Efficient Computation of Feedback Arc Set at Web-Scale

Michael Simpson
University of Victoria
Victoria, Canada
simpsonm@uvic.ca

Venkatesh Srinivasan
University of Victoria
Victoria, Canada
srinivas@uvic.ca

Alex Thomo
University of Victoria
Victoria, Canada
thomo@uvic.ca

ABSTRACT

The minimum feedback arc set problem is an NP-hard problem on graphs that seeks a minimum set of arcs which, when removed from the graph, leave it acyclic. In this work, we investigate several approximations for computing a minimum feedback arc set with the goal of comparing the quality of the solutions and the running times. Our investigation is motivated by applications in Social Network Analysis such as misinformation removal and label propagation. We present careful algorithmic engineering for multiple algorithms to improve the scalability of each approach. In particular, two approaches we optimize (one greedy and one randomized) provide a nice balance between feedback arc set size and running time complexity. We experimentally compare the performance of a wide range of algorithms on a broad selection of large online networks including Twitter, LiveJournal, and the Clueweb12 dataset. The experiments reveal that our greedy and randomized implementations outperform the other approaches by simultaneously computing a feedback arc set of competitive size and scaling to web-scale graphs with billions of vertices and tens of billions of arcs. Finally, we extend the algorithms considered to the probabilistic case in which arcs are realized with some fixed probability and provide detailed experimental comparisons.

1. INTRODUCTION

Recently, numerous studies have been carried out on social networks regarding the clearing or limiting of misinformation (cf. [10, 16, 27, 33]). For instance, [33] presents an approximation algorithm for clearing misinformation from a social network that models the network as a general directed graph and hinges on the ability of making the input graph acyclic first by placing guards on a subset of arcs. Here, it is important to remove as few arcs as possible since such arcs need to be guarded throughout the clearing process.

For another perspective on combating misinformation, [10] considers the problem of influence limitation of a bad campaign by selecting a set of seeds from where to start a good (limiting) campaign. This involves determining the eventual influence of the nodes already infected by the bad campaign. Unfortunately, computing influence is #P-hard. Nonetheless, if the network is a directed acyclic graph (DAG), then as shown by [11], computing influence is linear-time solvable. Here as well, placing guards on a subset of arcs, in effect making the graph acyclic, allows for scalable solutions.

Another prominent area of study in Social Network Analysis that can benefit from the elimination of cycles is label propagation [15]. The goal of label propagation is to consider the labels held by a subset of users in the network and attempt to predict the labels of the remaining users. This prediction is useful in determining, for example, the political leanings of users in the network or promising pairings on a dating website. The label propagation technique of choice is Belief Propagation (BP) [28] and it has also been successfully applied in fraud and malware detection. Unfortunately, BP has known convergence problems in graphs with cycles [15, 32] and the exact criteria for convergence is not known [26]. As a result, alternative avenues to applying BP directly have been explored in which an acyclic graph is first obtained from the original network [3, 5].

Thus, for all these areas of research, having the ability to produce acyclic graphs that very closely represent large networks while minimizing the perturbation to the overall structure of the network would be highly advantageous. This goal can be accomplished through the computation of an appropriate *feedback arc set* (FAS).

A feedback arc set of a directed graph G is a subset of arcs F of G such that removing F from G leaves an acyclic graph. Equivalently, a feedback arc set contains at least one arc from each cycle in G. The *minimum feedback arc set problem*, a widely studied combinatorial optimization problem on graphs, seeks to minimize the size of the subset of arcs F. The decision version of the FAS problem is shown to be NP-complete in Karp's seminal paper [17]. The problem remains NP-hard on several subclasses of graphs including tournaments [1] and Eulerian graphs [29]. Investigation into approximation algorithms for the FAS problem [12, 14, 20] in recent years has led to steady improvements; unfortunately, the approaches that yield good approximation guarantees are unable to scale due to unfavorable running time complexities.

Present day research continues to analyze datasets of increasing size. This requires algorithms for computing solutions to graph problems, such as FAS, to be able to scale to handle massive input graphs. The ability to scale is most relevant in social network analysis and web-related problems

Proceedings of the VLDB Endowment, Vol. 10, No. 3

where current datasets can reach billions of vertices and tens of billions of arcs [7]. However, despite the importance of the FAS problem and the many efforts devoted to studying it, a complete and clear picture of the problem from an empirical viewpoint still appears elusive with existing empirical studies being incomplete by only considering a restricted version of the problem, for example [12], or not scaling the considered approaches to real graphs, such as [2, 9].

We carry out an extensive experimental comparison of various algorithms for the FAS problem with runtime complexity $O(n^2)$ or less that have not been compared before in the literature on a full range of dataset sizes. The various approaches we consider occupy the broad classes of sorting-based, traversal-based, and randomized and fall into three running time complexity classes: $O(m+n)$, $O(n \log n)$, and $O(n^2)$. In terms of scalability, we observe that it is the algorithms in the $O(m+n)$ class that can handle graphs of billions of vertices and arcs. However, some algorithms needed careful engineering to best be placed in a complexity class. Our optimizations are discussed further in Section 2. In addition, we provide insights into the relative performance of the various approaches and provide global graph properties that indicate when the algorithms perform favourably.

Furthermore, we extend our work to the probabilistic case in which arcs are realized with a fixed probability. We adapt our optimized algorithms for the FAS problem to the probabilistic case and conduct experiments on several large datasets. We show that the expected number of arcs in a FAS is usually a small fraction of the expected number of total arcs in the graph.

Our goal is to significantly speed-up the computation of a FAS and scale-up to massive graphs with tens of billions of arcs. Furthermore, we would like to achieve this using only a medium-range machine. In order to make the graph footprint as small as possible, we used *webgraph*, a highly efficient and actively maintained graph compression framework [7].

In summary, the contributions of this paper are:

1. We present algorithmic engineering of several approaches for computing a feedback arc set in order to improve their scalability.

2. We present a thorough experimental study of the known methods for computing feedback arc sets with the potential to scale. For the comparison of these algorithms, we focus on the parameters of running time and feedback arc set size.

3. We provide insights into the relative performance of the considered approaches and present global graph properties that indicate when the algorithms perform favourably.

4. We extend our optimized algorithms for the FAS problem to the probabilistic case and carry out further experimental comparisons.

The layout of this work is as follows. Section 2 details each of the algorithms considered and outlines the optimizations proposed. Section 3 presents our experimental results and in Section 4 we offer a discussion of our results. In Section 5 we investigate the probabilistic case and Section 6 considers related works. Finally, we conclude in section 7.

2. ALGORITHMS

In this section we outline the scalable approaches we consider for the FAS problem. For a discussion of the various approximations algorithms, refer to the related work in Section 6. For most algorithms we provide pseudocode and give the asymptotic running times. Furthermore, we discuss the various optimizations we make to each algorithm.

The algorithm that produced the best FAS size for our largest datasets, but whose running time needed care to bring down to $O(m+n)$, was GreedyFAS due to Eades, Lin & Smyth [13]. A direct implementation of GreedyFAS runs in $O(n^2)$ time, which is impractical for large social and web networks since they are sparse graphs with $m \ll n^2$. To remedy this, we present an engineering of its data structures which brings its complexity to $O(m+n)$, thus making it scalable to the largest dataset we consider on tens of billion of arcs. Another algorithm we engineer is SortFAS of Brandenburg & Hanauer [9]. A direct implementation of it runs in $O(n^3)$ and we bring it down to $O(n^2)$ with our optimized implementation. Finally, we also optimize a randomized algorithm, BergerShorFAS, by Berger & Shor [4], which computes a reasonably small FAS while running in $O(m+n)$ time.

2.1 Graph Representation

We consider directed simple graphs, i.e. there are no self-loops or multiple arcs. A graph G on n vertices and m arcs has a vertex set V and arc set E. The vertices are labelled $1 \dots n$. Furthermore, due to the large size of the datasets we consider, our graph data structure uses adjacency lists to maintain neighbour relationships. We make use of the *webgraph* framework [7] which is a highly efficient graph compression framework that allows accessing a graph without fully decompressing it by providing only the required components on the fly.

The data structure offered by webgraph is an immutable graph. As such, all the vertex or arc removals in the considered algorithms are implemented as logical removals.

We will illustrate the algorithms using the graph shown in Figure 1 on 8 vertices and 13 arcs. Observe that the minimum FAS contains the single arc $(3, 4)$.

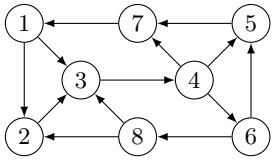
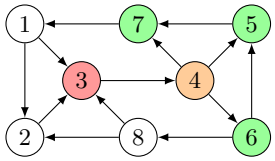


Figure 1: An example graph with a minimum FAS of size 1.

2.2 Equivalent Formulations

The FAS problem has an equivalent formulation called the Linear Arrangement (LA) problem. The LA problem takes as input a directed graph G and outputs an ordering of the vertices for which the number of arcs pointing backward from right to left is a minimum. The backward arcs are exactly those arcs that make up a FAS since removing them from G leaves the graph acyclic. We make use of the LA formulation for a subset of the approaches considered.

The dual of the FAS problem is the Maximum Acyclic Subgraph (MAS) problem. The MAS problem takes as input a directed graph G and returns the maximum acyclic



$\delta(3) \leftarrow \delta(3) - 1 \quad \delta(6) \leftarrow \delta(6) + 1$

$\delta(5) \leftarrow \delta(5) + 1 \quad \delta(7) \leftarrow \delta(7) + 1$

Figure 2: Processing vertex 4 during the execution of GreedyFAS on the example graph.

subgraph of G. Thus, the arcs of G that do not appear in the MAS solution are exactly the arcs that make up a minimum FAS for G. The SimpleFAS and BergerShorFAS (see subsections 2.4 & 2.5) approaches are based on algorithms for the MAS problem.

2.3 GreedyFAS

GreedyFAS was introduced by Eades, Lin & Smyth [13] as an efficient approximation algorithm for the feedback arc set problem. For each vertex $u \in V$, let $d^+(u)$ denote the outdegree of u and $d^-(u)$ denote the indegree of u. In each iteration of GreedyFAS, the algorithm removes vertices from G that are sources or sinks followed by a vertex u for which $\delta(u) = d^+(u) - d^-(u)$ is currently a maximum. If a vertex u removed from G is a sink it is prepended to a vertex sequence s_2, otherwise it is appended to a vertex sequence s_1. Once all the vertices of G have been removed, the sequence $s = s_1 s_2$ is returned as a linear arrangement for which the backward arcs make up a feedback arc set. The pseudocode for GreedyFAS is presented in Algorithm 1.

The intuition behind this approach is to greedily move all the "sink-like" vertices to the right-side of the ordering and all the "source-like" vertices to the left-side in an attempt to minimize the number of arcs oriented from right to left.

As an illustrative example, consider the execution of GreedyFAS on the graph in Figure 1. Initially, there are no sinks or sources so we remove the vertex u for which $\delta(u)$ is a maximum: vertex 4. The vertex and its arcs are removed from G and vertex 4 is appended to s_1. After removing vertex 4 from G a sink is created at vertex 3. As such, vertex 3 is removed from G and prepended to s_2. Again, a new sink is created at vertex 2. The process continues removing vertices 1, 7, 5, 8, and finally 6 which are all prepended to s_2. The resulting sequence is $s = s_1 s_2 = [4, 6, 8, 5, 7, 1, 2, 3]$. Thus, we can extract a feedback arc set of size 1 by observing that this linear arrangement has only a single backward arc from vertex 3 to 4. In Figure 2 we illustrate the execution of GreedyFAS in its first iteration when processing vertex 4 in the example graph. We show in red and green the vertices that have their δ value decremented and incremented, respectively.

Eades, Lin & Smyth discuss the following implementation details for their algorithm. To begin, it is convenient to partition the vertices of G into sources, sinks, and δ-classes as follows:

$$\begin{aligned} V_{n-2} &= \{u \in V \mid d^-(u) = 0;\ d^+(u) > 0\} \\ V_{-n+2} &= \{u \in V \mid d^+(u) = 0\} \\ V_d &= \{u \in V \mid d = \delta(u);\ d^+(u) > 0;\ d^-(u) > 0\} \end{aligned}$$

Algorithm 1 GreedyFAS

Input: Input directed graph $G = (V, E)$
Output: Linear arrangement A

```
s1 ← ∅, s2 ← ∅
while G ≠ ∅ do
    while G contains a sink do
        choose a sink u
        s2 ← u s2
        G ← G \ u
    while G contains a source do
        choose a source u
        s1 ← s1 u
        G ← G \ u
    choose a vertex u for which δ(u) is a maximum
    s1 ← s1 u
    G ← G \ u
return s = s1 s2
```

with $d \in [-n + 3, n - 3]$. Then, we can see that every vertex $u \in V$ falls into exactly one of these $2n-3$ classes. For a given directed graph G an initialization phase computes these classes.

Each vertex class is implemented as a bin with the vertices in each class linked together by a doubly-linked list. Then, using the bins, we can recognize a sink, source, or vertex u for which $\delta(u)$ is a maximum. Finally, consider how to form $G \setminus u$. The vertex u itself can be logically removed by eliminating it from its bin list. As a result, every vertex v adjacent to u will either become a sink, a source, or an element of an adjacent bin.

We optimized the GreedyFAS implementation into the following two versions. In the first version, referred to as *dllFAS*, we implement custom doubly-linked lists for the bins in order to directly manipulate the list nodes and alleviate a bottleneck suffered by generic (library) lists. When moving a vertex to an adjacent bin we must first delete the vertex from its current list. In a generic list this deletion requires a linear pass over the list to find the appropriate list node to delete. Only then can we add the vertex to the bin corresponding to its updated vertex class. In the worst case, the size of a bin could be $O(n)$. In order to facilitate moving a vertex to an adjacent bin in $O(1)$ time, we maintain an array containing each individual list node of every vertex in G. As such, we are able to manipulate the node's previous and next node references directly and complete the update efficiently regardless of the size of each bin.

In the second version, referred to as *ArrayFAS*, we do away with the doubly-linked lists all together by maintaining three flat arrays that mimic the behaviour of the lists. The first array, *bins*, maintains the tail of bin i. That is, at position i in array *bins* we store the vertex currently at the tail of the doubly-linked list for bin i. The remaining two arrays, *prev* and *next*, contain information regarding the node references for the list node corresponding to vertex i. That is, at position i in array *prev* (*next*) we store the index of the vertex that is before (after) vertex i in i's vertex class bin. ArrayFAS aims to further reduce the space overhead and initialization time required by GreedyFAS. By tightly packing the data structure in flat arrays we allow for better memory management and achieve more efficient execution.

In conclusion, we analyze the complexity of forming $G \setminus u$. The vertex u itself can be logically removed in $O(1)$ time by eliminating it from its bin list. Then, the update for every vertex v adjacent to u moving to an adjacent bin requires $O(1)$ time for each arc incident to u. As such, GreedyFAS runs in time $O(m+n)$ and uses $O(m+n)$ space. It has an approximation guarantee of $\frac{1}{2}|E| - \frac{1}{6}|V|$ but we observe in the experiments that the size of the FAS produced is drastically smaller than the size suggested by the worst-case bound.

2.3.1 GreedyAbsFAS

In [12] a variation of GreedyFAS was proposed in which the greedy selection of a vertex is determined by the absolute value of the degree difference, $max(|d^+(u) - d^-(u)|)$, and the vertex is treated as a sink or source depending on whether the outdegree is greater than the indegree or vice versa. Unlike our GreedyFAS implementation where only the current maximum δ-class needs to be tracked, the improved algorithm must maintain both the maximum and minimum δ-classes that are currently occupied in order to efficiently make successive greedy selections for vertex processing. However, the approach did not offer a significant performance improvement as the difference between the FAS size computed by each approach was always less than 1%, often much less.

2.4 SimpleFAS

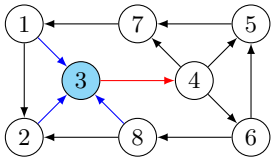
SimpleFAS is based on a very simple 2-approximation algorithm for the MAS problem. First, we fix an arbitrary permutation P of the vertices of G. Then, we construct two subgraphs L and R, containing the arcs (u, v) where $u < v$ in P and those where $u > v$ in P, respectively. After this construction, both L and R are acyclic subgraphs of G, and at least one of them is at least half the size of the maximum acyclic subgraph. Therefore, we can return $m - max(|L|, |R|)$ as the size of a feedback arc set for G. The runtime complexity of SimpleFAS is $O(m+n)$.

2.5 BergerShorFAS

BergerShorFAS is based on an approximation algorithm for the MAS problem due to Berger & Shor [4]. The algorithm begins by choosing a random permutation P of the vertices of G. Then, the vertices are processed in order according to P as follows. When processed, if a vertex has more incoming arcs than outgoing ones, the incoming arcs are removed from the graph and added to a set E' and the outgoing arcs are removed from the graph and discarded. If there are at least as many outgoing arcs as incoming arcs, instead the outgoing arcs are added to E' and the incoming arcs are discarded and removed from G. When all the vertices have been processed, $G' = (V, E')$ is returned as an acyclic subgraph. Therefore, the arcs from $E \setminus E'$ make up a feedback arc set.

The intuition behind the above approach is as follows. At each step, selecting either the incoming or the outgoing arcs but not both ensures that the resulting graph is acyclic. Moreover, choosing at each step the set of arcs of bigger size, ensures that resulting acyclic graph has a large number of edges.

This randomized approach runs in time $O(m+n)$ and produces an acyclic subgraph containing at least $(1/2 + \Omega(1/\sqrt{d_{max}}))|E|$ arcs, where d_{max} is the maximum vertex degree of G. In our experiments, BergerShorFAS far outperforms the worst-case bound provided.



$E' \leftarrow E' \cup \{(1,3),(2,3),(8,3)\}$

Figure 3: Processing vertex 3 during the execution of the Berger-Shor algorithm on the example graph.

As an illustrative example, consider the execution of Berger-Shor algorithm on the graph in Figure 1 with an initial ordering of $[3, 6, 4, 8, 7, 1, 2, 5]$. First, we process vertex 3 and observe that it has one outgoing arc and three incoming arcs. Thus, we add the incoming arcs $(1, 3)$, $(2, 3)$, and $(8, 3)$ to E' before removing them and the outgoing arc $(3, 4)$ from G. Next, we process vertex 6 adding the arcs $(6, 5)$ and $(6, 8)$ to E'. This process continues adding the arcs $(4, 5)$, $(4, 7)$, $(8, 2)$, $(7, 1)$, and $(1, 2)$ to E'. Finally, we can return the arcs $E \setminus E' = [(3, 4), (4, 6), (5, 7)]$ as a feedback arc set of size 3. In Figure 3 we illustrate the execution of the algorithm when processing vertex 3 in the example graph G.

In Algorithm 2, we give BergerShorFAS, which adapts the above algorithm to compute the feedback arc set, F, directly without first computing E'. We do this because maintaining E' is more memory demanding than maintaining F.[1]

Algorithm 2 BergerShorFAS

Input: Input directed graph $G = (V, E)$
Output: A feedback arc set for G

```
Fix an arbitrary permutation P of the vertices of G
F ← ∅
for all vertices v processed in order based on P do
    if inDegree(v) > outDegree(v) then
        F ← F ∪ {(v, u) : u ∈ G.succ(v)}
    else
        F ← F ∪ {(u, v) : u ∈ G.pred(v)}
    E ← E \ ({(v, u) : u ∈ G.succ(v)} ∪
             {(u, v) : u ∈ G.pred(v)})
return F
```

Notice that the BergerShorFAS algorithm manipulates the graph data structure during its execution by deleting arcs. However, since we use the webgraph framework, our graph data structure is in an immutable compressed state and therefore we are unable to delete parts of the graph in this way. Instead, we observe that in each iteration, all the arcs going-to or leaving-from a vertex v are deleted. In essence this amounts to deleting v itself. As such, in each iteration, we label vertices as deleted, but do not physically remove them from graph. We consider only those vertices that have not been labelled as deleted in the execution of the algorithm.

[1]In applications, once a feedback arc set F is computed, using the webgraph API, we can generate a new immutable compressed graph not containing the arcs in F in only a single pass over the original graph G while consulting F. This process needs memory mainly to hold F.

Furthermore, instead of marking both vertices and arcs as deleted, we only mark when a vertex is deleted and infer if an arc is deleted by checking to see if either endpoint has been labelled as deleted. In this way, we are able to achieve an equivalence with only a single auxiliary data structure of size $O(n)$ as opposed to two data structures of size $O(n)$ and $O(m)$, respectively, if we labelled arcs as well. We maintain the status of a vertex, deleted or present, via a bit set to minimize the space overhead.

2.6 dfsFAS

The hallmark algorithm for graph traversals, depth-first search (DFS), can be used to compute a feedback arc set. Removing all the *back arcs* of a depth-first traversal ensures the resulting graph is acyclic. We follow the standard vertex colouring approach to identifying the back arcs. The runtime complexity of dfsFAS is $O(m + n)$.

Note, if the number of back arcs exceeds half the number of arcs in G, we can instead return the complementary arcs as a FAS since the back arcs of a DFS cannot contain a cycle. Recall that in the DFS ordering a back arc points from a vertex with a higher label to a smaller one. Therefore, by transitivity, we cannot have a cycle among back arcs.

Observe that DFS has the ability to output a FAS as a side effect of its execution, but at no point in the execution does it make any intelligent decisions which act to minimize the resulting FAS size.

2.7 KwikSortFAS

The KwikSortFAS heuristic, originally introduced by Ailon et al. [1] as a 3-approximation algorithm for the FAS problem on tournaments, was later extended by Brandenburg & Hanauer [9] as a heuristic for general directed graphs.

Brandenburg and Hanauer [9] extend several classical sorting algorithms to heuristics for the FAS problem using the underlying idea that the vertices of a directed graph G can be treated as items to be sorted into a favourable linear arrangement based on the number of back arcs induced. These sorting-based approaches are applied to an initial linear arrangement and output a sorted linear arrangement.

The motivation for KwikSortFAS is based on the classical sorting algorithm Quicksort. The algorithm uses the 3-way partition variant of Quicksort due to it being highly adaptive in the case of sorting with many equal keys. In our application, two vertices without an arc connecting them are treated as equal when making a comparison and thus we have many equal items. Our implementation follows the optimized version due to Sedgewick and Wayne [31] which only uses $O(\log n)$ additional space.

The application of Quicksort to the vertices of G is implemented as follows. Given a starting linear arrangement, we move the vertices to the left or to the right relative to a random pivot element based on whether there is an arc to or from the pivot. The algorithm then proceeds recursively. We put the pivot vertex and the vertices equal to it, with no arc from or to the pivot, in the middle and recurse on the left, middle, and right subsets. Note that unlike in the case of sorting numbers, we need to recurse on the middle since these vertices, though equal to the pivot, could have arcs between them and hence may not be equal to each other. When ties must be broken, in the case where the remaining vertices are disconnected, their order is left unaltered. The pseudocode for KwikSortFAS is presented in Algorithm 3.

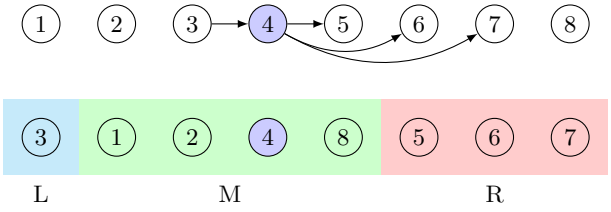


Figure 4: Initial recursive step of KwikSortFAS on the example.

Algorithm 3 KwikSortFAS

Input: Linear arrangement A, vertex lo, vertex hi
  if $lo < hi$ **then**
    $lt \leftarrow lo$, $gt \leftarrow hi$, $i \leftarrow lo$
    $p \leftarrow$ random pivot in range $[lo, hi]$
    while $i \leq gt$ **do**
      if arc (i, p) exists **then**
        $swap(lt, i)$
        $lt \leftarrow lt + 1$, $i \leftarrow i + 1$
      else if arc (p, i) exists **then**
        $swap(i, gt)$
        $gt \leftarrow gt - 1$
      else
        $i \leftarrow i + 1$
    $KwikSortFAS(A, lo, lt - 1)$
    if at least one swap was made **then**
      $KwikSortFAS(A, lt, gt)$
    $KwikSortFAS(A, gt + 1, hi)$

As an illustrative example, consider the execution of KwikSortFAS on the graph in Figure 1 with an initial ordering of $[1, 2, 3, 4, 5, 6, 7, 8]$. In the first level of recursion we randomly select vertex 4 as the pivot. This places vertex 3 to the left, vertices 1, 2, 4, and 8 in the middle and vertices 5, 6, and 7 to the right. We show this step in Figure 4. Now we recurse on each part.

The resulting sorted ordering after the recursive calls return is $[3, 1, 8, 2, 4, 5, 6, 7]$. Thus, we can extract a feedback arc set of size 6 by observing that this linear arrangement has six backward arcs.

The runtime complexity of $O(n \log n)$ for KwikSortFAS assumes that arc membership can be tested in constant time, i.e. when the graph is represented with an adjacency matrix. However, given that our graph data structure uses an adjacency list we must search the adjacency list in order to test for the presence of an arc. Conveniently, the adjacency list for each vertex is kept in sorted order and therefore we can utilize a binary search to test for the presence of an arc in G. With an adjacency list representation, the runtime complexity becomes $O(n \log n \log(d_{max}))$ where d_{max} is the maximum vertex degree in G.

Furthermore, since KwikSortFAS is randomized, each run may yield a different result. To this end, as presented in [9], we also consider KwikSort200FAS which runs KwikSortFAS 200 times on random initial linear arrangements and takes the best result.

2.8 InsertionSortFAS

The Insertion Sort-based approaches are "monotone", that is they always output a linear arrangement with at most the same number of backward arcs as the input linear ar-

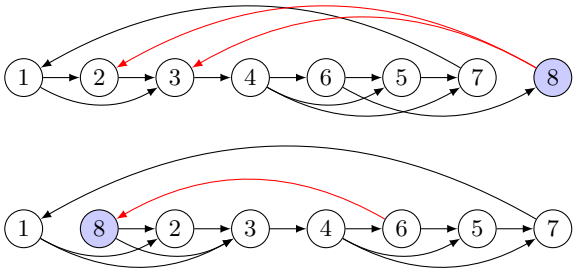


Figure 5: Iteration 8 of SortFAS on the example graph.

rangement and thus they can be applied repeatedly until convergence. The resulting algorithms are indicated by *.

2.8.1 *SortFAS*

SortFAS is equivalent to sorting by insertion for the linear arrangement problem. The vertices are processed in order according to an ordering of the vertices, $(v_1 \dots v_n)$. In the i-th iteration v_i is inserted at the optimal position among the already sorted set of the first $i-1$ vertices. In case of a tie the leftmost position is taken. The optimal position is defined as the position with the least number of backward arcs induced by v_i. Notice that only the arcs between v_i and the first $i-1$ vertices are relevant in the i-th iteration.

As an illustrative example, consider the execution of SortFAS on the graph in Figure 1 with an initial ordering of $[1,2,3,4,5,6,7,8]$. In the first iteration, the single vertex 1 is trivially sorted. In the second iteration, vertex 2 is inserted to the right of vertex 1, i.e. its position is unchanged, as there is an arc from vertex 1 to 2 and thus swapping their locations would induce a backward arc. This behaviour continues in iterations 3 through 5. Then, in iteration 6 we insert vertex 6 in between vertices 4 and 5 as this placement induces 0 backward arcs. In iteration 7 we again leave vertex 7 in place. Finally, in iteration 8, we insert vertex 8 in between vertices 1 and 2 which reduces the number of backward arcs induced from 2 to 1. The resulting sorted arrangement is $[1,8,2,3,4,6,5,7]$. Thus, we can extract a feedback arc set of size 2 by observing that this linear arrangement has two backward arcs: $(7,1)$ and $(6,8)$. In Figure 5 we show the execution of SortFAS in iteration 8 to provide some insight into the selection of the position that induces the least number of backward arcs. In this iteration, we see that vertex 8 in its original location induced two backward arcs, $(8,2)$ and $(8,3)$ shown in red, and after it is inserted at position 2, it only induces a single backward arc, $(6,8)$.

Now, while the runtime of a traditional Insertion Sort is $O(n^2)$, we must take care in the analysis when considering sorting graph vertices. In particular, in a standard Insertion Sort, we repeatedly compare the item at position i to the left until a smaller value is reached. The comparison done at each stage is typically a simple arithmetic operation, as in the case of integer values, but requires more thought when comparing two vertices in G. A naive approach is to consider inserting v_i at each of the possible $i-1$ locations. However, this would require a pass over the first $i-1$ vertices to count the number of back arcs induced for each of the $i-1$ possible locations. The resulting sum is $\sum_{i=1}^{n}(i-1)^2 \in O(n^3)$.

Instead, we engineer a more sophisticated approach that can identify the optimal location of v_i using a single pass over the $i-1$ possible locations. We begin by initializing a counter variable to zero. Then, for each possible location, j, we determine if there is an arc from v_i to v_j and from v_j to v_i. We increment, or decrement, the counter variable if the arc from v_i to v_j, or v_j to v_i, is present, respectively. This process assumes that v_i and v_j will swap locations and keeps tracks via the counter variable whether or not any current arcs between v_i and v_j would switch direction by inserting v_i at position j. For example, if there is currently a backward arc from v_i to v_j then, if v_i is inserted at position j, the arc will become a forward arc. Thus, incrementing and decrementing the counter variable indicates the potential loss or gain of a backward arc. Therefore, we can identify the minimum value achieved by the counter variable and record the value of j for which this minimum was achieved to use as the optimal location to insert v_i at since the minimum value of the counter variable corresponds to the location which induces the least number of backward arcs. The pseudocode for SortFAS is presented in Algorithm 4.

Algorithm 4 SortFAS

```
Input: Linear arrangement A
  for all vertices v in A do
      val ← 0, min ← 0, loc ← position of v
      for all positions j from loc − 1 down to 0 do
          w ← vertex at position j
          if arc (v, w) exists then
              val ← val − 1
          else if arc (w, v) exists then
              val ← val + 1
          if val ≤ min then
              min ← val, loc ← j
      insert v at position loc
```

The above implementation of SortFAS has a runtime complexity of $O(n^2)$ under the assumption that arc membership can be tested in constant time. With an adjacency list implementation the runtime complexity of SortFAS becomes $O(n^2 \log(d_{max}))$. Furthermore, SortFAS is monotone which yields the additional algorithm SortFAS*.

2.8.2 *SiftFAS*

SiftFAS, similar to SortFAS, is equivalent to two-sided Insertion Sort for the linear arrangement problem since we can place a vertex v, not only in the first $i-1$ positions, but on either side of v's current location. Here, in the i-th iteration, v_i is inserted at the optimal position across the entire current linear arrangement and in the case of a tie the leftmost position is taken. Additionally, the same technique used for SortFAS to determine the optimal location to insert v_i can be utilized to ensure the running time is equivalent to that of a traditional two-sided insertion sort.

SiftFAS also has a runtime complexity of $O(n^2)$ and is monotone which yields the additional algorithm SiftFAS*. Similiar to SortFAS, the runtime complexity given for SiftFAS assumes constant time arc testing. For an adjacency list representation, the runtime complexity is $O(n^2 \log(d_{max}))$.

3. EXPERIMENTS

In this section, we present our experimental results. All of our algorithms are implemented in Java (available at `https://github.com/stamps/FAS`) and tested on a machine with

dual 6 core 2.10GHz Intel Xeon CPUs, 32GB RAM and running Ubuntu 14.04.2.

The network statistics for all of the datasets we consider are shown in Table 1. We obtained the datasets from Laboratory of Web Algorithmics.[2] We divide the datasets by horizontal lines according to their size, small (S), medium (M), large (L), and extra-large (XL).

The algorithms are summarized in Table 2. The last column gives the sizes of the datasets that each algorithm can handle. The SimpleFAS and dfsFAS approaches serve as baseline algorithms in our experiments.

The measures of effectiveness in our experiments are the *FAS size* defined as the number of arcs in a FAS output by a particular algorithm and the *algorithm efficiency* measured in running time. Our goal is to keep both parameters as small as possible.

Name	$\|V\|$	$\|E\|$	Size (Gb)
word_assoc	10,617	72,172	0.235
enron	69,244	276,143	0.721
uk-2007	100,000	3,050,615	1.764
cnr-2000	325,557	3,216,152	3.328
uk-2002	18,520,486	298,113,762	220.945
arabic-2005	22,744,080	639,999,458	342.399
uk-2005	39,459,925	936,364,282	514.289
webbase-2001	118,142,155	1,019,903,190	1,207.959
twitter-2010	41,652,230	1,468,365,182	5,286.142
clueweb12	978,408,098	42,574,107,469	23,830.734

Table 1: Dataset Statistics

3.1 Small Datasets

The small datasets we consider are the *word_assoc* network with 10,617 vertices and 72,172 arcs, and the *enron* dataset with 69,244 vertices and 276,143 arcs.

First, we plot the size of the FAS computed by each of the algorithms considered in Figures 6a and 6b. We observe that dfsFAS, SimpleFAS, and the KwikSort-based algorithms perform the worst compared to the rest of the algorithms. GreedyFAS and Insertion Sort-based algorithms perform well, with feedback arc sets in the range 17-25% for *word_assoc* and 12-20% for *enron*. Both SortFAS* and SiftFAS* required 4 iterations to converge. We see that even though the best algorithm with respect to the size of FAS is SortFAS*, GreedyFAS is very close. BergerShorFAS computed a feedback arc set that was 25% and 17% of the size of the network for *word_assoc* and *enron*, respectively.

Second, we plot the runtimes achieved by each algorithm in Figures 6c and 6d. Note that the runtime plots use a logarithmic scale, illustrating how much faster the non-sorting based algorithms execute. Each of the sorting based algorithms, with the exception of KwikSortFAS, required at least 70 seconds to complete on the *word_assoc* network. The difference is even more severe on the *enron* network where dfsFAS, SimpleFAS, BergerShorFAS, KwikSortFAS, and GreedyFAS each complete in about 2 seconds followed by a jump up to 257 seconds for KwikSort200FAS, 1,628 for SortFAS, 3,456 for SiftFAS, 6,847 seconds for SiftFAS* with 2 iterations, and 8,429 seconds for SortFAS* with 5 iterations.

[2] `http://law.di.unimi.it/datasets.php`

In summary, on the small datasets, the feedback arc set of minimum size was computed by SortFAS*, followed closely by GreedyFAS. On the other hand, the latter was more than one order of magnitude faster than the former. As such, GreedyFAS provides simultaneously a good quality FAS and a small running time.

3.2 Medium Datasets

The medium datasets we consider are the *uk-2007* network containing 100,000 vertices and 3,050,615 arcs and the *cnr-2000* network containing 325,557 vertices and 3,216,152 arcs. On the medium sized datasets, the SortFAS and SiftFAS algorithms and their converging versions fail to complete in a reasonable amount of time and are therefore omitted from the following plots.

First, we plot the size of the FAS computed by each of the algorithms considered in Figures 7a and 7b. We see that the dfsFAS, SimpleFAS, and the KwikSort based algorithms perform the worst. For the *uk-2007* network, dfsFAS and SimpleFAS compute a feedback arc set in the range 47-50% of the size of the network, while the KwikSort based algorithms compute feedback arc sets in the range $24-26\%$. In contrast, GreedyFAS and BergerShorFAS achieve impressively small feedback arc sets of 10% and 13%, respectively. We observe very similar results for the *cnr-2000* network.

Second, we plot the runtimes achieved by each algorithm in Figures 7c and 7d. Again, we employ a logarithmic scale to best illustrate the substantial gap in runtimes among the tested algorithms.

We observe that the KwikSort-based algorithms are from one to several orders of magnitude slower than the non-sorting-based algorithms. The latter all completed in under 3 seconds on both networks with dfsFAS, SimpleFAS, and BergerShorFAS each about a second quicker than the GreedyFAS implementations.

In summary, on the medium datasets, as we move beyond the capabilities of the Insertion Sort-based algorithms, we observe the feedback arc set of minimum size being computed by GreedyFAS, followed by BergerShorFAS. With respect to runtimes, dfsFAS, SimpleFAS, and BergerShorFAS narrowly beat GreedyFAS, with all algorithms running in a very reasonable amount of time.

3.3 Large and Extra-Large Datasets

The largest, or web-scale, datasets we consider are *uk-2002*, *arabic-2005*, *uk-2005*, *webbase-2001*, *twitter-2010*, and *clueweb12*. On these datasets, the KwikSort-based algorithms fail to complete in a reasonable amount of time and are therefore omitted from the following plots, leaving us with only those algorithms that run linear in the number of arcs and vertices of the graph.

We plot the size of the FAS computed by each of the algorithms considered in Figure 8 (left-half) and the runtimes achieved by each algorithm in Figure 8 (right-half). On these datasets we observe a consistent trade-off between running time and feedback arc set size. dfsFAS and SimpleFAS complete the quickest across all the datasets, followed closely by BergerShorFAS and then by GreedyFAS.

On *clueweb12* because of its massive size, only the array-based implementation of GreedyFAS could run with the available memory.

GreedyFAS trades running time for feedback arc set quality. It outperforms all of the tested algorithms on every

Algorithm	Abbrev.	Section	Complexity	Problem	Type	Dataset-Size
SortFAS	Sort	2.8	$O(n^2)$	LA	Sorting	S
SortFAS*	Sort*	2.8	$O(n^2)$	LA	Sorting	S
SiftFAS	Sift	2.8	$O(n^2)$	LA	Sorting	S
SiftFAS*	Sift*	2.8	$O(n^2)$	LA	Sorting	S
KwikSortFAS	KS	2.7	$O(n \log n)$	LA	Sorting	S, M
KwikSortFAS200	KS200	2.7	$O(n \log n)$	LA	Sorting	S, M
GreedyFAS (dll)	G-dll	2.3	$O(m+n)$	LA	Greedy	S, M, L
GreedyFAS (array)	G-arr	2.3	$O(m+n)$	LA	Greedy	S, M, L, XL
SimpleFAS	Simple	2.4	$O(m+n)$	MAS	Randomized	S, M, L, XL
BergerShorFAS	BS	2.5	$O(m+n)$	MAS	Randomized	S, M, L, XL
dfsFAS	DFS	2.6	$O(m+n)$	-	Traversal	S, M, L, XL

Table 2: Algorithms considered

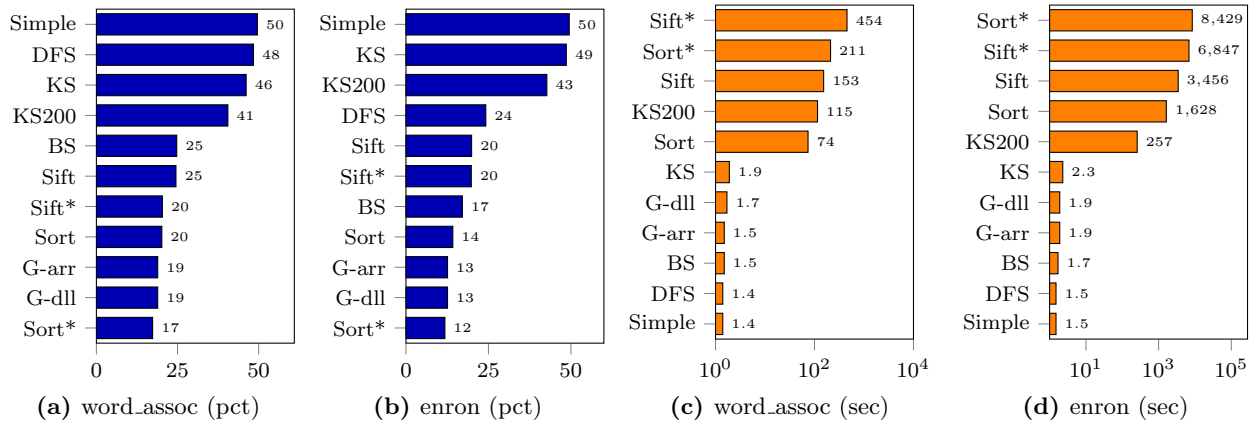


Figure 6: FAS size (left-half) and running time (right-half) for small datasets. The resulting FAS size is given as a percentage of the total number of arcs in the corresponding graph.

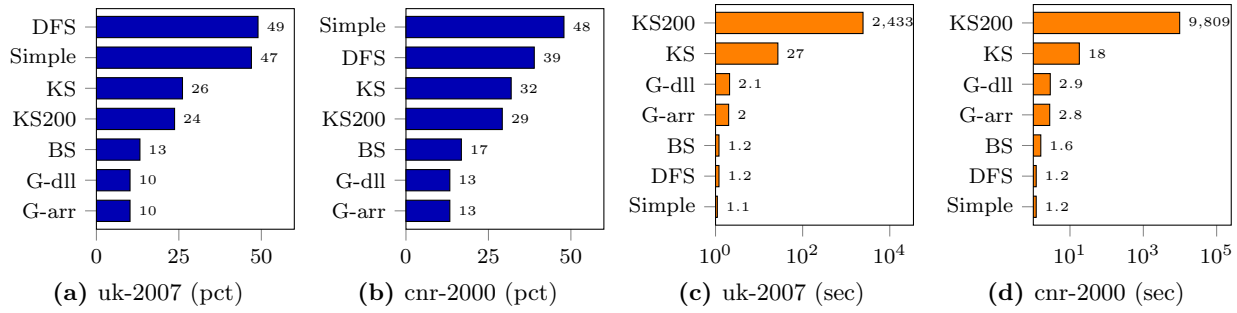


Figure 7: FAS size (left-half) and running time (right-half) for medium datasets. The resulting FAS size is given as a percentage of the total number of arcs in the corresponding graph.

dataset with respect to the FAS size. The dfsFAS and SimpleFAS algorithms produce unacceptably large feedback arc sets that are approximately 50% of the size of the network while GreedyFAS is around 15% on average. Furthermore, GreedyFAS beats its closest competitor, BergerShorFAS, by about 4%.

Remarkably, on *clueweb12*, a dataset of more than 42 billion arcs, GreedyFAS achieves an impressive 3.6% FAS size, compared to BergerShorFAS with 16%. Being able to scale greedyFAS to such an extra-large dataset is a significant contribution of this paper.

4. DISCUSSION

In this section, we summarize and discuss the most important results of the presented experiments. In particular, we provide insight into the expected and actual performance of the algorithms both independently and relative to each other.

First, we discuss when we expect each approach to perform well independently and provide global graph properties that suggest when a particular approach will perform well.

Both the GreedyFAS and BergerShorFAS approaches benefit when there are many source and sink-like vertices present in G. Therefore, indication of when they will perform well

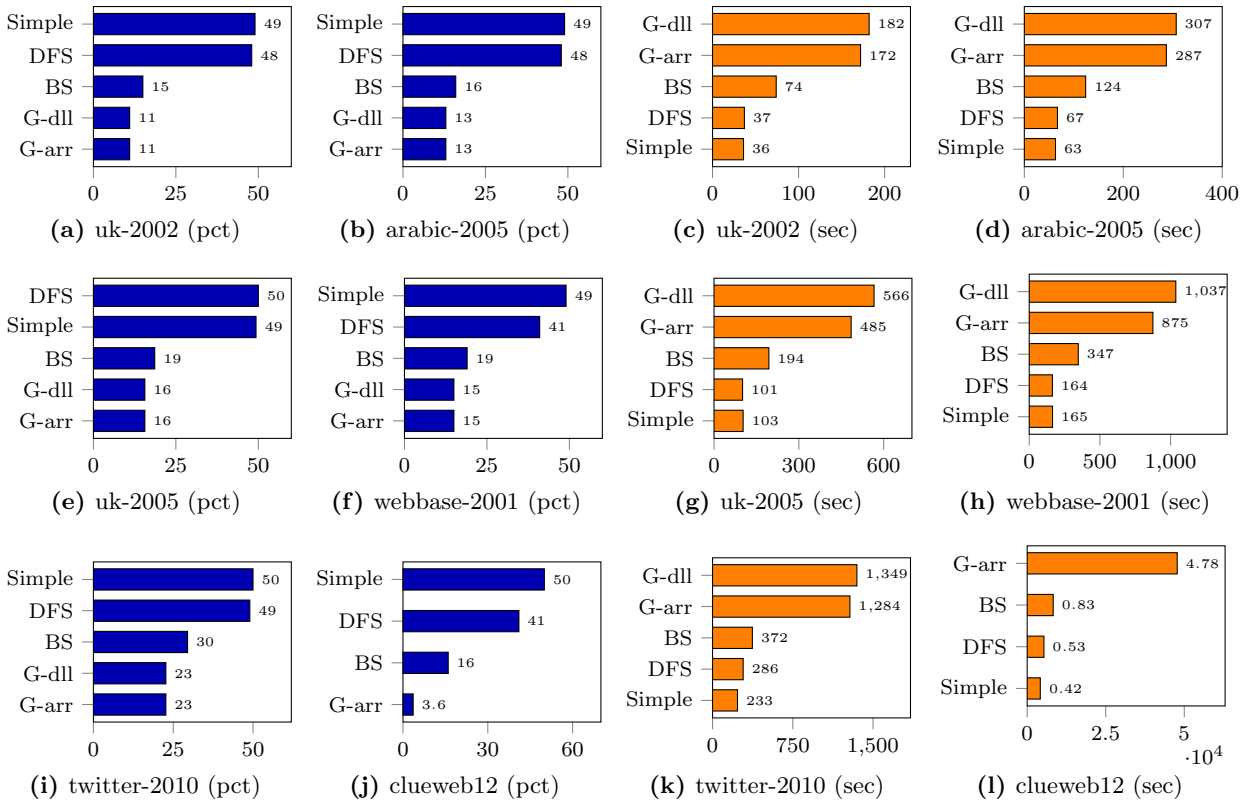


Figure 8: FAS size (left-half) and running time (right-half) for large and extra-large datasets. The resulting FAS size is given as a percentage of the total number of arcs in the corresponding graph.

can be extracted from the δ-values of the top-k vertices. Recall that during the execution of BegerShorFAS the vertices are processed in a random order. However, since the top-k vertices have such large neighbourhoods, we expect the neighbourhood of a top-k vertex u when it is processed to be similar to its initial neighbourhood. It is unlikely that a significant fraction of u's neighbours have been removed from G by the time u is processed due to the sheer number of them. Therefore, we can look at the δ-values of the top-k vertices as an indication of whether or not a large number of arcs will be added to the FAS from such vertices. Note that for the networks considered, the top 1% of vertices by degree comprise, on average, 60% of the arcs of the entire network with a peak of 98% for twitter-2010. Thus, if the δ-values for these vertices are favorable ($|\delta| \gg 0$) then we can expect a small fraction of the top-k vertices arc's to be included in the FAS. In Table 3 we show the percentage of total arcs accounted for by the top 1% of vertices and the distribution of their *skews* defined as the lesser of the indegree and outdegree of a vertex u as a fraction of u's total degree. Note, a small skew corresponds to a large absolute δ-value and a very source/sink-like vertex.

Unlike traditional numerical sorting problems, where there is a total ordering on the data, a difficulty in applying sorting techniques to the FAS problem is a lack of transitivity which sorting algorithms are designed to exploit. In fact, real-world networks are far from exhibiting a total ordering because of their sparsity. Consider the KwikSortFAS algorithm, in which the order of equal vertices is left unaltered. We can expect that in sparse graphs there will be many vertices determined to be equal in each iteration which can lead to poor performance since large subsections of the ordering will not be modified in a meaningful way. This fact is highlighted in Table 3 where the fraction of iterations of KwikSortFAS for which the size of the equal items is greater than 90% of the total vertices in G is presented.

Second, we will discuss how we expect the various approaches to perform relative to each other and compare this to the experimental results.

As mentioned previously, BergerShorFAS and GreedyFAS both function around δ-values, but GreedyFAS has a distinct advantage in that it updates the δ-values during execution. This leads to better decision making by ensuring the processing of the most sink and source-like vertices first. Furthermore, notice that BergerShorFAS always adds some number of arcs to the FAS for non-source/sink vertices. In contrast, GreedyFAS has the potential to avoid adding arcs to the FAS depending on the previous vertices added to the vertex ordering. That is, GreedyFAS may append a non-source/sink vertex u to the ordering for which all outgoing arcs point to vertices to the right of u in the ordering and all incoming arcs point to vertices to the left of u. Thus, we expect GreedyFAS to outperform BergerShorFAS which is exactly what we observe in our experiments.

In comparing GreedyFAS and the sorting-based approaches, notice that GreedyFAS can be considered a 2-sided selection sort. However, GreedyFAS relies less on the transitivity property since it makes local greedy choices. Therefore,

Name	F_{topk}	$s \leq 0.1$	$s \in (0.1, 0.4)$	$s \geq 0.4$	$f \leq 0.1$	$f \in (0.1, 0.4)$	$f \geq 0.4$	L-$\lvert M \rvert$
word_assoc	19.40	0.3208	0.6792	0.0	0.5402	0.3552	0.1348	0.4789
enron	74.30	0.3035	0.4682	0.2298	0.8577	0.0924	0.0557	0.5156
uk-2007	63.42	0.8250	0.1690	0.0060	0.5242	0.3730	0.1277	0.3750
cnr-2000	62.40	0.4104	0.3853	0.2061	0.3779	0.3590	0.3001	0.5059
uk-2002	60.98	0.7086	0.1664	0.1253	0.4088	0.4242	0.2048	-
arabic-2005	79.94	0.5414	0.1296	0.3292	0.4303	0.4359	0.1671	-
uk-2005	41.98	0.2748	0.4491	0.2769	0.3969	0.4238	0.2146	-
webbase-2001	60.16	0.5291	0.2749	0.1977	0.3846	0.3723	0.2872	-
twitter-2010	98.23	0.2049	0.3683	0.4271	0.3143	0.4365	0.2913	-
clueweb12	59.19	0.4920	0.3025	0.2058	0.5249	0.3675	0.1299	-

Table 3: Additional Dataset Statistics. The fraction (in %) of total arcs accounted for by the top-k vertices is given by F_{topk} while the skew of the top-k vertices is given by s. The fraction of vertices added to the FAS in each iteration of BS is given by f. Finally, L-$|M|$ represents the fraction of recursive iterations in KS that have $|M| > 0.9n$.

in sparse networks, we expect GreedyFAS to outperform the other sorting-based approaches which happens to be the case with the exception of the SortFAS* algorithm.

Finally, we investigated the effect of two graph properties related to social networks: power-law degree distribution and the small-world phenomenom. We constructed synthetic networks using iGraph in the R programming language on 10,000 vertices according to the preferential attachment model [8] for power-law degree distribution and the Watts-Strogatz model [34] for the small-world effect. In the Watts-Strogatz model we vary the re-wiring probability from $5-25\%$ and observe a linear decrease in the size of the FAS output by both GreedyFAS and BergerShorFAS. The FAS computed by GreedyFAS ranges from $39-27\%$ and $43-36\%$ for BergerShorFAS. Observe that such a decrease lines up with our intuition since re-wiring leads to larger $|\delta|$-values for the vertices of G. In the preferential attachment model we did not observe any meaningful difference across a wide range of parameter values.

5. THE PROBABILISTIC CASE

Traditionally, much of the investigation into the FAS problem from the practitioner's perspective has focused on the unweighted case. However, large datasets often contain information that is uncertain (probabilistic) in nature. For example, in large social networks, the arc probability may denote the accuracy of a *link prediction* [25], or the influence of one user on another, e.g., in *viral marketing* [19]. Uncertainty can also be injected intentionally for obfuscating the identity of users for privacy reasons [6]. Our confidence in such relations is commonly quantified using probability, and we say that the relation exists with a probability of existence p. In this section, we consider *probabilistic graphs* (also called uncertain graphs), whose arcs are labeled with a probability of existence. Probabilistic graphs have been used extensively in modeling, for example, communication networks, social networks, protein interaction networks, and regulatory networks in biological systems.

More formally, let $\mathcal{G} = (V, E, p)$ be a probabilistic graph, where $p : E \rightarrow [0, 1]$ is the function that assigns a probability of existence to each arc $e \in E$. Following the literature (cf. [22, 24]), we assume that the existence of different arcs are mutually independent events. A probabilistic graph is a probability distribution over 2^m deterministic graphs, each of which is a subgraph of the directed graph (V, E). The set of possible deterministic graphs is called the set of "possible worlds" of $\mathcal{G}$. In a probabilistic graph, the expected number of arcs in a possible world $G = (V, E_G)$ of $\mathcal{G}$ is given by $\mathbb{E}(|E_G|) = \sum_{e \in E} p(e)$. Thus, removing a feedback arc set from a probabilistic graph $\mathcal{G}$ ensures that all possible worlds of $\mathcal{G}$ will be acyclic with the exact weight associated with a particular world's feedback arc set depending on the exact composition of arcs that are realized. Then, we can compute the expected number of arcs in a FAS F as $\sum_{e \in F} p(e)$.

5.1 Algorithms

We investigate adaptations of the most promising class of algorithms from the previous section to apply to probabilistic graphs since, to the best of our knowledge, there are no specialized algorithms for the weighted feedback arc set problem in the literature that run in linear time. We consider those $O(m+n)$ algorithms that present the opportunity to take arc probabilities into consideration during execution. For example, we do not consider SimpleFAS since it has no opportunity to take actions based on arc probabilities. On the other hand, we consider versions of the GreedyFAS and BergerShorFAS algorithms for probabilistic graphs.

5.1.1 Probabilistic Greedy (pG)

Recall that in the standard GreedyFAS approach we compute a delta class, $\delta(u)$, for each vertex $u \in G$. In the unweighted case, by definition, $\delta(u)$ only takes on integer values and therefore we have an exact expression for the number of possible δ-classes. However, in the probabilistic case, a natural extension to the original approach would be to compute $\delta(u)$ as $w^+(u) - w^-(u)$ where $w^+(u)$ and $w^-(u)$ are the sum of the out and in-probabilities of u, respectively. As such, $\delta(u)$ gives the expected difference of the outdegree from the indegree of u.

Unfortunately, the above adaptation of $\delta(u)$ leads to the set of possible δ-class values as being any real value in the range $[3-n, n-3]$. Instead, we introduce the concept of an *approximate δ-class* where we maintain the exact $\delta(u)$ value for each vertex $u \in G$, but place u in the approximate $\delta(u)$-class computed by $\lfloor w^+(u) - w^-(u) \rfloor$. Using the concept of approximate δ-classes we are able to maintain the logic of the original GreedyFAS. However, we must take care when deleting a vertex v from G. In GreedyFAS, a neighbour v of u has its δ-class incremented or decremented when forming $G \setminus u$, whereas in the probabilistic case we only move v to an adjacent δ-class if the probability on the arc (u, v) changes the exact $\delta(v)$ value enough to cross the threshold into a different approximate δ-class.

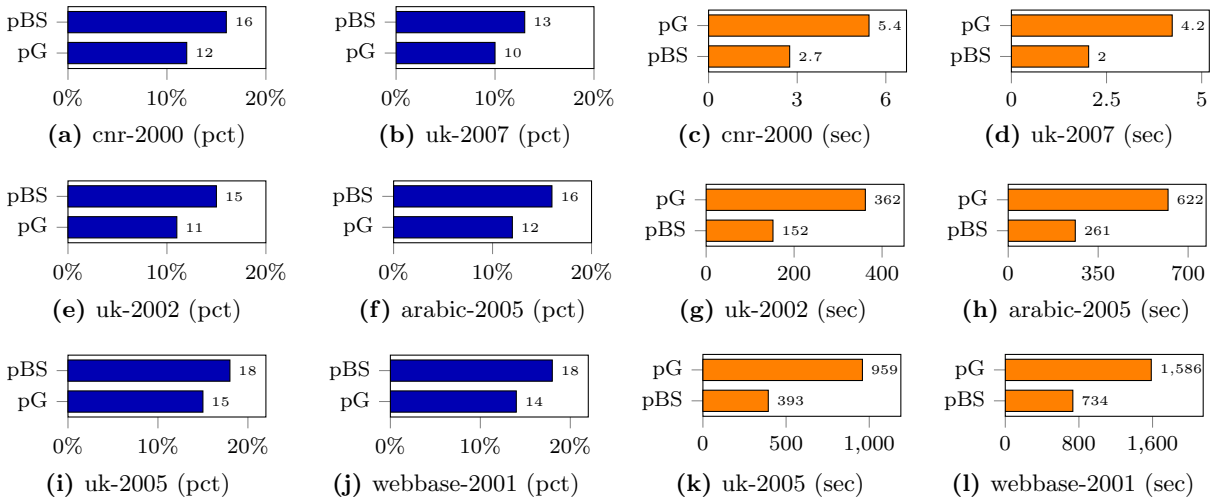


Figure 9: Expected FAS sizes (left-half) and running time (right-half) for probabilistic versions of algorithms. The resulting expected FAS size is given as a percentage of the total number of expected arcs in the corresponding graph.

5.1.2 *Probabilistic Berger-Shor (pBS)*

The probabilistic version of BergerShorFAS follows a natural extension by altering its decision function for updating set F to incorporate the probabilities on the arcs. Here, when processing a vertex u, if the expected number of outgoing arcs (as computed by the sum of probabilities of outgoing arcs) is greater than or equal to the sum of the probabilites of the incoming arcs, then the incoming arcs are removed from G and added to F while the outgoing arcs are removed from G and discarded. If the sum of the probabilities of the incoming arcs is greater, instead the incoming arcs are discarded and the outgoing arcs are added to F. After processing each vertex, the arcs in F form a probabilistic FAS in which the expected number of arcs for a possible world is $\sum_{e \in F} p(e)$.

5.2 Experiments

The measures of effectiveness in the probabilistic case are *expected FAS size* defined as the expected number of arcs in a FAS output by a particular probabilistic FAS algorithm and the efficiency measured in running time, both of which we aim to minimize.

Due to the increased storage size of a probabilistic graph from the requirement of storing arc probabilities in addition to the network structure, we consider the medium datasets and a subset of the large datasets. We construct probabilistic versions of the datasets by randomly assigning probabilities in the range $[0, 1]$ to the arcs of each graph. In Figure 9 (left-half) we plot the ratio of the expected size of the FAS to the expected number of arcs in a possible world, i.e. the sum of the probabilities of the FAS against the sum of the probabilities of the arcs in the graph, while Figure 9 (right-half) shows the running times.

We observe that the adapted algorithms for GreedyFAS and BergerShorFAS see an improvement in terms of the expected FAS size for several datasets compared to the unweighted case. For example, for GreedyFAS, the improvement is around 8% ($= 1 - 12/13$) on cnr-2000 and arabic-2005, 7% ($= 1 - 15/16$) on webbase-2001, and 6% ($= 1 - 15/16$) on uk-2005.

The additional information available leads to the algorithms choosing feedback arc sets that contain low probability arcs resulting in a smaller expected value.

6. RELATED WORK

Up until now, the theory community has focused on achieving the best possible approximation ratios for the FAS problem with less emphasis on the running time of the resulting algorithms. The first approximation algorithm for the FAS problem was given by Leighton and Rao [23] with an approximation ratio of $O(\log^2 n)$ by using an $O(\log n)$ approximation algorithm for balanced cuts. The authors appealed to linear programming techniques to show that the problem can be solved in polynomial time. This approach was improved by Klein, Stein, & Tardos [21] to a $O(m^2 \log m)$ expected time randomized algorithm. The current best known approximation algorithm, due to [14], for computing the minimum feedback arc set has a ratio $O(\log n \log \log n)$ and runs in $O(n^2 M(n) \log^2 n)$ time, where $M(n)$ denotes the complexity of matrix multiplication.

Recent work on the FAS problem has led to advances on tournament graphs (see [12] for a survey). The restricted problem admits a polynomial-time approximation scheme due to Kenyon-Mathieu & Schudy [20]. Furthermore, a subexponential fixed parameter algorithm for the weighted version was given by Karpinski & Schudy [18].

A body of work also exists that investigates various heuristics for computing a minimum feedback arc set. Saab [30] provides a divide-and-conquer heuristic based on graph partitions and strongly connected components. However, the costly subalgorithms required by this approach preclude it from being considered in this paper. Finally, there are the heuristics considered in this paper: GreedyFAS [13], InsertionSortFAS & KwikSortFAS [9], and BergerShorFAS [4].

7. CONCLUSIONS

To conclude, we presented a thorough experimental investigation into the FAS problem. We presented highly optimized implementations of GreedyFAS, BergerShorFAS, and

the sorting-based heuristics. Within each of the three complexity classes of algorithms we observe an overall trend showing a trade off between scalability and quality. We observe an approximate maximum scalability of 300K arcs for the $O(n^2)$ algorithms, 3.5M arcs for the $O(n \log n)$ algorithms, and 50B arcs for the $O(m + n)$ algorithms. We achieve approximate FAS sizes of 3-20%, 23-40%, and 11-17% for the best algorithms of the $O(m + n)$, $O(n \log n)$, and $O(n^2)$ runtime categories, respectively. GreedyFAS and BergerShorFAS provide the best balance between scalability and quality. GreedyFAS is the algorithm that produces always either the smallest or a very close second smallest FAS size while being a fast algorithm in general. In particular our G-arr implementation scales to the biggest dataset we consider, clueweb12, with more than 42 billion arcs.

In addition, we can look to the skew of the top-k vertices as an indication of when the GreedyFAS and BergerShorFAS algorithms are expected to perform favourably. For the sorting-based approaches, we can use the sparsity of G as an indication of the expected performance.

8. REFERENCES

[1] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: Ranking and clustering. *J. ACM*, 2008.

[2] A. Baharev, H. Schichl, and A. Neumaier. An exact method for the min. feedback arc set problem, 2015.

[3] R. Bar-Yehuda, A. Becker, and D. Geiger. Randomized algorithms for the loop cutset problem. *JAIR*, 2000.

[4] B. Berger and P. W. Shor. Approximation algorithms for the maximum acyclic subgraph problem. In *SODA'90*, 1990.

[5] B. Bidyuk and R. Dechter. Cutset sampling for bayesian networks. *JAIR*, 2007.

[6] P. Boldi, F. Bonchi, A. Gionis, and T. Tassa. Injecting uncertainty in graphs for identity obfuscation. *Proceedings of the VLDB Endowment*, 2012.

[7] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW'04*, 2004.

[8] B. Bollobás, C. Borgs, J. Chayes, and O. Riordan. Directed scale-free graphs. In *SODA'03*, 2003.

[9] F. J. Brandenburg and K. Hanauer. Sorting heuristics for the feedback arc set problem. Technical report, University of Passau, Germany, 2011.

[10] C. Budak, D. Agrawal, and A. El Abbadi. Limiting the spread of misinformation in social networks. In *WWW'11*, 2011.

[11] W. Chen, Y. Yuan, and L. Zhang. Scalable influence maximization in social networks under the linear threshold model. In *ICDM'10*, 2010.

[12] T. Coleman and A. Wirth. Ranking tournaments: Local search and a new algorithm. *J. Exp. Algorithmics*, 2010.

[13] P. Eades, X. Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Inf. Process. Lett.*, 1993.

[14] G. Even, J. S. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. In *IPCO'95*, 1995.

[15] W. Gatterbauer, S. Günnemann, D. Koutra, and C. Faloutsos. Linearized and single-pass belief propagation. *PVLDB*, 2015.

[16] X. He, G. Song, W. Chen, and Q. Jiang. Influence blocking maximization in social networks under the competitive linear threshold model. In *SDM'12*, 2012.

[17] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer US, 1972.

[18] M. Karpinski and W. Schudy. Faster algorithms for feedback arc set tournament, kemeny rank aggregation and betweenness tournament. In *ISAAC'10*, 2010.

[19] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *KDD'03*, 2003.

[20] C. Kenyon-Mathieu and W. Schudy. How to rank with few errors. In *STOC'07*, 2007.

[21] P. Klein, S. Plotkin, C. Stein, and E. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM J. Comput.*, 1994.

[22] G. Kollios, M. Potamias, and E. Terzi. Clustering large probabilistic graphs. *TKDE*, 2013.

[23] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *FOCS'88*, 1988.

[24] R. Li, J. X. Yu, R. Mao, and T. Jin. Recursive stratified sampling: A new framework for query evaluation on uncertain graphs. *TKDE*, 2016.

[25] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *JASIST'07*, 2007.

[26] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[27] S. A. Myers and J. Leskovec. Clash of the contagions: Cooperation and competition in information diffusion. In *ICDM'12*, 2012.

[28] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.

[29] K. Perrot and V. T. Pham. Np-hardness of minimum feedback arc set problem on eulerian digraphs and minimum recurrent configuration problem of chip-firing game. *CoRR*, 2013.

[30] Y. Saab. A fast and effective algorithm for the feedback arc set problem. *Journal of Heuristics*, 2001.

[31] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[32] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad. Collective classification in network data. *AI magazine*, 2008.

[33] M. Simpson, V. Srinivasan, and A. Thomo. Clearing contamination in large networks. *TKDE*, 2016.

[34] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998.