Effective Heuristics for Finding Small Minimal Feedback Arc Set Even for Large Graphs

Claudia Cavallaro[1,*], Vincenzo Cutello[1] and Mario Pavone[1]

[1]Department of Mathematics and Computer Science, University of Catania, 95125 Catania, Italy

Abstract

Given a directed graph $G = (V, E)$, we present a heuristic based algorithm to tackle the general Minimum Feedback Arc Problem with the goal of finding good (and often optimal) Feedback Arc Sets which are minimal, i.e. such that none of the arcs can be reintroduced in the graph without disrupting acyclicity. Our algorithm has a good polynomial upper bound which makes it suitable even when dealing with applications on large graphs useful on Big Data problems such as in Social Networks.

Keywords

Minimal Feedback Arc Set, Big Data, Heuristics, Optimization Problem, Experimental Analysis

1. Introduction

The Minimum Feedback Arc Set Problem, along with the related Minimum Feedback Vertex Set Problem, is one of the historical $\mathcal{NP}$-hard problems, specifically $\mathcal{NP}$-complete, listed in the historical paper written by Karp in 1972 [1]. The problem can be formalized as follows: given a directed graph $G = (V, E)$ find a subset of its set of edges, i.e. a subset $F \subseteq E$, whose removal from G makes the graph acyclic. An equivalent formulation of the problem is called the Linear Arrangement problem. In this case, we take as input a directed graph G and we look for an ordering of the vertices of the graph such that the number of forward (respectively backward) arcs, i.e. arcs directed from left to right (respectively right to left) is minimum. The forward arcs or backward arcs make up a Feedback Arc Set, since their removal makes the graph acyclic. Moreover, such a property of forward and backward arcs, proves that any minimum feedback arc set has certainly size at most $\frac{1}{2}|E|$ (see also the algorithm presented in [2]).

The literature on the problem is very vast and lots of heuristics and approximation algorithms have been produced over the span of more than 50 years, ranging from purely mathematical problems to applications in various fields, including, as an example of application to Big Data, large-scale biological systems [3].

We will refer to Kudelic's recent monography [4] for a thorough and quite interesting review on the problem, including a description of the cases for which we do have a polynomial time algorithm, such as for instance planar graphs [5].

ITADATA2023: The 2nd Italian Conference on Big Data and Data Science, September 11–13, 2023, Naples, Italy

*Corresponding author.

claudia.cavallaro@unict.it (C. Cavallaro); cutello@unict.it (V. Cutello); mpavone@dmi.unict.it (M. Pavone)
0000-0003-3938-0947 (C. Cavallaro); 0000-0002-7521-3516 (V. Cutello); 0000-0003-3421-3293 (M. Pavone)

In our work, we will make use of the idea of finding good linear arrangements of the vertices. Such an idea was exploited quite successfully in [6], where the authors developed a linear time algorithm to find a good ordering so to put all backward arcs in the feedback arc set.

We found, in particular, two published papers quite inspirational.

The first one is [7], where the authors investigate and compare several approximation algorithms. Their goal is to compare their efficiency, especially when dealing with Big Data problems such as misinformation removal in Social Networks.

They perform their comparison of a many different approximation algorithms on very large online networks including Twitter, and they conclude that the above mentioned greedy algorithm in [6] performs quite well.

The second is [8] where the authors propose a $\mathcal{O}(|V||E|^4)$–heuristic for the directed FASP, and by empirical validation they achieve an approximation rate of $r \leq 2$ with $r \approx 1.3606$ being a lower bound for the APX–hardness. The computational time is quite high, but the results in terms of optimization are often optimal.

Our goal is to develop a heuristic which is comparable to the one presented in [8] in terms of optimality of results and it is still polynomially fast enough to be applied to Big Data problems such as, for example, the ones discussed in [7].

2. Computing a Feedback Arc Set

We will now describe our approach to the problem. In Section 4 we will show the obtained results and compare them to the results obtained in [8] and [6].

Given a directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ and no self loops, we know that the vertices which participate in any cycle, belong to the same Strongly Connected Component (SCC) of G. It follows that when searching for a Feedback Arc Set (FAS) for the given graph, we can ignore all the edges going from a vertex in one component to a vertex in a different component. Once we find a FAS for each SCC, the union of the found FAS's is a FAS for the entire graph.

To find a FAS for a strongly connected graph or a strongly connected component of a graph, we follow the simple idea to pick a permutation of the set of vertices and delete all the forward edges (1st case) or alternatively (2nd case) all the backward edges, since both sets of arcs so obtained, are a Feedback Arc Set. Algorithms 1 and 2 show how to compute, respectively, the forward edges and backward edges of a vertex, given a permutation of the set of vertices. Both procedures are linear in the size of the graph, i.e. $\mathcal{O}(|V| + |E|)$, since for each vertex we can simply go trough its adjacency list and search for all the vertices which follow (or precede) in the ordering.

Three questions naturally arise from such an approach:

Q1: Is it possible to stop before reaching the last vertex?

Q2: Which particular permutation to choose?

Q3: Is the FAS produced minimal, i.e. is it the case that none of its arcs can be reintroduced into the graph without disrupting acyclicity?

Algorithm 1: Forward edges: $FE(G, L, i)$

```
Inputs : Directed graph $G = (V, E)$,
         permutation of vertices $L$,
         vertex position $i$
Output : Subset $E_f$ of $E$, forward edges
         out of vertex $L[i]$
1 $E_f = []$
2 for $j = i + 1$ to $|L|$ do
3     if $(L[i], L[j]) \in E$ then
4         add $((L[i], L[j])$ to $E_f$
5     end
6 end
7 return $E_f$
```

Algorithm 2: Backward edges: $BE(G, L, i)$

```
Inputs : Directed graph $G = (V, E)$,
         permutation of vertices $L$,
         vertex position $i$
Output : Subset $E_b$ of $E$, backward
         edges in vertex $L[i]$
1 $E_b = []$
2 for $j = i + 1$ to $|L|$ do
3     if $(L[j], L[i]) \in E$ then
4         add $((L[j], L[i])$ to $E_b$
5     end
6 end
7 return $E_b$
```

2.1. Question 1: Checking acyclity

The answer to question $Q1$ is obviously yes. It is not necessary to eliminate all the forward (or backward) edges in order to obtain a FAS. We could check for acyclicity after all the forward or backward edges of a vertex are eliminated and stop when the graph is acyclic. Such a check changes the computational cost and, in particular, since acyclicity can be tested in $\mathcal{O}(|V| + |E|)$, an upper bound for the overall cost is $\mathcal{O}(|V|(|V| + |E|))$. Such an asymptotic extra cost might be compensated by the fact that we may stop, in general, much earlier in the search of a FAS. For instance, it could be the case that eliminating all the edges exiting the first vertex, or the first few vertices, the graph is already acyclic.

2.2. Question 2: Sorting the vertices

In our approach, we do not pick a random permutation of the vertices but we order them according to the 2 most intuitive criteria: out-degree of a vertex, in-degree of a vertex. In turn, considering the two possibility of having increasing or decreasing order, we will have 4 different orderings: decreasing out-degree (do), increasing out-degree (io), decreasing in-degree (di), increasing in-degree (ii). Such ordered lists of vertices, as we experimentally noted, help in stopping much sooner the process of finding a FAS.

For each ordering, in turn, we can produce then 2 different FAS, one eliminating forward edges and the other eliminating backward edges. The initial asymptotic cost of ordering the vertices is $\mathcal{O}(|V| \log |V|)$. If the graph is not sparse, and so $|E| \sim |V|^2$ such an extra cost is absorbed into the $\mathcal{O}(|V| + |E|)$ cost of computing all the in and out-degrees of the vertices.

2.3. Question 3: Producing a Minimal Feedback Arc Set

Given a directed graph G and a set of edges F whose removal makes G acyclic how can we check if F is minimal and, in case it is not, how can we add back some of the edges in the graph?

The simplest way is to go through the edges in F and, one by one, add them to the graph. If, after adding one edge, the graph is no longer acyclic, we remove it again. Every edge needs to be tested only once, since the insertion of other edges will not change the fact that the edge introduces a cycle. Finally, since $1 \leq |F| \leq |E|$, the overall cost of this procedure is $\mathcal{O}(|E|(|V| + |E|))$.

We can, however, improve its effectiveness, by taking into consideration that the edges in F follow the ordering of the vertices. So, if for instance the vertices were ordered in decreasing order of out-degree (do), we know that the first, say k, edges in F are all edges exiting the first vertex and so on.

Experimentally, we saw that the way to add back as many edges as possible is to avoid to reinsert all the edges from a vertex, but try, instead, to reinsert edges from many vertices as possible. More in details, we will go through the list F in many rounds, but at step i, if in that round, h edges were added to the graph, we go from the edge in position i to the edge in position $i + h$. Such a heuristic for adding edges, which we called smartAE and which is formally described in Algorithm 3, ensures that no more than half of the edges related to a specific vertex can be added at any round, and, experimentally, it has given us excellent results.

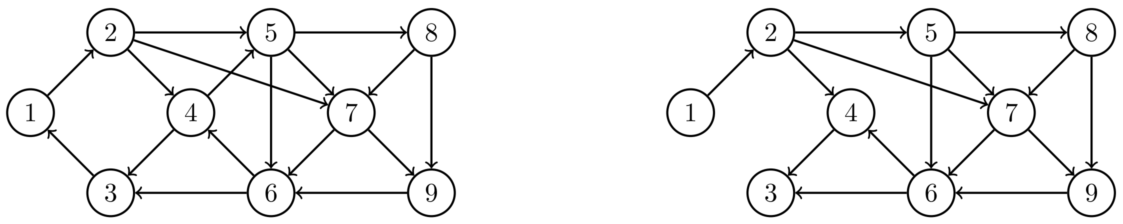
Algorithm 3: Algorithm smartAE

```
Inputs : Acyclic graph $G = (V, E)$, list of arcs $F$, not in $G$
Output: Subset $E$ of $F$, which is a minimal feedback arc set for graph $G$ which, when extended
        with all the arcs in $F \setminus E$, is still acyclic
$AA = []$ ;                                    /* Initialize list of added arcs */
$E = []$ ;                                /* Initialize list of eliminated arcs */
$count = 0$ ;                                          /* Number of added arcs */
while $F$ not empty do
    $count = 0$
    for $i = 1$ to $|F|$ do
        pick edge $e = F[i + count]$ and add it to $G$
        if $G$ is acyclic then
            add $e$ to $AA$
            $count = count + 1$
        else
            remove $e$ from $G$ and add $e$ to $E$
        end
    end
end
return minimal feedback arc set $E$
```

Finally, let us briefly comment on a different approach to the problem of finding a minimal FAS, given the set of edges that were removed from the graph to make it acyclic.

Clearly, given an acyclic graph we can find a topological sort of its vertices, i.e. a sorting such that all edges of the graph go from left to right (forward edges). So, it seems quite logical that the first approach to add back edges to the graph, is to find a topological sorting of its vertex and add all the edges which are forward edges. None of these edges will create a new cycle.

We did test this approach and found out that, although it is faster than smartAE, its perfor-



(a) Strongly connected graph (b) Acyclic after two arcs removal

Figure 1: Example graph.

mance is quite poor. In order to add more edges, we had to call smartAE on the remaining set of edges, not added because of the ordering in the topological sort.

A small advantage, as we just mentioned, is the fact that it reduces the size of the set on which to call smartAE, but a disadvantage, which causes poor results, is due to the fact that the topological sort approach ignores the order in which the edges were eliminated, which is the reason why smartAE gives us very good results.

2.4. Simple Example

Before describing our set of heuristics, let us work through a simple example and consider the graph in Figure 1. The graph contains a simple cycle $1-2-4-5-8-7-9-6-3-1$ which involves all the vertices, so it is strongly connected. Moreover, it has 2 edge disjoint cycles, namely $1-2-4-3-1$ and $5-7-6-4-5$, thus any minimum FAS must contain at least 2 arcs. Indeed, if we remove the arcs $(1,2)$ and $(4,5)$ we obtain an acyclic graph.

If we sort the vertices following the 4 orderings, we have (degree between parentheses):

$$\begin{array}{rcl} do & : & 2(3),5(3),4(2),6(2),7(2),8(2),1(1),3(1),9(1) \\ io & : & 1(1),3(1),9(1),4(2),6(2),7(2),8(2),2(3),5(3) \\ di & : & 6(3),7(3),3(2),4(2),5(2),9(2),1(1),2(1),8(1) \\ ii & : & 1(1),2(1),8(1),3(2),4(2),5(1),9(2),6(3),7(3) \end{array}$$

For each of those 4 orderings, the algorithm will compute the two lists of forward edges and backward edges and it will pick the shortest one.

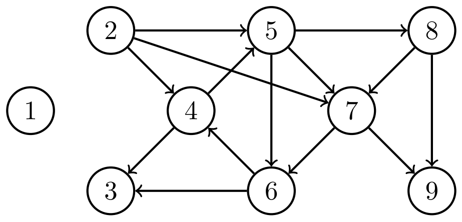
In particular, following the order io, and eliminating all the forward edges we have:

- the arcs exiting 1, 3, and 9 are eliminated, obtaining the graph in Figure 2, case (a);
- since the graph is not acyclic, we eliminate the arcs exiting 4, i.e. $(4,3),(4,5)$.

The graph is now acyclic and the list of eliminated arcs, in the order of elimination, is $E_f = [(1,2),(3,1),(9,6),(4,3),(4,5)]$.

The algorithm smartAE will try to insert them into the graph again, so that the ones remaining will be a minimal feedback arc set, i.e. no other arc could be inserted without disrupting acyclicity.

At step 1, smartAE adds the first edge $(1,2)$ to the graph and it remains acyclic. So, $(1,2)$ is added to the list AA and the value of count is now 1.


(a) After removal of arcs exiting vertices $1, 3, 9$

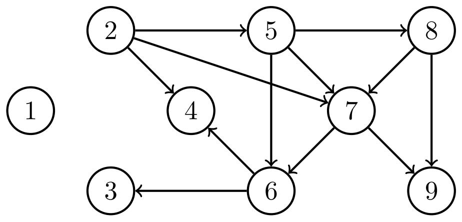

(b) Acyclic after removals of arcs exiting 4

Figure 2: Following the io ordering.

At step 2, smartAE picks the edge in position $2 + count = 3$, i.e. the edge $(9, 6)$. The edge is added to the graph which remains acyclic. So, $(9, 6)$ is added to the list AA and $count = 2$.

At step 3, smartAE picks the edge in position $3 + count = 5$, i.e. the edge $(4, 5)$. It adds it to the graph and now the graph is no longer acyclic. So $(4, 5)$ is added to the list EA. We have reached the end of the for loop, the arcs in AA and EA are removed from E_f which now has 2 arcs left, i.e. $E_f = [(3, 1), (4, 3)]$. The for loop restarts with $count = 0$, smartAE picks the arc $(3, 1)$, it adds it to the graph and it creates a cycle. So, it removes it from the graph and adds it to the list EA and now $EA = [(4, 5), (3, 1)]$. The variable $count$ remains equal to 0, so smartAE moves to the next and last element $(4, 3)$. It adds it to the graphs and the graph remains acyclic (Figure 1 (b) is exactly the graph obtained after smartAE adds the edges).

3. Our heuristics Ordered Feedback Arc Set (OFAS)

We will now give a complete description of our approach to the problem.

Given a directed graph $G = (V, E)$ our overall reasoning is the following:

- compute all the Strongly Connected Components (SCC) of G, and concentrate just on the components with at least 2 vertices.

For each SCC,

- we sort the vertices in 4 different ways: decreasing order of out-degree (do), increasing order of out-degree (io), decreasing order of in-degree (di), increasing order of in-degree (ii),
- for each of the above orderings and following the vertices in the ordering, we have 2 possible stopping criteria which characterize our 2 different heuristics:
  0) Stopping criterion is "end of the list of vertices", i.e. we go through the list and vertex by vertex we delete all forward edges and store them into a list E_f. Analogously, we go through the list again and delete all backward edges and store them into a list E_b.
  1) Stopping criterion is "the graph acyclicity", i.e. we go through the list and vertex by vertex we delete all forward edges and store them into a list E_f. However, vertex

by vertex, we remove its forward edges from the graph and check whether or not the graph has become acyclic, in which case we stop. Again, removed edges are stored into the list E_f. Analogously, for the backward edges. When we stop, all the removed backward edges are stored into the list E_b.

- For each of these two cases, which will give us two different heuristics *OFAS0*, *OFAS1*, the smallest of the two sets E_f and E_b (they are both Feedback Arc Sets) will be passed on to the procedure *smartAE*, which will try to add back to the component as many edges as possible and return a minimal Feedback Arc Set.

In Section 4 we will compare the obtained results to the results obtained in [8] and [6].

3.1. The basic minimal ordered FAS

The pseudo-code for the basic heuristic Ordered Feedback Arc Set *OFAS0*, is described formally in Algorithm 4.

In Table 1, we can see that *OFAS0* produces always better results than GR, but it clearly is more computationally expensive, due the computational cost of *smartAE*, and thus globally is $\mathcal{O}(|F|(|V|+|E|))$ where F is the smallest between the set of forward edges and the set of backward edges. Thus, worst case scenario gives us the asymptotic complexity $\mathcal{O}(|E|(|V|+|E|))$.

Algorithm 4: Heuristic OFAS0

```
Inputs : Strongly connected graph $G = (V, E)$, permutation $P$ of vertices in $V$
Output: Minimal FAS
1  $n = |V|, m = |E|, P = \{p_1, \dots, p_n\}$
2  $E_f = [], E_b = []$ ;                                  /* Initialize lists */
3  $L = []$
4  for $i = 1, 2, \dots, n$ do
5      $L = FE(G, P, i)$ append $L$ to $E_f$
6      $L = BE(G, P, i)$ append $L$ to $E_b$
7  end
8  if $|E_f| < |E_b|$ then
9      Remove edges in $E_f$ from $E$
10     return $smartAE(G, E_f)$
11 else
12     Remove edges in $E_b$ from $E$
13     return $smartAE(G, E_b)$
14 end
```

3.2. Acyclicity test and the heuristic *OFAS1*

The pseudo-code of the improved version of Ordered Feedback Arc Set *OFAS1*, is formally described in Algorithm 5. To figure out its computational complexity we start from $\mathcal{O}(|E|(|V|+|E|))$ which is the complexity of *OFAS0* and add to it the complexity of checking if the graph is acyclic after each vertex which is taken into consideration. Thus, worst case scenario, is $\mathcal{O}(|E|(|V|+|E|)+|V|(|V|+|E|))$ which gives us, again, $\mathcal{O}(|E|(|V|+|E|))$.

Algorithm 5: Heuristic OFAS1

```
Inputs  : Strongly connected graph $G = (V, E)$, permutation $P$ of vertices in $V$
Output: Minimal FAS
$n = |V|, m = |E|, P = \{p_1, \ldots, p_n\}$
$G_1 = (V_1, E_1)$ copy of $G = (V, E)$
$E_f = [], E_b = []$ $L = []$ ;                    /* Initialize lists */
for $i = 1, 2, \ldots, n$ do
    $L = FE(G, P, i)$
    append $L$ to $E_f$ and remove its edges from $E$
    if $G$ is acyclic break
end
for $i = n, \ldots, 1$ do
    $L = BE(G_1, P, i)$
    append $L$ to $E_b$ and remove its from $E_1$
    if $G_1$ is acyclic break
end
if $|E_f| < |E_b|$ then
    return $smartAE(G, E_f)$
else
    return $smartAE(G_1, E_b)$
end
```

3.3. Putting it all together: *minOFASx*

We can now put together the two different heuristics and produce our main Algorithms to produce a Minimal Feedback Arc Set. We have 2 different algorithms, according to the choice we make: *minOFAS0*, *minOFAS1*. Algorithm 6, shows the way they both work, by simply referring to *minOFASx* where, obviously, $x = 0, 1$. From what discussed in Subsection 3.2, we clearly have that the overall asymptotic complexity of Algorithm 6 is also $\mathcal{O}(|E|(|V| + |E|))$.

4. Results

In order to understand the efficacy of our algorithm, we ran it on the same test cases used in [8].

The Minimum FAS Problem has been extensively applied to circuit testing [9] and to this end an ISCAS circuit testing dataset is available at https://github.com/alidasdan/graph-benchmarks. The results and comparisons are shown in Table 1.

Another very interesting and useful set of tests, was introduced and described in [10], also available at the url above cited, where one can find much larger graphs. Following [8], we ran our algorithms on three of them, and the relative results and comparisons are shown in Table 2.

In both tables, GR refers to the Algorithm in [6] and Tight to the Algorithm [8]. For all the tests, except for 1 in Table 1 and 2 in Table 2, the size of the minimum FAS is known and it is shown in column "Min". Column SCC refers to the number of Strongly Connected Components of size at least 2. By inspecting both tables, we can see that *minOFAS1* outperforms GR in all instances.

Algorithm 6: Algorithm minOFASx

```
Inputs :Directed graph $G = (V, E)$
Output:Minimal Feedback Arc Set
Compute set $SCC = \{H_1, H_2, \dots, H_k\}$ of Strongly Connected Components of $G$ with at least 2
  vertices.
$Total = 0$ ;                                /* Total amount of removed edges */
$mFAS = []$ ;                                     /* Minimal Feedback Arc Set */
for $i = 1, 2, \dots, k$ do
    Let $V_i$ be set of vertices of $H_i$ and $E_i$ the set of edges of $H_i$
    Compute:
    $L_{do}$ ;            /* list $V_i$ sorted in decreasing order of out-degree */
    $L_{io}$ ;            /* list $V_i$ sorted in increasing order of out-degree */
    $L_{di}$ ;             /* list $V_i$ sorted in decreasing order of in-degree */
    $L_{io}$ ;             /* list $V_i$ sorted in increasing order of in-degree */
    $E_{do}$ =OFASx $(H_i, L_{do})$
    $E_{io}$ =OFASx $(H_i, L_{io})$
    $E_{di}$ =OFASx $(H_i, L_{di})$
    $E_{ii}$ =OFASx $(H_i, L_{ii})$
    $M = min(|E_{do}|, |E_{io}|, |E_{di}|, |E_{ii}|)$
    Total=Total+M
    $F$ = smallest of $\{E_{do}, E_{io}, E_{di}, E_{ii}\}$
    append $F$ to $mFAS$
end
return $mFAS$
```

Let us now compare the results of *minOFAS1* to the results obtained by the Algorithm Tight. We remind that, asymptotically, *minOFAS1* is $\mathcal{O}(|E|(|V|+|E|))$ whereas Tight is $\mathcal{O}((|V||E|)^4)$. The columns with the results of our two methodology also include, for completeness of information, the execution time on an Intel Core i7, 2.8 GHz, with 16 GB of RAM. All tests were performed in Python 3 language. Moreover, the best of the two methodologies is in boldface and, when they both give the same result, only the faster one is in boldface.

Out of the first 33 cases, we can see that in 5 cases Tight has a better result, in 1 case *minOFAS1* has a better result. In all other 27 cases the results are the same. In other words, even though our algorithm has a much lower asymptotic cost, it almost always reaches the same results.

As a final note, although not shown in the results table, the ordering which in the majority of the cases gave us the best result was *do*, i.e. decreasing order of out-degree. We intend to perform a full statistical analysis of the contributions of the 4 orderings in future work.

5. Conclusions and future work

We have proposed a fast algorithm to find approximate (if not optimal) solutions to the Minimum FAS Problem, even for very large graphs, and producing a minimal FAS, i.e. a minimal set of arcs whose removal renders the graph acyclic and such that no other arc can be added to the graph without violating the aciclycity property. We also showed how our algorithm is very much competitive with the algorithm presented in [8] and *loses* slightly only in very few cases.

Table 1
Comparing algorithms

	ISCAS Code	Vertices-Edges	SCC	Min	GR	Tight	minOFAS0	minOFAS1
1	s27	55-87	1	2	2	2	**2: 610ms**	2: 832ms
2	s208	83-119	5	5	5	5	5: 267ms	**5: 193ms**
3	s420	104-178	1	1	1	1	1: 265ms	**1: 182ms**
4	mm4a	170-454	2	8	16	8	8: 296ms	**8: 196ms**
5	s382	273-438	6	15	29	15	16: 317ms	**15: 257ms**
6	s344	274-388	6	15	23	15	16: 362ms	**15: 254ms**
7	s349	278-395	6	15	24	15	16: 1.5s	**15: 241ms**
8	s400	287-462	6	15	28	15	17: 327ms	**15: 221ms**
9	s526n	292-560	15	21	29	21	21: 336ms	**21: 247ms**
10	mult16a	293-582	1	16	23	16	20: 482ms	**16: 362ms**
11	s444	315-503	6	15	20	15	18: 884ms	**15: 242ms**
12	s526	318-576	15	21	31	21	22: 363ms	**21: 309ms**
13	mult16b	333-545	15	15	22	15	15: 335ms	**15: 232ms**
14	s641	477-612	1	11	16	11	12: 498ms	**11: 325ms**
15	s713	515-688	1	11	16	11	11: 556ms	**11: 303ms**
16	mult32a	565-1142	1	32	45	32	45: 1.2s	**32: 1.15s**
17	mm9a	631-1182	11	27	29	27	27: 590ms	**27: 546ms**
18	s838	665-941	32	32	37	32	**32: 450ms**	32: 520ms
19	s953	730-1090	1	6	11	6	8: 606ms	**7: 550ms**
20	mm9b	777-1452	10	26	31	27	28: 1.25s	**26: 798ms**
21	s1423	916-1448	6	71	112	71	79: 7.34s	**71: 2.9s**
22	sbc	1147-1791	5	17	21	17	19: 557ms	**17: 500ms**
23	ecc	1618-2843	57	115	137	115	120: 1.22s	**115: 1.39s**
24	ph_decoder	1671-3379	31	55	64	55	56: 1.98s	**55: 1.93s**
25	da_receiver	1942-3749	30	83	123	83	87: 4.86s	**83: 6s**
26	mm30a	2059-3912	2	60	62	60	61: 5.35s	**60: 2.29s**
27	parker1986	2795-5021	33	178	313	178	195: 1m 20s	**178: 1m 36s**
28	s5378	3076-4589	1	30	75	30	35: 20.2s	**34: 8.62s**
29	s9234	3083-4298	21	90	163	91	92: 27.3s	**91: 37.4s**
30	bigkey	3661-12206	112	224	224	224	**224: 1.51s**	224: 1.57s
31	dsip	4079-6602	2	-	165	153	176: 22.7s	**159: 24.1s**
32	s38584	20349-34562	1	1080	1601	1080	1178: 1h18m44s	**1089: 1h21m52s**
33	s38417	24255-34876	437	1022	1638	1022	1074: 6m9s	**1023: 23m8s**

Table 2
Comparing algorithms

	IBM Code	Vertices-Edges	SCC	Min	GR	Tight	minOFAS0	minOFAS1
1	ibm01	12752-36048	6	-	3254	1761	1917: 1h24m28s	**1840: 53m13s**
2	ibm02	19601-57753	1	-	5726	3820	3922: 2h44m20s	**3837: 2h41m21s**
3	ibm05	29347-98793	93	4769	5979	4769	4770 2h8m10s	**4769 2h49m53s**

So, we think that our *minOFAS1* represents a very good balance between the ability to reach a minimum value (or close to it) and the possibility of application to problems of high dimensions, like the ones described in [7].

One of the weaknesses of our algorithm, which we intend to address in the future, is that it performs a lot better when dealing with scale free graphs. Obviously, for graphs whose vertices have all the same degree, such as for instance tournament graphs, our ordering based solely on both in-degrees and out-degrees might not give us good results, except when either just in-degrees or just out-degrees in the overall distribution differentiate significantly the vertices.

By recalling the steps of our algorithms described in Section 3, it is clear that to speed up the computation time, we could have 8 parallel threads working on each SCC and the minimum of the 8 results would be taken into account. We did not perform any tests on it but it is clear how our algorithm lends itself easily to a parallel/distributed implementations.

As a future research project, we also intend to work on extending our algorithm by considering Strongly Connected Components during the arc elimination phases. We know that edges connecting vertices in different Strongly Connected Components, do not participate in any cycle. Therefore, if, after having eliminated for instance all the forward edges of the first i vertices, the initial strongly connected graph (or component) has been broken into several Strongly Connected Components, any edge exiting from the $(i+1)$-th vertex and entering a vertex in a different Strongly Connected Component remains in the graph. Thus, after having eliminated all the edges outgoing from (or incoming to) the first vertex in the permutation, instead of checking whether or not the graph is acyclic, as suggested in Subsection 2.1, we could compute from that moment on the Strongly Connected Components of the graph and only eliminate the edges connecting vertices within the same component. The asymptotic cost of such a check is still $\mathcal{O}(|V|+|E|)$ as for the acyclicity check, however, the constant factor involved is certainly higher and for graphs with thousands or more vertices and edges, as it is the case for graphs representing Big Data relations, the heuristic, in this simple form, becomes very slow. We have already tested it and obtain some slight improvements. For instance, for test case ibm01 we were able to obtain a minimal FAS of size 1803, which is a significant improvement with respect to the 1840 obtained the *minOFAS1*. So, we are planning to pursue such an extension, try to optimize its running time and generate test cases of known minimum FAS using the algorithm introduced in [11].

Another line of research we intend to follow, is related to the work done in [12, 13, 14], where the Minimum Feedback Vertex Set was tackled by means of population based, evolutionary metaheuristics, even in the case of very large instances. We are confident that such an approach could lead to very good results for the Minimum FAS Problem as well.

Acknowledgments

This research is supported by the project Future Artificial Intelligence Research (FAIR) – PNRR MUR Cod. PE0000013 - CUP: E63C22001940006
We wish to thank the anonymous referees for their very valuable comments and suggestions.

References

[1] R. M. Karp, Reducibility among Combinatorial Problems, Springer US, Boston, MA, 1972, pp. 85–103. doi:10.1007/978-1-4684-2001-2_9.

[2] B. Berger, P. W. Shor, Approximation alogorithms for the maximum acyclic subgraph problem, in: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '90, Society for Industrial and Applied Mathematics, USA, 1990, p. 236–243.

[3] N. Soranzo, F. Ramezani, G. Iacono, C. Altafini, Decompositions of large-scale biological systems based on dynamical properties, Bioinformatics 28 (2011) 76–83. doi:10.1093/bioinformatics/btr620.

[4] R. Kudelić, Feedback Arc Set: A History of the Problem and Algorithms, Springer Nature, 2022.

[5] C. L. Lucchesi, D. H. Younger, A minimax theorem for directed graphs, Journal of the London Mathematical Society s2-17 (1978) 369–374. doi:https://doi.org/10.1112/jlms/s2-17.3.369.

[6] P. Eades, X. Lin, W. Smyth, A fast and effective heuristic for the feedback arc set problem, Information Processing Letters 47 (1993) 319–323. doi:https://doi.org/10.1016/0020-0190(93)90079-O.

[7] M. Simpson, V. Srinivasan, A. Thomo, Efficient computation of feedback arc set at web-scale, Proc. VLDB Endow. 10 (2016) 133–144. doi:10.14778/3021924.3021930.

[8] M. Hecht, K. Gonciarz, S. Horvát, Tight localizations of feedback sets, ACM J. Exp. Algorithmics 26 (2021). doi:10.1145/3447652.

[9] A. Dasdan, Experimental analysis of the fastest optimum cycle ratio and mean algorithms, ACM Trans. Des. Autom. Electron. Syst. 9 (2004) 385–418. doi:10.1145/1027084.1027085.

[10] C. J. Alpert, The ispd98 circuit benchmark suite, in: Proceedings of the 1998 International Symposium on Physical Design, ISPD '98, Association for Computing Machinery, New York, NY, USA, 1998, p. 80–85. doi:10.1145/274535.274546.

[11] Y. Saab, A fast and effective algorithm for the feedback arc set problem, Journal of Heuristics 7 (2011) 235–250. doi:10.1023/A:1011315014322.

[12] V. Cutello, F. Pappalardo, Targeting the minimum vertex set problem with an enhanced genetic algorithm improved with local search strategies, in: D.-S. Huang, V. Bevilacqua, P. Premaratne (Eds.), Intelligent Computing Theories and Methodologies, Springer International Publishing, Cham, 2015, pp. 177–188.

[13] V. Cutello, M. Oliva, M. Pavone, R. A. Scollo, An immune metaheuristics for large instances of the weighted feedback vertex set problem, in: 2019 IEEE Symposium Series on Computational Intelligence (SSCI), 2019, pp. 1928–1936. doi:10.1109/SSCI44817.2019.9002988.

[14] V. Cutello, M. Oliva, M. Pavone, R. A. Scollo, A hybrid immunological search for the weighted feedback vertex set problem, in: N. F. Matsatsinis, Y. Marinakis, P. Pardalos (Eds.), Learning and Intelligent Optimization, Springer International Publishing, Cham, 2020, pp. 1–16.