Theory Comput Syst (2016) 58:357–376
DOI 10.1007/s00224-015-9631-7

A Fast Branching Algorithm for Cluster Vertex Deletion

**Anudhyan Boral[1] · Marek Cygan[2] ·
Tomasz Kociumaka[2] · Marcin Pilipczuk[2]**

Published online: 25 April 2015

Abstract In the family of clustering problems we are given a set of objects (vertices of the graph), together with some observed pairwise similarities (edges). The goal is to identify clusters of similar objects by slightly modifying the graph to obtain a cluster graph (disjoint union of cliques). Hüffner et al. (Theory Comput. Syst. **47**(1), 196–217, 2010) initiated the parameterized study of CLUSTER VERTEX DELETION, where the allowed modification is vertex deletion, and presented an elegant $\mathcal{O}\left(\min(2^k k^6 \log k + n^3, 2^k km\sqrt{n}\log n)\right)$-time fixed-parameter algorithm, parameterized by the solution size. In the last 5 years, this algorithm remained the fastest known algorithm for CLUSTER VERTEX DELETION and, thanks to its simplicity, became one of the textbook examples of an application of the iterative compression principle. In our work we break the 2^k-barrier for CLUSTER VERTEX DELETION and present an $\mathcal{O}(1.9102^k(n + m))$-time branching algorithm. We achieve this improvement by a number of structural observations which we incorporate into the algorithm's branching steps.

A preliminary version of this work has been presented at CSR 2014.

✉ Marcin Pilipczuk
malcin@mimuw.edu.pl

Anudhyan Boral
anudhyan@cmi.ac.in

Marek Cygan
cygan@mimuw.edu.pl

Tomasz Kociumaka
kociumaka@mimuw.edu.pl

1 Chennai Mathematical Institute, Chennai, India

2 Institute of Informatics, University of Warsaw, Warsaw, Poland

Keywords Fixed-parameter algorithms · Cluster vertex deletion · Branching

1 Introduction

The problem to cluster objects based on their pairwise similarities has arisen from applications both in computational biology [5] and machine learning [4]. In the language of graph theory, as an input we are given a graph where vertices correspond to objects, and two objects are connected by an edge if they are observed to be similar. The goal is to transform the graph into a cluster graph (a disjoint union of cliques) using a minimum number of modifications.

The set of allowed modifications depends on the particular problem variant and the application under consideration. Probably the most studied variant is the CLUSTER EDITING problem, known also as CORRELATION CLUSTERING, where we seek for a minimal number of edge edits to obtain a cluster graph. The CLUSTER EDITING problem is APX-hard [9], and admints constant-factor approximations [3, 9]. Also, no subexponential time algorithm exists [13, 18] unless exponential time hypothesis fails. From the parameterized perspective, currently the fastest algorithm runs in $\mathcal{O}(1.62^k + n + m)$ time [6], and both a small [10] and efficient [19] polynomial kernels. For more references on the CLUSTER EDITING problem, see [7].

The main principle of parameterized complexity is that we seek algorithms that are efficient if the considered parameter is small. However, the distance measure in CLUSTER EDITING, the number of edge edits, may be quite large in practical instances, and, in the light of recent lower bounds refuting the existence of subexponential FPT algorithms for CLUSTER EDITING [13, 17], it seems reasonable to look for other distance measures (see, e.g., Komusiewicz's PhD thesis [17]) and/or different problem formulations.

In 2003, Gramm et al. [14] initiated the parameterized study of the CLUSTER VERTEX DELETION problem (CLUSTERVD for short). Here, the allowed modifications are vertex deletions.

CLUSTER VERTEX DELETION (CLUSTERVD) **Parameter:** k
Input: An undirected graph G and an integer k.
Question: Does there exist a set S of at most k vertices of G such that $G \setminus S$ is a cluster graph, i.e., a disjoint union of cliques?

In terms of motivation, we want to refute as few objects as possible to make the set of observations completely consistent. Since a vertex deletion removes as well all its incident edges, we may expect that this new editing measure may be significantly smaller in practical applications than the edge-editing distance.

It can be shown that a graph is a cluster graph if and only if it contains no induced P_3s (paths on 3 vertices). As CLUSTERVD can be equivalently stated as the problem of hitting, with minimum number of vertices, all induced P_3s in the input graph, CLUSTERVD can be solved in $\mathcal{O}(3^k(n + m))$ time by a straightforward branching algorithm [8], where n and m denote the number of vertices and edges of G, respectively. The dependency on k has been improved by considering more elaborate case distinction in the branching algorithm, first to $\mathcal{O}(2.26^k + nm)$ using problem-specific rules by Gramm et al. [14], and later to $\mathcal{O}(2.179^k + nm)$ via a general algorithm by Fernau for the 3-HITTING SET problem [11]. The $\mathcal{O}(2.0755^k + mn)$ running time can be achieved if instead of the latter one applies an algorithm for 3-HITTING SET described in the PhD thesis of Wahlström [21]. Hüffner et al. [15] provided an elegant $\mathcal{O}(2^k k^9 + nm)$-time algorithm for CLUSTERVD, using the iterative compression principle [20] and a reduction to the weighted maximum matching problem. This algorithm quickly became one of the textbook examples of an application of the iterative compression technique.

In our work we pick up this line of research and obtain the fastest algorithm for (unweighted) CLUSTERVD.

Theorem 1 CLUSTER VERTEX DELETION *can be solved in* $\mathcal{O}(1.9102^k(n + m))$ *time and polynomial space on an input* (G, k) *with* $|V(G)| = n$ *and* $|E(G)| = m$.

The source of the exponential 2^k factor in the time complexity of the algorithm of [15] comes from enumeration of all possible intersections of the size-k solution we are looking for with a solution of size $k + 1$ obtained from the previous phase of the iterative compression process. As the next step in each subcase is a reduction to the weighted maximum matching problem (with a definitely nontrivial polynomial-time algorithm), it seems hard to break the 2^k-barrier using the approach of [15]. Hence, in the proof of Theorem 1 we go back to the bounded search tree approach. However, to achieve the promised time bound, and at the same time avoiding very extensive case analysis, we do not follow the general 3-HITTING SET approach. Instead, our methodology is to carefully investigate the structure of the graph and an optimum solution around a vertex already guessed to be not included in the solution. We note that a somehow similar approach has been used in [15] to cope with a variant of CLUSTERVD where we restrict the number of clusters in the resulting graph.

More precisely, the main observation in the proof of Theorem 1 is that, if for some vertex v we know that there exists a minimum solution S not containing v, then in the neighbourhood of v the CLUSTERVD problem reduces to VERTEX COVER. Let us define N_1 and N_2 to be the vertices at distance 1 and 2 from v, respectively, and define the auxiliary graph H_v to be a graph on $N_1 \cup N_2$ having an edge for each edge of G between N_1 and N_2 and for each non-edge in $G[N_1]$. In other words, two vertices are connected by an edge in H_v if, together with v, they form a P_3 in G. We observe that a minimum solution S not containing v needs to contain a vertex cover of H_v. Moreover, one can show that we may greedily choose a vertex cover with inclusion-wise maximal intersection with N_2, as deleting vertices from N_2 helps us resolve the remaining part of the graph.

Branching to find the 'correct' vertex cover of H_v is very efficient, with worst-case $(1, 2)$ (i.e., golden-ratio) branching vector. However, we do not have the vertex v beforehand, and branching to obtain such a vertex is costly. Our approach is to get as much gain as possible from the vertex cover-style branching on the graph H_v, to be able to balance the loss from some inefficient branches used to obtain the initial vertex v. Consequently, we employ involved analysis of properties and branching algorithms for the auxiliary graph H_v.

Note that the algorithm of Theorem 1 can be pipelined with the kernelization algorithm of 3-HITTING SET [1], yielding the following corollary.

Corollary 2 CLUSTER VERTEX DELETION *can be solved in* $\mathcal{O}(1.9102^k k^4 + nm)$ *time and polynomial space on an input* (G, k) *with* $|V(G)| = n$ *and* $|E(G)| = m$.

However, due to the $\mathcal{O}(nm)$ summand in the complexity of Corollary 2, for a wide range of input instances the running time bound of Theorem 1 is better than the one of Corollary 2. In fact, the advantage of our branching approach is that the obtained dependency on the graph size in the running time is linear, whereas with the approach of [15], one needs to spend at least quadratic time either on computing weighted maximum matching or on kernelizing the instance.

We also analyse the co-cluster setting, where one aims at obtaining a co-cluster graph instead of a cluster one, and show that the linear dependency on the size of the input can be maintained also in this case. In other words, CLUSTERVD problem can be also solved in $\mathcal{O}(1.9102^k(n+\bar{m}))$ if input graph is represented by its complement, which has $\bar{m} = \binom{n}{2} - m$ edges. Such representation leads to a more time- and space-efficient solution for very dense input graphs. In particular, this would be a better choice if one expects most of the vertices of the resulting cluster graph to form a single clique.

The paper is organised as follows. We give some preliminary definitions and notation in Section 2. In Section 3 we analyse the structural properties of the auxiliary graph H_v. Then, in Section 4 we prove Theorem 1, with the main tool being a subroutine branching algorithm finding all relevant vertex covers of H_v. In Section 5 we analyse the co-cluster setting. Section 6 concludes the paper. The Appendix contains a Python script for computing the worst-case complexity of the algorithm.

2 Preliminaries

We use standard graph notation. All our graphs are undirected and simple. For a graph G, by $V(G)$ and $E(G)$ we denote its vertex- and edge-set, respectively. For $v \in V(G)$, the set $N_G(v) = \{u \mid uv \in E(G)\}$ is the neighbourhood of v in G and $N_G[v] = N_G(v) \cup \{v\}$ is the closed neighbourhood. We extend these notions to sets of vertices $X \subseteq V(G)$ by $N_G[X] = \bigcup_{v \in X} N_G[v]$ and $N_G(X) = N_G[X] \setminus X$. We omit the subscript if it is clear from the context. For a set $X \subseteq V(G)$ we also define $G[X]$ to be the subgraph induced by X and $G \setminus X$ is a shorthand for $G[V(G) \setminus X]$. An even cycle is a cycle with an even number of edges, and an even path is a path with an even number of edges. A set $X \subseteq V(G)$ is called a *vertex cover* of G if $G \setminus X$ is edgeless. By $\text{VC}(G)$ we denote the size of the minimum vertex cover of G.

In all further sections, we assume we are given an instance (G, k) of CLUSTER VERTEX DELETION, where $G = (V, E)$. That is, we use V and E to denote the vertex- and edge-set of the input instance G.

A P_3 is an ordered set of 3 vertices (u, v, w) such that $uv, vw \in E$ and $uw \notin E$. A graph is a cluster graph if and only if it does not contain any P_3; hence, in

CLUSTERVD we seek for a set of at most k vertices that hits all P_3s. We note also the following.

Lemma 3 *Let G be a connected graph which is not a clique. Then, for every $v \in V(G)$, there is a P_3 containing v.*

Proof Consider $N(v)$. If there exist vertices $u, w \in N(v)$ such that $uw \notin E(G)$ then we have a $P_3(u, v, w)$. Otherwise, since $N[v]$ induces a clique, we must have $w \in N(N[v])$ such that $uw \in E(G)$ for some $u \in N(v)$. Thus we have a P_3, (v, u, w) involving v. □

If at some point a vertex v is fixed in the graph G, we define sets $N_1 = N_1(v)$ and $N_2 = N_2(v)$ as follows: $N_1 = N_G(v)$ and $N_2 = N_G(N_G[v])$. That is, N_1 and N_2 are sets of vertices at distance 1 and 2 from v, respectively. For a fixed $v \in V$, we define an auxiliary graph H_v with $V(H_v) = N_1 \cup N_2$ and

$$E(H_v) = \{uw \mid u, w \in N_1, uw \notin E\} \cup \{uw \mid u \in N_1, w \in N_2, uw \in E\}.$$

Thus, H_v consists of the vertices in N_1 and N_2 along with non-edges among vertices of N_1 and edges between N_1 and N_2. Note that N_2 is an independent set in H_v. Observe the following.

Lemma 4 *For $u, w \in N_1 \cup N_2$, we have $uw \in E(H_v)$ if and only if u, w and v form a P_3 in G.*

Proof For every $uw \in E(H_v)$ with $u, w \in N_1$, (u, v, w) is a P_3 in G. For $uw \in E(H_v)$ with $u \in N_1$ and $w \in N_2$, (v, u, w) forms a P_3 in G. In the other direction, for any P_3 in G of the form (u, v, w) we have $u, w \in N_1$ and $uw \notin E$, thus $uw \in E(H_v)$. Finally, for any P_3 in G of the form (v, u, w) we have $u \in N_1$, $w \in N_2$ and $uw \in E$, hence $uw \in E(H_v)$. □

We call a subset $S \subseteq V$ a *solution* when $G \setminus S$ is a cluster graph, that is, a collection of disjoint cliques. A solution with minimal cardinality is called a *minimum solution*.

Our algorithm is a typical branching algorithm, that is, it consists of a number of *branching steps*. In a step $(A_1, A_2, \ldots, A_r)$, $A_1, A_2, \ldots, A_r \subseteq V$, we independently consider r subcases. In the i-th subcase we look for a minimum solution S containing A_i: we delete A_i from the graph and decrease the parameter k by $|A_i|$. If k becomes negative, we terminate the current branch and return a negative answer from the current subcase.

The *branching vector* for a step $(A_1, A_2, \ldots, A_r)$ is $(|A_1|, |A_2|, \ldots, |A_r|)$. It is well-known (see e.g. [12]) that the number of final subcases of a branching algorithm is $\mathcal{O}(c^k)$, where c is the largest positive root of the equation $1 = \sum_{i=1}^{r} x^{-|A_i|}$ among all branching steps $(A_1, A_2, \ldots, A_r)$ in the algorithm.

At some places, the algorithm makes a greedy (but optimal) choice of including a set $A \subseteq V$ into the constructed solution. We formally treat it as length-one branching step (A) with branching vector $(|A|)$.

3 The Auxiliary Graph H_v

In this section we investigate properties of the auxiliary graph H_v. Hence, we assume that a CLUSTERVD input (G, k) is given with $G = (V, E)$, and a vertex $v \in V$ is fixed.

3.1 Basic Properties

First, note that an immediate consequence of Lemma 4 is the following.

Corollary 5 *Let S be a solution such that $v \notin S$. Then S contains a vertex cover of H_v.*

In the other direction, the following holds.

Lemma 6 *Let X be a vertex cover of H_v. Then, in $G \setminus X$, the connected component of v is a clique.*

Proof Suppose the connected component of v in $G \setminus X$ is not a clique. Then by Lemma 3, there is a P_3 involving v. Such a P_3 is also present in G. However, by Lemma 4, as X is a vertex cover of H_v, X intersects such a P_3, a contradiction. □

Lemma 7 *Let S be a solution such that $v \notin S$. Denote by X the set $S \cap V(H_v)$. Let Y be a vertex cover of H_v. Suppose that $X \cap N_2 \subseteq Y \cap N_2$. Then $T := (S \setminus X) \cup Y$ is also a solution.*

Proof Since Y (and hence, $T \cap V(H_v)$) is a vertex cover of H_v and $v \notin T$, we know by Lemma 6 that the connected component of v in $G \setminus T$ is a clique. If T is not a solution, then there must be a P_3 contained in $Z \setminus T$, where $Z = V \setminus (\{v\} \cup N_1)$. But since $S \cap Z \subseteq T \cap Z$, $G \setminus S$ would also contain such a P_3. □

Lemma 7 motivates the following definition. For vertex covers of H_v, X and Y, we say that Y *dominates* X if $|Y| \le |X|$, $Y \cap N_2 \supseteq X \cap N_2$ and at least one of these inequalities is sharp. Two vertex covers X and Y are said to be *equivalent* if $X \cap N_2 = Y \cap N_2$ and $|X \cap N_1| = |Y \cap N_1|$. We note that the first aforementioned relation is transitive and strongly anti-symmetric, whereas the second is an equivalence relation.

As a corollary of Lemma 7, we have:

Corollary 8 *Let S be a solution such that $v \notin S$. Suppose Y is a vertex cover of H_v which either dominates or is equivalent to the vertex cover $X = S \cap V(H_v)$. Then $T := (S \setminus X) \cup Y$ is also a solution with $|T| \le |S|$.*

3.2 Special Cases of H_v

We now carefully study the cases where H_v has small vertex cover or has a special structure, and discover some possible greedy decisions that can be made.

Lemma 9 *Suppose X is a vertex cover of H_v. Then there is a minimum solution S such that either $v \notin S$ or $|X \setminus S| \geq 2$.*

Proof Suppose S is a minimum solution such that $v \in S$ and $|X \setminus S| \leq 1$. We are going to convert S to another minimum solution T that does not contain v.

Consider $T := (S \setminus \{v\}) \cup X$. Clearly, $|T| \leq |S|$. Since T contains X, a vertex cover, by Lemma 6, the connected component of v in $G \setminus T$ is a clique. Thus, there is no P_3 containing v. Since any P_3 in $G \setminus T$ which does not include v must also be contained in $G \setminus S$, contradicting the fact that S is a solution, we obtain that T is also a solution. Hence, T is a minimum solution. □

Corollary 10 *If $\mathrm{VC}(H_v) = 1$, then there is a minimum solution S such that $v \notin S$.*

Lemma 11 *Let C be the connected component of G containing v, and assume that neither C nor $C \setminus \{v\}$ is a cluster graph. If $X = \{w_1, w_2\}$ is a minimum vertex cover of H_v, then there exists a connected component $\widehat{C}$ of $G \setminus \{v\}$ that is not a clique and $\widehat{C} \cap \{w_1, w_2\} \neq \emptyset$.*

Proof Consider a component $\widehat{C}$ of $C \setminus \{v\}$ which is not a clique. Since v must be adjacent to each connected component of $C \setminus \{v\}$, $\widehat{C} \cap N_1$ must be non-empty. For any $w \in \widehat{C} \cap N_1$, we have that $w_1, w_2 \neq w$ and $ww_1, ww_2 \notin E(G)$, since otherwise the result follows. If $uw \in E(G)$ with $u \in N_2$, then, as $\{w_1, w_2\}$ is a vertex cover of H_v we must have $u = w_1$ or $u = w_2$. We would then have w_1 or w_2 contained in a non-clique $\widehat{C}$, contradicting our assumption. Hence $uw \in E(G) \Rightarrow u \in N_1$. Thus $\widehat{C} \subseteq N_1$. As w_1 and w_2 are not contained in $\widehat{C}$ and they cover all edges in H_v, $\widehat{C}$ must be an independent set in H_v. In $G \setminus \{v\}$, therefore, $\widehat{C}$ must be a clique, a contradiction. □

We now investigate the case when H_v has a very specific structure. The motivation for this analysis will become clear in Section 4.3.

A *seagull* is a connected component of H_v that is isomorphic to a P_3 with middle vertex in N_1 and endpoints in N_2. The graph H_v is called an *s-skein* if it is a disjoint union of s seagulls and some isolated vertices.

Lemma 12 *Let $v \in V$. Suppose that H_v is an s-skein. Then there is a minimum solution S such that $v \notin S$.*

Proof Let H_v consist of seagulls $(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_s, y_s, z_s)$ and some isolated vertices. That is, the middle vertices y_i are in N_1, while the endpoints x_i and z_i are in N_2. If $s = 1$, $\{y_1\}$ is a vertex cover of H_v and Corollary 10 yields the result. Henceforth, we assume $s \geq 2$.

Let $X = \{y_1, y_2, \ldots, y_s\}$. Clearly, X is a vertex cover of H_v. Thus, we may use X as in Lemma 9 and obtain a minimum solution S. If $v \notin S$ we are done, so let us assume $|X \setminus S| \geq 2$. Take an arbitrary i such that $y_i \in X \setminus S$. As $|X \setminus S| \geq 2$, we may pick another $j \neq i$, $y_j \in X \setminus S$. The crucial observation from the definition of H_v is that (y_j, y_i, x_i) and (y_j, y_i, z_i) are P_3s in G. As $y_i, y_j \notin S$, we have $x_i, z_i \in S$.

Hence, since the choice of i was arbitrary, we infer that for each $1 \leq i \leq s$ either $y_i \in S$ or $x_i, z_i \in S$, and, consequently, S contains a vertex cover of H_v. By Lemma 6, $S \setminus \{v\}$ is also a solution in G, a contradiction to the minimality of S. □

4 Algorithm

In this section we show our algorithm for CLUSTERVD, proving Theorem 1. The algorithm is a typical branching algorithm, where at each step we choose one branching rule and apply it. In each subcase, a number of vertices is deleted, and the parameter k drops by this number. If k becomes negative, the current subcase is terminated with a negative answer. On the other hand, if k is non-negative and G is a cluster graph, the vertices deleted in this subcase form a solution of size at most k.

4.1 Preprocessing

At each step, we first preprocess simple connected components of G.

Lemma 13 *For each connected component C of G, in linear time, we can:*

1. *conclude that C is a clique; or*
2. *conclude that C is not a clique, but identify a vertex w such that $C \setminus \{w\}$ is a cluster graph; or*
3. *conclude that none of the above holds.*

Proof On each connected component C, we perform a depth-first search. At every stage, we ensure that the set of already marked vertices induces a clique.

When we enter a new vertex, w, adjacent to a marked vertex v, we attempt to maintain the above invariant. We check if the number of marked vertices is equal to the number of neighbours of w which are marked; if so then the new vertex w is marked. Since w is adjacent to every marked vertex, the set of marked vertices remains a clique. Otherwise, there is a marked vertex u such that $uw \notin E(G)$, and we may discover it by iterating once again over edges incident to w. In this case, we have discovered a $P_3(u, v, w)$ and C is not a clique. At least one of u, v, w must be deleted to make C into a cluster graph. We delete each one of them, and repeat the algorithm (without further recursion) to check if the remaining graph is a cluster graph. If one of the three possibilities returns a cluster graph, then (2) holds. Otherwise, (3) holds.

If we have marked all vertices in a component C while maintaining the invariant that marked vertices form a clique, then the component C is a clique. □

For each connected component C that is a clique, we disregard C. For each connected component C that is not a clique, but $C \setminus \{w\}$ is a cluster graph for some w, we may greedily delete w from G: we need to delete at least one vertex from C, and w hits all P_3s in C. Thus, henceforth we assume that for each connected component C of G and for each $v \in V(C)$, $C \setminus \{v\}$ is not a cluster graph. In other words, we assume that we need to delete at least two vertices to solve each connected component of G.

4.2 Accessing H_v in Linear Time

Let us now fix a vertex $v \in V$ and let C be its connected component in G. Note that, as H_v contains parts of the complement of G, it may have size superlinear in the size of G. Therefore we now develop a simple oracle access to H_v that allows us to claim linear dependency on the graph size in the time bound.

Lemma 14 *Given a designated vertex $v \in V$, one can, in time linear in the size of G, either compute a vertex w of degree at least 3 in H_v, together with its neighbourhood in H_v, or explicitly construct the graph H_v.*

Proof First, mark vertices of N_1 and N_2. Second, for each vertex of G compute its number of neighbours in N_1 and N_2. This information, together with $|N_1|$, suffices to compute degrees of vertices in H_v. Hence, we may identify a vertex of degree at least 3 in H_v, if it exists. If we find such a vertex, say w, then computing $N_{H_v}(w)$ takes time linear in the size of G. If no such vertex w exists, the complement of $G[N_1]$ has size linear in $|N_1|$ and we may construct H_v in linear time in a straightforward manner. □

In the algorithm of Theorem 1, we would like to make a decision depending on the size of the minimum vertex cover of H_v. By the preprocessing step, C is not a clique, and by Lemma 3, H_v contains at least one edge, thus $\mathrm{VC}(G) \geq 1$. We now note that we can find a small vertex cover of G in linear time.

Lemma 15 *In linear time, we can determine whether H_v has a minimum vertex cover of size 1, of size 2, or of size at least 3. Moreover, in the first two cases we can find the vertex cover in the same time bound.*

Proof We use Lemma 14 to find, in linear time, a vertex w with degree at least 3, or generate H_v explicitly. In the latter case, H_v has vertices of degree at most 2, and it is straightforward to compute its minimum vertex cover in linear time.

If we find a vertex w of degree at least 3 in H_v, then w must be in any vertex cover of size at most 2. We proceed to delete w and restart the algorithm of Lemma 14 on the remaining graph to check if H_v in $G \setminus w$ has a vertex cover of size 0 or 1. We perform at most 2 such restarts. Finally, if we do not find a vertex cover of size at most 2, it must be the case that the minimum vertex cover contains at least 3 vertices. □

4.3 Subroutine: Branching on H_v

We are now ready to present a branching algorithm that guesses the 'correct' vertex cover of H_v, for a fixed vertex v. That is, we are now working in the setting where we look for a minimum solution to CLUSTERVD on (G, k) not containing v, thus, by Corollary 5, containing a vertex cover of H_v. Our goal is to branch into a number of subcases, in each subcase picking a vertex cover of H_v. By Corollary 8, our branching algorithm, to be correct, needs only to generate at least one element from each equivalence class of the 'equivalent' relation, among maximal elements in the 'dominate' relation.

The algorithm consists of a number of branching steps; in each subcase of each step we take a number of vertices into the constructed vertex cover of H_v and, consequently, into the constructed minimum solution to ClusterVD on G. At any point, the first applicable rule is applied.

First, we disregard isolated vertices in H_v. Second, we take care of large-degree vertices.

Rule 1 If there is a vertex $u \in V(H_v)$ with degree at least 3 in H_v, include either u or $N_{H_v}(u)$ into the vertex cover. That is, use the branching step $(u, N_{H_v}(u))$.

Note that Rule 1 yields a branching vector $(1, d)$, where $d \geq 3$ is the degree of u in H_v. Henceforth, we can assume that vertices have degree 1 or 2 in H_v. Assume there exists $u \in N_1$ of degree 1, with $uw \in E(H_v)$. Moreover, assume there exists a minimum solution S containing u. If $w \in S$, then, by Lemma 7, $S \setminus \{u\}$ is also a solution, a contradiction. If $w \in N_2 \setminus S$, then $(S \setminus \{u\}) \cup \{w\}$ dominates S. Finally, if $w \in N_1 \setminus S$, then $(S \setminus \{u\}) \cup \{w\}$ is equivalent to S. Hence, we infer the following greedy rule.

Rule 2 If there is a vertex $u \in N_1$ of degree 1 in H_v, include $N_{H_v}(u)$ into the vertex cover without branching. (Formally, use the branching step $(N_{H_v}(u))$.)

Now we assume vertices in N_1 are of degree exactly 2 in H_v. Suppose we have vertices $u, w \in N_1$ with $uw \in E(H_v)$. We would like to branch on u as in Rule 1, including either u or $N_{H_v}(u)$ into the vertex cover. However, note that in the case where u is deleted, Rule 2 is triggered on w and consequently the other neighbour of w is deleted. Hence, we infer the following rule.

Rule 3 If there are vertices $u, w \in N_1$, $uw \in E(H_v)$ then include either $N_{H_v}(w)$ or $N_{H_v}(u)$ into the vertex cover, that is, use the branching step $(N_{H_v}(w), N_{H_v}(u))$.

Note that Rule 3 yields the branching vector $(2, 2)$.

We are left with the case where the maximum degree of H_v is 2, there are no edges with both endpoints in N_1, and no vertices of degree one in N_1. Hence H_v must be a collection of even cycles and even paths (recall that N_2 is an independent set in H_v). On each such cycle C, of $2l$ vertices, the vertices of N_1 and N_2 alternate. Note that we must use at least l vertices for the vertex cover of C. By Lemma 7 it is optimal to greedily select the l vertices in $C \cap N_2$.

Rule 4 If there is an even cycle C in H_v with every second vertex in N_2, include $C \cap N_2$ into the vertex cover without branching. (Formally, use the branching step $(C \cap N_2)$.)

For an even path P of length $2l$, we have two choices. If we are allowed to use $l + 1$ vertices in the vertex cover of P, then, by Lemma 7, we may greedily take

$P \cap N_2$. If we may use only l vertices, the minimum possible number, we need to choose $P \cap N_1$, as it is the unique vertex cover of size l of such a path. Hence, we have an $(l, l+1)$ branch with our last rule.

Rule 5 Take the longest possible even path P in H_v and either include $P \cap N_1$ or $P \cap N_2$ into the vertex cover. That is, use the branching step $(P \cap N_1, P \cap N_2)$.

In Rule 5, we pick the longest possible path to avoid the branching vector $(1, 2)$ as long as possible; this is the worst branching vector in the algorithm of this section. Moreover, note that if we are forced to use the $(1, 2)$ branch, the graph H_v has a very specific structure.

Lemma 16 *The algorithm of* Section 4.3 *is forced to use Rule 5 with the branching vector* $(1, 2)$ *if and only if* H_v *is an* s*-skein for some* $s \geq 1$. *Moroever, Rule 5 applied to an* s*-skein results in an* $(s-1)$*-skein in both branches.*

We note that the statement of Lemma 16 is our sole motivation for introducing the notion of skeins and proving their properties in Lemma 12.

We conclude this section with the observation that the oracle access to H_v given by Lemma 14 allows us to execute a single branching step in linear time.

4.4 Main Algorithm

We are now ready to present our algorithm for Theorem 1. We assume the preprocessing (Lemma 13) is done. Pick an arbitrary vertex v. We first run the algorithm of Lemma 15 to determine if H_v has a minimum vertex cover of size 1 or 2. Then we run the algorithm of Lemma 14 to check if H_v is an s-skein for some s.

We consider the following cases2

Case 1. $\texttt{vc}(H_v) = 1$ **or** H_v **is an** s**-skein for some** s. Then, by Corollary 10 and Lemma 12, we know there exists a minimum solution not containing v. Hence, we run the algorithm of Section 4.3 on H_v.

Case 2. $\texttt{vc}(H_v) = 2$ **and** H_v **is not a 2-skein.**[1] Assume the application of Lemma 15 returned a vertex cover $X = \{w_1, w_2\}$ of H_v. By Lemma 9, we may branch into the following two subcases: in the first we look for minimum solutions containing v and disjoint with X, and in the second, for minimum solutions not containing v.

In the first case, we first delete v from the graph and decrease k by one. Then we check whether the connected component containing w_1 or w_2 is not a clique. By Lemma 11, for some $w \in \{w_1, w_2\}$, the connected component of $G \setminus \{v\}$ containing w is not a clique; finding such w clearly takes linear time. We invoke the algorithm of Section 4.3 on H_w.

In the second case, we invoke the algorithm of Section 4.3 on H_v.

Case 3. $\texttt{vc}(H_v) \geq 3$ **and** H_v **is not an** s**-skein for any** $s \geq 3$. We branch into two cases: we look for a minimum solution containing v or not containing v. In the first branch, we simply delete v and decrease k by one. In the second branch, we invoke the algorithm of Section 4.3 on H_v.

4.5 Complexity Analysis

In the previous discussion we have argued that invoking each branching step takes linear time. As in each branch we decrease the parameter k by at least one, the depth of the recursion is at most k. In this section we analyse branching vectors occurring in our algorithm. The main idea of the analysis is to treat the initial branching in Case 2. and Case 3., together with a few first branching steps of the subsequent invocation of the algorithm of Section 4.3, as a single huge branching step, with a single (long) branching vector. With such an analysis, the potential inefficiency coming from the initial branchings is outweighted by the very efficient branchings of the algorithm of Section 4.3. To finish the proof of Theorem 1, we show that the largest positive root of the equation $1 = \sum_{i=1}^{r} x^{-a_i}$ among all possible branching vectors $(a_1, a_2, \ldots, a_r)$ obtained in the analysis is strictly less than 1.9102.

As the number of resulting branching vectors in the analysis is rather large, we use a Python script for automated analysis.[2] The main reason for a large number of branching vectors is that we need to analyse branchings on the graph H_v in case where we consider v not to be included in the solution. Let us now proceed with formal arguments.

4.5.1 Analysis of the Algorithm of Section 4.3

In a few places, the algorithm of Section 4.3 is invoked on the graph H_v and we know that $\texttt{VC}(H_v) \geq h$ for some $h \in \{1, 2, 3\}$. Consider the branching tree $\mathbb{T}$ of this algorithm. For a node $x \in V(\mathbb{T})$, the *depth* of x is the number of vertices of H_v deleted on the path from x to the root. We mark some nodes of $\mathbb{T}$; the marked nodes correspond to the branching steps that we analyse together with the initial branching in Case 2. and Case 3.

Each node of depth less than h is marked. Moreover, if a node x is of depth $d < h$ and the branching step at node x has branching vector $(1, 2)$, Lemma 16 lets us infer that the graph H_v at this node is an s-skein for some $s \geq h - d$. Consquently, for all the descendants of x in $V(\mathbb{T})$ the graph H_v is a (smaller) skein and thus, again by Lemma 16, their branching vectors are all $(1, 2)$. In this case, we mark all descendants of x that are within distance (in $\mathbb{T}$) less than $h - d$. Note that in this way we may mark some descendants of x of depth equal to or larger than h. We refer to Fig. 1 for some examples of branching trees and marked vertices.

We split the analysis of an application of the algorithm of Section 4.3 into two phases: the first one contains all branching steps performed on marked nodes, and

[1]Note that the size of a minimum vertex cover of an s-skein is exactly s, so this case is equivalent to '$\texttt{VC}(H_v) = 2$ and H_v is not an s-skein for any s'.

[2]Available at http://www.mimuw.edu.pl/malcin/research/cvd and in the Appendix.

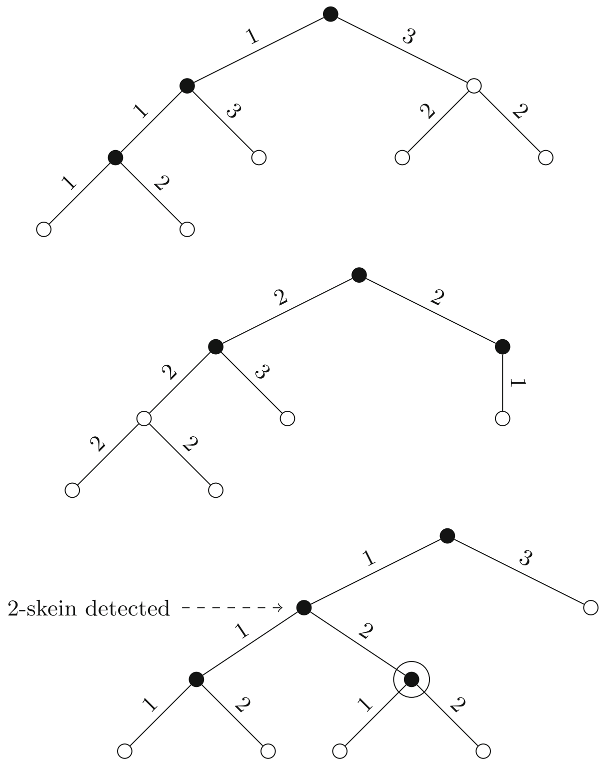


Fig. 1 Examples of branching trees for the subroutine on the graph H_v. Marked nodes for $h = 3$ are coloured black. In the last case, a 2-skein occurs in one branch, and an extra node (*encircled*) of depth 3 is marked

the second on the remaining nodes. In the second phase, we simply observe that each branching step has branching vector not worse than $(1, 2)$. In the first phase, we aim to write a single branching vector summarizing the phase, so that with its help we can balance the loss from other branches when v is deleted.

We remark that, although in the analysis we aggregate some branching steps to prove a better time bound, we always aggregate only a constant number of branches (that is, we analyse the branching on marked vertices only for constant h). Consequently, we maintain a linear dependency on the size of the graph in the running time bound.

The main property of the marked nodes in $\mathbb{T}$ is that their existence is granted by the assumption $\texttt{VC}(H_v) \geq h$. That is, each leaf of $\mathbb{T}$ has depth at least h, and, if at some node x of depth $d < h$ the graph H_v is an s-skein, we infer that $s \geq h - d$ (as the size of minimum vertex cover of an s-skein is s) and the algorithm performs s independent branching steps with branching vectors $(1, 2)$ in this case. Overall, no leaf of $\mathbb{T}$ is marked.

To analyse such branchings for $h = 2$ and $h = 3$ we employ the Python script. The procedure `branch_Hv` generates all possible branching vectors for the first phase, assuming the algorithm of Section 4.3 is allowed to pick branching vectors (1), $(1, 3)$, $(2, 2)$ or $(1, 2)$ (option `allow_skein` enables/disables the use of the $(1, 2)$ vector in the first branch). Note that all other vectors described in Section 4.3 may be simulated by applying a number of vectors (1) after one of the aforementioned branching vectors.

4.5.2 Analysis of the Algorithm of Section 4.4

Case 1 Here the algorithm of Section 4.3 performs branchings with vectors not worse than $(1, 2)$.

Case 2 If v is deleted, we apply the algorithm of Section 4.3 to H_w, yielding at least one branching step (as the connected component with w is not a clique). Hence, in this case the resulting branching vector is any vector that came out of the algorithm of Section 4.3, with all entries increased by one (for the deletion of v). Recall that in the algorithm of Section 4.3, the worst branching vector is $(1, 2)$, corresponding to the case of H_w being a skein. Consequently, the worst branching vector if v is deleted is $(2, 3)$.

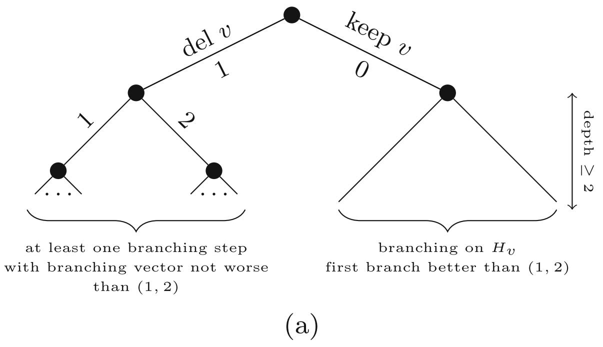
If v is not deleted, the algorithm of Section 4.3 is applied to H_v. The script invokes the procedure `branch_Hv` on $h = 2$ and `allow_skein=False` to obtain a list of possible branching vectors. For each such vector, we append entries $(2, 3)$ from the subcase when v is deleted. See Fig. 2a for an illustration of the branching tree in this case.

Case 3 The situation is analogous to the previous case. The script invokes the procedure `branch_Hv` on $h = 3$ and `allow_skein=False` to obtain a list of possible branching vectors. For each such vector, we append the entry (1) from the subcase when v is deleted. See Fig. 2b for an illustration of the branching tree in this case.

4.5.3 Summary

We infer that the largest root of the equation $1 = \sum_{i=1}^{r} x^{-a_i}$ occurs for the branching vector $(1, 3, 3, 4, 4, 5)$ and is less than 1.9102. This branching vector corresponds to Case 3. and the algorithm of Section 4.3, invoked on H_v, first performs a branching step with the vector $(1, 3)$ and in the branch with 1 deleted vertex, finds H_v to be a 2-skein and performs two independent branching steps with vectors $(1, 2)$. Note that the last example on Fig. 1 corresponds to the branching on H_v in this worst-case scenario.

This analysis concludes the proof of Theorem 1. We remark that the worst branching vector in Case 2. is $(2, 2, 3, 3, 3)$ (with solution $x < 1.8933$), corresponding to the case with a single $(1, 2)$-branch when v is deleted and a 2-skein in the case when v is kept. Obviously, the worst case in Case 1. is the golden-ratio branch $(1, 2)$ with solution $x < 1.6181$.


del v
1
keep v
0
1
2
depth ≥ 2
at least one branching step with branching vector not worse than $(1,2)$
branching on H_v first branch better than $(1,2)$
(a)

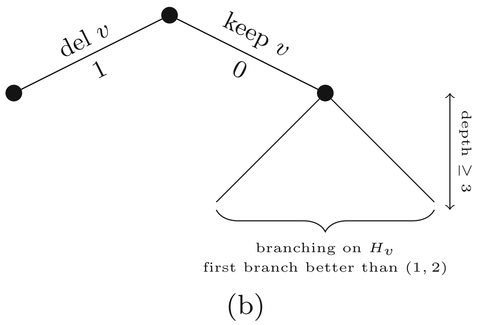


Fig. 2 Branching trees for Case 2. and Case 3

5 Co-cluster Setting

In this section we show that the same result as in Theorem 1 holds for the complement version of the problem, called CO-CLUSTER VERTEX DELETION (COCLUSTERVD for short). Here, one wants to delete at most k vertices from the input graph to obtain a co-cluster graph (a complement of a cluster graph).

Theorem 17 CO-CLUSTER VERTEX DELETION *can be solved in* $\mathcal{O}(1.9102^k(n+m))$ *time and polynomial space on an input* (G,k) *with* $|V(G)|=n$ *and* $|E(G)|=m$.

Observe that, if one wants to solve COCLUSTERVD, one may complement the input graph and solve CLUSTERVD instead. However, with such an approach we do not obtain a linear dependency on the size of the input. To obtain it, we need to reengineer our preprocessing routine (Lemma 13) and the oracle access to the graph H_v (Section 4.2) to work in the co-cluster setting.

5.1 Preprocessing

We need to show the following variant of Lemma 13.

Lemma 18 *Given the complement of the graph G, in time linear in the input size, we can for each connected component C of G:*

1. *conclude that C is a clique; or*
2. *conclude that C is not a clique, but identify a vertex w such that $C \setminus \{w\}$ is a cluster graph; or*
3. *conclude that none of the above holds.*

Proof Denote by $\bar{G}$ the complement of G, given as an input. First, we compute the connected components of G, i.e., the complement-connected components of $\bar{G}$. For this, we may use a linear-time algorithm of Ito and Yokohama [16]. Next, we construct $\bar{G}[C]$ for each of the discovered components C. This also takes linear time.

Now, we can process each component separately. On each of them we shall spend time proportional to the size of $\bar{G}[C]$. The algorithm of [16] actually lets us compute a DFS tree of $G[C]$. We exploit this feature to simulate our approach from Lemma 13: we arrange vertices in the preorder of the DFS tree. Now, it suffices to find the first vertex w which in $\bar{G}[C]$ has a neighbour u among the preceding vertices. We conclude that in $G[C]$ vertices u, w are endpoints of a P_3 whose midpoint is v, the parent of w in the tree.

Thus, as in the proof of Lemma 13, u, v, w are the tree vertices which we need to verify. In order to check these candidates we simply compute the complement-connected components, again using the results of [16]. □

5.2 Accessing H_v in Linear Time

Recall that we have fixed vertex v, and we are to give an oracle access to H_v, given $\bar{G}$ as an input. We first note the following.

Lemma 19 *We can compute sets N_1 and N_2 in time linear in the size of $\bar{G}$.*

Proof First compute N_1 by marking all non-neighbours of v in $\bar{G}$. Then, for each $u \in V \setminus (N_1 \cup \{v\})$ observe that $u \in N_2$ if and only if $|N_{\bar{G}}(u) \cap N_1| < |N_1|$. This condition can be verified in time linear in the size the neighbourhood of u in the graph $\bar{G}$, and hence in time linear in the size of $\bar{G}$ for all vertices u. □

We now prove an analogoue of Lemma 14.

Lemma 20 *Given a designated vertex $v \in V$, one can in time linear in the size of $\bar{G}$ either compute a vertex w of degree at least 3 in H_v, together with its neighbourhood in H_v, or explicitly construct the graph H_v.*

Proof First, mark vertices of N_1 and N_2 using Lemma 19. Second, for each vertex of V compute its number of neighbours in N_1 and N_2; note that this can be done by inspecting all edges of $\bar{G}$. This information, together with $|N_1|$, suffices to compute degrees of vertices in H_v. Hence, we may identify a vertex of degree at least 3 in H_v, if it exists. For such a vertex w, we may compute $N_{H_v}(w)$ in time linear in the size of $\bar{G}$ by inspecting all vertices $u \in N_1 \cup N_2$ one-by-one.

If no such vertex w exists, the number of non-edges of $\bar{G}$ (i.e., edges of G) between N_1 and N_2 is linear in $|N_1| + |N_2|$ and we can compute them in time linear in the size of $\bar{G}$. Together with $\bar{G}[N_1]$, they form H_v. □

Finally, we observe that the following analogoue of Lemma 15 is straightforward, as the proof of Lemma 15 accesses the graph H_v and G only via Lemma 14, and we have already adapted this lemma to the co-cluster setting.

Lemma 21 *In time linear in the size of $\bar{G}$, we can determine whether H_v has a minimum vertex cover of size 1, of size 2, or of size at least 3. Moreover, in the first two cases we can find the vertex cover in the same time bound.*

This analysis concludes the proof of Theorem 17.

6 Conclusions and Open Problems

We have presented a new branching algorithm for CLUSTER VERTEX DELETION. We hope our work will trigger a race for faster FPT algorithms for CLUSTERVD, as it was in the case of the famous VERTEX COVER problem.

Repeating after Hüffner et al. [15], we would like to re-pose here the question for a linear vertex-kernel for CLUSTERVD. As CLUSTERVD is a special case of the 3-HITTING SET problem, it admits an $\mathcal{O}(k^2)$-vertex kernel in the unweighted case and an $\mathcal{O}(k^3)$-vertex kernel in the weighted one [1, 2]. However, CLUSTER EDITING is known to admit a much smaller $2k$-vertex kernel, so there is a hope for a similar result for CLUSTERVD.

Acknowledgments This work has been partially supported by NCN grant N206567140 and Foundation for Polish Science.

Appendix

Python Script Automating Complexity Analysis

Below we include a Python script for automated complexity analysis, together with a comment and pseudocode of the main routine. The script is also available at www.mimuw.edu.pl/malcin/research/cvd.

```
import scipy.optimize

def value(vector):
  """compute the value of a branching vector"""
  def h(x):
    return sum([x**(-v) for v in vector])-1
  return scipy.optimize.brenth(h,1, 100)


def join(first, then):
  """peform 'then' in each branch after the execution of 'first' """
  return [x+y for x in first for y in then]


def add(a, vector):
  """add a to each element of a vector"""
  return join([a], vector)


golden_branch = [1,2]   # golden-ratio branch, worst branch in Hv


def skein_vector(s):
  """returns branching vector from s-skein"""
  if s == 0:
    return [0]
  else:
    return join(skein_vector(s-1), golden_branch)


Hv_branches = dict()

def branch_Hv(h, allow_skein=True):
  """return list of possible branching vectors on Hv, where each subcase
  deletes at least h vertices; if allow_skein=False, ignore the case when
  Hv is a skein"""
  if h <= 0:
    return [[0]]
  # Memoize for speed-up
  if Hv_branches.has_key((h, allow_skein)):
    return Hv_branches[(h, allow_skein)]
  res = []
  # If skein is allowed, add appriopriate vector.
  if allow_skein:
    res.append(skein_vector(h))
  # Greedy step.
  # Can be applied multiple times to simulate larger drop.
  res += [add(1, v) for v in branch_Hv(h-1)]
  # Rule 1: (1,3) branch.
  # Branches (1,d) for d>3 may be simulated by subsequent greedy steps
  res += [add(1, v1) + add(3, v2) for v1 in branch_Hv(h-1) for v2 in branch_Hv(h-3)]
  # Rule 3: (2,2) branch
  res += [add(2, v1) + add(2, v2) for v1 in branch_Hv(h-2) for v2 in branch_Hv(h-2)]
  # Rule 5, if Hv is not a skein, yields (2,3) branch which can be simulated
  # by (2,2) branch + greedy step in one branch, so we omit it here.
  Hv_branches[(h, allow_skein)] = res
  return res


vectors = []  # all branching vectors

# (1,2) vector from standard branching on Hv
vectors.append(golden_branch)

# Case: MinVC(Hv) = 2, Hv is not a 2-skein
vectors += [add(1, golden_branch) + v for v in branch_Hv(2, allow_skein=False)]

# Case: MinVC(Hv) >= 3, Hv is not a skein
vectors += [[1] + v for v in branch_Hv(3, allow_skein=True)]

for v in vectors:

  print ("%.11f  : " % value(v)), v

print "Largest root: %.11f" % max([value(v) for v in vectors])
```

The script uses three small auxiliary routines.

- The `join` routine takes as input two lists (branching vectors) l_1 and l_2 and produces a branching vector $(a + b : a \in l_1, b \in l_2)$. It corresponds to the case when we independently apply a branching step with branching vector l_2 in each subcase of a branching step with branching vector l_1.
- The `add` routine takes as input an integer a and a list (branching vector) l_1 and adds a to each element of l_1. This is equivalent to `join` of l_1 and a single-element list $[a]$.
- The `skein_vector` routine returns a branching vector on an s-skein, where s is given as input. Note that the result is the golden ratio vector $(1, 2)$, `join`ed with itself s times.

Most of the work is done in the `branch_Hv` routine. The subsequent steps are as follows.

1. For $h \leq 0$ (no lower bound on the size of minimum vertex cover of H_v), we may perform no branching at all, so we return a single branching vector (0).
2. We check whether the result for input values has been already computed and memoized in some global dictionary.
3. If `allow_skein` $=$ `True`, we append to the result a branching vector that happens if the input graph is an h-skein. This vector is obtained through the `skein_vector` routine.
4. We consider a greedy step, where some vertex is greedily added to the solution. We invoke recursively `branch_Hv`$(h-1)$ (`allow_skein` is set to `True` by default) and add one to each element of each of the resulting branching vectors. All computed vectors are appended to the result.
5. We consider Rule 1, where a $(1, d)$ branch occurs for $d \geq 3$. Such a branch can be simulated by a $(1, 3)$ branch and some subsequent greedy steps. Hence, we compute $v_1 =$ `branch_Hv`$(h-1)$, $v_2 =$ `branch_Hv`$(h-3)$, add 1 to each element of each vector of v_1, add 3 to each element of each vector of v_2 and concatenate each vector of v_1 with each vector of v_2. All computed concatenations are appended to the result.
6. We consider Rule 3, where a $(2, 2)$ branch occurs. We proceed as in the previous case, with $v_1 = v_2 =$ `branch_Hv`$(h-2)$.
7. We observe that all other branches may be simulated by either the $(1, 3)$ branch or the $(2, 2)$ branch with some subsequent greedy steps. Hence, we return the computed result after memoizing it in some global dictionary.

In the main body of the script, it first computes candidate branching vectors.

1. We first consider the golden-ratio branching vector $(1, 2)$ from Case 1.
2. In Case 2, any branching vector consists of two parts. In the first part, we delete v and perform at least one branch, not worse than the golden ratio branch; hence, we add 1 to each element of the golden ratio branch. In the second part, we consider all possible branching vectors returned by `branch_Hv`$(2,$ `allow_skein` $=$ `False`$)$.

3. In Case 3, we append (1) to each branching vector returned by `branch_Hv` (3, `allow_skein` = `False`).

Finally, the script computes the corresponding base of the exponent for each candidate branching vector and outputs the largest one.

References

1. Abu-Khzam, F.N.: A kernelization algorithm for d-Hitting Set. J. Comput. Syst. Sci. **76**(7), 524–531 (2010)
2. Abu-Khzam, F.N., Fernau, H. Kernels: Annotated, proper and induced. In: Proceedings of IWPEC 2006, volume 4169 of Lecture Notes in Computer Science, pp. 264–275. Springer (2006)
3. Ailon, N., Charikar, M., Newman, A.: Aggregating inconsistent information: ranking and clustering. J ACM **55**(5), 23:1–23:27 (2008)
4. Bansal, N., Blum, A., Chawla, S.: Correlation clustering. Mach. Learn. **56**, 89–113 (2004)
5. Ben-Dor, A., Shamir, R., Yakhini, Z.: Clustering gene expression patterns. J. Comput. Biol. **6**(3/4), 281–297 (1999)
6. Böcker, S.: A golden ratio parameterized algorithm for cluster editing. J. Discrete Algorithms **16**, 79–89 (2012)
7. Böcker, S., Baumbach, J.: Cluster editing. In: Bonizzoni, P., Brattka, V., Löwe, B. (eds.) The Nature of Computation. Logic, Algorithms, Applications, volume 7921 of Lecture Notes in Computer Science, pp. 33–44. Springer, Berlin Heidelberg (2013)
8. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Inf. Process. Lett. **58**(4), 171–176 (1996)
9. Charikar, M., Guruswami, V., Wirth, A.: Clustering with qualitative information. J. Comput. Syst. Sci. **71**(3), 360–383 (2005)
10. Chen, J., Meng, J.: A $2k$ kernel for the cluster editing problem. J. Comput. Syst. Sci. **78**(1), 211–220 (2012)
11. Fernau, H.: A top-down approach to search-trees: improved algorithmics for 3-hitting set. Algorithmica **57**(1), 97–118 (2010)
12. Fomin, F., Kratsch, D.: Exact Exponential Algorithms. Texts in Theoretical Computer Science. Springer, Berlin Heidelberg (2010)
13. Fomin, F.V., Kratsch, S., Pilipczuk, M., Pilipczuk, M., Villanger, Y.: Tight bounds for parameterized complexity of cluster editing with a small number of clusters. J. Comput. Syst. Sci. **80**(7), 1430–1447 (2014)
14. Gramm, J., Guo, J., Hüffner, F., Niedermeier, R.: Automated generation of search tree algorithms for hard graph modification problems. Algorithmica **39**(4), 321–347 (2004)
15. Hüffner, F., Komusiewicz, C., Moser, H., Niedermeier, R.: Fixed-parameter algorithms for cluster vertex deletion. Theory Comput. Syst. **47**(1), 196–217 (2010)
16. Ito, H., Yokoyama, M.: Linear time algorithms for graph search and connectivity determination on complement graphs. Inf. Process. Lett. **66**(4), 209–213 (1998)
17. Komusiewicz, C.: Parameterized Algorithmics for Network Analysis: Clustering & Querying. PhD thesis, Technische Universität Berlin. Available at http://fpt.akt.tu-berlin.de/publications/diss-komusiewicz.pdf (2011)
18. Komusiewicz, C., Uhlmann, J.: Cluster editing with locally bounded modifications. Discret. Appl. Math. **160**(15), 2259–2270 (2012)
19. Protti, F., da Silva, M.D., Szwarcfiter, J.L.: Applying modular decomposition to parameterized cluster editing problems. Theory Comput. Syst. **44**(1), 91–104 (2009)
20. Reed, B.A., Smith, K., Vetta, A.: Finding odd cycle transversals. Oper. Res. Lett. **32**(4), 299–301 (2004)
21. Wahlström, M.: Algorithms, measures, and upper bounds for satisfiability and related problems. PhD thesis, Linköping Studies in Science and Technology. Available at http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-8714 (2007)