On the Effective Enumerability of NP Problems

Jianer Chen* Iyad A. Kanj† Jie Meng* Ge Xia§ Fenghui Zhang*

Abstract

In the field of computational optimization, it is often the case that we are given an instance of an NP problem and asked to enumerate the first few "best" solutions to the instance. Motivated by the recent research performed in these fields, we propose in this paper a new framework to measure the effective enumerability of NP optimization problems. More specifically, given an instance of an NP problem, we consider the problem of enumerating a given number of best solutions for the instance, and study its average complexity in terms of the number of solutions. Our framework is different from the previously-proposed ones, which studied the counting complexity of a problem, or the complexity of enumerating all solutions to a given instance of the problem. For example, even though it was shown by Flum and Grohe that counting the number of k-paths in a graph is fixed-parameter intractable, we present a fixed-parameter enumeration algorithm for the problem. The developed enumeration framework consists of two phases: the structure-generation phase and the solution-enumeration phase. We show that most algorithm-design techniques for fixed-parameter tractable problems, such as search trees, color coding, and bounded treewidth, can be transformed into techniques for the structure-generation phase. We design elegant enumeration techniques, and combine them with the use of effective data structures, to show how to generate small-size structures and enumerate them efficiently.

1 Introduction

Most computational problems are concerned with finding a single solution for a problem instance. For example, decision problems ask for the existence of *a* solution (to a given instance) that satisfies certain properties, while optimization problems seek *a* solution (to a given instance) that optimizes a certain function [27].

On the other hand, many computational problems in practice seek a number of good solutions rather than a single one. It is a natural practice in many branches in science and technology that experts in the area like to see and check many "good solutions" in order to choose the solutions that appear to be the most proper ones, given their additional background knowledge in the subject. Examples of such cases include seeking significant sub-structures in biological networks [22, 30], studying sequence Motifs and alignments [28], studying evolutionary trees [18], and constructing a list of codewords in list decoding [19]. Moreover, because of the proneness of computation to errors, the computed optimal solution may not be the "real" optimal solution. Therefore, it becomes desirable to generate several "best" solutions rather than a single one.

Several approaches towards meeting this need have been proposed. The most notable one is the study of the counting complexity of a problem, which is the computational complexity of counting all the solutions to a given instance of the problem. Since its initialization by Valiant [32], significant work has been done on the study of counting complexity. Most of this work has focused on the negative side, i.e., proving the intractability of certain counting problems. For example, Valiant [32] proved that counting the number of perfect matchings in a bipartite graph is #P-complete. Hunt et al. [20] proved the #P-hardness of a number of counting problems on planar graphs. Flum and Grohe [17] studied the

*Supported in part by NSF Grants CCR-0311590 and CCF-4030683, Department of Computer Science, Texas A&M University, College Station, TX 77843, USA. Emails: {chen,jmeng,fhzhang}@cs.tamu.edu.

†Supported in part by DePaul University Competitive Research Grant. School of CTI, DePaul University, 243 S. Wabash Avenue, Chicago, IL 60604, USA. Email: ikanj@cs.depaul.edu.

§Supported in part by NSF Grant CCF-4030683. Department of Computer Science, Lafayette College, Easton, PA 18042, USA. Emails: gexia@cs.lafayette.edu.

parameterized complexity of counting problems and, in particular, proved that the problem of counting the number of k-paths in a graph is #W[1]-complete. Positive results along this line of research lead to a number of exact algorithms (e.g., [3, 11, 29]) and approximation algorithms (e.g., [9, 12]) for a number of counting problems that are intractable.

Another approach along this line of research studied the complexity of enumerating *all* solutions to a given problem instance. Tomita, Tanaka, and Takahashi [31] presented an exponential time algorithm that enumerates all maximal cliques in a graph. Gramm and Niedermeier [18] gave an algorithm that enumerates all minimum solutions for the QUARTET INCONSISTENCY problem. Fernau [16] considered a number of enumeration paradigms and studied their respective complexities.

None of the above approaches, however, has perfectly met the practical needs of the corresponding applications associated with the problems under consideration. The study of counting complexity does not provide hints on how the solutions to a given instance of the problem can be generated. Even worse, the counting complexity of a problem can be significantly different from that of generating a single solution (e.g., perfect matchings in bipartite graphs from the standard complexity viewpoint [32], k-paths in a graph from the parameterized complexity viewpoint [17]). The enumeration approach (i.e., enumerating all solutions to a given instance) may easily become computationally infeasible, not because of the difficulty of generating each single solution, but simply because the number of solutions is too large. For example, the problem of constructing a vertex cover of k vertices in a graph is practically feasible for small values of k [6], but the problem of enumerating all vertex covers of k vertices in a graph is computationally infeasible simply because there can be too many such vertex covers in the graph [16].

On the other hand, many computational applications *do not* ask for the entire set of solutions, instead, they only seek a certain number of "best" solutions [18, 22, 28, 30].

Motivated by the above, we propose in this paper a new framework to study the effective enumerability of NP optimization problems. Needless to say, in order to be able to effectively enumerate a set of solutions, we must be able to generate a single solution first. Therefore, we will be mainly interested in the NP optimization problems that have efficient algorithms for generating a single solution. We will also be seeking solutions of small size k, and study the enumerability of problems whose first solution can be generated in time $f(k)n^{O(1)}$, where f is a recursive function. The readers who are familiar with fixed-parameter tractability theory should realize that this refers to the class of fixed-parameter tractable problems [13]. We associate each problem solution with a "weight" that indicates the quality/ranking of the solution. We say that an NP optimization problem is *fixed-parameter enumerable* if there is an algorithm that, for a given problem instance (x, k) and an integer K, generates the K best (in terms of the solution weight) solutions of size k to x in time $f(k)n^{O(1)}K^{O(1)}$.

Our enumeration model is meaningful from both the theoretical and practical perspectives. Indeed, generating K solutions takes time at least $O(K)$, therefore, it should be acceptable to require that generating the K best solutions takes time polynomial in K. Besides the polynomial factor in K, we require that generating a solution takes time $f(k)n^{O(1)}$, which is feasible for small values of k [13]. The model is specially suitable for applications that require a moderate number of best solutions, i.e, in which $K = n^{O(1)}$.

By setting $K = 1$, we can easily see that a fixed-parameter enumerable problem is also fixed-parameter tractable. It will be interesting to know whether these two notions are equivalent. Along this line, we examine the most popular techniques used in developing fixed-parameter tractable algorithms, including the bounded search-tree method, color coding schemes, and bounded tree-width algorithms. The developed enumeration framework consists of two phases: the structure-generation phase and the solution-enumeration phase. We show that most algorithm-design techniques for fixed-parameter tractable problems, such as search trees, color coding, and bounded treewidth, can be non-trivially transformed into techniques for the structure-generation phase. We design elegant enumeration techniques, and combine them with the use of effective data structures, to show how to generate small-size structures and enumerate them efficiently. For instance, we present a fixed-parameter enumerable algorithm for the k-PATH problem, even though counting the number of k-paths in a graph is known to be fixed-parameter intractable [17].

There has been some research in the literature that is related to this research. For example, Chegireddy and Hamacher [8] developed algorithms for finding the K largest perfect matchings in a weighted graph, Kapoor and Ramesh [21] studied the complexity of generating the K smallest spanning trees in a weighted graph, and Eppstein considered the problem of enumerating the K shortest paths in a digraph [14]. For a more comprehensive summary of this line of research, the readers are referred to [25]. However, to the authors' knowledge, all this research dealt with very specific optimization problems that are solvable in polynomial time. On the other hand, the current paper mainly focuses on NP-hard optimization problems, and on developing a systematical approach for the effective enumeration of a large class of such problems.

2 Definitions and preliminaries

Recall that a *parameterized problem* consists of instances of the form (x, k), where $x \in \Sigma^*$ for a finite alphabet set Σ, and k is a non-negative integer. A parameterized problem Q is *fixed parameter tractable* if there is an algorithm A that on input (x, k) decides if (x, k) is a yes-instance of Q in time $f(k)n^{O(1)}$, where f is a recursive function independent of $n = |x|$. We extend the standard definition of NP optimization problems [4] to encompass their parameterized versions.

Definition A *parameterized NP optimization problem* Q is a 4-tuple (I_Q, S_Q, f_Q, opt_Q) where:

1. I_Q is the set of input instances of the form (x, k), with $x \in \Sigma^*$ for a fixed finite alphabet Σ, and k is a non-negative integer called the *parameter*. The input instances are recognizable in polynomial time.
2. For each instance (x, k) in I_Q, $S_Q(x, k)$ is the set of *feasible solutions* for (x, k), which is defined by a polynomial p and a polynomial time computable predicate Φ (p and Φ depend only on Q) as $S_Q(x, k) = \{y : |y| \leq p(|x|) \text{ and } \Phi(x, k, y)\}$.
3. $f_Q(x, k, y)$ is the objective function mapping a pair $(x, k) \in I_Q$ and $y \in S_Q(x, k)$ to a real number. The function f_Q is computable in polynomial time.
4. $opt_Q \in \{\max, \min\}$.

Note that since the length of a solution y to an instance (x, k) in Q is bounded by a polynomial of $|x|$, the number of solutions to the instance (x, k) is bounded by $2^{q(|x|)}$ for some fixed polynomial q. Therefore, the values of the solutions in the set $S_Q(x, k)$ can be given in a finite sorted list $L = [f_Q(x, k, y_1), f_Q(x, k, y_2), \ldots]$, in a non-decreasing order when $opt_Q = \min$, and in a non-increasing order when $opt_Q = \max$. We say that a set $\{y'_1, \ldots, y'_K\}$ of K solutions in $S_Q(x, k)$ are the K *best solutions* for the instance (x, k), if the values $f_Q(x, k, y'_1)$, ..., $f_Q(x, k, y'_K)$, when sorted accordingly, are identical to the first K values in the list L.

Definition A parameterized NP optimization problem Q is *fixed-parameter enumerable* if there are two algorithms A_1 and A_2 such that the following are true.

1. Given an instance (x, k) of Q, the algorithm A_1 generates a structure $\tau_{x,k}$ in time $f(k)n^{O(1)}$, where f is a recursive function independent of $n = |x|$.
2. Given the structure $\tau_{x,k}$ and an integer $K \geq 0$, the algorithm A_2 generates the K best solutions to the instance (x, k) in time $O(|\tau_{x,k}|^{O(1)} K^{O(1)})$.[1]

The algorithm A_1 will be called the *structure algorithm*, and the algorithm A_2 will be called the *enumeration algorithm*. We say that the problem Q is *linearly fixed-parameter enumerable* if the running

[1] Note that it is possible that the total number $|S_Q(x, k)|$ of solutions is smaller than K. To avoid repeatedly distinguishing the two possible cases, we will simply use K to refer to the value $K_0 = \min\{K, |S_Q(x, k)|\}$.

time of the enumeration algorithm A_2 is $|\tau_{x,k}|^{O(1)}K$.

We comment on the above definition. Since the algorithm A_1 runs in time $f(k)n^{O(1)}$, the size $|\tau_{x,k}|$ of the structure $\tau_{x,k}$ is bounded by $f(k)n^{O(1)}$. In consequence, the running time of the enumeration algorithm A_2 is bounded by $f_1(k)n^{O(1)}K^{O(1)}$, where f_1 is a recursive function independent of n. Moreover, we require that for each input instance (x, k), the fixed-parameter enumerable problem Q have a small structure $\tau_{x,k}$ whose size is independent of the number K of solutions to be generated. The following theorem follows directly from the above definitions.

Theorem 2.1 *If a parameterized NP optimization problem Q is fixed-parameter enumerable then it is fixed-parameter tractable.*

The following simple observation will be useful when designing enumeration algorithms. Suppose that we have a list of n real numbers. By first finding the K-th largest (or the K-th smallest) number a in the list in time $O(n)$ [10], then partitioning the list using a as a "pivot", we can generate the K largest (or the K smallest) numbers in the list in time $O(n)$.

3 Effective enumerations based on branch-and-search

The branch-and-search method based on bounded search-trees has been a very popular and powerful technique in the development of efficient exact and parameterized algorithms [13]. The unfamiliar reader is referred to [13] for more information about the bounded search tree technique and the analysis of its running time.

We discuss how this technique can be employed in designing algorithms for the structure-generation phase of enumeration algorithms for parameterized NP optimization problems. As a running example, we describe the algorithm with VERTEX COVER as the underlying problem. Recall that a vertex set C in a graph G is a *vertex cover* for G, if each edge in G has at least one end in C. A vertex cover of k vertices will be called a *k-vertex cover*. The VERTEX COVER problem is a well-known fixed-parameter tractable problem, and parameterized algorithms for it have been extensively studied (e.g., [6]). Moreover, the counting complexity (i.e., counting the number of solutions to a given instance) and the complexity of enumerating all solutions to an instance of the problem, have also been examined. Arvind and Raman [3] (see also [17]) showed that counting the total number of k-vertex covers can be done in time $O(2^{k^2+k}k + 2^k n)$. The complexity of enumerating all k-vertex covers, however, depends on whether k is the size of a minimum vertex cover of the graph or not. Fernau [16] showed that if k is equal to the size of a minimum vertex cover, then enumerating all k-vertex covers can be done in time $O(2^k k^2 + kn)$, while if k is not equal to the size of a minimum vertex cover, then no algorithm of running time $f(k)n^{O(1)}$, for any recursive function, f can enumerate all k-vertex covers. The latter fact simply holds because in such case the number of k-vertex covers can be too large to be enumerated in such time.

We investigate the fixed parameter enumerability of the problem. We assume that the input graph G is weighted, and each vertex is associated with a real number (the *vertex weight*). The *weight* of a vertex cover C is the sum of the weights of the vertices in C. Therefore, a vertex cover C_1 is *smaller than* a vertex cover C_2 if the weight of C_1 is smaller than the weight of C_2.

> WEIGHTED VERTEX COVER: Given a weighted graph G on n vertices, and non-negative integers k and K, generate the K smallest k-vertex covers in G.

3.1 The structure algorithm

Let (G, k) be an instance of the WEIGHTED VERTEX COVER problem, where G is a graph on n vertices. Since a vertex of degree larger than k must be in every k-vertex cover of G, we can first remove all vertices of degree larger than k in the graph and then work on the remaining graph. This pre-processing can be

done in time $O(kn)$ even when the number of edges in G is larger than kn. Now the resulting instance (G', k') consists of a graph G' of $O(n)$ vertices and $O(kn)$ edges, and a parameter $k' \leq k$. Therefore, without loss of generality, we will assume that the input graph G has n vertices and $O(kn)$ edges.

The structure algorithm for WEIGHTED VERTEX COVER is a recursive algorithm based on the branch-and-search method, which on an input instance (G, k) returns a collection $\mathcal{L}(G, k)$ of triples (I, O, R), where each (I, O, R) is a partition of the vertex set of the graph G, representing the set of all k-vertex covers that include all vertices in I and exclude all vertices in O. Moreover, we require that in the subgraph induced by the vertex set R, all the vertices have degree bounded by 2. The structure algorithm is given in Figure 1.

Algorithm **structure-vc**

INPUT: *G: a weighted graph; k: an integer;*

1. **if** $(k < 0)$ or $(k = 0$ but the edge set of G is not empty$)$ **then return** $\mathcal{L}(G, k) = \emptyset$;
2. **if** there is no vertex of degree larger than 2 in G **then return** $\mathcal{L}(G, k) = \{(\emptyset, \emptyset, V)\}$;
3. pick any vertex v of degree $d \geq 3$;
4. let $G_1 = G - v$ and $G_2 = G - (v \cup N(v))$, where $N(v)$ is the set of neighbors of v;
5. recursively call **structure-vc**$(G_1, k-1)$ and **structure-vc**$(G_2, k-d)$;
   let the returned collections be $\mathcal{L}(G_1, k-1)$ and $\mathcal{L}(G_2, k-d)$, respectively;
6. $\mathcal{L}(G, k) = \emptyset$;
7. **for** each triple (I_1, O_1, R_1) in $\mathcal{L}(G_1, k-1)$ **do** add $(I_1 \cup \{v\}, O_1, R_1)$ to $\mathcal{L}(G, k)$;
8. **for** each triple (I_2, O_2, R_2) in $\mathcal{L}(G_2, k-d)$ **do** add $(I_2 \cup N(v), O_2 \cup \{v\}, R_2)$ to $\mathcal{L}(G, k)$;

Figure 1: The structure algorithm for WEIGHTED VERTEX COVER.

Theorem 3.1 (Theorem 7.1, Appendix) *On an input (G, k), the algorithm* **structure-vc** *runs in time $O(1.47^k n)$, and returns a collection $\mathcal{L}(G, k)$ of at most 1.466^k triples.*

We say that a vertex cover C of the graph G is *consistent with* a partition (I, O, R) of the vertex set of G if C contains all vertices in I and excludes all vertices in O. The following lemma can be proved by a simple inductive proof.

Lemma 3.2 *Let $\mathcal{L}(G, k)$ be the collection returned by the algorithm* **structure-vc** *on input (G, k). Then every k-vertex cover of G is consistent with exactly one triple in $\mathcal{L}(G, k)$.*

The collection $\mathcal{L}(G, k)$ forms the structure $\tau_{G,k}$ for the instance (G, k) of the WEIGHTED VERTEX COVER problem. By Theorem 3.1, the structure $\tau_{G,k}$ can be constructed in time $O(1.47^k n)$.

3.2 The enumeration algorithm

Let $\mathcal{L}(G, k)$ be the structure returned by the algorithm **structure-vc** on the input (G, k). By Lemma 3.2, every k-vertex cover C of G is consistent with exactly one triple (I, O, R) in $\mathcal{L}(G, k)$, in the sense that C contains all the vertices in I and excludes all the vertices in O. Therefore, the k-vertex cover C must consist of the vertex set I, plus a vertex cover for the subgraph $G(R)$ induced by the vertex set R of $k - |I|$ vertices. Therefore, the K smallest k-vertex covers for the graph G that are consistent with the triple (I, O, R) can be generated by generating the K smallest $(k - |I|)$-vertex covers for the induced subgraph $G(R)$. Finally, the K smallest k-vertex covers for the original graph G can be obtained by performing the above process on all the triples in the structure $\mathcal{L}(G, k)$, and then picking the K smallest $k-$vertex covers among all the generated k-vertex covers.

By looking at the algorithm **structure-vc**, we observe that all the vertices in the induced subgraph $G(R)$ have degree bounded by 2. Therefore, we first discuss how we deal with such graphs.

Lemma 3.3 *Let G be a graph of n vertices in which all vertices have degree bounded by 2. Then the K smallest k-vertex covers of G can be generated in time $O(Kkn)$.*

PROOF. Since all the vertices in G have degree bounded by 2, every connected component of G is either an isolated vertex, a simple path, or a simple cycle. Order the vertices of G to form a list $W = [v_1, v_2, \ldots, v_n]$ such that the vertices of each connected component of G appear consecutively in W. In particular, the vertices of a simple path appear in W in the order by which we traverse the path from an arbitrary end to the other end, and the vertices of a simple cycle appear in W in the order by which we traverse the entire cycle starting from an arbitrary vertex in the cycle. A vertex $v_i \in G$ is a *type-1 vertex* if it has degree 0, a *type-2 vertex* if it is in a simple path of length at least 1, and a *type-3 vertex* if it is in a simple cycle.

For each i, $1 \leq i \leq n$, let G_i be the subgraph of G induced by the vertex set $\{v_1, v_2, \ldots, v_i\}$. For each induced subgraph G_i, we build a list $L_i = [S_{i,0}, S_{i,1}, \ldots, S_{i,k}]$, where $S_{i,j}$ is a set of j-vertex covers for G_i, defined as follows:

(1) If v_i is of type-1, then $S_{i,j}$ is the set of the K smallest j-vertex covers for G_i (recall that by this we really mean "the K smallest j-vertex covers or all the j-vertex covers if the total number of j-vertex covers is smaller than K"— this remark also applies to the following discussion);

(2) If v_i is of type-2, then $S_{i,j}$ consists of two sets $S'_{i,j}$ and $S''_{i,j}$, where $S'_{i,j}$ contains the K smallest j-vertex covers of G_i that contain v_i, and $S''_{i,j}$ contains the K smallest j-vertex covers of G_i that do not contain v_i;

(3) If v_i is of type-3 and appears in a simple cycle $[v_h, \ldots, v_i, \ldots, v_t]$ in G, then $S_{i,j}$ consists of four sets $S'_{i,j}$, $S''_{i,j}$, $S'''_{i,j}$ and $S''''_{i,j}$, where $S'_{i,j}$ is the set of the K smallest j-vertex covers of G_i that contain both v_h and v_i, $S''_{i,j}$ is the set of the K smallest j-vertex covers of G_i that contain v_h but not v_i, $S'''_{i,j}$ is the set of the K smallest j-vertex covers of G_i that contain v_i but not v_h, and $S''''_{i,j}$ is the set of the K smallest j-vertex covers of G_i that contain neither v_h nor v_i.

Note that since each set $S_{i,j}$ contains at most $4K$ j-vertex covers, the set $S^0_{i,j}$ consisting of the K smallest j-vertex covers of the graph G_i can be constructed from $S_{i,j}$ in time $O(K)$.

The list L_1 can be trivially constructed: (1) if v_1 is of type-1, then all the sets $S_{1,j}$ are empty except $S_{1,0} = \{\emptyset\}$ and $S_{1,1} = \{(v_1)\}$; (2) if v_i is of type-2, then all the sets $S'_{1,j}$ and $S''_{1,j}$ are empty except $S''_{1,0} = \{\emptyset\}$ and $S'_{1,1} = \{(v_1)\}$; and (3) if v_i is of type-3, then all the sets $S'_{1,j}$, $S''_{1,j}$, $S'''_{1,j}$, and $S''''_{1,j}$ are empty except $S''''_{1,0} = \{\emptyset\}$ and $S'_{1,1} = \{(v_1)\}$.

Inductively, suppose that we have built the list L_{i-1}. To build the list L_i, we distinguish the following cases based on the type of the vertex v_i.

Case 1. The vertex v_i is of type-1. Then the graph G_i is the graph G_{i-1} plus an isolated vertex v_i. For each j, $0 \leq j \leq k$, let $S^0_{i-1,j}$ be the set of the K smallest j-vertex covers of the graph G_{i-1}, which can be constructed in time $O(K)$. Since each vertex cover of G_i is either a vertex cover of G_{i-1}, or a vertex cover of G_{i-1} plus the vertex v_i, the set $S_{i,j}$ in L_i can be constructed as follows: take each $(j-1)$-vertex cover of G_{i-1} from $S^0_{i-1,j-1}$ and add the vertex v_i to it to make a j-vertex cover of G_i. This gives a set F of K j-vertex covers for G_i. It is clear that the K smallest j-vertex covers of G_i must be contained in the set $F \cup S^0_{i-1,j}$, which is a set of $2K$ j-vertex covers for G_i. Thus, the K smallest j-vertex covers in the set $F \cup S^0_{i-1,j}$ make the set $S_{i,j}$. Each set $S_{i,j}$ can be constructed in time $O(K)$, and the list L_i can be constructed from the list L_{i-1} in time $O(Kk)$.

Case 2. The vertex v_i is of type-2. Then v_i is on a simple path $[v_h, \ldots, v_i, \ldots, v_t]$ in G of length at least 1. As in Case 1, for each j, let $S^0_{i-1,j}$ be the set of the K smallest j-vertex covers for G_{i-1}.

If $v_i = v_h$ is the first vertex on the path in the listing W, then the graph G_i is the graph G_{i-1} plus an isolated vertex v_i. Thus, the set $S'_{i,j}$ can be obtained from $S^0_{i-1,j-1}$ by adding the vertex v_i to each $(j-1)$-vertex cover of G_{i-1} in $S^0_{i-1,j-1}$. The set $S''_{i,j}$ is equal to the set $S^0_{i-1,j}$.

If $h < i$ and v_i is not the first vertex on the path in the listing W, then the graph G_i is the graph G_{i-1} plus the vertex v_i and the edge $[v_{i-1}, v_i]$. Therefore, each vertex cover of G_i is either a vertex cover of G_{i-1} plus v_i, or a vertex cover of G_{i-1} that contains v_{i-1}. Thus, the set $S'_{i,j}$ is again obtained from

$S^0_{i-1,j-1}$ by adding the vertex v_i to each $(j-1)$-vertex cover of G_{i-1} in $S^0_{i-1,j-1}$. On the other hand, now the set $S''_{i,j}$ is equal to the set $S'_{i-1,j}$.

Again in this case, the list L_i can be constructed from the list L_{i-1} in time $O(Kk)$.

Case 3. The vertex v_i is of type-3. Then v_i is on a simple cycle $[v_h, \ldots, v_i, \ldots, v_t]$ of G. Again for each j, let $S^0_{i-1,j}$ be the set of the K smallest j-vertex covers of G_{i-1}.

If $v_i = v_h$ is the first vertex on the cycle in the listing W, then the graph G_i is the graph G_{i-1} plus an isolated vertex v_i. Thus, the set $S'_{i,j}$ can be obtained from $S^0_{i-1,j-1}$ by adding the vertex v_i to each $(j-1)$-vertex cover of G_{i-1} in $S^0_{i-1,j-1}$, and the set $S''''_{i,j}$ is equal to the set $S^0_{i-1,j}$. By the definition, the sets $S''_{i,j}$ and $S'''_{i,j}$ are empty.

If $h < i < t$, then the graph G_i is the graph G_{i-1} plus the vertex v_i and the edge $[v_{i-1}, v_i]$. Therefore, the set $S'_{i,j}$ can be obtained by adding the vertex v_i to each $(j-1)$-vertex cover in the union $S'_{i-1,j-1} \cup S''_{i-1,j-1}$ then selecting the K smallest ones; the set $S''_{i,j}$ is equal to the set $S'_{i-1,j}$; the set $S'''_{i,j}$ is obtained by adding the vertex v_i to each $(j-1)$-vertex cover in the union $S'''_{i-1,j-1} \cup S''''_{i-1,j-1}$ then selecting the K smallest ones; and the set $S''''_{i,j}$ is equal to the set $S'''_{i-1,j}$.

If $v_i = v_t$ is the last vertex on the cycle in the listing W, then the graph G_i is the graph G_{i-1} plus the vertex v_i and two edges $[v_h, v_i]$ and $[v_{i-1}, v_i]$. In this case, the set $S'_{i,j}$ can be obtained by adding the vertex v_i to each $(j-1)$-vertex cover in the union $S'_{i-1,j-1} \cup S''_{i-1,j-1}$ then selecting the K smallest ones; the set $S''_{i,j}$ is equal to the set $S'_{i-1,j}$; the set $S'''_{i,j}$ is obtained by adding the vertex v_i to each $(j-1)$-vertex cover in the union $S'''_{i-1,j-1} \cup S''''_{i-1,j-1}$ then selecting the K smallest ones; and the set $S''''_{i,j}$ is empty because $[v_h, v_i]$ is an edge in G_i.

The correctness of the above constructions can be easily verified using the definitions of the sets $S'_{i,j}$, $S''_{i,j}$, $S'''_{i,j}$, and $S''''_{i,j}$. Moreover, it is also easy to see that the list L_i can be constructed from the list L_{i-1} in time $O(Kk)$.

Summarizing all the above, we conclude that the list L_n can be constructed in time $O(Kkn)$. Now the K smallest k-vertex covers of the graph $G = G_n$ can be easily obtained in time $O(K)$ from the set $S_{n,k}$ in the list L_n. This completes the proof of the lemma. □

Now it should be obvious in principle how we can generate the K smallest k-vertex covers for the graph G: they can be obtained by first generating the K smallest consistent k-vertex covers, for each triple in $\mathcal{L}(G,k)$. However, by applying some enumeration tricks, we can significantly speedup this enumeration process, as shown in the following theorem.

Theorem 3.4 *Let (G,k) be an instance of the* WEIGHTED VERTEX COVER *problem, and let $\mathcal{L}(G,k)$ be the structure returned by the algorithm* **structure-vc** *on (G,k). Then the K smallest k-vertex covers of the graph G can be generated in time $O(1.47^k n + 1.22^k Kn)$.*

PROOF. Let (I,O,R) be a triple in $\mathcal{L}(G,k)$ and let $k_1 = k - |I|$. By Lemma 3.3, the K smallest k_1-vertex covers of the induced subgraph $G(R)$ can be constructed in time $O(Kk_1n)$. The vertex set I, plus each of these k_1-vertex covers for $G(R)$, form one of the K smallest k-vertex covers consistent with (I,O,R) for the graph G. Thus, the K smallest k-vertex covers of G consistent with (I,O,R) can be constructed in time $O(Kkn)$. Moreover, by Lemma 3.2, every k-vertex cover of G is consistent with a triple in $\mathcal{L}(G,k)$. Therefore, if we generate the K smallest consistent k-vertex covers for each triple in $\mathcal{L}(G,k)$, and pick the K smallest among all these generated k-vertex covers, then we will obtain the K smallest k-vertex covers for the graph G.

Let L be the total number of triples in $\mathcal{L}(G,k)$.

If $K > \sqrt{L}$, then let $K' = K/\sqrt{L}$. For each triple (I,O,R) in $\mathcal{L}(G,k)$, construct the K' smallest k-vertex covers consistent with (I,O,R), and form the set S_1 of the K smallest k-vertex covers among all these LK' k-vertex covers. This takes time $O(LK'kn) = O(\sqrt{L}Kkn)$. For each triple (I,O,R) whose K' smallest consistent k-vertex covers are not all in the set S_1, only those k-vertex covers consistent with (I,O,R) that are already in the set S_1 can be possibly among the K smallest k-vertex covers of the graph

G. Therefore, we will discard each triple whose K' smallest consistent k-vertex covers are not all in the set S_1 from any further consideration. Since no k-vertex cover is consistent with more than one triple in $\mathcal{L}(G,k)$, there are at most $\sqrt{L}$ triples in $\mathcal{L}(G,k)$ for which the K' smallest consistent k-vertex covers are all in the set S_1. Therefore, the number L_2 of the remaining triples to be considered is bounded by $\sqrt{L}$. Now in time $O(L_2Kkn) = O(\sqrt{L}Kkn)$, we can apply Lemma 3.3 to each of these L_2 triples to generate the K smallest consistent k-vertex covers with this triple. Let S_2 be the set of all k-vertex covers constructed in this step. Then $|S_2| \leq L_2K$. Let $S = S_1 \cup S_2$ and note that $|S| = (L_2+1)K = O(L_2K)$. The K smallest k-vertex covers in S are the K smallest k-vertex covers of the graph G, which can be found in time $O(L_2K)$. In summary, in this case, the K smallest k-vertex covers of the graph G can be generated in time $O(\sqrt{L}Kkn)$.

If $K \leq \sqrt{L}$, then by letting $K' = 1$, and using a similar procedure to the above, we can show that in this case the K smallest k-vertex covers of G can be generated in time $O(Lkn + \sqrt{L}Kkn)$.

In conclusion, given the structure $\mathcal{L}(G,k)$, the K smallest k-vertex covers of the graph G can be generated in time $O(Lkn + \sqrt{L}Kkn)$. The theorem now follows from Theorem 3.1 because $L \leq 1.466^k$ and hence $Lk = O(1.47^k)$, and $\sqrt{L}k = O(1.22^k)$. □

Corollary 3.5 *The* WEIGHTED VERTEX COVER *problem is linearly fixed-parameter enumerable. More specifically, given an instance (G,k) and a nonnegative integer K, the K smallest k-vertex covers of the graph G can be generated in time $O(1.47^k n + 1.22^k Kn)$, where n is the number of vertices in the graph.*

4 Effective enumeration based on color coding

The *color coding* technique [2] is very powerful and useful in the development of efficient parameterized algorithms. In particular, the technique has been used in developing improved parameterized algorithms for the k-PATH problem [2, 7], for matching and set packing problems [15, 24], and for problems in computational biology [30]. In this section, we show that the color coding technique is also very helpful in developing effective algorithms for the structure-generation phase of enumeration algorithms for parameterized NP optimization problems. We will illustrate this fact by presenting an enumeration algorithm for the k-PATH problem. A simple path in a graph G is a *k-path* if it contains exactly k vertices. The *weight* of a path in a weighted graph is the sum of the weights of the vertices in the path. The problem can be formally defined as follows.

> WEIGHTED k-PATH: given a weighted graph G and integers k and K, generate the K largest k-paths in G.

4.1 The structure algorithm

A *k-coloring* of a set S is a function from S to $\{1,2,\ldots,k\}$. A collection $\mathcal{F}$ of k-colorings of S is a *k-color coding scheme* for S if for any subset W of k elements in S, there is a k-coloring f_W in $\mathcal{F}$ such that no two elements in W are assigned the same color by f_W. The *size* of the k-color coding scheme $\mathcal{F}$ is equal to the number of k-colorings in $\mathcal{F}$. Alon, Yuster, and Zwick [2] showed that there is a k-color coding scheme of size $2^{O(k)}n$ for a set of n elements. This bound has been improved recently to $O(6.4^k n)$ [7]. In the following discussion, we will assume a k-color coding scheme $\mathcal{F}$ of size $O(6.4^k n)$ for a set of n elements.

On a given instance (G,k) of the WEIGHTED k-PATH problem, where G is a graph of n vertices, the structure algorithm for WEIGHTED k-PATH produces $h = O(6.4^k n)$ copies $\{G_1, G_2, \ldots, G_h\}$ of the graph G, where each copy G_i is colored by a k-coloring in the k-color coding scheme $\mathcal{F}$. Note that by the definition of k-color coding schemes, every k-path in the graph G has all its vertices colored with different colors in at least one of these copies of the graph G. The list $\tau_{G,k} = \{G_1, G_2, \ldots, G_h\}$ is the structure returned by the structure algorithm for the WEIGHTED k-PATH problem, whose running time is $O(6.4^k n^2)$.

4.2 The enumeration algorithm

The enumeration algorithm for WEIGHTED k-PATH is a careful and non-trivial generalization of the dynamic programming algorithm described in [2] that finds a k-path in a k-colored graph. We first discuss how we deal with each copy G_i of the colored graphs in the list $\tau_{G,k}$. We say that a k-path in a k-colored graph is *properly colored* if no two vertices on the path are colored with the same color. Consider the algorithm given in Figure 2, where $c(w)$ denotes the color assigned to the vertex w in the k-colored graph G. Inductively, before the j-th execution of the loop in steps 2.1-2.5 of the algorithm, we assume that each vertex w is associated with a collection $\mathcal{C}_j(w)$ of pairs (C, P), where C is a subset of j colors in the k-color set, and P is the set of up to K largest properly colored j-paths ending at w that use exactly the colors in C. Then the j-th execution of steps 2.1-2.5 will produce a similar collection $\mathcal{C}_{j+1}(w)$ for $(j+1)$-paths in G based on the collection $\mathcal{C}_j(w)$ of j-paths.

Algorithm **enumerate-path**(G, k, K)

INPUT: *a k-colored graph G, and integers k and K*

1. **for** each vertex w in G **do** $\mathcal{C}_1(w) = [(\{c(w)\}; \{w\})]$;
2. **for** $j = 1$ **to** $k-1$ **do** 2.1. **for** each edge $[v, w]$ in G **do**
   2.2. **for** each pair (C, P) in $\mathcal{C}_j(v)$ **do**
   2.3. **if** $(c(w) \notin C)$ **then**
   2.4. construct $|P|$ $(j+1)$-paths ending at w by extending each path in P to the vertex w;
   2.5. add these $(j+1)$-paths to P' in the pair $(C \cup \{c(w)\}, P')$ in $\mathcal{C}_{j+1}(w)$ and only keep the K largest $(j+1)$-paths in P';

Figure 2: The enumeration algorithm for WEIGHTED k-PATH.

Note that at the end of the algorithm **enumerate-path**(G, k, K), for each vertex w in the k-colored graph G, the collection $\mathcal{C}_k(w)$ is either empty, or contains a single pair (C, P), where C is the set of all k colors and P is a set of properly colored k-paths ending at w in G.

Lemma 4.1 (Lemma 7.2, Appendix) *For each vertex w in the k-colored graph G, the pair (C, P) in the collection $\mathcal{C}_k(w)$ returned by the algorithm* **enumerate-path**(G, k, K) *contains the K largest properly colored k-paths ending at w. The running time of the algorithm* **enumerate-path**(G, k, K) *is $O(2^k k^2 n^2 K)$.*

Theorem 4.2 *[Theorem 7.3, Appendix] Given the structure $\tau_{G,k}$ and an integer K, the K largest k-paths in the graph G can be generated in time $O(12.8^k k^2 n^3 K)$.*

Using more sophisticated enumeration techniques, we can show that the above running time can be improved. We have the following theorem whose proof is omitted for lack of space.

Theorem 4.3 *Given the structure $\tau_{G,k}$ and an integer K, the K largest k-paths in the graph G can be generated in time $O(12.8^k + 6.4^k k^2 n^3 K)$.*

Corollary 4.4 *The* WEIGHTED k-PATH *problem is linearly fixed-parameter enumerable.*

Remark. Corollary 4.4 may look a bit surprising. Although the k-PATH problem is fixed-parameter tractable [2], Flum and Grohe [17] proved that counting the number of k-paths in a graph G is $\#W[1]$-hard. This means that it is unlikely that there is an algorithm of running time $f(k)n^{O(1)}$, for some function f, that can count the number of k-paths in a graph of n vertices precisely. On the other hand, Corollary 4.4 shows that enumerating the K largest k-paths in the graph G takes time $f(k)n^{O(1)}K$, where f is a function independent of n. This means that in a feasible amount of average time $f(k)n^{O(1)}$ per path, we can generate the paths in decreasing order of the path weights, which shows that the hardness of the problem of counting the number of k-paths is mainly due to the (possible) large number of such paths in the graph.

5 Effective enumeration based on tree decomposition

The concept of the tree decomposition of a graph has played an important role in the study of algorithmic graph theory [5] and in developing efficient exact and parameterized algorithms for graph problems on planar graphs (see [1]). In this section, we discuss how this approach can be used to develop algorithms for the structure-generation phase of enumeration algorithms for NP optimization problems.

A set D of vertices in a graph G is a *dominating set* of G if every vertex in G is either in D or adjacent to a vertex in D. A dominating set of k vertices will be called a *k-dominating set*. Given a weighted graph, the *weight* of a dominating set D is the sum of the weights of the vertices in D. Our running example will be the following problem.

> WEIGHTED PLANAR DOMINATING SET. Given (G, k), where G is a weighted planar graph and k is a nonnegative integer, and a nonnegative integer K, generate the K smallest k-dominating sets in the graph G.

5.1 The structure algorithm

The reader is referred to Subsection 7.1 in the appendix for the terminology in this section and the proofs of the theorems.

Theorem 5.1 (Section 7.1, Theorem 7.6, Appendix) *There is an $O(\sqrt{k}n)$ time algorithm that given a planar graph G on n vertices and a positive integer k, either constructs a nice tree decomposition $(\mathcal{V}, \mathcal{T})$ for G of width $O(\sqrt{k})$ and $O(n)$ nodes, or reports that no dominating set for G of size bounded by k exists.*

The structure $\tau_{G,k}$ is simply the nice tree decomposition $(\mathcal{V}, \mathcal{T})$ obtained by the above theorem.

5.2 The enumeration algorithm

Given the nice tree decomposition $\tau_{G,k} = (\mathcal{V}, \mathcal{T})$, we can generate the K smallest k-dominating sets in the graph G using dynamic programming. We have the following theorem.

Theorem 5.2 (Theorem 7.7, Appendix) *Given a planar graph G on n vertices and two nonnegative integers k and K, the K smallest k-dominating sets in G can be generated in time $2^{O(\sqrt{k})}nK \log K$.*

Corollary 5.3 WEIGHTED PLANAR DOMINATING SET *is fixed-parameter enumerable.*

6 Final remarks

We have introduced the concept of effective enumerability, or more precisely, fixed-parameter enumerability of NP optimization problems. Our objective is solving enumeration problems that have an increasing demand in computational science. We outline below some of the main differences between our approach and some of the previously-proposed approaches.

• Whereas previous models studied the complexity of counting or enumerating all (optimal) solutions, we study the complexity of generating the first K best solutions of size k, parameterized by both the solution size k, and the number of solutions K. In consequence, some of the results obtained using the current approach are orthogonal to some of the results obtained previously.

• The current approach meets the practical needs for a large number of computational problems [18, 22, 28, 30]. The current approach stipulates that solutions of small size be reported efficiently

by requiring the exponential factor in the running time of the algorithm to be a function of the solution size only. It also stipulates that the overhead due to the required additional computation of the first K solutions should not exceed a polynomial (in K) multiplicative factor.

• The developed framework has a positive goal, in the sense that it intends to design and develop effective techniques that allow us to enumerate a certain number of best solutions to a problem, even though enumerating or simply counting all the solutions might be an intractable problem. In particular, the developed framework measures the intrinsic complexity of generating each single solution, and avoids running into the realm of infeasibility resulting from the large number of solutions.

• The developed framework targets mainly NP-hard optimization problems, and is a systematic approach rather than being a problem-specific one.

We finally indicate that even though we illustrated our results by picking specific problems for each technique, each of the considered problems is a representative for a huge set of problems to which the technique is applicable as well.

References

[1] J. Alber, H. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier, Fixed parameter algorithms for dominating set and related problems on planar graphs, *Algorithmica* **33**, pp. 461-493, (2002).

[2] N. Alon, R. Yuster, and U. Zwick, Color-coding, *Journal of the ACM* **42**, pp. 844-856, (1995).

[3] V. Arvind and V. Raman, Approximation algorithms for some parameterized counting problems, ISAAC'02, pp. 453-464, (2002).

[4] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, *Complexity and Approximation, Combinatorial optimization problems and their approximability properties*, Springer-Verlag, 1999.

[5] H. Bodlaender, Treewidth: algorithmic techniques and results, *Lecture Notes in Computer Science* **1295**, pp. 19-36, (1997).

[6] J. Chen, I. A. Kanj, and W. Jia, Vertex cover: further observations and further improvements, *Journal of Algorithms 41*, pp. 280-301, (2001).

[7] J. Chen, S. Lu, S.-H. Sze, and F. Zhang, Improved algorithms for the k-path problem, *Manuscript*, (2005).

[8] C. Chegireddy and H. Hamacher, Algorithms for finding K-best perfect matchings, *Discrete Applied Mathematics 18*, pp. 155-165, (1987).

[9] S. Chien, A determinant-based algorithm for counting perfect matching in a general graph, SODA'04, pp. 728-735, (2004).

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, McGraw-Hill Book Company, Boston, MA, 2001.

[11] V. Dahllof and P. Jonsson, An algorithm for counting maximum weighted independent sets and its applications, SODA'02, pp. 292-298, (2002).

[12] M. Dyer, Approximate counting by dynamic programming, STOC'03, pp. 693-699, (2003).

[13] R.G. Downey and M.R. Fellows, *Parameterized Complexity*, Springer-Verlag, 1999.

[14] D. Eppstein, Finding the k shortest paths, *SIAM J. Computing* **28-2**, pp. 652-673, (1998).

[15] M. Fellows, C. Knauer, N. Nishimura, P. Ragde, F. Rosamond, U. Stege, D. Thilikos, and S. Whitesides, Faster fixed-parameter tractable algorithms for matching and packing problems, *Lecture Notes in Computer Science 3221*, (2004), pp. 311-322.

[16] H. Fernau, On parameterized enumeration, COCOON'02, pp. 564-573, (2002).

[17] J. Flum and M. Grohe, The parameterized complexity of counting problems, *SIAM Journal on Computing 33*, pp. 892-922, (2004).

[18] J. Gramm and R. Niedermeier, Quartet inconsistency is fixed parameter tractable, CPM'01, pp. 241-256, (2001).

[19] V. Guruswami, *List decoding of error-correcting codes*, *Lecture Notes in Computer Science* **3282**, 2005.

[20] H. Hunt III, M. Marathe, V. Radhakrishnan, and R. Stearns, The complexity of planar counting problems, *SIAM Journal on Computing 27*, pp. 1142-1167, (1998).

[21] S. Kapoor and H. Ramesh, Algorithms for enumerating all spanning trees of undirected and weighted graphs, *SIAM Journal on Computing 25*, pp. 247-265, (1995).

[22] B. Kelley, R. Sharan, R. Karp, T. Sittler, D. Root, B. Stockwell, and T. Ideker, Conserved pathways within bacteria and yeast as revealed by global protein network alignment, *Proc. Natl. Acad. Sci. USA* **100**, pp. 11394-11399, (2003).

[23] T. Kloks, Treewidth, computations and approximations, *Lecture Notes in Computer Science* **842**, (1994).

[24] I. Koutis, A faster parameterized algorithm for set packing, *Information Processing Letters 94*, (2005), pp. 7-9.

[25] Y. Matsui and T. Matsui, http://dmawww.epfl.ch/roso.mosaic/kf/enum/comb/combenum.html.

[26] S. Nakano, Efficient generation of triconnected plane triangulations, COCOON'01, pp. 131-141, (2001).

[27] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[28] P. Pevzner and S.-H. Sze, Combinatorial approaches to finding subtle signals in DNA sequences, ISMB'2000, pp. 269-278, (2000).

[29] S. Ravi and H. Hunt III, An application of the planar separator theorem to counting problems, *Information Processing Letters 25*, pp. 317-321, (1987).

[30] J. Scott, T. Ideker, R. Karp, and R. Sharan, Efficient algorithms for detecting signaling pathways in protein interaction networks, RECOMB 2005, to appear.

[31] E. Tomita, A. Tanaka, and H. Takahashi, The worst-case time complexity for generating all maximal cliques, COCOON'04, pp. 161-170, (2004).

[32] L. Valiant, The complexity of computing the permanent, *Theoretical Computer Science 8*, pp. 189-201, (1979).

7 Appendix

Theorem 7.1 *On an input (G, k), the algorithm* **structure-vc** *runs in time $O(1.47^k n)$, and returns a collection $\mathcal{L}(G, k)$ of at most 1.466^k triples.*

PROOF. We first prove the second claim. Let $L(k)$ be the number of triples in the collection $\mathcal{L}(G, k)$ returned by the algorithm **structure-vc** on the input (G, k). If the input (G, k) satisfies the conditions in step 1 or step 2, then $L(k) \leq 1$. In particular, $L(k) \leq 1$ for $k \leq 0$. Otherwise, the value $L(k)$ satisfies the recurrence relation $L(k) \leq L(k-1) + L(k-d)$, where $d \geq 3$. Using the standard techniques for solving such recurrence relations, we get $L(k) \leq \alpha^k$, where $\alpha = 1.4655 \cdots < 1.466^k$, is the unique positive root of the polynomial $x^k - x^{k-1} - x^{k-3}$. This proves the second claim of the theorem.

Let $T(k, G)$ be the running time of the algorithm **structure-vc** on the input (G, k). If (G, k) satisfies the conditions in step 1 or step 2, then $T(k, G) = O(kn)$ (recall that we can assume that the size of the graph G is $O(kn)$). In particular, $T(k, G) = O(kn)$ for $k \leq 0$. Otherwise, the value $T(k, G)$ satisfies the following recurrence relation

$$T(k, G) \leq T(k-1, G-v) + T(k-d, G-(v \cup N(v))) + O(1.466^k n),$$

where v is a vertex of degree $d \geq 3$ in G picked by the algorithm in step 3, $N(v)$ is the set of neighbors of v in G, and the term $O(1.466^k n)$ is an upper bound on the time for constructing the graphs $G_1 = G - v$ and $G_2 = G - (v \cup N(v))$, and for constructing the collection $\mathcal{L}(G, k)$ from the collections $\mathcal{L}(G_1, k-1)$ and $\mathcal{L}(G_2, k-d)$ (here we have used the fact that $\mathcal{L}(G, k)$ contains at most 1.466^k triples). Using induction on k and n, it can be easily verified that $T(k, G) = O(1.466^k kn) = O(1.47^k n)$. □

Lemma 7.2 *For each vertex w in the k-colored graph G, the pair (C, P) in the collection $\mathcal{C}_k(w)$ returned by the algorithm* **enumerate-path**(G, k, K) *contains the K largest properly colored k-paths ending at w. The running time of the algorithm* **enumerate-path**(G, k, K) *is $O(2^k k^2 n^2 K)$.*

PROOF. It is not difficult to prove by induction on j that after the j-th execution of the loop 2.1-2.5, for each vertex w, the collection $\mathcal{C}_{j+1}(w)$ contains the K largest properly colored $(j+1)$-paths ending at w. We note that it can never be the case that a pair $(C \cup \{c(w)\}, P')$ in the collection $\mathcal{C}_{j+1}(w)$ already contains a path p and step 2.5 of the algorithm adds again the path p to $(C \cup \{c(w)\}, P')$. This is because each edge $[v, w]$ is considered exactly once in each execution of the loop 2.1-2.5, and only when the edge $[v, w]$ is considered in step 2.1, $(j+1)$-paths *whose second ending vertex is w* can be added to the pair $(C \cup \{c(w)\}, P')$ in $\mathcal{C}_{j+1}(w)$.

Now we consider the time complexity of the algorithm. Since each of the sets (C, P) and $(C \cup \{c(w)\}, P')$ contains at most K paths, step 2.5 can be executed in time $O(Kk)$ by first merging the $|P|$ paths constructed in step 2.4 with the set $(C \cup \{c(w)\}, P')$, then identifying the K-th largest path in the merged set so that the paths smaller than this path can all be removed from the set (note that by the above discussion, all the paths in the set are distinct). This will keep the size of $(C \cup \{c(w)\}, P')$ bounded by K. Since each collection $\mathcal{C}_j(v)$ may have up to 2^k j-subsets of colors, we conclude that the running time of the algorithm is bounded by $O(2^k k^2 n^2 K)$. □

Theorem 7.3 *Given the structure $\tau_{G,k}$ and an integer K, the K largest k-paths in the graph G can be generated in time $O(12.8^k k^2 n^3 K)$.*

PROOF. From the output of the algorithm **enumerate-path**(G, k, K), for each vertex w in G, we get the $O(K)$ largest properly colored k-paths ending at w. Collecting these paths over all vertices in G, we get a set P of $O(Kn)$ properly colored k-paths that obviously contains the K largest properly colored k-paths in G. From the $O(Kn)$ k-paths in P, we can find the K largest properly colored k-paths in time $O(Kn)$ by first identifying the K-th largest path in the set P in time $O(Kn)$, and then removing all the smaller paths in the set (again, note that all these paths are distinct).

Since there are $O(6.4^k n)$ k-colored graphs in the list $\tau_{G,k} = \{G_1, G_2, \ldots, G_h\}$, we apply Lemma 7.2 to each of these k-colored graphs. This takes $O(12.8^k k^2 n^3 K)$ time. We get a set P' of $O(6.4^k nK)$ k-paths, each of them is properly colored in some of the k-colored graphs in the list $\tau_{G,k}$. Since the k-colorings we used to color the graph vertices come from the k-color coding scheme $\mathcal{F}$, every k-path among the K largest k-paths in G is among the K largest properly colored k-paths in some k-colored graph G_i in the list $\tau_{G,k}$, and hence is contained in the set P'. Therefore, selecting the K largest k-paths in P' gives the K largest k-paths in the graph G. To do this, we first use BucketSort to sort all the k-paths in P' (using the vertex labels along a path as the key for the path). This sorting takes time $O(6.4^k knK)$ and removes duplicate copies of each path in P'. Then we find the K-th largest k-path in the remaining set in P' in time $O(6.4^k nK)$, and identify the K largest distinct k-paths in the set P' in time $O(6.4^k nK)$. Combining all the above, the theorem follows. □

7.1 Preliminaries and proofs for Section 5 in the paper

We start by reviewing some related terminologies. For a more detailed discussion on the tree decomposition of a graph, the reader is referred to [5].

Definition Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair $(\mathcal{V}, \mathcal{T})$, where $\mathcal{V}$ is a collection of subsets of V satisfying $\bigcup_{X_i \in \mathcal{V}} = V$, and $\mathcal{T}$ is a tree whose node set is $\mathcal{V}$, such that:

1. for every edge $[u, v] \in E$, there is an $X_i \in \mathcal{V}$, such that $\{u, v\} \subseteq X_i$;
2. for all $X_i, X_j, X_k \in \mathcal{V}$, if the node X_j lies on the path between the nodes X_i and X_k in the tree $\mathcal{T}$, then $X_i \cap X_k \subseteq X_j$.

The *width* of the tree decomposition $(\mathcal{V}, \mathcal{T})$ is defined to be $\max\{|X_i| \mid X_i \in \mathcal{V}\} - 1$. The *treewidth* of the graph G is the minimum width over all tree decompositions of G.[2]

A tree decomposition $(\mathcal{V}, \mathcal{T})$ is *nice* if it satisfies the following conditions:

1. Each node in the tree $\mathcal{T}$ has at most two children;
2. If a node X_i has two children X_j and X_k in the tree $\mathcal{T}$, then $X_i = X_j = X_k$;
3. If a node X_i has only one child X_j in the tree $\mathcal{T}$, then either $|X_i| = |X_j| + 1$ and $X_j \subset X_i$, or $|X_i| = |X_j| - 1$ and $X_i \subset X_j$.

For a given graph of treewidth k, a tree decomposition of width k for G can be constructed in time $f(k)n$; however, the function value $f(k)$ is very large even for very small values of k [5]. Alternatively, for planar graphs that have k-dominating sets, tree decompositions of small width can be constructed using more practical algorithms, as stated in the following theorem that can be found in [1].

Theorem 7.4 ([1]) *If a planar graph G of n vertices has a k-dominating set, then a tree decomposition for G of treewidth $O(\sqrt{k})$ and $O(n)$ nodes can be constructed in time $O(\sqrt{k}n)$.*

The proof of the following theorem can be found in [23].

Theorem 7.5 *There is an algorithm that, for a given tree decomposition of a graph G of treewidth h and $O(n)$ nodes, constructs a nice tree decomposition of treewidth h and $O(n)$ nodes in linear time.*

Theorem 7.6 *There is an $O(\sqrt{k}n)$ time algorithm that given an instance (G, k) of* WEIGHTED PLANAR DOMINATING SET *where G has n vertices, either constructs a nice tree decomposition $(\mathcal{V}, \mathcal{T})$ for G of width $O(\sqrt{k})$ and $O(n)$ nodes, or reports that no dominating set for G of size bounded by k exists.*

PROOF. On an instance (G, k) of WEIGHTED PLANAR DOMINATING SET, we first call the algorithm of running time $O(\sqrt{k}n)$ in Theorem 7.4. If the algorithm does not return a desired tree decomposition,

[2] To avoid confusion, we will use "nodes" for the tree in the tree decomposition, and use "vertices" for the underlying graph.

then G has no k-dominating set. Otherwise, the returned tree decomposition $(\mathcal{V}, \mathcal{T})$ has width $O(\sqrt{k})$ and $O(n)$ nodes. Now we can apply Theorem 7.5 to the tree decomposition $(\mathcal{V}, \mathcal{T})$ to obtain a nice tree decomposition of width $O(\sqrt{k})$ and $O(n)$ nodes. The total running time is $O(\sqrt{k}n)$. □

Theorem 7.7 *Given a planar graph G on n vertices and two nonnegative integers k and K, the K smallest k-dominating sets in G can be generated in time $2^{O(\sqrt{k})}nK \log K$.*

PROOF. Consider the nice tree decomposition $\tau_{G,k} = (\mathcal{V}, \mathcal{T})$. Let $X_i = \{v_1, \ldots, v_q\}$ be a node in the tree $\mathcal{T}$, where each v_j, $j = 1, \cdots, q$, is a vertex in G. Let Y_i be the set of all the vertices in G that are contained in the nodes of the subtree rooted at X_i in the tree $\mathcal{T}$, and note that $X_i \subseteq Y_i$. Let $A = [c(v_1), \ldots, c(v_q)]$ be any assignment to the vertices in X_i that assigns each vertex v_j, $j = 1, \cdots, q$, a value $c(v_j) \in \{-1, 0, 1\}$, and let D' be a subset of Y_i. We say that the set D' is *consistent* with the value assignment $A = [c(v_1), \ldots, c(v_q)]$ for X_i if and only if the three following conditions are satisfied:

1. $c(v_j) = 1$ if v_j is in D';
2. $c(v_j) = -1$ if v_j is not in D' but is adjacent to a vertex in D';
3. $c(v_j) = 0$ if v_j is not in D' and is not adjacent to any vertex in D'.

Note that there can be many subsets of Y_i which are consistent with the same value assignment A for X_i. For a value assignment A to X_i and an integer $r \leq k$, a subset D' in Y_i is an (A, r)-*subset* of Y_i if D' has exactly r vertices, D' is consistent with the value assignment A, and for each vertex w in $Y_i - X_i$, either w is in D' or w is adjacent to a vertex in D'. Intuitively, an (A, r)-subset is a candidate for a dominating set for the graph G that has r vertices in Y_i.

For a node X_i that contains q vertices, there are 3^q possible value assignments to X_i. For each value assignment A to X_i, we associate with A a collection of k lists $\mathcal{L}_A = [L_1, \ldots, L_k]$, where L_r is a list containing the K smallest (A, r)-subsets of Y_i. Observe that since no vertex w in $Y_i - X_i$ is adjacent to any vertex not in Y_i, the selection of the vertices in a dominating set from the set $V - Y_i$ is totally independent of the status of w, but may (only) depend on the status of the vertices in X_i. Therefore, if in each list L_r we record the K smallest (A, r)-subsets of Y_i that are consistent with the value assignment A, then for the k-dominating sets of G consistent with the value assignment A, only these (A, r)-subsets of Y_i can be subsets of the K smallest k-dominating sets of G.

For each node X_i in the tree $\mathcal{T}$, and for each valid value assignment A to X_i, we construct the corresponding collection $\mathcal{L}_A$. Using dynamic programming, we proceed from the leaves of the tree $\mathcal{T}$ in a bottom-up fashion. For each leaf X_i of q vertices, we construct each of the 3^q value assignments to X_i. Note that in this case, $Y_i = X_i$, so it is fairly easy to determine if a value assignment is valid, and for each valid value assignment A, the collection $\mathcal{L}_A = [L_1, \ldots, L_k]$ can be directly constructed. As a matter of fact, for each assignment A that assigns s vertices in X_i the value 1, the collection $\mathcal{L}_A = [L_1, \ldots, L_k]$ consists of empty lists L_j, $j = 1, \cdots, k$, with the exception of the list L_s that could possibly contain one subset: the subset of the s vertices in X_i assigned the value 1 by A.

Now we discuss how the construction proceeds. Suppose that the value assignments and the related collections have been constructed for all the children of a node X_i in the tree $\mathcal{T}$. To construct the value assignments and the corresponding collections for the node X_i, since $\tau_{G,k}$ is a nice tree decomposition, there are three cases to be distinguished.

Case 1. X_i has a single child X_j, $|X_j| = |X_i| - 1$, and $X_j \subset X_i$.

Let $v \in X_i - X_j$, then $v \notin Y_j$. Now for each value assignment A_j to X_j, we can get three different value assignments for X_i by assigning $c(v) = -1$, 0, and 1, respectively. Since v is not adjacent to any vertex in $Y_j - X_j$, it is easy to check if these value assignments are valid. For example, if we assign $c(v) = 1$, then any vertex w in X_i that is adjacent to v in G cannot have value $c(w) = 0$. Now to construct the corresponding collection $\mathcal{L}_{A_i}$, for a valid value assignment A_i for X_i, suppose that $c(v) = 1$ and that A_i is obtained from a value assignment A_j for X_j. Then each (A_j, r)-subset in the collection $\mathcal{L}_{A_j}$ plus the vertex v becomes a $(A_i, r + 1)$-subset in the collection $\mathcal{L}_{A_i}$. The other cases for the other values of $c(v)$

can be handled similarly. Finally, we only keep the smallest K (A, r)-subsets in the list L_r in $\mathcal{L}_{A_i}$, if the list contains more than K subsets.

Case 2. X_i has a single child X_j, $|X_j| = |X_i| + 1$, and $X_i \subset X_j$.

Let $v \in X_j - X_i$, then any value assignment A to X_j with the value $c(v)$ dropped makes a value assignment to X_i. Again, we can check if these value assignments are valid. For example, if $c(v) = 0$ then the value assignment to X_j with $c(v)$ dropped is invalid. Each (A_j, r)-subset in the collection $\mathcal{L}_{A_j}$, where A_j is a value assignment to X_j becomes a (A_i, r)-subset in the collection $\mathcal{L}_{A_i}$, where A_i is a valid value assignment to X_i obtained from the value assignment A_j with $c(v)$ dropped. Again we only keep the K smallest subsets in a list if a list contains more than K subsets.

Case 3. X_i has two children X_j and X_h and $X_j = X_i = X_h$.

We say that two values assignments A_j and A_h, where A_j is an assignment to X_j and A_h is an assignment to X_h, are *mergeable* if for any vertex v in X_j (and X_h), either v has the same value in A_j and A_h, or one of A_j, A_h assigns v the value -1 and the other assigns v the value 0. A value assignment A_i can be obtained form two mergeable values assignments A_j and A_h as follows. The value of v in A_i is equal to its value in both A_j and A_h when the two values are equal, and is equal to -1 when the values are not equal. It is not difficult to verify that this value assignment A_i is consistent with the definitions of value assignments to the nodes in $\mathcal{T}$ and to the subsets associated with these value assignments. Moreover, the subsets in the collections $\mathcal{L}_{A_j}$ and $\mathcal{L}_{A_h}$ are also merged into subsets in the collection $\mathcal{L}_{A_i}$ accordingly. More specifically, an (A_j, r)-subset in $\mathcal{L}_{A_j}$ and an (A_h, r')-subset in $\mathcal{L}_{A_h}$ are merged into an (A_i, r'')-subset in $\mathcal{L}_{A_i}$, where r'' is equal to $r + r'$ minus the number of vertices in X_i that were assigned the value 1. Again we only keep the K smallest subsets in a list $\mathcal{L}_{A_i}$ if the list contains more than K subsets.

It is not difficult to prove, by induction, that the above dynamic programming process correctly constructs the value assignments and the corresponding collections for each node in the tree $\mathcal{T}$. After this process, every value assignment A to the root node X of $\mathcal{T}$ that assigns no value 0 to any vertex in X is valid, and the list L_k in the collection $\mathcal{L}_A$ contains the K smallest k-dominating sets consistent with the value assignment A. Since every k-dominating set for the graph G is consistent with some value assignment to the root node X, by searching through the collections $\mathcal{L}_A$ over all valid value assignments for X, we will be able to generate the K smallest k-dominating sets for the graph G.

Let us discuss the time complexity of this process. Since the treewidth of the tree decomposition $(\mathcal{V}, \mathcal{T})$ is h, each node X_i in the tree $\mathcal{T}$ contains at most h vertices in the graph G, and there can be at most 3^h valid value assignments to X_i. For each valid value assignment A to X_i, the collection $\mathcal{L}_A$ contains k lists $L_1, \ldots, L_k$, and each list L_r contains at most (A, r)-subsets of Y_i. The most time-consuming step occurs in **Case 3**, in which we need to find mergeable value assignments and merge the lists associated with the mergeable assignments. To merge a list L_r in $\mathcal{L}_{A_j}$ and a list L_s in $\mathcal{L}_{A_j}$, we have up to K^2 possible combinations of the subsets in L_r and L_s. It can be shown that such a merge can be done in time $O(k^3 K \log K)$ (the details of this process are omitted for lack of space). In conclusion, on each node X_i in **Case 3**, the process takes time $O(5^h k^3 K \log K)$. It is not difficult to see that the running time of **Cases 1-2** is also bounded by this upper bound. It follows that the running time of the algorithm is bounded by $2^{O(\sqrt{k})} k^3 n K \log K = 2^{O(\sqrt{k})} n K \log K$, and the proof is complete. □