Close Relatives (Of Feedback Vertex Set), Revisited

Hugo Jacob
ENS Paris-Saclay, France

Thomas Bellitto
Sorbonne Université, CNRS, LIP6 UMR 7606, Paris, France

Oscar Defrain
Aix-Marseille Université, CNRS, LIS UMR 7020, Marseille, France

Marcin Pilipczuk
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland

Abstract

At IPEC 2020, Bergougnoux, Bonnet, Brettell, and Kwon (*Close Relatives of Feedback Vertex Set Without Single-Exponential Algorithms Parameterized by Treewidth*, IPEC 2020, LIPIcs vol. 180, pp. 3:1–3:17) showed that a number of problems related to the classic Feedback Vertex Set (FVS) problem do not admit a $2^{o(k \log k)} \cdot n^{\mathcal{O}(1)}$-time algorithm on graphs of treewidth at most k, assuming the Exponential Time Hypothesis. This contrasts with the $3^k \cdot k^{\mathcal{O}(1)} \cdot n$-time algorithm for FVS using the Cut&Count technique.

During their live talk at IPEC 2020, Bergougnoux et al. posed a number of open questions, which we answer in this work.

- Subset Even Cycle Transversal, Subset Odd Cycle Transversal, Subset Feedback Vertex Set can be solved in time $2^{\mathcal{O}(k \log k)} \cdot n$ in graphs of treewidth at most k. This matches a lower bound for Even Cycle Transversal of Bergougnoux et al. and improves the polynomial factor in some of their upper bounds.
- Subset Feedback Vertex Set and Node Multiway Cut can be solved in time $2^{\mathcal{O}(k \log k)} \cdot n$, if the input graph is given as a cliquewidth expression of size n and width k.
- Odd Cycle Transversal can be solved in time $4^k \cdot k^{\mathcal{O}(1)} \cdot n$ if the input graph is given as a cliquewidth expression of size n and width k. Furthermore, the existence of a constant $\varepsilon > 0$ and an algorithm performing this task in time $(4 - \varepsilon)^k \cdot n^{\mathcal{O}(1)}$ would contradict the Strong Exponential Time Hypothesis.

A common theme of the first two algorithmic results is to represent connectivity properties of the current graph in a state of a dynamic programming algorithm as an auxiliary forest with $\mathcal{O}(k)$ nodes. This results in a $2^{\mathcal{O}(k \log k)}$ bound on the number of states for one node of the tree decomposition or cliquewidth expression and allows to compare two states in $k^{\mathcal{O}(1)}$ time, resulting in linear time dependency on the size of the graph or the input cliquewidth expression.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases feedback vertex set, treewidth, cliquewidth

Digital Object Identifier 10.4230/LIPIcs.IPEC.2021.21

Related Version *Full Version*: https://arxiv.org/abs/2106.16015

Funding This research is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme Grant Agreement 714704. This research was conducted while Hugo Jacob was doing a research internship at the University of Warsaw.

1 Introduction

Treewidth, introduced by Robertson and Seymour in their seminal Graph Minors series [28], but also independently introduced under different names by other authors, is probably the most successful graph width notion. (For the formal definition of treewidth and other width notions mentioned in this introduction, we refer to Section 2.) From the algorithmic point of view, its applicability is described by Courcelle's theorem [10] that asserts that every problem expressible in monadic second order logic with quantification over vertex sets and edge sets, can be solved in linear time (in the size of the graph) on graphs of bounded treewidth.

Due to the abundance of algorithms for graphs of bounded treewidth, their use in practice, and since Courcelle's theorem provides a very weak bound on the dependency of the running time of the algorithm on the treewidth of the input graph, a lot of research in the last decade has been devoted to understanding optimal running time bounds for algorithms on graphs of bounded treewidth. One of the first methodological approaches was provided by two works of Lokshtanov, Marx, and Saurabh [22, 23, 24, 25]. Their contribution can be summarized as follows.

- For a number of classic problems, the known (and very natural) dynamic programming algorithm, given an n-vertex graph G and a tree decomposition of width k, runs in time $c^k \cdot n^{\mathcal{O}(1)}$ for a constant $c > 1$. [22, 24] shows that in most cases the constant c is optimal, assuming the Strong Exponential Time Hypothesis.[1]
- [23, 25] introduces a framework for proving lower bounds (assuming the Exponential Time Hypothesis) against $2^{o(k \log k)} \cdot n^{\mathcal{O}(1)}$-time algorithms with the same input as above.

Both aforementioned works seemed to point to a general conclusion that the natural and naive dynamic programming algorithms on graphs of bounded treewidth are probably optimal in essentially all interesting cases. This intuition has been refuted by Cygan et al. [16] who presented the Cut&Count technique which allowed $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$-time algorithms on graphs of treewidth k for many connectivity problems where the natural and naive algorithm runs in time $2^{\mathcal{O}(k \log k)} \cdot n^{\mathcal{O}(1)}$. One of the prominent examples of such problems is Feedback Vertex Set (FVS) where, given a graph G and an integer p, one asks for a set of at most p vertices that hits all cycles of G.

Since then, the intricate landscape of optimal algorithms parameterized by the treewidth has been explored by many authors, see e.g. [1, 2, 3, 4, 7, 8, 14, 15, 27, 29]. Last year at IPEC 2020, Bergougnoux, Bonnet, Brettell, and Kwon [5] presented an in-depth study of problems related to FVS, showing that for most of them $2^{\mathcal{O}(k \log k)}$ is the optimal (assuming ETH) dependency on treewidth in the running time bound. During their live talk at IPEC 2020, they asked a number of open questions. In this work, we continue this line of research and answer all of them.

Hitting cycles in graphs of bounded treewidth

We first focus on the problems Odd Cycle Transversal (OCT) and Even Cycle Transversal (ECT) where, given a graph G and an integer p, the goal is to pick a set of at most p vertices of G that hits all odd cycles (resp. even cycles) of G. These problems are thus closely related to the aforementioned FVS problem that asks to hit *all* cycles. Using the fact that graphs without odd cycles are exactly bipartite graphs, it is relatively easy to obtain a $3^k \cdot k^{\mathcal{O}(1)} \cdot n$-time algorithm for OCT for graphs equipped with a tree decomposition of width k [17], and the base 3 of the exponent is optimal assuming SETH [22, 24].

[1] For a discussion on the complexity assumptions used, namely the Exponential Time Hypothesis (ETH) and the Strong Exponential Time Hypothesis (SETH), we refer to Chapter 14 of [13].

In contrast to FVS and OCT, Bergougnoux et al. [5] showed that, assuming ETH, ECT admits no $2^{o(k \log k)} \cdot n^{\mathcal{O}(1)}$-time algorithm and asked for a matching upper bound. Our first result is a positive answer to this question, even in a more general setting of Subset Even Cycle Transversal (SECT). In Subset Feedback Vertex Set (SFVS), Subset Odd Cycle Transversal (SOCT), and Subset Even Cycle Transversal, given a graph G, a set $S \subseteq V(G)$, and an integer p, the goal is to find a set of at most p vertices that hits every cycle (resp. odd cycle, even cycle) that passes through a vertex of S.

▶ **Theorem 1.** *Subset Feedback Vertex Set, Subset Odd Cycle Transversal and Subset Even Cycle Transversal, even in the weighted setting, can be solved in time* $2^{\mathcal{O}(k \log k)} \cdot n$ *on* n*-vertex graphs of treewidth* k*.*

Here, and in later statements, by *weighted setting* we mean the following: every vertex has its positive integer weight, and the input integer p becomes an upper bound on the total weight of the solution.

Misra, Raman, Ramanujan, and Saurabh [26] showed that a graph G does not contain an even cycle if and only if every block (2-connected component) of G is an edge or an odd cycle. The key ingredient of the proof of Theorem 1 for SECT is a characterization (in the same spirit, but more involved) of graphs G with sets $S \subseteq V(G)$ that do not contain an even cycle passing through a vertex of S.

This improves the polynomial factor of the running time bound of [5] for SOCT and SFVS from cubic to linear.

Clique-width parameterization

We then switch our attention to clique-width. Clique-width is a width measure aiming at capturing simple yet (contrary to treewidth) dense graphs. It originates from works of Courcelle, Engelfriet, and Rozenberg [11] and of Wanke [30] from early 90s. Informally speaking, a graph G is of clique-width at most k if one can provide an expression (called a *k-expression*) that constructs G using only k *labels* which essentially are names for vertex sets. Clique-width plays the role of treewidth for dense graphs in the following sense: any problem expressible in monadic second order logic with quantification over vertex sets (but not edge sets) can be solved in time $f(k) \cdot n$, given a k-expression of size n constructing the input graph, where f is some computable function [12]. Similarly as for treewidth, it is natural to investigate optimal functions f in such running time bounds. Here, the most relevant works are due to Bui-Xuan, Suchý, Telle, and Vatshelle [9] who showed an algorithm with $f(k) = 2^{\mathcal{O}(k \log k)}$ for FVS, and Bergougnoux and Kanté [6] who later showed an algorithm with $f(k) = 2^{\mathcal{O}(k)}$ by adapting the algorithm of Bodlaender et al. [7] for graphs of bounded treewidth to the context of bounded clique-width.

One should also mention a long line of work [18, 19, 20] searching for optimal running time bounds on graphs of bounded clique-width for problems *not* captured by the aforementioned meta-theorem and that provably (unless FPT = W[1]) do not have algorithms with the running time bound $f(k) \cdot n^{\mathcal{O}(1)}$, given a k-expression building the input graph.

Following on the open questions provided by Bergougnoux et al., we focus on SFVS and Node Multiway Cut (NMwC). In the second problem, given a graph G, a set $T \subseteq V(G)$, and an integer p, the goal is to find a set of at most p vertices that does not contain any vertex of T, but hits all paths with both endpoints in T. We show the following.

▶ **Theorem 2.** *Subset Feedback Vertex Set and Node Multiway Cut, even in the weighted setting, can be solved in time* $2^{\mathcal{O}(k \log k)} \cdot n$ *if the input graph is given as a* k*-expression of size* n*.*

Note that the running time bound of Theorem 2 matches the lower bound of Bergougnoux et al. [5] for pathwidth parameterization[2] of SFVS and NMwC, and it is straightforward to turn a path decomposition of width ℓ into a k-expression for $k = \ell + \mathcal{O}(1)$.

Observe also that, if vertex weights are allowed, NMwC reduces to SFVS. Namely, given a NMwC instance (G, T, p), set the weights of all vertices of T to $+\infty$, create a graph G' by adding to G a new vertex s of weight $+\infty$ adjacent to all vertices of T and set $S := \{s\}$; the SFVS instance (G', S, p) is easily seen to be equivalent to the input NMwC instance (G, T, P). Since it is straightforward to turn a k-expression of G into a $(2k)$-expression of G', in Theorem 2 it suffices to focus only on the SFVS problem.

A common theme in the dynamic programming algorithm of Theorem 1 and of Theorem 2 is the representation of the connectivity in the currently analyzed graph as an auxiliary forest of size $\mathcal{O}(k)$ with some annotations. This allows a neat description of the essential connectivity features, avoiding involved case analysis. The $\mathcal{O}(k)$ bound serves two purposes. First, it implies a bound of $2^{\mathcal{O}(k \log k)}$ on the number of states of the dynamic programming algorithm at one node of the tree decomposition or k-expression. Second, it allows to perform computations on states in $k^{\mathcal{O}(1)}$ time, giving the final linear dependency on the size of the graph or the input k-expression in the running time bound.

Hitting odd cycles in graphs of bounded clique-width

Finally, we restrict our attention to Odd Cycle Transversal. Recall that in graphs of treewidth k, OCT admits an algorithm with running time bound $3^k \cdot k^{\mathcal{O}(1)} \cdot n$ [17] and the base 3 is optimal assuming SETH [22, 24]. We show that for clique-width, the optimal base is 4.

▶ **Theorem 3.** *Odd Cycle Transversal, even in the weighted setting, can be solved in time $4^k \cdot k^{\mathcal{O}(1)} \cdot n$ if the input graph is given as a k-expression of size n. Furthermore, the existence of a constant $\varepsilon > 0$ and an algorithm performing the same task in time $(4-\varepsilon)^k \cdot n^{\mathcal{O}(1)}$ contradicts the Strong Exponential Time Hypothesis.*

The key insight in the OCT algorithm of [17] is to reformulate the problem into finding explicitly a partition $V(G) = X \uplus A \uplus B$ that minimizes $|X|$ while keeping $G[A]$ and $G[B]$ both edgeless. Then, in a dynamic programming algorithm on a tree decomposition, one remembers the assignment of the vertices of the current bag into X, A, and B; this yields the 3^k factor in the time complexity. For clique-width, a similar approach yields 4^k states: every label may be allowed to contain only vertices of X, allowed to contain vertices of X or A but not B, allowed to contain vertices of X or B but not A, or allowed to contain vertices of any of the three sets. To obtain the upper bound of Theorem 3, one needs to add on top of the above an appropriate convolution-like treatment of the disjoint union nodes of the k-expression. The lower bound of Theorem 3 combines a way to encode evaluation of two variables of a CNF-SAT formula into one of the four aforementioned states of a single label with a few gadgets for checking in the OCT regime if a clause is satisfied, borrowed from the corresponding reduction for pathwidth from [22, 24].

This extended abstract contains only an overview of the proofs of Theorems 1 and 2.

[2] We do not formally define pathwidth in this work, as it is not used except for this paragraph.

2 Preliminaries

For most standard definitions and notations used in this paper, we refer to the preliminaries section in the full version of the paper. We recall here only more nonstandard definitions.

A *rooted tree* is a tree T together with a special vertex $r \in V(T)$, called the *root*. It induces a natural ancestor-descendent relation $\leq$ on its vertex set, where a vertex $s \in V(T)$ is said to be a *descendent* of a vertex $t \in V(T)$, denoted $s \leq t$, if t is on the (unique) path from s to r in T.

To capture the parity of lengths of paths in a robust manner, we use graphs with edges labeled with elements of $\mathbb{F}_2$. Let G be a graph where every edge $e \in E(G)$ is assigned an element $\lambda(e) \in \mathbb{F}_2$. With a walk W in G we can associate then the sum of the elements assigned to the edges on W (with multiplicities, i.e., if an edge e appears c times in W, then we add $c \cdot \lambda(e)$ to the sum). An important observation is that if in G in every closed walk the edge labels sum up to 0, then for every $u, v \in V(G)$, in every walk from u to v the edge labels sum up to the same value, depending only on u and v. Furthermore, one can in linear time (a) check if every closed walk in G sums up to 0 and, if this is the case, (b) compute for every u a value $x_u \in \mathbb{F}_2$, called henceforth the *potential*, such that for every $u, v \in V(G)$ and every walk W from u to v in G, the sum of the labels of W equals $x_v - x_u$. Indeed, it suffices to take any rooted spanning forest F of G, define x_u to be the sum of the labels on the path from u to the root of the corresponding tree in F, and check for every $uv \in E(G)$ if $\lambda(uv) = x_v - x_u$.

▶ **Definition 4** (Nice tree decomposition). *A nice tree decomposition of a graph G is a rooted tree decomposition $(T, \{X_t\}_{t \in V(T)})$ such that:*

- *the root and leaves of T have empty bags; and*
- *other nodes are of one of the following types:*
  - ***Introduce vertex node**: a node t with exactly one child t' such that $X_t = X_{t'} \cup \{v\}$ with $v \notin X_{t'}$. We say that v is introduced at t;*
  - ***Forget vertex node**: a node t with only one child t' such that $X_t = X_{t'} \setminus \{v\}$ with $v \in X_{t'}$. We say that v is forgotten at t; and*
  - ***Join node**: a node t with two children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$.*

For each node t of the decomposition, we define a partial *graph $G_t = G\left[\bigcup_{s \leq t} X_s\right] - E(G[X_t])$.*

Note that edges of partial graphs appear at forget vertex nodes and that they correspond to adding edges between the forgotten vertex and its neighbours.

From a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width k, a nice tree decomposition of width k with $\mathcal{O}(k|V(G)|)$ nodes can be computed in time $\mathcal{O}(k^2 \cdot \max(|V(T)|, |V(G)|))$, see [21].

A *k-labeled* graph is a graph G together with a labeling function $\Gamma : V(G) \to [k]$. For k-labeled graphs H, G, and integers $i, j \in [k]$, we consider the following operations:

- **Vertex creation:** $i(v)$ is the k-labeled graph consisting of a single vertex v with label i;
- **Disjoint union:** $H \oplus G$ is the k-labeled graph consisting of the disjoint union of H and G;
- **Join:** $\eta_{i \times j}(G)$ is the k-labeled graph obtained by adding an edge between any pair of vertices one being of label i, the other of label j, if the edge does not exist; and
- **Renaming label:** $\rho_{i \to j}(G)$ is the k-labeled graph obtained by changing the label of every vertex labeled i to label j: $\forall v \in \Gamma^{-1}(\{i\}),\ \Gamma(v) := j$

The *clique-width* of a graph G, denoted $\mathbf{cw}(G)$, is the least integer k such that a k-labeled graph isomorphic to G can be constructed using these operations. We call *k-expression* of a k-labeled graph G a sequence of operations that leads to the construction of G. Note that such a sequence defines a tree, called *tree associated to the k-expression* in the following.

Consider a k-expression of a k-labeled graph G, and its associated tree T. For a node $t \in V(T)$, we denote by T_t the subtree of T rooted at t, and associate it with the labeled graph G_t it describes. For an integer $i \in [k]$, we denote by $V_i(G)$ the set of vertices of label i in G. By an abuse of notations in the following, by "label i" for a labeled graph G we may refer to both the integer i, or the set $V_i(G)$.

We define *partially k-labeled graphs* as labeled graphs with a labeling function $\Gamma(G) : V(G) \to [k] \cup \{\perp\}$ and call *unlabeled* the vertices of $\Gamma^{-1}(\{\perp\})$.

Given a graph G and a set of vertices $S \subseteq V(G)$, we call S-vertex a vertex that is part of S and we call S-path (resp. S-cycle) a path that contains at least one S-vertex.

When a 2-connected multigraph contains a cycle, we call it *nontrivial*. Other 2-connected multigraphs are the degenerate cases of a single vertex and a bridge, i.e., two vertices connected by a single edge. A *nontrivial* 2-connected component of a multigraph is a 2-connected component which is a nontrivial 2-connected multigraph, it is not an isolated vertex or a bridge.

Since our algorithms solve weighted variants of the problems, we will denote by $c : V(G) \to \mathbb{Z} \cup \{+\infty\}$ the weight function of the instance. We extend this notation to sets of vertices with $c(U) = \sum_{v \in U} c(v)$. The unweighted variant corresponds to having $c(v) = 1$ for all $v \in V(G)$.

In the context of a dynamic programming algorithm, a state is a tuple of parameters used to index the table in which computations are done. We denote our table by d. We call transition from a set of states $\mathcal{A}$ to a single state B the action of updating the entry indexed by B based on the values of states in $\mathcal{A}$. Since we consider only minimizing problems, for a function f, such a transition will consist in applying the operation

$$d[B] := \min\{d[B], f(\mathcal{A})\}.$$

We denote this operation by $d[B] \leftarrow f(\mathcal{A})$ and say that value $f(\mathcal{A})$ is propagated to state B.

3 Hitting even cycles in graphs of bounded treewidth

In this section we highlight the main ideas behind Theorem 1. Rather than simply giving an algorithm for just SOCT and SECT, we also show how our method gives less involved algorithms for SFVS and ECT. All these problems can be seen as looking for a minimum deletion set such that the resulting graph has no odd S-cycle, no even S-cycle, no S-cycle, and no even cycle. In order to have a common notation, we will call $\square$-cycles the cycles that have to be hit in the problem and $\square$-cycle-free the graphs that do not contain $\square$-cycles.

To transform a $\square$-cycle-free graph into a forest, we will replace its nontrivial 2-connected components with tree structures. We use labeled vertices to store efficiently the properties of these nontrivial 2-connected components.

We begin by giving a characterisation of nontrivial 2-connected $\square$-cycle-free graphs for each problem. This implies characterisations of $\square$-cycle-free graphs.

▶ **Lemma 5.** *Let G be a nontrivial 2-connected multigraph.*

1. *G contains no S-cycle if and only if it contains no S-vertex.*
2. *G contains no even cycle if and only if it is an odd cycle.*
3. *G contains no odd S-cycle if and only if it has one of the following forms:*
   - *G contains no S-vertex and is not bipartite*
   - *G contains no S-vertex and is bipartite*
   - *G contains at least one S-vertex and is bipartite.*

4. *G contains no even S-cycle if and only if it has of one of the following forms:*
   - *G contains no S-vertex and is not bipartite*
   - *G contains no S-vertex and is bipartite*
   - *G contains at least one S-vertex, the connected components of $G - S$ are bipartite, together with S-vertices they form a cycle: each S-vertex has degree 2 and each connected component of $G - S$ has degree 2. One S-cycle is odd. We later call* bipartite subcomponents *the connected components of $G - S$. This is illustrated in Figure 1a.*

The first point is immediate, the second follows from [26, Lemma 1], and the third was observed in [5, Lemma 13]. The last point was not known to us and we provide a proof.

Proof. Suppose that G is a nontrivial 2-connected multigraph containing no even S-cycle.

If G contains no S-vertex, it is either bipartite or not, leading to the first possible forms.

If G contains an S-vertex, it must contain an S-cycle C due to being a nontrivial 2-connected multigraph and C is odd because G contains no even S-cycle.

▷ Claim 6. If two vertices are connected by three disjoints paths at least two of which are S-paths then two of the paths form an even S-cycle.

Proof. The three cycles formed by combining the paths are S-cycles and they cannot all be odd: if we denote them C_1, C_2, C_3, $|C_3| = |C_1| + |C_2| - 2|C_1 \cap C_2|$. ◁

Consider a connected component A of $G - V(C)$.

Consider an S-vertex v of A, because G is a nontrivial 2-connected multigraph, there exist two disjoint paths that connect v to distinct vertices a, b of C, a and b satisfy the conditions of claim 6 leading to a contradiction. Hence, A cannot contain an S-vertex.

Since G is a nontrivial 2-connected multigraph, there are at least 2 edges between A and distinct vertices of C. Consider 2 arbitrary distinct such edges, they cut C into two paths P_1 and P_2 with extremities u and v. Since A is connected, there is a third u–v path P_3 through A. Only one of P_1 and P_2 may contain an S-vertex, by claim 6 applied to P_1, P_2, P_3. In particular, u and v cannot be S-vertices, this implies that the only edges incident to S-vertices in G are edges of cycle C, so S-vertices have degree 2.

Let $\widetilde{A}$ be the connected component of $G - S$ containing A. $\widetilde{A}$ contains a maximal S-free path of C because A is connected to C and cannot be adjacent to S-vertices. $\widetilde{A}$ contains only one maximal S-free path of C because otherwise either we get two edges from A to C that separate C in two S-paths and this was excluded in the previous paragraph, or we have a chord ab in C that connects two distinct maximal S-free-paths and this is excluded by Claim 6. In particular, note that this shows that $\widetilde{A}$ has outdegree 2 in G.

Consider a cycle C' of $\widetilde{A}$, then there are 2 disjoint paths from it to the S-vertices adjacent to $\widetilde{A}$. If they are distinct we can connect them with a disjoint path via C. This constructions contains two S-cycles C and $C \Delta C'$ which must both be odd so C' can only be even. Hence $\widetilde{A}$ contains no odd cycle so it is bipartite.

We can conclude that all connected components of $G - S$ are bipartite and that together with S-vertices they form a cycle.

Conversely, if G contains no S-vertex it does not contain any even S-cycle. If it is a cycle of bipartite components and S-vertices with one S-cycle C being odd, then each S-cycle C' goes through all bipartite components and S-vertices. Replacing the path of C by the path of C' in each bipartite component preserves parity because endpoints are unique. We conclude that all S-cycles are odd in G. ◀

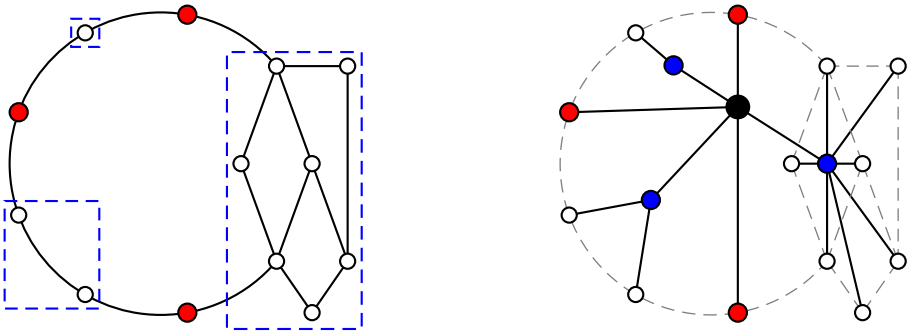
▶ **Definition 7.** *Given a $\square$-cycle-free graph G, we define its* underlying forest $F(G)$ *as the graph obtained from G by modifying independently each nontrivial 2-connected component C as follows:*

SFVS. *Remove edges inside C and add an unlabeled vertex adjacent to all vertices of C.*

ECT. *Remove edges inside C and add a vertex adjacent to all vertices of C and label it "odd cycle".*

SOCT. *Remove edges inside C and add a vertex adjacent to all vertices of C, label it "bipartite" or "not bipartite" based on the property of C and make it an S-vertex if C contains an S-vertex.*

SECT. *In the two first forms we remove edges inside C and add a vertex adjacent to all vertices of C and label it "bipartite" or "not bipartite" based on the property of C. For the last form, for each bipartite subcomponent of C, we remove its edges, add a vertex labeled "internal bipartite" adjacent to its vertices. Then remove edges of C incident to S, add an S-vertex labeled "odd cycle" adjacent to S-vertices and vertices labeled "internal bipartite". This is illustrated in Figure 1b.*



(a) An example of graph with no even S-cycle. The vertices of S are depicted in red. The blue boxes denote the bipartite subcomponents.

(b) The underlying forest we build from the graph on Figure 1a. The "internal bipartite" vertices are depicted in blue and the "odd cycle" vertex is black.

Figure 1 The last form of SECT: "internal bipartite" vertices.

Observe that, because labeled vertices are only introduced by this underlying forest, to each labeled vertex v, we can associate a nontrivial 2-connected component C: the one that resulted in the creation of v. Observe also that for a path P between two unlabeled vertices, if it contains a labeled vertex, then it contains a vertex of its associated component before it on P and another vertex of its associated component after it on P.

We now introduce reduction rules that allow us to maintain a simplified description of underlying forests relatively to a given subset of vertices, that we call *active*. Vertices that are not active are called *inactive*. Theses rules and this terminology largely resemble what is done in [5].

▶ **Definition 8.** *Given a $\square$-cycle-free graph G, its underlying forest $F(G)$ and subset of active vertices X, a reduced underlying forest F_r is obtained by applying exhaustively the following rules on $F(G)$:*

- *Delete inactive vertices of degree at most one.*
- *For each maximal path P with internal inactive vertices of degree 2, we replace it with a path P' with same endpoints, such that P' contains exactly one occurrence of each label present in P and a single S-vertex if P contained one, where endpoints are considered to be contained in P and P'.*

For SECT we add another rule: if a maximal path with internal inactive vertices of degree 2 contains at least 2 vertices labeled "internal bipartite" but no vertex labeled "odd cycle", we keep 2 occurences of the label "internal bipartite".

The set of reduced underlying forests obtained from $F(G)$ with active vertices X is denoted $F(G, X)$.

Observe that a reduced forest is not unique, however properties that we will show on them will not depend on the choice of representative.

Standard arguments show that a reduced forest has bounded size.

▶ **Lemma 9.** *In a problem using K label symbols (including S-membership), $F \in F(G, X)$ has at most $(K+1)(2|X|-2)+1$ vertices.*

The crucial property preserved by a reduced forest is the following.

▶ **Lemma 10.** *For $F \in F(G, X)$, for each pair of active vertices u and v, there is a path between them in G if and only if there is a path between them in F. For each type of nontrivial 2-connected component, a u–v path in G goes through at least one such component if and only if the u–v path in F contains a vertex with the corresponding label symbol ("internal bipartite" counts for the bipartite subcomponent but also the S-cycle containing it). There exists a u–v path in G containing an S-vertex if and only if there exists a u–v path in F containing an S-vertex or a vertex labeled "internal bipartite". If there is a u–v path in F, every unlabeled vertex that is on the u–v path in F is also on all u–v paths in G.*

A property that is not preserved by a reduced forest is the length of paths. Since we are only interested in parity, we maintain a $\mathbb{F}_2$-labeling α of edges. We say that α is a *valid* $\mathbb{F}_2$-labeling of $F \in F(G, X)$ if, there exists β a $\mathbb{F}_2$-labeling of the edges of $F(G)$ such that edges incident to vertices labeled "bipartite" or "internal bipartite" are labeled 0 for one side of the bipartition and 1 for the other side, edges incident to other labeled vertices are labeled 0, and edges between unlabeled vertices are labeled 1, and for each edge uv of F, its label is the sum of labels on the edges of the u–v path in $F(G)$. During the application of reduction rules, each edge is given as its label the sum of labels of the path that was connecting its endpoints.

▶ **Lemma 11.** *For $F \in F(G, X)$, for each pair of active vertices u and v connected in G, all u–v paths in G have same parity if and only if the path between u and v in F contains no vertex with label symbol "odd cycle", "not bipartite" or "internal bipartite". Furthermore, when this condition is satisfied, the parity of the paths in G is given by the sum of labels on the edges of F.*

The main technical engine of our algorithms is the following join operation.

▶ **Lemma 12.** *There exists a polynomial-time algorithm that, for every pair of $\square$-cycle-free graphs G_1 and G_2 with $V(G_1) \cap V(G_2) = X$, given on input two reduced forests with valid $\mathbb{F}_2$-labelings (F_1, α_1) and (F_2, α_2), with $F_1 \in F(G_1, X)$ and $F_2 \in F(G_2, X)$, decides whether $G_1 \cup G_2$ is $\square$-cycle-free and, in case of a positive answer, computes a reduced forest $F \in F(G_1 \cup G_2, X)$ and, except for the SFVS problem, a valid $\mathbb{F}_2$-labeling α.*

With Lemma 12 in hand, assembling a dynamic programming algorithm of Theorem 1 is a tedious but straightforward exercise. The proof of Lemma 12 requires a careful consideration of all possible 2-connected components the graph $F_1 \cup F_2$ may contain and how to treat them in the considered problems.

4 Subset Feedback Vertex Set in graphs of bounded cliquewidth

We describe a dynamic programming algorithm to solve Subset Feedback Vertex Set on clique-width expressions. With a bottom-up computation, it builds small labeled forests that describe the graphs that can be obtained by vertex deletion.

A state of our dynamic programming will consist of a node of the k-expression, a partially labeled forest, and a label state assignment $\mathcal{P} : [k] \to \mathcal{Q}$, with $\mathcal{Q} = \{Q_\varnothing, Q_1, Q_1^*, Q_2, Q_w, Q_w^*, Q_f\}$ the set of label states. State $Q_\varnothing$ is assigned to labels that are completely contained in the current deletion set. States Q_1 and Q_1^* are assigned to labels consisting of a single non-S-vertex, or a single S-vertex, respectively. States Q_w and Q_w^* are called *waiting states*: they are assigned to labels for which we have guessed that they will be joined (only once) to a non-S-vertex from a label in state Q_1, or to an S-vertex from a label in state Q_1^*, respectively. State Q_2 is assigned to labels having at least two vertices not in S: it is assigned to labels for which we have guessed that they will be joined (potentially several times) to either a vertex from a label in state Q_1, or to vertices from a label in state Q_2. These guessing tricks can be seen as a form of what is called "expectation from the outside" in [9]. We point that guessing these joins implies that labels in states Q_w, Q_w^*, Q_2 will eventually be connected – this is detailed below. At last, state Q_f is called *final state*: it will contain vertices that will not be joined anymore, and hence that may be unlabeled. To summarize, states in $\mathcal{Q}$ express the following constraints on joins:

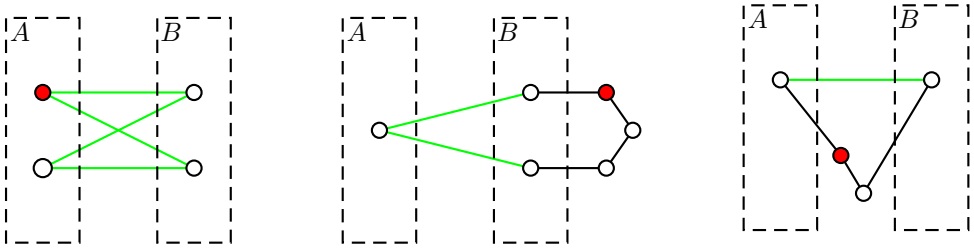
- joins with a label in state $Q_\varnothing$ will be ignored;
- no join with a label in state Q_f will be performed;
- labels in state Q_w (resp. Q_w^*) will only be joined with those in state Q_1 (resp. Q_1^*); and
- labels in state Q_2 will never be joined with those in state Q_1^*.

Now, considering an S-cycle-free graph $\widetilde{G}$ obtained by vertex deletion, we will say that a label i is *compatible* with label state:

- $Q_\varnothing$ if no vertex of $\widetilde{G}$ is labeled i;
- Q_1 if exactly one vertex of $\widetilde{G}$ is labeled i, and it is not in S;
- Q_1^* if exactly one vertex of $\widetilde{G}$ is labeled i, and it is in S;
- Q_2 if at least two vertices of $\widetilde{G}$ are labeled i, they are not in S, and no S-path in $\widetilde{G}$ has both its endpoints labeled i;
- Q_w if at least two vertices of $\widetilde{G}$ are labeled i, at least one S-vertex is labeled i, and no S-path in $\widetilde{G}$ has both its endpoints labeled i;
- Q_w^* if at least two vertices of $\widetilde{G}$ are labeled i, no path in $\widetilde{G}$ has both its endpoints labeled i; and
- Q_f if at least two vertices of $\widetilde{G}$ are labeled i.

These conditions, together with the constraints on joins that are expressed above, aim to capture cases for which a join between labels of pairs of label states will not create S-cycles – this will be explicited in proofs and illustrated in Figure 2. In the following, we say that a label state assignment $\mathcal{P}$ is *compatible* with $\widetilde{G}$ if each label is compatible with its state in this graph. Note that looking at the properties of vertices in a label in part gives the label state assignment that it should have: the conflicts are for choosing between Q_f, Q_w^* and, based on the presence or not of an S-vertex, either Q_w or Q_2. This is expected because these states contain the information on a guess on what will later be added to the graph.

Let us now introduce an auxiliary partially labeled graph which will conveniently represent the connectedness implied by guesses we made so far when assigning labels to label states, while simplifying the manipulation of labels. We point that this auxiliary graph will *not* be computed by the algorithm: it shall only be used in the proofs. Given a labeled graph $\widetilde{G}$



(a) Case 1: $|A| \geq 2$, $|B| \geq 2$, and $|A \cap S| + |B \cap S| \geq 1$.

(b) Case 2: $|A| \geq 1$, $|B| \geq 2$, and there is an S-path with endpoints in B.

(c) Case 3: $|A| \geq 1$, $|B| \geq 1$, and there is an S-path connecting a vertex of A to one of B.

Figure 2 The three cases when a join (depicted in green) creates an S-cycle. The figures illustrates the smallest number of vertices of S required. Thus, up to symmetry, the vertices depicted in red have to be in S while the vertices in white may or may not be in S.

and a label state assignment $\mathcal{P}$, we denote by $H(\widetilde{G}, \mathcal{P})$ the partially labeled graph obtained from $\widetilde{G}$, by conducting the following modifications for each label i:

- if i is in state Q_2 or Q_w, we add a vertex labeled i, connect it to other vertices labeled i, and unlabel these vertices, making the added vertex the only vertex labeled i;
- if i is in state Q_w^*, we add an S-vertex labeled i, connect it to other vertices labeled i, and unlabel these vertices, making the added vertex the only vertex labeled i; and
- if i is in state Q_f, we unlabel vertices labeled i.

Note that in the auxiliary graph, we add vertices that are not part of the original graph. The role of these vertices – for states Q_2, Q_w, and Q_w^* – is to represent the label i as if it was connected (which will eventually be the case as we guessed a later join), as well as manipulating nonempty labels as single vertices: for $\widetilde{G}$ compatible with $\mathcal{P}$, each nonempty label i contains exactly one vertex in $H(\widetilde{G}, \mathcal{P})$, which we call *representative* of i in $H(\widetilde{G}, \mathcal{P})$, and that we denote by $h(i)$.

Recall that, when $\mathcal{P}$ is compatible with $\widetilde{G}$, some connectedness conditions are satisfied by label states. We say that a partially labeled multigraph $\widehat{F}$ *expresses the connectedness* in $H(\widetilde{G}, \mathcal{P})$, for $\widetilde{G}$ and $\mathcal{P}$ compatible, if:

- for each label i, there is at most one vertex labeled i in $\widehat{F}$;
- to every vertex $h(i)$ in $H(\widetilde{G}, \mathcal{P})$ corresponds a vertex $r(i)$ labeled i in $\widehat{F}$: we call it the *representative* of label i in $\widehat{F}$, and $r(i)$ is an S-vertex if and only if $h(i)$ is an S-vertex; and
- for any two vertices $h(i), h(j)$ in $H(\widetilde{G}, \mathcal{P})$, there exists a $h(i)$–$h(j)$ path in $H(\widetilde{G}, \mathcal{P})$ if and only if there exists a $r(i)$–$r(j)$ path in $\widehat{F}$, and there exists a $h(i)$–$h(j)$ S-path in $H(\widetilde{G}, \mathcal{P})$ if and only if there exists a $r(i)$–$r(j)$ S-path in $\widehat{F}$.

We are now ready to introduce reduction rules which, when applied on the multigraph $\widehat{F}$ expressing the connectedness in $H(\widetilde{G}, \mathcal{P})$, will produce the aforementioned partially labeled forest. The idea behind this forest is that, to check the existence of (S-)paths linking representatives of labels i and j, unlabeled vertices of degree at most two in such (S-)paths may be "contracted" as long as we do not remove all (S-)vertices on these paths. In the following for a partially labeled multigraph $\widehat{F}$, we denote by $\mathrm{Red}(\widehat{F})$ the forest obtained from $\widehat{F}$ by applying the following reduction rules:

- for each nontrivial 2-connected component C, we introduce an unlabeled vertex, call it *central vertex* of C, connect it to vertices of C, and remove all other edges inside the component;

- we iteratively remove unlabeled vertices of degree at most one;
- for each maximal S-path with internal unlabeled vertices of degree two, we replace it by connecting the endpoints to a single new unlabeled S-vertex; and
- for each maximal path with internal unlabeled vertices of degree two that is not an S-path, we replace it by a single edge between its endpoints.

It is easily seen that the produced graph is indeed a forest as the graph of nontrivial 2-connected components of any graph is a tree, and each nontrivial 2-connected component is replaced by a star. Furthermore, reducing F preserves the fact that it expresses the connectedness in $H(\widetilde{G}, \mathcal{P})$: the formal statement is omitted here. As in Lemma 9, we can bound the size of the reduced forest:

▷ Claim 13. $\mathrm{Red}(\widehat{F})$ has $\mathcal{O}(k)$ vertices.

A *state* of the dynamic programming algorithm is a tuple $(t, F, \mathcal{P})$, where $t \in V(T)$, F is a partially labeled forest, and $\mathcal{P} : [k] \to \mathcal{Q}$ is a label state assignment. We say that $(t, F, \mathcal{P})$ is *admissible* if there exists $X \subseteq V(G)$ such that $\mathcal{P}$ is compatible with $G_t - X$, $H(G_t - X, \mathcal{P})$ is S-cycle-free, and F expresses the connectedness in $H(G_t - X, \mathcal{P})$. Our dynamic programming algorithm will not consider all possible states, but compute a value $d[t, F, \mathcal{P}]$ for some states $(t, F, \mathcal{P})$. We call *reachable* a state that is considered by the algorithm. We will show that reachable states are admissible, that for every $t \in V(T)$, for each $X \subseteq V(G_t)$, if $G_t - X$ is S-cycle-free, then there exists a reachable state $(t, F, \mathcal{P})$ such that $d[t, F, \mathcal{P}] \leq |X|$, and that the optimal value for SFVS on the given instance is the minimum of values $d[r, F, \mathcal{P}]$ where r is the root of the k-expression.

First, let us slightly modify our clique-width expression in order to simplify the description of our computations. We double the set of labels, denoting them by $\{1, ..., k, 1', ..., k'\}$, and replace each disjoint union node t with children t_1, t_2 by the following subexpression: $\rho_{1' \to 1}(\dots \rho_{k' \to k}(G_{t_1} \oplus (\rho_{1 \to 1'}(\dots \rho_{k \to k'}(G_{t_2})))))$. This gives the property that in disjoint union nodes, each label is used by at most one of the children nodes.

We now describe the bottom-up computation of reachable states for each possible type of node in the clique-width expression.

Leaf node. If t is a leaf node with $G_t = i(v)$, two cases arise. Either v is deleted which is described by state $(t, F_\varnothing, \mathcal{P}_\varnothing)$ initialized with value $c(v)$, where $F_\varnothing$ is the empty graph, and $\mathcal{P}_\varnothing$ is the function that maps every $i \in [k]$ to $Q_\varnothing$. Otherwise we keep v, which is described by state $(t, F, \mathcal{P})$ where F consists of the isolated vertex v, $\mathcal{P}(i) = Q_1^*$ if $v \in S$, $\mathcal{P}(i) = Q_1$ otherwise, and, for all $j \neq i$, $\mathcal{P}(j) = Q_\varnothing$.

Join node. Let t be a join node with $G_t = \eta_{i \times j}(G_{t'})$. For each reachable state $(t', F', \mathcal{P}')$, we proceed as follows. If the representatives of i and j are connected by an S-path in F', we do nothing. Otherwise, we will construct states $(t, F, \mathcal{P})$ defined in the following cases, depending on $\mathcal{P}'(i)$ and $\mathcal{P}'(j)$, starting with $F := F'$ and $\mathcal{P} := \mathcal{P}'$:

- if one of i and j is in state $Q_\varnothing$, we do not modify F nor $\mathcal{P}$;
- if i and j are in states Q_1 or Q_1^*, we add an edge between the representatives of i and j in F;
- if i and j are in states Q_1 or Q_2, we add an edge between the representatives of i and j in F, and if i or j are in state Q_2 they are allowed to change to Q_f in $\mathcal{P}$, if they do we also unlabel their representative: we enumerate all possibilities here;
- if i and j are in states Q_1^* and Q_w^*, we identify their representative in F: the resulting vertex has its label in state Q_1^*, and the label in state Q_w^* is assigned state Q_f in $\mathcal{P}$; and
- if i and j are in states Q_1 and Q_w, we identify their representative in F: the resulting vertex has its label in state Q_1, and the label in state Q_w is assigned state Q_f in $\mathcal{P}$.

For each such cases, we reduce F and propagate the value $d[t', F', \mathcal{P}']$ to the states $(t, F, \mathcal{P})$, where $\mathcal{P}$ is the modified label state assignment.

Renaming label node. Let t be a renaming label node with $G_t = \rho_{i\to j}(G_{t'})$. For each reachable state $(t', F', \mathcal{P}')$, we construct states $(t, F, \mathcal{P})$ starting with $\mathcal{P} := \mathcal{P}$ and $F := F'$ by first setting $\mathcal{P}(i) = Q_\varnothing$, and proceeding as follows depending on $\mathcal{P}'(i)$ and $\mathcal{P}'(j)$:

- if i and j are in a state among $\{Q_f, Q_1, Q_1^*\}$, we unlabel the representatives of i and j in F', and set $\mathcal{P}(j) = Q_f$;
- if one of i and j is in state $Q_\varnothing$, then either i is in state $Q_\varnothing$ and we do nothing, or j is in state $Q_\varnothing$, we assign it to the other label state, and the vertex of F labeled i is relabeled j;
- if i and j are in state Q_1, and the representatives of i and j are not connected by a path in F', in F, we add an S-vertex labeled j, connect it to these vertices, and unlabel them. Label j is then assigned state Q_w^* in $\mathcal{P}$;
- if one of i and j is in state Q_1^*, the other is in state Q_1 or Q_1^*, and the representatives of i and j are not connected by a path in F', we consider two possibilities depending on whether they will be joined to a vertex of S, or to a vertex of $V(G) \setminus S$. First, in F, we add a new vertex labeled j, connect it to the representatives of i and j, and unlabel the representatives of i and j. Then, if the new vertex is chosen to be in S, j is assigned state Q_w^* in $\mathcal{P}$. Otherwise, j is assigned state Q_w in $\mathcal{P}$;
- if i and j are in states Q_α and Q_β, for $\alpha, \beta \in \{1, 2, w\}$, and the representatives of i and j are not connected by an S-path in F', in F, we identify the representatives of i and j: the resulting vertex is of label j, and j is assigned state Q_δ in $\mathcal{P}$ with $\delta := \max_{1<2<w} \{2, \alpha, \beta\}$;
- if one of i and j is in state Q_1^*, the other is in state Q_w or Q_2, and the representatives of i and j are not connected by a path in F', in F, we add an edge between the representatives of i and j, and the representative of the label in state Q_1^* becomes unlabeled, while the other vertex is given label j. Label j is assigned state Q_w in $\mathcal{P}$; and
- if one of i and j is in state Q_w^*, the other is in state Q_1 or Q_1^*, and the representatives of i and j are not connected by a path in F', in F, we add an edge between the representatives of i and j, and the representative of the label in state Q_1 or Q_1^* becomes unlabeled, while the other vertex is given label j. Label j is assigned state Q_w^* in $\mathcal{P}$.

For each such cases, we reduce F, and we propagate the value $d[t', F', \mathcal{P}']$ to the state $(t, F, \mathcal{P})$, where $\mathcal{P}$ is the modified label state assignment.

Disjoint union node. If t is a disjoint union node with $G_t = G_{t_1} \oplus G_{t_2}$, for each pair of reachable states $(t_1, F_1, \mathcal{P}_1)$, $(t_2, F_2, \mathcal{P}_2)$, since they use disjoint sets of labels, we can simply define $F = F_1 \oplus F_2$. The label state assignment $\mathcal{P}$ is defined by $\mathcal{P}(i) = \mathcal{P}_1(i)$ for $i \in [k]$ and $\mathcal{P}(i') = \mathcal{P}_2(i')$ for $i' \in \{1', \ldots, k'\}$. The value $d[t_1, F_1, \mathcal{P}_1] + d[t_2, F_2, \mathcal{P}_2]$ is propagated to state $(t, F, \mathcal{P})$.

We conclude the section noting that, with the above described transitions, proving the correctness of the algorithm by induction is a tedious, but rather straightforward exercise. It basically consists on considering all the different label states in which labels i and j may lie when performing join and relabel operations, together with the compatibility of the current S-cycle-free graph and label states assignments.

References

1 Julien Baste, Ignasi Sau, and Dimitrios M. Thilikos. Hitting minors on bounded treewidth graphs. IV. an optimal algorithm. *CoRR*, abs/1907.04442, 2019. arXiv:1907.04442.

2 Julien Baste, Ignasi Sau, and Dimitrios M. Thilikos. Hitting minors on bounded treewidth graphs. I. general upper bounds. *SIAM J. Discret. Math.*, 34(3):1623–1648, 2020. doi:10.1137/19M1287146.

3 Julien Baste, Ignasi Sau, and Dimitrios M. Thilikos. Hitting minors on bounded treewidth graphs. II. single-exponential algorithms. *Theor. Comput. Sci.*, 814:135–152, 2020. doi:10.1016/j.tcs.2020.01.026.

4 Julien Baste, Ignasi Sau, and Dimitrios M. Thilikos. Hitting minors on bounded treewidth graphs. III. lower bounds. *J. Comput. Syst. Sci.*, 109:56–77, 2020. doi:10.1016/j.jcss.2019.11.002.

5 Benjamin Bergougnoux, Édouard Bonnet, Nick Brettell, and O-joung Kwon. Close Relatives of Feedback Vertex Set Without Single-Exponential Algorithms Parameterized by Treewidth. In Yixin Cao and Marcin Pilipczuk, editors, *15th International Symposium on Parameterized and Exact Computation (IPEC 2020)*, volume 180 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.IPEC.2020.3.

6 Benjamin Bergougnoux and Mamadou Moustapha Kanté. Fast exact algorithms for some connectivity problems parameterized by clique-width. *Theor. Comput. Sci.*, 782:30–53, 2019. doi:10.1016/j.tcs.2019.02.030.

7 Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Inf. Comput.*, 243:86–111, 2015. doi:10.1016/j.ic.2014.12.008.

8 Marthe Bonamy, Lukasz Kowalik, Jesper Nederlof, Michal Pilipczuk, Arkadiusz Socala, and Marcin Wrochna. On directed feedback vertex set parameterized by treewidth. In Andreas Brandstädt, Ekkehard Köhler, and Klaus Meer, editors, *Graph-Theoretic Concepts in Computer Science - 44th International Workshop, WG 2018, Cottbus, Germany, June 27-29, 2018, Proceedings*, volume 11159 of *Lecture Notes in Computer Science*, pages 65–78. Springer, 2018. doi:10.1007/978-3-030-00256-5_6.

9 Binh-Minh Bui-Xuan, Ondrej Suchý, Jan Arne Telle, and Martin Vatshelle. Feedback vertex set on graphs of low clique-width. *Eur. J. Comb.*, 34(3):666–679, 2013. doi:10.1016/j.ejc.2012.07.023.

10 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

11 Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Handle-rewriting hypergraph grammars. *J. Comput. Syst. Sci.*, 46(2):218–270, 1993. doi:10.1016/0022-0000(93)90004-G.

12 Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory Comput. Syst.*, 33(2):125–150, 2000. doi:10.1007/s002249910009.

13 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

14 Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast hamiltonicity checking via bases of perfect matchings. *J. ACM*, 65(3):12:1–12:46, 2018. doi:10.1145/3148227.

15 Marek Cygan, Dániel Marx, Marcin Pilipczuk, and Michal Pilipczuk. Hitting forbidden subgraphs in graphs of bounded treewidth. *Inf. Comput.*, 256:62–82, 2017. doi:10.1016/j.ic.2017.04.009.

16 Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on*

Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011, pages 150–159. IEEE Computer Society, 2011. doi:10.1109/FOCS.2011.23.

17 Samuel Fiorini, Nadia Hardy, Bruce A. Reed, and Adrian Vetta. Planar graph bipartization in linear time. *Discret. Appl. Math.*, 156(7):1175–1180, 2008. doi:10.1016/j.dam.2007.08.013.

18 Fedor V. Fomin, Petr A. Golovach, Daniel Lokshtanov, and Saket Saurabh. Intractability of clique-width parameterizations. *SIAM J. Comput.*, 39(5):1941–1956, 2010. doi:10.1137/080742270.

19 Fedor V. Fomin, Petr A. Golovach, Daniel Lokshtanov, and Saket Saurabh. Almost optimal lower bounds for problems parameterized by clique-width. *SIAM J. Comput.*, 43(5):1541–1563, 2014. doi:10.1137/130910932.

20 Fedor V. Fomin, Petr A. Golovach, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. Clique-width III: hamiltonian cycle and the odd case of graph coloring. *ACM Trans. Algorithms*, 15(1):9:1–9:27, 2019. doi:10.1145/3280824.

21 Ton Kloks. *Treewidth: computations and approximations*, volume 842. Springer Science & Business Media, 1994.

22 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 777–789. SIAM, 2011.

23 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. In Dana Randall, editor, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 760–776. SIAM, 2011. doi:10.1137/1.9781611973082.60.

24 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. *ACM Trans. Algorithms*, 14(2):13:1–13:30, 2018. doi:10.1145/3170442.

25 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. *SIAM J. Comput.*, 47(3):675–702, 2018. doi:10.1137/16M1104834.

26 Pranabendu Misra, Venkatesh Raman, MS Ramanujan, and Saket Saurabh. Parameterized algorithms for even cycle transversal. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 172–183. Springer, 2012.

27 Michal Pilipczuk. Problems parameterized by treewidth tractable in single exponential time: A logical approach. In Filip Murlak and Piotr Sankowski, editors, *Mathematical Foundations of Computer Science 2011 - 36th International Symposium, MFCS 2011, Warsaw, Poland, August 22-26, 2011. Proceedings*, volume 6907 of *Lecture Notes in Computer Science*, pages 520–531. Springer, 2011. doi:10.1007/978-3-642-22993-0_47.

28 Neil Robertson and Paul D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. doi:10.1016/0095-8956(84)90013-3.

29 Ignasi Sau and Uéverton dos Santos Souza. Hitting forbidden induced subgraphs on bounded treewidth graphs. In Javier Esparza and Daniel Král', editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 82:1–82:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.MFCS.2020.82.

30 Egon Wanke. k-nlc graphs and polynomial algorithms. *Discret. Appl. Math.*, 54(2-3):251–266, 1994. doi:10.1016/0166-218X(94)90026-4.