A Tutorial on the Feedback Arc Set and Feedback Vertex Set Problems on Directed Graphs

Nader Al-Naji

1 Introduction

In this paper, we discuss results for two well known problems: the feedback arc set problem and the feedback vertex set problem. While both of these problems have been well-studied for decades now, as we will find, new results are still being discovered.

In this paper, we begin by describing exactly what the problems are and then move to discussing results related to their complexity. We start with a proof that the feedback arc set problem is NP-Hard in general; we show this by reducing from vertex cover. After doing this, we show that there exists a linear-time reduction from feedback arc set to feedback vertex set and vice versa, thus showing that they can be seen as two representations of the same problem. Then, we show that the feedback vertex set problem is fixed-parameter tractable (as is the feedback arc set problem), a ground-breaking result that was discovered just in 2008. We then quickly summarize some results related to the approximatability of the two problems and finish off with a discussion of some results related to greedy Markov Chain based approaches to solving the problems.

2 Problem Specification

2.1 Decision Problem

Feedback Arc Set (FAS):
Given a directed graph $G = (V, E)$ and a value $k \geq 0$, does there exist a set of *arcs* $F \subseteq E$ with $|F| \leq k$ such that every directed cycle $c \in G$ has at least one *arc* in F? If so, return F; otherwise, return "NO".

Equivalently, given a directed graph $G = (V, E)$ and a value $k \geq 0$, does there exist a set of *arcs* $F \subseteq E$ with $|F| \leq k$ such that the subgraph $G' = (V, E - F)$ is acyclic? If so, return F; otherwise, return "NO".

Feedback Vertex Set (FVS):
Same as above, only now we seek a set $F \subseteq V$ of *vertices* instead of a set of *arcs*.

Given a directed graph $G = (V, E)$ and a value $k \geq 0$, does there exist a set of *vertices* $F \subseteq V$ with $|F| \leq k$ such that every directed cycle $c \in G$ has at least one *vertex* in F? If so, return F; otherwise, return "NO".

Equivalently, given a directed graph $G = (V, E)$ and a value $k \geq 0$, does there exist a set of *vertices* $F \subseteq V$ with $|F| \leq k$ such that the subgraph $G' = G[F]$ (where $G[F]$ denotes the subgraph of G *induced* by F) is acyclic? If so, return F; otherwise, return "NO".

As we will show, these two problems are reducible to each other in linear time and thus can be viewed as representing different instances of the same problem.

2.2 Optimization Problem

Given a directed graph $G = (V, E)$, remove the fewest number of edges (vertices in the case of FVS) from the graph so as to make the graph acyclic.

3 Applications

3.1 Deadlock Mitigation

In operating systems, processes can be seen as vertices and their dependence on other processes can be seen as directed edges. That is, when process A is waiting on process B to finish some task for it, we can draw a directed edge pointing from A to B. Thinking of processes in this way, directed cycles in the graph correspond to potential deadlocks in the system, where, for example, process A is waiting on process B *and* process B is waiting on process A. Such situations are generally undesirable because the processes wait for one another without making any progress until at least one process on every cycle is terminated, or until a dependency is removed. Finding the minimum number of process to terminate in order to relieve a deadlock is equivalent to solving an instance of the FVS problem, and finding the minimum number of dependencies to do away with is equivalent to solving an instance of the FAS problem.

3.2 Circuit Design

In circuit design, circuits can often be represented by graphs in which cycles typically imply race conditions. That is, if a cycle occurs in a circuit, it can mean that some circuit element has the potential to receive new inputs before it stabilizes. These race conditions can be avoided by placing clocked registers at each cycle in the circuit. Doing so, however, involves some cost, and so one naturally wants to keep the number of clocked registers as low as possible. This is exactly the minimum FVS problem.

3.3 Ranking Tournaments

Suppose one ran a chess tournament where everybody played everybody and that one wanted to use the results to rank all the players. One could start by constructing a graph where players are represented by vertices and games are represented by directed edges pointing from the winner to the loser. Then, if the graph were acyclic, one could simply topologically sort all of the players and obtain a ranking of the players based on who beat whom. But, unfortunately, unless one is very lucky, the graph will not be acyclic, meaning that there will be instances where A beats B, B beats C, and C beats A, which we will refer to as "upsets". As such, a natural goal becomes to construct a ranking that minimizes the number of upsets. Since each upset corresponds to an edge on a directed cycle in the tournament graph, finding the acyclic subgraph of the tournament constructed by removing the smallest number of edges directly corresponds to finding a ranking with the smallest number of upsets, and this is exactly the minimum feedback arc set problem.

4 FAS is NP-Complete

In this section, we show that FAS is NP-Complete by reducing from Vertex Cover. Recall that the vertex cover problem is an NP-complete problem in which we are tasked with finding a set of vertices V' of size $|V'| \leq k$ in a graph $G = (V, E)$ such that every edge has at least one of its vertices in V'. If no such set of vertices exists, we must return "NO".

This section is a based on a proof found in [1].

4.1 FAS is in NP

Given a directed graph $G = (V, E)$ and a FAS F of size $|F| \leq k$, we can check the validity of this FAS simply by attempting to perform a topological sort on the subgraph $G' = (V, E - F)$, which we can do in linear time, to see if it is indeed acyclic.

This section is based on constructions found in [6]. They were presented without proof in that source, however, and so I basically filled in all the holes myself.

4.2 Vertex Cover is Reducible to FAS

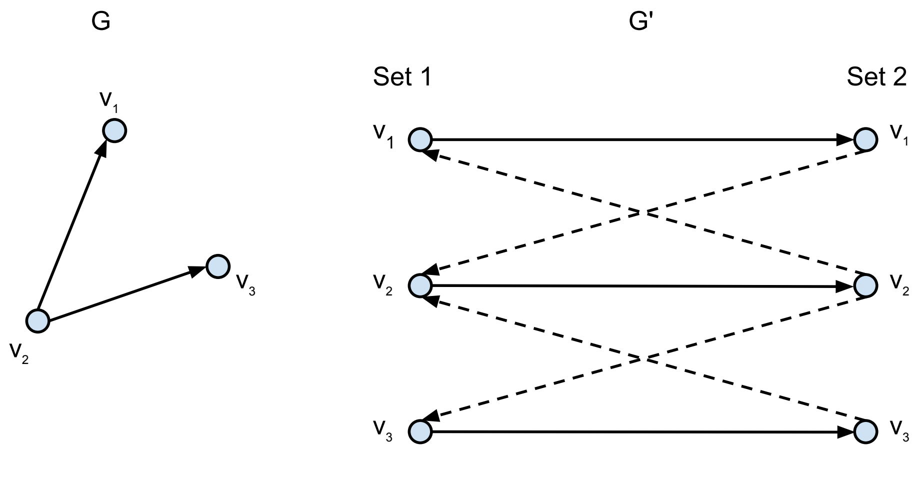
4.2.1 Conversion Procedure

Consider an instance of vertex cover: a directed graph $G = (V, E)$ and a number k. We construct the following graph on $2|V|$ vertices and $|V| + 2|E|$ edges:

$$V' = \bigcup_{v \in V} \{v^1, v^2\}$$

$$E' = \left[\bigcup_{v \in V} \{(v^1, v^2)\} \right] \cup \left[\bigcup_{(u,v) \in E} \{(v^2, u^1), (u^2, v^1)\} \right]$$

An example of the transformation above on a small graph appears in the figure below:



The claim is that $G' = (V', E')$ has a FAS of size k if and only if $G = (V, E)$ has a vertex cover of size k.

4.2.2 VC in G ⇒ FAS in G'

Suppose $X = \{v_1, v_2, ..., v_k\}$ is a VC of size k for G. The claim is that $Y = \{(v_1^1, v_1^2), ..., (v_k^1, v_k^2)\}$ is a FAS of size k for G'. It is clearly of size k, so now we prove that it is also a FAS by showing that every cycle in G' must have an edge in Y.

Consider any cycle $c \subseteq E'$. Since G' is bipartite, this cycle must contain an edge of the form (u^2, v^1). Further, since (u^1, u^2) is the only edge that *enters* u^2 and since (v^1, v^2) is the only edge that *leaves* v^1, by construction, we must have $(v^1, v^2) \in c$ and $(u^1, u^2) \in c$. Then from before, because $(u^2, v^1) \in E'$, we must have $(u, v) \in E$ and, therefore, either $u \in X$ or $v \in X$. This implies that either $(v^1, v^2) \in Y$ or $(u^1, u^2) \in Y$ by construction of Y. Since both of these edges are edges in c, we conclude that Y has at least one edge in every cycle and, therefore, that Y is a FAS for G' of size k.

4.2.3 FAS in G' $\Rightarrow$ VC in G

Suppose $Y = \{e_1, ..., e_k\} \subseteq E'$ is a FAS for G' of size k. First, we show that we can find a feedback arc set Y' that consists only of edges of the form (v^1, v^2). Suppose that e_i is not of this form. Then, it must be of the form (u^2, v^1) for some pair of vertices u, v such that $u \neq v$. But any cycle that goes through (u^2, v^1) goes through v^1 and must exit v^1. By construction, (v^1, v^2) is the only edge leaving v^1 and so any cycle intersecting e_i will also intersect (v^1, v^2) and so we can replace e_i with (v^1, v^2) and still have a FAS. Doing this for all $e_i \in Y$ yields a new FAS Y' with all edges of the form (v^1, v^2).

Now, we assume $Y' = \{(v_1^1, v_1^2), ..., (v_k^1, v_k^2)\}$ is a FAS of size k. We now show that $X = \{v_1, ..., v_k\}$ is a vertex cover of G. Every edge $(u, v) \in G$ is directly responsible for a directed cycle of the form (u^1, u^2), (u^2, v^1), (v^1, v^2), (v^2, u^1) in G'. So, we must have one of (u^1, u^2) or (v^1, v^2) in Y' and thus one of u, v is in X by construction, completing the proof.

4.2.4 Conclusion

Having proven the claim, we now know that if G' has no FAS, then G has no vertex cover. Further, the second part of the proof gave us a way to directly convert any FAS in G' into a vertex cover in G. Thus, an algorithm that returns "NO" if no FAS is found in G' and otherwise returns the converted vertex cover correctly solves the vertex-cover problem, showing that vertex cover is indeed reducible to FAS. Thus, since vertex cover is NP-complete, FAS is NP-complete as well.

5 FAS and FVS are Reducible to Each Other

In this section, we show that there exists a linear-time reduction from FAS instances to FVS instances. We also show the reverse, that there exists a linear-time reduction from FVS instances to FAS instances.

5.1 FAS is Reducible to FVS in Linear Time

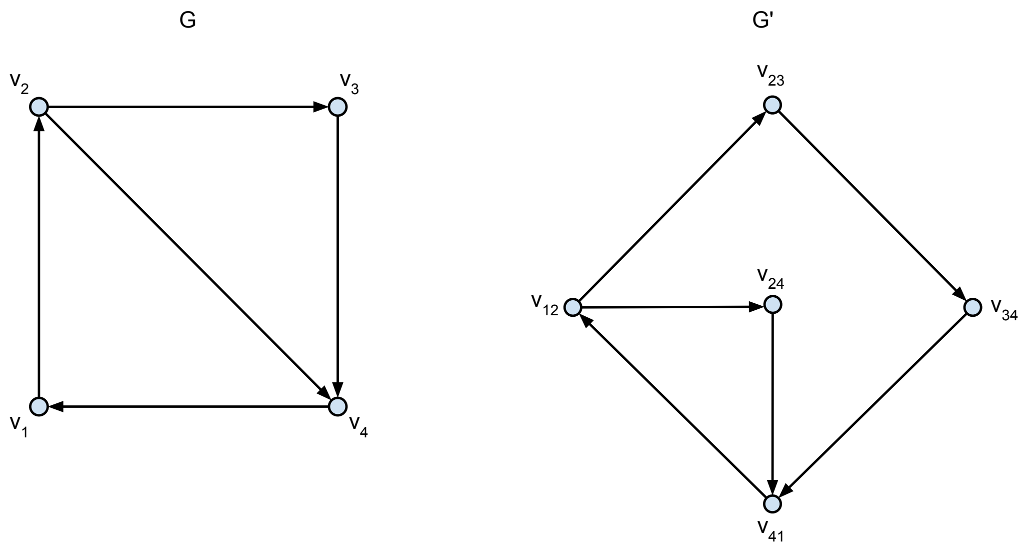
5.1.1 Conversion Procedure

To show this, we need to show that solving FVS instances efficiently enables us to solve FAS instances efficiently with a linear amount of overhead. We do so by converting FAS instances into FVS instances by performing the following transformation. Consider a FAS

instance: a directed graph $G = (V, E)$ and a number k. From this instance, we construct the following graph; without loss of generality label all vertices in V as $\{v_1, ..., v_n\}$ where $n = |V|$:

- First, convert all edges in E to vertices in a new vertex set V' as follows: for each edge $(v_i, v_j) \in E$, create a new vertex $v_{ij} \in V'$.
- Then, create a new edge set E' connecting vertices in V' as follows: create a new edge (v_{ij}, v_{kl}) if and only if $j = k$. So, for example, we connect v_{12}, v_{23} but not v_{12}, v_{34}.
- Let G' = (V', E').

An example of the transformation above on a small graph appears in the figure below:



The claim is that $G' = (V', E')$ has a FVS of size k if and only if $G = (V, E)$ has a FAS of size k.

5.1.2 FVS in G' ⇒ FAS in G:

Suppose we have a FVS of size k for G', denote it by X. Then $Y = \{(v_i, v_j) | v_{ij} \in X\}$ is a FAS of size k for G. To see this, consider a cycle $c \in G$ such that c consists of the consecutive edges $\langle e_1, ..., e_j, e_1 \rangle$ with $e_i \in E$ for all $i \leq j$. Labelling vertices arbitrarily using a superscript, we can rewrite this cycle's set of edges as consecutive vertex tuples of the form: $\langle (v^1, v^2), (v^2, v^3), ..., (v^j, v^{j+1}), (v^{j+1}, v^1) \rangle$. Note that the second vertex in each tuple corresponds to the first vertex in another tuple because these edges form a cycle. Then, by construction of G', each of these edges (v^i, v^j) corresponds *directly* to a vertex $v_{ij} \in V'$. Further, because we connect vertices in G' of the form (v_{ij}, v_{jk}), this set of tuples also corresponds directly to a cycle $c' \in G'$. Since X is a FVS for G', and therefore contains at least one vertex in each cycle in G', it must contain a vertex on the cycle c'. Without loss of generality, call this vertex v_{ij}. As mentioned, because v_{ij} is a vertex on c', it corresponds *directly* to an edge (v^i, v^j) in c by construction of G'. Finally, because Y includes this edge by construction, we guarantee that every cycle $c \in G$ has at least one edge in Y and that Y is therefore a FAS by definition.

5.1.3 FAS in G $\Rightarrow$ FVS in G':

This argument is essentially the reverse of the previous one. Suppose we have a FAS of size k for G, denote it by Y. Then $X = \{v_{ij} | (v_i, v_j) \in Y\}$ is a FVS of size k for G'. To see this, suppose we have a cycle $c' \in G'$. We can convert this cycle directly to a cycle $c \in G$ by converting each vertex $v_{ij} \in c'$ to an edge $(v_i, v_j) \in c$ in the original graph by the same argument given previously. Then, because Y is a FAS, Y must include an edge $(v_i, v_j) \in c$. By construction, X therefore contains a vertex $v_{ij} \in c'$ corresponding directly to $(v_i, v_j) \in Y$. Thus X has at least one vertex in every cycle $c' \in G'$ and therefore X is a FVS for G'. This completes the proof.

5.1.4 Conclusion

Having proven the claim, we know that if G' has no FVS, there does not exist a FAS for G. Further, the first part of the proof gives a way to directly convert any FVS for G' into a FAS for G of the same size. Thus, an algorithm that returns "NO" if it fails to find a FVS for G' and otherwise returns the converted FAS for G solves the FAS problem. Since the original transformation clearly takes a linear amount of overhead, we thus conclude that this is a linear-time reduction from FAS to FVS.

5.2 FVS is reducible to FAS in linear time:

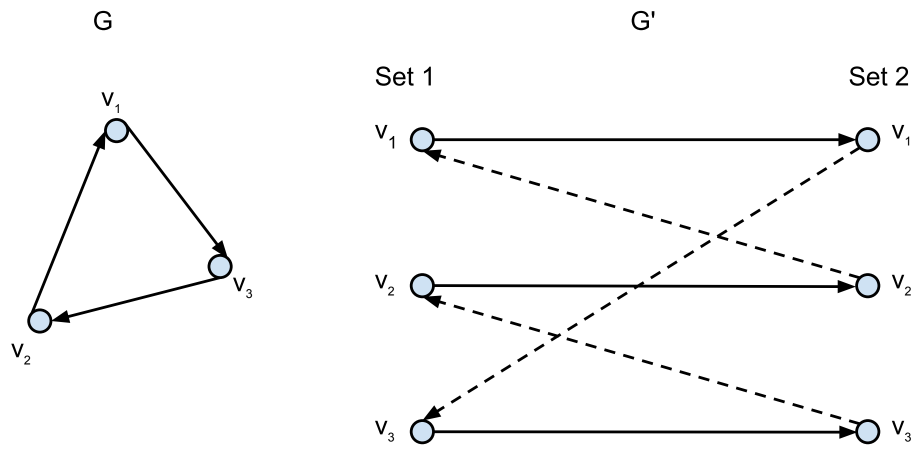
5.2.1 Conversion Procedure

We show this by converting FVS instances into FAS instances by performing the following transformation. Consider a FVS instance: a directed graph $G = (V, E)$ and a number k. From this instance, we construct the following graph:

$$V' = \bigcup_{v \in V} \{v^1, v^2\}$$

$$E' = \left[\bigcup_{v \in V} \{(v^1, v^2)\} \right] \cup \left[\bigcup_{(u,v) \in E} \{(u^2, v^1)\} \right]$$

An example of the above transformation appears in the figure below. Note that this transformation is very similar to the transformation used in the vertex cover reduction; the only difference is that here we only create a single edge $(u^2, v^1) \in E'$ for each edge $(u, v) \in E$,



rather than also creating the edge (v^2, u^1):
The claim is that $G' = (V', E')$ has a FVS of size k if and only if $G = (V, E)$ has a FAS of size k.

5.2.2 FAS in G' ⇒ FVS in G:

Suppose we have $Y = \{e_1, ..., e_k\}$ a FAS of size k for G'. First, we transform this FAS into a new set of edges, Y', where all edges in Y' are of the form (v^1, v^2). We do this as follows. Take any edge e_i from Y. By construction of the graph, if this edge is not of the form (v^1, v^2), it must be of the form (u^2, v^1). In the first case, we leave the edge alone. In the second case, note that while many edges can enter v^1, only one edge, (v^1, v^2), emanates from v^1 by construction. Thus, any cycle that contains the edge (u^2, v^1) must also contain the edge (v^1, v^2) and we can replace the edge $e_i = (u^2, v^1)$ with (v^1, v^2) and still have a FAS, because this new edge intersects all of the same cycles as e_i.

Now, suppose we've transformed our original FAS into a new FAS of size k for G': $Y' = \{(v_1^1, v_1^2), ..., (v_k^1, v_k^2)\}$. The claim is that $X = \{v_1, ..., v_k\}$ (corresponds to "collapsing" all the vertices in Y') is a FVS of size k for the original graph G. Take any cycle in $c \in G$ consisting of the vertices $\langle v_1, ..., v_j, v_1 \rangle$, where each consecutive pair of vertices is connected by an edge. By construction of G', each vertex $v_i \in c$ corresponds to a pair of vertices v_i^1, v_i^2 connected by an edge $e_i = (v_i^1, v_i^2) \in E'$. Further, by construction of the graph, each consecutive pair of vertices $v_i, v_{i+1} \in c$, connected by an edge in G, is also connected by an edge in G', namely (v_i^2, v_{i+1}^1). Thus the original cycle of vertices $c \in G$ corresponds directly to a cycle of edges $c' \in G'$ of the form $\langle (v_1^1, v_1^2), (v_1^2, v_2^1), ...(v_j^2, v_1^1) \rangle$. Because Y' is a FAS in G', it contains an edge in c' and, further, because Y' contains only edges of the form (v^1, v^2), this edge corresponds directly to a vertex in c, implying X will contain a vertex $v \in c$ by construction. Thus, X contains a vertex on every cycle $c \in G$, and therefore X is a FVS for G.

5.2.3 FVS in G $\Rightarrow$ FAS in G':

Suppose we have $X = \{v_1, ..., v_k\}$ a FVS of size k for G. The claim is that $Y = \{(v_1^1, v_1^2), ..., (v_k^1, v_k^2)\}$ is a FAS for G'. Suppose we have a cycle $c' \in G'$. By construction of G', c' must be of the form $\langle (v_1^1, v_1^2), (v_1^2, v_2^1), (v_2^1, v_2^2), ..., (v_j^2, v_1^1) \rangle$, since edges go to the corresponding vertex on the right or to a distinct different vertex on the left (assume no self-loops). Now, also by construction of G', we have that each edge (v_i^1, v_i^2) corresponds to a vertex $v_i \in G$, and each edge (v_i^2, v_{i+1}^1) corresponds to an edge $(v_i, v_{i+1}) \in G$. Thus, the cycle $c' \in G'$ consisting of edges $\langle (v_1^1, v_1^2), (v_1^2, v_2^1), (v_2^1, v_2^2), ..., (v_j^2, v_1^1) \rangle$ corresponds to a cycle $c \in G$ consisting of edges and vertices $\langle v_1, (v_1, v_2), v_2, ..., (v_j, v_1) \rangle$. Because X is a FVS for G, it must contain some $v_i \in c$, which implies Y must contain some $(v_i^1, v_i^2) \in c'$ by construction. Thus, Y contains an edge intersecting withf every $c' \in G'$, and therefore Y is a FAS for G'. This completes the proof.

5.2.4 Conclusion

Having proven the claim, we know that if G' has no FAS, there does not exist a FVS for G. Further, the first part of the proof gives a way to directly convert any FAS for G' into a FVS for G of the same size. Thus, an algorithm that returns "NO" if it fails to find a FAS for G' and otherwise returns the converted FVS for G solves the FAS problem. Since the original transformation clearly takes a linear amount of overhead, we thus conclude that this is a linear-time reduction from FVS to FAS.

6 Fixed Parameter Tractability

An algorithm is fixed parameter tractable if it can be solved in time $f(k)n^{O(1)}$ for some function f, where k is an input parameter for the algorithm that is *independent* of n, which is the size of the input. Recall that because instances of the FAS problem and instances of the FVS problem are reducible to one another in linear time, showing that either of the problems is fixed-parameter tractable automatically implies that the other is fixed-parameter tractable as well. The fixed parameter tractability of the directed FAS/FVS problems was posted as an open problem in the very first papers about fixed-parameter tractability and remained an open problem for decades. [-¿citation Downey and Fellows 1992, 1995] However, in 2008, Jianer Chen et al resolved this open problem by developing new techniques that lead to the conclusion that the FVS problem on directed graphs is fixed-parameter tractable. In particular, they discovered an algorithm for finding a feedback vertex set of size k, or correctly reporting that none exist, for directed graphs that runs in $O(4^k k^3 k! n^4)$ time. In this section we replicate the arguments used in their paper for completeness.

The proof proceeds by first showing that the FVS problem on directed graphs can be reduced in time $f(k)n^{O(1)}$ to a special version of the multi-cut problem called the *skew-separator problem.* Then, they develop an algorithm that shows the fixed-parameter tractability of the skew separator problem. The combination of these results gives an algorithm with running time $O(4^k k^3 k! n^4)$ for the FVS problem on directed graphs, proving its fixed-parameter tractability. Note that the skew-separator problem was invented by Chen et al for the sole purpose of proving the fixed-parameter tractability of the FVS problem on directed graphs and does not appear anywhere else, at least to my knowledge.

Although presenting this proof may seem like overkill for a tutorial paper, this result seemed so important that it warranted a full treatment. Most of the arguments are just rephrasings from [2] recreated from my own understanding, but a few parts, where the arguments in the paper were particularly concise and clear, are direct quotes. All of the actual content is also pulled straight from the paper. Diagrams are obviously all mine.

6.1 Skew-Separator Problem

6.1.1 Definition:

Let $\langle S_1, ..., S_l \rangle$ and $\langle T_1, ..., T_l \rangle$ be two collections of l vertex subsets in a directed graph $G = (V, E)$. A *skew-separator* X for $(\langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle)$ is a vertex subset in $V - \bigcup_{i=1}^{l}(S_i \cup T_i)$ such that for any pair of indices $1 \leq j \leq i \leq l$, there is no path from S_i to T_j in the graph $G - X$.

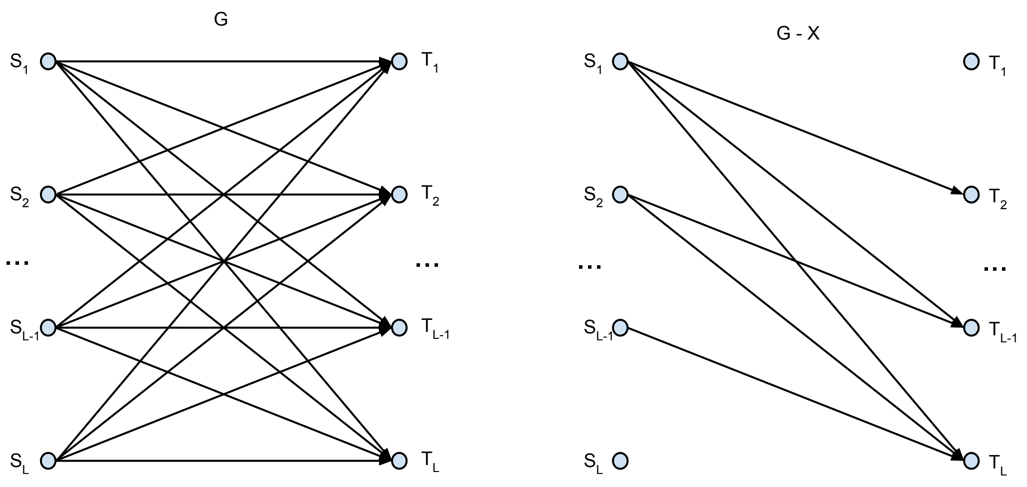
A skew-separator is essentially a set of vertices that, when removed, disconnect all paths between vertices in S_i and T_j where $i \geq j$. Below is a diagram showing a possible skew-separation. Points in the diagram denote sets of vertices and we use arrows to denote paths that exist between vertices in these sets. The diagram on the left is the input graph (composed of sets of vertices and paths that exist between them), while the graph on the right is the graph (also composed of sets of vertices and paths between them) induced by removing a skew separator X.

6.1.2 Problem Specification

Given $(G, \langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle, k)$, where G is a directed graph, $\langle S_1, ..., S_l \rangle$ is a collection of l sets of "source" vertices in G, $\langle T_1, ..., T_l \rangle$ is a collection of l sets of "sink" vertices in G, and a parameter k such that:

1. All sets $S_1, ..., S_l$, $T_1, ..., T_l$ are pairwise disjoint.
2. Only S_l is allowed to have edges coming into it. None of the S_i for $i < l$ are allowed incoming edges.
3. None of the T_i are allowed to have edges coming out of them.

The goal is to either construct a skew separator of at most k vertices or report that no such separator exists. Requirements (2) and (3) may seem odd, but they turn out to be necessary for the reduction from FVS to skew-separator.



6.1.3 Solving the Skew-Separator Problem

Let T_{all} denote $\bigcup_{1\leq i\leq l} T_i$. The algorithm for solving the skew-separator problem consists of explicitly defining what to do in three relatively simple cases, and then defining what to do in one slightly more complex case where none of the three simpler cases hold. We now walk through the construction of this algorithm starting with the three simple cases.

- **Rule R1.** There is no path from S_l to T_{all}. In this case, S_l is technically already "separated" from all of the sink sets and so we can exclude S_{l-1} and T_{l-1} and work instead on the instance $(G, \langle S_1, ..., S_{l-1}\rangle, \langle T_1, ..., T_{l-1}\rangle, k)$. Note we can also remove T_l since it only imposed a meaningful constraint when S_l was present.

- **Rule R2.** There is a direct edge from S_l to T_{all}. Recall from the definition of the skew-separator problem that vertices we remove must lie in $V - \bigcup_{i=1}^{l}(S_i \cup T_i)$. Thus, if there exists a direct edge between S_l and T_{all} as described, we can immediately report "NO".

- **Rule R3.** There exists a vertex $w \notin V - \bigcup_{i=1}^{l}(S_i \cup T_i)$ such that there exists a direct edge from S_l to w *and* a direct edge from w to T_{all}. In this case, the vertex w *must* be included in the skew separator and we can run on the reduced instance $(G - w, \langle S_1, ..., S_l\rangle, \langle T_1, ..., T_l\rangle, k - 1)$ and recursively find a skew-separator of size $k - 1$.

After considering these three cases, we have just one more case to consider. This last case is where we have a direct edge from S_l to some vertex $w \notin T_{all}$ but no direct edge from w to T_{all}. Further, because rules $R1 - R3$ do not hold, such a vertex must exist. So, fix one such vertex w and let $S'_l = S_l \cup w$; we will call w the "extended" vertex. After doing this, the following simple lemma holds (the proof is straightforward and omitted here):

Lemma 1: Let X be a subset of vertices in G that does not contain the extended vertex w. Then X is a skew-separator for $(\langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle)$ if and only if X is a skew-separator for $\langle S_1, ..., S_l' \rangle, \langle T_1, ..., T_l \rangle$.

And this lemma directly implies the following two corollaries:

Corollary 1: A skew-separator for $\langle S_1, ..., S_l' \rangle, \langle T_1, ..., T_l \rangle$ is also a skew-separator for $\langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle$.

Corollary 2: The size of a min-cut from S_l' to T_{all} in the graph G is at least as large as the size of a min-cut from S_l to T_{all} in G.

Finally, using this lemma and its corollaries allows us to derive the central theorem behind the algorithm for solving the skew-separator problem. The proof of this utilizes everything we've done so far but it is somewhat involved. As such, we state it below without proof, but note that the details are available on page 7 of the original paper.

Theorem 1: If the size of a min-cut from S_l to T_{all} is equal to the size of a min-cut from S_l' to T_{all}, then the pair $\langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle$ has a skew-separator of size at most k if and only if $\langle S_1, ..., S_l' \rangle, \langle T_1, ..., T_l \rangle$ has a skew-separator of size at most k.

This theorem allows us to immediately construct an algorithm for finding a skew-separator of size k. In particular, it allows us to simply return the skew-separator for $\langle S_1, ..., S_l' \rangle, \langle T_1, ..., T_l \rangle$ if we find that the min-cut between (S_l, T_{all}) and (S_l', T_{all}) are equal. Below, we give the full algorithm for finding a skew-separator for a graph G, putting together all the work we've just done.

Algorithm SMC($G, \langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle, k$): an algorithm for solving the skew-separator problem.

1. **If:** $l = 1$ **Then** Solve the problem in time $O(kn^2)$ using an efficient min-cut algorithm.
2. **If:** R2 applies or $k < 0$ **Then:** Return "NO"
3. **If:** R1 applies **Then:** Return SMC(G, $\langle S_1, ..., S_{l-1} \rangle, \langle T_1, ..., T_{l-1} \rangle$, k)
4. **If:** R3 applies on a vertex w **Then:** Return $\{w\} \cup$ SMC(G, $\langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle$, $k - 1$)
5. Pick an extended vertex w'; let $S_l' = S_l \cup \{w'\}$
6. Let m be the size of a min-cut from S_l to $T_{all} = \bigcup_{1 \leq i \leq l} T_{all}$
7. **If:** $m > k$ **Then:** return "NO''"

```
8. let $m'$ be the size of a min-cut from $S_l'$ to $T_{all}$
9. **If:** $m = m'$ **Then:** Return SMC(G, $\langle S_1, ..., S_l' \rangle, \langle T_1, ..., T_l \rangle$, $k$)
10. **Else:**
    10.1. $X = \{w'\} \cup$ SMC($G - w'$, $\langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle$, $k - 1$)
    10.2. **If:** $X \neq$ "NO" **Then:** Return $X$
    10.3. **Else:** Return SMC(G, $\langle S_1, ..., S_l' \rangle, \langle T_1, ..., T_l \rangle$, $k$)
```

6.1.4 Correctness of Algorithm

In this section, we argue that the SMC algorithm given in the previous section correctly computes a skew-separator of size at most k or returns "NO" if one does not exist.

The first four lines are obvious given our discussion of the initial base cases. In line 7, it makes sense to return "NO" because if we cannot find a min-cut of size $\leq k$, we know a skew-separator cannot exist, since just separating S_l takes more than k vertices. Line 9 follows directly from the theorem proved in the previous section. Finally, if the min-cut sizes are not equal, we run two exhaustive searches where we include the vertex w' in the skew-separator in one search, and omit it from the skew-separator in the other. If we do not find a skew-separator after completing both of these exhaustive searches, we can be sure that no skew-separator exists and return "NO". This completes the verification of the correctness of the algorithm.

6.1.5 Complexity

In this section, we show that the running time for the SMC algorithm is $O(k4^k n^3)$. We use induction on k, the input parameter, and m, the size of the min-cut between S_l and T_{all}.

The recursive execution of the algorithm can be described as a search tree, T. We first count the number of leaves in the search tree, noting that only step 10 corresponds to branches in T. Let $D(k, m)$ be the number of leaves in T for the algorithm SMC(G, $\langle S_1, ..., S_l \rangle$, $\langle T_1, ..., T_l \rangle$, k), where m is the size of the min-cut from S_l to T_{all}. Then steps 9 and 10 induce the following recurrence relation:

$$D(k, m) \leq D(k - 1, m_1) + D(k, m_2)$$

where m_1 is the size of a min-cut from S_l to T_{all} in the graph $G - w'$ as given in step 10.1, and m_2 is the size of a min-cut from S_l' to T_{all} in the graph G as given in step 10.3. Note that $m - 1 \leq m_1 \leq m$ since removing w' cannot increase the size of a min-cut in the reduced graph. Further, by corollary 2, we must have $m_2 \geq m + 1$. Summarizing, we have:

$$m - 1 \leq m_1 \leq m$$
$$m_2 \geq m + 1$$

We prove by induction on $2k - m$ that $D(k,m) \leq 2^{2k-m}$. First, note that by the time we hit step 9, it must be true that $k \geq m \geq 0 \Rightarrow 2k - m \geq 0$. In the initial case, we have $2k - m = 0$ since $k = m = 0$ and, in this case, the algorithm can solve the instance without further branching. Thus, we have $D(k,m) = 1$ when $2k - m = 0$. For the inductive step, the inequalities from above give us:

$$\begin{aligned} 2(k-1) - m_1 \leq 2(k-1) - (m-1) &= 2k - m - 1 \\ 2k - m_2 \leq 2k - (m+1) &= 2k - m - 1 \end{aligned}$$

And, therefore, we can apply the inductive hypothesis on the first inequality for $D(k,m)$, which yields:

$$\begin{aligned} D(k,m) &\leq D(k-1, m_1) + D(k, m_2) \\ &\leq 2^{2(k-1)-m_1} + 2^{2k-m_2} \\ &\leq 2^{2k-m-1} + 2^{2k-m-1} \\ &= 2^{2k-m} \end{aligned}$$

This completes the proof. Note that other non-branching steps such as 3, 4, and 9 might also change the values of k and m, affecting the value $2k - m$ used in the induction, but it can be shown none of these steps *increase* the value of $2k - m$ (see page 11 of the original paper for a full treatment).

Summarizing this result, we have that the total number of leaves in T satisfies:

$$D(k,m) \leq 2^{2k-m} \leq 4^k$$

.

It can be shown that all non-branching steps such as 1 and $6 - 8$ have their running time bounded by $O(kn^2)$, where n is the number of vertices in G, since they rely on efficiently computing min-cuts for graphs of size bounded by n. Further, note that for each recursive call in an execution of the algorithm, either the number of source-sink pairs is decreased by 1 (step 3) or the number of non-terminal vertices is decreased by 1 (steps 9, 10). When the number of source-sink pairs is equal to 1, the problem is solved in $O(kn^2)$ time by step 1, and when the number of vertices no in source or sink sets is zero, steps 2 or 3 can be directly applied. Thus, along each root-leaf path in T, there are at most $O(n)$ recursive calls to the algorithm. Thus, the running time along each root-leaf path in the tree is $O(kn^3)$.

6.1.6 Conclusion

Finally, to summarize the discussions, we conclude that the running time of the algorithm SMC as a whole, which computes a skew-separator of size bounded by k for instances $(G, \langle S_1, ..., S_l \rangle, \langle T_1, ..., T_l \rangle, k)$ or reports correctly that none exist, is bounded by $O(k4^k n^3)$, which implies that the skew-separator problem is fixed-parameter tractable.

6.2 The DAG-Bipartition FVS Problem

In this section, we define a new problem, the DAG-Bipartition FVS problem and show that it is reducible to the skew-separator problem. After we do this, we can show that the FVS problem on directed graphs is reducible to the skew-separator problem by first reducing it to the DAG-Bipartition problem.

6.2.1 Definition:

Let $G = (V, E)$ be a directed graph and let (D_1, D_2) be a bipartition of the vertex set V of G; that is, $D_1 \cup D_2 = V$ and $D_1 \cap D_2 = \emptyset$. The bi-partition (D_1, D_2) is a DAG-bipartition for the graph G if both induced subgraphs $G(D_1)$ and $G(D_2)$ are DAG's. A vertex subset F in the graph G is a D_1-FVS if F is an FVS for G and $F \subseteq D_1$.

6.2.2 Problem Specification:

Given (G, D_1, D_2, k) where G is a directed graph, (D_1, D_2) is a DAG-bipartition for G, and k is the parameter, either consruct a D_1-FVS of size bounded by k for G, or report correctly that no such D_1-FVS exists.

6.2.3 Reducing DAG-Bipartition to Skew-Separator

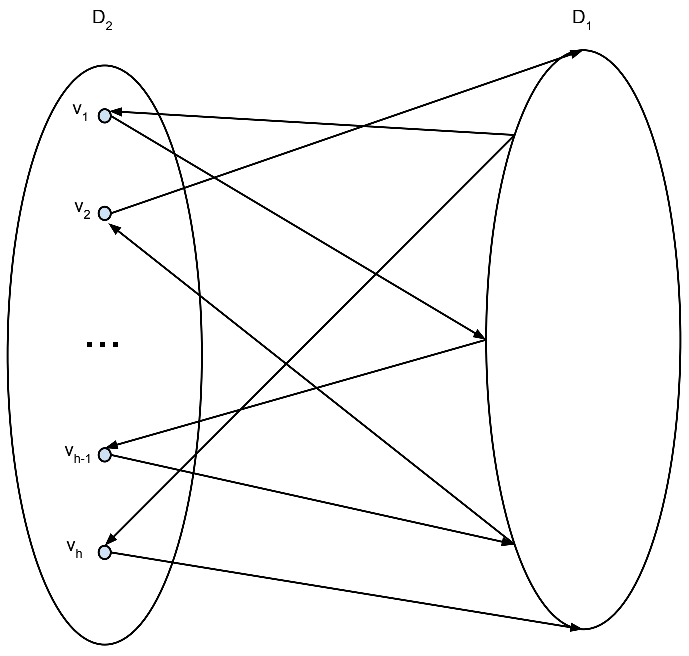
In this section, we describe how to use the skew-separator problem to solve an instance of the DAG bipartition FVS problem.

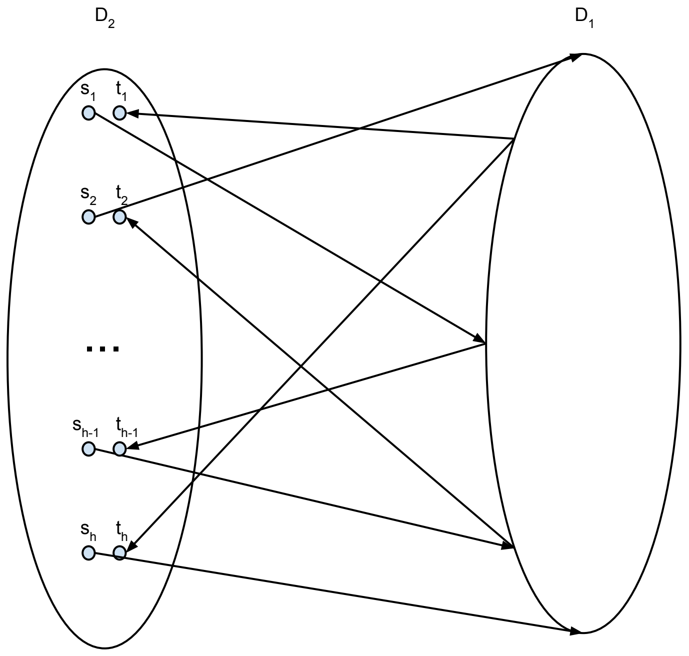
Start with an instance of the DAG biparition FVS problem: (G, D_1, D_2, k). Since $G(D_1)$ and $G(D_2)$, the graphs induced by D_1 and D_2, are acyclic by definition, let $\pi = \{v_1, ..., v_h\}$ be a topologically sorted order of the vertices in the DAG induced by D_2. We construct an instance of the skew-separator problem as follows:

1. Let G' be the graph obtained from G by removing all edges in $G(D_2)$.
2. In the graph G', replace each vertex $v_i \in D_2$ by a pair (s_i, t_i) of vertices such that all outgoing edges from v_i are now going out from s_i and all incoming edges to v_i are now coming into t_i. Let the resulting graph be G_π.

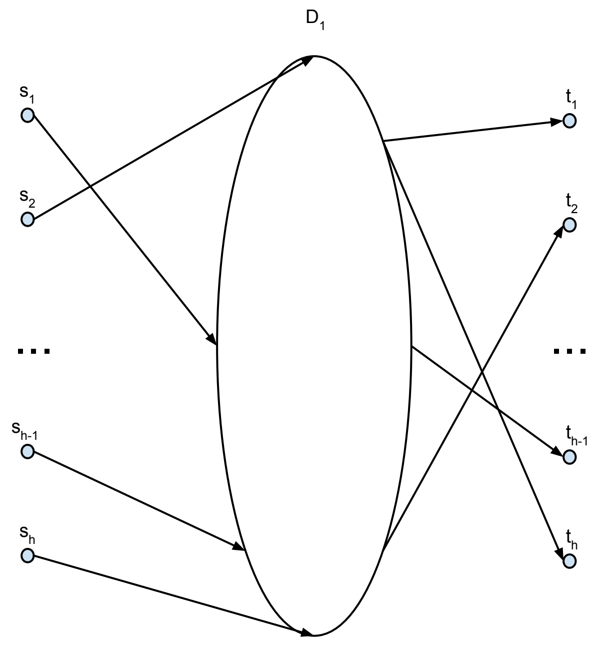
It is fairly easy to verify that the graph that results from this process will be a valid instance of the skew-separator problem but we omit these details here.

An illustration of the above transformation is shown below. In the first step, we show the sets of vertices D_1 and D_2, with edges connecting vertices in D_2 to vertices in D_1 drawn as arrows, and edges connecting vertices in D_2 to each other removed (as per step (1)). The next diagram, shows the splitting of each vertex in $v_i \in D_2$ into a pair s_i, v_i. Finally, the last diagram rearranges the graph to emphasize the similarity between the DAG bipartition problem and the skew-separator problem:





Having described the transformation, we now state the following important theorem, which will enable us to develop an algorithm for solving the DAG-Bipartition problem.



Theorem 1: Let (G, D_1, D_2, k) be an instance of the DAG-bipartition FVS problem, and let X be a D_1-FVS for the graph G. Then there is a topologically sorted order $\pi = \{v_1, ..., v_h\}$ of the vertices in $G(D_2)$ such that in the instance $(G_\pi, \langle\{s_1\}, ..., \{s_h\}\rangle, \langle\{t_1\}, ..., \{t_h\}\rangle, k)$ induced by (G, D_1, D_2, k) and π: (1) X is a skew-separator for the pair $(\langle\{s_1\}, ..., \{s_h\}\rangle, \langle\{t_1\}, ..., \{t_h\}\rangle)$ in the graph G_π and (2) every skew-separator for the pair $\langle\{s_1\}, ..., \{s_h\}\rangle, \langle\{t_1\}, ..., \{t_h\}\rangle$ in G_π is a D_1-FVS.
The proof of this theorem is somewhat involved and we omit it here, but it can be found on page 13 of the original paper.

This theorem is extremely powerful and immediately suggests an algorithm for computing a DAG-bipartition using the SMC algorithm for the skew-separator problem. Since every skew-separator for G_π is a D_1-FVS for G, in order to find a D_1-FVS for G, all we need to do is search through all topological orderings of $G(D_2)$ and report the first skew-separator we find as the D_1-FVS for G. Further, since if there exists a D_1-FVS for G then there exists a skew-separator for G_π, if we are not successful in finding a skew-separator after searching all of the G_π, we are assured that no D_1-FVS exists. The following algorithm codifies this logic, which also serves as its proof of correctness.

Algorithm DBF(G, D_1, D_2, k): an algorithm for solving the DAG bipartition FVS problem

```
1. for each topologically sorted order $\pi = \{v_1, ..., v_h\}$ of the vertices in $G(D_2)$ do:
    1.1. construct the instance $(G_\pi, \langle\{s_1\}, ..., \{s_h\}\rangle, \langle\{t_1\}, ..., \{t_h\}\rangle, k)$ of the skew-separator
```

```
        problem induced by $(G, D_1, D_2, k)$ and $\pi$
    1.2. let $X = SMC(G_\pi, \langle\{s_1\}, ..., \{s_h\}\rangle, \langle\{t_1\}, ..., \{t_h\}\rangle, k)$
    1.3. if $X$ is a $D_1$-FVS of size bounded by $k$ for $G$, then return $X$
2. return "NO"
```

The correctness of the algorithm follows directly from the theorem, since the algorithm searches all possible skew-separators over all possible orderings π.

6.2.4 Complexity

The running time of the algorithm $DBF(G, D_1, D_2, k)$, which solves the DAG-Bipartition FVS problem, is $O(k4^k h! n^3)$, where h is the number of vertices in D_2 and n is the number of vertices in the input graph G. This running time is obvious since the for-loop in step 1 executes at most $h!$ times, and each time the execution is dominated by the skew-separator sub-routine, which runs in time $O(k4^k n^3)$.

6.2.5 Conclusion:

Putting together all the work we've done in this section, we conclude that DAG-Bipartition FVS problem instances (G, D_1, D_2, k) can be solved in time bounded by $O(k4^k h! n^3)$, where k is the input parameter, h is the number of vertics in D_2, and n is the number of vertices in the input graph G. The algorithm DBF correctly computes such a FVS, or correctly reports that none exist.

6.3 The FVS Problem on Directed Graphs

We are almost ready to reduce the FVS problem on directed graphs. In this section, we introduce one more problem, the Directed FVS Reduction problem, show that it can be solved in time $O(k!k^3 4^k n^3)$, and then finally reduce the general FVS problem to this reduction problem.

6.3.1 Directed FVS Reduction Problem

This is a slightly more restricted version of the general FVS problem. It is defined as follows. Given a triple (G, F, k), where G is a directed graph and F is a FVS of size $k+1$ for G, either construct an FVS of size bounded by k for G or report that no such FVS exists. We can solve this problem in $O(k!k^3 4^k n^3)$ by reducing to the DAG-Bipartition FVS problem. We give a rough sketch of the proof below.

First, consider any FVS for G of size k, call it F'. This FVS can be split into two pieces, F_{in} and F_{out} such that $F_{in} \in F$ and $F_{out} \notin F$. Because we know that no vertex in $F - F_{in}$ is in F', and because we know that F' is a FVS for G, we also know that the graph $G(F - F_{in})$ induced must be a DAG. Now, for each subset $F_{in} \subseteq F$ such that $G(F - F_{in})$ is a DAG, we want to look for a subset F_{out} with size $|F_{out}| \leq k - |F_{in}|$ such that $F_{in} \cup F_{out}$ is a FVS of size k for G.

Now fix some $F_{in} \subseteq F$ where $G(F - F_{in})$ is a DAG. The graph G has a FVS $F_{in} \cup F_{out}$ of size bounded by k if and only if F_{out}, which has no vertices in common with F, is a FVS for the reduced graph $G - F_{in}$. Thus, if we want to solve the original problem of finding a FVS of size bounded by k for G, we can instead focus on constructing a FVS F_{out} for the graph $G - F_{in}$ such that $|F_{out}| \leq k - |F_{in}|$.

We want to find a FVS for $G - F_{in}$, calling it F_{out}, that contains only vertices in $V - F$. Now this is where we make use of the DAG bipartition FVS problem. Let $D_1 = V - F$ and let $D_2 = F - F_{in}$. Because we are guaranteed that both $V - F$ and $F - F_{in}$ are acyclic, and that $V - F \cap F - F_{in} = \emptyset$, we can use the DAG-Bipartition algorithm to find a FVS $F_{out} \subseteq V - F$ for $V - F \cup F - F_{in} = V - F_{in}$, which is exactly what we want. In particular, having fixed $F_{in} \subseteq F$, we get that $F_{out} = DBF(G(V - F_{in}), V - F, F - F_{in}, k - |F_{in}|)$.

Taking j to be $|F_{in}|$, our DBF instance will have $|D_2| = |F - F_{in}| = k + 1 - j$. Further, we will be looking for a FVS of size $k - j$. Thus, the running time of this DBF sub-routine will be $O((k - j)(k + 1 - j)!4^{k-j}n^3)$.

Now, to solve the problem, we simply unfix F_{in} and enumerate all possible subsets $F_{in} \subseteq F$, check if the induced graph is a DAG and, if it is, apply $DBF(G(V - F_{in}), V - F, F - F_{in}, k - |F_{in}|)$. Putting all of this together results in a run-time that is bounded by:

$$\sum_{j=0}^{k} \binom{k+1}{j} (k - j)(k + 1 - j)!4^{k-j}n^3$$
$$= O(k!k^3 4^k n^4)$$

This completes the proof.

6.3.2 Reducing Directed FVS to Directed FVS Reduction

Finally, we are ready to prove our bound for the general FVS problem on directed graphs. The following is an algorithm for finding a FVS of size k for a graph G, or correctly reporting that no such FVS exists. The following algorithm uses a technique calles "iterative compression", so called because at each step we increase the size of the FVS *and* the size of the graph we are considering by one, then squash only the FVS down by one. Doing this $n = |V|$ times results in a FVS for the full graph.

Algorithm DFVS(G, k): an algorithm for solving the FVS problem on directed graphs.

1. Start with an arbitrary subset of $k+1$ vertices; call this the "working set": $V_{working} \subseteq G$ and $|V_{working}| = k + 1$.

2. Choose an arbitrary subset of k vertices from the working set; call this the "working FVS": $F_{working} \subseteq V_{working}$ and $|F_{working}| = k$. Note that this subset is automatically a FVS for the graph induced by the working set, since $V_{working} - F_{working}$ contains a single vertex and is therefore acyclic (assume no self-loops).

3. while $|V_{working}| < |V|$ do

    3.1. Choose a vertex $w \in V - V_{working}$ not in the working set.

    3.2. Add this vertex to the working set and the working FVS. That is, let $V_{working} \leftarrow V_{working} \cup \{w\}$ and $F_{working} \leftarrow F_{working} \cup \{w\}$.

    3.3. At this point, $F_{working}$ is a FVS of size $k+1$ for the graph induced by $V_{working}$, since the new vertex is in both the working set and the working FVS. We can therefore use the Directed FVS Reduction algorithm on the instance $(G(V_{working}), F_{working}, k)$.

    3.4. If running the Directed FVS Reduction algorithm on the above instance results in a "NO", return "NO". We do this because if we can't find a FVS of size k for the subgraph induced by $V_{working}$, we definitely cannot find one for the full graph G.

4. return $F_{working}$

The algorithm above clearly computes a FVS of size k for the original graph G if one exists, since at each step we are simply shrinking something that we know is already a FVS using the FVS Reduction algorithm previously defined.

6.3.3 Complexity

The algorithm above calls the FVS Reduction algorithm as a sub-routine at most $n - k - 1$ times. Since each call takes time $O(k!k^3 4^k n^3)$, the total running time is bounded by $O(k!k^3 4^k n^3(n-k-1)) = O(k!k^3 4^k n^4)$. Thus, we have our final result.

6.4 Conclusion:

All this work by Chen et al shows that the FVS problem on directed graphs can be solved in $O(k!k^3 4^k n^4)$ time, implying that the problem is fixed-parameter tractable. While this result closed a problem that had been open for over a decade, namely the question of whether FVS was fixed-parameter tractable on directed graphs, the result is also interesting in its approach. Rather than directly attack the problem of finding a FVS, they first solved a multi-cut problem (the skew-separator problem) and then reduced from this seemingly unrelated problem to the problem of finding a FVS. In this way, Chen et al provide us with multiple results in one paper, and with techniques that may prove useful in solving many other problems in the future.

7 FAS/FVS is APX-Hard

7.1 APX Definition

The class APX is the set of NP optimization problems that admit polynomial-time algorithms whose approximation ratio is bounded by a constant. Problems in this class have efficient algorithms that can find solutions to problems that are within some fixed percentage of the optimal solution. A problem that admits a 2-approximation, for example, would be in APX. A problem is APX-hard if there is a reduction from every problem in APX to that problem.

7.2 PTAS Definition

The class PTAS is the set of problems that admit polynomial-time algorithms that can solve the problem within *every* fixed percentage greater than zero, one algorithm for each percentage. That is, it consists of problems that have algorithms that can compute solutions that are within a factor of $(1+\epsilon)$ of the optimal answer in polynomial-time for every *fixed* ϵ (so an algorithm that is $O(n^{\frac{1}{\epsilon}})$ is still a PTAS). Having a problem that admits a polynomial-time approximation scheme (PTAS) is generally better than having one that only admits a fixed-approximation algorithm (APX), for obvious reasons. Further, although some problems in APX are also in PTAS, there exist problems in APX that cannot be in PTAS *unless* $P = NP$. In particular, $P \neq NP \Rightarrow PTAS \neq APX \Rightarrow$ no APX-hard problem is in PTAS. In this sense, having a problem that is APX-hard is generally bad news, as an answer to arbitrary precision, which could be computed if a PTAS existed, cannot be computed in general.

7.3 FAS/FVS Hardness

In [4], Vigo Kann showed that the FAS problem is APX-Hard. In particular, he showed that there is a constant c such that, assuming $P \neq NP$, there is no polynomial-time approximation algorithm that always finds an edge set at most c times bigger than the optimal result. Today, the highest value of c for which such an impossibility result is known is $c = 1.3606$, due to Irit Dinur et al in [7]. This means that there cannot exist an algorithm that guarantees that its output edge set is at most a factor of 1.3606 time larger than the optimal edge set. The best known approximation algorithm has a ratio of $O(\log n \log\log n)$, due to Guy Even et al in [8].

8 FAS on Tournaments

As mentioned previously, a tournament is a directed graph obtained by assigning a direction for each edge in an undirected complete graph. This is otherwise known as an *orientation* of a complete graph. These graphs are important objects of study, since they arise when trying to produce rankings of competing individuals.

The problem of computing a FAS when the input graphs are restricted to be tournaments, it turns out, admits a polynomial-time approximation scheme. The problem of computing FAS on tournaments is called the minimum feedback arc set problem on tournaments or FAST for short. Unfortunately, this problem is still NP-hard but, fortunately, it does admit a PTAS. The first PTAS was discovered in 2007 by Claire Kenyon-Mathieu et al in [3]. In particular, the algorithm they discovered computes a FAS of size less than $(1+\epsilon)OPT$ in time $O(\frac{1}{\epsilon}n^6 + n^2 2^{O(\frac{1}{\epsilon})} + n 2^{2^{O(\frac{1}{\epsilon})}})$, ignoring logarithmic factors, where OPT is the size of the minimum FAS.

9 Markov-Chain Based Heuristics

Some work, although very little, has been done on applying Markov-Chain based analysis to the problem of breaking cycles in graphs. What little work we reference here was done by Mile Lamaic et al in [5], whose work we summarize below.

The basic idea is as follows. First, strongly connected components are extracted from the graph. The reason this is done is two-fold. First, it is done because cycles occur only within strongly connected components. Second, it is done because computing the *limiting distribution*, a common thing to do when working with Markov Chains, can only be done if the underlying graph is strongly connected or "irreducible", to use the formalism associated with Markov Chains. Once the strongly connected components are identified, the algorithm selects a vertex according to some heuristic (in this case one based on Markov Chains), and then recursively adds that vertex to the FVS being considered and calls itself on the reduced graph with the vertex removed. This logic is codified in the algorithm below:

Algorithm MinFVS(G): an algorithm to compute the minimum FAS

```
1. Let $F$ be a vertex set initially empty.
2. Remove any vertices that are not connected to anything.
3. For all vertices $v$ that have exactly one unique predecessor (degree one), $u$, remove $v$
   from consideration and connect $u$ directly to all nodes $v$ was connected to. This can
   be done since any cycle that contains $v$ must also contain $u$.
4. for each strongly connected component $G_i$ of $G$, do
    4.1. $v \leftarrow$SelectVerted($G_i$); This function will be explained later.
    4.2. $F \leftarrow \{v\} \cup MinFVS(G_i - v)$
5. return $F$
```

The steps of the algorithm above are fairly straightforward and it will clearly always return a FVS, although not necessarily a minimum FVS. All of the magic happens in the SelectVertex method. Depending on the heuristic used to select the next vertex to try to include in the FVS, the approximation yielded by the algorithm could be good or bad. Here, we discuss Markov Chain based heuristics explored by Lamaic et al.

The first thing to consider as a heuristic for SelectVertex is each vertex's value in the stationary distribution. The stationary distribution for a graph is a vector of values, π, one for each node, whose entries sum to one, essentially encoding the proportion of time a random walker spends at each node. Taking the inverse of each element, $\frac{1}{\pi_i}$ gives the mean amount of time a surfer starting at node i would take to return to node i. Nodes that are members of many (short) cycles are expected to have a high mean return time and so using mean return time as a heuristic makes intuitive sense. Computing the stationary distribution is simple. First, the graph is encoded as a transition matrix. Once this is done, the eigenvector of the matrix associated with eigenvalue 1 is the stationary distribution, π.

As mentioned, one very reasonable heuristic for SelectVertex would be to simply select the vertex with the lowest mean return time, highest value π_i in the limiting distribution, and include it in the FVS. This greedy approach is exactly what Lamaic et al explored. In addition to this, however, they added one slightly more sophisticated improvement. Because the graph G and its inverse have the same FVS, it makes sense to also take the limiting distribution of G^{-1} into consideration. One could likely conceive of many ways of doing this but the one that worked best in practice for Lamaic et al was taking the arithmetic mean between π_i^G and $\pi_i^{G^{-1}}$ and using that as the heuristic to decide which vertex to add to the FVS in the SelectVertex method of the algorithm above. In particular, the vertex with the highest $(\pi_i^G + \pi_i^{G^{-1}})/2$ is the one always returned by SelectVertex.

Lamaic et al conducted some experimental results to see how their heuristics perform and the results looked promising when compared to previous experimental results, but no rigorous non-emperical results were shown.

10 Conclusion

In this paper we have covered a few seminal results related to the feedback arc set and feedback vertex set problems on directed graphs. Overall, while these problems are NP-complete, and APX-hard, they admit a polynomial-time approximation scheme (PTAS) in the special case where the input graphs are tournaments, and greedy algorithms using Markov Chain based heuristics appear to perform well in practice. Finally, very recently, a ground-breaking theoretical result showed that FAS/FVS was fixed-parameter tractable while also paving the way for the analysis of other problems as well.

References

[1] Jean H. Gallier. *NP Completeness of the Feedback Arc Set Problem.* http://www.cis.upenn.edu/ cis511/homework/sol5.pdf.

[2] Jianer Chen, Yang Liu, Songjian Lu, Barry O'Sullivan, Igor Razgon. *A Fixed-Parameter Algorithm for the Directed Feedback Vertex Set Problem.* October 2008.

[3] Claire Kenyon-Mathieu, Warren Schudy. *A PTAS for Weighted Feedback Arc Set on Tournaments.* 2007.

[4] Vigo Kann. *On the Approximability of NP-complete Optimization Problems.* May 1992.

[5] Mile Lemaic, Ewald Speckenmeyer. *Markov-Chain-Based Heuristics for the Minimum Feedback Vertex Set Problem.*

[6] Ding-Zhu Du, Panos M. Parlados. *Handbook of Combinatorial Optimization: Supplement, Volume 1.* Pages 240-241.

[7] Irit Dinut, Samuel Safra. *On the hardness of approximating minimum vertex cover.* 2005.

[8] Guy Even. Joseph Naor. Baruch Schieber. Madhu Sudan. *Approximating Minimum Feedback Sets and Multi-Cuts in Directed Graphs.* 1998.