Improvement and evaluation of a heuristic method for the minimum feedback arc set problem

Jure Pustoslemšek
jp76466@student.uni-lj.si
Faculty of Computer and Information Science,
University of Ljubljana
Večna pot 113
SI-1000 Ljubljana, Slovenia

Ema Črne
ec51731@student.uni-lj.si
Faculty of Computer and Information Science,
University of Ljubljana
Večna pot 113
SI-1000 Ljubljana, Slovenia

Nejc Rihter
nr5256@student.uni-lj.si
Faculty of Computer and Information Science,
University of Ljubljana
Večna pot 113
SI-1000 Ljubljana, Slovenia

ABSTRACT

This paper addresses the problem of finding minimal feedback arc sets in directed graphs, a critical issue in various domains such as computational biology, scheduling and network analysis. We implement, analyse and improve a heuristic approach proposed by Cavallaro et al. [2]. Our improved method reuses their heuristic method for reducing solution size and uses other established techniques from both exact and approximate algorithms to speed up the algorithm. The implementation makes use of a fast network analysis library for additional speed-up. We also describe a simple parallel version of the algorithm and its potential capabilities.

KEYWORDS

Directed graphs, minimum feedback arc set, NP-hard problems, combinatorial optimization, heuristic methods

1 INTRODUCTION

Directed graphs are a fundamental tool in network theory. They are widely used to model systems where the direction of relationships between entities is crucial. A feedback arc set (*abbr. FAS*) is a set of edges in a directed graph such that removing those edges from the graph makes it acyclic. While the entire set of vertices V can trivially serve as a feedback arc set, the challenge lies in finding a minimal feedback arc set, i.e. a feedback arc set with the smallest number of edges.

The problem of finding a minimal FAS is NP-hard. The decisional version of the problem, that is finding a FAS of a certain size, is one of the first known NP-complete problems [6]. As such, it is believed that there exists no algorithm that can solve it in polynomial time. Additionally, the problem is also very challenging to approximate. There is no known algorithm with a constant bound on the approximation ratio, making it an APX-hard [5] problem. The problem is complementary to the maximum acyclic subgraph problem and there is a natural reduction to the linear arrangement problem [2].

In this work, we focus on the implementation and improvement of a heuristic algorithm for finding a minimal FAS, as described by Cavallaro et al. [2]. We shall refer to this algorithm as the original algorithm. The algorithm begins by identifying strongly connected components (*abbr. SCC*) in the input graph G. Since any cycle is guaranteed to belong to a single SCC [3], we are able to split G into SCCs and run the rest of the algorithm on each SCC separately, collecting partial solutions into a single list to form a full solution. For the remainder of the algorithm, we assume that G is strongly connected. Two empty lists E_f and E_b are initialized to hold the forward and backward edges respectively. Vertices are than ordered according to some rule. For each vertex, we identify the forward edges and add them to E_f, thereby removing these edges from the graph. Once the graph becomes acyclic during this process, iteration stops. We then iterate in reverse order to identify the backward edges, adding them to E_b and removing them from G. Once the graph becomes acyclic, iteration stops.

To improve the solution, we apply Algorithm 1, also called *smartAE* to the smaller of the two sets, E_f or E_b. This key component aims to reduce the size of the found FAS by reintroducing the removed edges in a balanced way while avoiding creating cycles in the graph.

Algorithm 1 The smartAE heuristic

```
procedure SMARTAE(graph $G$, edge list $F$)
    $AE \leftarrow []$
    while $F$ not empty do
        $PE \leftarrow []$, $count \leftarrow 0$, $i \leftarrow 0$
        while $i + count < |V(G)|$ do
            $e \leftarrow F[i + count]$
            add $e$ to $PE$, add $e$ to $G$
            if $G$ is acyclic then
                add $e$ to $AE$
                $count \leftarrow count + 1$
            else
                remove $e$ from $G$
            end if
            $i \leftarrow i + 1$
        end while
        remove all of $PE$ from $F$
    end while
    return $AE$
end procedure
```

The *smartAE* heuristic begins with an acyclic graph G and a list of edges F, which were removed from G. An empty list AE is initialized to store edges that can be successfully reintroduced into the graph without creating a cycle. We iterate through all of the edges in F, where instead of sequentially reintroducing each edge from F, the algorithm employs a counter *count* to strategically skip over edges. This approach ensures that we reinsert edges from as many vertices as possible. During each iteration, a temporary list PE tracks the edges being tested. If adding an edge does not create a cycle, it is added to *AE* and *count* is incremented. Overall time complexity of the entire algorithm is $O(|E|(|V|+|E|))$.

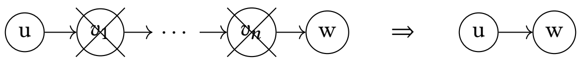
2 IMPROVEMENTS

In this section, we outline our improvements of the original algorithm. Improvements are grouped into four categories: size reduction, vertex ordering strategies, forward/backward edge removal and acyclicity checks.

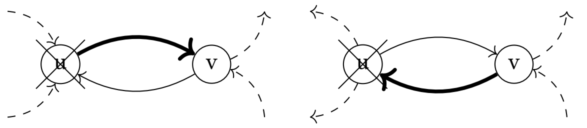
2.1 Size reduction

To reduce the size of the input graph G, we apply reduction rules based on a more general method by Baharev et al. [1]. The general method removes edges inside and on the boundary of an induced subgraph H with the following property: the size of a minimal FAS equals the upper bound of the size of the smallest edge set F whose removal breaks all cycles in G with vertices in H. In this case, we remove edges in F from G. We implemented a few straightforward rules for eliminating some simple and common patterns - we applied the following rules in rounds until a round produces no further reduction in graph size.

(1) For every self-loop e, i.e. an edge with the same entering and exiting vertex, add e to the solution.
(2) For every directed path $uv_1 \dots v_n w$ where $v_1, \dots v_n$ have in-degree 1 and out-degree 1, delete $v_1, \dots v_n$ and add the edge uw to G.



(3) For every 2-cycle uv where u has out-degree 1, delete u and add uv to the solution.
(4) For every 2-cycle uv where u has in-degree 1, delete u and add vu to the solution.



Having reduced the graph using the rules described above, we computed the strongly connected components (SCCs) as in the original algorithm. However, after computing the SCCs, we applied a technique described by Park and Akers [8], in which each SCC is further divided into its biconnected components. This efficiently breaks up the graph into even smaller strongly connected subgraphs. As biconnected components are traditionally defined for undirected graphs, we treat the SCCs as undirected to compute these components. The remainder of the algorithm is then applied to these biconnected components. Throughout this article, we will refer to these components as the SCCs.

2.2 Vertex ordering strategies

The original algorithm uses four vertex orderings in their experiments: in-degree and out-degree, both in increasing and decreasing order. We adopt all these orderings and add five more orderings. The first two are based on the difference of the in-degree ($d^-(v)$) and out-degree ($d^+(v)$) and two more are based on the difference of their degrees and their ratio. We also included a random ordering for comparison.

$$\text{degdiff}(v) = \max \left| d^+(v) - d^-(v), d^-(v) - d^+(v) \right| \tag{1}$$

$$\text{degratio}(v) = \max \left(\frac{d^-(v)}{d^+(v)}, \frac{d^+(v)}{d^-(v)} \right) \tag{2}$$

2.3 Forward/backward edge removal

The forward edge removal phase proceeds as follows. For each vertex v in G, ordered according to the chosen ordering, we remove all edges exiting v and entering vertices that follow v in the ordering, then we check if G has become acyclic. If G is acyclic, we end the edge removal phase. In the worst case, we remove forward edges of every vertex, at which point G is guaranteed to be acyclic, since the current ordering has become a topological ordering. The backward edge removal phase is almost identical, the only difference being that it removes edges that precede v in the ordering. The original algorithm chooses the smaller of E_f and E_b and only performs *smartAE* on that. We challenged this approach and performed *smartAE* on both, only then taking the smaller as the solution.

The purpose of the proposed improvement is to calculate an SCC decomposition before forward/backward edges of a vertex are removed. Then, instead of adding all edges to the solution, we skip edges that exit one SCC and enter another. Such an edge cannot be a part of a cycle. A cycle would imply that vertices from both SCCs are reachable from one another, but then all those vertices would be in the same SCC. As before, the edge removal phase terminates when G becomes acyclic. This improvement is expected to increase the running time but reduce the size of the solution.

2.4 Acyclicity checks

The most time-consuming aspect of the algorithm is restoration and removal of edges during *smartAE* process. A standard method for checking acyclicity is to find a topological ordering of the graph's vertices, i.e. a ordering of vertices such that all out-neighbors of a vertex x appear after x. If the graph contains a cycle, then it does not have a topological ordering [3]. This ordering provides an efficient way to check if inserting an edge would create a cycle. This ordering provides an efficient way to check if inserting an edge would create a cycle. If it's a backward edge, i.e. it ends in a vertex that comes before its starting vertex, it forms cycles.

During *smartAE* we check acyclicity after restoring each edge. If the resulting graph has cycles, we immediately remove that edge, making the graph acyclic again. We avoid many calculations of a new topological ordering by exploiting this sequence of operations. Right before the use of *smartAE* at the end of forward/backward edge removal, an acyclic check is performed. During the check, we calculate a topological ordering and, if successful, store it in the variable T_c. We are then able to use this variable to make subsequent acyclicity checks trivial. When we restore an edge, we check if it is a backward edge in the T_c. If it is, the graph is no longer acyclic. We move the topological ordering from T_c into a background variable T_p and unset T_c. When we remove the last restored edge, we move the ordering in T_p back into T_c. We thus avoid a recalculation in the next acyclicity check.

3 EXPERIMENTAL RESULTS

We evaluated our implementation and improvements on the ISCAS circuit benchmark dataset [4] and on a selection of directed networks from the SNAP Large Network Dataset Collection [7]. We experimented with different configurations of the algorithm to assess the effect of each option on both speed and solution size. The ISCAS dataset primarily served as a speed benchmark. The largest instance in this dataset took about 5 to 10 minutes to complete, depending on the configuration; while the implementation of the original algorithm, which used four orderings instead of seven, took around 90 minutes. This gave us confidence to apply our algorithm on larger and more diverse networks from the SNAP dataset.

Our results from testing our implementation on the SNAP dataset are presented in Table 1. To illustrate the complexity and size of each graph, the first column lists the number of vertices and edges for each instance, while the second column provides the number of vertices and edges in the largest strongly connected component (SCC). The figures in both columns are based on reduced graph. The last column presents the best results from our different configurations, including the size of the minimal feedback arc set and the time taken to compute it.

We implemented the algorithm in Python, using a graph library written in C++. By implementing it this way, the source code is relatively easy to understand while also keeping it reasonably fast and efficient. We tested the implementation on the Arnes computing cluster with a 12-hour time limit. Each run gave us solution sizes for all vertex orderings. For runs that did not finish within the time limit, we examined the sizes of SCCs that were being computed. We searched for the largest SCC that computed at least one ordering within the time limit. This provided a rough estimate of the largest SCC size manageable in a parallel or distributed setting where each ordering is processed by two separate threads or nodes, one for forward edge removal and one for backward edge removal. In our experiments this number is between 2.8 and 2.9 million edges. The source code and raw result data is available at [9].

Instance V-E	max-SCC V-E	Best result
7115-103689	1300-39456	7966: 14s
6301-20777	2068-9313	531: 15.3s
8114-26013	2624-10776	713: 20.9s
8717-31525	3226-13589	1186: 39s
8846-31839	3234-13453	1023: 38.8s
10876-39994	4317-18742	1721: 59.1s
22687-54705	5153-17695	1706: 1m37s
26518-65369	6352-22928	2254: 2m34s
36682-88328	8490-31706	2531: 3m44s
62586-147892	14149-50916	3361: 12m14s
75879-508837	32223-443506	141733: 37m53s
265214-418956	34203-151930	61598: 3m31s
325729-1469679	53968-304685	409862: 53m24s
281903-2312497	150532-1576314	313852: 9h43m2s
77360-828161	70355-888662	394218: 1h44m37s
82168-870161	71307-912381	403623: 1h48m43s

Table 1: Solution sizes and running times for SNAP instances

3.1 Impact of size reduction

The first option we tested was the use of size reductions. This reduction has two distinct effects. First, it reduces the size of the input graph. More importantly, the removal of edges and vertices can lead to a smaller SCCs, effectively shrinking the problem size and greatly reducing running time. Second, reduction rules should help decrease the size of the solution. While we add some removed edges to the solution, those edges are already guaranteed to be in the optimal solution.

For instances with the largest SCC having more than about 14,000 edges, the algorithm's running time was shorter when using reductions, with time savings increasing with input size. We find that for instances above this complexity, the benefits of reductions outweigh the cost. For a third of instances, the algorithm produced smaller solutions when reductions were not used, which is quite surprising. For these instances, *smartAE* was particularly effective, but its effect diminished after reductions. There is one instance for which the algorithm failed to complete within the time limit when not using reductions, but successfully computed a solution with configurations that used reductions.

3.2 Vertex orderings

Different vertex orderings bring drastically different results. The orderings based on degree difference and ratio, as well as the random ordering, give consistently worse solutions than those based on in-degree and out-degree. We should note that when comparing apparently analogous orderings, e.g. forward edges of an ascending ordering and backward edges of a descending ordering, there were minor, but non-zero differences in solution size. Such orderings were different due to the order of vertices with the same score.

The speed of the serial algorithm can be multiplied by a factor of $\frac{9}{4}$ with no effect on solution size, by only using the in-degree and out-degree orderings. Speed can be further doubled by only computing forward edge removals, but at a cost of slightly worse solution sizes. This gives us a speedup factor of $\frac{9}{2} = 4.5$. By taking into account quadratic time complexity, this would theoretically multiply the upper bound on feasible input size by 1.5, with no impact on solution size or approximately 2.12 with slightly worse solutions.

3.3 Impact of the SCC-based modification of edge removal

As described in Section 2.3, we implemented a modified version of the edge removal phase that is based on an SCC decomposition. The running time of the algorithm with this modification was more time consuming than the unmodified version. For some instances the running time doubled, while for others the impact on running time was less than 10%. The impact on solution size is more complex, though. If *smartAE* is not used, the solution is always smaller with the modification, as expected. However, when *smartAE* is applied, the solution can sometimes be larger with the modification. The difference is relatively small in those cases, ranging from 0.04% to 0.5%. This finding is somewhat unexpected as we expected this method to significantly improve solution size. This and our findings regarding reduction highlight unpredictability of *smartAE*. Similarly to our finding on vertex orderings, one can run the algorithm with

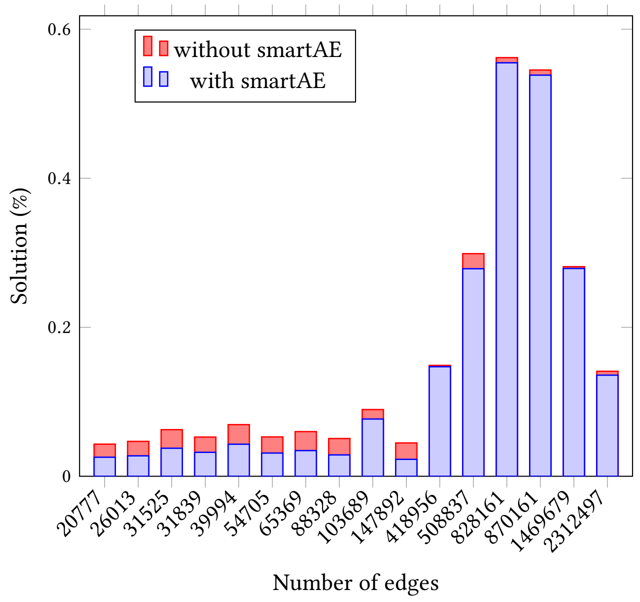


Figure 1: Percentage of solution edges compared to all edges in the graph, with and without the *smartAE* procedure.

both edge removal procedures to potentially achieve a slightly better solution size at a cost of running time.

3.4 Impact of smartAE

The addition of *smartAE* roughly doubles the running time for all cases without the modified edge removal phase. The improvement in solution size, however, is heavily dependent on input size. Generally, larger graphs exhibit smaller improvements compared to smaller graphs. This is clearly illustrated in Figure 1, which shows the percentage of edges selected for the FAS solution, both with and without the *smartAE* procedure. The red portion of the bars represents the improvement achieved with *smartAE*. For smaller graphs, *smartAE* can reduce the solution size by half, whereas for the largest tested graphs, the improvement is as little as 1%. This observation raises questions about the usefulness of *smartAE* for very large graphs.

4 CONCLUSIONS AND FUTURE WORK

The heuristic algorithm presented in this paper has proven effective in computing small feedback arc sets in large graphs. Multiple configurations have been tested with varying degrees of success.

Our implementation achieved comparable solution quality in significantly less time than the original approach on the same dataset, and it was also able to handle larger graphs. By employing size reduction techniques, we effectively decreased the complexity of input graphs, leading to faster processing times and more manageable SCCs, especially in larger graphs. We further reduced computational complexity by avoiding unnecessary recalculations during the *smartAE* process and by splitting SCCs into biconnected components. The experiments also revealed that selection of vertex ordering has a great impact on both the speed and quality of the solutions. Orderings based on in-degree and out-degree, particularly in descending and ascending orders, consistently outperformed other strategies. The modification covered in Section 3.3 did not consistently improve performance and, when it did, the benefits were usually insignificant. Additionally, it has led to less predictable results when *smartAE* was applied. The *smartAE* heuristic, while effective for reducing solution sizes in smaller graphs, showed lesser returns as graph size increased, raising questions about its efficiency and practicality in larger graphs.

However, there is still room for improvement. One is the speedup described in Section 3.3, but there are also other avenues. An obvious improvement is to implement the algorithm entirely in a compiled language like C++, eliminating the significant overhead of an interpreter. We plan on also adding more complex reduction rules based on Baharev's method [1], for example by searching for instances of tournaments or other common patterns and reducing based on those.

ACKNOWLEDGMENTS

We sincerely thank prof. dr. Uroš Čibej for his guidance, advice and for introducing us to this topic.

REFERENCES

[1] Ali Baharev, Hermann Schichl, Arnold Neumaier, and Tobias Achterberg. 2021. An Exact Method for the Minimum Feedback Arc Set Problem. *ACM J. Exp. Algorithmics* 26, Article 1.4 (apr 2021), 28 pages. https://doi.org/10.1145/3446429

[2] Claudia Cavallaro, Vincenzo Cutello, and Mario Pavone. 2023. Effective Heuristics for Finding Small Minimal Feedback Arc Set Even for Large Graphs. In *itaDATA*. https://api.semanticscholar.org/CorpusID:267035776

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3 ed.). MIT Press.

[4] Ali Dasdan. 2004. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems* 9, 4 (2004), 385–418. https://doi.org/10.1145/1027084.1027085

[5] Guy Even, Joseph Naor, Baruch Schieber, and Leonid Zosin. 1997. Approximating the minimum feedback arc set in tournaments. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 464–472.

[6] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, Raymond E. Miller and James W. Thatcher (Eds.). Springer US, Boston, MA, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[7] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[8] S. Park and S.B. Akers. 1992. An efficient method for finding a minimal feedback arc set in directed graphs. In *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*, Vol. 4. 1863–1866 vol.4. https://doi.org/10.1109/ISCAS.1992.230449

[9] Jure Pustoslemšek, Ema Črne, and Nejc Rihter. 2024. Implementation of a smartAE-based minFAS algorithm. https://github.com/jurepustos/fas-smartAE/. [Online; accessed 26-July-2024].