



Parameterized approximation algorithms for weighted vertex cover ☆,☆☆

Soumen Mandal^{a,*}, Pranabendu Misra^b, Ashutosh Rai^a, Saket Saurabh^{c,d}

^a Department of Mathematics, IIT Delhi, New Delhi, India

^b Chennai Mathematical Institute, Chennai, India

^c Institute of Mathematical Sciences, Chennai, India

^d University of Bergen, Bergen, Norway

ARTICLE INFO

Keywords:

Weighted vertex cover
Parameterized algorithms
Approximation algorithms
Parameterized approximation

ABSTRACT

A *vertex cover* of a graph is a set of vertices of the graph such that every edge has at least one endpoint in it. In this work, we study WEIGHTED VERTEX COVER with solution size as a parameter. Formally, in the (k, W) -VERTEX COVER problem, given a graph G , an integer k , a positive rational W , and a weight function $w : V(G) \rightarrow \mathbb{Q}^+$, the question is whether G has a vertex cover of size at most k of weight at most W , with k being the parameter. An (a, b) -bi-criteria approximation algorithm for (k, W) -VERTEX COVER either produces a vertex cover S such that $|S| \leq ak$ and $w(S) \leq bW$, or decides that there is no vertex cover of size at most k of weight at most W . We obtain the following results.

- A simple $(2, 2)$ -bi-criteria approximation algorithm for (k, W) -VERTEX COVER in polynomial time by modifying the standard LP-rounding algorithm.
- A simple exact parameterized algorithm for (k, W) -VERTEX COVER running in $\mathcal{O}^*(1.4656^k)$ time¹.
- A $(1 + \epsilon, 2)$ -approximation algorithm for (k, W) -VERTEX COVER running in $\mathcal{O}^*(1.4656^{(1-\epsilon)k})$ time.
- A $(1.5, 1.5)$ -approximation algorithm for (k, W) -VERTEX COVER running in $\mathcal{O}^*(1.414^k)$ time.
- A $(2 - \delta, 2 - \delta)$ -approximation algorithm for (k, W) -VERTEX COVER running in $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \left(\frac{\delta k+i}{\delta k - \frac{2i\delta}{1-2\delta}}\right)\right)$ time for any $\delta < 0.5$. For example, for $(1.75, 1.75)$ and $(1.9, 1.9)$ -approximation algorithms, we get running times of $\mathcal{O}^*(1.272^k)$ and $\mathcal{O}^*(1.151^k)$ respectively.

Our algorithms (expectedly) do not improve upon the running times of the existing algorithms for the unweighted version of VERTEX COVER. When compared to algorithms for the weighted version, our algorithms are the first ones to the best of our knowledge which work with arbitrary weights, and they perform well when the solution size is much smaller than the total weight of the desired solution.

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

☆☆ Preliminary version of this article will appear in LATIN 2024.

* Corresponding author.

E-mail addresses: soumen.mandal@maths.iitd.ac.in (S. Mandal), pranabendu@cmi.ac.in (P. Misra), ashutosh.rai@maths.iitd.ac.in (A. Rai), saket@imsc.res.in (S. Saurabh).

¹ Here, the $\mathcal{O}^*$ notation hides factors polynomial in n .

1. Introduction

In the VERTEX COVER problem, we are given a graph G , and are asked to find a smallest set $S \subseteq V(G)$ of vertices such that every edge in the graph has at least one endpoint in S . This problem is known to be NP-complete and is in fact one of the 21 NP-complete problems listed by Karp [1]. This makes it unlikely that a polynomial time algorithm exists for the problem, and hence it has been studied extensively in the fields of *approximation algorithms* and *parameterized algorithms*.

For VERTEX COVER, there are several polynomial time 2-approximation algorithms [2]. On the other hand, it is known that VERTEX COVER is APX-complete, and hence cannot have a Polynomial Time Approximation Scheme (PTAS) unless $P = NP$ [3]. In fact, under the assumption that $P \neq NP$, there cannot be a $(\sqrt{2} - \epsilon)$ -approximation algorithm for VERTEX COVER [4]. Further, assuming the Unique Games Conjecture, there cannot be a $(2 - \epsilon)$ -approximation algorithm for VERTEX COVER [5]. This result basically shows that we cannot do better than the known approximation algorithms in polynomial time, unless the Unique Games Conjecture is false.

VERTEX COVER is also one of the most well-studied problems in parameterized algorithms, with the size of the vertex cover being the parameter. Formally, given a graph G and an integer k , it asks whether G has a vertex cover of size at most k . A branching algorithm for VERTEX COVER runs in time $\mathcal{O}^*(2^k)$, and there have been several results improving the running time to $\mathcal{O}^*(1.25284^k)$ [6].

There is another emerging field, *parameterized approximation algorithms*, that combines the fields of approximation algorithms and parameterized algorithms to get *better approximations* for problems in FPT time, than what is achievable in polynomial time. Naturally, the running time of a parameterized approximation algorithm, if one exists, is smaller than the running time of the exact FPT algorithm for the same problem. For VERTEX COVER, a parameterized $(1 + \epsilon)$ -approximation algorithm is known to run in $\mathcal{O}^*(c^{(1-\epsilon)k})$ time [7,8], where the constant c is the base of the exponent of the best known FPT algorithm for vertex cover, running in time $\mathcal{O}^*(c^k)$. The running time was further improved by Brankovic and Fernau [9] for any $\epsilon \in (0, 0.5]$, and in particular, they gave a 1.5-approximation algorithm running in time $\mathcal{O}^*(1.0883^k)$. Recently, Kulik and Shachnai [10] introduced randomized branching as a tool for parameterized approximation and develop the mathematical machinery for its analysis. Using this they obtained the best known running times of parameterized approximation algorithms for VERTEX COVER and 3-HITTING SET for a wide range of approximation ratios. We refer to the survey [11] for other developments in the field of parameterized approximation algorithms.

While parameterized approximation algorithms for the unweighted version of VERTEX COVER have been extensively studied, the same cannot be said about the weighted version of the problem. In the weighted version of the VERTEX COVER problem, we are asked to find a vertex cover of minimum weight in the graph. In the decision version of the problem, we are given a graph G , weight function $w : V(G) \rightarrow \mathbb{Q}^+$, and a positive rational W ; and the question is whether G has a vertex cover of weight at most W . We can get a 2-approximation algorithm for the weighted vertex cover problem by doing a simple LP-rounding, and the negative results for approximation of the unweighted version carry over to weighted version of the problem as well. From the point of view of parameterized complexity, the weighted version is less studied than the unweighted version, but still a number of results have progressively brought the running time down to $\mathcal{O}^*(1.347^W)$, when parameterized by the weight W [12–14]. Shachnai and Zehavi [14] also designed an algorithm, that as an input to WEIGHTED VERTEX COVER, takes a vertex weighted graph with all vertex weights being at least 1, and a rational number W . The algorithm either returns a vertex cover of weight at most W or returns that there is no vertex cover of weight at most W . The running time of this algorithm is $\mathcal{O}^*(1.363^s)$, where s is the minimum size of a Vertex Cover of weight at most W . If no such Vertex Cover exists, $s = \min\{W, m\}$, where m is the maximum size of a Vertex Cover (for the unweighted version). One issue with all these algorithms is that they require all the vertex weights to be at least 1, because it can be seen through a simple reduction that otherwise the problem is not likely to be FPT when parameterized by W . Also, all these algorithms do not give any *guarantee on the number of vertices in the solution*, which can be at most W , but can be much smaller, since all the weights are at least 1.

This gives rise to the natural question of finding parameterized (approximation) algorithms for WEIGHTED VERTEX COVER, where we do not want too many vertices in the solution, with the size of the solution being the parameter. Formally, given a graph G , weight function $w : V(G) \rightarrow \mathbb{Q}^+$, integer k and positive rational W , the question is whether G has a vertex cover of size at most k and weight at most W . We call the problem (k, W) -VERTEX COVER, where the parameter is k . In [14], the authors define this problem, and show that it is NP-complete even on bipartite graphs.

Such a parameterization was considered in [15], where the authors gave exact parameterized algorithm for WEIGHTED FEEDBACK VERTEX SET parameterized by the solution size. Later, Saurabh [16] asked a similar question for the weighted version of DIRECTED FEEDBACK VERTEX SET, which was resolved by Kim et al. [17] recently using the technique of flow-augmentation. There is some other recent work being done with similar parameterizations for cut problems [18,19], but somehow no work has been done on the WEIGHTED VERTEX COVER problem with this parameterization to the best of our knowledge. We fill this gap by studying (k, W) -VERTEX COVER from the perspective of parameterized (approximation) algorithms.

1.1. Our problem, results and methods

We investigate (k, W) -VERTEX COVER from the point of view of approximation algorithms, parameterized algorithms, and parameterized approximation algorithms, with the size of the vertex cover k being the parameter for the latter two. This parameter is arguably better than the parameter W , since parameterization by W requires the weights to be at least 1 for the problem to be FPT, and in those cases the value of k is at most W , but it can be much smaller.

For approximation, we look at the notion of bi-criteria approximation, where we approximate both the size and weight of the vertex cover, within some factors a and b respectively. More formally, given an instance (G, w, k, W) of (k, W) -VERTEX COVER, an (a, b) -approximation algorithm either correctly determines that there is no vertex cover of size at most k and weight at most W or

outputs a vertex cover S of G , such that $|S| \leq ak$ and $w(S) \leq bW$. It is easy to see that the $(2 - \epsilon)$ hardness for VERTEX COVER rules out any $(c, 2 - \epsilon)$ - and $(2 - \epsilon, c)$ -approximation algorithms running in polynomial time for any constants $c, \epsilon > 0$. In particular, it rules out $(2 - \epsilon, 2 - \epsilon)$ -approximation algorithms running in polynomial time. We complement this negative result by observing a simple $(2, 2)$ -approximation algorithm. This is done by modifying the known factor 2-approximation LP-rounding algorithm for VERTEX COVER. After this we move to parameterized algorithms and give the following results.

- A simple exact parameterized algorithm for (k, W) -VERTEX COVER running in $\mathcal{O}^*(1.4656^k)$ time.
- A $(1 + \epsilon, 2)$ -approximation algorithm for (k, W) -VERTEX COVER running in $\mathcal{O}^*(1.4656^{(1-\epsilon)k})$ time.
- A $(1.5, 1.5)$ -approximation algorithm for (k, W) -VERTEX COVER in $\mathcal{O}^*(1.414^k)$ time.
- A general $(2 - \delta, 2 - \delta)$ -approximation algorithm for (k, W) -VERTEX COVER in $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right)$ time for any $\delta < 0.5$. For example, for $(1.75, 1.75)$ and $(1.9, 1.9)$ -approximation algorithms, the running times are $\mathcal{O}^*(1.272^k)$ and $\mathcal{O}^*(1.151^k)$ respectively.

The exact parameterized algorithm does a simple two-way branching on a vertex of degree at least 3, and solves (k, W) -VERTEX COVER in polynomial time on graphs of maximum degree at most 2 in the base case using dynamic programming. The result can also be obtained by observing fact that such graphs have treewidth at most 2, and giving a fixed parameter tractable algorithm for (k, W) -VERTEX COVER on graphs of bounded treewidth. We include the direct approach by dynamic programming in the main body of the paper and defer the treewidth algorithm to the appendix. Unlike VERTEX COVER, we do not have any clever branching rule that can avoid branching on vertices of degree higher than 2. In the VERTEX COVER problem, one only needs to care about the solution's size (weight in case of WEIGHTED VERTEX COVER). To get to better running times, faster algorithms for VERTEX COVER and WEIGHTED VERTEX COVER avoid branching on small degree vertices by exploiting the fact that we only need to bound either the size or the weight of the solution. In the case of (k, W) -VERTEX COVER, we need to take care of both the parameters simultaneously. This makes it difficult to design reduction rules to avoid branching even on a degree 3 vertex and match the running time of improved algorithms for VERTEX COVER.

Then we use polynomial time $(2, 2)$ -approximation algorithm along with a simple branching to get a $(1 + \epsilon, 2)$ -approximation algorithm running in time $\mathcal{O}^*(1.4656^{(1-\epsilon)k})$. This algorithm follows along the ideas of [7] and [8].

Then we turn our attention to bringing both the approximation factors below 2. This is the main technical contribution of the paper. The main idea behind our algorithm is that if we do the branching on the vertices of large weight initially, applying the $(2, 2)$ -approximation algorithm on the remaining instance cannot make the solution much worse. Branching on a maximum weight vertex first till the budget has dropped sufficiently, and then combining that with $(2, 2)$ -approximation algorithm gives us a $(1.5, 1.5)$ -approximation algorithm running in time $\mathcal{O}^*(2^{\frac{k}{2}})$.

Finally, we generalize this algorithm and give a $(2 - \delta, 2 - \delta)$ -approximation algorithm running in $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right)$ time for any $\delta < 0.5$. This generalization requires different stopping criteria and a more involved analysis as compared to the previous algorithm to achieve the desired approximation factor. For any fixed δ , notice that in the running time, there are only polynomially many (in k) terms inside the summand and each of them is of the form $\binom{A}{B}$ which can be at most 2^A . Here maximum value of A will be $\frac{k}{2}$ corresponding to $i = \frac{\delta k(1-2\delta)}{2\delta}$. So the maximum term inside the summand will always be bounded by $2^{\frac{k}{2}}$ and since there are only polynomially many (in k) terms inside the summand, $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right)$ will be at most $\mathcal{O}^*(2^{\frac{k}{2}})$ which is of the form $\mathcal{O}^*(c^k)$ for $c = 1.414 > 1$. We do a finer analysis to compute this constant c for each $\delta < 0.5$, where c is strictly smaller than 1.414. For example, putting the value of δ to be 0.25 and 0.1, that is, for $(1.75, 1.75)$ and $(1.9, 1.9)$ -approximation algorithms the value of c turns out to be 1.272 and 1.151 respectively. Details of how this c has been computed can be seen in the Section 7.1 (see Fig. 1 for a graph of running times for different values of δ).

Unlike the existing FPT algorithms for weighted version of the problem parameterized by W , which work for the weights of vertices being at least one, our algorithms work for all positive rational weights.

2. Preliminaries

In this section, we give the notations and definitions, along with preliminaries and some known results which are used in the paper.

For a graph $G = (V, E)$, we denote the set of vertices of the graph by $V(G)$ and the set of edges of the graph by $E(G)$. We denote $|V(G)|$ by n , where the graph is clear from the context. For a set $S \subseteq V(G)$, the subgraph of G induced by S is denoted by $G[S]$ and it is defined as the subgraph of G with vertex set S and edge set $\{\{u, v\} \in E(G) : u, v \in S\}$. We denote the subgraph $G[V(G) \setminus S]$, obtained after deleting S (and the edges incident to the vertices in S) as $G - S$. For ease of notation, we will use uv to denote an edge of a graph instead of $\{u, v\}$ where u and v are the endpoints of the edge. For any graph $G = (V, E)$ and $v \in V$, a vertex $u \in V$ is said to be a neighbor of v if $vu \in E(G)$. The open neighborhood of v is denoted by $N(v)$ and it is defined as $N(v) = \{u \in V : vu \in E(G)\}$. We denote degree of a vertex v by $d(v)$ where $d(v) = |N(v)|$. The closed neighborhood of a vertex v is $N[v] = N(v) \cup \{v\}$. By $\Delta(G)$, we denote the maximum degree of a vertex in G . We denote a polynomial function of n as $\text{poly}(n)$.

We need to define the notion of a branching tree for the analysis of our algorithms. An algorithm or a procedure $\mathcal{A}$ on an instance I of a problem is called a branching algorithm if $\mathcal{A}$ can potentially recursively call itself on instances $I_1, I_2, \dots, I_t$ for $t \geq 2$ such that

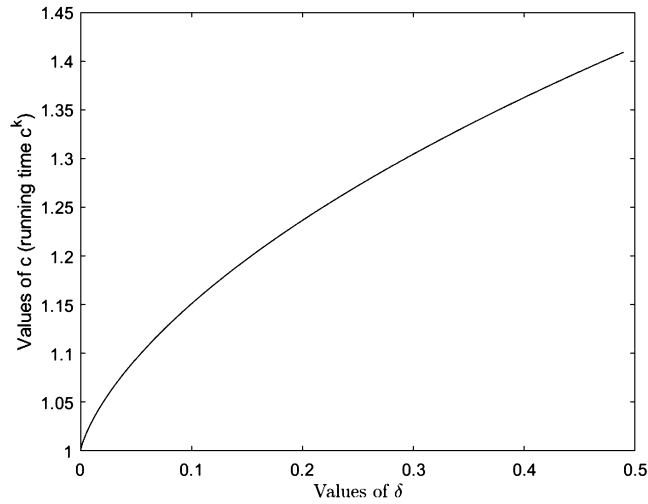


Fig. 1. Running time of $(2 - \delta, 2 - \delta)$ -approximation algorithm for different values of δ .

output of $\mathcal{A}$ depends on the output of the recursive calls. A run of the algorithm on an instance I is said to have executed a branching step, if it recursively calls itself on at least two instances. If the algorithm applies a reduction rule and makes a call to itself on only one instance, then we say that branching step is not executed.

A branching tree associated with an algorithm $\mathcal{A}$ and an input instance I is a tree where the root node of the tree is associated with the instance I . Further, if the algorithm on input I makes a recursive call to instance I' (which itself can be during a recursive call) without executing a branching step, then I' is associated with the same node in the branching tree as the instance I is. Thus whenever a reduction rule is applied to an instance, both the original and reduced instances are associated with the same node. If on an instance I , branching step is executed and recursive call is made on the instances $I_1, I_2, \dots, I_t$, then the node associated with I in the branching tree would have t children, each associated with one of the instances on which the recursive call was made. If there is no branching step executed on any of the instances associated with a node in the branching tree, then that node would be a leaf node in the branching tree. We also use the concept of *layers* in the branching tree, where layer 0 consists of the root node of the branching tree and layer i consists of the nodes which are at distance i from the root in the branching tree.

We define the branching tree in this way because application of a reduction rule will only take polynomial time in our case, and associating the instances obtained after applying a reduction rule with the same node in the branching tree will make the analysis easier. If there are multiple instances $I_1, I_2, \dots, I_p$ associated with a node in a branching tree, then we use the term *first instance associated with that node* for the first instance that was associated with the node while executing the algorithm. In other words, first instance I associated with a node in a branching tree is such that all the other instances are obtained by making successive recursive calls starting from I . We also define the *last instance associated with a node* in the branching tree to be an instance I such that either no recursive call is made on I , or a branching step is executed when the algorithm is run on I .

For analyzing the running time of a branching algorithm, we associate a parameter $\mu(I)$ with each input instance I of the algorithm. If a branching step is executed on instance I , and there are l possible recursive calls and the parameter drops by a_i for the instance in the i^{th} recursive call, then the vector $(a_1, a_2, \dots, a_l)$ is called the *branching vector* for the branching step. The *branching number* of a branching vector $(a_1, a_2, \dots, a_l)$ is the unique positive root of the equation $\sum_{i=1}^l x^{-a_i} - 1 = 0$. If c is the largest branching number for all possible branching vectors in the branching algorithm and the measure always remains non-negative, then the number of nodes in that branching tree is at most $\mathcal{O}(c^k)$ where k is the initial parameter. Also, if the algorithm takes only polynomial time on every node of the branching tree, then the total running time of the algorithm will be $\mathcal{O}^*(c^k)$. For a more in-depth introduction to branching algorithms, one can see chapter 3 of [20].

For a (k, W) -VERTEX COVER instance (G, w, k, W) , we say that the instance is a Yes instance if there is a vertex cover S of G of size at most k and weight at most W , otherwise, we say that it is a No instance. Let (G', w, k', W') be the instance after applying a reduction rule to an instance (G, w, k, W) of (k, W) -VERTEX COVER, then we say that the reduction rule is correct when (G, w, k, W) is a Yes instance if and only if (G', w, k', W') is a Yes instance. Now, we define (a, b) -approximate solution and (a, b) -approximation algorithm for (k, W) -VERTEX COVER.

Definition 1. For an instance (G, w, k, W) of (k, W) -VERTEX COVER a vertex cover S of G is said to be an (a, b) -approximate solution if the size of S is at most ak and the weight of S , i.e. $\sum_{v \in S} w(v)$, is at most bW . For a given instance, (G, w, k, W) of (k, W) -VERTEX COVER an algorithm is called (a, b) -approximation algorithm for (k, W) -VERTEX COVER if the algorithm either returns an (a, b) -approximate solution or decides that (G, w, k, W) is a No instance.

We now state and prove a general lemma which will be used later.

Lemma 2. *If v is a vertex of a graph G with $d(v) \geq 1$, then an instance (G, w, k, W) of (k, W) -VERTEX COVER is a No instance if and only if both $(G - v, w, k - 1, W - w(v))$ and $(G - N[v], w, k - |N(v)|, W - w(N(v)))$ are No instances.*

Proof. Let v be a vertex of a graph G with $d(v) \geq 1$. First, let (G, w, k, W) be a No instance of (k, W) -VERTEX COVER. We claim both $(G - v, w, k - 1, W - w(v))$ and $(G - N[v], w, k - |N(v)|, W - w(N(v)))$ are No instances. If $(G - v, w, k - 1, W - w(v))$ is not a No instance, then there exists a vertex cover S' of $G - v$ such that size of S' is at most $k - 1$ and weight of S' is at most $W - w(v)$. Then just by adding v into S' , we can make $S = S' \cup \{v\}$, a vertex cover of G of size at most k with weight at most W . This contradicts that (G, w, k, W) is a No instance. Similarly, if $(G - N[v], w, k - |N(v)|, W - w(N(v)))$ is not a No instance, then there exists a vertex cover S'' of $G - N[v]$ such that size of S'' is at most $k - |N(v)|$ and weight at most $W - w(N(v))$. Now, since $S = S'' \cup N(v)$ will be a vertex cover of G of size at most k and weight at most W , we get a similar contradiction to (G, w, k, W) being a No instance.

Conversely, let both $(G - v, w, k - 1, W - w(v))$ and $(G - N[v], w, k - |N(v)|, W - w(N(v)))$ be No instances. We know that for any vertex cover S of G , either $v \in S$ or $N(v) \in S$. Now, if (G, w, k, W) is a Yes instance then let S be one such vertex cover of size at most k and weight at most W . So, either $S \setminus \{v\}$ is a vertex cover for $G - v$ of size at most $k - 1$ and weight at most $W - w(v)$, or $S \setminus N(v)$ is a vertex cover for $G - N[v]$ of size at most $k - |N(v)|$ and weight at most $W - w(N(v))$. This implies that either $(G - v, w, k - 1, W - w(v))$ or $(G - N[v], w, k - |N(v)|, W - w(N(v)))$ is a Yes instance which contradicts that both $(G - v, w, k - 1, W - w(v))$ and $(G - N[v], w, k - |N(v)|, W - w(N(v)))$ are No instances. Thus, we can say that (G, w, k, W) is a No instance. $\square$

3. A (2,2)-approximation algorithm for (k, W) -VERTEX COVER

In this section, we will show a (2,2)-approximation algorithm for (k, W) -VERTEX COVER.

Theorem 3. *There exists a polynomial-time (2,2)-approximation algorithm for (k, W) -VERTEX COVER.*

Proof. Let us look at the standard Integer Linear Program for the weighted vertex cover problem.

$$\text{minimize} \quad \sum_{j=1}^n w_j x_j \quad (1)$$

$$\text{subject to} \quad x_i + x_j \geq 1 \quad \forall (v_i, v_j) \in E(G) \quad (2)$$

$$x_j \in \{0, 1\} \quad \forall j \quad (3)$$

Since the above constraints do not take into account the size of the vertex cover obtained, we add the following constraint.

$$\sum_{j=1}^n x_j \leq k \quad (4)$$

We refer to the above integer linear program as ILP. It is easy to see that ILP is feasible if and only if the graph has a vertex cover of size at most k and the optimum value of ILP corresponds to the minimum weight of a vertex cover of size at most k . We relax the ILP in a standard way by replacing constraint (3) by $0 \leq x_j \leq 1$ for all j . We call this linear program LP, and we denote the value of an optimum solution for LP by $\text{OPT}(\text{LP})$.

Now, we do the standard rounding of an optimal solution of LP. Let us name this rounding algorithm as 'Polyapprx_algo'. In this algorithm, for a given input (G, w, k, W) and an optimum solution $\{x_i\}_{v_i \in V(G)}$ for its LP, we construct a solution $\{x_j^*\}_{v_j \in V(G)}$ for ILP as $x_j^* = 0$ if $x_j < 1/2$ and $x_j^* = 1$ if $x_j \geq 1/2$. Then we construct a set $S \subseteq V(G)$ as $S = \{v_i \mid x_i^* = 1\}$. This basically means that $v_i \in S$ if and only if $x_i \geq 1/2$. If the LP is not feasible or $\text{OPT}(\text{LP}) = \sum_{j=1}^n w_j x_j > W$, then we return No. Otherwise, we claim that S is a vertex cover of G of size at most $2k$ and weight at most $2W$.

If LP is infeasible, then clearly there is no vertex cover of size at most k , and the algorithm returns No correctly. Also, if $\text{OPT}(\text{LP}) > W$, we can correctly conclude that there is no vertex cover of G of size at most k and weight at most W . In the case that the algorithm returns a set S , it is easy to see that S is a vertex cover for G , as for every edge in G , at least one of the endpoints will have to be in S , otherwise constraint (2) corresponding to that edge will be violated in LP for the solution $\{x_i\}_{v_i \in V(G)}$. Also, we have that

$$w(S) = \sum_{v_i \in S} 1 \cdot w_i \leq \sum_{v_i \in S} 2 \cdot x_i w_i \leq 2\text{OPT}(\text{LP}) \leq 2W.$$

The last inequality is true because the algorithm returns a solution only if $\text{OPT}(\text{LP}) \leq W$, and the second inequality follows from the fact that for all $v_i \in S$, $2x_i \geq 1$. For the size of S , we have

$$|S| = \sum_{v_j \in S} x_j^* \leq \sum_{v_j \in S} 2 \cdot x_j \leq 2 \cdot \sum_{j=1}^n x_j \leq 2k.$$

Here, the last inequality follows from constraint (4) of LP, and the second inequality follows from the fact that $x_i^* \leq 2x_i$ for all $v_i \in V(G)$. Since an optimum solution of LP can be obtained in polynomial time, this finishes the proof of the theorem. $\square$

4. Parameterized algorithm for (k, W) -VERTEX COVER

In this section, we will show a parameterized algorithm for (k, W) -VERTEX COVER running in $\mathcal{O}^*(1.4656^k)$ time. For that, we first show that we can solve (k, W) -VERTEX COVER in $\text{poly}(n)$ time for any graph G with $\Delta(G) \leq 2$. We know that such a graph G is only a disjoint collection of paths and cycles. By utilizing this observation and a simple dynamic programming algorithms, we obtain a polynomial time algorithm for (k, W) -VERTEX COVER on such graphs. We deferred this algorithm and the proof of Theorem 4 to the appendix (Appendix A).

Theorem 4. (k, W) -VERTEX COVER can be solved in $\text{poly}(n)$ time for every graph G with $\Delta(G) \leq 2$.

Algorithm 1: Parameterized algorithm “Param_kwvc” for (k, W) -VERTEX COVER.

```

Input : An instance  $(G, w, k, W)$  of  $(k, W)$ -VERTEX COVER.
Output: A vertex cover  $S$  of size at most  $k$  with weight at most  $W$  if exists, No otherwise.
1 Algorithm Param_kwvc  $(G, w, k, W)$ 
2   if  $\min\{k, W\} < 0$  then
3     return No;
4   end
5   if  $E(G) = \emptyset$  then
6     return  $\emptyset$ ;
7   end
8   if  $\Delta(G) \leq 2$  then
9     return Deg2exact_algo  $(G, w, k, W)$ ;
10  else
11    Let  $u$  be a vertex of  $G$  such that  $d(u) \geq 3$ ;
12    // branching step
13    if Param_kwvc  $(G - u, w, k - 1, W - w(u))$  returns  $S$  then
14      return  $S \cup \{u\}$ ;
15    else if Param_kwvc  $(G - N[u], w, k - |N(u)|, W - w(N(u)))$  returns  $S$  then
16      return  $S \cup N(u)$ ;
17    else
18      return No;
19    end
20  end

```

Now, we describe the algorithm for (k, W) -VERTEX COVER. Given an instance (G, w, k, W) of (k, W) -VERTEX COVER, we first see if $\Delta(G) \leq 2$. If so, we solve (k, W) -VERTEX COVER in polynomial time using Theorem 4. Otherwise, we pick a vertex v of degree at least 3 from G and branch on including it or its neighbors in the solution. That is, we recursively solve the problems $(G - v, w, k - 1, W - w(v))$ and $(G - N[v], w, k - |N(v)|, W - w(N(v)))$, and return Yes if and only if at least one of the recursive calls returns Yes. A constructive version of this algorithm is presented in the form of pseudo-code in Algorithm 1.

Theorem 5. (k, W) -VERTEX COVER can be solved in time $\mathcal{O}^*(1.4656^k)$.

Proof. We use the above described algorithm to solve an instance (G, w, k, W) of (k, W) -VERTEX COVER. The correctness of the algorithm follows from the correctness of the algorithm Deg2exact_algo of Theorem 4 and from Lemma 2 that at least one of the recursive calls return Yes if and only if the original instance was a Yes instance. Since we are branching on a vertex of degree at least 3 and all the leaf nodes of the algorithm take polynomial time, this gives a branching vector of $(1, d)$, where $d \geq 3$, which results in running time being $\mathcal{O}^*(1.4656^k)$ as desired. $\square$

5. A simple $(1 + \epsilon, 2)$ -approximation algorithm

In this section, we give a $(1 + \epsilon, 2)$ -approximation algorithm for (k, W) -VERTEX COVER by combining Polyapprx_algo of Theorem 3 and Algorithm 1. As a result, the running time of the algorithm scales between the running times of Polyapprx_algo and Algorithm 1 with the value of ϵ . The algorithm either outputs No, which means that there is no vertex cover of size at most k with weight at most W , or outputs a vertex cover of size at most $(1 + \epsilon)k$ and weight at most $2W$.

Now we describe the algorithm. The algorithm checks at every step if the graph has maximum degree at most two, and if so, finds a vertex cover of weight at most W and size at most k , if exists, in polynomial time using Deg2exact_algo. Otherwise, the algorithm does a $(1, d)$ -branching with $d \geq 3$ by picking a vertex v of degree at least 3, and picking either v or $N(v)$ into the solution, same as Algorithm 1, while the size-budget is at least ϵk (the algorithm starts with a size-budget k and decreases it by i whenever it picks i vertices in the solution). When the size-budget drops below ϵk , the algorithm uses Polyapprx_algo of Theorem 3 to get a $(2, 2)$ -approximate solution of the remaining instance in polynomial time and combines the solution with the vertices picked in the branching steps to get a vertex cover of G . The algorithm returns No if every recursive call returns No. We describe the pseudo-code of the algorithm in Algorithm 2.

Algorithm 2: A $(1 + \epsilon, 2)$ -approximation algorithm “Sizeapprx_kwvc” for (k, W) -VERTEX COVER.

Input : An instance (G, w, k, W) of (k, W) -VERTEX COVER.
Output: A vertex cover S of size at most $(1 + \epsilon)k$ with weight at most $2W$ or returns No.

```

1 Algorithm Sizeapprx_kwvc( $G, w, k, W$ )
2   Fix  $x = \epsilon k$  as a global variable, the value of which never changes;
3   return Sizeapprx_kwvc_proc( $G, w, k, W$ );
4 Procedure Sizeapprx_kwvc_proc( $G, w, k, W$ )
5   if  $\min\{k, W\} < 0$  then
6     return No;
7   end
8   if  $E(G) = \emptyset$  then
9     return  $\emptyset$ ;
10  end
11  if  $\Delta(G) \leq 2$  then
12    return Deg2exact_algo( $G, w, k, W$ );
13  else if  $k \leq x$  then
14    return Polyapprx_algo( $G, w, k, W$ );
15  else
16    Let  $u$  be a vertex of  $G$  such that  $d(u) \geq 3$ ;
17    if Sizeapprx_kwvc_proc( $G - u, w, k - 1, W - w(u)$ ) returns  $S$  then
18      return  $S \cup \{u\}$ ;
19    else if Sizeapprx_kwvc_proc( $G - N[u], w, k - |N(u)|, W - w(N(u))$ ) returns  $S$  then
20      return  $S \cup N(u)$ ;
21    else
22      return No;
23    end
24  end

```

Now we argue the correctness and running time of this algorithm. Observe that if no recursive calls are made by the algorithm, either we solve the instance exactly in polynomial time using Theorem 4 or we apply Polyapprx_algo that runs in polynomial time (Theorem 3).

Theorem 6. For any $\epsilon \geq 0$, there exists a $(1 + \epsilon, 2)$ -approximation algorithm for (k, W) -VERTEX COVER with running time $\mathcal{O}^*(1.4656^{(1-\epsilon)k})$.

Proof. Let the input instance be (G, w, k, W) of (k, W) -VERTEX COVER, Algorithm 2 returns what Sizeapprx_kwvc_proc returns on (G, w, k, W) . We first argue the correctness. We first show that if the algorithm returns No for the instance $I = (G, w, k, W)$, then I is indeed a No instance of (k, W) -VERTEX COVER. When $k < 0$ or $W < 0$, Sizeapprx_kwvc_proc correctly returns No in line 6. If Sizeapprx_kwvc_proc does not make any recursive calls, then it returns No in line 12 or line 14 of Algorithm 2 after calling Deg2exact_algo or Polyapprx_algo respectively, and the statements Theorem 4 and Theorem 3 imply that the instance I is indeed a No instance. For the case when Sizeapprx_kwvc_proc does make a recursive call (reaches line 16), we apply induction on k . For the base case $k = 0$, let recursive calls are made to the instances $I_1 := (G - u, w, k - 1, W - w(u))$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)))$. Here both I_1 and I_2 are No instances from line 6 as both $k - 1$ and $k - |N(u)|$ are negative and thus by Lemma 2, I is a No instance. The induction hypothesis here is that for $k' < k$, if the algorithm returns No for the instance $I' = (G', w, k', W')$, then I' is indeed a No instance of (k, W) -VERTEX COVER. Now, the algorithm returns No for the instance (G, w, k, W) only if both the recursive calls on instances $I_1 := (G - u, w, k - 1, W - w(u))$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)))$ return No. Applying induction hypothesis, both I_1 and I_2 are No instances in such a case. Hence, due to Lemma 2, we know that I is a No instance as well.

To argue the correctness of the algorithm in the case it returns a set S , we first show that the set S is a vertex cover of G . On an input instance $I = (G, w, k, W)$, observe that while the algorithm might call Polyapprx_algo and Deg2exact_algo on multiple instances due to branching, if it returns a set S as a solution, then the set S is constructed starting from output of at most one of the calls to Polyapprx_algo and Deg2exact_algo (in particular, to any one of them), and then after the recursive calls adding vertices on which the branching step was performed. To show that S is indeed a vertex cover, let $S = S_1 \cup S_2$, where S_1 corresponds to the vertices included in the solution after the recursive calls (either u or $N(u)$ in lines 18 and 20), and $S_2 \subseteq S$ is the set returned by a call to Polyapprx_algo or Deg2exact_algo on the instance (G', w, k', W') as described above if such a call is made. If no such calls are made while constructing set S , then we take $S_2 = \emptyset$ and G' to be the empty graph. Clearly, S_2 is a vertex cover of G' , and for every edge uv not incident on S_2 , it must be the case that either u or v was picked in the solution by one of the branching steps in the path, and hence either $u \in S_1$ or $v \in S_1$. This shows that S is a vertex cover of G . To argue the size of S , we have that

$$|S| = |S_1| + |S_2| \leq k - k' + 2k' = k + k' \leq k + \epsilon k = (1 + \epsilon)k.$$

Also, for the weight of S , we have

$$w(S) = w(S_1) + w(S_2) \leq W - W' + 2W' = W + W' \leq 2W.$$

To see the running time, observe that if no recursive calls are made, then the algorithm takes polynomial time. If the input instance for the algorithm is (G, w, k, W) , then we define a measure $\mu(I)$ for an instance $I = (G', w, k', W')$ as $\mu(I) = k' - \epsilon k$. Initially, the

measure is $(1 - \epsilon)k$, and for each branching step, when the call is made to $(G - u, w, k - 1, W - w(u))$, it drops by 1, and when the call is made to $(G - N[u], w, k - |N(u)|, W - w(N(u)))$, it drops by d , where $d \geq 3$ is the degree of the vertex on which the branching is done. When the measure drops to zero, i.e., $k' = \epsilon k$, then we do not do any further branching and the algorithm takes only polynomial time. This gives a branching vector of $(1, d)$ with $d \geq 3$, and the branching number of the branching vector $(1, 3)$ is 1.4656. Since the measure is initially $(1 - \epsilon)k$, this bounds the total number of nodes in the branching tree by $\mathcal{O}(1.4656^{(1-\epsilon)k})$ and gives $\mathcal{O}^*(1.4656^{(1-\epsilon)k})$ as the running time for the algorithm. $\square$

6. A (1.5, 1.5)-approximation algorithm for (k, W) -VERTEX COVER

In this section, we give a (1.5, 1.5)-approximation algorithm for (k, W) -VERTEX COVER running in time $\mathcal{O}^*(2^{\frac{k}{2}})$. For that, we first make use of the following reduction rule.

Reduction Rule 1. For any instance (G, w, k, W) of (k, W) -VERTEX COVER, if u is a maximum weight vertex of G and $d(u) \leq 1$, then delete $N[u]$. The new instance will be $(G - N[u], w, k - |N(u)|, W - w(N(u)))$.

Lemma 7. Reduction Rule 1 is correct.

Proof. To prove Lemma 7, we have to show that for any instance (G, w, k, W) of (k, W) -VERTEX COVER, if u is a maximum weight vertex of G and $d(u) \leq 1$, then (G, w, k, W) has a vertex cover of size at most k and weight at most W if and only if $(G - N[u], w, k - |N(u)|, W - w(N(u)))$ has a vertex cover of size at most $k - |N(u)|$ and weight at most $W - w(N(u))$.

Let u be a maximum weight vertex in G and $d(u) = 1$. So, $|N(u)| = 1$ and let $N(u) = \{v\}$. Also, for any vertex cover S , we must have $S \cap \{u, v\} \neq \emptyset$. Now we want show that (G, w, k, W) has a vertex cover S of size at most k and weight at most W if and only if $(G - N[u], w, k - 1, W - w(N(u)))$ has a vertex cover S' of size at most $k - 1$ and weight at most $W - w(N(u))$.

Let us assume, (G, w, k, W) has a vertex cover S of size at most k and weight at most W . As said earlier, at least one of u and v must be in S . If $v \in S$, then $S' = S \setminus \{v\}$ will be a vertex cover for $(G - N[u], w, k - 1, W - w(N(u)))$ where size of S' is at most $k - 1$ and weight of S' is at most $W - w(N(u))$. Now, if $u \in S$, then $S_1 = (S \setminus \{u\}) \cup \{v\}$ will also be a vertex cover of G as uv is the only edge adjacent to u , and is covered by $v \in S_1$. The size of S_1 is the same as the size of S and $w(S_1) = w(S) - w(u) + w(v) \leq w(S)$ as u is a maximum weight vertex. Thus, S_1 is a vertex cover for (G, w, k, W) of size at most k , weight at most W , and $v \in S_1$. Then again we can say $S' = S_1 \setminus \{v\}$ is a vertex cover for $(G - N[u], w, k - 1, W - w(N(u)))$ where size of S' is at most $k - 1$ with weight at most $W - w(N(u))$. Conversely, if S' is a vertex cover $(G - N[u], w, k - 1, W - w(N(u)))$ of size at most $k - 1$ with weight at most $W - w(N(u))$, then $S = S' \cup \{v\}$ is a vertex cover of G as v was adjacent to all the edges of G which were not in $G - N[u]$. Also, we have that $|S| = 1 + |S'| \leq 1 + (k - 1) = k$ and $w(S) = w(S') + w(v) \leq W$.

Now, if any maximum weight vertex is of degree zero, then the lemma follows from the fact that no minimal vertex cover contains isolated vertices. $\square$

Now we describe the algorithm for (1.5, 1.5)-approximation.

Given an instance (G, w, k, W) , the algorithm first applies Reduction Rule 1 exhaustively, so that in the resulting graph, the degree of every maximum weight vertex is at least 2. Then the algorithm does branching on a maximum weight vertex till the depth $\frac{k}{2}$ in the branching tree. That is, branching is done on a node of the branching tree only if its distance from the root is less than $\frac{k}{2}$. Note that here we assume every internal node of the branching tree to have exactly two children, i.e., we assume that recursive call to the reduction rule is done at the same node of the branching tree in polynomial time. Any call to the algorithm returns No only if both the child nodes in the branching tree return No. At depth $\frac{k}{2}$, if the instance is $I = (G', w, k', W')$, then the algorithm applies the Polyapprx_algo of Theorem 3 to get a (2, 2)-approximate solution for I , combines it with the vertices picked in the solution while applying the reduction rule and branching, and outputs that as the solution.

We describe the pseudo-code of the algorithm in Algorithm 3. To keep track of the depth of the branching tree at which we are working and for analysis, we pass the algorithm additional parameters α, β . We also introduce a global variable x and fix its value $\frac{k}{2}$ if (G, w, k, W) was the original instance. For the branching steps, we increase α in the branch where the maximum weight vertex is taken into the solution, and β is increased in the branch when its neighborhood is taken into the solution. Observe that when the reduction rule is applied, α and β remain unchanged, and we remain at the same node of the branching tree. Hence, if $\alpha + \beta = \ell$, then the instance is associated with a node at level ℓ of the branching tree. We stop branching when $\alpha + \beta = x = \frac{k}{2}$ and use the Polyapprx_algo of Theorem 3. In the remainder of this section, we prove the correctness of Algorithm 3 and do the running time analysis.

Lemma 8. If Algorithm 3 returns No for the instance (G, w, k, W) , then (G, w, k, W) is indeed a No instance.

Proof. Given an instance $I = (G, w, k, W)$ of (k, W) -VERTEX COVER, Algorithm 3 returns what Approx1.5_kwvc_proc returns on $(G, w, k, W, \alpha, \beta)$ where $\alpha = \beta = 0$. Here we show that if the algorithm returns No for the instance $I = (G, w, k, W)$, then I is indeed a No instance of (k, W) -VERTEX COVER. When $k < 0$ or $W < 0$, Approx1.5_kwvc_proc correctly returns No in line 8. If Approx1.5_kwvc_proc does not make any recursive calls, then it returns No in line 14 of Algorithm 3 after calling Polyapprx_algo of

Algorithm 3: (1.5, 1.5)-approximation algorithm “Apprx1.5_kwvc” for (k, W) -VERTEX COVER.

Input : An instance (G, w, k, W) of (k, W) -VERTEX COVER.
Output: A vertex cover S of size at most $1.5k$ with weight at most $1.5W$ or returns No.

```

1 Algorithm Apprx1.5_kwvc( $G, w, k, W$ )
2   Fix  $x = \frac{k}{2}$  as a global variable, the value of which never changes;
3   Initialize,  $\alpha \leftarrow 0$  and  $\beta \leftarrow 0$ ;
4   return Apprx1.5_kwvc_proc( $G, w, k, W, \alpha, \beta$ );
5 /* We pass the parameters  $\alpha, \beta$  to the procedure with the instance  $(G, w, k, W)$  */
6 Procedure Apprx1.5_kwvc_proc( $G, w, k, W, \alpha, \beta$ )
7   if  $\min\{k, W\} < 0$  then
8     | return No;
9   end
10  if  $E(G) = \emptyset$  then
11    | return  $\emptyset$ ;
12  end
13  if  $\alpha + \beta = x$  then
14    | return Polyapprx_algo( $G, w, k, W$ );
15  end
16  Let  $u$  be a maximum weight vertex in  $G$ ;
17  if  $d(u) \leq 1$  then
18    // reduction rule
19    if Apprx1.5_kwvc_proc( $G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta$ ) returns  $S$  then
20      | return  $N(u) \cup S$ ;
21    else
22      | return No;
23    end
24  else
25    // branching step
26    if Apprx1.5_kwvc_proc( $G - u, w, k - 1, W - w(u), \alpha + 1, \beta$ ) returns  $S$  then
27      | return  $S \cup \{u\}$ ;
28    else if Apprx1.5_kwvc_proc( $G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1$ ) returns  $S$  then
29      | return  $S \cup N(u)$ ;
30    else
31      | return No;
32    end
33  end

```

Theorem 3, and the correctness of Polyapprx_algo implies that the instance I is indeed a No instance. When Apprx1.5_kwvc_proc makes a recursive call (when it reaches line 18 or line 25), we apply induction on k . For the base case $k = 0$, recursive call (in line 19) can be made to instance $I' = (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta)$ or recursive calls (in lines 26 and 28) can be made to the instances $I_1 := (G - u, w, k - 1, W - w(u), \alpha + 1, \beta)$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1)$. If recursive call is made to $I' = (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta)$, then I' is a No instance due to line 8 as $k - |N(u)|$ is negative and by the correctness of Reduction Rule 1, I is indeed a No instance. If recursive calls are made to the instances $I_1 := (G - u, w, k - 1, W - w(u), \alpha + 1, \beta)$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1)$, then both I_1 and I_2 are No instances from line 8 as both $k - 1$ and $k - |N(u)|$ are negative and thus by Lemma 2, I is a No instance. The induction hypothesis here is that for $k' < k$, if the algorithm returns No for the instance $I' = (G', w, k', W', \alpha', \beta')$, then I' is indeed a No instance of (k, W) -VERTEX COVER. If the algorithm returns No due to line 22, then the recursive call to the instance $I' = (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta)$ returns No. Then from induction hypothesis and the correctness of Reduction Rule 1, we can say I is indeed a No instance. Else, if Apprx1.5_kwvc_proc returns No from the recursive call when it reaches line 25, it returns No if both the recursive calls on instances $I_1 := (G - u, w, k - 1, W - w(u), \alpha + 1, \beta)$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1)$ return No. Applying induction hypothesis, both I_1 and I_2 are No instances in such a case. Hence, due to Lemma 2, we know that I is a No instance as well. $\square$

Lemma 9. If Algorithm 3 does not return No on an instance (G, w, k, W) of (k, W) -VERTEX COVER, then it returns a vertex cover S of G .

Proof. To show that Algorithm 3 returns a vertex cover of G , it suffices to show that Apprx1.5_kwvc_proc returns a vertex cover of G on $(G, w, k, W, \alpha, \beta)$. Similar to as described in the proof of Theorem 6, let $S = S_1 \uplus S_2$, where S_1 corresponds to the vertices included in the solution after the recursive calls (either u or $N(u)$ in lines 26 and 28 and $N(u)$ in line 19), and S_2 is the set returned by the call to Polyapprx_algo on the instance (G', w, k', W') if such a call is made. If no such calls are made while constructing S , then we take $S_2 = \emptyset$ and G' to be the empty graph. Clearly, S_2 is a vertex cover of G' , and for every edge uv not incident on S_2 , it must be the case that either u or v was picked in the solution by one of the branching steps in the path, and hence either $u \in S_1$ or $v \in S_1$. This shows that S is a vertex cover of G . $\square$

Next, we move to the following lemma, which says that at the leaf nodes of the branching tree, where we apply the Polyapprx_algo of Theorem 3, the value of the parameter is bounded.

Lemma 10. If `Approx1.5_kwvc_proc`, on an instance $(G', w, k', W', \alpha, \beta)$, calls `Polyapprx_algo` of Theorem 3 on the instance (G', w, k', W') in line 13 (without making any recursive call), then $k' \leq \alpha$.

Proof. Let (G', w, k', W') be an instance on which we apply `Polyapprx_algo` of Theorem 3. Then values of α and β corresponding to (G', w, k', W') must satisfy $\alpha + \beta = x = \frac{k}{2}$. Also, whenever values of α increase by one in the branching step, the size-budget drops by one and when the value of β increases by one in the branching step, the size-budget drops by at least two. Since k' is the remaining size-budget, we have that $k' \leq k - (\alpha + 2\beta) = k - \frac{k}{2} - \beta = \alpha$. $\square$

Now, before we show the size and weight bounds on the vertex set returned by the algorithm, we define some notions that will help us with the analysis. We classify the instances on which we call the `Polyapprx_algo` in line 13 into two types.

Let (G, w, k, W) be the original instance passed to `Approx1.5_kwvc_proc`. We say that an instance (G', w, k', W') on which `Polyapprx_algo` is called in line 13 is Type-I, if $W' \leq W/2$ and is Type-II otherwise. We also partition the set S returned by Algorithm 3 into two parts. Let S_1 be the set of vertices in the returned vertex cover picked during the branching or the application of the reduction rule and let $S_2 = S \setminus S_1$. It can happen that $S_2 = \emptyset$ when the algorithm does not call `Polyapprx_algo` while constructing S . Otherwise, S_2 comes from the `Polyapprx_algo` of Theorem 3. The next two lemmas give the size and weight bounds of the returned solution.

Lemma 11. Let Algorithm 3 return a set $S = S_1 \uplus S_2$ for an input instance (G, w, k, W) , where S_1 and S_2 are as defined above. If S_2 is obtained after calling `Polyapprx_algo` on a Type-I instance (G_2, w, k_2, W_2) in line 13, then $|S| \leq 1.5k$ and $w(S) \leq 1.5W$.

Proof. Since for every vertex included in the solution we decrease the size-budget by one, we have that $|S_1| = k - k_2$, and by the Theorem 3, we have that $|S_2| \leq 2k_2$. Also, since α and β remain non-negative throughout and $\alpha + \beta \leq \frac{k}{2}$, we have that $\alpha \leq \frac{k}{2}$. Again, from Lemma 10, we have $k_2 \leq \alpha$, which gives us

$$|S| = |S_1| + |S_2| \leq k - k_2 + 2k_2 = k + k_2 \leq k + \alpha \leq k + 0.5k = 1.5k.$$

Similarly, for the weight we have that $w(S_1) = W - W_2$, and $w(S_2) \leq 2W_2$. This gives us

$$w(S) = w(S_1) + w(S_2) \leq W - W_2 + 2W_2 = W + W_2 \leq 1.5W.$$

Here, the last inequality is due to $W_2 \leq \frac{W}{2}$ as (G_2, w, k_2, W_2) is a Type-I instance. $\square$

Lemma 12. Let Algorithm 3 return a set $S = S_1 \uplus S_2$ for an input instance (G, w, k, W) , where S_1 and S_2 are as defined above. If S_2 is obtained after calling `Polyapprx_algo` on a Type-II instance (G_3, w, k_3, W_3) in line 13, then $|S| \leq 1.5k$ and $w(S) \leq 1.5W$.

Proof. For a Type-II instance (G_3, w, k_3, W_3) , we have $k_3 \leq \alpha$ due to Lemma 10 and $W_3 > \frac{W}{2}$. Also, whenever we increase α by one in the branching, we include a maximum weight vertex available at that time in the set S_1 . We also have that $w(S_1) = W - W_3 < \frac{W}{2}$. So, the average weight of those α high-weight vertices is less than $\frac{W}{2\alpha}$. Since we branch on a maximum weight vertex available at that time, there should not be any $v \in S_2$ such that $w(v) \geq \frac{W}{2\alpha}$. So we must have $w(v) < \frac{W}{2\alpha}$ for all $v \in S_2$. Since S_2 is the $(2, 2)$ -approximate solution for (G_3, w, k_3, W_3) , we have

$$|S_2| \leq 2k_3 \leq 2\alpha,$$

and

$$w(S_2) < |S_2| \frac{W}{2\alpha} \leq 2\alpha \frac{W}{2\alpha} = W$$

as $w(v) < \frac{W}{2\alpha}$, for all $v \in S_2$. From here, we can say

$$|S| = |S_1| + |S_2| \leq k - k_3 + 2k_3 = k + k_3 \leq k + \alpha \leq k + \frac{k}{2} = 1.5k.$$

Also,

$$w(S) = w(S_1) + w(S_2) \leq W - W_3 + W = 2W - W_3 < 1.5W.$$

The last inequality holds because (G_3, w, k_3, W_3) is a Type-II instance and hence $W_3 > \frac{W}{2}$. $\square$

Theorem 13. There exists a $(1.5, 1.5)$ -approximation algorithm for (k, W) -VERTEX COVER with running time $\mathcal{O}^*(2^{\frac{k}{2}})$.

Proof. Given an instance (G, w, k, W) of (k, W) -VERTEX COVER we call Algorithm 3 on it. In Lemmas 8-12, we have already argued that if the algorithm returns No, then (G, w, k, W) is indeed a No instance, and if the algorithm returns a set S after calling `Polyapprx_algo`, then S is a vertex cover of G with $|S| \leq 1.5k$ and $w(S) \leq 1.5W$. If the algorithm does not call `Polyapprx_algo` while

constructing set S , then it should come due to line 11 of Algorithm 3 for some instance (G', w, k', W') with $E(G') = \emptyset$ and both k', W' are non-negative. In this case, all the vertices of the set S will be picked from the branching step only. We can easily observe that S is a vertex cover as if there is any edge left to cover then $E(G')$ must be non-empty. Again, since both k' and W' are non-negative when $E(G')$ becomes empty, we will have $|S| = k - k' \leq k$ and $w(S) = W - W' \leq W$.

So all we need to show is that Algorithm 3 takes time $\mathcal{O}^*(2^{\frac{k}{2}})$ on an instance (G, w, k, W) . Our algorithm branches till depth $\frac{k}{2}$ in the branching tree, so the branching tree has $\mathcal{O}(2^{\frac{k}{2}})$ nodes, and on each of the nodes we spend at most polynomial time. Hence the running time follows. $\square$

7. A $(2 - \delta, 2 - \delta)$ -approximation algorithm for $\delta < 0.5$

In this section, we will present a $(2 - \delta, 2 - \delta)$ -approximation algorithm for (k, W) -VERTEX COVER for any $\delta < 0.5$. We first give a description of this algorithm, which we call Algorithm 4.

Given an instance (G, w, k, W) of (k, W) -VERTEX COVER to Algorithm 4, it introduces two global variable x and Y . Algorithm fixes $x := k - \delta k$ and $Y := W - \delta W$. We also initialize $\alpha = 0$, $\beta = 0$ and pass these additional parameters with the input instance. Then the algorithm calls `Approxgen_kwvc_proc` on instance $(G, w, k, W, \alpha, \beta)$ and returns the solution returned by `Approxgen_kwvc_proc` (x and Y are never changed by `Approxgen_kwvc_proc`). `Approxgen_kwvc_proc` first applies Reduction Rule 1 exhaustively, so that in the resulting graph, every maximum weight vertex has degree at least 2. Then it branches on a maximum weight vertex u , calling itself on $(G - u, k - 1, W - w(u), \alpha + 1, \beta)$ and $(G - N[u], k - |N(u)|, W - w(N(u)), \alpha, \beta + 1)$ recursively. `Approxgen_kwvc_proc` is very similar to `Approx1.5_kwvc_proc`. For the branching steps, we increase α in the branch where the maximum weight vertex is taken into the solution, and β is increased in the branch when its neighborhood is taken into the solution. The main difference between `Approx1.5_kwvc_proc` and `Approxgen_kwvc_proc` is their stopping criteria. `Approxgen_kwvc_proc` stops branching on an instance $(G', w, k', W', \alpha', \beta')$ in the branching tree when either i) G' becomes empty, or ii) $k' \leq x$ and either $W' \leq Y$ or there is no vertex of weight $\frac{1}{k'}$ in G' . Note that here we assume every internal node of the branching tree to have exactly two children, i.e., we assume that the recursive call after the reduction rule is applied is made at the same node of the branching tree in polynomial time. Any recursive call to `Approxgen_kwvc_proc` returns No only if both the child nodes in the branching tree return No. If we stop branching on an instance $(G', w, k', W', \alpha', \beta')$ and G' is empty, then `Approxgen_kwvc_proc` returns the vertices picked in the solution during branching and while applying Reduction Rule 1 to obtain $(G', w, k', W', \alpha', \beta')$. Otherwise, `Approxgen_kwvc_proc` applies the `Polyapprx_algo` of Theorem 3 (that is, $(2, 2)$ -approximation algorithm) to the instance (G', w, k', W') to get a $(2, 2)$ -approximate solution for (G', w, k', W') and combines it with the vertices added to the solution while applying the reduction rule and branching, and outputs that as a solution. The pseudo-code in Algorithm 4 describes this algorithm more formally.

The following two lemmas give the correctness of Algorithm 4.

Lemma 14. *If Algorithm 4 returns No for the instance (G, w, k, W) , then (G, w, k, W) is indeed a No instance.*

Proof. Given an instance $I = (G, w, k, W)$ of (k, W) -VERTEX COVER, Algorithm 4 returns what `Approxgen_kwvc_proc` returns on $(G, w, k, W, \alpha, \beta)$, on $\alpha = \beta = 0$. If the algorithm returns No for the instance $I = (G, w, k, W)$, then we argue that I is indeed a No instance of (k, W) -VERTEX COVER. When $k < 0$ or $W < 0$, `Approxgen_kwvc_proc` correctly returns No in line 8. If `Approxgen_kwvc_proc` does not make any recursive calls, then it returns No in line 11 and line 17 of Algorithm 3 after calling `Polyapprx_algo` of Theorem 3, and the correctness of `Polyapprx_algo` implies that the instance I is indeed a No instance. In the case, when `Approxgen_kwvc_proc` does make a recursive call (when it reaches line 21 or line 28), we apply induction on k . For the base case $k = 0$, recursive call (in line 22) can be made to instance $I' = (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta)$ or recursive calls (in line 29 and 31) can be made to the instances $I_1 := (G - u, w, k - 1, W - w(u), \alpha + 1, \beta)$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1)$. If recursive call is made to $I' = (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta)$, then I' is a No instance due to line 8 as $k - |N(u)|$ is negative and by the correctness of Reduction Rule 1, I is indeed a No instance. If recursive calls are made to the instances $I_1 := (G - u, w, k - 1, W - w(u), \alpha + 1, \beta)$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1)$, then both I_1 and I_2 are No instances from line 8 as both $k - 1$ and $k - |N(u)|$ are negative and thus by Lemma 2, I is a No instance. The induction hypothesis here is that for $k' < k$, if the algorithm returns No for the instance $I' = (G', w, k', W', \alpha', \beta')$, then I' is indeed a No instance of (k, W) -VERTEX COVER. If the algorithm returns No due to line 25, then the recursive call to the instance $I' = (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta)$ returns No. Then from induction hypothesis and the correctness of Reduction Rule 1, we can say I is indeed a No instance. Else, if Algorithm 3 returns No from the recursive call when it reaches line 28, it returns No if both the recursive calls on instances $I_1 := (G - u, w, k - 1, W - w(u), \alpha + 1, \beta)$ and $I_2 := (G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1)$ return No. Applying induction hypothesis, both I_1 and I_2 are No instances in such a case. Hence, due to Lemma 2, we know that I is a No instance as well. $\square$

Lemma 15. *If Algorithm 4 does not return No on an instance (G, w, k, W) of (k, W) -VERTEX COVER, then it returns a vertex cover S of G such that $|S| \leq (2 - \delta)k$ and $w(S) \leq (2 - \delta)W$.*

Proof. To show that Algorithm 4 returns a vertex cover of G , it suffices to show that `Approxgen_kwvc_proc` returns a vertex cover of G on $(G, w, k, W, \alpha, \beta)$. Let $S = S_1 \uplus S_2$, where S_1 corresponds to the vertices included in the solution after the recursive calls (either u or $N(u)$ in lines 30 and 32 and $N(u)$ in line 23), and $S_2 \subseteq S$ is the set returned by the call to `Polyapprx_algo` on the instance

Algorithm 4: $(2 - \delta, 2 - \delta)$ -approximation algorithm “Apprxgen_kwvc” for (k, W) -VERTEX COVER.

Input : An instance (G, w, k, W) of (k, W) -VERTEX COVER and $0 < \delta < 0.5$.
Output: A vertex cover S of G such that $|S| \leq (2 - \delta)k$ and $w(S) \leq (2 - \delta)W$ or returns No.

```

1 Algorithm Apprxgen_kwvc( $G, w, k, W$ )
2   Fix  $x = (1 - \delta)k$  and  $Y = (1 - \delta)W$  as global variables, the value of which never changes;
3   Initialize,  $\alpha \leftarrow 0$  and  $\beta \leftarrow 0$ ;
4   /* We pass parameters  $\alpha, \beta$  to the procedure with the instance  $(G, w, k, W)$  */
5   return Approxgen_kwvc_proc( $G, w, k, W, \alpha, \beta$ );
6 Procedure Approxgen_kwvc_proc( $G, w, k, W, \alpha, \beta$ )
7   if  $\min\{W, k\} < 0$  then
8     | return No;
9   end
10  if  $W \leq Y$  and  $k \leq x$  then
11    | return Polyapprx_algo( $G, w, k, W$ );
12  end
13  if  $E(G) = \emptyset$  then
14    | return  $\emptyset$ ;
15  end
16  if  $w(v) < \frac{Y}{k} \forall v \in V(G)$  and  $k \leq x$  then
17    | return Polyapprx_algo( $G, w, k, W$ );
18  end
19  Let  $u$  be a maximum weight vertex in  $G$ ;
20  if  $d(u) \leq 1$  then
21    // reduction rule
22    if Approxgen_kwvc_proc( $G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta$ ) returns  $S$  then
23      | return  $N(u) \cup S$ ;
24    else
25      | return No;
26    end
27  else
28    // branching step
29    if Approxgen_kwvc_proc( $G - u, w, k - 1, W - w(u), \alpha + 1, \beta$ ) returns  $S$  then
30      | return  $S \cup \{u\}$ ;
31    else if Approxgen_kwvc_proc( $G - N[u], w, k - |N(u)|, W - w(N(u)), \alpha, \beta + 1$ ) returns  $S$  then
32      | return  $S \cup N(u)$ ;
33    else
34      | return No;
35    end
36  end

```

(G', w, k', W') if such a call is made. If no such calls are made, then we take $S_2 = \emptyset$ and G' to be the empty graph. Clearly, S_2 is a vertex cover of G' , and for every edge uv not incident on S_2 , it must be the case that either u or v was picked in the solution by one of the branching steps in the path, and hence either $u \in S_1$ or $v \in S_1$. This shows that S is a vertex cover of G .

Now, if Algorithm 4 returns a vertex cover S of G on (G, w, k, W) , we show $|S| \leq (2 - \delta)k$ and $w(S) \leq (2 - \delta)W$. Like before, we partition the set S returned by Algorithm 4 into two parts. Let S_1 be the set of vertices in the returned vertex cover that were picked during the branching or during the application of the reduction rule and let $S_2 = S \setminus S_1$. It can happen that $S_2 = \emptyset$ when the algorithm does not call Polyapprx_algo while constructing S and then the solution should come due to line 14 of Algorithm 4 for some instance (G', w, k', W') with $E(G') = \emptyset$ and both k', W' are non-negative. Since both k' and W' are non-negative when $E(G')$ becomes empty, we will have $|S| = k - k' \leq k$ and $w(S) = W - W' \leq W$. If algorithm calls Polyapprx_algo while constructing S , then S_2 comes from the Polyapprx_algo of Theorem 3 (lines 11 and 17) for some instance $(G' = G[V \setminus S_1], w, k' = k - |S_1|, W' = W - w(S_1))$.

If S_2 comes from line 17, then $k' \leq x = (1 - \delta)k$, $W' > Y = (1 - \delta)W$ and $w(z) < \frac{Y}{k'}, \forall z \in V(G')$. As S_2 is a $(2, 2)$ -approximate solution for (G', w, k', W') , $|S_2| \leq 2k'$ and $w(S_2) \leq 2k' \frac{Y}{k'} = 2Y = 2(1 - \delta)W$. Now we have

$$|S| = |S_1| + |S_2| \leq k - k' + 2k' = k + k' \leq k + k - \delta k = (2 - \delta)k,$$

and

$$w(S) = w(S_1) + w(S_2) \leq W - W' + 2(1 - \delta)W \leq \delta W + 2(1 - \delta)W = (2 - \delta)W.$$

If S_2 comes from line 11, then $k' \leq x = (1 - \delta)k$ and $W' \leq Y = (1 - \delta)W$. As S_2 is a $(2, 2)$ -approximate solution for (G', w, k', W') , $|S_2| \leq 2k'$ and $w(S_2) \leq 2W'$. Now,

$$|S| = |S_1| + |S_2| \leq k - k' + 2k' = k + k' \leq k + k - \delta k = (2 - \delta)k$$

and

$$w(S) = w(S_1) + w(S_2) \leq W - W' + 2W' = W + W' \leq W + (1 - \delta)W = (2 - \delta)W.$$

Hence the proof of the lemma. $\square$

Now, before we argue the running time of the algorithm, we first define some notations and prove some lemmas about the types of instances that can be generated during the course of the algorithm.

Due to Reduction Rule 1, the branching we are doing in `Approxgen_kwvc_proc` is a $(1, d)$ -branching (where $d \geq 2$) but here we have additional stopping criteria for any instance $(G', w, k', W', \alpha', \beta')$ with $E(G') \neq \emptyset$. The criteria are i) $W' \leq Y$ together with $k' \leq x$, or ii) there is no vertex in G' of weight $\frac{Y}{k'}$ together with $k' \leq x$. Here our main goal is to get a good upper bound on the number of nodes in the branching tree.

For any instance $(G', w, k', W', \alpha', \beta')$ of `Approxgen_kwvc_proc`, we call it one of Type-I, Type-II, Type-III, and Type-IV depending on values of k' and W' . In the case $E(G') = \emptyset$, `Approxgen_kwvc_proc` correctly returns a vertex cover of size at most k with weight at most W in polynomial time, so we assume that $E(G') \neq \emptyset$. Now, let us define above mentioned four different types of instances.

Type-I: $(G_1, w, k_1, W_1, \alpha_1, \beta_1)$ is a Type-I instance if $k_1 \leq x$ and $W_1 \leq Y$.

Type-II: $(G_2, w, k_2, W_2, \alpha_2, \beta_2)$ is a Type-II instance if $k_2 \leq x$, $W_2 > Y$, and there is no vertex of weight $\frac{Y}{k_2}$ in G_2 .

Type-III: $(G_3, w, k_3, W_3, \alpha_3, \beta_3)$ is a Type-III instance if $k_3 \leq x$, $W_3 > Y$, and there exists at least one vertex of weight at least $\frac{Y}{k_3}$ in G_3 .

Type-IV: All other instances $((G_4, w, k_4, W_4, \alpha_4, \beta_4))$ such that $k_4 > x$.

A node in the branching tree of `Approxgen_kwvc_proc` is said to be of Type-I, Type-II, Type-III, or Type-IV if the first instance associated with that node is a Type-I, Type-II, Type-III, or Type-IV instance respectively.

If any instance is of Type-I or Type-II, then `Approxgen_kwvc_proc` immediately applies the `Polyapprx_algo` of Theorem 3 to those instances. So, we continue branching on an instance when the instance is of Type-III or Type-IV. Also, note that the Type-IV nodes can be in the first δk layers of the branching tree only. Otherwise, after δk layers, the size-budget (k) will decrease by at least δk , and thus the node cannot be of Type-IV anymore. For this reason, the total number of Type-IV nodes is bounded by $2^{\delta k}$. Let us assign two values α and β to every node of the branching tree of `Approxgen_kwvc_proc` where α, β come from instance $(G, w, k, W, \alpha, \beta)$ associated to that node. Note that values of α, β of every associated instance on a node will be the same because any associated instance (except the first associated instance) on a node is obtained by applying Reduction Rule 1 to some instance associated on the same node and when Reduction Rule 1 is applied, α, β remain unchanged. Now let us prove the following lemma.

Lemma 16. *If $(G', w, k', W', \alpha, \beta)$ is an instance of Type-III, then α corresponding to the instance $(G', w, k', W', \alpha, \beta)$ is at most $k'(\frac{\delta}{1-\delta})$.*

Proof. For the sake of contradiction, assume that $(G', w, k', W', \alpha, \beta)$ is an instance of Type-III and α is more than $k'(\frac{\delta}{1-\delta})$. Since $(G', w, k', W', \alpha, \beta)$ is an instance of Type-III, we have, $W' > W - \delta W$ and there exists a vertex of weight at least $\frac{W - \delta W}{k'}$ in G' . Since every time we branch on a maximum weight vertex available, all the vertices on which we have already branched on to obtain the instance $(G', w, k', W', \alpha, \beta)$ are of weight at least $\frac{W - \delta W}{k'}$. Also, note that at least α vertices of weight at least $\frac{W - \delta W}{k'}$ are picked in the solution. So, here we have

$$W' \leq W - \left(\alpha \cdot \frac{W - \delta W}{k'} \right) \leq W - \left(k' \cdot \frac{\delta}{1-\delta} \cdot \frac{W - \delta W}{k'} \right) \leq W - \delta W = Y,$$

which is a contradiction to $(G', w, k', W', \alpha, \beta)$ being an instance of Type-III. $\square$

Next we want to compute the number of Type-III nodes at each layer of the branching tree. It is clear that upto δk layers, there can be a total of $\mathcal{O}(2^{\delta k})$ Type-III or Type-IV nodes. Also, note that for any node at any layer j , we have $\alpha + \beta = j$ corresponding to that node. Now we prove the following lemmas which determine the number of Type-III nodes at layers beyond the first δk layers.

Lemma 17. *There are no Type-III nodes at layer $\frac{k}{2}$.*

Proof. Let $(G', w, k', W', \alpha, \beta)$ be the first instance associated to a node at layer $\frac{k}{2}$. Here, $k' \leq k - (\alpha + 2\beta) = \frac{k}{2} - \beta = \alpha$ (as at layer $\frac{k}{2}$, $\alpha + \beta = \frac{k}{2}$). Thus, $k' \leq \alpha \leq \frac{k}{2} \leq (1 - \delta)k$, $\forall \delta < 0.5$. Now, if $(G', w, k', W', \alpha, \beta)$ is a Type-III instance, then we must have $W' > Y = (1 - \delta)W$ and there is at least one vertex with weight $\frac{Y}{k'} \geq \frac{W}{2\alpha}$ (as $Y = (1 - \delta)W > \frac{W}{2} \forall \delta < 0.5$ and $k' \leq \alpha$) in G' . So, all the vertices we have branched on to obtain the instance $(G', w, k', W', \alpha, \beta)$ are of weight at least $\frac{W}{2\alpha}$, and there are α such vertices. This implies, $W' \leq W - \alpha \frac{W}{2\alpha} = \frac{W}{2} < (1 - \delta)W = Y$, $\forall \delta < 0.5$, which contradicts that $(G', w, k', W', \alpha, \beta)$ is a Type-III instance. $\square$

We have already seen that there are no Type-IV nodes at layer $\frac{k}{2}$, and the number of Type-IV nodes is bounded by $2^{\delta k}$. Also, since no branching is done for Type-I or Type-II instances, the branching tree does not have any nodes beyond layer $\frac{k}{2}$. Observe that $\delta k + i = \frac{k}{2}$ for $i = \frac{\delta k(1-2\delta)}{2\delta}$, and hence we only need to bound the number of Type-III nodes for layers $\delta k + i$ where $i \leq \frac{\delta k(1-2\delta)}{2\delta}$. This is what we do in the following lemma.

Lemma 18. *At layer $\delta k + i$, the number of Type-III nodes can be at most $2^{\delta k + i}$ when $0 \leq i \leq \frac{\delta k(1-2\delta)}{1+2\delta}$ and bounded by $\mathcal{O}^*\left(\frac{\delta k + i}{\delta k - \frac{2i\delta}{1-2\delta}}\right)$ when $\frac{\delta k(1-2\delta)}{1+2\delta} \leq i \leq \frac{\delta k(1-2\delta)}{2\delta}$.*

Proof. For any Type-III node at layer $\delta k + i$, we must have $\alpha + \beta = \delta k + i$. So, for any Type-III instance $(G', w, k', W', \alpha, \beta)$ associated to a node at layer $\delta k + i$, we have that

$$k' \leq k - (\alpha + 2\beta) = k - (\alpha + \beta) - \beta = k - 2(\delta k + i) + \alpha.$$

Since $(G', w, k', W', \alpha, \beta)$ is an instance of Type-III, we have that $W' > W - \delta W$ and there exists a vertex of weight at least $\frac{W - \delta W}{k'}$ in G' . Now, by Lemma 16,

$$\begin{aligned} \alpha &\leq k' \frac{\delta}{1 - \delta} \implies \alpha \leq (k - 2(\delta k + i) + \alpha) \frac{\delta}{1 - \delta} \\ \implies (1 - \delta)\alpha - \delta\alpha &\leq (1 - 2\delta)\delta k - 2i\delta \\ \implies (1 - 2\delta)\alpha &\leq (1 - 2\delta)\delta k - 2i\delta \implies \alpha \leq \delta k - \frac{2i\delta}{1 - 2\delta}. \end{aligned}$$

From here we can say that the number of nodes at layer $\delta k + i$ with $\alpha \leq \delta k - \frac{2i\delta}{1 - 2\delta}$ becomes an upper bound on the number of Type-III nodes at that layer.

Notice that for any node t of the branching tree at layer l , the root to t path in the branching tree can be expressed as a unique sequence of L and R , where j th place of the sequence will be L or R depending on α or β increases on the recursion to reach the node at j th layer in the root to t path from its parent node in the branching tree respectively. Also, the length of the sequence will be exactly l which is the sum of the number of L and R in that particular sequence. So, the number of nodes with $\alpha = i$ at any layer l can be at most the number of sequences of L and R of length l having exactly i number of L . Thus the number of nodes with $\alpha = i$ at any layer l can be at most $\binom{l}{i}$.

We know that at layer $\delta k + i$, $\alpha + \beta$ is exactly $\delta k + i$. Thus the number of nodes with $\alpha = 1$ at layer $\delta k + i$ can be at most $\binom{\delta k + i}{1}$, the number of nodes with $\alpha = 2$ at layer $\delta k + i$ can be at most $\binom{\delta k + i}{2}$ and so on. Hence, the number of nodes with $\alpha = \delta k - \frac{2i\delta}{1 - 2\delta}$ at layer $\delta k + i$ can be at most $\binom{\delta k + i}{\delta k - \frac{2i\delta}{1 - 2\delta}}$.

The number of nodes at layer $\delta k + i$ with $\alpha \leq \delta k - \frac{2i\delta}{1 - 2\delta}$ (and hence the number of Type-III nodes in that layer) is at most

$$\binom{\delta k + i}{1} + \binom{\delta k + i}{2} + \dots + \binom{\delta k + i}{\delta k - \frac{2i\delta}{1 - 2\delta}}.$$

At $i = \frac{\delta k(1 - 2\delta)}{1 + 2\delta}$, the last term in the above summation becomes $\binom{\delta k + i}{\frac{\delta k + i}{2}}$. So, for $\frac{\delta k(1 - 2\delta)}{1 + 2\delta} \leq i \leq \frac{\delta k(1 - 2\delta)}{2\delta}$, the last term is the largest, and there are at most $\frac{k}{2}$ such terms, hence the summation is bounded by $\mathcal{O}^*\left(\binom{\delta k + i}{\delta k - \frac{2i\delta}{1 - 2\delta}}\right)$. Also, it is easy to see that the summation is bounded by $2^{\delta k + i}$ when $0 \leq i \leq \frac{\delta k(1 - 2\delta)}{1 + 2\delta}$, since each term in the summation can be thought of as the number of subsets of a certain size, of a set containing $(\delta k + i)$ elements. Hence the proof of the lemma. $\square$

Now we are ready to prove the lemma about the running time of Algorithm 4.

Lemma 19. Algorithm 4 runs in time $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1 - 2\delta)}{1 + 2\delta}}^{\frac{\delta k(1 - 2\delta)}{2\delta}} \binom{\delta k + i}{\delta k - \frac{2i\delta}{1 - 2\delta}}\right)$.

Proof. We need to show that the total number of nodes in the branching tree is bounded by $\mathcal{O}^*(\Gamma)$, where $\Gamma := \left(\sum_{i=\frac{\delta k(1 - 2\delta)}{1 + 2\delta}}^{\frac{\delta k(1 - 2\delta)}{2\delta}} \binom{\delta k + i}{\delta k - \frac{2i\delta}{1 - 2\delta}}\right)$. For that, first observe that the number of Type-I and Type-II nodes can be at most twice the number of Type-III and Type-IV nodes in the branching tree, as the parent of every Type-I and Type-II node is either a Type-III node or a Type-IV node and the branching tree is a binary tree. So all we need to do is to bound the number of Type-III and Type-IV nodes by $\mathcal{O}^*(\Gamma)$.

The first term in the expression of Γ is $\binom{\delta k + i}{\delta k - \frac{2i\delta}{1 - 2\delta}}$ for $i = \frac{\delta k(1 - 2\delta)}{1 + 2\delta}$, which turns out to be $\binom{2\delta k}{\frac{\delta k}{1 + 2\delta}}$. We know that the central binomial coefficient, $\binom{n}{n/2} \geq 2^n / (n + 1)$. This gives us

$$\binom{\frac{2\delta k}{1 + 2\delta}}{\frac{\delta k}{1 + 2\delta}} \geq \frac{2^{\frac{2\delta k}{1 + 2\delta}}}{\frac{2\delta k}{1 + 2\delta} + 1}.$$

This implies that

$$2^{\frac{2\delta k}{1 + 2\delta}} = \mathcal{O}^*\left(\binom{\frac{2\delta k}{1 + 2\delta}}{\frac{\delta k}{1 + 2\delta}}\right) = \mathcal{O}^*(\Gamma).$$

Since the number of Type-III and Type-IV nodes in the first δk layers is bounded by $2^{\delta k}$, which is $\mathcal{O}(2^{\frac{2\delta k}{1+2\delta}})$ for $\delta < 0.5$, we get that the number of Type-III and Type-IV nodes in the first δk layers is bounded by $\mathcal{O}^*(\Gamma)$. We have already seen that there are no Type-IV nodes beyond the first δk layers, so this bounds the total number of Type-IV nodes by $\mathcal{O}^*(\Gamma)$.

Number of Type-III nodes at layers $\delta k + i$, where $\frac{\delta k(1-2\delta)}{1+2\delta} \leq i \leq \frac{\delta k(1-2\delta)}{2\delta}$, is already bounded by $\mathcal{O}^*(\Gamma)$ due to Lemma 18. So all we are left to bound now is the number of Type-III nodes at layers $\delta k + i$, where $0 \leq i \leq \frac{\delta k(1-2\delta)}{1+2\delta}$. From Lemma 18, each such layer contains at most $2^{\delta k+i}$ nodes of Type-III. This attains the maximum value of $2^{\frac{2\delta k}{1+2\delta}}$ at $i = \frac{\delta k(1-2\delta)}{2\delta}$, which we have already seen to be bounded by $\mathcal{O}^*(\Gamma)$. Since there are at most $\frac{k}{2}$ such layers, the total number of Type-III nodes, and hence the total number of nodes in the branching tree is bounded by $\mathcal{O}^*(\Gamma)$.

It is also clear from the algorithm that we are either spending only polynomial time at every node of the branching tree or applying Polyapprx_algo which also runs in polynomial time. Hence the total time taken by the algorithm will be at most $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right)$. This completes the proof of the Lemma 19. $\square$

The proof of the following theorem follows from Lemma 14, Lemma 15 and Lemma 19.

Theorem 20. *There exists a $(2-\delta, 2-\delta)$ -approximation algorithm for (k, W) -VERTEX COVER with running time $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right)$.*

7.1. Computation of the running time

Let us consider a function, $f_\delta : \left[\frac{\delta(1-2\delta)}{1+2\delta}, \frac{\delta(1-2\delta)}{2\delta}\right] \rightarrow \mathbb{R}$, such that

$$f_\delta(x) = \left[\left(\frac{(1-2\delta)(\delta+x)}{\delta-2\delta^2-2x\delta} \right)^{\left(\frac{\delta-2\delta^2-2x\delta}{1-2\delta} \right)} \cdot \left(\frac{(1-2\delta)(\delta+x)}{x} \right)^{\left(\frac{x}{1-2\delta} \right)} \right]$$

and $M_\delta = \max \left\{ f_\delta(x) : x \in \left[\frac{\delta(1-2\delta)}{1+2\delta}, \frac{\delta(1-2\delta)}{2\delta}\right] \right\}$. We claim that the running time of $(2-\delta, 2-\delta)$ -approximation will be $\mathcal{O}^*((M_\delta)^k)$.

In Theorem 20, we have seen the running time of $(2-\delta, 2-\delta)$ -approximation is $\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right)$. Here each term inside the summand is of the form $\binom{\delta k+i'}{\delta k-\frac{2i'\delta}{1-2\delta}}$ for some integer i' such that $\frac{\delta(1-2\delta)k}{1+2\delta} \leq i' \leq \frac{\delta k(1-2\delta)}{2\delta}$. i' can also be viewed as $\frac{\delta(1-2\delta)k}{1+2\delta} + s$ for some integer s such that $0 \leq s \leq \frac{(1-2\delta)k}{2(1+2\delta)}$. So, we can write $i' = \frac{\delta(1-2\delta)k}{1+2\delta} + s = \left(\frac{\delta(1-2\delta)}{1+2\delta} + \frac{s}{k}\right)k = tk$ where $t = \frac{\delta(1-2\delta)}{1+2\delta} + \frac{s}{k}$. Since, $0 \leq s \leq \frac{(1-2\delta)k}{2(1+2\delta)}$, we have $\frac{\delta(1-2\delta)}{1+2\delta} \leq t \leq \frac{\delta(1-2\delta)}{2\delta}$. Thus, t belongs to the domain of f_δ and hence, $M_\delta \geq f_\delta(t)$. This implies,

$$M_\delta \geq \left[\left(\frac{(1-2\delta)(\delta+t)}{\delta-2\delta^2-2t\delta} \right)^{\left(\frac{\delta-2\delta^2-2t\delta}{1-2\delta} \right)} \cdot \left(\frac{(1-2\delta)(\delta+t)}{t} \right)^{\left(\frac{t}{1-2\delta} \right)} \right]. \text{ Again, by Stirling's approximation,}$$

$$\begin{aligned} \binom{\delta k+i'}{\delta k-\frac{2i'\delta}{1-2\delta}} &= \binom{\delta k+tk}{\delta k-\frac{2tk\delta}{1-2\delta}} \leq \left[\left(\frac{(1-2\delta)(\delta+t)}{\delta-2\delta^2-2t\delta} \right)^{\left(\frac{\delta-2\delta^2-2t\delta}{(1-2\delta)(\delta+t)} \right)} \cdot \left(\frac{(1-2\delta)(\delta+t)}{t} \right)^{\left(\frac{t}{(1-2\delta)(\delta+t)} \right)} \right]^{(\delta+t)k} \\ &= \left[\left(\frac{(1-2\delta)(\delta+t)}{\delta-2\delta^2-2t\delta} \right)^{\left(\frac{\delta-2\delta^2-2t\delta}{1-2\delta} \right)} \cdot \left(\frac{(1-2\delta)(\delta+t)}{t} \right)^{\left(\frac{t}{1-2\delta} \right)} \right]^k = (f_\delta(t))^k \leq (M_\delta)^k. \end{aligned}$$

Since there are only $\text{poly}(k)$ many terms inside the summand, $\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right) \leq \text{poly}(k)(M_\delta)^k$. This implies,

$$\mathcal{O}^*\left(\sum_{i=\frac{\delta k(1-2\delta)}{1+2\delta}}^{\frac{\delta k(1-2\delta)}{2\delta}} \binom{\delta k+i}{\delta k-\frac{2i\delta}{1-2\delta}}\right) = \mathcal{O}^*((M_\delta)^k).$$

So, our new goal is to find M_δ for a given δ with $0 < \delta < 0.5$. We write a computer program which, given δ , goes through all possible $x \in \left[\frac{\delta(1-2\delta)}{1+2\delta}, \frac{\delta(1-2\delta)}{2\delta}\right]$ in steps of $dx = 0.00000001$, computes the value of $\{f_\delta(x)\}$ on all these possible values of x and returns the maximum. This way, the program gives us $M_\delta = \max \left\{ f_\delta(x) : x \in \left[\frac{\delta(1-2\delta)}{1+2\delta}, \frac{\delta(1-2\delta)}{2\delta}\right] \right\}$, for any given $\delta \in (0, 0.5)$. From the program, we observe that $M_{0.25} = 1.272$, $M_{0.10} = 1.151$, etc. Please refer to Table 1 for a table of running times for different values of δ . We have added a MATLAB code in Fig. C.2 that finds M_δ in the appendix.

8. Conclusion

In this paper, we considered the WEIGHTED VERTEX COVER problem parameterized by the solution size. We gave an exact algorithm running in time $\mathcal{O}^*(1.4656^k)$ and $(2-\delta, 2-\delta)$ -approximation algorithms for $\delta \in [0, 0.5]$ which run faster than the exact

Table 1
Table of running times for different values of δ .

No.	δ	approximation factor	M_δ	Run Time
1	0.05	(1.95, 1.95)	1.0943	1.0943 ^k
2	0.10	(1.90, 1.90)	1.1510	1.1510 ^k
3	0.15	(1.85, 1.85)	1.1968	1.1968 ^k
4	0.20	(1.80, 1.80)	1.2365	1.2365 ^k
5	0.25	(1.75, 1.75)	1.2720	1.2720 ^k
6	0.30	(1.70, 1.70)	1.3045	1.3045 ^k
7	0.35	(1.65, 1.65)	1.3345	1.3345 ^k
8	0.40	(1.60, 1.60)	1.3626	1.3626 ^k
9	0.45	(1.55, 1.55)	1.3891	1.3891 ^k

algorithm. It will be interesting to obtain a parameterized approximation scheme for (k, W) -VERTEX COVER which works for every $\delta \in (0, 1)$ where the running time scales between the running time of $(2, 2)$ -approximation algorithm and the running time of the exact algorithm.

CRedit authorship contribution statement

Soumen Mandal: Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Conceptualization. **Pranabendu Misra:** Writing – original draft, Methodology, Conceptualization. **Ashutosh Rai:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Saket Saurabh:** Writing – original draft, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgement

The authors would like to thank the editor and the anonymous reviewer for their valuable comments and suggestions, which helped to improve the quality of the paper.

The third author would like to thank Andreas Emil Feldmann and Cornelius Brand for some initial discussions on the problem.

Funding: Soumen Mandal was supported by Council of Scientific and Industrial Research (File No. 09/0086(11535)/2021-EMR-I), India. Ashutosh Rai was supported by Science and Engineering Research Board (SERB) under Grant SRG/2021/002412. Saket Saurabh was supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 819416), and Swarnajayanti Fellowship (No. DST/SJF/MSA01/2017-18).

Appendix A. Poly time algorithm for (k, W) -VERTEX COVER on graphs of maximum degree 2

Graphs with maximum degree 2 are disjoint collections of paths and cycles only. To solve (k, W) -VERTEX COVER on graphs with maximum degree 2, we first solve (k, W) -VERTEX COVER on paths and cycles using simple dynamic programming approach. Then using these solutions, we design another dynamic programming that solves (k, W) -VERTEX COVER problem for graphs whose connected components are either a path or a cycle, that is, graphs with maximum degree 2.

Lemma 21. (k, W) -VERTEX COVER can be solved in $\text{poly}(n)$ time on a path.

Proof. Let (P, w, k, W) be an input instance to (k, W) -VERTEX COVER where $P = (v_1, v_2, \dots, v_n)$ is a path on n vertices. We take a simple dynamic programming approach to solve (k, W) -VERTEX COVER on P . Throughout this proof, we denote $P[\{v_1, \dots, v_i\}]$ by P_i . Let us define the following.

$$A[i, k'] = \begin{cases} \text{weight of a minimum weight vertex cover of size exactly } k' \text{ containing } v_i \text{ for } P_i & \text{if one exists} \\ \infty & \text{otherwise.} \end{cases}$$

$$B[i, k'] = \begin{cases} \text{weight of a minimum weight vertex cover of size exactly } k' \text{ for } P_i & \text{if one exists} \\ \infty & \text{otherwise.} \end{cases}$$

First, we compute $A[i, k']$ and $B[i, k']$ for every $i \in \{0, \dots, n\}$ and $k' \in \{0, \dots, k\}$ and then we compare the value of $\min_{0 \leq k' \leq k} B[n, k']$ with W . We return Yes if $\min_{0 \leq k' \leq k} B[n, k']$ is at most W and return No otherwise. Notice that $B[n, k]$ is the weight of a minimum weight vertex cover of size exactly k for P .

Now we will see how $A[i, k']$ and $B[i, k']$ can be computed. We initialize, $A[i, k'] = \infty$, $B[i, k'] = \infty$ for $k' < 0$; $A[0, k'] = \infty$, $B[0, k'] = \infty$ for $k' > 0$ and $A[0, 0] = 0$, $B[0, 0] = 0$. Then for every $i \in \{0, \dots, n\}$ and $k' \in \{0, \dots, k\}$, we update $A[i, k']$ and $B[i, k']$ in the following way.

$$A[i, k'] = B[i-1, k'-1] + w(v_i). \\ B[i, k'] = \min\{A[i-1, k'], A[i, k']\}.$$

While updating $A[i, k'] = B[i-1, k'-1] + w(v_i)$, we use the convention $\infty + x = \infty$ for any $x \in \mathbb{R}$. We will show the correctness of the recurrence using induction on i . When $i = 0$, the only vertex cover of P_0 is $\emptyset$, and thus $A[0, 0] = 0$, $B[0, 0] = 0$. Since for P_0 , we can not have any vertex cover of any positive or negative size, so we must have $A[0, k'] = \infty$, $B[0, k'] = \infty$ for $k' < 0$ and $A[0, k'] = \infty$, $B[0, k'] = \infty$ for $k' > 0$. Hence, the base case (the case when $i = 0$) is true by the initial assumptions. Now, for the induction step, let us assume that for some $i' \leq i-1$ we have computed $A[i-1, k']$ and $B[i-1, k']$ correctly for every k' .

First, we will show that $A[i, k']$ or $B[i, k']$ are not finite if and only if there is no vertex cover for the path P_i of size exactly k' containing the vertex v_i or there is no vertex cover of size k' respectively.

Let $A[i, k'] = \infty$. We want to show that there is no vertex cover for the path P_i of size k' containing the vertex v_i . For the sake of contradiction, let S be one such vertex cover for P_i of size k' such that $v_i \in S$. Then $S \setminus \{v_i\}$ is a vertex cover for P_{i-1} of size $k'-1$. Thus by induction hypothesis, $B[i-1, k'-1]$ should be finite and hence $A[i, k'] = B[i-1, k'-1] + w(v_i)$ is finite. We get a contradiction that $A[i, k'] = \infty$. On the other hand, let us consider that there is no vertex cover for the path P_i of size k' containing the vertex v_i . Then there will be no vertex cover for P_{i-1} of size $k'-1$, otherwise, that vertex cover together with v_i would be a vertex cover for the path P_i of size k' containing the vertex v_i . So, we have $B[i-1, k'-1] = \infty$ which implies that $A[i, k'] = B[i-1, k'-1] + w(v_i) = \infty$.

Now, when $B[i, k'] = \infty$, we will show that there is no vertex cover for P_i of size k' . If S is a vertex cover for P_i of size k' , then at least one of v_i or v_{i-1} must be there in S as $v_i v_{i-1} \in E(P_i)$. This implies that at least one of $A[i-1, k']$ or $A[i, k']$ must be finite contradicting that $B[i, k'] = \min\{A[i-1, k'], A[i, k']\} = \infty$. On the other hand, let there be no vertex cover for P_i of size k' . Then both $A[i-1, k']$ and $A[i, k']$ must be infinite. Otherwise, the vertex cover corresponding to $A[i-1, k']$ or $A[i, k']$ would be a vertex cover for P_i of size k' .

Now, let us come to the case when $A[i, k']$ and $B[i, k']$ are finite. Let S be a vertex cover for P_i of size k' such that $v_i \in S$ and $w(S) = A[i, k']$. Also, let S' be a vertex cover for P_{i-1} of size $k'-1$ such that $w(S') = B[i-1, k'-1]$. We can see that $S' \cup \{v_i\}$ is a vertex cover of P_i of size k' and it also contains v_i . Thus by the definition of $A[i, k']$, $w(S) = A[i, k'] \leq w(S' \cup \{v_i\}) = w(S') + w(v_i) = B[i-1, k'-1] + w(v_i)$. On the other hand, $S \setminus \{v_i\}$ is a vertex cover for P_{i-1} of size $k'-1$ and thus $B[i-1, k'-1] = w(S') \leq w(S \setminus \{v_i\})$. It implies $B[i-1, k'-1] + w(v_i) = w(S') + w(v_i) \leq w(S \setminus \{v_i\}) + w(v_i) = w(S) = A[i, k']$. Hence, $A[i, k'] = B[i-1, k'-1] + w(v_i)$.

The above paragraph shows that we computed $A[i, k']$ correctly before computing $B[i, k']$. Here both $A[i-1, k']$ and $A[i, k']$ correspond to the weight of a vertex cover of size k' of P_i . Thus by the definition of $B[i, k']$, $B[i, k'] \leq \min\{A[i-1, k'], A[i, k']\}$. Let S be a vertex cover corresponding to $B[i, k']$. Since $v_{i-1} v_i \in E(P_i)$, at least one of v_{i-1}, v_i must belong to S . If $v_{i-1} \in S$, then we have $A[i-1, k'] \leq B[i, k']$ and if $v_i \in S$ then we have $A[i, k'] \leq B[i, k']$. Thus, we get $B[i, k'] \geq \min\{A[i-1, k'], A[i, k']\}$ and hence, $B[i, k'] = \min\{A[i-1, k'], A[i, k']\}$.

Now, notice that each table $A[.,.]$ and table $B[.,.]$ has nk entries and each entry can be computed in constant time. So in total, we need $\mathcal{O}(nk)$ time which is $\mathcal{O}(n^2)$ as $k \leq n$. $\square$

Corollary 22. (k, W) -VERTEX COVER can be solved in $\text{poly}(n)$ time on cycles.

Proof. Let (C, w, k, W) be an input instance to (k, W) -VERTEX COVER where C is a cycle. Also let $uv \in E(C)$. We know that both $C - u$ and $C - v$ are paths of $n-1$ vertices. Let $B_u[n-1, k-1]$ and $B_v[n-1, k-1]$ be the weight of a minimum weight vertex cover of size at most $k-1$ for $C - u$ and $C - v$ respectively. We know that at least one of u, v must be there in every vertex cover of C . Now, to find the weight of a minimum weight vertex cover for C containing u of size at most k , we need to first find the weight of a minimum weight vertex cover for the path $C - u$ of size at most $k-1$ and then add the weight of u with it. Similarly, to find the weight of a minimum weight vertex cover for C containing v of size at most k , we need to first find the weight of a minimum weight vertex cover for the path $C - v$ of size at most $k-1$ and then add the weight of v with it. Since every vertex cover of C contains at least one of u or v , the weight of a minimum weight vertex cover of size at most k for C will be $\min\{w(u) + B_u[n-1, k-1], w(v) + B_v[n-1, k-1]\}$. Let this value be $wt_k(C)$. We compare $wt_k(C)$ with W and return Yes or No, depending on $wt_k(C) \leq W$ or $wt_k(C) > W$ respectively. Since $B_u[n-1, k-1]$ and $B_v[n-1, k-1]$ can be computed in $\text{poly}(n)$ time due to Lemma 21, the weight of a minimum weight vertex cover for C of size at most k can also be computed in $\text{poly}(n)$ time. $\square$

A.1. Proof of Theorem 4

Proof. Let (G, w, k, W) be an input instance to (k, W) -VERTEX COVER where $\Delta(G) \leq 2$. Every component of G is either a path or a cycle. Let us first describe our simple dynamic programming algorithm and then we will see the proof of correctness. We name this algorithm as 'Deg2exact_algo'. First, we order the components of G arbitrarily. Let $C_1, C_2, \dots, C_t$ be the ordering where each C_i is either a path or a cycle. Let us denote the weight of a minimum weight vertex cover of size exactly j of C_i by $B_i[j]$. Throughout this proof, we denote $G[C_1 \cup C_2 \cup \dots \cup C_s]$ by G_s . Now, we define $A[s, k']$ in the following way.

$$A[s, k'] = \begin{cases} \text{weight of a minimum weight vertex cover of size exactly } k' \text{ of } G_s & \text{if one exists} \\ \infty & \text{otherwise.} \end{cases}$$

We compute $A[s, k']$ for every $s \in \{1, \dots, t\}$ and $k' \in \{0, \dots, k\}$ and finally compare the value $\min_{0 \leq k' \leq k} A[t, k']$ with W . We return Yes if $\min_{0 \leq k' \leq k} A[t, k'] \leq W$ and No otherwise. Now, we will see how $A[s, k']$ can be computed. We initialize, $A[1, k'] = B_1[k']$ and for $s \in \{2, \dots, t\}$ we have

$$A[s, k'] = \min_{0 \leq j \leq k'} [B_s[j] + A[s-1, k' - j]].$$

Now we will show the proof of correctness, that is, we will show that each $A[s, k']$ is computed correctly. The base case $A[1, k'] = B_1[k']$ is true because of Lemma 21 and Corollary 22. Whenever we compute $A[s, k']$ for some $2 \leq s \leq t$, by induction hypothesis we will assume that $A[s', k']$ has been computed correctly for every $s' < s$.

First, we will show that $A[s, k'] = \infty$ if and only if G_s does not have a vertex cover of size k' . Let $A[s, k'] = \infty$, which means that for every $j \in \{0, \dots, k'\}$, at least one of $B_s[j]$ or $A[s-1, k' - j]$ will not be finite. For the sake of contradiction, if there is a vertex cover S of size k' for G_s . Let us take $|S \cap G[C_s]| = j'$. Clearly, $0 \leq j' \leq k'$. Also, $S \setminus G[C_s]$ is a vertex cover for G_{s-1} of size $k - j'$. This implies that $B_s[j']$ is finite (by Lemma 21 and Corollary 22) and $A[s-1, k' - j]$ is finite by induction hypothesis. Thus $B_s[j'] + A[s-1, k' - j]$ is finite, contradicting that $A[s, k'] = \infty$. For the converse, let us assume that G_s does not have a vertex cover of size k' . This means for every j such that $0 \leq j \leq k'$, at least one of $G[C_s]$ and G_{s-1} does not have a vertex cover of size j and $k' - j$ respectively. Thus for every j , at least one of $B_s[j]$ (by Lemma 21 and Corollary 22), $A[s-1, k' - j]$ (induction hypothesis) is not finite.

Now, let us assume that there exists a vertex cover of G_s of size k' . Let minimum of $B_s[j] + A[s-1, k' - j]$ be obtained for some $j = j'$. Now, let S_1 be minimum weight vertex cover of C_s of size j' and S_2 be minimum weight vertex cover of G_{s-1} of size $k' - j'$. Clearly, $S_1 \cup S_2$ is a vertex cover of G_s of size k' and thus $A[s, k'] \leq w(S_1 \cup S_2) = w(S_1) + w(S_2) = B_s[j'] + A[s-1, k' - j'] = \min_{0 \leq j \leq k'} [B_s[j] + A[s-1, k' - j]]$.

On the other hand, let S be a minimum weight vertex cover of G_s of size k' . Let $S \cap C_s = S_1$ ($0 \leq |S_1| \leq k'$) and $S \setminus S_1 = S_2$. Clearly, S_1 is a vertex cover for C_s and S_2 is a vertex cover for G_{s-1} . Thus, $A[s, k'] = w(S) = w(S_1) + w(S_2) \geq B_s[|S_1|] + A[s-1, k' - |S_1|] \geq \min_{0 \leq j \leq k'} [B_s[j] + A[s-1, k' - j]]$.

Now we are only left to argue the running time. Notice that table of $A[.,.]$ has tk entries and each $B_s[j]$ can be computed in $\mathcal{O}(n^2)$ time. So, each entry of table $A[.,.]$ can be computed in $\mathcal{O}(k'n^2)$ time which is at most $\mathcal{O}(kn^2)$ as we need to compute $B_s[j]$ for all $0 \leq j \leq k'$. Thus, in total, we need $\mathcal{O}(tk^2n^2)$ time which is $\text{poly}(n)$ only. $\square$

Appendix B. Treewidth algorithm for (k, W) -VERTEX COVER

Here we present an FPT algorithm for (k, W) -VERTEX COVER parameterized by the treewidth of the input graph. First let us see the definitions of *tree-decompositions* and *treewidth*.

Definition 23 ([20]). A tree-decomposition of a graph G is a pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where T is a tree where every node t is assigned a vertex subset $X_t \subseteq V(G)$, called a bag, such that the following three conditions hold:

- (T1) $\bigcup_{t \in V(T)} X_t = V(G)$. In other words, every vertex of G is in at least one bag.
- (T2) For every $uv \in E(G)$, there exists a node t of T such that bag X_t contains both u and v .
- (T3) For every $u \in V(G)$, the set $T_u = \{t \in V(T) : u \in X_t\}$, i.e., the set of nodes whose corresponding bags contain u , induces a connected sub-tree of T .

The width of tree-decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ equals $\max_{t \in V(T)} |X_t| - 1$, that is, the maximum size of its bag minus 1. The treewidth of a graph G , denoted by $\text{tw}(G)$, is the minimum possible width of a tree-decomposition of G .

For proving Theorem 25, we use the standard dynamic programming for weighted vertex cover on bounded treewidth graph, where for each bag X_t of the tree-decomposition, each $k' \leq k$, and each subset $S \subseteq X_t$ of vertices of the bag, we store the weight of a minimum vertex cover of graph induced by the bags below X_t the size exactly k' such that its intersection with X_t is exactly S . Finally, looking at some entry corresponding to the root bag will give us the answer. Before going into recurrences of this dynamic programming, let us look at the definition of a *nice tree-decomposition*.

A rooted tree-decomposition $(T, \{X_t\}_{t \in V(T)})$ is *nice* if the following conditions are satisfied:

- $X_r = \emptyset$ and $X_\ell = \emptyset$ for every leaf ℓ of T . In other words, all the leaves as well as the root contain empty bags.
- Every non-leaf node of T is of one of the following three types:
 1. Introduce node: a node t with exactly one child t' such that $X_t = X_{t'} \cup \{v\}$ for some vertex $v \notin X_{t'}$; we say that v is introduced at t .
 2. Forget node: a node t with exactly one child t' such that $X_t = X_{t'} \setminus \{w\}$ for some vertex $w \in X_{t'}$; we say that w is forgotten at t .
 3. Join node: a node t with two children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$.

Now, we will state a lemma that says that if we have a tree-decomposition of width at most d for a graph G , then we can always compute a nice tree-decomposition G of width at most d in polynomial time.

Lemma 24 ([20]). If a graph G admits a tree-decomposition of width at most d , then it also admits a nice tree-decomposition of width at most d . Moreover, given a tree-decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width at most d , one can in time $\mathcal{O}(d^2 \cdot \max(|V(T)|, |V(G)|))$ compute a nice tree-decomposition of G of width at most d that has at most $\mathcal{O}(d|V(G)|)$ nodes.

So, we always assume tree-decomposition of a graph G as a nice tree-decomposition whose internal nodes are of three types only. For a node t of T , let V_t be the union of all the bags present in the subtree of T rooted at t , including X_t . Clearly, we can see $V_r = V(G)$. Now for a given instance (G, w, k, W) of (k, W) -VERTEX COVER with $\Delta(G) \leq 2$, we can compute a nice tree-decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G . For every node t , every $S \subseteq X_t$ and every $i \leq k$, we define the following value:

$c[i, t, S]$ = minimum possible weight of a vertex cover $\hat{S}$ for $G[V_t]$ of size exactly i

such that $S \subseteq \hat{S} \subseteq V_t$ and $\hat{S} \cap X_t = S$.

If no such $\hat{S}$ exists, we put $c[i, t, S] = \infty$. This can happen when S is not a vertex cover for $G[X_t]$ or there is no vertex cover $\hat{S}$ of size i for $G[V_t]$ with $\hat{S} \cap X_t = S$.

Now, we give the recurrence formula to compute $c[., ., .]$.

Leaf node. If t is a leaf node, then

$$c[i, t, \emptyset] = \begin{cases} 0 & \text{if } i = 0; \\ \infty & \text{if } i > 0. \end{cases}$$

Since the bag corresponds to a leaf node of a nice tree decomposition is empty, the only vertex cover of the graph induced on subtree rooted at a leaf node will be an empty set. Thus $c[i, t, \emptyset] = 0$ corresponding to the empty vertex cover, and $c[i, t, \emptyset] = \infty$ when $i > 0$ as it can't have a vertex cover of bigger than 0.

Introduce node. Let t be a introduce node with only child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Let S be any subset of X_t . If S is not a vertex cover for $G[X_t]$ or $i < |S|$, then $c[i, t, S] = \infty$. Otherwise,

$$c[i, t, S] = \begin{cases} c[i, t', S] & \text{if } v \notin S; \\ c[i-1, t', S \setminus \{v\}] + w(v) & \text{if } v \in S. \end{cases}$$

We will prove this formula in two cases depending on $v \notin S$ and $v \in S$. When $v \notin S$, the families of sets $\hat{S}$ considered in the definition of $c[i, t, S]$ and $c[i, t', S]$ are equal which implies that $c[i, t, S] = c[i, t', S]$.

Now the case when $v \in S$, let $\hat{S}$ be a set for which the minimum is attained in the definition of $c[i, t, S]$. Then it follows that $\hat{S} \setminus \{v\}$ is one of the sets considered in the definition of $c[i-1, t', S \setminus \{v\}]$, which implies that $c[i-1, t', S \setminus \{v\}] \leq w(\hat{S} \setminus \{v\}) = w(\hat{S}) - w(v) = c[i, t, S] - w(v)$. This implies $c[i-1, t', S \setminus \{v\}] + w(v) \leq c[i, t, S]$.

On the other hand, let $\hat{S}'$ be a set for which the minimum is attained in the definition of $c[i-1, t', S \setminus \{v\}]$. So, $\hat{S}'$ is a vertex cover for $G[V_{t'}]$ of size exactly $i-1$. The only edges $G[V_t]$, $\hat{S}'$ might not cover are the edges whose one end is v . Thus $\hat{S}' \cup \{v\}$ is a vertex cover for $G[V_t]$ of size exactly i . Also, the intersection of $\hat{S}' \cup \{v\}$ with X_t is exactly S and S is a vertex cover for $G[X_t]$. Thus it is considered in the definition of $c[i, t, S]$ and we get $c[i, t, S] \leq w(\hat{S}' \cup \{v\}) = w(\hat{S}') + w(v) = c[i-1, t', S \setminus \{v\}] + w(v)$.

Forget node. Suppose t is a forget node with child t' such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_{t'}$. Let S be any subset of X_t . Again if, S is not a vertex cover for $G[X_t]$ or $i < |S|$ then $c[i, t, S] = \infty$. Otherwise,

$$c[i, t, S] = \min \{c[i, t', S], c[i, t', S \cup \{w\}]\}.$$

To prove this formula, let us consider the $\hat{S}'$ for which minimum is attained in the definition of $c[i, t, S]$. If $w \notin \hat{S}'$ then $\hat{S}'$ is one of the set considered in the definition of $c[i, t', S]$ and thus $c[i, t', S] \leq c[i, t, S]$. Now, if $w \in \hat{S}'$ then $\hat{S}' \cup \{w\}$ is one of the set considered in the definition of $c[i, t', S \cup \{w\}]$, which implies $c[i, t', S \cup \{w\}] \leq c[i, t, S]$. As exactly one of these alternatives happens, we can say:

$$c[i, t, S] \geq \min \{c[i, t', S], c[i, t', S \cup \{w\}]\}.$$

On the other hand, each set is considered for the definition of $c[i, t', S]$ or $c[i, t', S \cup \{w\}]$ is also considered for $c[i, t, S]$. This implies:

$$c[i, t, S] \leq \min \{c[i, t', S], c[i, t', S \cup \{w\}]\}.$$

Join node. Now for join node, t , with children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$. Let S be any subset of X_t . Again if, S is not a vertex cover for $G[X_t]$ or $i < |S|$ then $c[i, t, S] = \infty$. Otherwise,

$$c[i, t, S] = \min_{i_1+i_2=i+|S|} \{c[i_1, t_1, S] + c[i_2, t_2, S] - w(S)\}.$$

Let $\hat{S}_1$ and $\hat{S}_2$ be two sets consider for $\min_{i_1+i_2=i+|S|} \{c[i_1, t_1, S] + c[i_2, t_2, S] - w(S)\}$ corresponding to $c[i_1, t_1, S]$ and $c[i_2, t_2, S]$ respectively. Thus, we have, $|\hat{S}_1| = i_1$, $|\hat{S}_2| = i_2$ and $i_1 + i_2 = i + |S|$. Let us consider $\hat{S} = \hat{S}_1 \cup \hat{S}_2$. Since $\hat{S}_1 \cap X_{t_1} = \hat{S}_2 \cap X_{t_2} = S$, $\hat{S} \cap X_t = S$ and S will be a vertex cover for $G[X_t]$ as S is a vertex cover for both X_{t_1} and X_{t_2} . Also, $\hat{S}$ will be a vertex cover for $G[V_t]$ as both endpoints of edges having one end at V_{t_1} and another at V_{t_2} , must be in X_t . Hence the set $\hat{S}$ is one of the sets considered in the definition of $c[i, t, S]$. Thus,

$$c[i, t, S] \leq w(\hat{S}) = w(\hat{S}_1) + w(\hat{S}_2) - w(S) = \min_{i_1+i_2=i+|S|} \{c[i_1, t_1, S] + c[i_2, t_2, S] - w(S)\}.$$

Second equality is due to the fact $\hat{S}_1 \cap \hat{S}_2 = S$.

On the other hand, let the set $\hat{S}'$ is considered for $c[i, t, S]$. Thus, $|\hat{S}'| = i$ and $X_t \cap \hat{S}' = S$. Let $V_{t_1} \cap \hat{S}' = \hat{S}'_1$ and $V_{t_2} \cap \hat{S}' = \hat{S}'_2$. Also, let us consider $|\hat{S}'_1| = i_1$ and $|\hat{S}'_2| = i_2$. Now,

$$i_1 + i_2 = |\hat{S}'_1| + |\hat{S}'_2| = |\hat{S}'_1 \cup \hat{S}'_2| + |\hat{S}'_1 \cap \hat{S}'_2| = |\hat{S}'| + |S| = i + |S|.$$

Again, $\hat{S}'_1$ is a vertex cover for $G[V_{t_1}]$ and $\hat{S}'_2$ is a vertex cover for $G[V_{t_2}]$. Also, $\hat{S}'_1 \cap X_{t_1} = S$ and $\hat{S}'_2 \cap X_{t_2} = S$. Thus, $\hat{S}'_1$ is considered in the definition of $c[i_1, t_1, S]$ and $\hat{S}'_2$ is considered in the definition of $c[i_2, t_2, S]$. Hence we get,

$$c[i, t, S] = w(\hat{S}') = w(\hat{S}'_1) + w(\hat{S}'_2) - w(S) \geq c[i_1, t_1, S] + c[i_2, t_2, S] - w(S) \geq \min_{i_1+i_2=i+|S|} \{c[i_1, t_1, S] + c[i_2, t_2, S] - w(S)\}.$$

Theorem 25. Given an instance $I = (G, w, k, W)$ and a tree-decomposition of G of width tw , (k, W) -VERTEX COVER can be solved in $\mathcal{O}^*(2^{tw})$ time on I .

Proof. We first compute a nice tree-decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width tw , which can be done given a tree-decomposition of G of width tw in linear time. Then by the recursion describe above, we compute $c[i, t, S]$ for every node t of T and for every $i \leq k$. Here $c[i, t, S]$ gives the minimum possible weight of a vertex cover of size i for $G[V_t]$. We first start at leaf nodes and then gradually we go upwards in the tree T . If r is the root of T , then we know $V_r = V(G)$ and we also know that $X_r = \emptyset$. So, $c[i, r, \emptyset]$ will be the minimum possible weight of a vertex cover of size i . Now, if $c[i, r, \emptyset] > W$ for all $i \leq k$, then we say (G, w, k, W) is a No instance. If (G, w, k, W) is not a No instance, there is a $i \leq k$ for which $c[i, r, \emptyset] \leq W$. Then by backtracking in the recursions for the entry $c[i, r, \emptyset]$, we can compute a vertex cover S of size at most k with weight at most W .

At every node t , we compute $c[i, t, S]$ for every $S \subseteq X_t$ and $i \leq k$. Also, to compute each entry, we take only constant time (except at join nodes). So, other than join node, we take $k \cdot 2^{|X_t|}$ at each node t of T . If t is a join node then we need $\mathcal{O}(k^3 \cdot 2^{|X_t|})$ time. Since there can be at most $\mathcal{O}(tw \cdot n)$ nodes in the nice tree-decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G , the total time we need for this algorithm is $\mathcal{O}(k^3 \cdot 2^{tw} \cdot tw \cdot n)$, which is $\mathcal{O}^*(2^{tw})$. $\square$

Appendix C. MATLAB code for M_δ

```

clc
clear

dx=0.00000001;
prompt = "Enter the value of delta ";
d = input(prompt)

x=((d*(1-(2*d)))/(1+(2*d)))dx:((1-(2*d))/2);
for i=1:length(x)
    y(i)=((((1-(2*d))((d)+((i-1)*dx)))/((d)-(2*((d)^2))-(2*((i-1)*dx*d))))((d)-
    (2*((d)^2))-(2*((i-1)*dx*d)))/((1-2*d)*((d)+((i-1)*dx))))((1-(2*d))*((d)+((i-1)*dx)))/((i-1)*dx))((i-1)*dx))/((1-(2*d))*((d)+((i-1)*dx))))((d)+((i-1)*dx));
end
M_delta=max(y);

display(M_delta)

```

Fig. C.2. MATLAB code for M_δ .

References

- [1] R.M. Karp, Reducibility among combinatorial problems, in: Proceedings of a Symposium on the Complexity of Computer Computations, Held, March 20–22, 1972, in: The IBM Research Symposia Series, Plenum Press, New York, 1972, pp. 85–103.
- [2] G.L. Nemhauser, L.E. Trotter, Vertex packings: structural properties and algorithms, Math. Program. 8 (1) (1975) 232–248.
- [3] I. Dinur, S. Khot, G. Kindler, D. Minzer, M. Safra, Towards a proof of the 2-to-1 games conjecture?, in: Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25–29, 2018, ACM, 2018, pp. 376–389.
- [4] S. Khot, D. Minzer, M. Safra, Pseudorandom sets in Grassmann graph have near-perfect expansion, Electron. Colloquium Comput. Complex. TR18–006, <https://eccc.weizmann.ac.il/report/2018/006>, 2018.
- [5] S. Khot, O. Regev, Vertex cover might be hard to approximate to within 2-epsilon, J. Comput. Syst. Sci. 74 (3) (2008) 335–349.
- [6] D.G. Harris, N.S. Narayanaswamy, A faster algorithm for vertex cover parameterized by solution size, in: 41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, March 12–14, 2024, Clermont-Ferrand, France, in: LIPIcs, vol. 289, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 40:1–40:18.
- [7] N. Bourgeois, B. Escoffier, V.T. Paschos, Approximation of max independent set, min vertex cover and related problems by moderately exponential algorithms, Discrete Appl. Math. 159 (17) (2011) 1954–1970.
- [8] M.R. Fellows, A. Kulik, F.A. Rosamond, H. Shachnai, Parameterized approximation via fidelity preserving transformations, J. Comput. Syst. Sci. 93 (2018) 30–40.

- [9] L. Brankovic, H. Fernau, A novel parameterised approximation algorithm for minimum vertex cover, *Theor. Comput. Sci.* 511 (2013) 85–108.
- [10] A. Kulik, H. Shachnai, Analysis of two-variable recurrence relations with application to parameterized approximations, *IEEE*, 2020, pp. 762–773.
- [11] A.E. Feldmann, C.S. Karthik, E. Lee, P. Manurangsi, A survey on approximation in parameterized complexity: hardness and algorithms, *Algorithms* 13 (6) (2020) 146.
- [12] F.V. Fomin, S. Gaspers, S. Saurabh, Branching and treewidth based exact algorithms, in: *Algorithms and Computation, 17th International Symposium, ISAAC 2006, Kolkata, India, December 18–20, 2006, Proceedings*, in: *Lecture Notes in Computer Science*, vol. 4288, Springer, 2006, pp. 16–25.
- [13] R. Niedermeier, P. Rossmanith, On efficient fixed-parameter algorithms for weighted vertex cover, *J. Algorithms* 47 (2) (2003) 63–77.
- [14] H. Shachnai, M. Zehavi, A multivariate framework for weighted FPT algorithms, *J. Comput. Syst. Sci.* 89 (2017) 157–189.
- [15] J. Chen, F.V. Fomin, Y. Liu, S. Lu, Y. Villanger, Improved algorithms for feedback vertex set problems, *J. Comput. Syst. Sci.* 74 (7) (2008) 1188–1198.
- [16] S. Saurabh, What's next? Future directions in parameterized complexity, in: *Recent Advances in Parameterized Complexity School*, Tel Aviv, December 2017.
- [17] E.J. Kim, S. Kratsch, M. Pilipczuk, M. Wahlström, Directed flow-augmentation, in: *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing*, Rome, Italy, June 20 - 24, 2022, ACM, 2022, pp. 938–947.
- [18] E. Galby, D. Marx, P. Schepper, R. Sharma, P. Tale, Parameterized complexity of weighted multicut in trees, in: *Graph-Theoretic Concepts in Computer Science - 48th International Workshop, WG 2022, Tübingen, Germany, June 22–24, 2022*, in: *Lecture Notes in Computer Science*, vol. 13453, Springer, 2022, pp. 257–270.
- [19] E.J. Kim, M. Pilipczuk, R. Sharma, M. Wahlström, On weighted graph separation problems and flow-augmentation, *CoRR*, arXiv:2208.14841 [abs], 2022.
- [20] M. Cygan, F.V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, S. Saurabh, *Parameterized Algorithms*, Springer, 2015.