Computing a Feedback Arc Set Using PageRank

Vasileios Geladaris[(✉)], Panagiotis Lionakis, and Ioannis G. Tollis

Computer Science Department, University of Crete, Heraklion, Greece
{csd3926,lionakis,tollis}@csd.uoc.gr

Abstract. We present a new heuristic algorithm for computing a minimum Feedback Arc Set in directed graphs. The new technique produces solutions that are better than the ones produced by the best previously known heuristics, often reducing the FAS size by more than 50%. It is based on computing the PageRank score of the nodes of the directed line graph of the input directed graph. Although the time required by our heuristic is heavily influenced by the size of the produced line graph, our experimental results show that it runs very fast even for very large graphs used in graph drawing.

Keywords: Feedback arc set · Hierarchical graph drawing · PageRank · Line graph

1 Introduction

In a directed graph, G, a *feedback arc set* (*FAS*) is a set of edges whose removal leave G acyclic. The minimum FAS problem is important for visualizing directed graphs in hierarchical style [7]. In fact, the first step of both known frameworks for hierarchical graph drawing is to compute a minimum FAS [13,18]. Unfortunately, computing a minimum FAS is NP-hard and thus many heuristics have been presented in order to find a reasonably good solution. In this paper we present a new heuristic that uses a different approach and produces FAS that contain about half the number of edges of the best known heuristics. However, it requires superlinear time, and hence it may not be suitable for very large graphs. Finding a minimum FAS has many additional applications beyond Graph Drawing, including misinformation removal, label propagation, and many application domains motivated by Social Network Analysis [6,9,16].

A feedback arc set of a directed graph $G = (V, E)$ is a subset of edges F of E such that removing the edges in F from E leaves G acyclic (no directed cycles). In other words, a FAS contains at least one edge from each cycle of G. In hierarchical drawing algorithms the edges in a FAS are not removed, but instead their direction is inverted. Following the terminology of [7], a set of edges whose reversal makes the digraph acyclic is called a feedback set (FS). Notice that a FAS is not always a FS. However, it is easy to see that every minimal cardinality

FAS is also a FS. Hence it follows that the minimum FS problem is as hard as the well studied minimum FAS problem which is known to be NP-hard [10,11]. Clearly, any heuristic for solving the minimum FAS problem can be applied for solving the minimum FS problem, as discussed in [7,12].

There have been many heuristics for solving the FAS problem due to the multitude of its applications. Two of the most important heuristics/techniques are due to Eades, Lin & Smyth [8] and Brandenburg & Hanauer [4]. The first is a greedy heuristic, that will be called *GreedyFAS*, whereas the second presents a set of heuristics based on sorting. Simpson, Srinivasan & Thomo published an experimental study for the FAS problem on very large graphs at web-scale (also called *webgraphs*) [17]. They implemented and compared many FAS heuristics. According to their study, the aforementioned are the most efficient heuristics, but only GreedyFAS is suitable to run on their extra large webgraphs.

In this paper we present a new heuristic algorithm for computing a minimum FAS in directed graphs. The new technique produces solutions that are better than the ones produced by the best previous heuristics, sometimes even reducing the FAS size by more than 50%. It is based on computing the PageRank score of the nodes of a graph related to the input graph, and runs rather fast for graphs up to 4,000 nodes. However, it is slower than GreedyFAS for webgraphs.

2   Existing Algorithms

In this section we summarize and give a brief description of two important heuristics that currently give the best results for the FAS problem, according to the new experimental study of Simpson, Srinivasan & Thomo [17]. They implemented and compared many heuristics for FAS, and performed experiments on several large and very large webgraphs. Their results show that two of the known heuristic algorithms give the best results.

The first of the two heuristic algorithms that currently produce the best FAS size is called *GreedyFAS* and it is due to Eades, Lin & Smyth [8]. In [17] two different optimized implementations of GreedyFAS that run in $O(n+m)$ are presented and tested. These are the most efficient implementations in their study and are able to run even for their extra large webgraphs. The second algorithm is *SortFAS* of Brandenburg & Hanauer [4]. According to [17], SortFAS, as proposed runs in $O(n^3)$ time but Simpson et al. present an implementation that runs in $O(n^2)$ time.

We will present experimental results that show that our new heuristic algorithm performs better than both of them in terms of the size of the produced FAS. On the other hand, it takes more time than both of them for large graphs. However, for graphs that are typically used for visualization purposes, the running time is acceptable whereas the produced FAS size is about half.

2.1   GreedyFAS

The GreedyFAS algorithm was introduced by Eades, Lin & Smyth in 1993 [8]. It efficiently calculates an approximation to the FAS problem on a graph G. In order to understand the algorithm, we first discuss the *Linear Arrangement*

Problem (LA), which is an equivalent formulation to the FAS problem. The LA problem produces an ordering of the nodes of a graph G for which the number of arcs pointing backwards is minimum. The set of backwards arcs is a FAS since removing them from G leaves the graph acyclic.

GreedyFAS calculates a feedback arc set of a graph G by first calculating a Linear Arrangement of G. More specifically, in each iteration, the algorithm removes all nodes of G that are sinks followed by all the nodes that are sources. It then removes a node u for which $\delta(u) = d^+(u) - d^-(u)$ is a maximum, where $d^+(u)$ denotes the out-degree of u and $d^-(u)$ denotes the in-degree of u. The algorithm also makes use of two sequences of nodes s_1 and s_2. When any node u is removed from G then it is either prepended to s_2 if it's a sink, or appended to s_1 if it's not. The above steps are repeated until G is left with no nodes, then the sequence $s = s_1 s_2$ is returned as a linear arrangement for which the backward arcs make up a feedback arc set. For more details see [7,12]. Using the implementations of [17], GreedyFAS runs very fast, in $O(n + m)$ time, and is suitable for their extra large webgraphs. The pseudocode for GreedyFAS, as described in [7] and [17], is presented in Algorithm 1.

Algorithm 1. GreedyFAS

> **Input:** Directed graph $G = (V, E)$
> **Output:** Linear Arrangement A
> $s_1 \leftarrow \emptyset$, $s_2 \leftarrow \emptyset$
> **while** $G \neq \emptyset$ **do**
> > **while** G contains a sink **do**
> > > choose a sink u
> > > $s_2 \leftarrow u s_2$
> > > $G \leftarrow G \backslash u$
> > **while** G contains a source **do**
> > > choose a source u
> > > $s_1 \leftarrow s_1 u$
> > > $G \leftarrow G \backslash u$
> > choose a node u for which $\delta(u)$ is a maximum
> > $s_1 \leftarrow s_1 u$
> > $G \leftarrow G \backslash u$
> **return** $s = s_1 s_2$

2.2   SortFAS

The SortFAS algorithm was introduced in 2011 by Brandenburg & Hanauer [4]. The algorithm is an extension of the KwikSortFAS heuristic by Ailon et al. [1], which is an approximation algorithm for the FAS problem on tournaments. With SortFAS, Brandenburg & Hanauer extended the above heuristic to work for general directed graphs. It uses the underlying idea that the nodes of a graph can be sorted into a desirable Linear Arrangement based on the number of back arcs induced.

In the case of SortFAS, the nodes are processed in order of their ordering $(v_1...v_n)$. The algorithm goes through n iterations. In the i-th iteration, node v_i is inserted into the linear arrangement in the best position based on the first $i-1$ nodes which are already placed. The best position is the one with the least number of back arcs induced by v_i. In case of a tie the leftmost position is taken. Using the implementation of [17], SortFAS runs in $O(n^2)$ time. The pseudocode for SortFAS, as described in [17], is presented in Algorithm 2.

Algorithm 2. SortFAS

Input: Linear arrangement A
for each node v in A **do**
    $val \leftarrow 0$, $min \leftarrow 0$, $loc \leftarrow$ position of v
    for each position j from $loc - 1$ down to $-$ **do**
        $w \leftarrow$ node at position j
        if arc (v, w) exists **then**
            $val \leftarrow val - 1$
        else if arc (w, v) exists **then**
            $val \leftarrow val + 1$
        if $val \leq min$ **then**
            $min \leftarrow val, loc \leftarrow j$
    insert v at position loc

---

3 Our Proposed Approach

Our approach is based on running the well known PageRank algorithm [5,14] on the directed line digraph of the original directed graph. The *line graph* of an undirected graph G is another graph $L(G)$ that is constructed as follows: each edge in G corresponds to a node in $L(G)$ and for every two edges in G that are adjacent to a node v an edge is placed in $L(G)$ between the corresponding nodes. Clearly, the number of nodes of a line graph is m and the number of edges is proportional to the sum of squares of the degrees of the nodes in G, see [15]. If G is a directed graph, its *directed line graph* (or *line digraph*) $L(G)$ has one node for each edge of G. Two nodes representing directed edges incident upon v in G (one incoming into v, and one outgoing from v), called $L(u, v)$, and $L(v, w)$, are connected by a directed edge from $L(u, v)$ to $L(v, w)$ in $L(G)$. In other words, every edge in $L(G)$ represents a directed path in G of length two. Similarly, the number of nodes of a line digraph is m and the number of edges is proportional to $\sum_{u \in V}[d^+(u) \times d^-(u)]$. Hence, the size of $L(G)$ is $O(m + \sum_{u \in V}[d^+(u) \times d^-(u)])$.

Given a digraph $G = (V, E)$ our approach is to compute its line digraph, $L(G)$, run a number of iterations of PageRank on $L(G)$ and remove the node of highest PageRank in $L(G)$. Our experimental results indicate that PageRank values converge reasonably well within five iterations.

A digraph G is *strongly connected* if for every pair of vertices of G there is a cycle that contains them. If G is not strongly connected, it can be decomposed

into its *strongly connected components (SCC)* in linear time [19]. An SCC of G is a subgraph that is strongly connected, and is maximal, in the sense that no additional edges or vertices of G can be included in the subgraph without breaking its property of being strongly connected. If each SCC is contracted to a single vertex, the resulting graph is a directed acyclic graph (DAG). It follows that feedback arcs can exist only within some (SCC) of G. Hence we can apply this approach inside each SCC, using their corresponding line digraph, and remove the appropriate edges from each SCC. This approach will avoid performing several useless computations and thus reduce the running time of the algorithm.

3.1   Line Graph

In order to obtain the line digraph of G, we use a DFS-based approach. First, for each edge (u, v) of G, we create a node (u, v) in $L(G)$ and then run the following recursive procedure. For a node v, we mark it as visited and iterate through each one of its outgoing edges. For each outgoing edge (v, u) of v, we add an edge in $L(G)$ from the *prev* $L(G)$ node that was processed before the procedure's call to the node (v, u). Afterwards we call the same procedure for u if it's not visited with (v, u) as *prev*. If u is visited we add an edge from (v, u) to each one of $L(G)$'s nodes corresponding from u. Since this technique is based on DFS, the running time is $O(n + m + |L(G)|)$. The pseudocode for computing a line digraph is presented in Algorithm 3.

Algorithm 3. LineDigraph

Input: Digraph $G = (V, E)$
Output: Line Digraph $L(G)$ of G
Create a line digraph $L(G)$ with every edge of G as a node
$v \leftarrow$ random node of G

procedure GETLINEGRAPH$(G, L(G), v, prev)$
    mark v as *visited*
    for each edge $e = (v, u)$ outgoing of v **do**
        $z \leftarrow$ node of $L(G)$ representing e
        create an edge in $L(G)$ from *prev* to z ▷ Given that *prev* is not nill
        if u is not *visited* **then**
            GetLineGraph$(G, L(G), u, z)$
        else
            for each node k in $L(G)$ that originates from u **do**
                create an edge in $L(G)$ from z to k

3.2   PageRank

PageRank was first introduced by Brin & Page in 1998 [5,14]. It was developed in order to determine a measure of importance of web pages in a hyperlinked

network of web pages. The basic idea is that PageRank will assign a score of importance to every node (web page) in the network. The underlying assumption is that important nodes are those that receive many "recommendations" (in-links) from other important nodes (web pages). In other words, it is a link analysis algorithm that assigns numerical scores to the nodes of a graph in order to measure the importance of each node in the graph. PageRank works by counting the number and quality/importance of edges pointing to a node and then estimate the importance of that node. We use a similar approach in order to determine the importance of edges in a directed graph. The underlying assumption of our technique is that the number of cycles that contain a specific edge e will be reflected in PageRank score of e. Thus the removal of edges with high PageRank score is likely to break the most cycles in the graph.

Given a graph with n nodes and m edges, PageRank starts by assigning an initial score of $1/n$ to all the nodes of a graph. Then for a predefined number of iterations each node divides its current score equally amongst its outgoing edges and then passes these values to the nodes it is pointing to. If a node has no outgoing links then it keeps its score to itself. Afterwards, each node updates its new score to be the sum of the incoming values. It is obvious that after enough iterations all PageRank values will inevitably gather in the sinks of the graph. In use cases where that is a problem a damping factor is used, where each node gets a percentage of its designated score and the rest gets passed to all other nodes of the graph. For our use case we have no need for this damping factor as we want the scores of the nodes to truly reflect their importance. The number of iterations depends on the size and structure of a graph. We found that for small and medium graphs, which is the case in the scenario for graph visualization, about five iterations were enough for the scores of the nodes to converge. Depending on the implementation, PageRank can run in $O(k(n+m))$ time, where k is the number of iterations. The pseudocode for PageRank is presented in Algorithm 4.

Algorithm 4. PageRank

> **Input:** Digraph $G = (V, E)$, number of iterations k
> **Output:** PageRank scores of G
> **for each** node v in G **do**
> $PR(v) \leftarrow \frac{1}{|V|}$
> **for** k iterations **do**
>   **for each** node v in G **do**
> $PR(v) \leftarrow \sum_{u \in in(v)} \frac{PR_{old}(u)}{|out(u)|}$
> **return** PR

3.3   PageRankFAS

Our proposed algorithm is based on the concepts of PageRank and Line Digraphs. The idea behind *PageRankFAS* is that we can score the edges of

G based on their involvement in cycles: For each strongly connected compo-
nent $(s_1, s_2, ..., s_j)$ of G, it computes the line digraph $L(s_i)$ of the i-th strongly
connected component, to transform edges to nodes; next it runs the PageRank
algorithm on $L(s_i)$ to obtain a score for each edge of s_i in G.

We observed that the nodes of the line digraphs with the highest PageRank
score correspond to edges that are involved in the most cycles of G. We also
observed that the nodes of the line digraphs with lower score correspond to edges
of G with low involvement in cycles. Using this knowledge, we run PageRankFAS
for a number of iterations. In each iteration, we use PageRank to calculate the
node scores of each $L(s_i)$ and remove the node(s) with the highest PageRank
score, also removing the corresponding edge(s) from G. We repeat this process
until G becomes acyclic. The pseudocode is presented in Algorithm 5.

Algorithm 5. PageRankFAS

    Input: Digraph $G = (V, E)$
    Output: Feedback Arc Set of G
    $fas \leftarrow \emptyset$
    while G has cycles **do**
        Let $(s_1, s_2, ..., s_j)$ be the strongly connected components of G
        for each strongly connected component s_i **do**
            Create a line digraph $L(s_i)$ with every edge of s_i as a node
            $v \leftarrow$ random node of s_i
            GetLineGraph$(s_i, L(s_i), v, nill)$
            PageRank$(L(s_i))$
            $u \leftarrow$ node of $L(s_i)$ with highest PageRank value
            $e \leftarrow$ edge of G corresponding to u
            Add e to fas
            Remove e from G
    return fas

---

4   Experiments and Discussion

Here we report the experimental results and describe some details of our setup.
All of our algorithms are implemented in Java 8 using the WebGraph frame-
work [2,3] and tested on a single machine with Apple's M1 processor, 8GB of
RAM and running macOS Monterey 12.
Datasets: In order to evaluate our proposed heuristic algorithm we used four
different datasets:

1. Randomly generated graphs with 100, 200, 400, 1000, 2000, 4000 nodes and
   an average out-degree of 1.5, 3 and 5 each.
2. Three directed graphs from the datasets in graphdrawing.org, suitably mod-
   ified in order to contain cycles (since the originals are DAGs).

3. Randomly generated graphs with 50, 100 and 150 nodes and average out-degrees of 1.5, 3, 5, 8, 10 and 15 each.
4. Two webgraphs from the Laboratory of Web Algorithmics[1], also used in [17].

We randomly generate a total of 36 graphs using a predefined number of nodes, average out-degree and back edge percentage, and we repeat the process 10 times. By construction, this model has the advantage that we know in advance an upper bound to the FAS size, since the number of randomly created back edges divided by the total number of edges, is an upper-bound to the size of a minimum FAS. Finally, in order to avoid having abrupt results due to randomness, for each case we run the three algorithms on 10 created graphs and report the average numbers. This smooths out several points in our curves.

4.1   FAS with Respect to the Number of Nodes

The first set of experiments gives us an idea of how PageRankFAS performs on graphs, with varying number of nodes in comparison to the other two algorithms. It is noteworthy that in most cases the FAS found by PageRankFAS is less than 50% of the FAS found by GreedyFAS and SortFAS. As a matter of fact, for large visualization graphs with 4,000 nodes and 12,000 edges the reduction in the FAS size is almost 55% with respect to the FAS produced by GreedyFAS. The execution time taken by PageRankFAS is less than one second for graphs up to 1,000 nodes, which is similar to the time of the other two heuristics. For the larger graphs, even up to 4,000 nodes the time required is less than 8 s, whereas, the other heuristics run in about 1–2 s. The results of this experiment are shown in Fig. 1. It is interesting to note that the performance of SortFAS is better than the performance of GreedyFAS as the graphs become denser, and in fact, SortFAS actually out-performs GreedyFAS when the graphs have an average out-degree 5 and above, see Fig. 1(c).
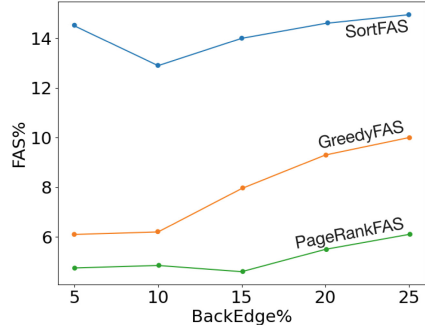
4.2   FAS with Respect to the Number of Back Edges

The second type of experiments make use of three graphs from graphdrawing.org. Since these graphs are directed acyclic, we randomly added back edges in different percentages of the total number of edges. We did this in a controlled manner in order to know in advance an upper bound of FAS. PageRankFAS gave by far the best FAS results and GreedyFAS also produced FAS with sizes mostly below 10%. SortFAS was not competitive in this dataset. The results are shown in Fig. 2. The execution time taken by PageRankFAS is well below 0.15 of a second for all graphs, which is similar to the other two heuristics.
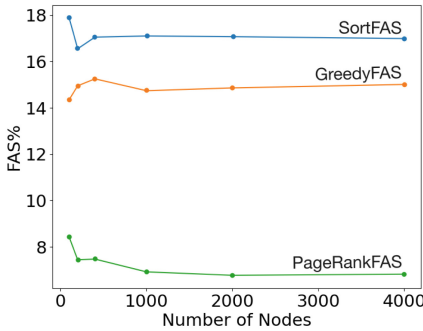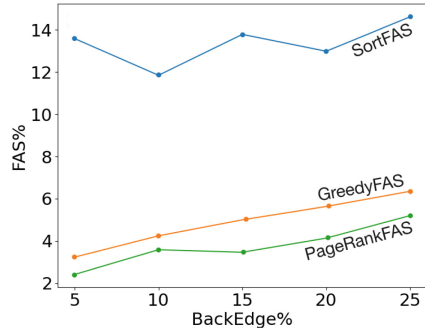
---

[1] https://law.di.unimi.it/datasets.php.
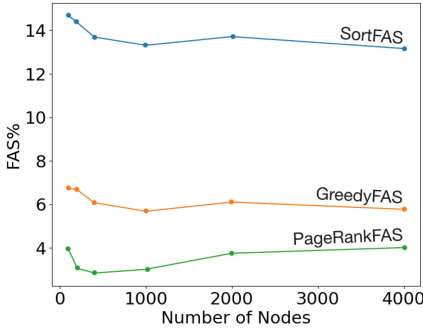
(a) Graphs with average out-degree 1.5

(b) Graphs with average out-degree 3

(c) Graphs with average out-degree 5

Fig. 1. FAS percentage for graphs with increasing number of nodes and three different average out-degrees.
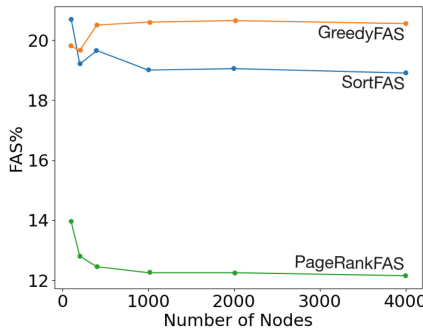

(a) Graph with 50 nodes and 75 edges before modification

(b) Graph with 75 nodes and 86 edges before modification
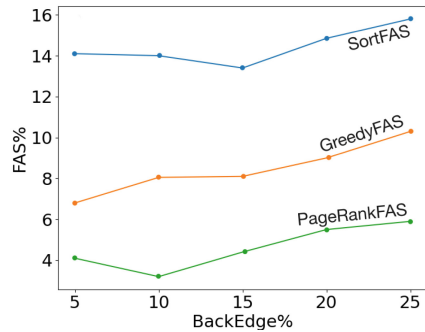
(c) Graph with 99 nodes and 154 edges before modification

Fig. 2. FAS percentage for 3 types of graphs from graphdrawing.org and for various numbers of back edges.

4.3   FAS with Respect to the Average Out-Degree

Motivated by the results shown in Fig. 1(c) we decided to investigate the correlation between the density of a graph and its potential FAS percentage. In this experiment, we created 18 different graphs, six of them with 50 nodes, six with 100 nodes and six with 150 nodes as follows: For each node size (i.e., 50, 100, 150) six graphs with average out-degrees 1.5, 3, 5, 8, 10 and 15. Again, as with our previous experiments, the results reported here are the averages of 10 runs in order to compensate for the randomness of each graph and to get smoother curves. The results of this experiment are shown in Fig. 3.

The results of PageRankFAS are consistently better than the results of GreedyFAS and SortFAS for all graphs. The results of GreedyFAS and Sort-FAS are very close to each other, for the graphs with 50 nodes. Notice however that, SortFAS outperforms GreedyFAS when the number of nodes exceeds 100 and the average out-degree exceeds five. This is aligned with the results shown in Fig. 1(c). Furthermore, as expected, when the average out-degree increases the FAS size clearly increases. Consequently, all techniques seem to converge at higher percentages of FAS size. Again, PageRankFAS runs in a small fraction of a second for all graphs, which is similar to the running times of the other two heuristics.

4.4   PageRankFAS on Webgraphs

The experiments reported in [17] use large and extra large benchmark webgraphs. Their smaller benchmarks are *wordassociation-2011* (with 10,617 nodes, 72,172 edges, which implies an average degree 6.80) and *enron* (with 69,244 nodes, 276,143 edges, which implies an average degree 3.86).

The authors report that the sizes of a FAS found by GreedyFAS and Sort-FAS for wordassociation-2011 are 18.89% and 20.17%, respectively [17]. We ran PageRankFAS for wordassociation-2011 and obtained a FAS of size 14.85%. Similarly, for webgraph enron they report a FAS of 12.54% and 14.16% respectively. We ran PageRankFAS on webgraph enron and obtained a FAS of size 11.05%. The results are shown in Fig. 4. As expected, and consistent with our experimental observations of the previous subsections, the FAS size of the denser webgraph (wordassociation-2011) is larger than the FAS size of the sparser graph (enron), as computed by all heuristics.
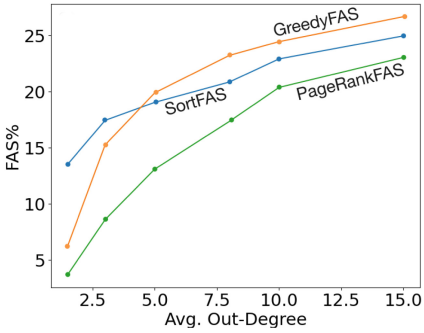
Unfortunately, the required execution time of our algorithm does not allow us to test it on the larger webgraphs used in [17]. However, it is interesting that there exists a FAS of smaller size for these large graphs, which, to the best of our knowledge, was not known before.
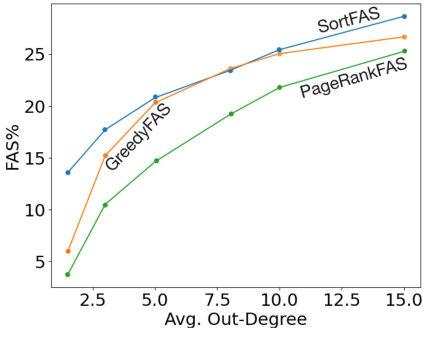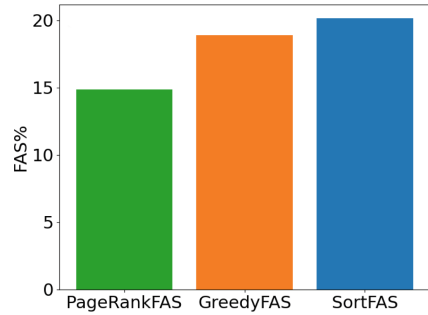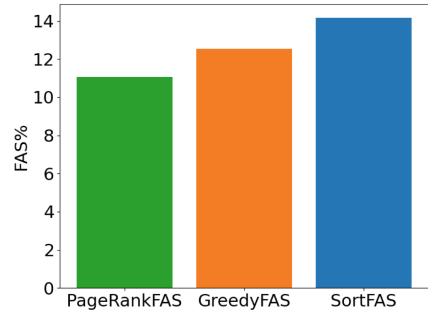
(a) Graphs with 50 nodes
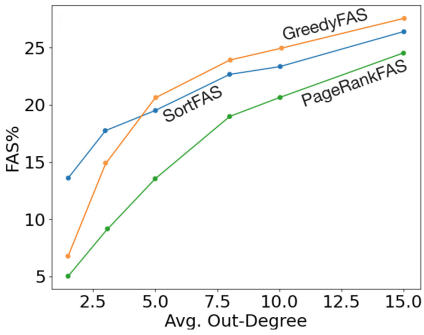
(b) Graphs with 100 nodes

(a) wordassociation-2011

(c) Graphs with 150 nodes

(b) enron

Fig. 3. FAS percentage depending on the average out-degree of three different types of graphs.

Fig. 4. FAS percentage on two web-graphs.

5   Conclusions

We presented a heuristic algorithm for computing a FAS of minimum size based on PageRank. Our experimental results show that the size of a FAS computed by our heuristic algorithm is typically about 50% smaller than the sizes obtained by the best previous heuristics. Our algorithm is more time consuming than the best previous heuristics, but it's running time is reasonable for graphs up to 4,000 nodes. For smaller graphs, up to 1,000 nodes, the execution time is well below one second, which is similar to the running times of the other two heuristics. Therefore, this is acceptable for graph drawing applications. An interesting side result is that we found out that the FAS-size of two large graphs is significantly less than it was known before. Since it is NP-hard to compute the minimum FAS, the optimum solution for these webgraphs is unknown. Hence, we do not know how close our solutions are to the optimum. It would be interesting to investigate techniques to speedup PageRankFAS in order to make it more applicable to larger webgraphs.

References

1. Ailon, N., Charikar, M., Newman, A.: Aggregating inconsistent information: ranking and clustering. Journal of the ACM (JACM) **55**(5), 1–27 (2008). https://doi.org/10.1145/1411509.1411513
2. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M.P., Bertino, E., Kumar, R. (eds.) Proceedings of the 20th international conference on World Wide Web. pp. 587–596. ACM Press (2011)
3. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proc. of the Thirteenth International World Wide Web Conference (WWW 2004). pp. 595–601. ACM Press, Manhattan, USA (2004)
4. Brandenburg, F.J., Hanauer, K.: Sorting heuristics for the feedback arc set problem. In: Technical Report MIP-1104. University of Passau Germany (2011)
5. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Comput. Networks **30**(1–7), 107–117 (1998)
6. Budak, C., Agrawal, D., El Abbadi, A.: Limiting the spread of misinformation in social networks. In: Proceedings of the 20th International Conference on World Wide Web, pp. 665–674 (2011). https://doi.org/10.1145/1963405.1963499
7. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing, vol. 357. Prentice Hall, Upper Saddle River (1999)
8. Eades, P., Lin, X., Smyth, W.F.: A fast and effective heuristic for the feedback arc set problem. Inf. Process. Lett. **47**(6), 319–323 (1993)
9. He, X., Song, G., Chen, W., Jiang, Q.: Influence blocking maximization in social networks under the competitive linear threshold model. In: Proceedings of the 2012 SIAM International Conference on Data Mining, pp. 463–474. SIAM (2012)
10. Johnson, D.S.: The np-completeness column: an ongoing gulde. J. Algorithms **3**(4), 381–395 (1982)

11. Karp, R.M.: Reducibility among combinatorial problems. In: Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming 1958-2008, pp. 219–241. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-540-68279-0_8
12. Nikolov, N.S., Healy, P.: Hierarchical drawing algorithms. In: Handbook of Graph Drawing and Visualization, ed. Roberto Tamassia, pp. 409–453. CRC Press (2014)
13. Ortali, G., Tollis, I.G.: A new framework for hierarchical drawings. J. Graph Algorithms Appl. **23**(3), 553–578 (2019). https://doi.org/10.7155/jgaa.00502
14. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Tech. rep, Stanford InfoLab (1999)
15. Pemmaraju, S., Skiena, S.: Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica ®. Cambridge University Press (1990)
16. Simpson, M., Srinivasan, V., Thomo, A.: Clearing contamination in large networks. IEEE Trans. Knowl. Data Eng. **28**(6), 1435–1448 (2016). https://doi.org/10.1109/TKDE.2016.2525993
17. Simpson, M., Srinivasan, V., Thomo, A.: Efficient computation of feedback arc set at web-scale. Proc. VLDB Endowment **10**(3), 133–144 (2016). https://doi.org/10.14778/3021924.3021930
18. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Trans. Syst. Man Cybern. **11**(2), 109–125 (1981). https://doi.org/10.1109/TSMC.1981.4308636
19. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)