Listing Subgraphs by Cartesian Decomposition

Alessio Conte, Roberto P Grossi, Andrea Marino, Romeo Rizzi, Luca P Versari

Listing Subgraphs by Cartesian Decomposition

Alessio Conte
National Institute of Informatics, Tokyo, Japan
conte@nii.ac.jp

Roberto Grossi
Dipartimento di Informatica, Università di Pisa, Pisa, Italy
grossi@di.unipi.it

Andrea Marino
Dipartimento di Informatica, Università di Pisa, Pisa, Italy
marino@di.unipi.it

Romeo Rizzi
Dipartimento di Informatica, Università di Verona, Verona, Italy
romeo.rizzi@univr.it

Luca Versari
Dipartimento di Informatica, Università di Pisa, Pisa, Italy
luca.versari@di.unipi.it

Abstract

We investigate a decomposition technique for listing problems in graphs and set systems. It is based on the Cartesian product of some iterators, which list the solutions of simpler problems. Our ideas applies to several problems, and we illustrate one of them in depth, namely, listing all minimum spanning trees of a weighted graph G. Here iterators over the spanning trees for unweighted graphs can be obtained by a suitable modification of the listing algorithm by [Shioura et al., SICOMP 1997], and the decomposition of G is obtained by suitably partitioning its edges according to their weights. By combining these iterators in a Cartesian product scheme that employs Gray coding, we give the first algorithm which lists all minimum spanning trees of G in constant delay, where the delay is the time elapsed between any two consecutive outputs. Our solution requires polynomial preprocessing time and uses polynomial space.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Graph algorithms, listing, minimum spanning trees, constant delay

Digital Object Identifier 10.4230/LIPIcs.MFCS.2018.84

Funding This work has been partially supported by MIUR (Italian Ministry of University and Research) and JST CREST, Grant Number JPMJCR1401, Japan

1 Introduction

Listing problems in set systems have solutions corresponding to sets of elements from a ground set $\mathcal{U}$. Set systems formalize, among others, most subgraph listing problems as subgraphs are usually modeled as sets of vertices or edges (e.g. independent sets where $\mathcal{U}$ is the vertex set, or matchings where $\mathcal{U}$ is the edge set). A listing problem can be thus identified with the set $\mathcal{P} \subseteq 2^{\mathcal{U}}$ of its solutions to be listed, where typically each solution satisfies certain properties (e.g. maximality under inclusion).

In this paper we consider *decomposable* listing problems, where a problem $\mathcal{P}$ is decomposable if it can be modeled as the Cartesian product $\mathcal{P} = S_1 \times \cdots \times S_h$, with $S_1, \ldots, S_h \subseteq 2^{\mathcal{U}}$. We call this a *Cartesian decomposition*.

Many graph listing problems offer natural Cartesian decompositions. A trivial example is given by the (maximal or not) independent sets of a graph G, which can be decomposed using the connected components of G: indeed, any independent set of G corresponds to choosing an independent set from each of its connected components; hence letting S_i be all the independent sets in the i-th connected component in G, all those in G are obtained by combining them as $\mathcal{P} = S_1 \times \cdots \times S_h$, where $X \times Y$ returns all the unions of their vertex sets, $\{x \cup y : x \in X, y \in Y\}$. This same decomposition applies also to other types of subgraphs such as matchings and the edge sets.

This already can give improvements in the general case: even a simple reduction, such as reducing Steiner trees in a graph to combinations of Steiner trees in its biconnected components (see Section 4), already makes listing Steiner trees easier on graphs with small biconnected components. On the other hand, we will see in this paper that there are less trivial Cartesian decompositions, which allow us to reduce some problems to simpler ones, and design more efficient listing algorithms.

For a Cartesian decomposition $\mathcal{P} = S_1 \times \cdots \times S_h$, we observe that each set S_i is the output of some listing algorithm A_i, for $1 \leq i \leq h$. Thus S_i is not explicitly given, and may even have exponential size. As using polynomial space is one of the goals of this paper, among others, we consider A_i as a procedure that outputs all solutions of S_i in some order in polynomial space[1]. There are several definitions of efficiency for listing algorithms [15]: we consider the *delay*, that is, a worst-case measure representing the maximum time elapsed between any two consecutive outputs of A_i. If A_i has bounded delay (e.g. polynomial in the size of G for listing subgraphs of G), it is also called output-sensitive, as the total running time is proportional to the size $|S_i|$.

Given h listing algorithms $A_1, \ldots, A_h$, we model them as iterators over the sets $S_1, \ldots, S_h$, thus avoiding to store explicitly the latter ones. We thus consider the problem of listing implicitly the solutions of the Cartesian product $S_1 \times \cdots \times S_h$, which amounts to listing all the solutions of $\mathcal{P}$. This allow us to design the resulting listing algorithm that handle a complex counter with h digits with constant delay,[2] where the i-th digit goes through $|S_i|$ values and each configuration is a solution of $\mathcal{P}$. While intuitively a good representation, this is not the whole picture of the algorithmic challenge. For example, (re)setting digit i to "0" may be expensive, as we should pay for this solution the preprocessing cost of A_i; even if the latter is just $O(1)$ time, it does not guarantee a good delay as we may need to reset many counters at once, and this gives us a delay of $\Omega(h)$, and possibly worse; if opting for Gray codes, as we do, we should have some control on the order in which solutions of S_i are listed, which requires to modify A_i.

In this paper we propose a technique for iterating over the Cartesian product which allows us, under suitable conditions, to list the solutions of $\mathcal{P}$, with polynomial preprocessing cost and space usage, and with delay bounded by the worst case delay on any A_i. This exploits a form of Gray coding and common properties of state-of-the-art listing algorithms.

After describing how to list solutions using Cartesian decomposition in Section 2, we give some concrete examples of the wide class of problems which can be decomposed in Section 1.1 (or more in detail in Section 4), and study the case of minimum spanning trees

[1] Otherwise, if A_i takes exponential space, we can just use S_i directly, and our listing problem becomes of little interest.

[2] Constant amortized cost is easier to obtain, as shown in [10, Chapter 17].

(MSTs hereafter) in Section 3 to show how to iron all the details in a complete example. We will propose a decomposition which reduces the MSTs of a graph G to the Cartesian product among the spanning trees of a set of unweighted graphs $G_1, \ldots, G_h$. Furthermore, we will show that, by combining this decomposition with a modified version of the state of the art algorithm for listing spanning trees [29], we can obtain the first listing algorithm for MSTs with constant delay, using $O(mn)$ space.

1.1 Related Work

Solving complex problems by decomposing them into simpler problems is the main principle in algorithm design, and many techniques follow that principle: divide-and-conquer, dynamic programming, and so on. For enumeration and listing, some papers addressed this issue: backtracking algorithms [5, 27], and binary partition algorithms [4] in particular, can be seen as an implicit form of decomposition. Other forms of decompositions for listing include [24], which lists all arborescences in a digraph by modelling the solution space as a polynomial, and decomposing this into prime factors, [13] which deletes vertices and edges to generate formulas for counting subgraphs such as spanning trees and acyclic subgraphs, [25] which uses the treewidth decomposition for enumerating purposes, and [31] which shows how to decompose some problems into quasi-independent subproblems using clique separators.

In the case of fixed parameter algorithms, a form of decomposition has been explicited in the so-called *kernelization* which is a technique which preprocesses the input to get a smaller input, called *kernel*. The result of solving the problem on the kernel should either be the same as on the original input, or it should be easy to transform the output on the kernel to the desired output for the original problem [11]. As for listing algorithms for graphs, in many cases the instance of the problem can be partitioned in *several kernels*, so that the listing algorithm can be solved in each of them, and the result of the original problem can be obtained by the result of a Cartesian product among these set of solutions. Some examples are given next, and a more detailed discussion can be found in Section 4.

Application examples. Decomposition, for instance considering connected or biconnected components separately, applies to several listing problems including st-paths [4], unweighted spanning trees, Steiner trees, maximal independent sets [7], maximal induced matching [2], maximal k-degenerate subgraphs [8], minimal feedback vertex/arc sets [28], bipartite subgraphs [33], bounded girth subgraphs [9], dominating sets [19], and acyclic orientations [6] (see Section 4 for more details).[3] One of the goals of this paper is explicitly defining this Cartesian decomposition in a general way, so as to be used as a black box in future works.

MSTs, which we consider as case study for this technique, have a less trivial but powerful decomposition. Spanning trees (in short STs) and MSTs have a rich and long history which has been pointed out in several surveys [3, 26, 20, 14]. Due to this interest, several listing algorithms for STs and MSTs have been proposed over the years. One of the most popular listing algorithms for undirected unweighted graphs is the one by [16] which lists all the STs in optimal time, i.e. $O(\alpha+n+m)$ and space $O(nm)$, where α is the number of solutions. On the other hand, using reverse search, the algorithm in [22] has linear space with $O(m+n+\alpha n)$ time. The one in [29] is optimal time and has also linear memory. The algorithm in [30] lists all the STs in increasing order of weights with cost $O(\alpha m \log m + n^2)$ total time and $O(\alpha m)$ space.

[3] Moreover, it has been implicitly adopted in other graph enumeration papers, like [4].

Concerning more specifically weighted graphs, the algorithm in [23] works also for MSTs, as also one of the algorithms in [16] allows to list MSTs in a similar time bound, i.e. $O(\alpha n)$. Other algorithms have been proposed: the algorithm in [35] has cost per solution equal to $O(m \log n)$ and $O(m)$ space. A generalization of the Kruskal Algorithm for listing purposes has been done in [34]. The reduction in [12] allows us to reduce the enumeration of MSTs to STs in a different graph through edge sliding operations, which, using [16] as a subroutine, lists all MSTs in constant amortized time, but not constant delay. This reduction may be combined with the techniques proposed in this paper to obtain an alternative, equivalent, algorithm for listing MSTs with the same delay. To this end, our approach based on Cartesian decomposition is general and can be applied to a vast range of other graphs problems, as long as their set of solutions can be decomposed in some way.

1.2 Preliminaries

We consider an undirected, edge weighted, graph $G = (V(G), E(G))$. $N_G(v)$ represents the neighborhood of v in G. For simplicity, in the following we call $|V(G)| = n$ and $|E(G)| = m$. Whenever G is clear from the context, we may drop subscripts and use simplified notation like V, E, $N(v)$ instead of $V(G)$,$E(G)$, $N_G(v)$.

Let $W = \{w_1, \ldots, w_k\}$, with $w_1 < \cdots < w_k$, be the distinct edge weights of G, where $1 \leq k \leq m$. For a weight w_i, let $E_{<w_i}(G)$, $E_{w_i}(G)$, and $E_{>w_i}(G)$ be the sets of all edges in $E(G)$ which have weight respectively *smaller than* w_i, *equal to* w_i, and *larger than* w_i.

For a given edge $e = \{x, y\}$, we call *contracting* e in G the operation of deleting x and y from G, as well as all edges incident to them, and replacing them with a new node z such that $N(z) = N(x) \dot\cup N(y)$, observing that $N(z)$ can be a multiset (and thus we obtain a multigraph). We refer to the (multi)graph obtained by contracting e in G as G/e. We also implicitly maintain a correspondence between the new edges in G/e and those in G to make the operation reversible. Note that this is not a one-to-one correspondence as two edges $\{v, x\}$ and $\{v, y\}$ will correspond to the same edge $\{v, z\}$ in G/e. Instead, $G \setminus e$ represents the graph obtained by simply deleting e from G (but not its extremes). The operations G/X and $G \setminus X$ are similarly defined for a set of edges $X \subseteq E(G)$, where edges (or their corresponding ones) are respectively contracted or deleted one by one in any order.

Given a set of edges $E' \subseteq E(G)$, $V[E']$ is the set of nodes incident to at least one edge in E'. The subgraph of G *induced* by E' is the graph $G[E'] = (V[E'], E')$. A spanning tree of a connected graph G is a subgraph $T = G[E_T]$, for some E_T such that T is connected, acyclic, and contains all nodes of G, i.e., $V[E_T] = V(G)$. One can also define a spanning tree as a *maximal* set of edges E_T such that the corresponding subgraph $G[E_T]$ is acyclic. Yet another equivalent definition of spanning tree is a *minimal* set of of edges E_T which guarantees connectivity between all pairs of nodes, since any edge that partakes in a cycle may be deleted without affecting the connectivity of the graph [10]. We denote by $\mathcal{T}(G)$ the set of all spanning trees t_i of G.

When G is *not connected*, let $\mathcal{C}(G) = \{C_1, \ldots, C_j\}$ be the set of connected components of G; for simplicity, we define a spanning tree of the non connected graph G as the union of a spanning tree of each connected component. In this case, we can easily see how $\mathcal{T}(G)$ corresponds to the *Cartesian product* among the sets of trees of its connected components, i.e., $\mathcal{T}(C_1) \times \ldots \times \mathcal{T}(C_j)$.

2 Finding Solutions of the Cartesian Product via Gray Coding

In the previous section we have seen that the problem of listing patterns in several cases reduces to combining solutions from other listing subproblems in all the possible ways. This corresponds to our notion of decomposable problem and our goal here is to achieve efficient enumeration of the original problem knowing how to efficiently solve each of the subproblems, as summarized next.

Main Problem. Given h subproblems, whose solutions are sets $S_1, \ldots, S_h$, the solutions of our problem can be seen conceptually as tuples $\langle s_1, \ldots, s_h \rangle \in \mathcal{P} = S_1, \times \ldots \times S_h$. Now suppose that we have a listing algorithm which is able to iterate over the solutions of S_i for each i with polynomial setup time $O(P)$, delay time $O(D)$ and polynomial space, modeled as an iterator A_i which can yield the solutions of S_i one by one after an initial setup.[4] We say that the iterator A_i is *reversible* if there exists an iterator A_i^{-1} which scans the solutions of A_i in the opposite order with the same time and space costs. We will show that, having reversible iterators A_i, it is possible to build an iterator for $\mathcal{P}$, whose delay is the same of A_i.

▶ **Theorem 1.** *Given a reversible iterator A_i for each set of solutions S_i, $1 \leq i \leq h$, running with delay $O(D)$, setup time $O(P)$, and polynomial space, it is possible to list all the solutions in $\mathcal{P} = S_1, \times \cdots \times S_h$ with delay $O(D)$, setup $O(h \cdot P)$ and polynomial space.*

While the polynomial space of an iterator can increase by a factor h in the statement of Theorem 1, we observe that the delay does not depend on h nor includes the setup times. This is non-trivial as we could trivially start running the algorithm A_1 to get S_1, and each time a solution $s_j \in S_1$ is found, recursively start the iterator A_2 to scan all the solutions in S_2. This is not satisfactory as it requires, in the worst case, a delay which can be $O(h \cdot (P + D))$, so that for each solution $s_j \in S_i$ $(1 \leq i < h)$ we have to setup the iterator S_{i+1}. To meet the result in Theorem 1, we need to erase the setup cost from the delay, meaning that for each $s_j \in S_i$, the iterator for S_{i+1} is ready to use without calling a setup function, e.g. `init()` for A_{i+1}. On the other hand, we need to erase also the dependency from h. To overcome this we start from the well known *Gray coding* technique, in a generalized form given by Knuth, in "The Art of Computer Programming" Volume 4 Fascicle 2A [17]. This requires to design forward and backward iterators, i.e. meaning that for each iterator A_i that scans all the solutions of S_i (with $1 \leq i \leq h$) in a certain order we need to design a corresponding iterator, which we call A_i^{-1}, that scans the same solutions in the reversed order.

2.1 Setup costs

Given an iterator A_i, with $O(D)$ delay and a given setup cost of $O(P)$ time (corresponding to the cost of `init()`), the main objective of this section is showing that it is possible to scan the set of solutions S_i an arbitrary number of times, paying the setup cost $O(P)$ only once (the first time A_i is started), and without affecting the delay $O(D)$.

To this aim, first consider the trivial case in which the total execution time of A_i, after the setup is done, is equal or less than the setup cost $O(P)$. This implies that the output is bounded in size by $O(P)$. In this case, we can simply run A_i, and store the complete whole output: this takes $O(P)$ time and space, and clearly allows us to iterate on the solutions of A_i (i.e., the output) any number of times with an equal (or better) delay.[5]

[4] This can be realized for instance providing the well-known `init()` and `getNext()` methods of Java iterators, to respectively initialize and give the next solution for a given iterator.

[5] To recognize this case, we simply run A_i for $O(P)$ steps after the setup, without affecting its cost.

Otherwise, let M be the total amount of data generated by the setup of A_i. This can be anything, such as initialized data structures, pre-processed information on the input, buffered solutions, or other data. Once M is given, then A_i can start without further ado, so our goal is achieved by guaranteeing that an unaltered copy of M is ready at any time to start again.

To obtain this, we keep two copies of M, namely M_a and M_b. Initially these need $O(P)$ time to be computed, and clearly we can restore any of them at any time in $O(P)$ time by executing the setup again. Our strategy works as follows. The first time that A_i is executed it will use the data in M_a. After its execution, M_a may have been altered and may not be usable, but M_b is intact. For the second execution, we will run A_i using M_b, and restore M_a *while* running the algorithm: we perform alternatively one step of A_i and one step of restoring M_a. The next time A_i is run, it can use the data in M_a, and restore M_b while running in the same way. Hence, even executions of the iterator A_i use M_a and restore M_b, while odd ones use M_b while restoring M_a. Note that, since the execution of A_i takes $\Omega(P)$ time (we covered the other case above), M_a (or M_b) will have been fully restored before A_i terminates. As a result, we can start the iterator A_i any number of times, but the setup cost is only paid once in the beginning. Furthermore, this slows down A_i by just a factor of two, so the asymptotic complexity is unchanged.

2.2 Gray Coding via Forward and Backward Iterators

The idea of Gray coding is producing tuples one after the other such that any two consecutive tuples differ by just one entry, i.e. after the output of $\langle s_1, \ldots, s_{j-1}, s_j, s_{j+1} \ldots, s_h \rangle$ we output $\langle s_1, \ldots, s_{j-1}, s'_j, s_{j+1}, \ldots, s_h \rangle$ for some j. A result in [17] was given for explicit sets, but in the following we show how to generalize it for implicit sets, i.e. sets which are results of an iterator. Clearly, to do so, we take into account the delay between the output of s_j and s'_j in A_j, namely $O(D)$ in Theorem 1. The adaptation is not trivial as [17] assumes a certain ability of moving inside the objects in S_j, while a listing algorithm produces solutions in one fixed order.

Among the several variations of Gray coding, we consider *loopless reflected mixed-radix Gray generation* (Algorithm H in [17]), which we refer to as LRMG. This algorithm visits all the tuples changing only one coordinate ± 1 at each step. It maintains an array of focus pointers which says which iterator we have to call at each step, and an array of directions, which for each j (with $1 \leq j \leq h$) says whether we are iterating on S_j from the first solution to the last one or vice versa. Without loss of generality, we can assume $|S_i| > 1$ for each i, as if $|S_i| = 1$ then the only element in S_i is present in all solutions and needs to be output just once at the beginning.

The approach of LRMG can be generalized to deal with the tuples of $\mathcal{P} = S_1 \times \ldots \times S_h$, when the elements of each S_i are implicit, i.e. the result of an iterator. This generalization can be done if the following holds: for each i, on top of the iterator A_i that scans the solutions of S_i in a certain order π, with delay $O(D)$, we can obtain another iterator, called A_i^{-1}, that scans the same solutions in the *opposite* order of π, still with delay $O(D)$. The logic behind this is that the "direction" variable of each iterator is changed only at the end of each iteration, thus we only need to perform complete iteration on S_i in one order and its opposite.

An important remark is that whenever switching from using a certain A_i to the corresponding A_i^{-1}, we should not consider as output the first solution found by A_i^{-1}, since it is the same as the last one output by A_i. The same applies when switching from A_i^{-1} back to A_i. In both cases the delay and preprocessing cost remain the same.

If this is given, we can prove Theorem 1. Indeed, we can plug the suitable iterators A_i and A_i^{-1} into LRMG, and uses the setup cost factorization described in Section 2.1, obtaining an algorithm for iterating the solutions of $\mathcal{P}$ having as delay and setup time cost respectively the sum of the delay and setup time costs of A_i and A_i^{-1}, which asymptotically is the same.

Getting the backward iterator. We now show how to obtain A_i^{-1} given A_i, under suitable conditions which are met by most state-of-the-art listing algorithms. In particular, we assume A_i to be a recursive algorithm with a recursion tree in which each node outputs at most one solution, and generates children as nested recursive calls. Moreover, we assume to be able to generate children of a given recursive call in the opposite order as in A_i, which is generally true for binary partition [5, 4] or reverse search [1, 21] algorithms, where the i-th child call is not influenced by the computation of the subtrees of the previous $i-1$ children.

Let T be the tree induced by the recursive calls. Note that in reverse search algorithms each node in T outputs a solution, while in other backtracking algorithms output may be done in just some nodes, or some of the leaves of T. Consider the following two traversals of T, called FORWARD (resp. BACKWARD), starting from a node u, where d is the depth of the current recursion node in T.

1. If d is even (resp. odd) output u.
2. For each child i of u in increasing (resp. descending) order call FORWARD (u) (resp. BACKWARD (u)).
3. If d is odd (resp. even) output u.

The following observation clearly holds.

▶ **Observation 2.** *For any tree T, let π be the visiting order of the nodes of T obtained by* FORWARD. *Then* BACKWARD *scans the nodes of T in the opposite order with respect to π.*

The above traversal is similar to the so-called *alternative output* in [32], which is usually done to reduce the delay of listing algorithms, but we use this variation to get forward and backward iterators A_i and A_i^{-1} at the same time. In particular, we define A_i and A_i^{-1} as the iterators respectively induced by FORWARD and BACKWARD visit algorithms on T.

Applying Observation 2, we observe that in order to design A_i and A_i^{-1}, we only need a method to scan all the children in each internal node of T in one direction and in the opposite one with the same cost. As a result, both A_i have the same delay $O(D)$.

3 Case Study: Minimum Spanning Trees

In this section we describe how to use our technique to list all the MSTs of a graph, improving the state of the art to get constant delay for STs and applying the technique in Section 2 to get constant delay also for MSTs.

Overview. We will firstly reduce the problem of listing MSTs to the one of listing STs. To this aim, we order the weights of the graph as in the well-known Kruskal Algorithm, $w_1 < \cdots < w_k$, and we solve a STs listing problem for each different w_i, that is finding all the STs in a series of h graphs $C_1, \ldots, C_h$ (with $h \geq k$). For the latter task, on each C_i we run our improvement of the algorithm by [29], which achieves $O(1)$ delay to list STs and is presented in Section 3.2.1. We then need to compose the solutions, so that each combination of solutions for $C_1, \ldots, C_h$ corresponds to a different MSTs. As we have delay $O(1)$ on each C_i (with setup time $O(m^2)$ and space $O(mn)$), our main goal is to preserve these bounds when combining the solutions of $C_1, \ldots, C_h$. Indeed, the usual algorithm would setup an

iterator for C_1, and for each solution of C_1 would setup the iterator for the solutions of C_2 and so on. In this way, the delay can be up to $O(hm^2)$. Our goal is to reduce this delay to $O(1)$, which is the delay of a single iterator, maintaining polynomial space.

We hence apply the techniques in Section 2, adapting listing algorithms to use the Gray coding strategy, basically showing how to build backward iterators from forward iterators, how to factorize setup time costs and use and rebuild auxiliary data structures without afflicting the time complexity.

This problem is related to the more general problem of listing all the tuples of a Cartesian product which is usually solved using Gray coding. However, as in the scenario presented in Section 2, in our case the elements of the tuples are not explicitly given but they are generated through iterators which give solutions in a linear way (it is not possible to jump from one solution to an arbitrary one) and require setup time costs. In this section, we adapt our general solution to deal with the specific case of MSTs. In particular:

- we show how to reduce the problem of listing MSTs to the listing the solutions of a Cartesian product among h sets of solutions, each one corresponding to a set of STs in a graph.
- we show how to list STs with constant delay, as the state of the art algorithm [29] for STs runs in constant amortized time per solution but *linear* delay.
- we show how to use output queue technique to get rid of setup times and pay them just once at the beginning.
- we design *ad hoc* forward and backward iterators for STs, both with constant delay, which help us to successfully applying Gray coding to MSTs, so that the difference between two consecutive listed solutions is constant and can be computed in constant time.

3.1 From Minimum Spanning Trees to Spanning Trees

A well known way to compute a minimum spanning tree is using the greedy algorithm by Kruskal [18], which consists of the following steps:

1. Scan the edges of G according to an increasing weight order.
2. Add an edge to the solution T if it does not create a cycle in T, and discard it otherwise.
3. When all edges have been considered, T is a minimum spanning tree of G.

It is immediately evident how if all edges have different weights they will always be scanned in the same order, thus the algorithm will return the same result. However, if several edges have the same weight, the algorithm may scan them in different orders and produce different trees. Actually, it turns out that *any* minimum spanning tree of G may be found by Kruskal's algorithm, if that the adequate order of the edges is provided. More formally

▶ **Lemma 3.** *Let T be any* MST *of G, and $e_1, \ldots, e_m$ an ordering of $E(G)$ in increasing weight, such that any edge $e_i \in T$ appears in the order* before *any edge not in T which has the* same *weight as e_i. Running Kruskal's algorithm according to this order yields exactly T.*

By shifting our point of view, we can rephrase the algorithm in a more convenient way for our goal, considering the weights $w_1 < \cdots < w_k$ of G. The algorithm is essentially saying that we should greedily add as many edges of E_{w_1} as possible, without creating a cycle, before considering those of E_{w_2}. In other words, we are selecting a *maximal acyclic subgraph* of the graph $G[E_{w_1}]$, that is, a spanning tree T of $G[E_{w_1}]$. Once such a tree (or combination of trees, if $G[E_{w_1}]$ is not connected) has been found, all edges joining two vertices in the same connected component of T may be discarded, as they will create a cycle and not improve

the connectivity. Furthermore, adding to T any edge from a node x to any node in some connected component C of T is equivalent in terms of connectivity.

This means that we can *contract* each C into a single node to obtain a new graph G', and continue to execute Kruskal's algorithm, i.e., considering edges in E_{w_2}, without affecting the outcome. Greedily adding edges of E_{w_2} to T without creating cycles corresponds to selecting a maximal acyclic subgraph (i.e., spanning tree) of $G'[E_{w_2}]$. Again, once the edges of this spanning tree are selected and added to T, we can contract the connected components and connect as many nodes as possible using the edges of E_{w_3}.

A crucial point is that any spanning tree of $G[E_{w_1}]$ will contain exactly the same connected components as any other, corresponding to the connected components of $G[E_{w_1}]$. In other words, $G'[E_{w_2}]$ is always the same, regardless of which spanning tree of $G[E_{w_1}]$ was previously selected. This means that we can abstract the analysis of each weight w_i from all others: indeed, when considering w_i, we can consider as a single node all the nodes connected by edges of weights smaller than w_i, and we must create as many new connections as possible using edges of E_{w_i} without creating cycles. This corresponds to selecting a spanning tree in the graph obtained by contracting all edges of weight smaller than w_i, i.e., $E_{<w_i}$, and then deleting all those of weight greater than w_i; we call this graph G_i, whose formal definition is $G_i = (G/E_{<w_i}) \setminus E_{>w_i}$. We can thus give a recursive definition to the set of all minimum spanning trees of G, i.e. MSTs(G), as:

$$\text{MSTs}(G) = \prod_{w_i \in W} (\mathcal{T}(G_i)) \tag{1}$$

Where we recall that $\mathcal{T}(G_i)$ is the set of all spanning trees of G_i, $\prod$ corresponds to the Cartesian product between the given sets and, if G_i is not connected, we defined its spanning trees as combinations of a spanning tree of each connected components, as follows:

$$\mathcal{T}(G_i) = \prod_{C_j \in \mathcal{C}(G_i)} \mathcal{T}(C_j). \tag{2}$$

From what we said above, as all (and only) the minimum spanning trees of G are obtained by combining a minimum spanning tree of each G_i, for all $1 \leq i \leq k$, we get the following.

▶ **Lemma 4.** *Equation 1 gives all (and only) the minimum spanning trees of G.*

Furthermore, note that all edges in a single G_i have the same weight, so all spanning trees have the same weight as well, and listing minimum spanning trees correspond to just listing all spanning trees. As finding solutions for each G_i is relatively quick, modifying the constant amortized time provided by [29] in constant delay, combining these solutions maintaining the constant delay the same space bound is not trivial. In the following section, we will see that this corresponds to a more general problem, which is finding tuples of a Cartesian product among k sets, with time per solution independent from k and neglecting setup costs, where the elements of these sets are given implicitly, as result of an iterator.

3.2 MWST Gray Coding

In Section 3.1 we showed that we can solve the MST enumeration problem by listing (non weighted) STs in a series of multigraphs. In particular, we have seen that the problem of listing MSTs can be reduced to that of listing STs in several graphs and combining them in all the possible ways. Indeed, each MST in G corresponds to a tuple $\langle t_1, \ldots, t_k \rangle \in \mathcal{T}(G[w_1]) \times \ldots \times \mathcal{T}(G[w_k])$, where $k = |W|$ and $t_i \in \mathcal{T}(G_i)$ is one of the possible combination

of STs of the connected components of G_i according to Equation 2. Let $C_1, \ldots, C_h$ be a sequence containing the connected components of $G[w_1]$, followed by the ones of $G[w_2]$, and so on until those of $G[w_k]$. By combining Equations 1 and 2, we get that a MST is a tuple $\langle s_1, \ldots, s_h \rangle$ where s_i is any spanning tree of C_i, i.e. $s_i \in \mathcal{T}(C_i)$, for $1 \leq i \leq h$. Our goal is to list all the elements in $\mathcal{T}(C_1) \times \ldots \times \mathcal{T}(C_h)$ in an efficient way, given that we know how to list all the solutions in $\mathcal{T}(C_i)$ for each i. In this section, we show how to adapt the state of the art algorithm for listing STs to run in constant delay, and how to combine this resulting algorithm with the techniques in Section 2 to obtain a constant delay algorithm for listing MSTs.

3.2.1 Enumerating Spanning Trees in Constant Time Delay

In the following, we show how to improve the state of the art algorithm for listing STs to get constant delay per solution, proving the following result.

▶ **Lemma 5.** *There exists an algorithm* ST-CDEL *that lists spanning trees with constant delay, $O(m^2 n)$ preprocessing time, and $O(mn)$ space.*

The state of the art algorithm by Shoioura et al. [29], denoted as ST-CAT in the following, lists STs in constant amortized time per solution (CAT) in multigraphs,[6] but its delay is not constant. We will explain in the following how to modify ST-CAT properly to prove Lemma 5.

As each spanning tree has size $O(n)$, it is clearly impossible to output each solution entirely, so ST-CAT outputs just the first solution, and then the *differences* between one solution and the previous one. Combining all these differences, starting from the first solution output by the algorithm until the last differences output, will yield the "current solution" that the algorithm is considering.[7] In particular, using the notation in [29], this is done in the form of *output("$-e_k$, $+g$, tree,")*, meaning that the edge e_k must be deleted and edge g must be added with respect to the previously output solution, and *"tree"* means that a new solution has been reached, i.e., performing all the addition/deletion instructions output so far yields a new solution. However, ST-CAT has two features which cause its delay to be more than constant, namely *output size* and *update costs*. Both of these can be overcome by with proper use of deamortization techniques. We address them separately in the following.

Output size. Each time a recursive call is closed by ST-CAT, the changes done to the output are restored. Namely, each recursive call of the algorithm prints a first *output("$-e_k$, $+g$, tree,")*, then generates some children recursive calls, then prints a second *output("$-g$, $+e_k$,")* to undo these changes. When backtracking, several recursive calls may be terminated at once: in the worst case, up to the depth of the recursion tree, which is $O(n)$ for ST-CAT.

This causes the cumulative changes to be up to $\Omega(n)$, before the next output is performed by printing a "tree" token. Even if the computation time in reaching this output could be ignored, printing this number of changes clearly cannot be done in constant time.

An important property of this structure is that combining the output streams of all algorithms A_i will yield an output stream which iterates over the solutions of G: indeed, a solution of the main problem is a combination of solutions of the subproblems, and each output

[6] For reference, its pseudo code can be found in [29] at page 690.

[7] Reconstructing and outputting the whole solution from the stream would take more than constant time, but since all solutions have size $\Theta(n)$, a constant-delay algorithm for the problem would be impossible with this requirement.

of each A_i changes the solution of exactly one subproblem, we have that the combination of sub-solutions (i.e., the solution of G) will be a new one.

Furthermore, *every* recursive node X of A_i first alters the current solution with the *first* output, and then undoes these changes with the *second* output. Since the same is valid for children nodes of X, it follows that the "current solution" which can be read on the output stream *just before* the second output is equal to the one which can be read *just after* the first one.

In other words, we can chose to make the node X output a solution at the *end* of its execution (i.e., after its children nodes have terminated), rather than at the beginning, by simply printing the "tree" token at the beginning of the *second* output instruction instead of the end of the first. More formally, recursive calls which are tweaked to output the solution at the end will perform the first output call in the form *output("$-e_k$, $+g$,")*, and the second one in the form *output("tree, $-g$, $+e_k$,")*, instead of how described above.

With this property, we can apply the *alternative output* technique from [32]: for each recursive node X in the recursion tree T, we output its solution at the *beginning* of X if its depth is even, and at the *end* if it is odd.

This means that each output is performed after just a constant number of recursive calls has been either generated or closed by the algorithm. Since the changes performed by a single call are constant in size, it follows that the difference between one solution and the previously output one is always of constant size.

Finally, note that the differences between one solution and the next one are the same in the reversed order, thus this applies to the reversed iterator A^{-1} as well, which can be obtained as described in Section 2.2.

Update costs. Each recursive node of ST-CAT has to update some data structure. While the update cost is cleverly amortized to be just $O(1)$ per node, one update can be $O(m)$ in the worst case. We solve this issue using the technique from [32], called *output queue*. The general idea is to accumulate the output in memory, so that it is suitably delayed in a way that the time between the output of two "tree" tokens is constant.

Given the recursion tree T of ST-CAT, let T^* be an upper bound on the cumulative cost of nodes in a root-to-leaf path of T. As reported in [29], the depth of T is $O(n)$ and the maximum cost of a single recursive call is $O(m)$. Thus $T^* = O(mn)$. Furthermore, let $\bar{T}$ be the maximum, among all subtrees T' of T, of the total cost of T' divided by the number of nodes of T' corresponding to a solution. It can be seen that the amortization scheme in [29] that allows ST-CAT to have constant amortized time can be equivalently applied to any subtree, as it only considers the descendants of a recursive node. Thus we have that this ratio is constant for any subtree, and that $\bar{T} = O(1)$.

The output queue Q technique requires us to fill a queue containing $2 \cdot T^*/\bar{T} + 1 = O(mn)$ solutions, i.e. all the output required before printing $O(mn)$ "tree" tokens. By Theorem 2 in [32], the time required to initially fill the queue is bounded by $O(T^* + \bar{T}) = O(mn)$. As for the space, the first solution is output completely, but for all the others we just output the difference with respect to the previous solution, which as constant size, thus the size of the queue is always bounded by $O(mn)$.

As soon as the queue is full, dequeue the top solution (i.e. the set of changes before a "tree" token appear in the queue) and output it. Repeat this step every $O(\bar{T}) = O(1)$ time while the algorithm is executed. Finally, during the execution of the algorithm, each time an output is required, the request is enqueued in Q instead of being output. The output queue guarantees that Q is never empty until the end of the execution, and thus that the algorithm will have in this case $O(1)$ delay after the preprocessing time.

By combining what said so far in this section, we obtain an iterator A, which after a one time preprocessing cost of $O(mn)$, can iterate over all the spanning trees of a given graph with n vertices and m edges, not just once but any number of times without paying for the preprocessing cost again. Due to the memory requirements of the output queue, the space is $O(mn)$.

3.2.2 Forward and Backward Iterators for Spanning Trees

Given the iterator A, we need to show that is possible to obtain the backward iterator A^{-1}, which is the iterator processing the solutions in opposite order with respect to A, still ensuring constant delay. This follows by combining the techniques used above and in Section 2.2. Indeed, consider the design of A for spanning trees given above: if we do not consider the use of the *output queue* technique, this is a recursive algorithm which performs alternative-output and outputs at most one solution per recursion node. Indeed, we can invert the order in which the solutions are output precisely as described in Section 2.2, obtaining an A^{-1} algorithm. Furthermore, it's easy to see that the algorithm so obtained still maintains the same T^* and $\bar{T}$, as well as the same recursive structure, thus we can apply the output queue technique for A^{-1} as well, obtaining the following lemma.

▶ **Lemma 6.** *There exist forward and backward iterators for listing spanning trees with constant delay, $O(mn)$ preprocessing time, and $O(mn)$ space.*

3.2.3 Constant delay algorithm for MSTs

Finally, we present how to use the techniques presented in Sections 3.1, 2 and 3.2 to get a constant delay algorithm for the MSTs listing problem.

Preprocessing. Computing each G_i can clearly be done in $O(m)$ time by performing a BFS to identify the components connected by edges of $E_{<w_i}$, and then adding those of E_{w_i}. Note that, as every edge appears in at most one G_i, the total space for storing these graphs is $O(m)$ and the total computation time is $O(m^2)$.

Furthermore, as a spanning tree has $n-1$ edges, and all minimum spanning trees have the same number of edges of a given weight, there are at most $n-1$ distinct edge weights which can appear in any MST, thus at most n graphs G_i will be non-empty.

For each of these graphs, we must run the preprocessing phase of the ST-CDEL algorithm shown in Section 3.2.1, which takes $O(mn)$ time and uses $O(mn)$ space. Note that this is done twice for each G_i, as we need to start both the forward iterator A_i for the STs of G_i and the corresponding backward iterator A_i^{-1}. The preprocessing phase will thus take $O(m^2 + \sum_{i=1,\dots,h} |V(G_i)| \cdot |E(G_i)|)$ time. The space required to keep track of all the information is $O(\sum_{i=1,\dots,h} |V(G_i)| \cdot |E(G_i)|)$. While these are both trivially bounded by $O(mn^2)$, we can refine this bound: indeed, any edge of G appears in at most one G_i, we have $\sum_{i=1,\dots,h} |E(G_i)| = O(|E(G)|) = O(m)$, and while this is not true for the vertices, we have $|V(G_i)| \leq |V(G)| = n$, meaning that $\sum_{i=1,\dots,h} |V(G_i)| \cdot |E(G_i)| = n \cdot \sum_{i=1,\dots,h} |E(G_i)| = O(mn)$. We thus have that the preprocessing time is $O(m^2 + mn) = O(m^2)$ and the space usage is $O(mn)$.

Algorithm. Once all A_i and A_i^{-1} algorithms are ready, we can use them as iterators on their respective space of solutions, and run the Gray coding algorithm described in Section 2. The Gray coding algorithm alternatively uses A_i and A_i^{-1} depending on the direction set,

which for each i says whether we are iterating from the first solution to the last one or vice versa. The resulting algorithm has constant delay, as we have seen that both when we are using A_i or A_i^{-1} we get a new output with constant delay. Indeed, recall that both A_i and A_i^{-1} dequeue from their corresponding output queue until a "tree" token is found, and the number of dequeues is $O(1)$.

An important remark for A_i, the differences with respect to the previous ST output by A_i turns out to be differences with respect to the previous MSTs of G.

Indeed each output of A_i consists in a difference which turns one ST t'_i of G_i into the next one t''_i (which differ for a constant number of changes). The corresponding solutions for MSTs are tuples $\langle t_1, \ldots, t'_i, \ldots, t_h \rangle$ and $\langle t_1, \ldots, t''_i, \ldots t_h \rangle$ which still differ for a constant number of changes, as t_j corresponds to the same spanning tree for each G_j with $j \neq i$. Similarly, the same applies in the case of A_i^{-1}.

Finally, note that whenever switching from using A_i to A_i^{-1}, we must not output the first solution found by A_i^{-1}, as it corresponds to the last one output by A_i; the same applies when switching from A_i^{-1} to A_i. Clearly, this does not affect the complexity of the algorithm.

As a result, we have proved the following.

▶ **Theorem 7.** *There exists an algorithm which lists all the* MSTs *of a weighted graph with constant delay, $O(m^2)$ preprocessing time, and $O(mn)$ space.*

4 Applications

In this section, we consider several listing problems on graphs which turn out to be suitable for Cartesian decomposition. Some of them can be decomposed by looking at the biconnected components $B_1, \ldots B_h$ of the input graph. Applying Theorem 1, assuming to use forward and backward iterators for each B_i, we get that the delay of iterating over all the solutions of the graph is the maximum delay among the iterators for generating $S_1, \ldots, S_h$, where S_i is the set of solutions associated to B_i.

- ***st*-Paths.** Listing all the st-paths given two nodes s and t in a graph can be done looking at the bead string of biconnected components $B_1, \ldots, B_h$ for some h, with $s \in B_1$ and $t \in B_h$. In particular, let $b_1, \ldots, b_{h-1}$ be the corresponding sequence of articulation points, i.e. b_i is the articulation point between B_i and B_{i+1} $(1 \leq i < h)$. Then all the st-paths are all the tuples in $S_1 \times \ldots \times S_h$ where S_1 is the set of sb_1-paths, S_i is the set of $b_{i-1}b_i$-paths (with $2 \leq i \leq h-1$), S_h is the set of $b_{h-1}t$-paths.
- **Unweighted Spanning Trees.** Given a connected graph G and its biconnected components $B_1, \ldots, B_h$, it is easy to show that all the STs of G correspond to all the tuples in $S_1 \times \ldots \times S_h$, where S_i is the set of STs of B_i.
- **Steiner Trees.** Given a connected graph, all the Steiner trees with set of terminals W can be obtained looking at the biconnected components of G. For each B_i, S_i corresponds to the set of Steiner trees of the graph induced by B_i, fixing as set of terminals the nodes in $W \cap B_i$ and the articulation points x of G such that $x \in B_i$ and x can reach in G some node in W without passing through nodes in B_i.

Other problems on graphs turn out to be decomposable looking at the connected components. Given a graph G, let $Z_1, \ldots, Z_h$ be its connected components. Then the following patterns in G correspond to all the tuples in $S_1 \times \ldots \times S_h$, where S_i is the set of the same kind of patterns just for the graph Z_i. For instance, the following patterns, whose formal definition is given in the corresponding reference, belong to this category.

Maximal Independent Sets (see [7])
Maximal Induced Matching (see [2])
Maximal k-Degenerate Subgraph (see [8])
Minimal Feedback Vertex/Arc Sets (see [28])
Bipartite Subgraphs (see [33])
Bounded Girth Subgraphs (see [9])
Dominating Sets (see [19])
Acyclic Orientations (see [6])

Once again, designing forward and backward iterators as discussed in the previous section for the above problems, and running each of them for each Z_i, we get that the delay of the iterator over the solutions of G has delay equal to the maximum among the delays of the iterators over the solutions of $Z_1, \ldots, Z_h$.

Some other problems are decomposable in a more complex way which clearly is linked to their nature. This is the case of minimum spanning trees, which is object of the case study in Section 3.

5 Conclusions

In this paper we have studied a natural decomposition technique which consists in reducing an enumeration problem to the Cartesian product of the result of several easier sub-problems. We proposed an efficient way to implement this decomposition using a form of Gray coding and forward/backward iterators on the solutions of the sub-problems. In some cases, the sub-problems correspond to smaller instances of the input problem, while in others the problem itself may be a simpler one (e.g., for Minimum Weight Spanning Trees the sub-problem is listing spanning trees of an unweighted graph). We showed an in-depth analysis for the listing of Minimum Weight Spanning Trees, reducing with this technique the known bounds from Constant Amortized Time to Constant Delay. We also gave examples of several other problem which can benefit from this decomposition.

References

1 David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

2 Manu Basavaraju, Pinar Heggernes, Pim van 't Hof, Reza Saei, and Yngve Villanger. Maximal induced matchings in triangle-free graphs. *Journal of Graph Theory*, 83(3):231–250, 2016. doi:10.1002/jgt.21994.

3 Cüneyt F Bazlamaçcı and Khalil S Hindi. Minimum-weight spanning tree algorithms a survey and empirical study. *Computers & Operations Research*, 28(8):767–785, 2001.

4 Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1884–1896. SIAM, 2013.

5 Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Communications of the ACM*, 16(9):575–576, 1973.

6 Alessio Conte, Roberto Grossi, Andrea Marino, and Romeo Rizzi. Listing acyclic orientations of graphs with single and multiple sources. In *LATIN 2016: Theoretical Informatics - 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*, pages 319–333, 2016. doi:10.1007/978-3-662-49529-2_24.

7 Alessio Conte, Roberto Grossi, Andrea Marino, Takeaki Uno, and Luca Versari. Listing maximal independent sets with minimal space and bounded delay. In *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, pages 144–160, 2017. doi:10.1007/978-3-319-67428-5_13.

8 Alessio Conte, Mamadou Moustapha Kanté, Yota Otachi, Takeaki Uno, and Kunihiro Wasa. Efficient enumeration of maximal k-degenerate subgraphs in a chordal graph. In Yixin Cao and Jianer Chen, editors, *Computing and Combinatorics*, pages 150–161, Cham, 2017. Springer International Publishing.

9 Alessio Conte, Kazuhiro Kurita, Kunihiro Wasa, and Takeaki Uno. Listing acyclic subgraphs and subgraphs of bounded girth in directed graphs. In *Combinatorial Optimization and Applications - 11th International Conference, COCOA 2017, Shanghai, China, December 16-18, 2017, Proceedings, Part II*, pages 169–181, 2017. doi:10.1007/978-3-319-71147-8_12.

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, third edition, 2009.

11 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.

12 David Eppstein. *Representing all minimum spanning trees with applications to counting and generation*. Information and Computer Science, University of California, Irvine, 1995.

13 Ira M. Gessel. Enumerative applications of a decomposition for graphs and digraphs. *Discrete Mathematics*, 139(1):257–271, 1995. doi:10.1016/0012-365X(94)00135-6.

14 Ronald L Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

15 David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

16 Sanjiv Kapoor and Hariharan Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM Journal on Computing*, 24(2):247–265, 1995.

17 Donald E Knuth. The art of computer programming, volume 4, fascicle 2: Generating all tuples and permutations, 2005.

18 Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. URL: http://www.jstor.org/stable/2033241.

19 Kazuhiro Kurita, Kunihiro Wasa, Hiroki Arimura, and Takeaki Uno. Efficient enumeration of dominating sets for sparse graphs. *CoRR*, abs/1802.07863, 2018. arXiv:1802.07863.

20 F. Maffioli. Complexity of optimum undirected tree problems: a survey of recent results. In *Analysis and design of algorithms in combinatorial optimization*, pages 107–128. Springer, 1981.

21 Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Algorithm Theory - SWAT 2004*, pages 260–272, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

22 Tomomi Matsui. An algorithm for finding all the spanning trees in undirected graphs. *METR93-08, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo*, 16:237–252, 1993.

23 Tomomi Matsui. A flexible algorithm for generating all the spanning trees in undirected graphs. *Algorithmica*, 18(4):530–543, 1997.

24 Matúš Mihalák, Przemysław Uznański, and Pencho Yordanov. Prime factorization of the kirchhoff polynomial: Compact enumeration of arborescences. In *2016 Proceedings of the Thirteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 93–105. SIAM, 2016.

25 Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Counting and enumeration problems with bounded treewidth. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 387–404. Springer, 2010.

26 A. R. Pierce. Bibliography on algorithms for shortest path, shortest spanning tree, and related circuit routing problems (1956–1974). *Networks*, 5(2):129–149, 1975.

27 Ronald C Read and Robert E Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

28 Benno Schwikowski and Ewald Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(1-3):253–265, 2002. doi:10.1016/S0166-218X(00)00339-5.

29 Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM Journal on Computing*, 26(3):678–692, 1997.

30 Kenneth Sörensen and Gerrit K Janssens. An algorithm to generate all spanning trees of a graph in order of increasing cost. *Pesquisa Operacional*, 25(2):219–229, 2005.

31 Robert E. Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55(2):221–232, 1985. doi:10.1016/0012-365X(85)90051-2.

32 Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms, 2003. NII Technical Report NII-2003-004E, Tokyo, Japan.

33 Kunihiro Wasa and Takeaki Uno. Efficient enumeration of bipartite subgraphs in graphs. *CoRR*, abs/1803.03839, 2018. arXiv:1803.03839.

34 Perrin Wright. Counting and constructing minimal spanning trees. *Bulletin of the Institute of Combinatorics and its Applications*, 21:65–76, 1997.

35 Takeo Yamada, Seiji Kataoka, and Kohtaro Watanabe. Listing all the minimum spanning trees in an undirected graph. *International Journal of Computer Mathematics*, 87(14):3175–3185, 2010.