the *Search Step* was unsuccessful. Let L be the set of edges labelled in the search, and let e' be any edge in $E^+(u)$. Then, e belongs to any complete $(k-1)$-intersection contained in H, and no edge of L is in an enlarging path for e', with respect to the current A and T. This implies that for any $v \in X$, no edge of $E^+(v)$ has an enlarging path.

To make the search for enlarging paths fast, each unsuccessful search in a period contracts the edges in L that become labelled during an unsuccessful search for an edge $e \in E^+(u)$. The contraction is valid since, for each tree $T_i \in T$, $i = 1, 2 \ldots, k-1$, the edges in $L \cap (T_i - e)$ form a tree. Contracting $V(L)$ into a single vertex v results in a graph H' that has a complete $(k-1)$-intersection that contains all edges in $E^-_{H'}(v)$. Then, for any edges $e' \in E^+_{H'}(v)$, graphs H and H' have the same enlarging paths for e', and unsuccessful searches in $H - e'$ and $H - e$ label the same edges not in L.

To implement the above procedure efficiently, Gabow's algorithm performs the contractions implicitly. To that end, it maintains a partition of $V(G)$ into disjoint sets $S_1, \ldots, S_l$, such that each set S_j induces a tree in each $T_i \in T$. Each vertex is labelled with the name of the set that contains it and also, for each $i = 1, \ldots, k-1$, each set S_j is labelled with its root vertex in T_i. The *Search Step* for an edge $e = (u, w) \in T_i$ removes e from T and H, and searches for an augmenting path P from w. During this search, when a vertex v is reached, if $v \in S_j$ then the search continues from the root of S_j in T_i. If the search is successful, then we augment along P. Otherwise, when the search is unsuccessful, we add e back to H and T, and update the vertex partition $\{S_j\}$ by merging together all sets S_j that contain an end of an edge that was labelled during the search.

Gabow shows that this algorithm constructs an s-arborescence A in $O(kn^2)$ time; there are at most kn searches (at most one per edge), and at most $2n$ periods. The latter follows from the fact that a period enlarges A or X, and each set can be enlarged less than n times. Moreover, each period can be implemented to run in $O(kn)$ time. Since we compute k arborescences, the total running time is $O(k^2n^2)$.

Practical speedup

In order to speedup Gabow's algorithm in practice, we implemented the following simple heuristic. Let G be the current graph. We compute an s-arborescence A of G, e.g., by executing a depth-first search (DFS) from s, and test if A is good. To do that, it suffices to

Table 1 An overview of the algorithms considered in our experimental study. The bounds refer to a digraph G with minimum s-cut value $k = c_s(G)$, n vertices and m edges; $m \leq kn$ if G contains only the edges of a complete k-intersection.

Algorithm	Technique	Complexity	Ref.
Tarjan (Tar)	Test if a usable edge can be added via max-flow	$O(k^2m^2)$	[25]
	Run on a complete k-intersection	$O(k^4n^2)$	
Tong-Lawler (TL)	Graph splitting via min-cut	$O(k^2mn)$	[26]
	Run on a complete k-intersection	$O(k^3n^2)$	
Gabow (Gab)	Compute complete k'-intersections ($k' \leq k$) and enlarging paths	$O(k^2n^2)$	[11]

test if $c_{G-T}(s) = k - 1$. If this is the case, then we can keep A in the packing. Otherwise, we simply discard A, and compute a good s-arborescence of G using Gabow's algorithm. In either case, after we have computed a good s-arborescence A of G, we decrease k by one and repeat the procedure on $G - A$.

Despite its simplicity, the above modification provides significant speedups in practice, as the experimental results of Section 4 suggest. Moreover, we can immediately observe that the overall $O(k^2n^2)$ running time still holds for this variant of Gabow's algorithm.

4 Empirical Analysis

We implemented our algorithms in `C++`, using `g++ 7.5.0` with full optimization (flag -O4) to compile the code. The reported running times were measured on a GNU/Linux machine, with Ubuntu (18.04.6 LTS): a Dell Precision Tower 7820 server 64-bit NUMA machine with an Intel(R) Xeon(R) Gold 5220R processor and 192GB of RAM memory. The processor has 24.75MB of cache memory and 18 cores. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `high_resolution_clock` class of the standard library `chrono`, averaged over ten different runs.

Table 1 gives an overview of the algorithms we consider in our experimental study. We did not include the algorithm of Bhalgat et al. because some important details are omitted from the extended abstract of [2].[2]

We base our implementation of Gabow's arborescence packing algorithm on efficient implementations of Gabow's edge connectivity algorithm presented in [13]. Also, for the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), we report the running time of their stack-based implementations. We compare the stack-based against the queue-based implementations in Appendix A. For the experimental evaluation, we considered two types of graphs: (i) real-world directed graphs, augmented with additional edges in order to increase their edge-connectivity, and (ii) k-cores of undirected graphs.

Augmented graphs

In our first experiment, we consider how the running time of each algorithm is affected as the minimum s-cut value $k = c_s(G)$ increases. To that end, we augment some real-word graphs as follows. Let G be an input strongly connected digraph. For a given parameter

[2] We are unaware of a full version of [2].

β, we create an augmented instance G_β of G by executing the following procedure. We go through the vertices of G and, for each vertex v, we add $\beta - \delta^-(v)$ edges directed to v if $\delta^-(v) < \beta$, where each added edge originates from a randomly chosen vertex. Then, we make a second pass over the vertices and, for each vertex v, we add $\beta - \delta^+(v)$ edges directed away from v if $\delta^+(v) < \beta$, where each added edge is directed to a randomly chosen vertex. Notice that the resulting graph has minimum degree $\delta \geq \beta$.

Table 2 reports the characteristics of the augmented graphs G_β produced by the above procedure for $\beta \in \{2, 4, 8, 16\}$. Here, we also give the number of edges (m') in a complete k intersection T of G (with respect to the start vertex s). In Table 3 we report the corresponding running times of each algorithm. The execution of an algorithm was terminated if it exceeded one hour. We also report the running times of two versions of Gabow's algorithm that computes a complete k intersection T of G: the standard version (Gab-EC), and a version that uses DFS to do a fast initialization of the forest $T_{k'}$ at the beginning of the k'-th iteration (Gab-EC-DFS). Both implementations are taken from [13]. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report both the running time when the input is the original graph G (above) and a complete k intersection of G (below). In the latter case, the algorithms receive a complete k intersection T of G as input, and we do not account for the time required to compute T.

Table 2 Characteristics of augmented graphs, resulting from some real-world graphs after inserting some edges; n is the number of vertices, m the number of edges; δ denotes the minimum vertex (in or out) degree, and $c_s(G)$ denotes the cardinality of the minimum s-cut; m' is the number of edges in a complete k intersection of G (for the start vertex s).

Graph	n	m	δ	$c_s(G)$	m'	type and source
enron-EC2	8271	151651	2	2	8270	email network [19]
enron-EC4	8271	162999	4	4	24810	
enron-EC8	8271	190618	8	8	57890	
enron-EC16	8271	253345	16	16	124050	
p2p-Gnutella25-EC2	5152	19765	2	2	10302	peer2peer network [19]
p2p-Gnutella25-EC4	5152	27565	4	4	20604	
p2p-Gnutella25-EC8	5152	50076	8	8	41208	
p2p-Gnutella25-EC16	5152	100669	16	16	82416	
rome99-EC2	3352	9869	2	2	6702	road network [5]
rome99-EC4	3352	15468	4	4	13404	
rome99-EC8	3352	31952	8	8	26808	
rome99-EC16	3352	65542	16	16	53616	
s38584-EC2	16310	42128	2	2	32618	VLSI circuit [4]
s38584-EC4	16310	80250	4	4	65236	
s38584-EC8	16310	160297	8	8	130472	
s38584-EC16	16310	323963	16	16	260944	
web-Stanford-EC2	150475	2334929	2	2	300948	web graph [19]
web-Stanford-EC4	150475	3307506	4	4	601896	
web-Stanford-EC8	150475	2379878	8	8	1203792	
web-Stanford-EC16	150475	3643794	16	16	2407584	

From the results, we observe that TL has overall the worst performance, even compared to Tar despite the inferior upper bound of the latter. Indeed, on average Tar runs twice as fast compared to TL. This is due to the overhead incurred in TL for splitting a graph G into

Table 3 Running times in seconds of the algorithms for the augmented graphs of Table 2. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Gab-EC	Gab-EC-DFS	Tar	TL	Gab	Gab-DFS
enron-EC2	0.01	0.01	0.01 0.01	1.09 0.04	0.15	0.01
enron-EC4	0.02	0.01	36.34 7.43	118.17 7.59	9.81	0.05
enron-EC8	0.06	0.01	304.29 95.58	614.18 131.91	38.42	0.28
enron-EC16	0.17	0.02	1840.76 951.43	2782.78 1319.77	130.58	1.50
p2p-Gnutella25-EC2	0.01	0.01	1.19 0.79	3.90 1.12	1.39	0.02
p2p-Gnutella25-EC4	0.02	0.01	8.46 7.05	23.76 14.34	6.57	0.04
p2p-Gnutella25-EC8	0.04	0.01	55.21 53.41	124.36 91.05	21.63	0.17
p2p-Gnutella25-EC16	0.09	0.01	403.92 450.80	760.74 629.79	61.32	0.80
rome99-EC2	0.01	0.01	0.45 0.35	0.37 0.24	0.36	0.34
rome99-EC4	0.02	0.01	3.20 3.19	7.26 5.56	2.63	0.02
rome99-EC8	0.03	0.01	21.67 21.62	50.86 37.24	8.65	0.10
rome99-EC16	0.06	0.01	160.01 156.29	312.30 241.95	25.32	0.49
s38584-EC2	0.03	0.01	14.12 8.88	37.02 23.45	14.73	12.76
s38584-EC4	0.06	0.01	131.86 117.79	262.07 98.85	75.42	0.14
s38584-EC8	0.14	0.02	1012.85 878.79	2007.45 1258.60	245.01	0.68
s38584-EC16	0.34	0.04	>1h >1h	>1h >1h	717.20	3.29
web-Stanford-EC2	0.60	0.29	>1h 2307.26	>1h 2350.42	520.22	1.39
web-Stanford-EC4	1.45	0.69	>1h >1h	>1h >1h	>1h	5.40
web-Stanford-EC8	3.26	1.03	>1h >1h	>1h >1h	>1h	14.32
web-Stanford-EC16	7.73	3.28	>1h >1h	>1h >1h	>1h	70.98

two auxiliary graphs G_1 and G_2, and manipulating mappings from the vertex ids of G_1 and G_2 to those in G. Furthermore, we observe that TL runs consistently faster on the complete k intersection T of G compared to the original graph G. While this is expected since T has fewer edges, on the other hand we note that it may be easier to find good candidate edges to augment a partial arborescence if the graph contains some additional edges. The same

observation holds for Tar as well, but here we see that in one instance (p2p-Gnutella25-EC16) the algorithm runs faster on G rather than on T. Moreover, we note that the executions of TL and Tar on T outperform Gab in some instances.

Next, we turn to the algorithms of Gabow. First, we verify that the edge-connectivity algorithms Gab-EC and Gab-EC-DFS are very effective. Regarding the arborescence packing algorithms, we first note that Tar and TL perform close to Gab when the edge-connectivity $c_G(s)$ is small (EC2 instances), but quickly become uncompetitive when the edge-connectivity increases. Overall, in our experiment, Gab was 50% faster than Tar on average. Finally, we note that Gab-DFS is faster than Gab by two orders of magnitude on most instances. This is due to the fact that our simple heuristic very often manages to construct a good arborescence by a simple DFS traversal.

k-cores

A k-core of an undirected graph G is a maximal subgraph of G such that $\delta(v) \geq k$ for all $v \in V(H)$. This concept is useful in the analysis of social networks [3] as well as in several other applications [22]. In this experiment, we use the k-core decomposition algorithm of the SNAP software library and tools [20], and use subgraphs of this decomposition as inputs, for various values of k. We transform each such undirected graph to a directed graph by orienting each edge in both directions. Table 4 reports the characteristics of the resulting graphs. In Table 5 we report the corresponding running times of each algorithm. Again, we terminated the execution of an algorithm if that exceeded one hour.

Here too, we observe that Tar outperforms TL on most instances, but unlike the augmented graphs, their difference is marginal. Both TL and Tar run consistently faster on the complete k intersection T of G compared to the original graph G. Again, the executions of TL and Tar on T outperform Gab in some instances, but overall Gab is 50% faster. Also, our heuristic was very effective in this experiment as well, since Gab-DFS ran faster than Gab by two orders of magnitude.

Table 4 Characteristics of k-core graphs, extracted from real-world graphs in [19]; n is the number of vertices, m the number of edges; δ denotes the minimum vertex degree, and $c_s(G)$ denotes the cardinality of the minimum s-cut (which equal the edge-connectivity since the graphs are undirected); m' is the number of edges in a complete k intersection.

Graph	n	m	δ	$c_s(G)$	m'	type and source
facebook_combined-core02	3964	176318	2	2	7926	social circles from facebook
facebook_combined-core04	3754	175332	4	4	11258	
facebook_combined-core25	1366	118810	25	25	6824	
facebook_combined-core50	616	75246	50	30	19064	
Email-Enron-core09	5088	206472	9	9	45783	email network
Email-Enron-core10	4513	196594	10	10	45120	
Email-Enron-core16	2873	157506	16	16	45952	
Email-Enron-core18	2561	147332	18	18	46080	
CA-AstroPh-core18	5049	244004	18	18	90864	collaboration network
CA-AstroPh-core25	3202	175520	4	4	12804	
CA-AstroPh-core29	2441	139070	29	2	4880	
CA-AstroPh-core32	1926	112830	32	32	61600	
Gowalla_edges-core11	22742	851196	11	6	136443	social network
Gowalla_edges-core12	19938	791666	12	5	99684	
Gowalla_edges-core15	13833	639244	15	4	55328	
Gowalla_edges-core20	8161	456014	20	8	65280	

Table 5 Running times in seconds of the algorithms for the k-core graphs of Table 4. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Gab-EC	Gab-EC-DFS	Tar	TL	Gab	Gab-DFS
facebook_combined-core02	0.01	0.01	2.47 0.22	8.71 0.43	2.00	0.01
facebook_combined-core04	0.01	0.01	6.88 0.29	20.32 0.86	2.88	0.02
facebook_combined-core25	0.01	0.01	3.97 0.29	6.82 0.53	0.95	0.02
facebook_combined-core50	0.04	0.01	43.41 29.38	47.76 22.59	4.59	0.55
Email-Enron-core09	0.03	0.01	109.84 44.41	156.29 55.41	30.06	0.17
Email-Enron-core10	0.03	0.01	107.00 47.09	101.79 54.95	28.60	0.17
Email-Enron-core16	0.04	0.01	136.26 106.58	170.74 74.56	20.65	0.35
Email-Enron-core18	0.04	0.01	151.68 119.37	177.93 77.12	19.80	0.42
CA-AstroPh-core18	0.25	0.01	894.66 716.73	1029.08 412.42	63.18	1.86
CA-AstroPh-core25	0.02	0.01	6.74 1.83	16.29 1.16	2.03	0.04
CA-AstroPh-core29	0.01	0.01	0.72 0.08	2.68 0.21	0.53	0.01
CA-AstroPh-core32	0.15	0.01	556.99 537.45	545.93 270.58	28.09	1.61
Gowalla_edges-core11	0.10	0.06	2086.57 567.03	2559.07 394.37	404.29	0.49
Gowalla_edges-core12	0.07	0.04	915.08 80.10	1094.83 166.56	219.33	0.31
Gowalla_edges-core15	0.04	0.02	197.84 28.64	321.91 49.67	71.65	0.15
Gowalla_edges-core20	0.05	0.02	327.64 118.93	412.93 109.56	68.57	0.32

References

1 András A Benczúr and David R Karger. Augmenting undirected edge connectivity in Õ(n2) time. *Journal of Algorithms*, 37(1):2–36, 2000. doi:10.1006/jagm.2000.1093.

2 Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Fast edge splitting and edmonds' arborescence construction for unweighted graphs. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 455–464, USA, 2008. Society for Industrial and Applied Mathematics.

3 Kshipra Bhawalkar, Jon Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. Preventing unraveling in social networks: The anchored k-core problem. *SIAM Journal on Discrete Mathematics*, 29(3):1452–1475, 2015.

4 CAD Benchmarking Lab. ISCAS'89 benchmark information. http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html.

5 C. Demetrescu, A.V. Goldberg, and D.S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9/, 2007.

6 J. Edmonds. Submodular functions, matroids, and certain polyhedra. *Combinatorial Structures and their Applications*, pages 69–81, 1970.

7 J. Edmonds. Edge-disjoint branchings. *Combinatorial Algorithms*, pages 91–96, 1972.

8 D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, USA, 2010.

9 S. Fujishige. A note on disjoint arborescences. *Combinatorica*, 30(2):247–252, 2010. doi:10.1007/s00493-010-2518-y.

10 Satoru Fujishige and Naoyuki Kamiyama. The root location problem for arc-disjoint arborescences. *Discrete Applied Mathematics*, 160(13):1964–1970, 2012. doi:10.1016/j.dam.2012.04.013.

11 H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50:259–273, 1995.

12 Harold N. Gabow. Efficient splitting off algorithms for graphs. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 696–705, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/195058.195436.

13 Loukas Georgiadis, Dionysios Kefallinos, Luigi Laura, and Nikos Parotsidis. An experimental study of algorithms for computing the edge connectivity of a directed graph. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 85–97. SIAM, 2021. doi:10.1137/1.9781611976472.7.

14 M. Ghaffari, K. Nowicki, and M. Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, pages 1260–1279, USA, 2020. Society for Industrial and Applied Mathematics.

15 M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020.

16 N. Kamiyama, N. Katoh, and A. Takizawa. Arc-disjoint in-trees in directed graphs. *Combinatorica*, 29:197–214, 2009. doi:10.1007/s00493-009-2428-z.

17 D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, January 2000. doi:10.1145/331605.331608.

18 K.-I. Kawarabayashi and M. Thorup. Deterministic edge connectivity in near-linear time. *Journal of the ACM*, 66(1), December 2018. doi:10.1145/3274663.

19 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

20 Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

21 László Lovász. On two minimax theorems in graph. *Journal of Combinatorial Theory, Series B*, 21(2):96–103, 1976. doi:10.1016/0095-8956(76)90049-6.

22 F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis. The core decomposition of networks: theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020. doi:10.1007/s00778-019-00587-4.

23 H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, 2008. 1st edition.

24 Yossi Shiloach. Edge-disjoint branching in directed multigraphs. *Information Processing Letters*, 8(1):24–27, 1979. doi:10.1016/0020-0190(79)90086-3.

25 Robert Endre Tarjan. A good algorithm for edge-disjoint branching. *Information Processing Letters*, 3(2):51–53, 1974. doi:10.1016/0020-0190(74)90024-6.

26 Po Tong and E.L. Lawler. A faster algorithm for finding edge-disjoint branchings. *Information Processing Letters*, 17(2):73–76, 1983. doi:10.1016/0020-0190(83)90073-X.

A Stack-based vs queue-based implementations

Tables 6 and 4 compare the stack-based against the queue-based implementation of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for augmented and k-core graphs, respectively.

Table 6 Running times in seconds of the stack-based and queue-based implementations of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for the augmented graphs of Table 2. We report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Tar		TL	
	stack	queue	stack	queue
enron-EC2	0.01	0.01	1.09	2.41
	0.01	0.01	0.04	0.48
enron-EC4	36.34	111.44	118.17	131.24
	7.23	8.18	7.59	6.73
enron-EC8	304.29	591.30	614.18	786.56
	95.58	136.25	131.91	118.14
enron-EC16	1840.76	2662.05	2782.78	3340.18
	951.43	1298.16	1319.77	1343.09
p2p-Gnutella25-EC2	1.19	3.66	3.90	4.41
	0.79	1.13	1.12	1.59
p2p-Gnutella25-EC4	8.46	22.00	23.76	23.61
	7.05	14.67	14.34	14.73
p2p-Gnutella25-EC8	55.21	123.56	124.36	119.05
	53.41	99.95	91.005	88.94
p2p-Gnutella25-EC16	403.92	744.87	760.74	768.10
	450.80	662.15	629.79	620.41
rome99-EC2	0.45	0.40	0.37	0.45
	0.35	0.32	0.24	0.21
rome99-EC4	3.20	8.12	7.26	6.98
	3.19	6.08	5.56	3.88
rome99-EC8	21.67	49.48	50.86	47.44
	21.62	37.11	37.24	35.71
rome99-EC16	160.01	306.58	312.30	320.36
	156.29	239.31	241.95	243.20
s38584-EC2	14.12	39.62	37.02	44.71
	8.88	25.96	23.45	26.93
s38584-EC4	131.86	286.66	262.07	282.07
	117.79	102.47	98.85	114.64
s38584-EC8	1012.85	2040.86	2007.45	1867.42
	878.22	1541.89	1258.60	1265.24
s38584-EC16	>1h	>1h	>1h	>1h
	>1h	>1h	>1h	>1h
web-Stanford-EC2	>1h	>1h	>1h	>1h
	2307.26	3100.36	2350.42	>1h
web-Stanford-EC4	>1h	>1h	>1h	>1h
	>1h	>1h	>1h	>1h

Table 7 Running times in seconds of the stack-based and queue-based implementations of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for the k-core graphs of Table 4. We report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Tar		TL	
	stack	queue	stack	queue
facebook_combined-core02	2.47	8.04	8.71	6.63
	0.22	0.37	0.43	0.36
facebook_combined-core04	6.88	18.83	20.32	11.51
	0.29	0.83	0.86	0.95
facebook_combined-core25	3.97	6.32	6.82	3.97
	0.29	0.51	0.53	0.31
facebook_combined-core50	43.41	44.34	47.76	8.54
	29.38	21.88	22.59	3.11
Email-Enron-core09	109.84	146.24	156.29	221.64
	44.41	55.64	55.41	47.98
Email-Enron-core10	107.00	94.40	101.79	116.07
	47.09	53.62	54.95	41.25
Email-Enron-core16	136.26	161.54	170.74	226.42
	106.58	90.43	74.56	94.21
Email-Enron-core18	151.68	168.80	177.93	240.51
	119.37	90.58	77.12	104.76
CA-AstroPh-core18	894.66	991.31	1029.08	619.59
	716.73	99.58	412.42	213.50
CA-AstroPh-core25	6.74	15.42	16.29	19.77
	1.83	1.66	1.16	0.75
CA-AstroPh-core29	0.72	2.50	2.68	3.02
	0.08	0.19	0.21	0.10
CA-AstroPh-core32	556.99	526.15	545.93	991.77
	537.45	427.66	270.58	349.73
Gowalla_edges-core11	2086.57	2468.99	2559.07	>1h
	567.03	455.23	394.37	403.34
Gowalla_edges-core12	915.08	1425.66	1094.83	2482.81
	80.10	187.11	166.56	169.45
Gowalla_edges-core15	197.84	498.99	321.91	780.18
	28.64	53.81	49.67	47.25
Gowalla_edges-core20	327.64	552.16	412.93	847.61
	118.93	122.30	109.56	105.48