Output-Sensitive Listing of Bounded-Size Trees in Undirected Graphs

Rui Ferreira[1], Roberto Grossi[1], and Romeo Rizzi[2]

[1] Università di Pisa
{ferreira,grossi}@di.unipi.it
[2] Università degli Studi di Udine
romeo.rizzi@uniud.it

Abstract. Motivated by the discovery of combinatorial patterns in an undirected graph G with n vertices and m edges, we study the problem of listing all the trees with k vertices that are subgraphs of G. We present the first optimal output-sensitive algorithm, i.e. runs in $O(sk)$ time where s is the number of these trees in G, and uses $O(m)$ space.

1 Introduction

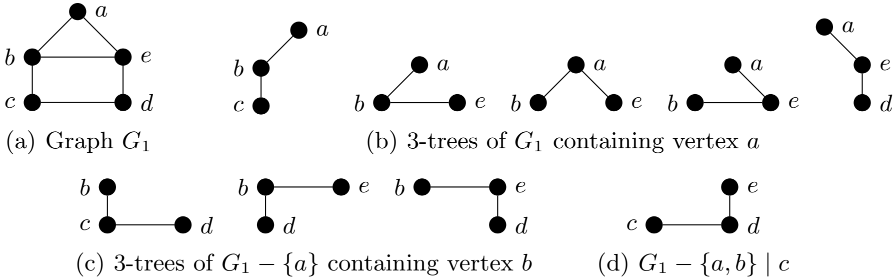
Graphs are employed to model a variety of problems ranging from social to biological networks. Some applications require the extraction of combinatorial patterns [1,5] with the objective of gaining insights into the structure, behavior, and role of the elements in these networks.

Consider an undirected connected graph $G = (V, E)$ of n vertices and m edges. We want to list all the k-trees in G. We define a k-tree T as an edge subset $T \subseteq E$ that is acyclic and connected, and contains k vertices. We denote by s the number of k-trees in G. For example, there are $s = 9$ k-trees in the graph of Fig. 1, where $k = 3$. We present the *first optimal output-sensitive* algorithm for listing all the k-trees in $O(sk)$ time, using $O(m)$ space.

As a special case, our basic problem models also the classical problem of listing the spanning trees in G, which has been largely investigated (here $k = n$ and s is the number of spanning trees in G). The first algorithmic solutions appeared in the 60's [6], and the combinatorial papers even much earlier [7]. Read and Tarjan gave an output-sensitive algorithm in $O(sm)$ time and $O(m)$ space [9]. Gabow and Myers proposed the first algorithm [3] which is optimal when the spanning trees are explicitly listed. When the spanning trees are implicitly enumerated, Kapoor and Ramesh [4] showed that an elegant incremental representation is possible by storing just the $O(1)$ information needed to reconstruct a spanning tree from the previously enumerated one, giving $O(m+s)$ time and $O(mn)$ space [4], later reduced to $O(m)$ space by Shioura et al. [10]. We are not aware of any non-trivial output-sensitive solution for the problem of listing the k-trees in the general case.

We present our solution starting from the well-known binary partition method. (Other known methods are those based on Gray codes and reverse search [2].)

C. Demetrescu and M.M. Halldórsson (Eds.): ESA 2011, LNCS 6942, pp. 275–286, 2011.
© Springer-Verlag Berlin Heidelberg 2011



(a) Graph G_1 (b) 3-trees of G_1 containing vertex a

(c) 3-trees of $G_1 - \{a\}$ containing vertex b (d) $G_1 - \{a, b\} \mid c$

Fig. 1. Example graph G_1 and its 3-trees

We divide the problem of listing all the k-trees in two subproblems by choosing an edge $e \in E$: we list the k-trees that contain e and those that do not contain e. We proceed recursively on these subproblems until there is just one k-tree to be listed. This method induces a binary recursion tree, and all the k-trees can be listed when reaching the leaves of the recursion tree.

Although output sensitive, this simple method is not optimal since it takes $O(s(m+n))$ time. One problem is that the adjacency lists of G can be of length $O(n)$ each, but we cannot pay such a cost in each recursive call. Also, we need a *certificate* that should be easily maintained through the recursive calls to guarantee *a priori* that there will be at least one k-tree generated. By exploiting more refined structural properties of the recursion tree, we present our algorithmic ideas until an optimal output-sensitive listing is obtained, i.e. $O(sk)$ time. Our presentation follows an incremental approach to introduce each idea, so as to evaluate its impact in the complexity of the corresponding algorithms.

2 Preliminaries

Given a simple (without self-loops or parallel edges), undirected and connected graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, and an integer $k \in [2, n]$, a *k-tree* T is an acyclic connected subgraph of G with k vertices. We denote the total number of k-trees in G by s, where $sk \geq m \geq n - 1$ since G is connected.

*Problem 1 (**k-tree listing**).* Given an input graph G and an integer k, list all the k-trees of G.

We say that an algorithm that solves Problem 1 is *optimal* if it takes $O(sk)$ time, since the latter is proportional to the time taken to explicitly list the output, namely, the $k - 1$ edges in each of the s listed k-trees. We also say that the algorithm has *delay* $t(k)$ if it takes $O(t(k))$ time to list a k-tree after having listed the previous one.

We adopt the standard representation of graphs using adjacency lists $\texttt{adj}(v)$ for each vertex $v \in V$. We maintain a counter for each $v \in V$, denoted by $|\texttt{adj}(v)|$, with the number of edges in the adjacency list of v. Additionally, as the graph is undirected, the notations (u, v) and (v, u) represent the same edge.

Let $X \subseteq E$ be a *connected edge* set. We denote by $V[X] \equiv \{u \mid (u,v) \in X\}$ the set of its endpoints, and its *ordered vertex list* $\hat{V}(X)$ recursively as follows: $\hat{V}(\{\cdot, u_0)\}) = \langle u_0 \rangle$ and $\hat{V}(X + (u,v)) = \hat{V}(X) + \langle v \rangle$ where $u \in V[X]$, $v \notin V[X]$, and $+$ denotes list concatenation. We also use the shorthand $E[X] \equiv \{(u,v) \in E \mid u, v \in V[X]\}$ for the induced edges. In general, $G[X] = (V[X], E[X])$ denotes the subgraph of G *induced* by X, which is equivalently defined as the subgraph of G induced by the vertices in $V[X]$.

The *cutset* of X is the set of edges $C(X) \subseteq E$ such that $(u,v) \in C(X)$ if and only if $u \in V[X]$ and $v \in V - V[X]$. Note that when $V[X] = V$, the cutset is empty. Similarly, the *ordered cutlist* $\hat{C}(X)$ contains the edges in $C(X)$ ordered by the rank of their endpoints in $\hat{V}(X)$. If two edges have the same endpoint vertex $v \in \hat{V}(S)$, we use the order as they appear in $\texttt{adj}(v)$ to break the tie.

Throughout the paper we represent an unordered k'-tree $T = \langle e_1, e_2, \ldots, e_{k'} \rangle$ with $k' \leq k$ as an *ordered*, connected and acyclic list of k' edges, where we use a *dummy* edge $e_1 = (\cdot, v_i)$ having a vertex v_i of T as endpoint. The order is the one by which we discover the edges $e_1, e_2, \ldots, e'_k$. Nevertheless, we do not generate two different orderings for the same T.

3 Basic Approach: Recursion Tree

We begin by presenting a simple algorithm that solves Problem 1 in $O(sk^3)$ time, while using $O(mk)$ space. Note that the algorithm is not optimal yet: we will show in Sections 4–5 how to improve it to obtain an optimal solution with $O(m)$ space and delay $t(k) = k^2$.

Top level. We use the standard idea of fixing an ordering of the vertices in $V = \langle v_1, v_2, \ldots, v_n \rangle$. For each $v_i \in V$, we list the k-trees that include v_i and do not include any previous vertex $v_j \in V$ $(j < i)$. After reporting the corresponding k-trees, we remove v_i and its incident edges from our graph G. We then repeat the process, as summarized in Algorithm 1. Here, S denotes a k'-tree with $k' \leq k$, and we use the dummy edge $(\cdot, v_i)$ as a start-up point, so that the ordered vertex list is $\hat{V}(S) = \langle v_i \rangle$. Then, we find a k-tree by performing a DFS starting from v_i: when we meet the kth vertex, we are sure that there exists at least one k-tree for v_i and execute the binary partition method with $\texttt{ListTrees}_{v_i}$; otherwise, if there is no such k-tree, we can skip v_i safely. We exploit some properties on the recursion tree and an efficient implementation of the following operations on G:

- $\texttt{del}(u)$ deletes a vertex $u \in V$ and all its incident edges.
- $\texttt{del}(e)$ deletes an edge $e = (u,v) \in E$. The inverse operation is denoted by $\texttt{undel}(e)$. Note that $|\texttt{adj}(v)|$ and $|\texttt{adj}(u)|$ are updated.
- $\texttt{choose}(S)$, for a k'-tree S with $k' \leq k$, returns an edge $e \in C(S)$: e^- the vertex in e that belongs to $V[S]$ and by e^+ the one s.t. $e^+ \in V - V[S]$.
- $\texttt{dfs}_k(S)$ returns the list of the tree edges obtained by a *truncated DFS*, where conceptually S is treated as a *single* (collapsed vertex) source whose adjacency list is the cutset $C(S)$. The DFS is truncated when it finds k tree edges (or less if there are not so many). The resulting list is a k-tree (or

Algorithm 1. ListAllTrees($G = (V, E)$, k)

1. for $i = 1, 2, \ldots, n-1$:
   (a) $S := \langle(\cdot, v_i)\rangle$
   (b) if $|\texttt{dfs}_k(S)| = k$ then $\texttt{ListTrees}_{v_i}(S)$
   (c) $\texttt{del}(v_i)$

Algorithm 2. $\texttt{ListTrees}_{v_i}(S)$

1. if $|S| = k$ then:
   (a) $\texttt{output}(S)$
   (b) `return`
2. $e := \texttt{choose}(S)$
3. $\texttt{ListTrees}_{v_i}(S + \langle e\rangle)$
4. $\texttt{del}(e)$
5. if $|\texttt{dfs}_k(S)| = k$ then $\texttt{ListTrees}_{v_i}(S)$
6. $\texttt{undel}(e)$

smaller) that includes all the edges in S. Its purpose is to check if there exists a connected component of size at least k that contains S.

Lemma 2. *Given a graph G and a k'-tree S, we can implement the following operations:* $\texttt{del}(u)$ *for a vertex u in time proportional to u's degree;* $\texttt{del}(e)$ *and* $\texttt{undel}(e)$ *for an edge e in $O(1)$ time;* $\texttt{choose}(S)$ *and* $\texttt{dfs}_k(S)$ *in $O(k^2)$ time.*

Recursion tree and analysis. The recursive binary partition method in Algorithm 2 is quite simple, and takes a k'-tree S with $k' \leq k$ as input. The purpose is that of listing all k-trees that include all the edges in S (excluding those with endpoints $v_1, v_2, \ldots, v_{i-1}$). The precondition is that we recursively explore S if and only if there is at least a k-tree to be listed. The corresponding recursion tree has some interesting properties that we exploit during the analysis of its complexity. The root of this binary tree is associated with $S = \langle(\cdot, v_i)\rangle$. Let S be the k'-tree associated with a node in the recursion tree. Then, left branching occurs by taking an edge $e \in C(S)$ using `choose`, so that the *left child* is $S + \langle e\rangle$. Right branching occurs when e is deleted using `del`, and the *right child* is still S but on the reduced graph $G := (V, E - \{e\})$. Returning from recursion, restore G using $\texttt{undel}(e)$. Note that we do *not* generate different permutations of the same k'-tree's edges as we either take an edge e as part of S or remove it from the graph by the binary partition method.

Lemma 3 (Correctness). *Algorithm 2 lists each k-tree containing vertex v_i and no vertex v_j with $j < i$, once and only once.*

A closer look at the recursion tree reveals that it is *k-left-bounded*: namely, each root-to-leaf path has exactly $k-1$ *left branches*. Since there is a one-to-one correspondence between the leaves and the k-trees, we are guaranteed that leftward branching occurs less than k times to output a k-tree.

What if we consider rightward branching? Note that the height of the tree is less than m, so we might have to branch rightward $O(m)$ times in the worst case. Fortunately, we can prove in Lemma 4 that for each internal node S of the recursion tree that has a right child, S has always its left child (which leads to one k-tree). This is subtle but very useful in our analysis in the rest of the paper.

Lemma 4. *At each node S of the recursion tree, if there exists a k-tree (descending from S's right child) that does not include edge e, then there is a k-tree (descending from S's left child) that includes e.*

Note that the symmetric situation for Lemma 4 does not necessarily hold. We can find nodes having just the left child: for these nodes, the chosen edge cannot be removed since this gives rise to a connected component of size smaller than k. We can now state how many nodes there are in the recursion tree.

Corollary 5. *Let s_i be the number of k-trees reported by* $\texttt{ListTrees}_{v_i}$. *Then, its recursion tree is binary and contains s_i leaves and at most $s_i\,k$ internal nodes. Among the internal nodes, there are $s_i - 1$ of them having two children.*

Lemma 6 (Time and space complexity). *Algorithm 2 takes $O(s_i\,k^3)$ time and $O(mk)$ space, where s_i is the number of k-trees reported by* $\texttt{ListTrees}_{v_i}$.

Theorem 7. *Algorithm 1 can solve Problem 1 in $O(nk^2 + sk^3) = O(sk^3)$ time and $O(mk)$ space.*

4 Improved Approach: Certificates

A way to improve the running time of $\texttt{ListTrees}_{v_i}$ to $O(s_i k^2)$ is indirectly suggested by Corollary 5. Since there are $O(s_i)$ binary nodes and $O(s_i k)$ unary nodes in the recursion tree, we can pay $O(k^2)$ time for binary nodes and $O(1)$ for unary nodes (i.e. reduce the cost of `choose` and $\texttt{dfs}_k$ to $O(1)$ time when we are in a *unary* node). This way, the total running time is $O(sk^2)$.

The idea is to maintain a certificate that can tell us if we are in a unary node in $O(1)$ time and that can be updated in $O(1)$ time in such a case, or can be completely rebuilt in $O(k^2)$ time otherwise (i.e. for binary nodes). This will guarantee a total cost of $O(s_i k^2)$ time for $\texttt{ListTrees}_{v_i}$, and lay out the path to the wanted optimal output-sensitive solution of Section 5.

4.1 Introducing Certificates

We impose an "unequivocal behavior" to $\texttt{dfs}_k(S)$, obtaining a variation denoted $\texttt{mdfs}_k(S)$ and called *multi-source truncated DFS*. During its execution, $\texttt{mdfs}_k$ takes the order of the edges in S into account (whereas an order is not strictly necessary in $\texttt{dfs}_k$). Specifically, given a k'-tree $S = \langle e_1, e_2, \ldots, e_{k'} \rangle$, the returned k-tree $D = \texttt{mdfs}_k(S)$ contains S, which is conceptually treated as a collapsed vertex: the main difference is that S's "adjacency list" is now the *ordered cutlist* $\hat{C}(S)$, rather than $C(S)$ employed for $\texttt{dfs}_k$.

Equivalently, since $\hat{C}(S)$ is induced from $C(S)$ by using the ordering in $\hat{V}(S)$, we can see $\texttt{mdfs}_k(S)$ as the execution of multiple standard DFSes from the vertices in $\hat{V}(S)$, in that order. Also, all the vertices in $V[S]$ are conceptually marked as visited at the beginning of $\texttt{mdfs}_k$, so u_j is never part of the DFS tree starting from u_i for any two distinct $u_i, u_j \in V[S]$. Hence the adopted terminology of multi-source. Clearly, $\texttt{mdfs}_k(S)$ is a feasible solution to $\texttt{dfs}_k(S)$ while the vice versa is not true.

We use the notation $S \sqsubseteq D$ to indicate that $D = \texttt{mdfs}_k(S)$, and so D is a *certificate* for S: it guarantees that node S in the recursion tree has at least one descending leaf whose corresponding k-tree has not been listed so far. Since the behavior of $\texttt{mdfs}_k$ is non-ambiguous, relation $\sqsubseteq$ is well defined. We preserve the following invariant on $\texttt{ListTrees}_{v_i}$, which now has two arguments.

Invariant 1 *For each call to* $\texttt{ListTrees}_{v_i}(S, D)$*, we have* $S \sqsubseteq D$*.*

Before showing how to keep the invariant, we detail how to represent the certificate D in a way that it can be efficiently updated. We maintain it as a partition $D = S \cup L \cup F$, where S is the given list of edges, whose endpoints are kept in order as $\hat{V}(S) = \langle u_1, u_2, \ldots, u_{k'} \rangle$. Moreover, $L = D \cap C(S)$ are the tree edges of D in the cutset $C(S)$, and F is the forest storing the edges of D whose both endpoints are in $V[D] - V[S]$.

(i) We store the k'-tree S as a sorted doubly-linked list of k' edges $\langle e_1, e_2, \ldots, e_{k'} \rangle$, where $e_1 := (\cdot, v_i)$. We also keep the sorted doubly-linked list of vertices $\hat{V}(S) = \langle u_1, u_2, \ldots, u_{k'} \rangle$ associated with S, where $u_1 := v_i$. For $1 \le j \le k'$, we keep the number of tree edges in the cutset that are incident to u_j, namely $\eta[u_j] = |\{(u_j, x) \in L\}|$.

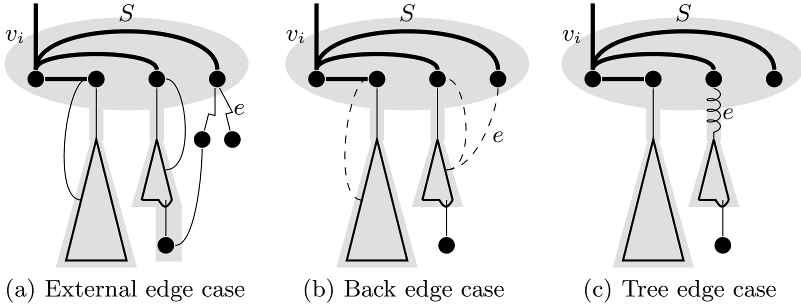
(ii) We keep $L = D \cap C(S)$ as an ordered doubly-linked list of edges in $\hat{C}(S)$'s order: it can be easily obtained by maintaining the parent edge connecting a root in F to its parent in $\hat{V}(S)$.

(iii) We store the forest F as a sorted doubly-linked list of the roots of the trees in F. The order of this list is that induced by $\hat{C}(S)$: a root r precedes a root t if the (unique) edge in L incident to r appears before the (unique) edge of L incident to t. For each node x of a tree $T \in F$, we also keep its number $\texttt{deg}(x)$ of children in T, and its predecessor and successor sibling in T.

(iv) We maintain a flag $\texttt{is_unary}$ that is true if and only if $|\texttt{adj}(u_i)| = \eta[u_i] + \sigma(u_i)$ for all $1 \le i \le k'$, where $\sigma(u_i) = |\{(u_i, u_j) \in E \mid i \ne j\}|$ is the number of internal edges, namely, having both endpoints in $V[S]$.

Throughout the paper, we identify D with both *(1)* the set of k edges forming it as a k-tree and *(2)* its representation above as a certificate. We also support the following operations on D, under the requirement that $\texttt{is_unary}$ is true (i.e. all the edges in the cutset $C(S)$ are tree edges), otherwise they are undefined:

- $\texttt{treecut}(D)$ returns the last edge in L.
- $\texttt{promote}(r, D)$, where root r is the last in the doubly-linked list for F: remove r from F and replace r with its children $r_1, r_2, \ldots, r_c$ (if any) in that list, so they become the new roots (and so L is updated).



(a) External edge case (b) Back edge case (c) Tree edge case

Fig. 2. Choosing edge $e \in C(S)$. The certificate D is shadowed

Lemma 8. *The representation of certificate $D = S \cup L \cup F$ requires $O(|D|) = O(k)$ memory words, and* $\texttt{mdfs}_k(S)$ *can build D in $O(k^2)$ time. Moreover, each of the operations* `treecut` *and* `promote` *can be supported in $O(1)$ time.*

4.2 Maintaining the Invariant Using a New `choose`

We now define `choose` in a more refined way to facilitate the task of maintaining the invariant $S \sqsubseteq D$ introduced in Section 4.1. As an intuition, `choose` selects an edge $e = (e^-, e^+)$ from the *cutlist* $\hat{C}(S)$ that interferes as least as possible with the certificate D. Recalling that $e^- \in V[S]$ and $e^+ \in V - V[S]$ by definition of cutlist, we consider the following case analysis:

(a) [external edge] Check if there exists an edge $e \in \hat{C}(S)$ such that $e \notin D$ and $e^+ \notin V[D]$. If so, return e, shown as a saw in Figure 2(a).
(b) [back edge] Otherwise, check if there exists an edge $e \in \hat{C}(S)$ such that $e \notin D$ and $e^+ \in V[D]$. If so, return e, shown dashed in Figure 2(b).
(c) [tree edge] As a last resort, every $e \in \hat{C}(S)$ must be also $e \in D$ (i.e. all edges in the cutlist are tree edges). Return $e :=$ `treecut`(D), the last edge from $\hat{C}(S)$, shown as a coil in Fig. 2(c).

Lemma 9. *For a given k'-tree S, consider its corresponding node in the recursion tree. Then, this node is* binary *when* `choose` *returns an external or back edge (cases (a)–(b)) and is* unary *when* `choose` *returns a tree edge (case (c)).*

We now present the new listing approach in Algorithm 3. If the connected component of vertex v_i in the residual graph is smaller than k, we delete its vertices since they cannot provide k-trees, and so we skip them in this way. Otherwise, we launch the new version of $\texttt{ListTrees}_{v_i}$, shown in Algorithm 4. In comparison with the previous version (Algorithm 2), *we produce the new certificate D' from the current D in $O(1)$ time when we are in a unary node.* On the other hand, we completely rebuild the certificate twice when we are in a binary nodes (since either child could be unary at the next recursion level).

Lemma 10. *Algorithm 4 correctly maintains the invariant $S \sqsubseteq D$.*

Algorithm 3. ListAllTrees($G = (V, E)$, k)

for $v_i \in V$:
1. $S := \langle(\cdot, v_i)\rangle$
2. $D :=$ `mdfs`$_k(S)$
3. if $|D| < k$ then
   (a) for $u \in V[D]$: `del`(u).
4. else
   (a) `ListTrees`$_{v_i}(S, D)$
   (b) `del`(v_i)

Algorithm 4. `ListTrees`$_{v_i}(S, D)$ $\{$*Invariant:* $S \sqsubseteq D\}$

1. if $|S| = k$ then:
   (a) `output`(S)
   (b) `return`
2. $e :=$ `choose`(S, D)
3. if `is_unary`:
   (a) $D' :=$ `promote`(e^+, D)
   (b) `ListTrees`$_{v_i}(S + \langle e\rangle, D')$
4. else:
   (a) $D' :=$ `mdfs`$_k(S + \langle e\rangle)$
   (b) `ListTrees`$_{v_i}(S + \langle e\rangle, D')$
   (c) `del`(e)
   (d) $D'' :=$ `mdfs`$_k(S)$
   (e) `ListTrees`$_{v_i}(S, D'')$
   (f) `undel`(e)

4.3 Analysis

We implement `choose`(S, D) so that it can now exploit the information in D. At each node S of the recursion tree, when it selects an edge e that belongs to the cutset $C(S)$, it first considers the edges in $C(S)$ that are external or back (cases (a)–(b)) before the edges in D (case (c)).

Lemma 11. *There is an implementation of* `choose` *in* $O(1)$ *for unary nodes in the recursion tree and* $O(k^2)$ *for binary nodes.*

Lemma 12. *Algorithm 4 takes* $O(s_i\, k^2)$ *time and* $O(mk)$ *space, where* s_i *is the number of* k*-trees reported by* `ListTrees`$_{v_i}$.

Theorem 13. *Algorithm 3 solves Problem 1 in* $O(sk^2)$ *time and* $O(mk)$ *space.*

Proof. The vertices belonging to the connected components of size less than k in the residual graph, now contribute with $O(m)$ total time rather than $O(nk^2)$. The rest of the complexity follows from Lemma 12. □

5 Optimal Approach: Amortization

In this section, we discuss how to adapt Algorithm 4 so that a more careful analysis can show that it takes $O(sk)$ time to list the k-trees. Considering $\texttt{ListTrees}_{v_i}$, observe that each of the $O(s_i k)$ unary nodes requires a cost of $O(1)$ time and therefore they are not much of problem. On the contrary, each of the $O(s_i)$ binary nodes takes $O(k^2)$ time: our goal is to improve over this situation.

Consider the operations on a binary node S of the recursion tree that take $O(k^2)$ time, namely: (I) $e := \texttt{choose}(S, D)$; (II) $D' := \texttt{mdfs}_k(S')$, where $S' \equiv S + \langle e \rangle$; and (III) $D'' := \texttt{mdfs}_k(S)$ in $G - \{e\}$. In all these operations, while scanning the adjacency lists of vertices in $V[S]$, we visit some edges $e' = (u, v)$, named *internal*, such that $e' \notin S$ with $u, v \in V[S]$. These internal edges of $V[S]$ can be visited even if they were previously visited on an ancestor node to S. In Section 5.1, we show how to amortize the cost induced by the internal edges. In Section 5.2, we show how to amortize the cost induced by the remaining edges and obtain a delay of $t(k) = k^2$ in our optimal output-sensitive algorithm.

5.1 Internal Edges of $V[S]$

To avoid visiting the internal edges of $V[S]$ several times throughout the recursion tree, we remove these edges from the graph G on the fly, and introduce a global data structure, which we call *parking lists*, to store them temporarily. Indeed, out of the possible $O(n)$ incident edges in vertex $u \in V[S]$, less than k are internal: it is simply too costly removing these internal edges by a complete scan of $\texttt{adj}(u)$. Therefore we remove them as they appear while executing `choose` and $\texttt{mdfs}_k$ operations.

Formally, we define *parking lists* as a global array P of n pointers to lists of edges, where $P[u]$ is the list of internal edges discovered for $u \in V[S]$. When $u \notin V[S]$, $P[u]$ is null. On the implementation level, we introduce a slight modification of the `choose` and $\texttt{mdfs}_k$ algorithms such that, when they meet for the first (and only) time an internal edge $e' = (u, v)$ with $u, v \in V[S]$, they perform $\texttt{del}(e')$ and add e' at the end of both parking lists $P[u]$ and $P[v]$. We also keep a cross reference to the occurrences of e' in these two lists.

Additionally, we perform a small modification in algorithm $\texttt{ListTrees}_{v_i}$ by adding a fifth step in Algorithm 4 just before it returns to the caller. Recall that on the recursion node $S + \langle e \rangle$ with $e = (e^-, e^+)$, we added the vertex e^+ to $V[S]$. Therefore, when we return from the call, all the internal edges incident to e^+ are no longer internal edges (and are the only internal edges to change status). On this new fifth step, we scan $P[e^+]$ and for each edge $e' = (e^+, x)$ in it, we remove e' from both $P[e^+]$ and $P[x]$ in $O(1)$ time using the cross reference. Note that when the node is unary there are no internal edges incident to e^+, so $P[e^+]$ is empty and the total cost is $O(1)$. When the node is binary, there are at most $k - 1$ edges in $P[e^+]$, so the cost is $O(k)$.

Lemma 14. *The operations over internal edges done in* $\texttt{ListTrees}_{v_i}$ *have a total cost of* $O(s_i k)$ *time.*

5.2 Amortization

Let us now focus on the contribution given by the remaining edges, which are not internal for the current $V[S]$. Given the results in Section 5.1, for the rest of this section *we can assume wlog that there are no internal edges* in $V[S]$, namely, $E[S] = S$. We introduce two metrics that help us to parameterize the time complexity of the operations done in binary nodes of the recursion tree.

The first metric we introduce is helpful when analyzing the operation `choose`. For connected edge sets S and X with $S \sqsubseteq X$, define the *cut number* γ_X as the number of edges in the induced (connected) subgraph $G[X] = (V[X], E[X])$ that are in the cutset $C(S)$ (i.e. tree edges plus back edges): $\gamma_X = |\, E[X] \cap C(S)\,|$.

Lemma 15. *For a binary node S with certificate D,* `choose`(S, D) *takes $O(k + \gamma_D)$ time.*

For connected edge sets S and X with $S \sqsubseteq X$, the second metric is the *cyclomatic number* ν_X (also known as circuit rank, nullity, or dimension of cycle space) as the smallest number of edges which must be removed from $G[X]$ so that no cycle remains in it: $\nu_X = |\, E[X]\,| - |\, V[X]\,| + 1$.

Using the cyclomatic number of a certificate D (ignoring the internal edges of $V[S]$), we obtain a lower bound on the number of k-trees that are output in the leaves descending from a node S in the recursion tree.

Lemma 16. *Considering the cyclomatic number ν_D and the fact that $|V[D]| = k$, we have that $G[D]$ contains at least ν_D k-trees.*

Lemma 17. *For a node S with certificate D, computing $D' =$* `mdfs`$_k(S)$ *takes $O(k + \nu_{D'})$ time.*

Recalling that the steps done on a binary node S with certificate D are: (I) $e :=$ `choose`(S, D); (II) $D' :=$ `mdfs`$_k(S')$, where $S' \equiv S + \langle e \rangle$; and (III) $D'' :=$ `mdfs`$_k(S)$ in $G - \{e\}$, they take a total time of $O(k + \gamma_D + \nu_{D'} + \nu_{D''})$. We want to pay $O(k)$ time on the recursion node S and amortize the *rest* of the cost to some suitable nodes descending from its *left child* S' (with certificate D'). To do this we are to relate γ_D with $\nu_{D'}$ and avoid performing step (III) in $G - \{e\}$ by maintaining D'' from D'. We exploit the property that the cost $O(k + \nu_{D'})$ for a node S in the recursion tree can be amortized using the following lemma:

Lemma 18. *Let S' be the left child (with certificate D') of a generic node S in the recursion tree. The sum of $O(\nu_{D'})$ work, over all left children S' in the recursion tree is upper bounded by $\sum_{S'} \nu_{D'} = O(s_i k)$.*

Proof. By Lemma 16, S' has at least $\nu_{D'}$ descending leaves. Charge $O(1)$ to each leaf descending from S' in the recursion tree. Since S' is a left child and we know that the recursion tree is k-left-bounded by Lemma 4, each of the s_i leaves can be charged at most k times, so $\sum_{S'} \nu_{D'} = O(s_i k)$ for all such S'. □

We now show how to amortize the $O(k + \gamma_D)$ cost of step (I). Let us define $comb(S')$ for a left child S' in the recursion tree as its maximal path to the right (its right spine) and the left child of each node in such a path. Then, $|comb(S')|$ is the number of such left children.

Lemma 19. *On a node S in the recursion tree, the cost of* `choose`(S, D) *is* $O(k + \gamma_D) = O(k + \nu_{D'} + |comb(S')|)$.

Proof. Consider the set E' of γ_D edges in $E[D] \cap C(S)$. Take D', which is obtained from $S' = S + \langle e \rangle$, and classify the edges in E' accordingly. Given $e' \in E'$, one of three possible situations may arise: either e' becomes a tree edge part of D' (and so it contributes to the term k), or e' becomes a back edge in $G[D']$ (and so it contributes to the term $\nu_{D'}$), or e' becomes an external edge for D'. In the latter case, e' will be chosen in one of the subsequent recursive calls, specifically one in $comb(S')$ since e' is still part of $C(S')$ and will surely give rise to another k-tree in a descending leaf of $comb(S')$. □

While the $O(\nu_{D'})$ cost over the leaves of S' can be amortized by Lemma 17, we need to show how to amortize the cost of $|comb(S')|$ using the following:

Lemma 20. $\sum_{S'} |comb(S')| = O(s_i k)$ *over all left children S' in the recursion.*

At this point we are left with the cost of computing the two $\mathtt{mdfs}_k$'s. Note that the cost of step (II) is $O(k + \nu_{D'})$, and so is already expressed in terms of the cyclomatic number of its left child, $\nu_{D'}$ (so we use Lemma 16). The cost of step (III) is $O(k + \nu_{D''})$, expressed with the cyclomatic number of the certificate of its *right* child. This cost is not as easy to amortize since, when the edge e returned by `choose` is a back edge, D' of node $S + \langle e \rangle$ can change *heavily* causing D' to have just S in common with D''. This shows that $\nu_{D''}$ and $\nu_{D'}$ are not easily related.

Nevertheless, note that D and D'' are the same certificate since we only remove from G an edge $e \notin D$. The only thing that can change by removing edge $e = (e^-, e^+)$ is that the right child of node S' is no longer binary (i.e. we removed the last back edge). The question is if we can check quickly whether it is unary in $O(k)$ time: observe that $|\mathtt{adj}(e^-)|$ is no longer the same, invalidating the flag `is_unary` (item (iv) of Section 4.1). Our idea is the following: instead of recomputing the certificate D'' in $O(k + \nu_{D''})$ time, we update the `is_unary` flag in just $O(k)$ time. We thus introduce a new operation $D'' = \mathtt{unary}(D)$, a valid replacement for $D'' = \mathtt{mdfs}_k(D)$ in $G - \{e\}$: it maintains the certificate while recomputing the flag `is_unary` in $O(k)$ time.

Lemma 21. *Operation* `unary`(D) *takes $O(k)$ time and correctly computes D''.*

Since there is no modification or impact on unary nodes of the recursion tree, we finalize the analysis.

Lemma 22. *The cost of* $\mathtt{ListTrees}_{v_i}(S, D)$ *on a binary node S is* $O(k + \nu_{D'} + |comb(S')|)$.

Lemma 23. *The algorithm* $\mathtt{ListTrees}_{v_i}$ *takes $O(s_i k)$ time and $O(mk)$ space.*

Note that our data structures are lists and array, so it is not difficult to replace them with *persistent* arrays and lists, a classical trick in data structures. As a result, we just need $O(1)$ space per pending recursive call, plus the space of the parking lists, which makes a total of $O(m)$ space.

Theorem 24. *Algorithm 3 takes a total of $O(sk)$ time, being therefore optimal, and $O(m)$ space.*

We finally show how to obtain an efficient delay. We exploit the following property on the recursion tree, which allows to associate a unique leaf with an internal node *before* exploring the subtree of that node (recall that we are in a recursion tree). Note that only the rightmost leaf descending from the root is not associated in this way, but we can easily handle this special case.

Lemma 25. *For a binary node S in the recursion tree, $\texttt{ListTrees}_{v_i}(S, D)$ outputs the k-tree D in the rightmost leaf descending from its left child S'.*

Nakano and Uno [8] have introduced this nice trick. Classify a binary node S in the recursion tree as even (resp., odd) if it has an even (resp., odd) number of ancestor nodes that are binary. Consider the simple modification to $\texttt{ListTrees}_{v_i}$ when S is binary: if S is even then output D *immediately before* the two recursive calls; otherwise (S is odd), output D *immediately after* the two recursive calls.

Theorem 26. *Algorithm 3 can be implemented with delay $t(k) = k^2$.*

References

1. Alm, E., Arkin, A.P.: Biological networks. Current Opinion in Structural Biology 13(2), 193–202 (2003)
2. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Applied Mathematics 65(1-3), 21–46 (1996)
3. Gabow, H.N., Myers, E.W.: Finding all spanning trees of directed and undirected graphs. SIAM Journal on Computing 7(3), 280–287 (1978)
4. Kapoor, S., Ramesh, H.: Algorithms for enumerating all spanning trees of undirected and weighted graphs. SIAM Journal on Computing 24, 247–265 (1995)
5. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: Simple building blocks of complex networks. Science 298, 824–827 (2002)
6. Minty, G.: A simple algorithm for listing all the trees of a graph. IEEE Transactions on Circuit Theory 12(1), 120 (1965)
7. Moon, J.: Counting Labelled Trees, Canadian Mathematical Monographs, No. 1. Canadian Mathematical Congress, Montreal (1970)
8. Nakano, S.I., Uno, T.: Constant time generation of trees with specified diameter. In: Hromkovič, J., Nagl, M., Westfechtel, B. (eds.) Graph -Theoretic Concepts in Computer Science. LNCS, vol. 3353, pp. 33–45. Springer, Heidelberg (2004)
9. Read, R.C., Tarjan, R.E.: Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. Networks (1975)
10. Shioura, A., Tamura, A., Uno, T.: An optimal algorithm for scanning all spanning trees of undirected graphs. SIAM Journal on Computing 26, 678–692 (1994)