Parameterized Complexity of the Sparsest k-Subgraph Problem in Chordal Graphs [⋆]

Marin Bougeret, Nicolas Bousquet, Rodolphe Giroudeau, and Rémi Watrigant

LIRMM, Université Montpellier 2, France

Abstract. In this paper we study the Sparsest k-Subgraph problem which consists in finding a subset of k vertices in a graph which induces the minimum number of edges. The Sparsest k-Subgraph problem is a natural generalization of the Independent Set problem, and thus is $\mathcal{NP}$-hard (and even $W[1]$-hard) in general graphs. In this paper we investigate the parameterized complexity of both Sparsest k-Subgraph and Densest k-Subgraph in chordal graphs. We first provide simple proofs that Densest k-Subgraph in chordal graphs is FPT and does not admit a polynomial kernel unless $\mathcal{NP} \subseteq co\mathcal{NP}/poly$ (both parameterized by k). More involved proofs will ensure the same behavior for Sparsest k-Subgraph in the same graph class.

1 Introduction

Presentation of the problem. Given a simple undirected graph $G = (V, E)$ and an integer k, the Sparsest k-Subgraph problem asks to find k vertices in G inducing the minimum number of edges. The decision version asks if there exists a k-subgraph inducing at most C edges. As a generalization of the classical independent set problem (for $C = 0$), Sparsest k-Subgraph is $\mathcal{NP}$-hard in general graphs, as well as $W[1]$-hard when parameterized by k (as independent set is $W[1]$-hard [11]). In addition, there is an obvious XP algorithm for Sparsest k-Subgraph when parameterized by k, as all subsets of size k can be enumerated in $\mathcal{O}(n^k)$ time, where n is the number of vertices in the graph.

Related problems. Several problems closely related to Sparsest k-Subgraph have been extensively studied in the past decades. Among them, one can mention the maximization version of Sparsest k-Subgraph, namely the Densest k-Subgraph, for which several results have been obtained in general or restricted graphs. In [9], the authors showed that Densest k-Subgraph remains $\mathcal{NP}$-hard in bipartite, comparability and chordal graphs, and is polynomial-time solvable in trees, cographs, and split graphs. The complexity status of Densest k-Subgraph in interval graphs, proper interval graphs and planar graphs is left as an open problem, and is still not answered yet. More recently, [6] improved some of these results by showing that both Densest k-Subgraph and Sparsest k-Subgraph are polynomial-time solvable in bounded cliquewidth

[⋆] This work has been funded by grant ANR **2010** BLAN **021902** graphs, and [3] developed exact algorithms for Sparsest k-Subgraph, Densest k-Subgraph and other similar problems in general graphs parameterized by k and the maximum degree Δ of the graph. During the past two decades, a large amount of work has been dedicated to the approximability of Densest k-Subgraph in general graphs. So far, the best approximation ratio is $O(n^{\delta})$ for some $\delta < 1/3$ [10], while the only negative result is due to Khot [15] ruling out a PTAS under some complexity assumptions. Still concerning Densest k-Subgraph but in restricted graph classes, [17] developed a PTAS in interval graphs, and [8, 16] developed constant approximation algorithms in chordal graphs. In [18], we recently proved that Sparsest k-Subgraph remains $\mathcal{NP}$-hard in chordal graphs and admits a 2-appoximation algorithm. We can also mention the dual version of Sparsest k-Subgraph, namely the maximum partial vertex cover problem, for which we are looking for k vertices in the input graph which *cover* the maximum number of edges. Very recently [1] and [14] independently proved the $\mathcal{NP}$-hardness of maximum partial vertex cover in bipartite graphs, which directly transfers to Sparsest k-Subgraph (since finding k vertices covering the maximum number of edges is equivalent to find $(n-k)$ vertices inducing the minimum number of edges).

More generally, Sparsest k-Subgraph, Densest k-Subgraph and maximum partial vertex cover fall into the family of *cardinality constrained optimization problems* introduced by Cai [7]. In its survey, the author proved that these three problems are $W[1]$-hard in regular graphs, and gives an XP algorithm for general graphs with a better running time than the trivial algorithm.

As said previously, Sparsest k-Subgraph and Densest k-Subgraph are natural generalizations of k-independent set and k-clique, and are thus important both from a theoretical and practical point of view. Our motivation is to study their computational (parameterized) complexity in graph classes where they remains $\mathcal{NP}$-hard whereas k-independent set and k-clique are polynomial-time solvable, such as the well-known class of perfect graphs and some of its subclasses. To that end, we study their parameterized complexity in the class of chordal graphs, an important subclass of perfect graphs which arises in many practical situations [13]. More precisely, we prove that both Sparsest k-Subgraph and Densest k-Subgraph in chordal graphs are fixed-parameter tractable and do not admit a polynomial kernel under some classical complexity assumptions. As we will see, the results are quite easy to obtain for Densest k-Subgraph, but require some efforts for Sparsest k-Subgraph.

Organization of the paper. The paper is organized as follows: in the following section (Section 2), we recall the classical definitions of parameterized complexity and chordal graphs. Our two main results, namely the FPT algorithm and kernel lower bound for Sparsest k-Subgraph in chordal graphs, are presented respectively in Sections 4 and 5. Before all these, we study as an appetizer the parameterized complexity of Densest k-Subgraph in chordal graphs in Section 3. Due to space restrictions, some proofs and figures were omitted. They can be found in the long version of the paper available in [5].

2 Parameterized Algorithms, Chordal Graphs

Parameterized algorithms. An interesting way to tackle $\mathcal{NP}$-hard problems is parameterized complexity. A parameterized problem Q is a subset of $\Sigma^* \times \mathbb{N}$, where the second component is called the *parameter* of the instance. A *fixed-parameter tractable* (FPT for short) problem is a problem for which there exists an algorithm which, given $(x, k) \in \Sigma^* \times \mathbb{N}$, decides whether $(x, k) \in Q$ in time $f(k)|x|^{O(1)}$ for some computable function f. Such an algorithm becomes efficient with an hopefully small parameter. A *kernel* is a polynomial algorithm which, given $(x, k) \in \Sigma^* \times \mathbb{N}$, outputs an instance (x', k') such that $(x, k) \in Q \Leftrightarrow (x', k') \in Q$ and $|x'| + k' \leq f(k)$ for some computable function f. The existence of a kernel is equivalent to the existence of an FPT-algorithm. Nevertheless one can ask the function f to be a polynomial. If so, then the kernel is called a *polynomial kernel*. If a problem admits a polynomial kernel, then it roughly means that we can, in polynomial time, compress the initial instance into an instance of size $poly(k)$ which contains all the hardness of the instance. In order to rule out polynomial kernels, we will use the recent technique of cross-composition [2].

Roughly speaking, a cross-composition is a polynomial reduction from t instances of a (non-parameterized) problem A to a single instance of a parameterized problem B such that the constructed instance is positive iff one of the input instances is positive. In addition, the parameter of the constructed instance must be of size polynomial in the maximum size of the input instances and the logarithm of t. It is known that if A is $\mathcal{NP}$-hard and A cross-composes into B, then B cannot admit a polynomial kernel under some complexity assumptions. For a stronger background concerning the parameterized complexity in general and to cross-compositions in particular, we refer the reader respectively to [11, 2].

Chordal graphs. A graph $G = (V, E)$ is a chordal graph if it does not contain an induced cycle of length at least four. As said previously, chordal graphs form an important subclass of perfect graphs. One can also equivalently define chordal graphs in terms of a special tree decomposition. Indeed, it is known [12] that a graph $G = (V, E)$ is a chordal graph if and only if one can find a tree $T = (\mathcal{X}, A)$ with $\mathcal{X} \subseteq 2^V$ such that for all $v \in V$, the set of nodes of T containing v, that is $\mathcal{X}_v = \{X \in \mathcal{X} : v \in X\}$, induces a (connected) tree, and such that for all $u, v \in V$ we have $\{u, v\} \in E$ if and only if $\mathcal{X}_u \cap \mathcal{X}_v \neq \emptyset$. Moreover, given a chordal graph, this corresponding tree can be found in polynomial time. From this definition, it is clear that each $X \in \mathcal{X}$ induces a clique in G.

3 Appetizer: Parameterized Complexity of Densest k-Subgraph in Chordal Graphs

Let us discuss the parameterized status of Densest k-Subgraph in chordal graphs. First, it is well-known that computing a maximal clique in chordal graphs can be done in polynomial time. Hence, if the input graph contains a clique of size k or more, we can immediately output it. Otherwise, we use another classical property of chordal graphs stating that that their treewidth equals the size of their maximal clique minus one. Thus, the treewidth of the input graph is bounded by $(k-1)$. Finally, using a classical dynamic programming on a tree decomposition, such as the one described in [4], we can compute an optimal solution in FPT time.

On the negative side, we can easily cross-compose DENSEST k-SUBGRAPH in chordal graphs into itself, by taking the disjoint union of t chordal graphs, and adding sufficiently enough universal vertices to each connected component. Due to space restrictions, this construction cannot be formally defined and proved here. To summarize, we have the following result:

Theorem 1. DENSEST k-SUBGRAPH *in chordal graphs is FPT and does not admit a polynomial kernel unless $\mathcal{NP} \subseteq co\mathcal{NP}/poly$, both parameterized by k.*

4 FPT Algorithm for Sparsest k-Subgraph in Chordal Graphs

Definitions and Notations. Let $G = (V, E)$ be a chordal graph and $T = (\mathcal{X}, A)$ be its corresponding tree decomposition as defined in section 2. Recall that for each $X \in \mathcal{X}$, X induces a clique in G.

We denote respectively by $\mathcal{L}$ and $\mathcal{I}$ the set of leaves and internal nodes of T (we have $\mathcal{X} = \mathcal{L} \cup \mathcal{I}$). In the following we suppose that T is rooted at an arbitrary node X_r. Let $X \in \mathcal{X}$, we denote by $pred(X)$ the unique predecessor of X in T (by convention $pred(X_r) = \emptyset$), and by $succ(X)$ the set of successors of X in T. For a vertex $v \in V$ (resp. a node $X \in \mathcal{X}$), we denote by $d(v)$ (resp. $d(X)$) its degree in G (resp. in T). For a set of vertices $U \subseteq V$ (resp. set of nodes $A \subseteq \mathcal{X}$), we denote by $G[U]$ (resp. $T[A]$) the subgraph of G induced by U (resp. the subforest of T induced by A). We say that a vertex $v \in V$ is a *lonely*[1] vertex (*resp. almost lonely* vertex) if $|\mathcal{X}_v| = 1$ (resp. $|\mathcal{X}_v| = 2$), *i.e.* if it appears in only one (resp. two) nodes of T.

First Observations. A maximum independent set can be computed in polynomial time in chordal graphs (since chordal graphs are perfect). Hence, we first determine if there exists an independent set of size k. In this case, we return this set which is naturally an optimal solution. Thus, we assume in the following that the graph G does not contain an independent set of size k.

Notice that we can assume that for every leaf L of the tree we do not have $L \subseteq pred(L)$ (otherwise we can contract the two nodes). Therefore, for each leaf L of the tree, there is a vertex $x \in L$ such that $x \notin pred(L)$, i.e. x is a lonely vertex. Since there is no independent set of size k in G, and since lonely vertices of leaves are pairwise non adjacent, we have the following:

[1] Notice that every lonely vertex is a so-called "simplicial vertex" (a vertex whose neighborhood is a clique). However, if a node of the tree is contained in another node, a simplicial vertex may not be a lonely one. Since we do not make any supposition on the tree T (we will in particular duplicate nodes during the algorithm), we will prefer the term "lonely".

Observation 1 *We can assume that $|\mathcal{L}| < k$.*

Let us now state a simple property verified by optimal solutions. Let S be a set of k vertices. Assume that there are vertices $x \in S$ and $y \in V \setminus S$ such that $\mathcal{X}_y \subsetneq \mathcal{X}_x$. Then it means that $N(y) \subseteq N(x)$. Thus, if we replace x by y in the solution, the number of edges in the solution cannot increase. A set S is *closed under inclusion* if there is no vertex x in S such that there exists $y \in V \setminus S$ such that $\mathcal{X}_y \subsetneq \mathcal{X}_x$. So there always exists an optimal solution closed under inclusion.

Idea of the Algorithm. Our goal is to find an optimal solution closed under inclusion. First note that any optimal solution closed under inclusion must contain a lonely vertex per leaf of T. Indeed, as each leaf L is not included in $pred(L)$, there exists a lonely vertex x in L. Thus, either the solution intersects L, and since it is closed by inclusion it contains a lonely vertex, or we can take a vertex of the solution and replace it by x, which does not create any additional edge (since no vertex of $N(x) = L \setminus \{x\}$ was in the solution).

Our method can be summarized as follows. First, we take a lonely vertex in each leaf and guess a binary flag $w(L) \in \{0, 1\}$ for each leaf L which indicates whether another vertex of L has to (with value 1) or does not have to (with value 0) be taken in the solution. The width of such a branching is bounded according to Observation 1. Then, given a leaf L with $w(L) = 1$, we first try to add to the solution the "most interesting" vertex of the leaf (for example a lonely vertex). When this is not possible (the neighborhood of the vertices of L can be incomparable if these vertices appear on incomparable subtrees), we apply some branching rules that re-structure the tree and create new "interesting vertices".

Terminology for the Algorithm. The algorithm is a branching algorithm composed of pre-processing rules (which do not require branching) and branching rules. When a rule is applied, we assume that previous rules cannot be applied.

During the algorithm, a partial solution S (initialized to $\emptyset$) will be constructed, and the input graph $G = (V, E)$ together with k, T (and thus $\mathcal{X}$, $\mathcal{L}$ and $\mathcal{I}$) will be modified. To avoid heavy notation we will keep these variables to denote the current input, and denote by G_0 the original graph, and by N_0 the neighborhood function of G_0.

In the following, *taking* a vertex $v \in V$ in the solution means that v is added to S, and v is removed both from the graph G and the tree T (removing each of its occurrences). *Deleting* a vertex means removing the vertex from G and from T. If a leaf of T becomes empty after taking or deleting a vertex, then simply remove the leaf.

Let $F \in \mathcal{I}$ be a leaf of $T[\mathcal{I}]$ (*i.e.* a node of T of which all successors are leaves). The node F is a *bad father* if there exists a vertex u which appears in at least two leaves of $succ(F)$. So a node is a bad father if the leaves attached to it are not vertex disjoint. We denote by $\#BF$ the number of bad fathers of the tree. Finally, we denote by $\#AL$ (for "almost leaf") the number of internal nodes of T such that at least one successor is a leaf. Notice that $\#AL, \#BF \leq |\mathcal{L}|$.

In addition, as said previously, we will put “flags” on some leaves $\mathcal{L}^* \subseteq \mathcal{L}$ by introducing a boolean function $w : \mathcal{L}^* \to \{0, 1\}$, which indicates whether it intersects the solution (value 1) or not (value 0). At the beginning of the algorithm we have $\mathcal{L}^* = \emptyset$. For a solution $S \subseteq V_0$, we say that S *respects* the flags w if for all $L \in \mathcal{L}^*$, $w(L) = 0$ iff $S \cap L = \emptyset$. During the algorithm we will use the term "guessing" the value $w(L)$ of $L \in \mathcal{L}$. By this, we mean that we try the two possible choices (consistent with the previous ones), creating at most two distinct executions of the algorithm. Notice that $\mathcal{L}^*$ will be implicitly updated (*i.e.* L belongs to $\mathcal{L}^*$ in the next executions if we have guessed $w(L)$).
We also add a function $g : \mathcal{L}^* \to 2^S$. Roughly speaking, we will modify g during the algorithm such that g remembers the neighbors of the remaining vertices V in the partial solution S already constructed. Notice that we introduced g only for the analysis, and more precisely for maintaining our invariants (see bellow).

Correctness and Time Complexity. As usually in a branching algorithm, we bound the time complexity by bounding both the depth and the maximum degree of the search tree. More precisely, we will show that:

– Each rule can be applied in FPT time.
– The branching degree of each branching rule is a function of k.
– Any branching rule strictly decreases $(k, \#AL, \#BF)$ using the lexicographic order, whose initial value only depends on the initial value of k (by Observation 1).
– Any pre-processing rule does not increase $(k, \#AL, \#BF)$ and decreases $|V| + |\mathcal{I}|$.

Thus, the number of branching steps of the search tree is a function of k only, whereas the number of steps between two branchings is polynomial in n (recall that $|\mathcal{X}|$ is polynomial in n), which leads to an FPT running time.

Recall that S denotes the partial current solution. Concerning the correctness of the algorithm, we will say that a rule is *safe* if it preserves all the following invariants:

1. The tree T is still a tree decomposition (as defined in 2) of G, which is an induced subgraph of G_0.
2. If a vertex of the partial solution is adjacent to a "surviving" vertex $v \in V$, then v must appear in a leaf where a flag is defined, *i.e.*:
$N_0(S) \cap V \subseteq \bigcup_{L \in \mathcal{L}^*} L$.
3. The neighborhood of a surviving vertex u in the partial solution is defined by the union of $g(L)$ for each L in which u appears, *i.e.* $g : \mathcal{L}^* \to 2^S$ is such that $\forall u \in V$ we have $N_0(u) \cap S = \bigcup_{L \in \mathcal{X}_u \cap \mathcal{L}^*} g(L)$.
In particular, this invariant implies that if there are $u, v \in V$ such that $\mathcal{X}_u \cap \mathcal{L}^* \subseteq \mathcal{X}_v \cap \mathcal{L}^*$ (*i.e.* v appears in at least as many labelled leaves as u), then we must have $N_0(u) \cap S \subseteq N_0(v) \cap S$ (*i.e.* v is adjacent to at least as many vertices of the solution as u).
4. If there is an optimal solution (closed under inclusion) $S^* \subseteq V$ such that $S \subseteq S^*$, and S^* respects the flags w, then one of the branching will output an optimal solution.

Reduction Rules. Notice that each of the following rules defines a new value for variables k, T, S, w and g. However, for the sake of readability we will not mention variables that are not modified. Due to space restrictions, all safeness proofs were omitted.

Pre-Processing Rule 1: useless duplicated node.
If there exists $X \in \mathcal{X}$ such that $X \notin \mathcal{L}^*$ and $X \subseteq pred(X)$, then contract X and $pred(X)$ (*i.e.* delete X, and connect every $Y \in succ(X)$ to $pred(X)$).

Pre-Processing Rule 2: removing a (almost) lonely vertex.
If there exists $L \in \mathcal{L}^*$ such that $w(L) = 0$, then if L contains a lonely vertex v, delete v. Otherwise, if L contains an almost lonely vertex v, then delete v.

Branching Rule 1: taking a lonely vertex.
If there exists $L \in \mathcal{L}^*$ such that $w(L) = 1$ and L contains a lonely vertex v, then take v in the solution and decrease k by one. In addition, add v into $g(L)$, and if L does not become empty, then guess a new value $w(L)$.

Remark 1. At this point, since *Pre-Processing Rule 1* does not apply, it is clear that every leaf $L \in \mathcal{L} \setminus \mathcal{L}^*$ contains a lonely vertex. The following branching rule aims to process these leaves.

Branching Rule 2: processing leaves with no flag.
If there exists $L \in \mathcal{L} \setminus \mathcal{L}^*$, then take a lonely vertex $v \in L$ in the solution and decrease k by one. In addition, add v into $g(L)$, and if L does not become empty, guess a value $w(L)$.

Remark 2. At this point, notice that $\mathcal{L}^* = \mathcal{L}$, *i.e.* a flag has been assigned to each leaf. Indeed, suppose that there exists $L \in \mathcal{L} \setminus \mathcal{L}^*$. If L contains a lonely vertex, then *Branching Rule 1* must apply. Otherwise, *Pre-Processing Rule 1* must apply. In addition, there is no lonely vertex in the leaves, as otherwise *Branching Rule 1* or *Pre-Processing Rule 2* would apply.

Branching Rule 3: partitioning leaves of a bad father.
If there exists a bad father $F \in \mathcal{X}$, let $\mathcal{L}'$ be the set of leaves in $succ(F)$ and $C = \bigcup_{L \in \mathcal{L}'} L$ be the set of vertices contained these leaves. Partition C into the equivalence classes $C_1, ..., C_t$ of the following equivalence relation: two vertices $u, v \in C$ are equivalent if $\mathcal{X}_u \cap \mathcal{L}' = \mathcal{X}_v \cap \mathcal{L}'$ (*i.e.* u and v appear in the same subset of leaves of F). For all $i \in \{1, ..., t\}$, let $\mathcal{L}^i \subseteq \mathcal{L}'$ denote the subset of leaves in which vertices of C_i were before the partitioning. Then, replace the leaves of F by $C_1, ..., C_t$, and for all $i \in \{1, ..., t\}$, guess $w(C_i)$ and set $g(C_i) = \bigcup_{L \in \mathcal{L}^i} g(L)$.

Let us give the intuition of *Branching Rule 3*. This rule ensures that the set of leaves attached to a same node are vertex disjoint and that the partition was made in such a way that two vertices in the same leaf after the application of the rule were in the same subset of leaves before it. Notice that the remaining Branching Rules can create bad fathers, but decrease k.

Branching Rule 4: taking a lonely vertex in a father.
If there exists $L \in \mathcal{L}$ such that $pred(L)$ contains a lonely vertex v, then take v in the solution, delete k by one, and create a new leaf N adjacent to $pred(L)$ and containing vertices of $L \backslash \{v\}$. Finally, guess a value for $w(N)$ and set $g(N) = \{v\}$.

Branching Rule 5: taking an almost lonely vertex in a leaf.
If there exists $L \in \mathcal{L}$ such that $w(L) = 1$ and L contains an almost lonely vertex v (thus contained in L and $pred(L)$), then take v in the solution, decrease k by one, and create a new leaf N adjacent to $pred(L)$ and containing vertices of $pred(L) \setminus \{v\}$. If L does not become empty, then guess a new value $w(L)$. Finally, guess a value $w(N)$, add v into $g(L)$, and set $g(N) = \{v\}$.

End of the Algorithm.

Lemma 1. *If no rule can be applied then either G is empty or $k = 0$.*

According to the introduction and all the safeness lemmas, the size of the search tree is a function of k. Then, let us remark that all rules can be applied in FPT time. This is clear for *Pre-Processing Rules 1* and *2*, as well as for *Branching Rules 1, 2, 4* and *5*. Concerning *Branching Rule 3*, which consists in partitioning a subset of leaves, it runs in FPT time as long as $|\mathcal{L}|$ is a function of k. This is obviously the case at the beginning of the algorithm (since $|\mathcal{L}| < k$), and the number of leaves only increase by one in *Branching Rule 4* and *5*, and by a function of the previous number of leaves in *Branching Rule 3*. Since the branching rules are applied at most $f(k)$ times, we get the desired result.

Theorem 2. *There is an FPT algorithm for* Sparsest k-Subgraph *in chordal graphs, parameterized by k.*

However, the running time of the algorithm may be a tower of 2 of height k, since *Branching Rule 3* may create 2^t new leaves, where t is the number of previous leaves of the node F. Nevertheless, we can slightly modify the algorithm in order to obtain a $O^*(2^{k^2})$ running time[2]. Indeed, after the application of this rule, all leaves L such that $w(L) = 0$ can be gathered into one leaf, since all these vertices are not in the solution. And since all leaves are vertex disjoint, the number of leaves L such that $w(L) = 1$ is at most k (since one vertex of each leaf is in the solution). Hence, the number of leaves of F after the application of *Branching Rule 3* can actually be bounded by $k+1$. Then, as said previously, the only other branching rules which increase the number of leaves are *Branching Rules 4* and *5*, which both add at most one leaf when they are applied. However, since these branching rules are decreasing k, the maximum number of leaves of a node F before the application of *Branching Rule 3* is $2k$. Hence, this rule (which upper bounds the running time of the algorithm) runs in time $O^*(2^{O(2k^2)})$ (we have at most 2^{2k} leaves and we choose at most k leaves such that $w(L) = 1$). For sake of readability, the presented algorithm does not contain this slight modification.

[2] The $O^*(.)$ notation avoids polynomial terms.

5 Kernel Lower Bound of Sparsest k-Subgraph in Chordal Graphs

Intuition of the proof. The following kernel lower bound is obtained using a cross-composition. It is an extension of our previous work [18], showing the $\mathcal{NP}$-hardness of Sparsest k-Subgraph in chordal graphs. Let us first give the intuition of this result, and then explain the modification we apply which leads to the kernel lower bound. We then explicit the whole construction of the cross-composition and give a formal proof of the result.

The $\mathcal{NP}$-hardness proof is a reduction from the classical k-clique problem in general graphs and roughly works as follows. Given an input instance $G = (V, E)$, $k \in \mathbb{N}$ of k-clique, we first build a clique A representing the vertices of G. We also represent each edge $e_j = \{u, v\} \in E$ by a gadget F_j, and connect the representative vertices of u and v in A to some vertices of F_j (see the left of Figure 1). The reduction will force the solution to take in A the representatives of $(n-k)$ vertices of G (corresponding to the complement of a solution S of size k in G), and also to take the same number of vertices among each gadget. The key idea is that the cost of a gadget F_j increases by one if it is adjacent to one of the selected vertices of A. Thus, since the goal is to minimize the cost, we will try to maximize the number of gadgets adjacent to the representatives of S (*i.e.* vertices we did not pick in A), the maximum being reached when S is a clique in G.

To adapt this reduction into a cross-composition, we add an *instance selector* composed of $2\log t$ gadgets adjacent to A (where t is the number of input instances of the cross-composition) which encodes the binary representation of each instance index. These gadgets have the same structure as the F_j. For technical reasons, this instance selector has to be duplicated many times, as well as the clique A which we must duplicate t times in order to encode the vertex set of each instance. The right of Figure 1 represents the construction in a simplified way. Let us now define formally the gadgets and state their properties.

Definition of a gadget. Let $T \in \mathbb{N}$ (we will set the value of T later). The vertex set of each gadget is composed of three sets of T vertices X, Y and Z, with $X = \{x_1, ..., x_T\}, Y = \{y_1, ..., y_T\}$ and $Z = \{z_1, ..., z_T\}$. The set X induces an independent set, the set Z induces a clique, and there is a $(T-1)$-clique on $\{y_2, ..., y_T\}$. In addition, for all $i \in \{1, ..., T\}$, we connect y_i to all vertices of Z and to x_i. The left of Figure 1 summarizes the construction.
In the following cross-composition, we will force the solution to take $2T$ vertices among each gadget F. It is easy to see that the sparsest $2T$-subgraph of F is composed of the sets X and Z, which induces $\binom{T}{2}$ edges. In addition, if we forbid the set Z to be in the solution (if the gadget is adjacent to some picked vertices of A), then the remaining $2T$ vertices (namely X and Y) induce $(\binom{T}{2}+1)$ edges.

Theorem 3. Sparsest k-Subgraph *does not admit a polynomial kernel in chordal graphs unless $\mathcal{NP} \subseteq co\mathcal{NP}/poly$ (parameterized by k).*

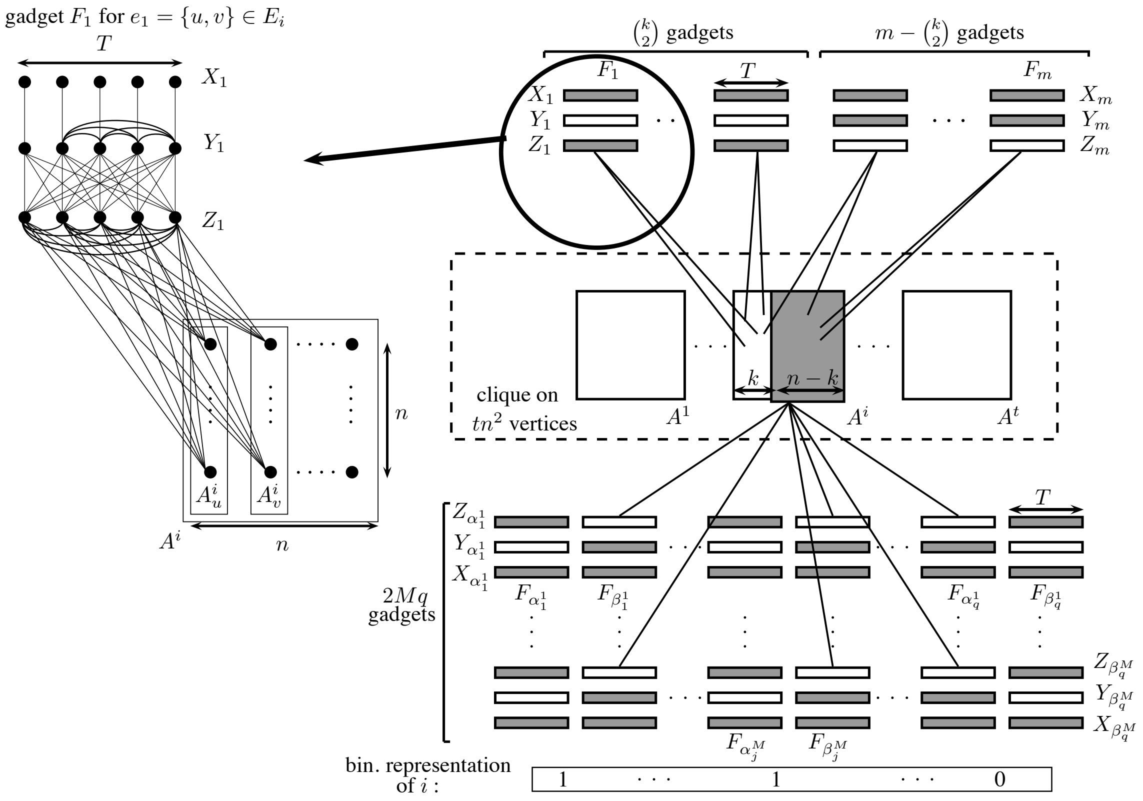

Fig. 1. Schema of the cross-composition (right) and a detailed gadget (left). Grey rectangles represent vertices of the solution, supposing that G_i contains a clique of size k. Notice that gadgets of the bottom have been drawn in the reverse direction (*e.g.* $X_{\beta_1^1}$ is below $Y_{\beta_1^1}$). Edges of the clique A have not been drawn for sake of clarity.

Proof. Let $(G_1, k_1), ..., (G_t, k_t)$ be a sequence of t instances of k-CLIQUE, with $G_i = (V_i, E_i)$ for all $i \in \{1, ..., t\}$. W.l.o.g. we suppose that $t = 2^q$ for some $q \in \mathbb{N}$, and define $T = n(n-k)$ and $M = n^6$.

Our polynomial equivalence relation is the following: for $1 \le i, j \le t$, (G_i, k_i) is equivalent to (G_j, k_j) if $|V_i| = |V_j| = n$, $|E_i| = |E_j| = m$ and $k_i = k_j = k$. One can verify that this relation is a polynomial equivalence relation. In what follows we suppose that all instances of the sequence are in the same equivalence class. The output instance $G' = (V', E'), k', C'$ is defined as follows (see Figure 1):

- For each $i \in \{1, ..., t\}$ we construct a clique A^i on n^2 vertices, where A^i is composed of n subcliques $A^i_1, ..., A^i_n$. We also add all possible edges between all cliques $(A^i)_{i=1..n}$. Hence, $A = \bigcup_{i=1}^t A^i$ is a clique of size tn^2.
- Since all instances have the same number of edges, we construct m gadgets $(F_j)_{j=1..m}$, where each F_j is composed of X_j, Y_j and Z_j as described

previously. For all $i \in \{1, ..., t\}$, if there is an edge $e_j = \{u, v\} \in E_i$, then we connect all vertices of Z_j to all vertices of A^i_u and A^i_v. Let us define $\mathcal{F} = \bigcup_{j=1}^{m} F_j$ the subgraph of all gadgets of the "edge selector".

- We add $2qM$ gadgets $(F_{\alpha_j^h})_{j=1..q}^{h=1..M}$ and $(F_{\beta_j^h})_{j=1..q}^{h=1..M}$, where all gadgets are isomorphic to the edge gadgets, and thus composed of $X_{\alpha_j^h}, Y_{\alpha_j^h}$ and $Z_{\alpha_j^h}$ (resp. $X_{\beta_j^h}, Y_{\beta_j^h}$ and $Z_{\beta_j^h}$) for all $h \in \{1, ..., M\}$ and all $j \in \{1, ..., q\}$. Let $i \in \{1, ..., t\}$, and consider its binary representation $b \in \{0, 1\}^q$. For all $j \in \{1, ..., q\}$, if the j^{th} bit of b equals 0, then connect all vertices of A^i to all vertices of $\bigcup_{h=1}^{M} Z_{\alpha_j^h}$. Otherwise, connect all vertices of A^i to all vertices of $\bigcup_{h=1}^{M} Z_{\beta_j^h}$. Let us define $\mathcal{B} = \bigcup_{h=1}^{M} \bigcup_{j=1}^{q} (F_{\alpha_j^h} \cup F_{\beta_j^h})$ the subgraph of all gadgets of the "instance selector".
- We set $k' = T + 2Tm + 4TqM$ and $C' = \binom{T}{2} + \binom{T}{2}(m + 2Mq) + (m - \binom{k}{2}) + Mq$.

It is clear that G', k' and C' can be constructed in time polynomial in $\sum_{i=1}^{t} |G_i| + k_i$. Then, one can verify that G' is a chordal graph. Indeed, it is known [13] that a graph is chordal if and only if one can repeatedly find a simplicial vertex (a vertex whose neighborhood is a clique) and delete it from the graph until it becomes empty. Such an ordering is called a simplicial elimination order. It is easily seen that for each gadget, X, Y and then Z is a simplicial elimination order (each gadget is only adjacent to the clique A *via* its set Z). Finally it remains the clique A which can be eliminated.

In addition, notice that the parameter k' is a polynomial in n, k and $\log t$ only and thus respect the definition of a cross-composition. We finally prove that there exists $i \in \{1, ..., t\}$ such that G_i contains a clique K of size k if and only if G' contains a set K' of k' vertices inducing C' edges or less.

Lemma 2. *If there exists $i \in \{1, ..., t\}$ such that G_i contains a k-clique, then G' contains k' vertices inducing at most C' edges.*

Proof. Suppose that $K \subseteq V_i$ is a clique of size k in G_i. W.l.o.g. suppose that $K = \{v_1, ..., v_k\}$, and that $\{\{u, v\}, u, v \in K\} = \{e_1, ..., e_{\binom{k}{2}}\}$. Let $b \in \{0, 1\}^q$ be the binary representation of i. We build K' as follows (see Figure 1).

- For all $j \in \{1, ..., \binom{k}{2}\}$, K' contains X_j and Z_j ($2T$ vertices inducing $\binom{T}{2}$ edges for each gadget F_j).
- For all $j \in \{\binom{k}{2} + 1, ..., m\}$, K' contains X_j and Y_j. ($2T$ vertices inducing $(\binom{T}{2} + 1)$ edges for each gadget F_j).
- For all $u \notin \{1, ..., k\}$, K' contains A^i_u (T vertices inducing $\binom{T}{2}$ edges).
- For all $h \in \{1, ..., M\}$, and all $j \in \{1, ..., q\}$, K' contains $X_{\alpha_j^h}$ and $X_{\beta_j^h}$. Moreover, if the j^{th} bit of b equals 1, then K' contains $Y_{\beta_j^h}$ and $Z_{\alpha_j^h}$, otherwise K' contains $Z_{\beta_j^j}$ and $Y_{\alpha_j^h}$ ($4T$ vertices inducing $(2\binom{T}{2} + 1)$ edges for each pair of gadgets $F_{\alpha_j^h}$ and $F_{\beta_j^h}$).

One can easily verify that K' is a set of k' vertices inducing C' edges. □

The following lemma, which proof has been omitted, terminates the proof.

Lemma 3. *If G' contains k' vertices inducing at most C' edges, then $\exists i \in \{1, ..., t\}$ such that G_i contains a k-clique.*

References

1. N. Apollonio and B. Simeone. The maximum vertex coverage problem on bipartite graphs. *Discrete Applied Mathematics*, (in press), 2013.
2. H. L. Bodlaender, B. M. P. Jansen, and S. Kratsch. Cross-composition: A new technique for kernelization lower bounds. In *STACS*, pages 423–434, 2011.
3. E. Bonnet, B. Escoffier, V. Th. Paschos, and E. Tourniaire. Multi-parameter complexity analysis for constrained size graph problems: using greediness for parameterization. to appear in *IPEC 2013*.
4. N. Bourgeois, A. Giannakos, G. Lucarelli, I. Milis, and V. Th. Paschos. Exact and approximation algorithms for densest k-subgraph. In *WALCOM*, pages 114–125, 2013.
5. N. Bousquet, M. Bougeret, R. Giroudeau, and R. Watrigant. Parameterized complexity of the sparsest k-subgraph problem in chordal graphs. Technical Report RR-13033, LIRMM, 2013.
6. H. Broersma, P. A. Golovach, and V. Patel. Tight complexity bounds for FPT subgraph problems parameterized by clique-width. In *Proceedings of the 6th international conference on Parameterized and Exact Computation*, IPEC'11, pages 207–218, Berlin, Heidelberg, 2012. Springer-Verlag.
7. L. Cai. Parameterized complexity of cardinality constrained optimization problems. *Computer Journal*, 51(1):102–121, 2008.
8. D. Chen, R. Fleischer, and J. Li. Densest k-subgraph approximation on intersection graphs. In *Proceedings of the 8th international conference on Approximation and online algorithms*, pages 83–93. Springer, 2011.
9. D.G. Corneil and Y. Perl. Clustering and domination in perfect graphs. *Discrete Applied Mathematics*, 9(1):27 – 39, 1984.
10. U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. *Algorithmica*, 29:2001, 1999.
11. J. Flum and M. Grohe. *Parameterized Complexity Theory.* Springer, 2006.
12. F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47 – 56, 1974.
13. M. C. Golumbic. *Algorithmic graph theory and perfect graphs.* Academic Press, New York, USA, 1980.
14. G. Joret and A. Vetta. Reducing the rank of a matroid. *CoRR*, abs/1211.4853, 2012.
15. S. Khot. Ruling out ptas for graph min-bisection, dense k-subgraph, and bipartite clique. *SIAM Journal of Computing*, 36:1025–1071, 2004.
16. M. Liazi, I. Milis, and V. Zissimopoulos. A constant approximation algorithm for the densest k-subgraph problem on chordal graphs. *Information Processing Letters*, 108(1):29–32, 2008.
17. T. Nonner. PTAS for densest k-subgraph in interval graphs. In *Proceedings of the 12th international conference on Algorithms and Data Structures*, pages 631–641. Springer, 2011.
18. R. Watrigant, M. Bougeret, and R. Giroudeau. Approximating the sparsest k-subgraph in chordal graphs. to appear in *WAOA 2013*.