Theory Comput Syst (2018) 62:1880–1909
https://doi.org/10.1007/s00224-018-9852-7

Parameterised Algorithms for Deletion to Classes of DAGs

Akanksha Agrawal[1] · Saket Saurabh[1,2,3] · Roohani Sharma[2,3] · Meirav Zehavi[4]

Published online: 28 February 2018

Abstract In the DIRECTED FEEDBACK VERTEX SET (DFVS) problem, we are given a digraph D on n vertices and a positive integer k, and the objective is to check whether there exists a set of vertices S such that $F = D - S$ is an acyclic digraph. In a recent paper, Mnich and van Leeuwen [*STACS 2016*] studied the kernelization complexity of DFVS with an additional restriction on F—namely that F must be an out-forest, an out-tree, or a (directed) pumpkin—with an objective of shedding some light on the kernelization complexity of the DFVS problem, a well known open problem in the area. The vertex deletion problems corresponding to obtaining an out-forest, an out-tree, or a (directed) pumpkin are OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET, respectively. They showed that OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET admit polynomial kernels. Another open problem regarding DFVS is that, does DFVS admit an algorithm with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$? We complement the kernelization

✉ Roohani Sharma
roohani@imsc.res.in

Akanksha Agrawal
akanksha.agrawal@uib.no

Saket Saurabh
saket@imsc.res.in

Meirav Zehavi
meiravze@bgu.ac.il

[1] University of Bergen, Bergen, Norway

[2] Institute of Mathematical Sciences, Chennai, HBNI, India

[3] UMI ReLax, Chennai, India

[4] Ben-Gurion University, Beersheba, Israel

programme of Mnich and van Leeuwen by designing fast FPT algorithms for the above mentioned problems. In particular, we design an algorithm for OUT-FOREST VERTEX DELETION SET that runs in time $\mathcal{O}(2.732^k n^{\mathcal{O}(1)})$ and algorithms for PUMPKIN/OUT-TREE VERTEX DELETION SET that runs in time $\mathcal{O}(2.562^k n^{\mathcal{O}(1)})$. As a corollary of our FPT algorithms and the recent result of Fomin et al. [*STOC 2016*] which gives a relation between FPT algorithms and exact algorithms, we get exact algorithms for OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET that run in time $\mathcal{O}(1.633^n n^{\mathcal{O}(1)})$, $\mathcal{O}(1.609^n n^{\mathcal{O}(1)})$ and $\mathcal{O}(1.609^n n^{\mathcal{O}(1)})$, respectively.

Keywords Out-forest · Out-tree · Pumpkin · Fixed parameter tractability · branching · Bounded search trees

1 Introduction

FEEDBACK SET problems are fundamental combinatorial optimisation problems. In these problems, we are given a graph D (directed or undirected) and a positive integer k, and the question is whether there exists a set S of vertices (edges/arcs) in $V(D)$ ($E(D)$) such that the graph obtained after deleting the vertices (edges/arcs) in S is a forest (directed acyclic graph). The vertex deletion versions of FEEDBACK SET problems are called UNDIRECTED FEEDBACK VERTEX SET (UFVS) and DIRECTED FEEDBACK VERTEX SET (DFVS), based on whether the input is an undirected graph or a digraph. Similarly, the edge versions of FEEDBACK SET problems are called UNDIRECTED FEEDBACK EDGE SET (UFES) and DIRECTED FEEDBACK ARC SET (DFAS). All these problems, except UNDIRECTED FEEDBACK EDGE SET, are known to be NP-complete. Furthermore, FEEDBACK SET problems are amongst Karp's original list of 21 NP-complete problems and have been the topic of active research from algorithmic [2, 4–10, 12, 13, 19, 21, 22, 24, 26, 30, 35] as well as structural point of view [18, 23, 25, 29, 32–34]. In particular, such problems constitute one of the most important topics of research in parameterised algorithms [6, 8–10, 12, 13, 22, 24, 26, 30, 35], spearheading the development of several new techniques. In this paper we will focus on restrictions of DFVS in the realm of parameterised algorithms.

In parameterised complexity each problem instance is accompanied by a parameter k, and a problem is FPT if instances (I, k) of the problem are solvable in time $\gamma(k)|I|^{\mathcal{O}(1)}$, where γ is an arbitrary function of k. Moreover, a parameterised problem is said to admit a *polynomial kernel* if there is a polynomial time algorithm, called a *kernelization algorithm*, that reduces the input instance to an instance whose size is bounded by a polynomial function in k, say $p(k)$, whilst preserving the answer. This reduced instance is called a $p(k)$ kernel for the problem. For more information we refer the reader to monographs such as [11, 17].

UFVS was amongst the first few problems for which an FPT algorithm was designed [27]. On the other hand the parameterised complexity of DFVS (equivalently, DFAS) was a well-known open problem in the area until 2008, when Chen et al. [9] showed that DFVS and DFAS are FPT and admit an algorithm with running time $2^{\mathcal{O}(k \log k)} n^{\mathcal{O}(1)}$. Since then, there has been very little progress on DFVS.

In particular, the following two questions regarding DFVS are considered to be challenging open problems in the area.

Question 1: Does DFVS admit a polynomial kernel?
Question 2: Does DFVS admit an algorithm with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$? That is, does DFVS admit a single exponential time algorithm?

The lack of progress on these problems led to consideration of various restrictions on input instances. In particular, we know of polynomial kernels for DFVS on tournaments and its various generalisations [1, 3, 15]. However, these problems are open even for planar digraphs.

Recently, in a very interesting article, to make progress on Question 1, Mnich and van Leeuwen [28] considered DFVS with an additional restriction on the *output rather than the input*. In particular, the basic philosophy of their programme is the following: what happens to the kernelization complexity of DFVS when we consider subclasses of directed acyclic graphs. Towards this end, they considered the class of out-forests, out-trees and (directed) pumpkins. An *out-tree* is a digraph where each vertex has in-degree at most 1 and the underlying (undirected) graph is a tree. An *out-forest* is a disjoint union of out-trees. On the other hand, a digraph is a *pumpkin* if it consists of a source vertex s and a sink vertex t ($s \neq t$), together with a collection of internally vertex-disjoint induced directed paths from s to t. Furthermore all the vertices except s and t have in-degree 1 and out-degree 1. All these are subclasses of directed acyclic graphs. The classes of out-forests and out-trees were motivated by the corresponding question of UFVS and TREE DELETION SET in the undirected settings. In particular these classes led to the following problems.

OUT-FOREST VERTEX DELETION SET (OFVDS)
Input: A directed graph D and a positive integer k.
Parameter: k
Question: Is there a set $S \subseteq V(D)$ of size at most k such that $F = D - S$ is an out-forest?

We similarly define OUT-TREE VERTEX DELETION SET (OTVDS) and PUMPKIN VERTEX DELETION SET (PVDS). Mnich and van Leeuwen [28] showed that OFVDS/OTVDS admit kernel of size $\mathcal{O}(k^3)$ and PVDS admits a kernel of size $\mathcal{O}(k^{18})$.

1.1 Our Results and Methods

The objective of this article to ask Question 2 in the framework introduced by Mnich and van Leeuwen [28]. Towards this we obtain the following results.

- OFVDS admits an algorithm with running time $\mathcal{O}^\star((1+\sqrt{3})^k)$.
- OTVDS and PVDS admit algorithms with running time $\mathcal{O}^\star(2.562^k)$.

An underlying undirected graph for both an out-forest and a out-tree does not contain a cycle and thus they are very close to forest and tree, respectively. Thus, it is imperative on us to compare the running time of our FPT algorithms for OFVDS

and OTVDS with UFVS and TREE DELETION SET (TDS), respectively. The best known deterministic and randomised algorithms for UFVS run in times $\mathcal{O}^\star(3.619^k)$ and $\mathcal{O}^\star(3^k)$, respectively [12, 26]. It is a well-known open problem in the area whether one can achieve a deterministic algorithm for UFVS matching the running time of the randomised algorithm. On the other hand the best known algorithm for TDS runs in time $\mathcal{O}^\star(5^k)$ [31]. In contrast to these results, we obtain deterministic algorithms for OFVDS and OTVDS with running times $\mathcal{O}^\star((1+\sqrt{3})^k) = \mathcal{O}^\star(2.732^k)$ and $\mathcal{O}^\star(2.562^k)$ respectively. All these results are summarized in Table 1. The main observation which enables us to design all our FPT algorithms is the following:

> Every vertex, except one (in the case of PVDS), in an out-tree, an out-forest or pumpkin has in-degree at most one; in pumpkin every vertex except s and t has in-degree one and out-degree one. Thus for any vertex v, amongst v and any two of its in-neighbours, say x and y, one of the vertices must belong to the deletion set. We will call this *3-hitting set* property. This results is a 3 way branching. When such a structure does not exist, in the case of OUT-FOREST and OUT-TREE vertex deletion set problems, each vertex has in-degree at most 1, and in the case of the PUMPKIN vertex deletion set problem, each vertex, except for one, has indegree at most 1. In the former case, observe that, when each vertex in a digraph has in-degree at most 1, then the digraph is a disjoint collection of out-trees and directed cycles. In the later case, if the input digraph is a YES instance, then the resulting digraph is a disjoint collection of a pumpkin, out-trees and directed cycles. In either case, the corresponding problem on the resulting digraph can be solved in polynomial time. This already gives us deterministic algorithms with running time $\mathcal{O}^\star(3^k)$ for OFVDS, OTVDS, and PVDS. We exploit this hitting set property together with the fact that out-forest, out-tree and pumpkin are all almost acyclic graphs, to design FPT algorithms faster than $\mathcal{O}^\star(3^k)$.

Here we would also like to mention the corresponding *extension* problems of the problems considered in the article, which we define as follows.

OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET - EXTENSION
Input: A directed graph D, $X \subseteq V(D)$ and a positive integer k.
Parameter: k
Question: Is there a set $S \subseteq V(D) \setminus X$ of size at most k such that $F = D-(S \cup X)$ is an out-forest?

Table 1 Problems with the best known deterministic and randomised FPT running times

Problem	Deterministic	Randomised
UFVS	3.619^k	3^k
OFVDS	$(1+\sqrt{3})^k$	–
TDS	5^k	–
OTVDS	2.562^k	–
PVDS	2.562^k	–

Observe that if we have FPT algorithms for OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET then to solve instances (D, X, k) of OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET - EXTENSION problems, we can run the FPT algorithms of OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET problems on the instance $(D - X, k - |X|)$. Thus, FPT algorithms for OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET problems imply FPT algorithms for OUT-FOREST/OUT-TREE/PUMPKIN VERTEX DELETION SET - EXTENSION problems. Combining this observation with Theorem 2 of [20] (which roughly states that if there an FPT algorithm for a vertex-deletion extension problem with running time $c^k n^{\mathcal{O}(1)}$ then there is an exact algorithm for the corresponding vertex-deletion problem with running time $\left(2 - \frac{1}{c}\right)^n n^{\mathcal{O}(1)}$), we get exact algorithms with running time $1.634^n n^{\mathcal{O}(1)}$, $1.610^n n^{\mathcal{O}(1)}$ and $1.610^n n^{\mathcal{O}(1)}$ respectively, for OUT-FOREST/OUT-TREE/ PUMPKIN VERTEX DELETION SET.

2 Preliminaries

We denote the set of natural numbers from 1 to n by $[n]$, and describe running times of algorithms using the $\mathcal{O}^\star$ notation, which suppresses factors polynomial in the input size.

Graphs We use standard terminology from the book of Diestel [14] for those graph-related terms which are not explicitly defined here. A directed graph D is a pair $(V(D), E(D))$ such that $V(D)$ is a set of vertices and $E(D)$ is a set of ordered pairs of vertices. Throughout the paper, we consider simple directed graphs. The underlying undirected graph G of D is a pair $(V(G), E(G))$ such that $V(G) = V(D)$ and $E(G)$ is a set of unordered pairs of vertices such that $\{u, v\} \in E(G)$ if and only if either $(u, v) \in E(D)$ or $(v, u) \in E(D)$.

Let D be a directed graph. For any $v \in V(D)$, we denote by $N^-(v)$ the set of in-neighbours of v, that is, $N^-(v) = \{u \mid (u, v) \in E(D)\}$. Similarly, let $N^+(v)$ denote the set of out-neighbours of v, that is, $N^+(v) = \{u \mid (v, u) \in E(D)\}$. We denote the in-degree of a vertex v by $d^-(v) = |N^-(v)|$ and its out-degree by $d^+(v) = |N^+(v)|$. We say that $P = (u_1, \ldots, u_l)$ is a directed path in a directed graph D if $u_1, \ldots, u_l \in V(D)$ and for all $i \in [l-1]$, $(u_i, u_{i+1}) \in E(D)$. Let C be an induced subgraph of D. We say that C is a weakly connected component of D if and only if the underlying undirected graph of C is connected.

Branching Algorithm For all our purposes, a branching algorithm is an ordered list of reduction rules and branching rules. A reduction rule takes an instance of a problem and outputs another equivalent instance of the same problem. Two instances are equivalent only when one is a YES instance if and only if the other is a YES instance. Formally, a reduction rule is a polynomial time procedure replacing an instance (I, k) of a parameterised language L by a new one (I', k'). It is *safe* if $(I, k) \in L$ if and only if $(I', k') \in L$. A branching rule takes an instance (I, k) of a parameterised language L and produces several instances, $(I_1, k_1), \ldots, (I_l, k_l)$ of the same problem.

A branching rule is *exhaustive* if (I, k) is a YES instance if and only if at least one of $(I_1, k_1), \ldots, (I_l, k_l)$ is a YES instance.

The branching algorithm applies the reduction rules and branching rules in the following preference. A branching rule can be applied only when none of the reduction rules are applicable. Reduction rule i can be applied only when none of the reduction rules from 1 to $i - 1$ are applicable. Similarly, branching rule i is applicable only when none of the branching rules from 1 to $i - 1$ are applicable.

As mentioned before, for all the problems considered in this paper, an $\mathcal{O}^*(3^k)$ algorithm is trivial. The non-trivial running times are obtained by choosing an appropriate vertex to branch on. The order of the branching rules imposes a lot of structure in the cases that appear later. This structure is exploited to achieve the target running time.

Analysing the Running Time of Branching Algorithms—Bounded Search Trees The running time of a branching algorithm can be analysed as follows (see, e.g., [11, 16]). Suppose that the algorithm executes a branching rule which has ℓ branching options (each leading to a recursive call with the corresponding parameter value), such that, in the ith branch option, the current value of the parameter decreases by b_i. Then, $(b_1, b_2, \ldots, b_\ell)$ is called the *branching vector* of this rule. We say that α is the *root* of $(b_1, b_2, \ldots, b_\ell)$ if it is the (unique) positive real root of $x^{b^*} = x^{b^*-b_1} + x^{b^*-b_2} + \cdots + x^{b^*-b_\ell}$, where $b^* = \max\{b_1, b_2, \ldots, b_\ell\}$. If $r > 0$ is the initial value of the parameter, and the algorithm (a) returns a result when (or before) the parameter is negative, and (b) only executes branching rules whose roots are bounded by a constant $c > 0$, then its running time is bounded by $\mathcal{O}^*(c^r)$.

In all our algorithms, if there exists a pair of vertices u, v in the input digraph such that both (u, v) and (v, u) belong to the arc set of the digraph, then observe that at least one of u and v belong to the solution. Thus, we branch on such vertex pairs, reducing the budget by one in each branch and stopping whenever the budget is zero or negative. Henceforth, without loss of generality, we assume that in the input digraph, for any pair u, v of vertices there is at most of arcs (u, v) or (v, u).

3 FPT Algorithm for Pumpkin Vertex Deletion Set

In this section, we give a branching algorithm for Pumpkin Vertex Deletion Set. Let (D, k) be an instance of PVDS. The algorithm starts by guessing the source vertex s and the sink vertex t in the graph obtained after the deletion of the solution vertices. There are $O(|V(D)|^2)$ choices for s and t. After this guesswork, we want to solve the problem where we are given a digraph D, a positive integer k and two vertices s and t, and the question is does there exist a set S of the vertices of D of size at most k such that $D - S$ is a pumpkin with s as the source vertex and t as the sink vertex. We call this new problem Restricted Pumpkin Vertex Deletion Set (RPVDS) and in the rest of this section we develop an algorithm for solving it. This algorithm together with the guessing step will give an FPT algorithm for PVDS. Formally, RPVDS is defined as follows.

RESTRICTED PUMPKIN VERTEX DELETION SET (RPVDS)
Input: A directed graph D, two distinct vertices $s, t \in V(D)$ and a positive integer k.
Parameter: k
Question: Is there a set $S \subseteq V(D)$ of size at most k such that $D - S$ is a pumpkin with s as the source vertex and t as the sink vertex?

Broadly speaking, our strategy is to branch on vertices having in-degree at least 2 or out-degree at least 2. The reduction rules and branching rules are so designed that when none of them is applicable, all vertices in the resulting digraph, except for s and t, have in-degree exactly 1 and out-degree exactly 1 and, s has in-degree 0 and t has out-degree 0. Such an instance becomes trivial to solve. To achieve this trivial instance, the algorithm systematically deals with vertices that do not satisfy the constraints of this trivial instance.

We now give the formal description of the algorithm. The measure μ that will be used to bound the depth of the search tree of our branching algorithm is the solution size, that is, $\mu(D, k, s, t) = k$.

With a slight abuse of notation, in the following, during the application of any reduction/branching rule we will refer to (D, k, s, t) as the instance that is reduced with respect to the rules in higher preference order. We first list the reduction rules used by the algorithm.

Reduction Rule 1 *If $k < 0$, return (D, k, s, t) is a NO instance.*

Reduction Rule 2 *If $k = 0$ and D is not a pumpkin with source s and sink t, return (D, k, s, t) is a NO instance.*

Reduction Rule 3 *If $k \geq 0$ and D is a pumpkin with source s and sink t, return (D, k, s, t) is a YES instance.*

Reduction Rule 4 *If there exists $v \in V(D) \setminus \{s\}$ such that $d^-(v) = 0$ then delete v from D and decrease k by 1. That is, the resulting instance is $(D - \{v\}, k - 1, s, t)$.*

Reduction Rule 5 *If there exists $v \in V(D) \setminus \{t\}$ such that $d^+(v) = 0$ then delete v from D and decrease k by 1. That is, the resulting instance is $(D - \{v\}, k - 1, s, t)$.*

Reduction Rule 6 *If there exists $v \in V(D)$ such that $s \in N^+(v)$ then delete v from D and decrease k by 1. That is, the resulting instance is $(D - \{v\}, k - 1, s, t)$.*

Reduction Rule 7 *If there exists $v \in V(D)$ such that $t \in N^-(v)$ then delete v from D and decrease k by 1. That is, the resulting instance is $(D - \{v\}, k - 1, s, t)$.*

Reduction Rule 8 *If C is a weakly connected component of D such that either $s \notin V(C)$ or $t \notin V(C)$, then the resulting instance is $(D - V(C), k - |C|, s, t)$.*

Reduction Rule 9 *If $s \notin V(D)$ or $t \notin V(D)$, return (D, k, s, t) is a NO instance.*

It is easy to see that reduction rules 1 to 9 are safe and can be applied in polynomial time.

We now describe the branching rules used by the algorithm. The algorithm branches on some vertex based on its in-degree and/or out-degree as per one of the 5 branching rules described later. Before giving the details of the branching rules, we first mention the invariants maintained by the algorithm after the exhaustive application of each of the branching rules.

More precisely, the invariant maintained at the end of branching rule i is a condition which always holds when none of the reduction rules and branching rules 1 to i are applicable.

Branching Rule 1: For all $v \in V(D) \setminus \{s,t\}$, if $d^-(v) \geq 2$, then $s \notin N^-(v)$. Symmetrically, if $d^+(v) > 1$, then $t \notin N^+(v)$.

Branching Rule 2: For all $v \in V(D) \setminus \{s,t\}$, $d^-(v) \leq 2$. Symmetrically, $d^+(v) \leq 2$.

Branching Rule 3: For all $v \in V(D) \setminus \{s,t\}$, if $d^-(v) = 2$, then $d^+(v) \leq 1$. Symmetrically, if $d^+(v) = 2$, then $d^-(v) \leq 1$.

Branching Rule 4: For all $v \in V(D) \setminus \{s,t\}$, if $d^-(v) = 2$ and $d^+(v) = 1$, then $N^+(v) = \{t\}$. Symmetrically, if $d^+(v) = 2$ and $d^-(v) = 1$, then $N^-(v) = \{s\}$.

Branching Rule 5: For all $v \in V(D) \setminus \{s,t\}$, if $d^-(v) = 2$ then $d^+(v) = 0$. Symmetrically, if $d^+(v) = 2$, then $d^-(v) = 0$.

We now give the description of the branching rules. The cases under which each of these branching rules is applied is pictorially depicted in Fig. 1. From the description of the branching rules it is easy to see that each branching rule is exhaustive.

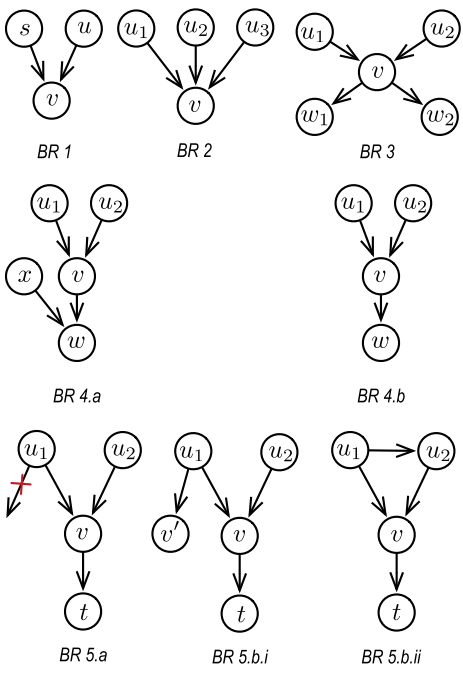

Fig. 1 The structures in the branching rules of PVDS

Branching Rule 1 If there is $v \in V(D)$ such that $d^-(v) \geq 2$ (or $d^+(v) \geq 2$) and $s \in N^-(v)$ (or $t \in N^+(v)$), then let u be the other in-neighbour (or out-neighbour) of v. In this case the algorithm branches as follows.

- When v belongs to the solution, then the resulting instance is $(D-\{v\}, k-1, s, t)$.
- When v does not belong to the solution, then u must belong to the solution as otherwise, v, u and s (or t) will belong to the pumpkin obtained after the solution vertices are deleted, which is not possible. Therefore, the resulting instance is $(D - \{u\}, k - 1, s, t)$.

The branching vector for this rule is (1, 1).

Branching Rule 2 If there is $v \in V(D)$ such that $d^-(v) \geq 3$ (or $d^+(v) \geq 3$), $v \neq t$ (or $v \neq s$), then let u_1, u_2, u_3 be some three distinct in-neighbours (or out-neighbours) of v. Note that none of u_1, u_2, u_3 is the same as s (or t), as otherwise branching rule 1 would be applicable. In this case, the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, s, t)$.
- When v does not belong to the solution, then note that at least 2 of u_1, u_2 and u_3 must belong to the solution. Thus, the algorithm further branches as follows.
  - When u_1 and u_2 belong to the solution, the resulting instance is $(D - \{u_1, u_2\}, k - 2, s, t)$.
  - When u_2 and u_3 belong to the solution, the resulting instance is $(D - \{u_2, u_3\}, k - 2, s, t)$.
  - When u_1 and u_3 belong to the solution, the resulting instance is $(D - \{u_1, u_3\}, k - 2, s, t)$.

The branching vector for this rule is (1, 2, 2, 2).

Branching Rule 3 If there is $v \in V(D) \setminus \{s, t\}$ such that $d^-(v) = 2$ and $d^+(v) = 2$, then let u_1, u_2 be the in-neighbours of v and w_1, w_2 be the out-neighbours of v. Observe that neither u_1 nor u_2 is the same as t, as otherwise branching rule 1 would be applicable. Similarly, neither w_1 nor w_2 is the same as s. In this case, the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, s, t)$.
- When v does not belong to the solution, then note that at least one of u_1 or u_2 and, at least one of w_1 or w_2 must belong to the solution. Thus, the algorithm further branches as follows.
  - When u_1 and w_1 belong to the solution, the resulting instance is $(D - \{u_1, w_1\}, k - 2, s, t)$.
  - When u_1 and w_2 belong to the solution, the resulting instance is $(D - \{u_1, w_2\}, k - 2, s, t)$.
  - When u_2 and w_1 belong to the solution, the resulting instance is $(D - \{u_2, w_1\}, k - 2, s, t)$.
  - When u_2 and w_2 belong to the solution, the resulting instance is $(D - \{u_2, w_2\}, k - 2, s, t)$.

The branching vector for this rule is (1, 2, 2, 2, 2).

Branching Rule 4 If there is $v \in V(D) \setminus \{s, t\}$ such that $d^-(v) = 2$ (or $d^+(v) = 2$), $d^+(v) = 1$ (or $d^-(v) = 1$) and the unique out-neighbour (or in-neighbour), say w of v, is not t (or s), then let u_1, u_2 be the two in-neighbours (or out-neighbours) of v. Observe that none of u_1 or u_2 is the same as s. The algorithm considers the following cases depending on the in-degree (or out-degree) of w.

Case 4.a If $d^-(w) = 2$ (or $d^+(w) = 2$), then let x be the other in-neighbour (or out-neighbour) of w different from v. Here x may or may not be equal to u_1 or u_2. In this case, the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, s, t)$.
- When v does not belong to the solution, then note that w does not belong to the solution (because if it does, then v must be a sink vertex and hence $v = t$, which is not true). Since $w \neq t$, and both v and w do not belong to the solution, x must belong to the solution. Therefore, the resulting instance is $(D - \{x\}, k - 1, s, t)$.

The branching vector in this case is (1, 1).

Case 4.b If $d^-(w) = 1$ (or $d^+(w) = 1$), the algorithm branches as follows.

- When v belongs to the solution, then delete v from the graph. In the resulting graph $d^-(w) = 0$. Since $w \neq s$, reduction rule 4 is applicable. Therefore, the resulting instance is $(D - \{v, w\}, k - 2, s, t)$.
- When v does not belong to the solution, then at least one of u_1 or u_2 must belong to the solution. Hence, the algorithm branches further into 2 cases. In the first case, the resulting instance is $(D - \{u_1\}, k - 1, s, t)$ and in the second case the resulting instance is $(D - \{u_2\}, k - 1, s, t)$.

The branching vector in this case is (2, 1, 1).

Branching Rule 5 If there is $v \in V(D) \setminus \{s, t\}$ such that $d^-(v) = 2$ (or $d^+(v) = 2$), $d^+(v) = 1$ (or $d^-(v) = 1$) and t (or s) is the unique out-neighbour (or in-neighbour) of v, then let u_1, u_2 be the two in-neighbours (or out-neighbours) of v. Observe that none of u_1 or u_2 is same as s (or t), otherwise branching rule 1 would be applicable. The algorithm considers the following cases based on the out-degree (or in-degree) of u_1 and u_2.

Case 5.a If either $d^+(u_1) = 1$ or $d^+(u_2) = 1$ ($d^-(u_1) = 1$ or $d^-(u_2) = 1$), then without loss of generality assume that $d^+(u_1) = 1$ (or $d^-(u_1) = 1$).

Observe that u_1 is distinct from t (or s) because u_1 is an in-neighbour (or out-neighbour) of v and t (or s) is an out-neighbour of v. In this case, the algorithm branches as follows.

- When u_1 belongs to the solution, then delete u_1 from the graph. Thus the resulting instance is $(D - \{u_1\}, k - 1, s, t)$.

- When u_1 does not belong to the solution, since $d^+(u_1) = 1$, and $N^+(u_1) = \{v\}$, v does not belong to the solution. Since $v \neq t$, and $u_2 \in N^-(v)$, u_2 belongs to the solution. Thus, the resulting instance is $(D - \{u_2\}, k - 1, s, t)$.

The branching vector for this case is (1, 1).

Case 5.b If $d^+(u_1) = 2$ and $d^+(u_2) = 2$ ($d^-(u_1) = 2$ and $d^-(u_1) = 2$), let v' be the other out-neighbour (or in-neighbour) of u_1. In this case, the algorithm considers the following sub-cases.

Sub-case 5.b.i If v' is different from u_2 and t, then the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, s, t)$.
- When v does not belong to the solution, then observe that at least one of u_1 and u_2 must belong to the solution. Thus, the algorithm branches as follows.
  - When u_1 belongs to the solution, the resulting instance is $(D - \{u_1\}, k - 1, s, t)$.
  - When u_1 does not belong to the solution, then u_2 must belong to the solution. Also v' must belong to the solution (because u_1 is distinct from s). Therefore, the resulting instance is $(D - \{u_2, v'\}, k - 2, s, t)$.

The branching vector for this case is (1, 1, 2).

Sub-case 5.b.ii If v' is the same as t, the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, s, t)$.
- When v does not belong to the solution then u_1 must belong to the solution (because $u_1 \neq s$). Therefore, the resulting instance is $(D - \{u_1\}, k - 1, s, t)$.

The branching vector for this case is (1, 1).

Sub-case 5.b.iii If v' is the same as u_2, then we know that $d^+(u_2) = 2$, and therefore, one of case 5.a.i or case 5.a.ii would be applicable.

This ends the description of the branching rules. In the upcoming lemma, we show that when all the reduction rules and branching rules have been applied exhaustively, that is when none of them is no longer applicable, all the vertices of the resulting instance, possibly except for s and t, have in-degree exactly 1 and out-degree exactly 1. In the later lemma we show that such an instance can be solved in polynomial time. Thus, after the exhaustive application of the reduction rules and the above mentioned branching rules, the algorithm uses the procedure described in Lemma 2 to solve the instance.

Lemma 1 *Let (D, k, s, t) be the instance where none of the above-mentioned reduction rules or branching rules are applicable. Then, for all $v \in V(D) \setminus \{s, t\}$, $d^-(v) = 1$, $d^+(v) = 1$, $d^-(s) = 0$ and $d^+(t) = 0$.*

Proof Since reduction rules 6 and 7 are no longer applicable, we have $d^-(s) = 0$ and $d^+(t) = 0$. Also since reduction rules 4 and 5 are not applicable, $d^-(v) > 0$ and $d^+(v) > 0$, for all $v \in V(D) \setminus \{s, t\}$. To show that for any vertex $v \in V(D) \setminus \{s, t\}$, $d^-(v) = 1$ we proceed as follows.

Let v be some vertex in $V(D) \setminus \{s, t\}$. For the sake of contradiction, let $d^-(v) > 1$. If $d^-(v) > 1$ and $s \in N^-(v)$, then branching rule 1 would be applicable. Thus, we can safely assume that $s \notin N^-(v)$.

We split the situation that $d^-(v) > 1$, into 2 cases as follows.

- If $d^-(v) \geq 3$, then branching rule 2 would be applicable.
- Otherwise $1 < d^-(v) \leq 2$, that is, $d^-(v) = 2$. We split this situation into the following 3 exhaustive cases.
  - If $d^+(v) \geq 2$, then branching rule 3 would be applicable.
  - If $d^+(v) = 1$, then based on whether the unique out-neighbour of v is t or not, either branching rule 4 or branching rule 5 would be applicable.
  - If $d^+(v) = 0$, then reduction rule 5 would be applicable.

Since, none of the reduction rules or branching rules are applicable, we conclude that for all $v \in V(D) \setminus \{s, t\}$, $d^-(v) = 1$. Using the same case analysis we can show that for any vertex $v \in V(D) \setminus \{s, t\}$, $d^+(v) = 1$. □

Lemma 2 *Let (D, k, s, t) be an instance of* Restricted Pumpkin Vertex Deletion Set *such that for all $v \in V(D) \setminus \{s, t\}$, $d^-(v) \leq 1$, $d^+(v) \leq 1$ and, $d^-(s) = 0$ and $d^+(t) = 0$. Then the instance (D, k, s, t) can be solved in $\mathcal{O}(n^{\mathcal{O}(1)})$ time, where $n = |V(D)|$.*

Proof Since reduction rule 8 and 9 are not applicable, D is weakly connected and both s and t belong to D. Since, for all $v \in V(D) \setminus \{s, t\}$, $d^-(v) = 1$, $d^+(v) = 1$ and, $d^-(s) = 0$ and $d^+(t) = 0$, and observe that (D, k, s, t) is a YES instance if and only if D is a pumpkin with s as the source vertex, t as the sink vertex and $k \geq 0$. □

In the next lemma, we formally prove the correctness of our algorithm.

Lemma 3 *The algorithm presented for* Restricted Pumpkin Vertex Deletion Set *is correct.*

Proof Let $I = (D, k, s, t)$ be an instance of RPVDS. We prove the correctness of the algorithm by induction on $\mu = \mu(I) = k$. The base case occurs in one of the following cases.

- If one of s or t does not belong to $V(D)$, the algorithm correctly concludes that (D, k, s, t) is a NO instance from reduction rule 8.
- If $\mu <= 0$, the algorithm correctly concludes whether (D, k, s, t) is a yes instance or not by reduction rules 1 to 3.
- If $\mu >= 0$ and D is a pumpkin, the algorithm correctly concludes that (D, k, s, t) is a YES instance.
- If $\mu \geq 0$ and for all $v \in V(D) \setminus \{s, t\}$, $d^-(v) = 1$, $d^+(v) = 1$ and, $d^-(s) = 0$, $d^+(t) = 0$, then from Lemma 1 the algorithm solves the instance correctly.

By induction hypothesis we assume that for all $\mu \leq l$, the algorithm is correct. We will now prove that the algorithm is correct when $\mu = l + 1$. The algorithm performs one of the following actions. If possible, it applies one of the reduction rules. By the safeness of the reduction rules we either correctly conclude that I is a YES/NO instance or produce an equivalent instance I' with $\mu(I') \leq \mu(I)$. If $\mu(I') < \mu(I)$, then by induction hypothesis and safeness of the reduction rules the algorithm correctly decides if I is a YES instance or not. Otherwise, $\mu(I') = \mu(I)$. If none of the reduction rules are applicable then the algorithm applies the first applicable Branching Rule. If some branching rule is applicable then, since μ decreases in each branch by at least one, by induction hypothesis the algorithm correctly concludes that I is a YES / NO instance. If none of the branching rules is applicable, then from Lemma 2, if $I = (D, k, s, t)$, then for all $v \in V(D) \setminus \{s, t\}$, $d^-(v) = 1$, $d^+(v) = 1$ and, $d^-(s) = 0$, $d^+(t) = 0$. Thus, in this situation we handle a base case correctly and hence, we conclude that the algorithm always outputs the correct answer. □

Next, we analyse the running time of our algorithm.

Theorem 1 *The presented algorithm solves* RESTRICTED PUMPKIN VERTEX DELETION SET *in time* $\mathcal{O}^\star(2.562^k)$.

Proof Observe that reduction rules 1 to 9 can be applied in time polynomial in the input size and are never applied more than polynomial number of times. Also, at each of the branches we spend a polynomial amount of time. At each of the recursive calls in a branch, the measure μ decreases by at least by 1. When $\mu \leq 0$, then we are able to solve the instance in polynomial time or correctly conclude that the corresponding branch cannot lead to a solution. At the start of the algorithm $\mu = k$.

The worst-case branching vector for the algorithm is $(1, 2, 2, 2, 2)$ (see Table 2). The recurrence for the worst case branching vector is

$$T(\mu) \leq T(\mu - 1) + 4T(\mu - 2)$$

The running time corresponding to the above recurrence relation is $\mathcal{O}^\star(2.562^k)$. □

Table 2 The branch vectors and their corresponding running times for RPVDS

Branching Rule (BR)	Case/Sub-Case	Branch vector	c^μ
BR 1		(1, 1)	2^μ
BR 2		(1, 2, 2, 2)	2.30278^μ
BR 3		(1, 2, 2, 2, 2)	2.56156^μ
BR 4	a	(1, 2, 1)	2.41422^μ
	b	(2, 1, 1)	2.41422^μ
BR 5	a	(1, 1)	2^μ
	b.i	(1, 1, 2)	2.41422^μ
	b.ii	(1, 1)	2^μ

As mentioned at the starting of the section, given an instance (D, k) of PVDS, one can design an algorithm for PVDS which guesses the source and sink vertices, s and t respectively, of the resulting pumpkin and for each such guess runs the algorithm for RPVDS on (D, k, s, t). Note that (D, k) is a YES instance of PVDS if and only if (D, k, s, t) is a YES instance of RPVDS for some guess of s and t. Since there are at most $|V(D)|^2$ choices for the pair s and t, Theorem 1 gives us the following theorem.

Theorem 2 PUMPKIN VERTEX DELETION SET *can be solved in time* $\mathcal{O}^\star(2.562^k)$.

4 FPT Algorithm for OUT-TREE VERTEX DELETION SET

In this section, we give a branching algorithm for OUT-TREE VERTEX DELETION SET (OTVDS). In the broader picture, this algorithm is in the same spirit as the one for PVDS with a difference only in the details of the reduction rules and branching rules.

Let (D, k) be an instance of OTVDS. The algorithm starts by guessing the root vertex r of the out-tree obtained after the deletion of the solution vertices. Note that there are $|V(D)|$ choices for r. After this guesswork, we would like to solve the following problem. Given a digraph D, an integer k and a vertex r of the digraph, does there exist a set of at most k vertices whose deletion results in an out-tree with root r. We call this new problem RESTRICTED OUT-TREE VERTEX DELETION SET and give an FPT algorithm for this problem parameterised by the solution size. Note that the algorithm for this new problem combined with the original guess of the vertex r gives an FPT algorithm for OTVDS. Formally, RESTRICTED OUT-TREE VERTEX DELETION SET is defined as follows.

RESTRICTED OUT-TREE VERTEX DELETION SET (ROTVDS)
Input: A directed graph D, a vertex $r \in V(D)$ and a positive integer k.
Parameter: k
Question: Is there a set $S \subseteq V(D)$ of size at most k such that $D - S$ is an out-tree with r as the root vertex?

We first give an outline of the algorithm for ROTVDS. Let (D, k, r) be an instance of OTVDS. The reduction rules and branching rules for this algorithm are so designed that after the exhaustive application of these rules, all vertices in the resulting instance, except r, have in-degree exactly 1 and the in-degree of r is 0. Such an instance becomes trivial to solve. To achieve this trivial instance, the algorithm systematically deals with vertices that do not satisfy the constraints of this trivial instance.

We now give the formal description of the algorithm. The measure μ that will be used to bound the depth of the search tree of our branching algorithm is the solution size, that is, $\mu(D, k, s, t) = k$. With a slight abuse of notation, in the following, during the application of any reduction/branching rule we will refer to (D, k, s, t) as the instance that is reduced with respect to the rules in higher preference order. We first list the reduction rules used by the algorithm.

Reduction Rule 1 *If $k < 0$, then (D, k, r) is a NO instance.*

Reduction Rule 2 *If $k = 0$ and D is not an out-tree, return (D, k, r) is a NO instance.*

Reduction Rule 3 *If $k \geq 0$ and D is an out-tree, return (D, k, r) is a YES instance.*

Reduction Rule 4 *If there exists $v \in V(D) \setminus \{r\}$ such that $d^-(v) = 0$ then delete v from D and decrease k by 1. That is, the resulting instance is $(D - \{v\}, k - 1, r)$.*

Reduction Rule 5 *If there exists $v \in V(D)$ such that $r \in N^+(v)$ then delete v from D and decrease k by 1. That is, the resulting instance is $(D - \{v\}, k - 1, r)$.*

Reduction Rule 6 *If there exists a weakly connected component C not containing r, then delete all the vertices in C. That is, the resulting instance is $(D - V(C), k - |V(C)|, r)$.*

Reduction Rule 7 *If $r \notin V(D)$, return (D, k, r) is a NO instance.*

The algorithm applies reduction rules 1 to 7 (in order) exhaustively. It is easy to see that reduction rules 1 to 7 are safe and can be applied in polynomial time.

We now describe the branching rules used by the algorithm. The algorithm branches on some vertex based on its in-degree and/or out-degree as per one of the 5 branching rules described later. Before giving the details of the branching rules, we first mention the invariants maintained by the algorithm after the exhaustive application of each of the branching rules.

Branching Rule 1: For all $v \in V(D) \setminus \{r\}$, if $d^-(v) \geq 2$, then $r \notin N^-(v)$.
Branching Rule 2: For all $v \in V(D) \setminus \{r\}$, $d^-(v) \leq 2$.
Branching Rule 3: For all $v \in V(D) \setminus \{r\}$, if $d^-(v) = 2$, then $d^+(v) = 0$.
Branching Rule 4: For all $v \in V(D) \setminus \{r\}$, if $d^-(v) = 2$ and $d^+(v) = 0$, then for all $u \in V(D)$, $u \neq v$, if $d^-(u) = 2$ and $d^+(u) = 0$, then v and u have no common in-neighbour.
Branching Rule 5: For all $v \in V(D) \setminus \{r\}$, $d^-(v) \leq 1$.

We now give the description of the branching rules. The cases under which each branching rule is applicable is described in Fig. 2. From the description of the branching rules it is easy to see that each branching rule is exhaustive.

Branching Rule 1 If there exists $v \in V(D)$ such that $d^-(v) \geq 2$ and $r \in N^-(v)$, let u be one of the other in-neighbours of v. In this case, the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, r)$.
- When v does not belong to the solution, then u must belong to the solution. Therefore, the resulting instance is $(D - \{u\}, k - 1, r)$.

The branching vector for this rule is (1, 1).

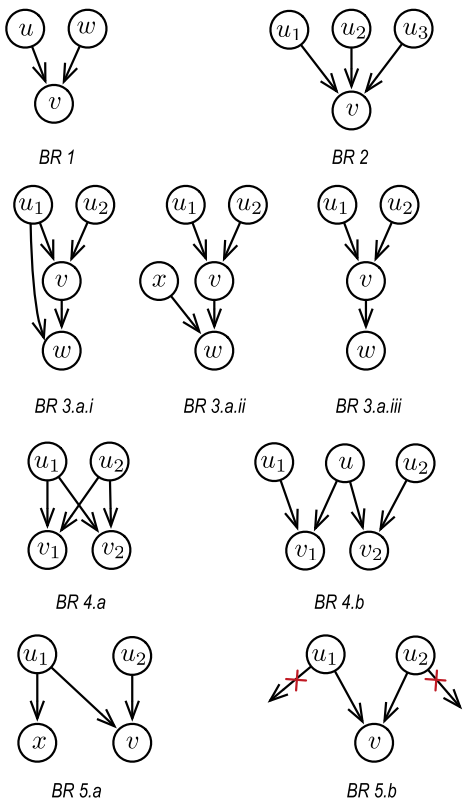

Fig. 2 The structures in the branching rules of OTVDS

Branching Rule 2 (**In-degree at least 3 Rule**) If there is $v \in V(D)$ such that $d^-(v) \geq 3$, let u_1, u_2, u_3 be some distinct in-neighbours of v. Note that none of u_1, u_2, u_3 is same as r, as otherwise branching rule 1 would be applicable. In this case, the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, r)$.
- When v does not belong to the solution then observe that at least 2 of u_1, u_2, u_3 must belong to the solution. Thus, the algorithm branches as follows.
  - When u_1 and u_2 belong to the solution, the resulting instance is $(D - \{u_1, u_2\}, k - 2, r)$.
  - When u_2 and u_3 belong to the solution, the resulting instance is $(D - \{u_2, u_3\}, k - 2, r)$.
  - When u_1 and u_3 belong to the solution, the resulting instance is $(D - \{u_1, u_3\}, k - 2, r)$.

The branching vector for this rule is $(1, 2, 2, 2)$.

After the exhaustive application of reduction rule 1 to 7 and when branching rules 1 and 2 are no more applicable, for all $v \in V(D) \setminus \{r\}$, $1 \leq d^-(v) \leq 2$ and $d^-(r) = 0$.

Branching Rule 3 If $v \in V(D)$ be such that $d^-(v) = 2$ and $d^+(v) \geq 1$, let u_1, u_2 be the two in-neighbours and w be one of the out-neighbours of v. The algorithm considers the following cases depending on the in-degree of w.

Case 3.a If $d^-(w) = 2$, let x be the other in-neighbour of w. The algorithm further considers the following sub-cases.

Sub-case 3.a.i If x is the same as u_1 (symmetrically u_2), then the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, r)$.
- When v does not belong to the solution, the algorithm branches on u_1 as follows.
  - When u_1 belongs to the solution, the resulting instance is $(D - \{u_1\}, k - 1, r)$.
  - When u_1 does not belong to the solution, then observe that both u_2 and w must belong to the solution. Thus, the resulting instance is $(D - \{u_2, w\}, k - 2, r)$.

The branching vector for this case is $(1, 1, 2)$.

Sub-case 3.a.ii Otherwise, x is distinct from u_1 and u_2. In this case, the algorithm branches as follows.

- When v belongs to the solution, the resulting instance is $(D - \{v\}, k - 1, r)$.
- When v does not belong to the solution then observe that at least one of u_1, u_2 and at least one of w, x must belong to the solution. Thus, the algorithm branches as follows.
  - When u_1 and w belongs to the solution, the resulting instance is $(D - \{u_1, w\}, k - 2, r)$.
  - When u_1 and x belongs to the solution, the resulting instance is $(D - \{u_1, x\}, k - 2, r)$.
  - When u_2 and w belongs to the solution, the resulting instance is $(D - \{u_2, w\}, k - 2, r)$.
  - When u_2 and x belongs to the solution, the resulting instance is $(D - \{u_2, x\}, k - 2, r)$.

The branching vector for this case is $(1, 2, 2, 2, 2)$.

Case 3.b If $d^-(w) = 1$, then observe that w are r are distinct because $d^-(r) = 0$ (as reduction rule 5 is no longer applicable). In this case, the algorithm branches as follows.

- When v belongs to the solution, then delete v from the graph. Observe that in the resulting graph $d^-(w) = 0$ and hence, reduction rule 4 would be applicable. Therefore, the resulting instance in this case is $(D - \{v, w\}, k - 2, r)$.

- When v does not belong to the solution then branch on u_1 as follows.
  - When u_1 belongs to the solution, the resulting instance is $(D-\{u_1\}, k-1, r)$.
  - When u_1 does not belong to the solution, then u_2 must belong to the solution. Thus the resulting instance is $(D-\{u_2\}, k-1, r)$.

The branching vector in this case is (2, 1, 1).

We now consider the case when there are two vertices of in-degree 2 that have a common in-neighbour.

Branching Rule 4 If $v_1, v_2 \in V(D)$ such that $d^-(v_1) = d^-(v_2) = 2$, $d^+(v_1) = d^+(v_2) = 0$ and there exists at least one common in-neighbour of v_1 and v_2, then the algorithm considers the following sub-cases.

Case 4.a If v_1 and v_2 have two common in-neighbours, say u_1, u_2, then the algorithm branches as follows.

- When u_1 belongs to the solution, then the resulting instance is $(D-\{u_1\}, k-1, r)$.
- When u_1 does not belong to the solution, then the algorithm further branches as follows.
  - When u_2 belongs to the solution, then the resulting instance is $(D-\{u_2\}, k-1, r)$.
  - When u_2 does not belong to the solution, then observe that both v_1 and v_2 must belong to the solution. Thus the resulting instance is $(D-\{v_1, v_2\}, k-2, r)$.

The branching vector for this case is (1, 1, 2).

Case 4.b Otherwise, v_1, v_2 have exactly one common in-neighbour, say u, and x_1, x_2 are the other in-neighbours of v_1, v_2 respectively ($x_1 \neq x_2$). In this case, the algorithm branches as follows.

- When u belongs to the solution, the resulting instance is $(D-\{u\}, k-1, r)$.
- When u does not belong to the solution, then observe that at least one of v_1 or x_1, and at least one of v_2 or x_2 must belong to the solution. Therefore, the algorithm branches as follows.
  - When v_1, v_2 belong to the solution, the resulting instance is $(D-\{v_1, v_2\}, k-2, r)$.
  - When v_1, x_2 belong to the solution, the resulting instance is $(D-\{v_1, x_2\}, k-2, r)$.
  - When x_1, v_2 belong to the solution, the resulting instance is $(D-\{x_1, v_2\}, k-2, r)$.
  - When x_1, x_2 belong to the solution, the resulting instance is $(D-\{x_1, x_2\}, k-2, r)$.

The branching vector for this case is (1, 2, 2, 2, 2).

Hereafter, we assume that no two vertices of in-degree 2 have a common in-neighbour. We consider the following cases based on the degree of the in-neighbours of vertices with in-degree 2.

Branching Rule 5 If $v \in V(D)$ such that $d^-(v) = 2$ and $d^+(v) = 0$, let u_1, u_2 be the two in-neighbours of v. In this case, the algorithm considers the following sub-cases based on the out-degrees of u_1 and u_2.

Case 5.a If at least one of u_1, u_2 has out-degree at least 2 (without loss of generality, let $d^+(u_1) \geq 2$), let x be the other out-neighbour of u_1.

Observe that $d^-(x) = 1$, as otherwise x and v are two vertices of in-degree 2 and u_1 is a common in-neighbour of x and v, and hence, case 4 would be applicable. Also $x \neq r$ because $d^-(r) = 0$ otherwise reduction rule 5 would be applicable. In this case, the algorithm branches as follows.

- When u_1 belongs to the solution, then delete u_1 from the graph. In the resulting graph $d^-(x) = 0$ and hence reduction rule 4 is applicable. Thus the resulting instance is $(D - \{u_1, x\}, k - 2, r)$.
- When u_1 does not belong to the solution, then either u_2 belongs to the solution or v belongs to the solution. Therefore, the algorithm branches as follows. In the first branch the resulting instance is $(D - \{u_2\}, k - 1, r)$ and in the second branch the resulting instance is $(D - \{v\}, k - 1, r)$.

The branching vector for this case is (2, 1, 1).

Case 5.b Otherwise, both u_1 and u_2 have out-degree 1. In this case, we first prove that if there is an out-tree deletion set in D, say S, of size at most k such that $v \in S$ and $u_1, u_2 \notin S$, then $S' = (S \setminus \{v\}) \cup \{u_1\}$ is also an out-tree deletion set in D. Let $F = D - S$. Note that u_1 and u_2 are leaves in the out-tree F. Let F' be the out-tree obtained from F after deleting u_1 and adding the vertex v and the edge (u_2, v). Clearly F' is an out-tree and $F' = D - S'$. Therefore, S' is an out-tree deletion set of D.

Thus, it is enough to branch as follows.

- When u_1 belongs to the solution, the resulting instance is $(D - \{u_1\}, k - 1, r)$.
- When u_2 belongs to the solution, the resulting instance is $(D - \{u_2\}, k - 1, r)$.

The branching vector for this case is (1, 1).

In the upcoming lemma, we show that when all the reduction rules and branching rules have been considered exhaustively, all the vertices of the resulting instance, except r, have in-degree exactly 1 and in-degree of r is 0. In the later lemma we show that such an instance can be solved in polynomial time. Thus, after the exhaustive application of the reduction rules and the above mentioned branching rules, the algorithm uses the procedure described in Lemma 5 to solve the instance.

Lemma 4 *Let (D, k, r) be the instance where none of the above-mentioned reduction rules or branching rules are applicable. Then, for all $v \in V(D) \setminus \{r\}$, $d^-(v) = 1$ and $d^-(r) = 0$.*

Proof Since reduction rule 5 is no longer applicable, we have $d^-(r) = 0$. Also since reduction rules 4 is not applicable, $d^-(v) > 0$, for all $v \in V(D) \setminus \{r\}$. To show that for any vertex $v \in V(D) \setminus \{r\}$, $d^-(v) = 1$ we proceed as follows.

Let v be some vertex in $V(D) \setminus \{r\}$. For the sake of contradiction, let $d^-(v) > 1$. If $d^-(v) > 1$ and $r \in N^-(v)$, then branching rule 1 would be applicable. Thus, we can safely assume that $r \notin N^-(v)$.

We split the situation that $d^-(v) > 1$, into the following cases.

- If $d^-(v) \geq 3$, then branching rule 2 would be applicable.
- Otherwise $1 < d^-(v) \leq 2$, that is, $d^-(v) = 2$. We split this situation into the following 3 exhaustive cases.
  - If $d^+(v) \geq 1$, then branching rule 3 would be applicable.
  - Otherwise, $d^+(v) = 0$. In this situation, if the graph has some 2 vertices of in-degree 2 with a common in-neighbour then branching rule 4 is applicable, otherwise branching rule 5 would be applicable.

Since, none of the reduction rules or branching rules are applicable, we conclude that for all $v \in V(D) \setminus \{r\}$, $d^-(v) = 1$. □

Lemma 5 *Let (D, k, r) be an instance of* Restricted Pumpkin Vertex Deletion Set *such that for all $v \in V(D) \setminus \{r\}$, $d^-(v) = 1$, and $d^-(r) = 0$. Then the instance (D, k, s, t) can be solved in $\mathcal{O}(n^{\mathcal{O}(1)})$ time, where $n = |V(D)|$.*

Proof Since, for all $v \in V(D) \setminus \{r\}$, $d^-(v) = 1$ and $d^-(r) = 0$, and observe that (D, k, r) is a YES instance if and only if D is an out-tree with r as the root vertex and $k \geq 0$. □

In the next lemma, we formally prove the correctness of our algorithm.

Lemma 6 *The algorithm for* Restricted Out-Tree Vertex Deletion Set *described above is correct.*

Proof Let $I = (D, k, r)$ be an instance of ROTVDS. We prove the correctness of the algorithm by induction on $\mu = \mu(I) = k$. The base case occurs in one of the following cases.

- If $\mu \leq 0$, the algorithm correctly concludes whether (D, k, r) is a YES / NO instance from reduction rules 1 to 3.
- If $\mu \geq 0$ and D is an out-tree, the algorithm correctly concludes that (D, k, s, t) is a YES instance from reduction rule 3.
- If for all $v \in V(D) \setminus \{r\}$, $d^-(v) = 1$ and $d^-(r) = 0$, then from Lemma 5, (D, k, r) is a YES instance if and only if $k \geq 0$ and D is an out-tree with r as its root vertex.
- If r does not belong to $V(D)$, the algorithm correctly concludes that (D, k, r) is a NO instance from reduction rule 7.

By induction hypothesis we assume that for all $\mu \leq l$, the algorithm is correct. We will now prove that the algorithm is correct when $\mu = l + 1$. The algorithm

does one of the following. Either applies one of the reduction rules if applicable. By the safeness of the reduction rules we either correctly conclude that I is a YES/NO instance or produces an equivalent instance I' with $\mu(I') \leq \mu(I)$. If $\mu(I') < \mu(I)$, then by induction hypothesis and safeness of the reduction rules the algorithm correctly decides if I is a YES instance or not. Otherwise, $\mu(I') = \mu(I)$. If none of the reduction rules are applicable then the algorithm applies the first applicable Branching Rules. If some branching rule is applicable then, since μ decreases in each of the branch by at least one, by induction hypothesis the algorithm correctly concludes that I is a YES / NO instance. If none of the branching rules are applicable, then from Lemma 5, if $I = (D, k, s, t)$, then for all $v \in V(D) \setminus \{r\}$, $d^-(v) = 1$ and $d^-(r) = 0$. Thus, in this case the base case appears and hence, we conclude that the algorithm always outputs the correct answer. □

Next, we analyse the running of the algorithm presented.

Theorem 3 *The algorithm described above solves* RESTRICTED OUT-TREE VERTEX DELETION SET *in time* $\mathcal{O}^\star(2.562^k)$.

Proof The reduction rules 1 to 7 can be applied in time polynomial in the input size. Also, at each of the branch we spend a polynomial amount of time. At each of the recursive calls in a branch, the measure μ decreases at least by 1. When $\mu \leq 0$, then we are able to solve the remaining instance in polynomial time or correctly conclude that the corresponding branch cannot lead to a solution. At the start of the algorithm $\mu = k$.

The worst-case branching vector for the algorithm is $(1, 2, 2, 2, 2)$ (see Table 3). The recurrence for the worst case branching vector is:

$$T(\mu) \leq T(\mu - 1) + 4T(\mu - 2)$$

The running time corresponding to the above recurrence relation is $\mathcal{O}^\star(2.562^k)$. □

Theorem 3 together with the guessing step of r described in the beginning of the section gives us the following theorem.

Table 3 The branch vectors and their corresponding running times for ROTVDS

Branching Rule (BR)	Case/Sub-Case	Branch vector	c^μ
BR 1		(1, 1)	2^μ
BR 2		(1, 2, 2, 2)	2.303^μ
BR 3	a.i	(1, 1, 2)	2.41422^μ
	a.ii	(1, 2, 2, 2, 2)	2.562^μ
	b	(2, 1, 1)	2.41422^μ
BR 4	a	(1, 1, 2)	2.41422^μ
	b	(1, 2, 2, 2, 2)	2.562^μ
BR 5	a	(2, 1, 1)	2.41422^μ
	b	(1, 1)	2^μ

Theorem 4 OUT-TREE VERTEX DELETION SET *can be solved in time* $\mathcal{O}^\star(2.562^k)$.

5 FPT Algorithm for OUT-FOREST VERTEX DELETION SET

In this section, we give a branching algorithm for OUT-FOREST VERTEX DELETION SET (OFVDS). We start with a small description of our algorithm. Let (D, k) be an instance of OTVDS. Unlike our algorithm for PVDS or OTVDS, this algorithm does not require the initial guessing step. Also, the reduction rules and branching rules are designed so that when they are not applicable, the graph is empty. In other words, at any point of time, there always exists a vertex for which some reduction rule or branching rule is applicable. To achieve this trivial base case, the algorithm first eliminates (by branching over them) all vertices with in-degree at least 3, followed by the elimination of vertices with in-degree 0, then in-degree 1 vertices and finally vertices with in-degree 2. In the intermediate cases, it branches on vertices based on other factors like their out-degree and common-neighbours. These intermediate cases help the algorithm to branch in the main cases efficiently.

Next, we proceed to the details of the algorithm. The measure μ that is used to bound the depth of the search tree is the solution size, that is $\mu(D, k) = k$.

With a slight abuse of notation, in the following, during the application of any reduction/branching rule we will refer to (D, k, s, t) as the instance that is reduced with respect to the rules in higher preference order. The algorithm first applies the following reduction rules exhaustively.

Reduction Rule 1 *If* $k < 0$*, or* $k = 0$ *and* D *is not an out-forest, then return* (D, k) *is a* NO *instance.*

Reduction Rule 2 *If* $k \geq 0$ *and* D *is an out-forest, then return* (D, k) *is a* YES *instance.*

The safeness of reduction rule 1 and 2 is easy to see. Next we give some of the reduction rules which will eliminate certain irrelevant vertices in the digraph.

Reduction Rule 3 *For any* $v \in V(D)$*, if* $d^+(v) = 0$ *and* $d^-(v) \leq 1$ *then delete* v*. That is, the resulting instance is* $(D - \{v\}, k)$.

Reduction Rule 4 *Let* $(u, v) \in E(D)$ *such that* $d^-(v) = 1$*,* $d^+(u) = 1$ *and all the out-neighbours of* v *have in-degree exactly 1. Let* $\{w_1, \ldots, w_l\}$ *be the out-neighbours of* v*. Delete* v *from the graph and add the edges* $\{(u, w_i) \mid i \in [l]\}$ *to the graph. Let* D' *be the resulting graph. That is, the resulting instance is* (D', k).

Reduction Rule 5 *Let* $v \in V(D)$ *such that* $d^-(v) = 0$ *and all out-neighbours of* v *have in-degree exactly 1. Then delete* v *from the graph. That is, the resulting instance is* $(D - \{v\}, k)$.

Observe that each of the above-mentioned reduction rules can be applied in polynomial time. The safeness of reduction rules 3 to 5 is given by Lemma 7 to Lemma 9.

Lemma 7 *Reduction rule 3 is safe.*

Proof Let $v \in V(D)$ be a vertex such that $d^+(v) = 0$ and $d^-(v) \leq 1$. Let $D' = D - \{v\}$. We need to prove that (D, k) is a YES instance if and only if (D', k) is a YES instance. For the forward direction, let S be an out-forest deletion set in D of size at most k. Since $D' - (S \setminus \{v\})$ is a sub-graph of $D - S$, therefore, it follows that $S \setminus \{v\}$ is an out-forest deletion set in D'.

For the backward direction, let S be an out-forest deletion set in D' of size at most k. If $d^-(v) = 0$, then clearly, S is also an out-forest deletion set in D. Otherwise, let u be the unique in-neighbour of v in D. If $u \in S$, then $(D' - S) \cup \{v\}$ is an out-forest with v as an isolated vertex. Otherwise, $u \notin S$, then $(D' - S) \cup \{v\}$ is an out-forest where v is in the out-tree of $D' - S$ that contains u. Therefore, S is an out-forest deletion set in D. □

Lemma 8 *Reduction rule 4 is safe.*

Proof Let $(u, v) \in E(D)$ such that $d^-(v) = 1$ and all the out-neighbours of v have in-degree exactly 1. Let $\{w_1, \ldots, w_l\}$ be the out-neighbours of v. Also, D' be the graph obtained after deleting v from D and adding the edges $\{(u, w_i) \mid i \in [l]\}$ to D. We need to prove that (D, k) is a YES instance if and only if (D', k) is a YES instance. For the forward direction, let S be an out-forest deletion set in D of size at most k, $F = D - S$ and $W = \{w_1, \ldots, w_l\} \setminus S$. If $u \in S$, then $F - \{v\} = D' - S$. Thus, S is an out-forest of D'. If $u \notin S$ and $v \in S$, then let $S' = (S \setminus \{v\}) \cup \{u\}$. Then $F - \{u\} = D' - S'$. Hence, S' is an out-forest deletion set in D' of size at most k. Otherwise, $u, v \notin S$. Let F' be a digraph where $V(F') = V(F) \setminus \{v\}$ and $E(F') = (E(F) \setminus (u, v)) \cup \{(u, w_i) \mid w_i \in W\}$. Observe that F' is obtained after contracting the edge (u, v) and out-forests are closed under contraction. Therefore, F' is an out-forest and $F' = D' - S$. Hence S is an out-forest deletion set of D'.

For the backward direction, let S be an out-forest deletion set in D' of size at most k, $F = D' - S$ and $W = \{w_1, \ldots, w_l\} \setminus S$. Suppose $u \notin S$. Let F' be a digraph where $V(F') = V(F) \cup \{v\}$ and $E(F') = (E(F) \setminus \{(u, w_i) \mid w_i \in W\}) \cup \{(v, w_i) \mid w_i \in W\} \cup (u, v)$. Clearly F' is an out-forest and $F' = D - S$. Therefore, S is an out-forest deletion set in D. On the other hand, if $u \in S$ then, since each $w_i \in W$ has in-degree exactly 1 in D', each $w_i \in W$ is a root in F. Let F' be a digraph where $V(F') = V(F) \cup \{v\}$ and $E(F') = E(F) \cup \{(v, w_i) \mid w_i \in W\}$. Clearly F' is an out-forest and $F' = D - S$. Hence S is an out-forest deletion set of D. □

Lemma 9 *Reduction rule 5 is safe.*

Proof Let $v \in V(D)$ such that $d^-(v) = 0$, $w_1, \ldots, w_l$ are the out-neighbours of v such that for all $i \in [l]$, $d^-(w_i) = 1$. Let $D' = D - v$. We need to prove that (D, k) is a YES instance if and only if (D', k) is a YES instance. For the forward

direction, let S be an out-forest deletion set in D of size at most k. Then $S \setminus \{v\}$ is an out-forest deletion set of $D - v$. For the backward direction, let S be an out-forest deletion set in D' of size at most k, $F = D - S$ and $W = \{w_1, \ldots, w_l\} \setminus S$. Note that for all $i \in [l]$, w_i has in-degree 0 in D'. For all $w_i \in W$, let T_i be the out-tree of F containing w_i. Note that there is a unique T_i for each w_i and w_i is the root of T_i. Consider another out-tree T where $V(T) = \cup_{w_i \in W} V(T_i) \cup \{v\}$ and $E(T_i) = \cup_{w_i \in W} V(T_i) \cup \{(v, w_i) \mid w_i \in W\}$. Clearly $(F - \cup_{w_i \in W} V(T_i)) \cup T$ is an out-forest of $D - S$. Hence S is an out-forest deletion set of D. □

We now describe the branching rules used by the algorithm. Our algorithm applies 5 branching rules in order. Before giving the details of the branching rules, we first mention the invariants maintained by the algorithm after the exhaustive application of each of the branching rules.

Branching Rule 1 For all $v \in V(D)$, $d^-(v) \leq 2$. Also, if for any $v_1, v_2 \in V(D)$ such that $d^-(v_1) = d^+(v_2) = 2$, then v_1 and v_2 have no common in-neighbour.
Branching Rule 2 For all $v \in V(D)$, $d^+(v) \geq 1$.
Branching Rule 3 For all $v \in V(D)$, $d^-(v) \geq 1$.
Branching Rule 4 For all $v \in V(D)$, if $d^-(v) = 1$, then $d^+(v) \geq 2$.
Branching Rule 5 D is an empty graph.

Since after the exhaustive application of branching rules 1–5, the graph is empty, either reduction rule 1 or 2 would be applicable. In other words, when none of the branching rules are applicable, the input instance will be trivial to solve.

We now give the description of the branching rules. The cases under which each branching rule is applied is described pictorially in Fig. 3. From the description of the branching rules it is easy to see that each branching rule is exhaustive.

We first define a *hitting triplet*. For any $v_1, v_2, v_3 \in V(D)$, (v_1, v_2, v_3) is called a *hitting triplet* in D if there exists $i \in [3]$ such that v_i has in-degree at least 2 and v_j, v_ℓ, $j, \ell \in [3] \setminus \{i\}$, are some in-neighbours of v_i. Observe that if (v_1, v_2, v_3) is a hitting triplet in D, then any out-forest deletion set of D contains at least one of v_1, v_2 or v_3.

Branching Rule 1 If (v_1, v_2, x) and (v_3, v_4, x) are two distinct hitting triplets in D then branch as follows.

- When x belongs to the solution, the resulting instance is $(D - \{x\}, k - 1)$.
- When x does not belong to the solution, at least one of v_1, v_2 and one of v_3, v_4 belongs to the solution. When v_1, v_2, v_3, v_4 are distinct, the resulting instances in the respective branches are as follows.
  - When v_1, v_3 belongs to the solution, the resulting instance is $(D - \{v_1, v_3\}, k - 2)$.
  - When v_1, v_4 belongs to the solution, the resulting instance is $(D - \{v_1, v_4\}, k - 2)$.
  - When v_2, v_3 belongs to the solution, the resulting instance is $(D - \{v_2, v_3\}, k - 2)$.
  - When v_2, v_4 belongs to the solution, the resulting instance is $(D - \{v_2, v_4\}, k - 2)$.

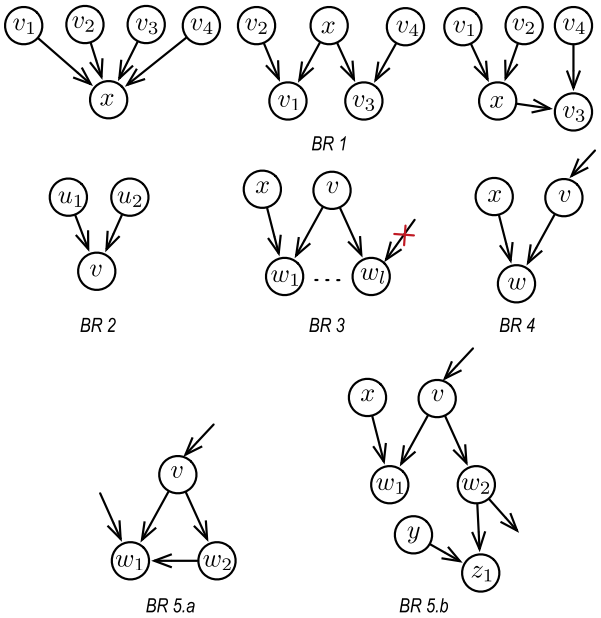

Fig. 3 The structures in the branching rules of OFVDS

We are now in the case when v_1, v_2, v_3, v_4 are not all distinct. Since (v_1, v_2, x) and (v_3, v_4, x) are hitting triplets, $v_1 \neq v_2$ and $v_3 \neq v_4$. Without loss of generality, let $v_1 = v_3$ (the other cases are symmetric). Since (v_1, v_2, x) and (v_3, v_4, x) are distinct hitting triplets and $v_1 = v_3$, $v_2 \neq v_4$. In this case, the resulting instances in the respective branches are as follows.

- When v_1 belongs to the solution, the resulting instance is $(D - \{v_1\}, k - 1)$.
- When v_1 does not belong to the solution (and x does not belong to the solution), both v_2 and v_4 belong to the solution. In this case, the resulting instance is $(D - \{v_2, v_4\}, k - 2)$.

The branching vector for this rule is either $(1, 2, 2, 2, 2)$ or $(1, 1, 2)$ depending on the case we are in.

Observe that after the application of Branching Rule 1, for each $v \in V(D)$, $d^-(v) \leq 2$. Suppose not. Let x, y, z be some three in-neighbours of v, then (v, x, y) and (v, x, z) form hitting triplets in D and hence Branching Rule 1 is applicable. Also, after the application of this branching rule, for any $v_1, v_2 \in V(D)$ such that $d^-(v_1), d^-(v_2) = 2$, v_1 and v_2 have no common in-neighbour.

Branching Rule 2 If $v \in V(D)$ is such that $d^+(v) = 0$, then since reduction rule 3 and branching rule 1 are not applicable, $d^-(v) = 2$.

Let u_1, u_2 be the in-neighbours of v. Note that one of v, u_1, u_2 must belong to the solution. Now observe that if there is an out-forest deletion set of D, say S, such that

$v \in S$ then $(S \setminus \{v\}) \cup \{u_i\}$, for any $i \in [2]$, is also an out-forest deletion set of D. Thus in this case it is enough to branch as follows.

- When u_1 belongs to the solution, the resulting instance is $(D - \{u_1\}, k - 1)$.
- When u_2 belongs to the solution, the resulting instance is $(D - \{u_2\}, k - 1)$.

The branching vector for this rule is (1, 1).

The next rule handles vertices that have in-degree 0. Note that if there is a vertex with in-degree 0 and its out-degree is 0 then reduction rule 3 would be applicable. If its in-degree is 0 and all its out-neighbours have in-degree exactly 1 then reduction rule 5 would be applicable. If two of its out-neighbours have in-degree 2 then Branching Rule 1 would be applicable. Thus, if none of the reduction rules and the above-mentioned branching rules are applicable, then any vertex which has in-degree 0 has at least one out-neighbour and exactly one of its out-neighbours has in-degree exactly two.

Branching Rule 3 If $v \in V(D)$ such that $d^-(v) = 0$, $\{w_1, \ldots, w_l\}$ are the out-neighbours of v ($l \geq 1$), for all $i \in \{2, \ldots, l\}$, $d^-(w_i) = 1$ and $d^-(w_1) = 2$, then let x be the other in-neighbour of w_1 (x may be one of w_i).

In this case, we claim that if S is an out-forest deletion set of D that contains v then $(S \setminus \{v\}) \cup \{w_1\}$ is also an out-forest deletion set of D. For the proof of this, observe that each $w_i \in \{w_1, \ldots, w_l\} \setminus S$ is a root in $D - S (= F$, say), except probably w_1. Also $d^-(v) = 0$. Therefore $(F \cup \{v\}) \setminus \{w_1\}$ is also an out-forest. Thus, S' is an out-forest deletion set of D. Therefore, it is enough to branch as follows.

- When x belongs to the solution, the resulting instance is $(D - \{x\}, k - 1)$.
- When w_1 belongs to the solution, the resulting instance is $(D - \{w_1\}, k - 1)$.

The branching vector in this branching rule is (1, 1). Again, note that hereafter, every vertex in the digraph has in-degree either 1 or 2. The upcoming branching rules 4 and 5, deal with vertices that have in-degree 1.

If $v \in V(D)$ such that $d^-(v) = 1$ and $d^+(v) = 1$, then if the unique out-neighbour, say w, of v has out-degree 0, then either reduction rule 3 or branching rule 2 would be applicable. If $d^-(w) = 1$ then, if $d^+(w) = 1$ then reduction rule 4 would be applicable, and if $d^+(w) \geq 2$ then reduction rule 4 would be applicable. Thus the case when in-degree of w is 1, is handled.

Branching rule 4 deals with the situation when there is a vertex v with in-degree 1 and out-degree 1 and its unique out-neighbour has in-degree 2. Thus, when none of the reduction rules and branching rules 1 to 4 are applicable, for any vertex v, either $d^-(v) = 2$ and $d^+(v) \geq 1$, or if $d^-(v) = 1$ then $d^+(v) \geq 2$.

Branching Rule 4 If $v \in V(D)$ such that $d^-(v) = 1$ and $d^+(v) = 1$, then if the unique out-neighbour, say w, has in-degree 2, then let x be the other in-neighbour of w.

We first make the following claim. If S is an out-forest deletion set in D such that $v \in S$, then $(S \setminus \{v\}) \cup \{w\}$ is also an out-forest deletion set in D. This is true because if $F = D - S$ is an out-forest then so is $(F - \{w\}) \cup \{v\}$. Thus, in this case, the algorithm branches on x as follows.

- When x belongs to the solution, the resulting instance is $(D - \{x\}, k - 1)$.
- When w belongs to the solution, the resulting instance is $(D - \{w\}, k - 1)$.

The branching vector for this rule is (1, 1). Recall that, when none of the reduction rules and branching rules 1 to 4 are applicable, for any vertex v, either $d^-(v) = 2$ and $d^+(v) \geq 1$, or if $d^-(v) = 1$ then $d^+(v) \geq 2$.

Let v be a vertex with in-degree 1 and out-degree at least 2. Let $\{w_1, \ldots, w_l\}$ be the out-neighbours of v. Then if all the out-neighbours of v have in-degree exactly 1, reduction rule 4 would be applicable. Also, if two of the in-neighbours have in-degree 2, then branching rule 1 would be applicable. Thus, without loss of generality assume that $d^-(w_1) = 2$ and for all $i \in \{2, \ldots, l\} d^-(w_i) = 1$. Let x be the other in-neighbour of w_1. The case when $x = w_2$ will be handled in case 5.*a*. Otherwise, x, v, w_1, w_2 are all distinct. Now observe that $d^+(w_2) \geq 2$, otherwise branching rule 4 was applicable on w_2. Also, all the out-neighbours of w_2 except 1 have in-degree 1, otherwise either reduction rule 4 was applicable or branching rule 1 was applicable on w_2. Let z_1 be the out-neighbour of w_2 of in-degree 2. Let y be the other in-neighbour of z_1. Observe that (v, w_1, x) and (w_2, z_1, y) are hitting triplets in D. If (v, w_1, x) and (w_2, z_1, y) are not distinct hitting triplets, then $x = w_2$ and we are in Case 5.*a*. Otherwise, since Branching Rule 1 is not applicable, all of v, w_1, x, w_2, z_1, y are distinct. This case is handled in Case 5.*b*.

After presenting the Cases 5.*a* and 5.*b*, we will discuss why after the exhaustive application of all the described reduction and branching rules, there is no vertex v such that either $d^-(v) = 2$ and $d^+(v) \geq 1$. This will show that after the exhaustive application of all the described reduction and branching rules, the graph is empty.

Branching Rule 5 Let $v \in V(D)$ be such that $d^-(v) = 1$ and $d^+(v) = l \geq 2$. Let $\{w_1, \ldots, w_l\}$ be the set of out neighbours of v and let $d^-(w_1) = 2$ and $d^-(w_i) = 1$ for all $i \in \{2, \ldots, l\}$. Let x be the other in-neighbour of w_1. Let $d^+(w_2) = p \geq 2$. Let $\{z_1, \ldots, z_p\}$ be the out-neighbours of w_2 and let $d^-(z_1) = 2$ and for all $i \in \{2, \ldots, p\}$, $d^-(z_i) = 1$. Let y be the other in-neighbour of z_1.

In this case, the algorithm considers the following sub-cases.

Case 5.a If $x = w_2$, then observe that $(w_2, w_1) \in E(D)$. In this case, the algorithm proceeds as follows.

We first claim the following. If S is an out-forest deletion set of D such that $w_2 \in S$, then $S' = (S \setminus \{w_2\}) \cup \{v\}$ is an out-forest deletion set of D. To prove this, let $F = D - S$. Note that each out-neighbour of w_2 in F except possibly w_1, is a root of some out-tree in F. This is because for all the out-neighbours of w_2, except w_1, w_2 is their unique in-neighbour. In fact, in $D - (S \cup \{v\})$, every neighbour of w_2 in $D - (S \cup \{v\})$ is an out-neighbour of w_2 in D and is a root of some out-tree in $D - (S \cup \{v\})$. Therefore, $D \setminus ((S \cup \{v\}) \setminus \{w_2\})$ is an out-forest. In other words, $(S \cup \{v\}) \setminus \{w_2\}$ is a solution of size at most k in D. This finishes the proof of the claim.

Thus, the algorithm branches as follows.

- When w_1 belongs to the solution, the resulting instance is $(D - \{w_1\}, k - 1)$.

- When w_1 does not belong to the solution and (D, k) is a YES instance, then there is a solution that contains v. Therefore the resulting instance is $(D - \{v\}, k - 1)$.

The branching vector in this case is (1, 1).

Case 5.b Let v, w_1, x, w_2, z_1, y be all distinct vertices. In this case, the algorithm branches as follows.

- When x belongs to the solution, the resulting instance is $(D - \{x\}, k - 1)$.
- When x does not belong to the solution, the algorithm branches on w_1 as follows.
  - When w_1 belongs to the solution, the resulting instance is $(D–\{w_1\}, k–1)$.
  - When w_1 does not belong to the solution, then v must belong to the solution. Therefore delete v. Note that after deleting v from the graph w_2 becomes an in-degree 0 vertex and all its out-neighbours have in-degree exactly 1 except for z_1. Therefore as argued before, if there is a solution that contains w_2 then there is another solution that avoids w_2 and contains z_1. Thus, the algorithm branches as follows.

    When y belongs to the solution, the resulting instance is $(D - \{v, y\}, k - 2)$.
    When y does not belong to the solution, delete z_1 from the graph, that is, the resulting instance is $(D - \{v, z_1\}, k - 2)$.

The branching vector in this case is (1, 1, 2, 2).

Let $v \in V(D)$ be such that $d^-(v) = 2$ and $d^+(v) \geq 1$. Let w be some out-neighbour of v. If in-degree of w is 1, then reduction rule 3, branching rule 4 or branching rule 5 would have applied on w. Otherwise, in-degree of w is 2. Let x be the other in-neighbour of w and u_1, u_2 be the in-neighbours of v. In this case, observe that (x, w, v) and (v, u_1, u_2) are hitting triplets in D and hence Branching Rule 1 is applicable.

Observe that when none of the above-mentioned reduction rules and cases are applicable, the graph is empty, that is there are no-vertices in the graph.

The following theorem proves the correctness of the algorithm presented.

Theorem 5 *The presented algorithm for* OUT-FOREST VERTEX DELETION SET *is correct.*

Proof Let $I = (D, k)$ be an instance of OUT-FOREST VERTEX DELETION SET. We prove the correctness of the algorithm by induction on $\mu = \mu(I) = k$. The base case occurs in one of the following cases.

- $\mu <= 0$ we correctly conclude whether (D, k) is a yes instance or not by reduction rule 1 or reduction rule 2.
- $\mu > 0$ and D is an out-forest then we correctly conclude that (D, k) is a YES instance by reduction rule 2.

By induction hypothesis we assume that for all $\mu \leq l$, the algorithm is correct. We will now prove that the algorithm is correct when $\mu = l + 1$. The algorithm

Table 4 The branch vectors and their corresponding running times for OFVDS

Branching Rule (BR)	Case/Sub-Case	Branch vector	c^{μ}
BR 1		(1, 2, 2, 2)	2.303^{μ}
		(1, 1, 2)	2.41422^{μ}
BR 2		(1, 1)	2^{μ}
BR 3		(1, 1)	2^{μ}
BR 4		(1, 1)	2^{μ}
BR 5	a	(1, 1)	2^{μ}
	b	(1, 1, 2, 2)	2.7321^{μ}

does the following. It applies one of the reduction rules if applicable. By the safeness of the reduction rules the algorithm either correctly concludes that I is a YES/ NO instance or produce an equivalent instance I' with $\mu(I') \leq \mu(I)$. If $\mu(I') < \mu(I)$, then by induction hypothesis and safeness of the reduction rules the algorithm correctly decides if I is a yes instance or not. Otherwise, $\mu(I') = \mu(I)$. If none of the reduction rules are applicable then the algorithm applies the first applicable Branching Rules. Branching Rules are exhaustive and covers all possible cases. Furthermore, μ decreases in each of the branch by at least one. Therefore, by the induction hypothesis, the algorithm correctly decides whether I is a yes instance or not. □

Theorem 6 OUT-FOREST VERTEX DELETION SET *can be solved in* $\mathcal{O}^*((1+\sqrt{3})^k)$.

Proof The reduction rules 1 to 5 can be applied in time polynomial in the input size. Also, at each of the branch we spend a polynomial amount of time. At each of the recursive calls in a branch, the measure μ decreases by at least 1. When $\mu \leq 0$, then reduction rule 1 or 2 is applicable and hence the algorithm correctly returns the answer and terminate. At the start of the algorithm $\mu = k$.

The worst-case branching vector for the algorithm is (1, 1, 2, 2) (see Table 4). The recurrence for the worst case branching vector is:

$$T(\mu) \leq 2T(\mu - 1) + 2T(\mu - 2)$$

The running time corresponding to the above recurrence relation is $\mathcal{O}^{\star}((1 + \sqrt{3})^k)$. □

Acknowledgements We thank the anonymous reviewers for their useful comments on improving the presentation of this article.

References

1. Abu-Khzam, F.N.: A kernelization algorithm for d-Hitting Set. JCSS **76**(7), 524–531 (2010)
2. Bafna, V., Berman, P., Fujito, T.: A 2-approximation algorithm for the undirected feedback vertex set problem. SIDMA **12**(3), 289–297 (1999)
3. Bang-Jensen, J., Maddaloni, A., Saurabh, S.: Algorithms and kernels for feedback set problems in generalizations of tournaments. Algorithmica **76**(2), 320–343 (2016)

4. Bar-Yehuda, R., Geiger, D., Naor, J., Roth, R.M.: Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and Bayesian inference. SICOMP **27**(4), 942–959 (1998)
5. Becker, A., Geiger, D.: Optimization of pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. Artif. Intell **83**(1), 167–188 (1996)
6. Cao, Y., Chen, J., Liu, Y.: On feedback vertex set new measure and new structures. In: SWAT, pp. 93–104 (2010)
7. Chekuri, C., Madan, V.: Constant factor approximation for subset feedback problems via a new LP relaxation. In: SODA, pp. 808–820 (2016)
8. Chen, J., Fomin, F.V., Liu, Y., Lu, S., Villanger, Y.: Improved algorithms for feedback vertex set problems. JCSS **74**(7), 1188–1198 (2008)
9. Chen, J., Liu, Y., Lu, S., O'Sullivan, B., Razgon, I.: A fixed-parameter algorithm for the directed feedback vertex set problem. Journal of the ACM (JACM) **55**(5), 21 (2008)
10. Chitnis, R.H., Cygan, M., Hajiaghayi, M.T., Marx, D.: Directed subset feedback vertex set is fixed-parameter tractable. TALG **11**(4), 28 (2015)
11. Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer, Berlin (2015)
12. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., Rooij, J.M.M., Wojtaszczyk, J.O.: Solving connectivity problems parameterized by treewidth in single exponential time. In: FOCS, pp. 150–159 (2011)
13. Cygan, M., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: Subset feedback vertex set is fixed-parameter tractable. SIDMA **27**(1), 290–309 (2013)
14. Diestel, R.: Graph Theory, 4th edn. Springer, Berlin (2012)
15. Dom, M., Guo, J., Hüffner, F., Niedermeier, R., Truß, A.: Fixed-parameter tractability results for feedback set problems in tournaments. JDA **8**(1), 76–86 (2010)
16. Downey, R.G., Fellows, M.R.: Parameterized complexity. Springer, Berlin (1997)
17. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Springer, Berlin (2013)
18. Erdős, P., Pósa, L.: On independent circuits contained in a graph. Canad. J. Math. **17**, 347–352 (1965)
19. Even, G., Naor, J., Schieber, B., Sudan, M.: Approximating minimum feedback sets and multicuts in directed graphs. Algorithmica **20**(2), 151–174 (1998)
20. Fomin, F.V., Gaspers, S., Lokshtanov, D., Saurabh, S.: Exact algorithms via monotone local search. In: STOC, pp. 764–775 (2016)
21. Guruswami, V., Lee, E.: Inapproximability of H-transversal/packing. In: APPROX/RANDOM, pp. 284–304 (2015)
22. Kakimura, N., Kawarabayashi, K., Kobayashi, Y.: Erdös-Pósa property and its algorithmic applications: parity constraints, subset feedback set, and subset packing. In: SODA, pp. 1726–1736 (2012)
23. Kakimura, N., Kawarabayashi, K., Marx, D.: Packing cycles through prescribed vertices. J. Comb. Theory, Ser. B **101**(5), 378–381 (2011)
24. Kawarabayashi, K., Kobayashi, Y.: Fixed-parameter tractability for the subset feedback set problem and the *S*-cycle packing problem. J. Comb. Theory, Ser. B **102**(4), 1020–1034 (2012)
25. Kawarabayashi, K., Král, D., Krcál, M., Kreutzer, S.: Packing directed cycles through a specified vertex set. In: SODA, pp. 365–377 (2013)
26. Kociumaka, T., Pilipczuk, M.: Faster deterministic feedback vertex set. IPL **114**(10), 556–560 (2014)
27. Mehlhorn, K.: Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness, EATCS Monographs on Theoretical Computer Science, vol. 2. Springer, Berlin (1984)
28. Mnich, M., van Leeuwen, E.J.: Polynomial kernels for deletion to classes of acyclic digraphs. In: STACS, pp. 1–13 (2016)
29. Pontecorvi, M., Wollan, P.: Disjoint cycles intersecting a set of vertices. J. Comb. Theory, Ser. B **102**(5), 1134–1141 (2012)
30. Raman, V., Saurabh, S., Subramanian, C.R.: Faster fixed parameter tractable algorithms for finding feedback vertex sets. TALG **2**(3), 403–415 (2006)
31. Raman, V., Saurabh, S., Suchý, O.: An FPT algorithm for tree deletion set. J. Graph Algorithms Appl. **17**(6), 615–628 (2013)
32. Reed, B.A., Robertson, N., Seymour, P.D., Thomas, R.: Packing directed circuits. Combinatorica **16**(4), 535–554 (1996)
33. Seymour, P.D.: Packing directed circuits fractionally. Combinatorica **15**(2), 281–288 (1995)
34. Seymour, P.D.: Packing circuits in eulerian digraphs. Combinatorica **16**(2), 223–231 (1996)
35. Wahlström, M.: Half-integrality, LP-branching and FPT Algorithms. In: SODA, pp. 1762–1781 (2014)