Published: 08 December 2025
Online AM: 22 October 2025
Accepted: 07 October 2025
Revised: 01 July 2025
Received: 20 August 2024

RESEARCH-ARTICLE

Fast Assessment of Eulerian Trails in Graphs with Applications

ALESSIO CONTE, University of Pisa, Pisa, PI, Italy

ROBERTO GROSSI, University of Pisa, Pisa, PI, Italy

GRIGORIOS LOUKIDES, King's College London, London, U.K.

NADIA PISANTI, University of Pisa, Pisa, PI, Italy

SOLON P PISSIS, Free University Amsterdam, Amsterdam, Noord-Holland, Netherlands

GIULIA PUNZI, University of Pisa, Pisa, PI, Italy

Fast Assessment of Eulerian Trails in Graphs with Applications

ALESSIO CONTE and ROBERTO GROSSI, University of Pisa, Pisa, Italy
GRIGORIOS LOUKIDES, King's College London, London, UK
NADIA PISANTI, University of Pisa, Pisa, Italy
SOLON P. PISSIS, CWI, Amsterdam, The Netherlands and Vrije Universiteit, Amsterdam, The Netherlands
GIULIA PUNZI, University of Pisa, Pisa, Italy

Enumerating or counting combinatorial objects in graphs is a fundamental data mining task. We consider the problem of assessing the number of Eulerian trails in directed graphs, which is formalized as follows: Given a directed graph $G = (V, E)$, with $|V| = n$ nodes and $|E| = m$ edges, and an integer z, *assess* whether the number $\#ET(G)$ of Eulerian trails of G is at least z. This problem underlies many applications in domains ranging from data privacy to computational biology, data compression, and transportation networks. Practitioners currently address this problem by applying the famous BEST theorem, which, in fact, *counts* $\#ET(G)$ instead of just assessing whether $\#ET(G) \geq z$. Unfortunately, this solution takes $O(n^\omega)$ arithmetic operations, where $\omega < 2.373$ denotes the *matrix multiplication exponent*. Since in most real-world graphs, the number m of edges is comparable to the number n of nodes, and z is moderate in practice, the algorithmic challenge is: *Can we solve the problem faster for certain values of m and z?* We want to design a combinatorial algorithm for assessing whether $\#ET(G) \geq z$, which does not resort to the BEST theorem and has a predictably bounded cost as a function of m and z. We address this challenge as follows. We first introduce a general algorithmic scheme for assessing (and enumerating) Eulerian trails. We then introduce a novel tree data structure to reduce the number of iterations in this general scheme. Finally, we complement the above with further combinatorial insight leading to an algorithm with a worst-case bound of $O(m \cdot \min\{z, \#ET(G)\})$ time. Our experiments using six benchmark datasets with multi-million edges from different domains show that our implementations are up to two orders of magnitude faster than the BEST theorem, perform much fewer than mz iterations and scale near-linearly with m in most cases. Our experiments further show that our implementations bring substantial efficiency benefits in a data privacy application which employs the BEST theorem for the assessment.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**;

This article is part of the PANGAIA project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no. 872539. A. Conte and R. Grossi are supported in part by MUR of Italy, under PRIN Project no. 2022TS4Y3N–EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data, and N. Pisanti is supported by MUR PRIN 2022YRB97K PINC. R. Grossi and N. Pisanti are partially supported by NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem Spoke 6 (CUP B63C2200068007 and I53C22000780001). G. Punzi is supported by the Italian Ministry of Research, under the complementary actions to the NRRP "Fit4MedRob–Fit for Medical Robotics" Grant (PNC0000007).
Authors' Contact Information: Alessio Conte, University of Pisa, Pisa, Italy; e-mail: alessio.conte@unipi.it; Roberto Grossi, University of Pisa, Pisa, Italy; e-mail: roberto.grossi@unipi.it; Grigorios Loukides (corresponding author), King's College London, London, UK; e-mail: grigorios.loukides@kcl.ac.uk; Nadia Pisanti, University of Pisa, Pisa, Italy; e-mail: nadia.pisanti@unipi.it; Solon P. Pissis, CWI, Amsterdam, The Netherlands and Vrije Universiteit, Amsterdam, The Netherlands; e-mail: solon.pissis@cwi.nl; Giulia Punzi, University of Pisa, Pisa, Italy; e-mail: giulia.punzi@unipi.it.

ACM 1556-472X/2025/12-ART14
https://doi.org/10.1145/3771997

Additional Key Words and Phrases: Eulerian trails, Assessment, BEST theorem

Associate Editor: Arijit Khan

ACM Reference format:
Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giulia Punzi. 2025. Fast Assessment of Eulerian Trails in Graphs with Applications. *ACM Trans. Knowl. Discov. Data.* 20, 1, Article 14 (December 2025), 33 pages.
https://doi.org/10.1145/3771997

1 Introduction

Eulerian trails (or Eulerian paths) were introduced by Euler in 1736: Given a multigraph $G = (V, E)$, an Eulerian trail traverses every edge in E exactly once, allowing for revisiting nodes in V. An Eulerian circuit is an Eulerian trail that starts and ends on the same node in V. An example is shown in the graph of Figure 1, where we indicate the edges as $e_1, e_2, \ldots, e_8$, and obtain one of the Eulerian trails from node s to node t by traversing the edges in the following order: $e_1e_3e_6e_5e_2e_4e_7e_8$; note that other Eulerian trails exist in the graph (e.g., $e_1e_3e_7e_2e_4e_6e_5e_8$).[1]

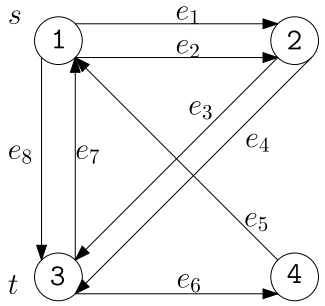
The perhaps most fundamental computational question related to Eulerian trails is whether we can efficiently find one of them. Hierholzer's paper ([1],[2,1B]) can be employed for both undirected and directed graphs to obtain a linear-time algorithm. Another fundamental question is *counting* Eulerian trails. For undirected graphs, this question is difficult to answer as it is $\#P$-complete [3]. Instead, for directed graphs, the number of Eulerian trails can be computed in polynomial time using the BEST theorem [4], named after de Bruijn, van Aardenne-Ehrenfest, Smith, and Tutte. The BEST theorem is the only available tool to achieve this counting in an *efficient* and *exact* manner.

We focus on the scenario where graphs are directed. We make the standard assumption that every node is represented by a unique integer from $[1, |V|]$ so that the node ids are printed while traversing an Eulerian trail, producing what we call the *node sequence* of the trail. For example, the Eulerian trail $e_1e_3e_6e_5e_2e_4e_7e_8$ in the graph of Figure 1 yields the node sequence: 1, 2, 3, 4, 1, 2, 3, 1, 3. This sequence can be seen as a string over $[1, |V|]$. As the same node sequence can arise from two distinct Eulerian trails,[2] Bernardini et al. [5] call two trails *node-distinct* if their node sequences are different and formalize the following *assessment* problem on directed (Eulerian) multigraphs:

Problem 1 (Eulerian Trail Assessment (ETA)). Given a directed multigraph $G = (V, E)$, with $|V| = n$ nodes and $|E| = m$ edges, two nodes $s, t \in V$, and an integer $z > 0$, assess whether the number $\#ET(G)$ of node-distinct Eulerian trails of G with source s and target t is at least z.

In our example from Figure 1, we have that $\#ET(G) = 6$. However, as we will show in this article, solving the ETA problem *does not necessarily imply that we must count*; i.e., compute $\#ET(G)$ explicitly. Actually, we show that *assessing* is much more efficient in practice than exact *counting*.

[1]For clarity, in the figures, we use different labels for the same edge when it has multiplicity more than 1.
[2]Traditionally, two trails are distinct if their *edge* sequences are different. It is more challenging to consider node-distinct Eulerian trails as they yield different node sequences (see Figure 1), whereas some Eulerian trails might correspond to the same node sequence. Our results easily extend to edge-distinct Eulerian trails: split each multi-edge (u, v) of multiplicity h by adding h middle nodes $z_1, \ldots, z_h$ and edges (u, z_i) and (z_i, v), for all i. This increases the total graph size by $O(m)$, and the node-distinct Eulerian trails of this new graph are exactly the edge-distinct Eulerian trails of the original graph.



Node-distinct Eulerian trails	Node sequences
$e_1e_3e_6e_5e_2e_4e_7e_8$	123412313
$e_1e_3e_7e_2e_4e_6e_5e_8$	123123413
$e_1e_3e_7e_8e_6e_5e_2e_4$	123134123
$e_8e_6e_5e_1e_3e_7e_2e_4$	134123123
$e_8e_7e_1e_3e_6e_5e_2e_4$	131234123
$e_1e_3e_6e_5e_8e_7e_2e_4$	123413123

Fig. 1. A directed multigraph over the set of nodes $[1, 4] = \{1, 2, 3, 4\}$; its complete set of node-distinct Eulerian trails starting at node $s = 1$ and ending at node $t = 3$; and the corresponding set of node sequences.

1.1 Why Is ETA Useful?

The ETA problem is the core computational task in several string-processing applications [5–7]. In Section 2, we describe concretely some applications: (A1) defending against reconstruction attacks; and (A2) investigating genome complexity. In these applications, graphs are huge in size (i.e., n is large) and exact counting is too slow due to the $\Omega(n^2)$ time bound of the BEST theorem [5, 6], as we discuss in the next subsection. Since most real-world graphs are sparse, the number m of edges is comparable to the number n of nodes, and since z is moderate in these applications, assessment algorithms having a bounded cost as a function of m and z are highly desirable. In particular, a time bound of $O(mz)$ would be highly desirable for ETA. Further applications are: (A3) finding min-cost trails; and (A4) compressing undirected graphs and networks. In these application domains, it is not always possible to define "the" best solution, but there are several best solutions that should be evaluated according to additional constraints (e.g., a cost function). As our ETA approach can as well enumerate those solutions, it can be employed as a key component in these cases.

On a more general note, there are many other graph objects beyond Eulerian trails that may need to be assessed (instead of being counted directly) or enumerated and for which ideas from our approach may prove useful. Examples include s-t paths (i.e., paths from a given source to a given target node) [8], k-hop constrained s-t simple paths [9], maximal cliques [10, 11], (maximal) k-plexes [12, 13], temporal cycles [14], triangles [15], network motifs [16], and subgraphs related to node centrality measures [17, 18]. In general, enumerating or counting subgraphs of a given type, like triangles or cliques, has applications in social network analysis [19], link prediction and recommendation [20], anomaly detection [21], spam filtering [22], and community discovery [23].

1.2 Why Is ETA Challenging?

The ETA problem can be solved in $O(n^\omega)$ arithmetic operations using matrix multiplication [5, 6], where $\omega < 2.373$ denotes the matrix multiplication exponent [24–26]. This is performed by first *counting* the Eulerian trails exactly using the BEST theorem [27] (see Section 2) and then comparing their number with z. The underlying assumption is that G is Eulerian, that is, the indegree equals the outdegree in each node, possibly except for the source s and the target t of the trail.

However, the $O(n^\omega)$ bound of the BEST theorem is large and in practice the computation is time consuming (see Section 2). Thus, directly using the BEST theorem is unsatisfactory for large graphs. Motivated by this, we address the following algorithmic challenge [5, Final Remarks]: design a combinatorial algorithm to solve the ETA problem, which does not resort to the BEST theorem and has a predictably bounded cost as a function of m and z; note that we do not need the exact value of $\#ET(G)$ to provide an answer for the assessment.

1.3 Our Techniques and Contributions

(1) *Assessment Algorithm.* Our main contribution is to introduce an approach that does not merely employ the structure of the BEST theorem, but goes beyond that: we design an efficient algorithm which directly provides an assessment for $\#ET(G) \geq z$ by looking at Eulerian trails, without considering the different factors of Equation (1). The main idea consists of providing a lower bound on $\#ET(G)$, based on the product of the lower bounds for the node-distinct trails of its **strongly connected components (SCCs)**. This lower bound is then progressively refined by improving the contribution of any arbitrarily chosen component: the component is further decomposed and its contribution updated according to structural properties of Eulerian graphs presented in Section 3. Even though it is written in an iterative form, our method conceptually provides a recursive enumeration approach whose calls enumerate the first z node-distinct Eulerian trails in $\Theta(m^2 \cdot \min\{z, \#ET(G)\})$ time. The first presented algorithm, AssessET, follows this scheme in an immediate way, and is described in Section 4.

(2) *Improved Algorithm.* Our second contribution is a practical improvement of AssessET, aimed at *avoiding repeated computation*. Indeed, the data structure employed by algorithm AssessET is a collection of tuples, whose elements are the SCCs of some subgraphs of G which are currently contributing to the lower bound. Such tuples are overall different, but the same SCC can occur in different tuples: two different considered subgraphs may have an SCC which is the same. By using structural properties of Eulerian trails and their lower bounds, we devise a compact tree data structure, which represents all tuples of AssessET at the same time. The central benefit of such data structure is that it can avoid repetition of the SCCs common to multiple intermediate subgraphs that are currently contributing to the lower bound. This in turn allows for a reduced number of iterations, since no computation will now be performed on the same SCC twice. We thus present algorithm TreeAssessET in Section 5, resulting from employing our novel tree data structure on the previous assessment scheme. This algorithm has the same theoretical running time as AssessET, but it has a better performance in practice, as we will show in our experimental evaluation. An important characteristic of our approach is that it can be modified to *enumerate* $\min\{z, \#ET(G)\}$ Eulerian trails within the same complexity, as described in Sections 4.1 and 5.1.

(3) *Further Improvements.* Both of our proposed algorithms may require quadratic time per Eulerian trail, because each conceptual recursive call might require $O(m)$ time. To overcome this drawback, in Section 6, we present two further improvements which can be applied to both our algorithms. First, we enhance their practical performance with a chain-compression strategy, which shortens long paths in the graph, to reduce the number of conceptual recursive calls. On the theoretical side, we present a way to bring the time complexity of both algorithms down by using a double numbering on the edges, which guarantees that every call generates *at least two distinct calls*. This double numbering gives us insight on the interior connectivity structure of the graph; namely, how SCCs change when we start removing edges, which is the source of the quadratic time. With this double numbering, we manage to instantly retrieve edges that generate new trails, no longer needing to iterate for $O(m)$ unsuccessful steps. We thus slash the time complexity of AssessET by a factor of m, yielding algorithm ImprovedAssessET, which is time-optimal for $z = O(1)$ or for $\#ET(G) = O(1)$. We formalize this theoretical result as follows:

Theorem 1. *Given a directed multigraph $G = (V, E)$, with $|E| = m$, and an integer z, assessing $\#ET(G) \geq z$ can be done in $O(m \cdot \min\{z, \#ET(G)\})$ time using $O(mz)$ space.*

(4) *Experimental Evaluation.* The last contribution of our article is the experimental evaluation of our assessment approach, detailed in Section 7. Since our main focus here is on engineering the

assessment $\#ET(G) \geq z$, we only provide implementations of AssessET and TreeAssessET (denoted AF and AT, respectively), equipped with a chain-compression preprocessing strategy. Indeed, the significantly more complex ImprovedAssessET algorithm is very unlikely to be competitive in practice: although it is designed, as explained above, to avoid the $O(m)$ unsuccessful calls in every call of the assessment, we observe that the unsuccessful calls are very few on average in real-world datasets. We use our implementations to provide an extensive experimental evaluation using six real-world datasets from different domains. As a baseline for comparison, we used the implementation of the BEST theorem from [28] that relies, among others, on the *highly optimized* sparse LU decomposition function of the open source Eigen library (v. 3.3.7) [29]. Our experimental results show that both AF and AT: (1) are up to two orders of magnitude faster than the aforementioned implementation of the BEST theorem and on average they are about one order of magnitude faster; (2) perform much fewer than mz iterations in most cases, despite their worst-case bound that is quadratic in m, and scale near-linearly with m in most cases; and (3) bring substantial efficiency benefits in application A1 from Section 1.1, as they speed up the approach of Bernardini et al. [28] by more than 5 times on average. Furthermore, our experiments show that, as expected, AT is several times faster than AF, due to the design of the tree data structure.

Let us remark that a preliminary version of this article *without* Contribution (2), the practical improvement in Contribution (3), and Contribution (4) appeared as [30].

2 Related Work and Applications

As discussed in Introduction, for counting Eulerian trails, the only available tool is the BEST theorem [27], which we discuss below:

Let $A = (a_{uv})$ be the adjacency matrix of G allowing both $a_{uv} > 1$ (multi-edges) and $a_{uu} > 0$ (self-loops). Let $r_u = d^+(u)$ for $u \neq t$, $r_t = d^+(t) + 1$, where $d^+(u)$ denotes the outdegree of u, and the edges are counted with multiplicity. We can apply the BEST theorem using one of its variants for counting node-distinct trails in directed multigraphs [27]:

$$\#ET(G) = (\det L) \cdot \left(\prod_{u \in V} (r_u - 1)! \right) \cdot \left(\prod_{(u,v) \in E} (a_{uv})! \right)^{-1}. \tag{1}$$

where $L = (l_{uv})$ is the $n \times n$ matrix with $l_{uu} = r_u - a_{uu}$ and $l_{uv} = -a_{uv}$.[3] The original BEST theorem states that the number of Eulerian trails of a directed graph can be obtained by multiplying the number of arborescences[4] rooted at any node of the graph (in our case, given by $\det L$) by the number of permutations of the edges outgoing from each node ($\prod_{u \in V}(r_u - 1)!$). In the multigraph version given in Equation (1), the formula is further divided by the number of permutations of multi-edges ($\prod_{(u,v) \in E} a_{uv}!$), in order to only count node-distinct trails. The problem thus reduces in computing the LU decomposition of A using matrix multiplication and then from thereon compute $\det L$.

However, directly applying the formula of Equation (1) is often too costly. The bottleneck is the computation of $\det L$, which can take several hours in typical instances, even with state-of-the-art (sparse) matrix multiplication libraries and other tricks (see [5] for more details).

One may also naturally wonder whether the computation of $\det L$ in Equation (1) can be bypassed. However, this is not the case. Specifically, in this equation, we can single out two factors: the determinant $\det L$ and the ratio of factorials $F = \prod_{u \in V}(r_u - 1)!(\prod_{(u,v) \in E} a_{uv}!)^{-1}$. However, the assessment based on Equation (1) cannot rely on the assumption that $F \geq z$ (which would imply

[3] Note that L as defined in Equation (1) is *not* the Laplacian of graph G (which is always singular): r_t is equal to $d^+(t) + 1$ instead of simply $d^+(t)$ due to the specific problem formulation of [27], to which we refer the reader for more details.

[4] A directed graph is called an *arborescence* if, from a given node u known as the root node, there is exactly one elementary path from u to every other node v.

that $\#ET(G) \geq z$), since we can have $F \ll 1$. As an example, consider a directed multi-cycle with n nodes, $u_1, \ldots, u_n$, each connected to the next (and u_n back to u_1) with k multi-edges: we have only one node-distinct Eulerian trail, so $\#ET(G) = 1$, but there are k^{n-1} arborescences, so $\det L = k^{n-1}$. In particular, we have that $F = \frac{k!(k-1)!^{n-1}}{k!^n} = \frac{1}{k^{n-1}} \ll 1$, for any choice of $s = t$. In addition, enumerating arborescences [31, 32], progressively bounding $\det L$ to check whether $\det L \geq z/F$, might be too costly because the number of arborescences could be exponential in $\#ET(G)$, as in the example.

Last, it is possible to assess $\#ET(G)$ in $O(mz)$ time by enumerating arborescences in a new graph $G' = (V, E')$, where E' is the set of E (i.e., E' is the set of edges obtained from E after neglecting their multiplicities).[5] For each listed arborescence, we obtain at least one Eulerian trail in G. For enumerating the complete set A of arborescences in G' one can use the classic algorithm by Gabow and Myers that takes $O(|E'| + |V| + |A||E'|)$ total time [31], with the idea that we can estimate after how many steps the algorithm would terminate if $\#ET(G)$ was less than z, and give a positive answer as soon as we reach this point rather than waiting for the end of the computation. The problem of such an approach *in practice* is that *one must know the exact constant c* in the $O(|A||E'|)$ term to determine the exact number of elementary operations of the algorithm in the worst case, in order to determine when exactly to terminate.[6]

Since all the aforementioned approaches for assessing $\#ET(G)$ are unsatisfactory, in this article, we embark on a fundamentally different approach that is not based on the BEST theorem or on the enumeration of arborescences. Our approach takes $O(m) = O(nk)$ time in our example from above and can be generalized to more involved graphs.

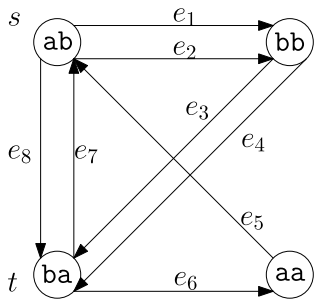
In the following, we discuss the applications of our approach for ETA in some more details. We begin by illustrating an application for defending against reconstruction attacks in strings, which serves as a case study in our article.

(A1) *Defending against Reconstruction Attacks.* Textual information can be strictly confidential [33, 34]; consider, for example, a long fragment of DNA. We would like on the one hand to enable searching its contents for analysis purposes, but on the other hand to prevent its full reconstruction due to privacy concerns. Bernardini et al. [28] proposed to *de-assemble* a string using the notion of **de Bruijn graph (dBG)**. Let $S = S[1]S[2]\ldots S[|S|]$ be a string of length $|S|$ over an alphabet Σ. The *order-d dBG* $G_{S,d}$ of a string S is defined as follows: the set $V_{S,d}$ of nodes is the set of the length-$(d-1)$ substrings of S; and there exists one edge in multiset $E_{S,d}$ from node u to node v for every occurrence in S of the length-d substring equal to $u[1]\,v = u\,v[d-1]$ (i.e., the suffix of length $d-2$ of u equals the prefix of length $d-2$ of v). The directed multigraph in Figure 2 is the dBG of order $d = 3$ of $S = \mathsf{abbaabbaba}$ over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$; e.g., e_1 and e_2 denote that $u[1]\,v = \mathsf{abb} = u\,v[d-1]$ occurs twice in S with $u = \mathsf{ab}$ and $v = \mathsf{bb}$ ($u[1] = \mathsf{a}$ and $v[d-1] = \mathsf{b}$).

The key idea of Bernardini et al. [28] is that *five distinct strings other than S* have *the same* order-3 dBG. These six strings in total are in one-to-one correspondence with the 6 node-distinct Eulerian trails of $G_{S,d}$. Bernardini et al. use the BEST theorem to compute the number $\#ET(G)$ of node-distinct Eulerian trails of $G_{S,d}$ in order to evaluate the level of ambiguity introduced by de-assembling S in fragments of length d: the more Eulerian trails, the more difficult it is for an attacker to uniquely reconstruct S. In this context, with no further assumptions on S, the assessment $\#ET(G) \geq z$ implies that an attacker has probability no more than $\frac{1}{z}$ to infer the correct trail corresponding to S. Bernardini et al. considered more powerful attack models as well, but again our assessment problem is the core computational task for a defense. In fact, the approach of Bernardini et al. aims

[5] Personal communication with Paweł Gawrychowski.

[6] Indeed, if the number of arborescences is less than, say, x, the algorithm would definitely terminate in $O(x|E'|)$ time. If we could determine the exact moment this time was exceeded, we would know that the number of arborescences is more than x, without actually waiting for the algorithm to terminate, and immediately output YES in $O(x|E'|)$ time.



Node-distinct Eulerian trails	Distinct strings
$e_1e_3e_6e_5e_2e_4e_7e_8$	abbaabbaba
$e_1e_3e_7e_2e_4e_6e_5e_8$	abbabbaaba
$e_1e_3e_7e_8e_6e_5e_2e_4$	abbabaabba
$e_8e_6e_5e_1e_3e_7e_2e_4$	abaabbabba
$e_8e_7e_1e_3e_6e_5e_2e_4$	ababbaabba
$e_1e_3e_6e_5e_8e_7e_2e_4$	abbaababba

Fig. 2. The directed de Bruijn multigraph $G_{S,d}$ of order $d = 3$ for string S = abbaabbaba over $\Sigma = \{a, b\}$; its complete set of node-distinct Eulerian trails starting at node s = ab and ending at node t = ba; and the corresponding set of distinct stings.

to find the maximal d for which $\#ET(G) \geq z$, which implies that it uses the BEST theorem a large number of times with different candidate values for the maximal d.

In our experimental evaluation in Section 7, we took this application as a case study. Our ETA-based algorithms substantially speed up the approach in [28], if they are used instead of the BEST theorem. Importantly, our implementations bring substantial time savings, incurring no accuracy loss for pattern matching queries on very long (i.e., up to length-d) substrings, and providing privacy against string reconstruction (i.e., probability of reconstruction of no more than $\frac{1}{z}$).

Further applications are listed below.

(A2) *Investigating Genome Complexity.* Assessment of Eulerian trails applied to dBGs can find interesting uses in bioinformatics as dBGs are a common tool to represent genome sequencing data (cf. [35]). Therein, the (number of) node-distinct Eulerian trails between a given source and target node correspond to (the number of) alternative sequences in between these nodes. In RNA-Seq data, assessment of Eulerian trails between a pair of nodes corresponding to suitable exons is useful for isoforms quantification [7]. In genome assembly (cf. [36]), assessment of Eulerian trails between a pair of nodes is a useful task for understanding the genome's repetitive structure, which is the main obstacle for high-quality assemblies. Although currently, this assessment is performed by *counting* the exact number of Eulerian trails via the BEST theorem [6], an assessment algorithm could provide efficiency benefits in large-scale graphs.

(A3) *Finding Min-Cost Trails.* Consider a large road network, where nodes are junctions and edges are roads. Further consider that a vehicle would like to begin from a starting point (source node), pass from every road of the network once (make an Eulerian trail) for patrolling purposes, and end at some other location (target node). In theory, the vehicle could choose any Eulerian trail of the network. However, taking a specific road (edge) e at a specific order r induces a penalty that is not fixed because, for example, the traffic depends on when a road of the network is traversed [37]. This can be modeled by a cost function $c(e, r)$, which maps every edge e taken as the rth edge of an Eulerian trail to a cost. The computational problem is: Given a directed multigraph G and a cost function c, compute a min-cost Eulerian trail in G. This problem has been studied by Hannenhalli et al. [38], who showed that the decision version of the problem is NP-complete even on graphs with out-degree at most two. Ben-Dor et al. [39] showed that the problem is NP-complete on dBGs as well. Given these negative results, one could resort to the enumeration of all Eulerian trails (if they are not too many) or to the enumeration of some of them (using branch and bound), compute their cost using function c, and choose a min-cost Eulerian trail among them. One could do this

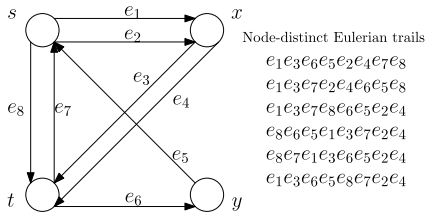


Fig. 3. A directed multigraph and its complete set of node-distinct Eulerian trails. Each trail can have a different cost.

only if the number of the enumerated trails is reasonably small; interestingly, the problem can be solved exactly in polynomial time when $\#ET(G)$ is small. Thus, an algorithm which *first assesses* $\#ET(G) \geq z$, for some integer z, could be applied as preprocessing to solve this challenging problem for such instances. Moreover, as we show in this work, our assessment algorithm can easily be extended to *enumerate Eulerian trails* within the same asymptotic time complexity.

The following table presents a cost function for the directed multigraph in Figure 3. For example $c(e_3, 2) = 7$: taking edge e_3 as the second edge of an Eulerian trail has cost 7.

r	1	2	3	4	5	6	7	8
e_1	6	8	5	2	3	7	7	3
e_2	4	7	4	8	3	3	9	6
e_3	1	7	3	7	4	9	3	1
e_4	9	5	3	6	6	7	5	3
e_5	4	6	2	9	7	8	2	1
e_6	5	5	10	1	8	4	1	6
e_7	2	2	3	6	2	10	6	5
e_8	7	4	5	7	2	1	5	3

The first Eulerian trail, $e_1e_3e_6e_5e_2e_4e_7e_8$, has cost: $6 + 7 + 10 + 9 + 3 + 7 + 6 + 3 = 51$. Thus, by enumerating all node-distinct Eulerian trails and calculating their cost, we can then select the one with a minimum cost. In fact, the Eulerian trail with the minimum cost is the second one, $e_1e_3e_7e_2e_4e_6e_5e_8$, and its cost is 39. This optimal solution is shown in red color.

(A4) *Compressing Undirected Graphs and Networks.* Social networks are often large in size, and their analysis can benefit from storing them in compressed format. The method presented in [40] exploits Eulerian trails to provide a compressed graph format supporting efficient neighbor queries, where the adjacency list of any given node is returned without decompressing the whole graph. The basic idea for the linearization of the graph using Eulerian trails is the following. Suppose first that the graph is Eulerian: any node-distinct Eulerian trail is a feasible linearization of the set of adjacency lists. For example, in the graph of Figure 3, the Eulerian trail $e_1e_3e_6e_5e_2e_4e_7e_8$ corresponds to the node sequence $Z = sxtysxtst$. We observe that all edges are encoded in Z: given any node x, its incoming neighbors in the graph are the nodes preceding x in $Z = \underline{s}\mathbf{x}t\underline{y}\underline{s}\mathbf{x}tst$, which correspond to the incoming edges e_1 and e_2 of x, and its outgoing neighbors are the nodes succeeding x in $Z = s\mathbf{x}\underline{t}ys\mathbf{x}\underline{t}st$, which correspond to the outgoing edges e_3 and e_4 of x. (Hence, multiple edges can be represented using Z). As a result, the adjacency list of x is obtained by retrieving all the occurrences of x in Z, along with the predecessors and successors of x. Suppose that the graph is not Eulerian: in [40–42] it is described how to augment a graph so that it becomes Eulerian with the smallest number of added edges. If the graph is undirected, each edge can be seen as a pair of directed edges, of opposite directions. Consequently, the aforementioned method shows that

compressing a graph amounts to compressing its linearization Z (i.e., Eulerian trail). There are several methods to compress Z so that we can efficiently retrieve the preceding and succeeding elements in Z for any x (e.g., see the textbook [43]). Moreover, any two different Eulerian trails can yield different sequences with different compression ratios. Hence, an enumeration of Eulerian trails (i.e., graph linearizations) would lead to the best compression ratio within this method.

3 Structure and Properties of Directed Eulerian Graphs

Definitions and Notation. Consider a directed graph $G = (V, E)$ with multi-edges (i.e., E is a multi-set) and self-loops, and let $|V| = n$ and $|E| = m$. A *trail* over G is a sequence of adjacent distinct edges. Two trails are *node-distinct* if their *node* sequences are different. A graph is called *Eulerian* if it has an *Eulerian trail*, i.e., a trail that traverses every edge exactly once. We consider *node-distinct* Eulerian trails. The set of node-distinct Eulerian trails of G is denoted by $ET(G)$ and its size is denoted by $\#ET(G)$. We may omit the term "node-distinct" when it is clear from its context.

Given a node $u \in V$, we define its *outdegree* (resp. *indegree*) as the number of edges of the form (u, v) (resp. (v, u)), counting multiplicity and self-loops. We then denote by $\Delta(u)$ the difference $\text{outdegree}(u) - \text{indegree}(u)$. Furthermore, we define the set of *out-neighbors* of u as $N^+(u) = \{v \in V \mid (u, v) \in E\}$. Finally, we use the notation $N^+_C(u) = N^+(u) \cap C$, when referring only to the out-neighbors inside some subgraph C of G.

G is called *strongly connected* if there is a trail in each direction between each pair of the graph nodes. An SCC of G is a strongly connected subgraph of G. G is called *weakly connected* if replacing all of its edges by undirected edges produces a *connected* graph, i.e., there is a trail between every pair of nodes. We recall the characterization of Eulerian graphs:

Remark 1. A directed graph $G = (V, E)$ is *Eulerian with source s and target t*, where $s, t \in V$, if it is weakly connected and either (i) $\Delta(s) = 1$, $\Delta(t) = -1$, and $\Delta(u) = 0\ \forall u \in V \setminus \{s, t\}$; or (ii) $\Delta(u) = 0\ \forall u \in V$. In Case (i), G has an *Eulerian trail from s to t*. In Case (ii), G has an *Eulerian circuit*: an Eulerian trail that starts and ends on $s = t$.

Structure and Properties. The SCCs of a directed Eulerian graph G induce a directed acyclic graph G_{SCC}. Considering this graph, we derive some useful properties, upon which we will heavily rely to design our algorithms for assessing the number of node-distinct Eulerian trails. Let us start with the following crucial lemma.

Lemma 1. *Let G be an Eulerian graph, with SCCs $C_0, \ldots, C_k$, source $s \in C_0$, and target $t \in C_k$. The corresponding G_{SCC} is a* chain graph *of the form $C_0 \to C_1 \to \ldots \to C_k$, where the arrow between C_i and C_{i+1} represents a single edge $(t_i, s_{i+1}) \in E$, called* bridging edge. *Furthermore, each C_i is Eulerian with source s_i and target t_i, where $s_0 = s, t_k = t$.*

Proof. We will first show that the graph forms a chain, and then that each component is Eulerian.

First, given a C_i with $i < k$, we will show that there exists a unique single edge $(t_i, v) \in E$ such that $t_i \in C_i$ and $v \notin C_i$. In other words, there is a unique edge directed from a node of C_i to a node of a different SCC. By contradiction, let there be two: $e = (t_i, v)$ and $e' = (t'_i, v')$ with $t_i, t'_i \in C_i$, not necessarily different, and $v, v' \notin C_i$, again not necessarily different. We know by hypothesis that there exists an Eulerian trail $\mathcal{P}$ in G from s to t. This trail will need to traverse both e, e'. Wlog, let it traverse e first; then we must have a trail from v back to t'_i to be able to traverse e'. This implies v is in the same SCC as t_i, i.e., $v \in C_i$: contradiction. From now onwards we will refer to this t_i as the unique *target* or *exit point* of C_i. Consider now C_k. In this case, we cannot have any outgoing edges (u, v) with $u \in C_k, v \notin C_k$. This is because all the Eulerian trails of G end in t, thus any trail $u \to v \rightsquigarrow t$ would imply $v \in C_k$, which is a contradiction. Let then $t_k = t$.

The "opposite" condition on the incoming edges can be easily proved symmetrically. That is, for each C_i with $i > 0$ there exists a unique incoming edge $(v, s_i) \in E$, with $s_i \in C_i$ called the unique *source* or *entry point* of C_i, and $v \notin C_i$. As for C_0, we consider $s_0 = s$, and again we can prove that there are no incoming edges whatsoever.

At this point we have proved that each SCC C_i with $i \neq 0, k$ has exactly one entry point and one exit point; while C_0 has one exit point and no entry, and C_k has one entry point and no exit. This immediately implies that the graph has a chain shape: $C_0 \to C_1 \to \cdots \to C_k$.

Consider now a given C_i; we will show that it is an Eulerian graph with source s_i and target t_i, as claimed. We consider all possible cases, recalling that (strong) connectivity is immediate, as we are working on an SCC, and that $\Delta(v) = 0$ for all $v \in G \setminus \{s, t\}$. In what follows, let $\Delta_i(v)$ be $\Delta(v)$ defined over the induced subgraph $G[C_i]$.

(1) $s, t \in C_i \Rightarrow G = C_i$ is Eulerian.
(2) $s \in C_i = C_0$, $t \notin C_0$. Note that $\Delta(s) = 1$ and $\Delta(t) = -1$. We know that in C_i we have $s_i = s$, and we have a unique exit point t_i. This leads us to two subcases:
   − $s = t_i \Rightarrow \Delta_i(s) = \Delta(s) - 1 = 0$, and $\Delta_i(u) = \Delta(u) = 0$ for all $u \in C_i \setminus \{s\}$. That is, we are in case (ii) of Remark 1.
   − $s \neq t_i \Rightarrow \Delta_i(s) = \Delta(s) = 1$, $\Delta_i(t_i) = \Delta(t_i) - 1 = -1$ and $\Delta_i(u) = \Delta(u) = 0$ for all $u \in C_i$, $u \neq s, t_i$. We thus satisfy case (i) of Remark 1.
(3) Case $t \in C_i = C_k$, $s \notin C_k$ is symmetric to Item 2.
(4) $s, t \notin C_i \Rightarrow$ consider the corresponding unique entry and exit points s_i, t_i. We show that these are the source and target, looking at Δ_i and considering two subcases:
   − $s_i \neq t_i \Rightarrow$ We have $\Delta_i(s_i) = \Delta(s_i) + 1 = 1$, $\Delta_i(t_i) = \Delta(t_i) - 1 = -1$ and $\Delta_i(u) = \Delta(u) = 0$ for all $u \in C_i \setminus \{s_i, t_i\}$, i.e., case (i) of Remark 1.
   − $s_i = t_i \Rightarrow \Delta_i(s_i) = \Delta_i(t_i) = \Delta(s) + 1 - 1 = 0$, and $\Delta_i(u) = \Delta(u) = 0$ for all $u \in C_i \setminus \{s_i, t_i\}$, i.e., case (ii) of Remark 1.

□

It follows from Lemma 1 that every trail from s to t must traverse all edges of $C_0, \ldots, C_i$ before crossing the bridging edge (t_i, s_{i+1}). As a consequence, we obtain the following.

Corollary 1. *Let G be an Eulerian graph with SCCs $C_0, \ldots, C_k$. We have $ET(G) = \prod_{i=0}^{k} ET(C_i)$, where $\prod$ denotes the cartesian product. Thus, the number of trails of G is the product of the number of trails of its SCCs.*

We can thus focus on an individual SCC or, equivalently, assume, wlog, that the Eulerian graph is strongly connected. The following lemma forms the basis of our technique.

Lemma 2. *Let C be a strongly connected Eulerian graph with source s and target t. For every edge (s, u), there is an Eulerian trail of C whose first two traversed nodes are s and u. Moreover, the residual graph $C \setminus (s, u)$ remains Eulerian with new source u.*

Proof. The proof is by case analysis. In what follows, let $e = (s, u)$ and $C' = C \setminus \{e\}$, and let Δ' be the difference between the indegrees and outdegrees of nodes in C'. Note that the strong connectivity of C immediately implies the connectivity of C'.

Suppose that $s = t$: we are in case (ii) of Remark 1. Then, $\Delta'(s) = \Delta(s) - 1 = -1$, while on the other hand $\Delta'(u) = \Delta(u) + 1 = 1$. All other nodes v remain with $\Delta'(v) = 0$. This proves that C' falls under case (i) of Remark 1.

Suppose that $s \neq t$: we are in case (i) of Remark 1. Here we consider three cases.

Case (a): $u \neq t, s$. We have $\Delta'(s) = \Delta(s) - 1 = 0$, $\Delta'(t) = \Delta(t) = -1$ and $\Delta'(u) = \Delta(u) + 1 = 1$. All other nodes stay unchanged with $\Delta' = 0$. This means that C' falls under case (i) of the remark.

Case (b): $u = t$. We have $\Delta'(s) = \Delta(s) - 1 = 0$, $\Delta'(u) = \Delta(u) + 1 = 0$ and $\Delta'(v) = \Delta(v) = 0$ for all other $v \in V$. Furthermore, C' is connected. These two hypotheses directly imply that there is an Eulerian circuit, thus the graph is actually strongly connected.

Case (c) $u = s$: That is, e is a self-loop. In this case, $\Delta'(v) = \Delta(v)$ for all $v \in V$, and the graph remains Eulerian. □

Corollary 2. *Let C_i be any SCC of an Eulerian graph with source s_i. Then*:

$$ET(C_i) = \bigcup_{u \in N^+_{C_i}(s_i)} (s_i, u) \cdot ET(C_i \setminus (s_i, u)),$$

i.e., the Eulerian trails of C_i are given by concatenating each possible start of the trail (s_i, u) with all its possible continuations (the trails in $ET(C_i \setminus (s_i, u))$). Thus the number of trails of C_i is the sum of the number of trails of the subgraphs with edges (s_i, u) removed, for every $u \in N^+_{C_i}(s_i)$ distinct out-neighbor of s_i in C_i, with u as the new source.

Proof. This follows from Lemma 2 applied to the SCCs: we know that each distinct out-neighbor of s_i leads to at least one trail; furthermore, no two of these trails can be equal since they begin with distinct edges. Lastly, all trails are accounted for, since we consider every trail starting from every distinct out-neighbor of s_i, and s_i is the source of C_i. □

Note the subtle point in the statement of Corollary 2, where we use $N^+_{C_i}(s_i)$ instead of $N^+(s_i)$: if the latter two differ, it is because s_i has an outgoing bridging edge, and this should be traversed *after* all other edges in C_i.

4 Assessment Algorithm for #ET(G)

We present AssessET, a simple but non-trivial algorithm for assessing the number of node-distinct Eulerian trails on a given directed graph, which will be refined in Section 6. AssessET takes the following input parameters: (i) a weakly connected Eulerian graph $G = (V, E)$ with source s and target t; (ii) a positive integer threshold z, and (iii) a function $lb(\cdot)$, which outputs a lower bound on the number of the node-distinct Eulerian trails in G. To achieve the desired complexity, $lb(\cdot)$ must be computable in $O(m)$ time and $lb(\cdot) \geq 1$ must hold.[7] We will prove the following result.

Proposition 1. *Given graph G, nodes s and t, integer z, and $lb(\cdot)$, AssessET assesses $\#ET(G) \geq z$ in $O(m^2 \cdot \min\{z, \#ET(G)\})$ time using $O(mz)$ space.*

Main Idea. Let $C_0, \ldots, C_k$ be the set of SCCs of an Eulerian graph G as illustrated in Lemma 1. AssessET exploits Corollary 1, Lemma 2, and Corollary 2, to provide a lower bound on the number of node-distinct Eulerian trails of graph G, denoted by $lb_{ET}(G)$, where $lb_{ET}(G) \leq \#ET(G)$. Initially, we set $lb_{ET}(G) = \prod_{i=0}^{k} lb(C_i)$, based on the product of the lower bounds for the number of node-distinct Eulerian trails of the SCCs of G by Corollary 1. Then $lb_{ET}(G)$ is progressively refined by considering any arbitrarily chosen component, say C_i, and in turn replacing its lower bound $lb(C_i)$ with a new lower bound $lb_{ET}(C_i)$ that exploits Lemma 2 and its Corollary 2. That is, we remove each different outgoing edge from the source s_i of C_i, and after computing the $lb(\cdot)$ function on all of the resulting graphs, we sum these lower bounds to obtain $lb_{ET}(C_i)$, and update $lb_{ET}(G)$. We proceed in this way until either $lb_{ET}(G) \geq z$, or we compute the actual number of trails: $lb_{ET}(G) = \#ET(G)$.

The requirements for the lower bound function are trivially satisfied by the constant function $lb(\cdot) \equiv 1$. However, we use a better lower bound given by Lemma 3.

[7] Observe that, when G has an Eulerian circuit, any node can be a source, and the choice of source can change the number of node-distinct Eulerian circuits of G. We assume every graph generated keeps track of the correct source.

Lemma 3. *For any Eulerian graph G, the function:*

$$lb(G) = 1 + \sum_{v \in V(G): |N_G^+(v)| \geq 3} (|N_G^+(v)| - 2). \tag{2}$$

is a lower bound for the number $\#ET(G)$ of node-distinct Eulerian trails of G.

Proof. Let us consider $T \in ET(G)$, and $v \in V(G)$ with $k \geq 3$ distinct out-neighbors. Consider the prefix T' of T up to the first edge reaching node v, and let $G' = G \setminus T'$ (i.e., G' is the graph obtained from G after removing the subgraph T'). G' is Eulerian with source v, and each $S \in ET(G')$ contributes to creating a distinct $T'S \in ET(G)$. Of the k out-neighbors of v, one was the continuation of T. As for the others, at most one can be a bridging edge, with the other endpoint belonging to a different SCC of G'. This is because of the chain structure of the SCCs given in Lemma 1. The remaining $k-2$ neighbors, by Lemma 2, correspond to distinct Eulerian trails of G'. These trails, when appended to T', generate distinct Eulerian trails of G, all different from T.

Repeating the above considerations for every node v with $|N_G^+(v)| \geq 3$, and accounting for the trail T we were basing the reasoning on, we obtain:

$$\#ET(G) \geq 1 + \sum_{v \in V(G): |N_G^+(v)| \geq 3} (|N_G^+(v)| - 2).$$

□

Function ComputeSCC. Our algorithm relies on a function ComputeSCC(G), which computes the SCCs of a given input graph G. This function only outputs the non-trivial components (i.e., comprised of multiple nodes), and it requires $O(m)$ time to achieve this (specifically, we use [44]).

Frontier Data Structure. In order to efficiently explore the different SCCs as discussed above, we introduce the *Frontier Data Structure*, denoted by $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\}$, representing the frontier of the recursive tree we are *implicitly* constructing when traversing a component. At any moment of the computation, an element $f_j \in \mathcal{F}$ is a tuple $\langle C_0^j, \ldots, C_{h_j}^j \rangle$, where $C_0, \ldots, C_{h_j}$ are the non-trivial SCCs of some Eulerian subgraph $G_j \subset G$. A component is considered *trivial* if it is comprised of a single node. A trivial component is omitted because it contributes to the product in Equation (3) below by a factor of one. Different G_j's are obtained from G by removing different edges that are outgoing from the source of a component, as per Lemma 2; thus G_j differs from any other G_l by at least one removed edge. In this way, each element of the frontier represents at least one node-distinct Eulerian trail of G. Furthermore, our data structure $\mathcal{F}$ retains an important invariant: at any moment, the elements of $\mathcal{F}$ are the SCC decompositions of the subgraphs which realize the current bound. That is,

$$lb_{ET}(G) = \sum_{j=1}^{|\mathcal{F}|} lb_{ET}(f_j) = \sum_{j=1}^{|\mathcal{F}|} \prod_{i=0}^{h_j} lb(C_i^j). \tag{3}$$

Each component $f_j[i] = C_i^j$, with source s_i^j and target t_i^j, is represented in $f \in \mathcal{F}$ as a tuple of the form $(V[C_i^j], E[C_i^j], s_i^j, t_i^j, lb(C_i^j))$. In what follows, we consider $\mathcal{F}$ implemented as a *stack*: both removing and inserting elements requires $O(1)$ time with pop and push operations. Performing these operations also modifies the size of $\mathcal{F}$, which is accounted for. We can thus answer whether the stack is empty in $O(1)$ time.

Algorithm AssessET. The algorithm maintains a running bound lb_{ET}, induced by the components currently forming the elements of the stack, according to Equation (3), where lb_{ET} is the current value of $lb_{ET}(G)$. We proceed as follows (see also Algorithm 1):

Algorithm 1: AssessET

```
1: procedure AssessET(G = (V, E), z = O(poly(|E|), lb(·))
2:     C_0, ..., C_k ← ComputeSCC(G)                    ▷ Only considers non-trivial SCCs
3:     f ← ⟨C_0, ..., C_k⟩
4:     if f is empty then lb_ET ← 1
5:     else Stack.push(f)                               ▷ Initialization
6:         lb_ET ← ∏_{j=0}^{k} lb(C_j)
7:     while lb_ET < z do
8:         if Stack.IsEmpty() then Output NO
9:         f ← Stack.pop()
10:        lb_ET ← lb_ET − lb_ET(f)                     ▷ lb_ET(f) = ∏_{i=1}^{|f|} lb(f[i])
11:        Choose any i; let C_i = f[i] and s_i be its source   ▷ f[i] is the i-th SCC of f
12:        Remove C_i from f
13:        for all u ∈ N^+_{C_i}(s_i) do
14:            C ← ComputeSCC(C_i \ (s_i, u))
15:            if f · C is not empty then               ▷ f · C: f with each SCC of C appended
16:                Stack.push(f · C)
17:                lb_ET ← lb_ET + lb_ET(f · C)
18:            else lb_ET ← lb_ET + 1
19:    Output YES
```

(1) Compute the (non-trivial) SCCs of graph G. If there is none, we only have one trail, and $lb_{ET} = 1$. Otherwise, we initialize the stack with the tuple $\langle C_0, \ldots, C_k \rangle$ of these SCCs, and also initialize the bound accordingly setting $lb_{ET} \leftarrow \prod_{j=0}^{k} lb(C_j)$.
(2) While $lb_{ET} < z$, we perform the following:
   (a) If the stack is empty, we output NO. Since non-trivial components are never added into the stack, the stack is empty if and only if $lb_{ET} = \#ET(G)$ and $lb_{ET} < z$.
   (b) Otherwise, we pop an element f from the stack, and remove its contribution from the current bound: $lb_{ET} \leftarrow lb_{ET} - lb_{ET}(f)$, where $lb_{ET}(f) = \prod_{i=1}^{|f|} lb(f[i])$.
   (c) We pick an arbitrary component $C_i = f[i]$ of tuple f, and let s_i be its source. We remove the component from f.
   (d) For all distinct out-neighbors $u \in N^+_{C_i}(s_i)$:
      (i) We compute the SCCs $\mathcal{C}$ of C_i with edge (s_i, u) removed.
      (ii) If f with the added new components $\mathcal{C}$ (i.e., $f \cdot \mathcal{C}$) is nonempty,[8] we add it into the stack and increase the running bound accordingly as $lb_{ET} \leftarrow lb_{ET} + lb_{ET}(f \cdot \mathcal{C})$. If $f \cdot \mathcal{C}$ is empty, it corresponds to a single Eulerian trail, so we increase the bound lb_{ET} by one.
(3) If we exit from the while loop in Item 2, then $lb_{ET} \geq z$ and we output YES.

The correctness of AssessET follows from Corollary 1, Lemma 2, and Corollary 3. With time and space complexity of AssessET, we complete the proof of Proposition 1.

Complexity Analysis. We aim at showing that growing Stack to size S takes $O(m^2 S)$ time.[9] As $lb(\cdot) \geq 1$ for each element on Stack, and since each tuple represents node-distinct Eulerian trails, this certifies $\#ET(G) \geq S$. We thus stop when $S = \min\{z, \#ET(G)\}$ after $O(m^2 S)$ time. Let us stress that a good $lb(\cdot)$ function can help us stop the algorithm even when S is significantly smaller than z. In fact, we experimentally show that this is the case with the lb function in Equation (2) we used.

[8] The tuple is nonempty if it contains at least one element. Since trivial SCCs are omitted from the tuples, this is equivalent to saying that the corresponding graph has at least one non-trivial SCC.
[9] This includes the case where we implicitly represent an empty tuple by increasing the bound lb_{ET} (Line 18).

Let us remember that each PUSH and POP operation on STACK requires $O(1)$ time, and that our $lb(\cdot)$ function can always be computed in time linear in m. Finally, remember that also COMPUTESCC requires $O(m)$ time.

We are now ready for a detailed analysis. By the discussion above, every line up to Line 6 requires $O(m)$ time. The while loop in Line 7 iterates until our bound lb_{ET} reaches z, and every operation up to Line 13 takes either $O(1)$ or $O(m)$ time. Once we get into the for loop in Line 13, we go over all (distinct) out-neighbors of node s_i. We apply COMPUTESCC to the current component with the chosen edge removed, which takes $O(m)$ time. Then we perform another set of $O(1)$- or $O(m)$-time operations, until we exit the loops. Thus, the for loop of Line 13 requires $O(m \cdot |N^+_{C_i}(s_i)|)$ time, and generates $|N^+_{C_i}(s_i)|$ new elements, meaning that the size of STACK increases by $|N^+_{C_i}(s_i)| - 1$ (as we popped one).

Whenever $|N^+_{C_i}(s_i)| > 1$, we are increasing the size of STACK, paying $O(m)$ time for each new element. However, if $|N^+_{C_i}(s_i)| = 1$, we have popped f, and spent $O(m)$ time to put back a single element f' on STACK, without increasing its size. In this case, we remark that f' corresponds to f minus at least one edge ((s_i, u) in Line 14). Thus, an element containing h edges may be processed in this way, which takes $O(m)$ time and does not increase the size of STACK, only $O(h)$ times. In turn, since $h \leq m$, the total time arising from this bad case on a STACK with S elements is $O(m^2 S)$, which implies our claimed bound.

As for the space complexity, our frontier data structure, represented by STACK, is comprised of elements of size $O(m)$ each. Since each element corresponds to at least one node-distinct Eulerian trail, we never have more than z elements stored. Thus, STACK requires $O(mz)$ space, and Proposition 1 is proved.

We finally remark that this complexity is tight for this algorithm: when $lb(\cdot)$ always returns 1, the contribution of each tuple is exactly 1, as it is a product of components whose bound is 1. Therefore, the lower bound corresponds to the number of tuples generated so far (including trivial ones omitted from the data structure), so ASSESSET needs to generate exactly z tuples. As stated in the analysis above, this requires $O(m^2 z)$ time.

4.1 Enumeration

While the algorithm focuses on deciding whether there are at least z trails, it is relevant to show how it can easily keep track of such trails and explicitly list them. Looking at the way the algorithm processes tuples of SCCs, we simply need to preserve the sources in these: when e.g., C_i is processed in the example in Figure 4, this corresponds to generating 3 branches, all of which traversed an edge outgoing from the source s_i. We can thus keep track of s_i by turning $\langle \ldots, C_i, \ldots \rangle$ into $\langle \ldots, s, C_i^1, \ldots \rangle \langle \ldots, s, C_i^2, \ldots \rangle \langle \ldots, s, C_i^3, \ldots \rangle$. In this way, whenever a tuple has been completely processed (i.e., all its elements correspond to single nodes), the tuple itself will be the node-sequence of the corresponding Eulerian trail; as for tuples still containing SCCs, it is sufficient to replace each one with one arbitrary Eulerian trail of the SCC.

5 Improved Tree-Based Data Structure

In this section, we will present some practical improvements of algorithm ASSESSET. Specifically, we present a modified and improved version of ASSESSET, called TREEASSESSET, with the same worst-case running time, but with a better performance in practice, as we will see in Section 7.

The idea behind TREEASSESSET is to represent the bound formula (copied from Equation (3)):

$$lb_{ET}(G) = \sum_{j=1}^{|\mathcal{F}|} lb_{ET}(f_j) = \sum_{j=1}^{|\mathcal{F}|} \prod_{i=0}^{h_j} lb(C_i^j),$$

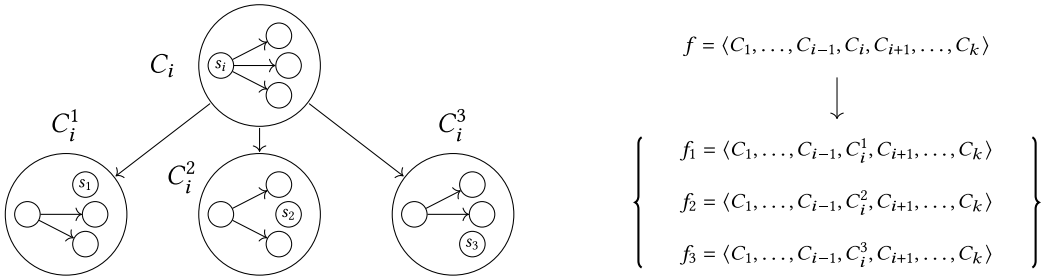


Fig. 4. Left: decomposition of the Eulerian trail of an SCC C_i according to the distinct neighbors of its source s_i (as per Lemma 2), where each C_i^j is strongly connected and has source s_j. Right: corresponding effect during AssessET in an element of the frontier $f \in \mathcal{F}$.

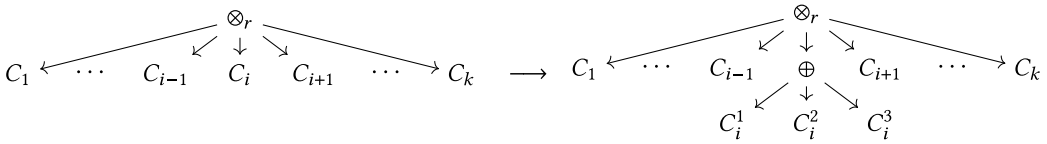


Fig. 5. Left: Initialization of the tree $\mathcal{T}_{ET}$. Right: After one step of TreeAssessET, corresponding to the decomposition shown in Figure 4. The tree node for C_i is replaced with a $\oplus$ node, which has the three new components as children. The lower bound is $lb_{ET} = lb_{\mathcal{T}}(\otimes_r) = (lb(C_i^1) + lb(C_i^2) + lb(C_i^3)) \times \prod_{j \neq i} lb(C_j)$.

in a tree-like structure, which we will denote $\mathcal{T}_{ET}$.

The tree $\mathcal{T}_{ET}$ will represent the bound in a way akin to a parse tree. Indeed, internal nodes will be of one of two types, $\otimes$ or $\oplus$, respectively corresponding to product and sum operations. The leaf nodes of the tree will be in a 1:1 correspondence with the set of components $\{C_i^j \mid j = 1, \ldots, |\mathcal{F}|;\ i = 0, \ldots, h_j\}$ at some moment of the computation of AssessET. Each node U of the tree will have an assigned bound value $lb_{\mathcal{T}}(U)$, which expresses the contribution of its rooted subtree to the global running bound lb_{ET}, and is defined recursively as follows:

—If node U is a leaf, then it corresponds to some component C_i^j. Then, $lb_{\mathcal{T}}(U) = lb(C_i^j)$.
—If node U is an internal node of type $\otimes$ with children $V_1, \ldots, V_k$, then $lb_{\mathcal{T}}(U) = lb_{\mathcal{T}}(V_1) \times lb_{\mathcal{T}}(V_2) \times \cdots \times lb_{\mathcal{T}}(V_k)$.
—If node U is an internal node of type $\oplus$ with children $V_1, \ldots, V_k$, then $lb_{\mathcal{T}}(U) = lb_{\mathcal{T}}(V_1) + lb_{\mathcal{T}}(V_2) + \cdots + lb_{\mathcal{T}}(V_k)$.

Along the computation, we wish to preserve a tree structure such that the lb_{ET} value for the root of tree $\mathcal{T}_{ET}$ is always equal to the current running bound lb_{ET} (Equation (3)).

The Algorithm. In AssessET, the frontier data structure implicitly represents the current frontier of the binary partition procedure. In algorithm TreeAssessET the tree $\mathcal{T}_{ET}$ will substitute the frontier data structure as follows:

(1) Let $C_1, \ldots, C_k$ be the SCCs of graph G. We start with $\mathcal{T}_{ET}$ equal to a single $\otimes$ node, the root $\otimes_r$, with a (leaf) child for each C_i (see Figure 5, left). This is equivalent to initializing $\mathcal{F} = \{\langle C_1, \ldots, C_k \rangle\}$. By definition, $lb_{ET} = lb_{\mathcal{T}}(\otimes_r) = \prod_{i=1}^{k} lb(C_i)$ is initialized correctly.
(2) Similarly, every time we compute the SCCs $B_1, \ldots, B_h$ of a (leaf) component C, we replace the node for C with a node of type $\otimes$, with $B_1, \ldots, B_h$ as children. Note that $B_1, \ldots, B_h$ have now become leaves of the tree. This is implicitly handled in AssessET, as $lb_{ET}(f)$ for $f \in \mathcal{F}$ is defined as the product of the bound of the single components of f.

(3) Recall that AssessET considers an arbitrary component C_i of one of the tuples $f = \langle C_1, \ldots, C_{h_f} \rangle$ and performs the following: it goes over all neighbors u of the source s_i of C_i (Line 13 and considers the tuples f_u obtained from f by substituting C_i with the SCCs resulting from removing edge (s_i, u) (Line 16). The contribution of f to the global bound will be replaced by the sum of the contributions of all such f_us. This structure will be represented in the tree version using $\oplus$ nodes: we will pick one leaf of $\mathcal{T}_{ET}$ (corresponding to an SCC of some instance), and we will substitute it with a $\oplus$ node, with a child for each sub-instance arising when traversing different neighbors of the source. Note that such children can in turn become $\otimes$ nodes, if they are not strongly connected themselves. Figure 5 shows an example of this, corresponding to the step of AssessET given in Figure 4.

Remark 2. Contrary to the components in the frontier data structure, the leaves of tree $\mathcal{T}_{ET}$ all correspond to different components. Indeed, they start as distinct SCCs of the original graph. Then, every time we replace a leaf L with a subtree, the newly created leaves are all disjoint subgraphs of L.

Instead, when using the frontier data structure at any one step of the algorithm, we redundantly add to the frontier one new vector f_u for each neighbor u of the source of the chosen C_i. In this scenario, the different vectors f_u all contain the SCCs different from C_i $(C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_{h_f})$ as they were in f; they only differ in what they replaced component C_i with. Therefore, in the frontier data structure, the same component C_i may appear as a component in several $f \in \mathcal{F}$. This allows us to avoid repeated computation, as the same component is never processed more than once, reducing the running time in practice (see Section 7).

The pseudocode for the modified algorithm TreeAssessET is shown in Algorithm 2. In the algorithm, we substitute $\mathcal{F}$ with $\mathcal{T}_{ET}$, proceeding as described above. Specifically, the tree is initialized as $\otimes$ node, with the SCCs of the original graph G as children, and only leaves (Line 3). This mirrors the initialization of the frontier $\mathcal{F}$ as the tuple of the SCCs. At each step, we pick a leaf L of $\mathcal{T}_{ET}$, and we expand the corresponding component. This has the effect of replacing L with a $\oplus$ node (Line 11), with a child for each neighbor u of the source of L. Each of these children might also further become $\otimes$ nodes, if their SCCs have become non-trivial (Lines 19–26). In this case, a new leaf tree node is created for each different SCC, associated with the corresponding bound (Lines 21–25). On the other hand, if one of the children has components that are all trivial, it becomes a trivial leaf (Lines 15–17). In any case, the correct contribution is added to $lb_{\mathcal{T}}(\oplus_L)$, at Lines 26 and 17, respectively. For simplicity, we keep a list $\mathcal{L}$ of non-trivial leaves, from which we extract the next leaf to expand at Line 9.

Remark 3. To ensure the correctness of the bound update and that no leaf is of type $\otimes$ or $\oplus$, when function ComputeSCC returns all trivial components, we instead create a dummy trivial leaf with $lb_{ET} = 1$ (Line 16). This leaf is not added to the list $\mathcal{L}$ of valid leaves.

Correctness. We prove correctness by showing that performing one step of TreeAssessET is equivalent to performing several steps of AssessET at once. First, note that the tree is initialized as a root $\otimes$ node, with the SCCs of the original graph $C_1, \ldots, C_k$ as children. As mentioned before, this is equivalent to the initialization of the frontier as $\mathcal{F} = \{\langle C_1, \ldots, C_k \rangle\}$.

When performing a step of TreeAssessET for some chosen component C_i, we take the corresponding leaf, and substitute it with a $\oplus$ node, splitting over the distinct neighbors of the source. Since component C_i may appear in several tuples of $\mathcal{F}$ (see Remark 2), this step is actually equivalent to performing that same expansion step in AssessET for *all* tuples $f \in \mathcal{F}$ which contain C_i

Algorithm 2: TreeAssessET

procedure TreeAssessET($G = (V, E)$, $z = O(\text{poly}(|E|), lb(\cdot))$
  $C_0, \ldots, C_k \leftarrow$ ComputeSCC(G)
  Initialize $\mathcal{T}_{ET}$ as a $\otimes$ root node $\otimes_r$, with children $C_0, \ldots, C_k$
  Initialize list $\mathcal{L}$ with non-trivial C_js
  if $\mathcal{L}$ is empty **then** $lb_{ET} = 1$
  else $lb_{ET} = \prod_{j=0}^{k} lb(C_j)$
  while $lb_{ET} < z$ **do**
    if $\mathcal{L}$ is empty **then Output NO**
    Pick an element L from $\mathcal{L}$, let s_L be its source
    $oldbound = lb_{\mathcal{T}}(L)$
    Substitute node for L in $\mathcal{T}_{ET}$ with a $\oplus$ node, denoted $\oplus_L$
    Intialize $lb_{\mathcal{T}}(\oplus_L) = 0$
    for all $u \in N_L^+(s_L)$ **do**
      $C \leftarrow$ ComputeSCC($L \setminus (s_L, u)$)
      if C is empty **then** ▷ All components are trivial
        Add a trivial leaf with $lb_{\mathcal{T}} = 1$ as a child of $\oplus_L$
        $lb_{\mathcal{T}}(\oplus_L) = lb_{\mathcal{T}}(\oplus_L) + 1$
      else
        Create a new node $\otimes_C$ of type $\otimes$ as a child of $\oplus_L$
        $lb_{\mathcal{T}}(\otimes_C) = 1$
        for all $C \in C$ **do**
          Add C as a leaf child of $\otimes_C$ with $lb_{\mathcal{T}}(C) = lb(C)$
          Let s_C, t_C source and target of C
          Add C to list $\mathcal{L}$
          $lb_{\mathcal{T}}(\otimes_C) = lb_{\mathcal{T}}(\otimes_C) \times lb(C)$
        $lb_{\mathcal{T}}(\oplus_L) = lb_{\mathcal{T}}(\oplus_L) + lb_{\mathcal{T}}(\otimes_C)$ ▷ Add contribution to $\oplus_L$
    UpdateTreeBound($oldBound$, $lb_{\mathcal{T}}(\oplus_L)$, $parent(\oplus_L)$)
    $lb_{ET} = lb_{\mathcal{T}}(\otimes_r)$
  Output YES

as a component, simultaneously. Consequently, by setting $lb_{ET} = lb_{\mathcal{T}}(\otimes_r)$ after one such step, we obtain the same running lower bound as after the corresponding multiple steps in AssessET.

Complexity. We have seen how one step of TreeAssessET corresponds to performing several simultaneous steps of AssessET, ensuring correctness of the algorithm. On the other hand, in this new framework updating the running global bound $lb_{ET}$is no longer constant-time. We show how, when we substitute a leaf with a subtree, we can propagate the bound update towards the root in a procedure called UpdateTreeBound that takes time proportional to the depth of the updated leaf (see Algorithm 3).

More specifically, the function takes as input a non-leaf node $\odot$ of $\mathcal{T}_{ET}$, and two values, *oldBound* and *newBound*. Let $V_1, \ldots, V_k$ be the children of $\odot$; the function updates the $lb_{\mathcal{T}}$ value of $\odot$ assuming that one of its children (wlog V_1) had its $lb_{\mathcal{T}}$ value changed from *oldBound* to *newBound*. It does so recursively:

—If $\odot$ is of type $\otimes$, then by definition $lb_{\mathcal{T}}(\odot) = lb_{\mathcal{T}}(V_1) \times lb_{\mathcal{T}}(V_2) \times \ldots \times lb_{\mathcal{T}}(V_k)$. Therefore, if $lb_{\mathcal{T}}(V_1)$ changed from *oldBound* to *newBound*, then $lb_{\mathcal{T}}(\odot)$ must change from $oldBound \times lb_{\mathcal{T}}(V_2) \times \ldots \times lb_{\mathcal{T}}(V_k)$ to $newBound \times lb_{\mathcal{T}}(V_2) \times \ldots \times lb_{\mathcal{T}}(V_k)$. Equivalently, $lb_{\mathcal{T}}(\odot)$ is updated to $lb_{\mathcal{T}}(\odot)/oldBound \times newBound$ (Line 4).

—If $\odot$ is of type $\oplus$, then $lb_{\mathcal{T}}(\odot) = lb_{\mathcal{T}}(V_1) + lb_{\mathcal{T}}(V_2) + \ldots + lb_{\mathcal{T}}(V_k)$, and consequently it must change from $oldBound + lb_{\mathcal{T}}(V_2) + \ldots + lb_{\mathcal{T}}(V_k)$ to $newBound + lb_{\mathcal{T}}(V_2) + \ldots + lb_{\mathcal{T}}(V_k)$. Therefore, $lb_{\mathcal{T}}(\odot)$ becomes $lb_{\mathcal{T}}(\odot) - oldBound + newBound$ (Line 7).

Algorithm 3: UpdateTreeBound

```
1: procedure UpdateTreeBound(oldBound, newBound, ⊙)
2:     old_⊙ = lb_T(⊙)
3:     if ⊙ is of type ⊗ then
4:         lb_T(⊙) = lb_T(⊙)/oldbound × newbound
5:         if ⊙ is the root ⊗_r then return
6:     else                                  ▷ ⊙ is necessarily of type ⊕, and cannot be root
7:         lb_T(⊙) = lb_T(⊙) − oldbound + newbound
       UpdateTreeBound(old_⊙, lb_T(⊙), parent(⊙))
8:     return
```

In any of the two cases, if we are not at the root, the function is recursively called on the parent of node $\odot$, passing both the old and new value of $lb_{\mathcal{T}}(\odot)$ (Line 7).

The levels of the tree alternate nodes of type $\otimes$ and of type $\oplus$, and each $\oplus$ step requires the loss of an edge. Thus, the depth of a leaf is bounded by $2|E|$. Overall, this yields the same total asymptotic time complexity of AssessET, as the cost of each step is still $O(|E|)$, and, at the worst case, we perform the same exact computation.

As for the space complexity of TreeAssessET, in the worst case it is the same as AssessET. First, note that, when creating $\otimes$ nodes, the same space is retained as before: indeed, a component is replaced with several ones, of the same total size. Consider now $\oplus$ nodes. On the other hand, every time a $\oplus$ node is created for some component C, we substitute component C with $x \leq |V|$ new components, *each* of size $O(|E(C)|) \subseteq O(|E|)$, and since each child C' has $lb(C') \geq 1$, the $\oplus$ node will contribute for at least x trails, i.e., in the worst case we pay an additional $O(|E(C)|)$ space for each new trail (if a $\oplus$ node has only one child, and thus the bound does not increase, then the space remains the same). Therefore, in the worst case, the algorithm takes up $O(|E|)$ space for each of the trails found, which are at most z, for a total space usage of $O(|E|\, z)$.

We have thus arrived at the following result.

Proposition 2. *Given graph G, nodes s and t, integer z, and $lb(\cdot)$, TreeAssessET assesses $\#ET(G) \geq z$ in $O(m^2 \cdot \min\{z, \#ET(G)\})$ time using $O(mz)$ space.*

5.1 Enumeration

As above, this algorithm is easily adapted for explicit enumeration of the solutions: whenever traversing a node v of the graph, we can save it in the $\oplus$ node generated. Whenever a leaf l of the tree is just a single node or a trivial SCC, we can then reconstruct the corresponding Eulerian trail from its root-to-leaf path, starting from the root: each $\oplus$ node corresponds to alternative choices, so we add the stored graph node and only consider the prosecution of the root-to-leaf path leading to l; on the other hand, the children of $\otimes$ nodes represent a sequence of SCCs, so we need to consider all of them (in the same order given by the decomposition algorithm). As above, leaves which contain a non-trivial SCC can be replaced by an arbitrary Eulerian trail of said SCC. Finally, observe that due to $\otimes$ nodes we may visit branches that do lead to l, and we may there encounter $\oplus$ nodes: in this case, it is sufficient to follow a single arbitrary child to obtain a valid trail.

The following section concerns some complexity improvements of the algorithms based on considering several nodes at once: observe how the enumeration strategy remains the same, and it is simply necessary to store the compressed sequences rather than a single node.

6 Practical and Theoretical Improvements

In this section, we showcase two improvements that can be applied to both algorithms AssessET and TreeAssessET. The first improvement, described in Section 6.1, is a practical pre-processing

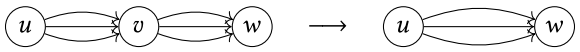


Fig. 6. Compression of chain node v.

step, which is also implemented in the code used for the experiments of Section 7. Section 6.2 instead presents a theoretical improvement which allows to reach the complexity bounds of Theorem 8, shaving an $O(m)$ factor from the theoretical complexity of the algorithms.

6.1 A Practical Improvement: Chain Compression

Our first improvement is a pre-processing step aimed at reducing the number of iterations of our algorithms. We do this by compressing *chain nodes*: a node v is a chain node if it has exactly one in-neighbor and one out-neighbor, that is, $|N^-(v)| = |N^+(v)| = 1$. Reducing the number of chain nodes immediately reduces the number of iterations of our algorithms: indeed, when considering a chain node as source, both of our algorithms would perform an iteration without incrementing the global lower bound lb_{ET} (see the complexity analysis at the end of Section 4). Note that chain nodes can have high degree, it is the number of neighbors that is fixed to one. Still, if v is a chain node with $N^-(v) = \{u\}$ and $N^+(v) = \{w\}$ then the multiplicities of (u, v) and of (v, w) are equal since the graph is Eulerian.

In what follows, let v be a chain node of a graph G, with $N^-(v) = \{u\}$ and $N^+(v) = \{w\}$. We say that we *compress* chain node v when we perform the following operations to graph G (see Figure 6):

—Remove node v and all its incident edges from the graph
—Add m copies of edge (u, w) to the graph, where m was the multiplicity of (u, v).

To ensure that we do not change the number of Eulerian trails during compression of chain nodes, we need the following remark:

Remark 4. If u is also a chain node, then the number of node-distinct Eulerian trails of the graph does not change during the compression of v.

Indeed, when u is a chain node as well, any Eulerian trail reaching u will be forced to go to node w after passing through v. Note that this is not necessarily true if u is not a chain node; for instance, consider the case where w was already a neighbor of u in G: $w \in N^+_G(u)$. In this case, there are two node-distinct choices for an Eulerian trail passing through u: going to v then w, or directly to w. After performing chain compression for node v, the only choice becomes going directly to w, losing thus some node-distinct trails in the process.

Our pre-processing procedure now follows naturally: we remove all chain nodes whose in-neighbor is a chain node as well. We do this by considering one node v at a time and checking whether (i) v is a chain node and (ii) its only in-neighbor is a chain node. If both conditions hold, we perform chain compression for v. Such procedure runs in linear time in the size of the graph, since every node is considered exactly once, and all operations take $O(1)$ time.

6.2 A Theoretical Improvement: Branching Sources

We may think of our algorithms AssessET and TreeAssessET as a recursive computation (handled explicitly with pop/push on a stack for the former, and with the list $\mathcal{L}$ in the latter) having the drawback that it makes $O(mz)$ recursive calls. To try and speed up the process, one could resort to existing decremental SCC algorithms [45]. However, these tend to add (poly)logarithmic factors, and do not immediately yield improvements unless further amortization is suitably designed.

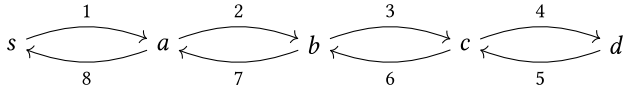


Fig. 7. This graph has a single node-distinct Eulerian trail from s to itself, even if all nodes except for s and d are branching.

We use a different approach, reducing the number of calls to $O(z)$ by guaranteeing that each call generates at least two further calls or immediately halts when one Eulerian trail is found. In this section, we show how to attain this goal with a theoretically efficient combinatorial procedure.

6.2.1 Introducing Function BranchingSource. Let C_i be the SCC chosen for a given step of our algorithms, and let s_i be its source. We call a node $u \in C_i$ *branching* if it has at least two distinct out-neighbors in C_i, that is, $|N^+_{C_i}(u)| \geq 2$. Thus, if s_i is branching (i.e., has at least two out-neighbors in C_i), we have at least two calls by Lemma 2. The issue comes when s_i has just one out-neighbor, as illustrated in Figure 7: some of the remaining nodes could be branching but, unfortunately, only one node-distinct Eulerian trail exists. Thus, the existence of branching nodes when the source s_i is not branching does not guarantee that we attain our goal.

One first solution comes to mind, as it is exploited in our lower bound $lb(\cdot)$ of Equation (2). Consider a trail $T \in ET(C_i)$, which is nonempty as C_i is Eulerian: a node u gives rise to at least $|N^+_{C_i}(u)| - 2$ further Eulerian trails by Lemma 2 as, when u becomes a source for the first time, one out-neighbor of u is part of T and at most one out-neighbor of u leads to a bridging edge; thus the remaining $|N^+_{C_i}(u)| - 2$ out-neighbors can be traversed in any order by so many other Eulerian trails. While this helps for $|N^+_{C_i}(u)| \geq 3$, it is not so useful in the situation illustrated in Figure 7, where all branching nodes have $|N^+_{C_i}(u)| = 2$.

Main Idea. A better solution is obtained by introducing a function BranchingSource to be applied to any tuple f of SCCs from the frontier data structure $\mathcal{F}$ of AssessET, or to any SCC from the list $\mathcal{L}$ of TreeAssessET. If any of these SCCs has a branching source, then BranchingSource returns f itself. Otherwise, it examines each SCC C in f: if $\#ET(C) = 1$, it removes C from f as it is trivial; otherwise, it finds the longest common prefix P of all trails in $ET(C)$, and computes the SCCs of $C \setminus P$, which take the place of C in f. Among these SCCs, one is guaranteed to have a branching source, so BranchingSource returns f updated in this way. Note that only trivial SCCs are removed by BranchingSource, and hence the number of Eulerian trails cannot change. If f is empty, then we have a single Eulerian trail as there is no choice. BranchingSource can be implemented in $O(m^2)$ time as it simulates what our algorithms do until a branching source is found. The challenge is to implement it in $O(m)$ time. Armed with that, we can modify both AssessET and TreeAssessET to get their improved versions ImprovedAssessET and ImprovedTreeAssessET, where we guarantee in $O(m)$ time that *the source s_i is always branching*. The modification is just a few lines, once BranchingSource is available, so we do not provide a detailed description of the pseudocode.

6.2.2 Linear-Time Computation of BranchingSource. Suppose that tuple f in the frontier data structure $\mathcal{F}$ contains only SCCs with non-branching sources (otherwise, BranchingSource returns f unchanged). Consider any SCC C in the tuple f. The main idea is to fix any trail $T \in ET(C)$, which can be found in $O(|E(C)|)$ time, and traverse T asking at each node u whether there is an alternative trail T' branching at u.

Swap Edges. Let us start with the following definition.

Definition 6.1. Given an Eulerian trail T of an SCC C, let T_u be the prefix of T from its source s to the first time u is met and (u, v) be the next edge traversed by T. An edge (u, v') in C is a *swap edge*

if $T_u \cdot (u, v')$ is prefix of another Eulerian trail $T' \neq T$ and $v' \neq v$. We say that u admits a swap edge and $T_u = T'_u$ is *the longest common prefix* of T and T'.

The discovery of swap edges in C is key to BranchingSource: although different Eulerian trails of C may give rise to different swap edges in C, these trails all share P, so the node u at the end of P can be identified by T_u (Definition 6.1), for *any* trail $T \in ET(C)$.

Lemma 4. *Suppose that all swap edges are known in an SCC C of f for any given trail $T \in ET(C)$. Then (i) $\#ET(C) = 1$ if and only if there are no swap edges in C; moreover, (ii) if $\#ET(C) > 1$, let u be the first node that is met traversing T and that admits a swap edge. Then $P = T_u$ is the longest common prefix of all the trails in $ET(C)$.*

Proof. Claim (i) is immediate. As for Claim (ii), we observe that, since P is common to all the trails in $ET(C)$, any trail suffices and thus we choose T to find node u: the first met node which admits a swap edge cannot be found in T_u earlier than u (i.e., P is shorter), as otherwise T_u would be shorter too. That node cannot be found later than u in the trail T (i.e., P is longer), as there is already an alternative trail at u. So, it must be u, and $T_u = P$. □

Using Swap Edges in BranchingSource. Based on Lemma 4, BranchingSource examines each $C \in f$: it tests whether C is trivial ($\#ET(C) = 1$), or it finds the longest common prefix $P = T_u$ of all the trails. If all SCCs are trivial, it returns an empty f. Otherwise, it deletes from f the trivial SCCs found so far, and for the current non-trivial SCC C, it computes the set $\mathcal{C}$ =ComputeSCC ($C \setminus T_u$) of SCCs. Note that u becomes the source of an SCC in $\mathcal{C}$ and u is branching as it admits a swap edge (u, v'), along with (u, v) from its trail T. Thus, u keeps at least two out-neighbors v and v' in $\mathcal{C}$. At this point, BranchingSource stops its computation, updates f by replacing C with the SCCs from $\mathcal{C}$, and returns f. As only trivial SCCs are removed from f, and the number of Eulerian trails in C is the product of those in the SCCs of $\mathcal{C}$, the overall number of Eulerian trails in f does not change before and after its update. This proves Lemma 5.

Lemma 5. *Given any tuple f in $\mathcal{F}$ and the set of swap edges in the SCCs of f, the function BranchingSource takes $O(m)$ time to update f, so that either f is empty (a single Eulerian trail exists in f), or f contains at least one SCC with branching source.*

Remark 5. Since every swap edge generates *at least one new Eulerian trail*, if we can find all swap edges in $O(m)$ time, we can employ $lb(G) = 1+$ "number of swap edges" in our algorithm. Any node with three different out-neighbors generates at least a swap edge. Thus, this new choice for the lower bound function necessarily performs better than the one shown in Equation (2). For example, in the right of Figure 8, there are 5 swap edges whereas $lb(\cdot) = 1$.

Finding Swap Edges in Linear Time. We are thus interested in finding all the swap edges in linear time. We need the following property to characterize them for an SCC C of f.

Lemma 6. *Let C be an SCC, and let T with prefix $T_u \cdot (u, v)$ be one of its Eulerian trails. Edge (u, v'), for $v' \neq v$, is a swap edge if and only if there is a trail from v' to u (i.e., u is reachable from v') in $C \setminus T_u$.*

Proof. ($\Rightarrow$) If (u, v') is a swap edge, then the new Eulerian trail T' traverses edge (u, v) after (u, v'). Let $T' = T_u \cdot (u, v') \cdot \mathcal{P}^u_{v'} \cdot (u, v) \cdot T''$. Since Eulerian trails go through each edge exactly once, $\mathcal{P}^u_{v'}$ is a directed trail from v' to u that does not use the edges in $T'_u = T_u$. ($\Leftarrow$) Let $T = T_u \cdot (u, v) \cdot \mathcal{P}^u_v \cdot (u, v') \cdot T''$, with $\mathcal{P}^u_v$ trail from v to u. Recall that, by hypothesis, v' has a trail to u in $C \setminus T_u$, and let us select a node x as follows:

— If v' still has a trail to u in $C \setminus \{T_u \cdot (u, v) \cdot \mathcal{P}^u_v\}$, consider $x = u$.

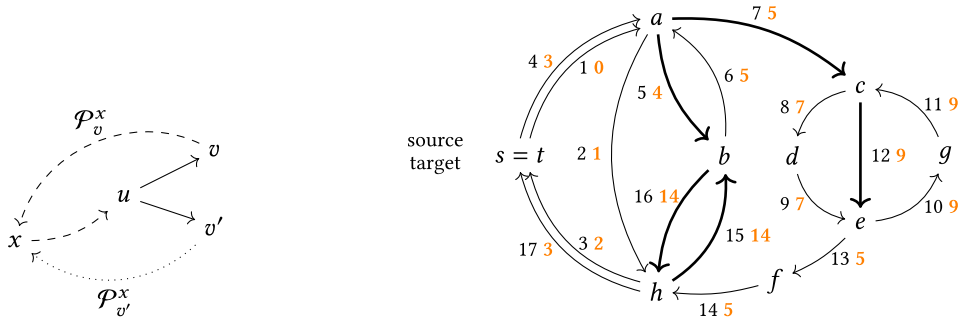


Fig. 8. Left: Choice of node x in the proof of Lemma 6. Right: Example of an Eulerian graph with $s = t$. The black (left) numbers on the edges are the Eulerian trail numbers *etn* for a given trail T; the orange (right) ones are the disconnecting indices *di*s. Swap edges are in bold.

—Otherwise, some edges needed for v' to reach u have been used by $\mathcal{P}_v^u$. Consider the first edge (in the traversal order) of $\mathcal{P}_v^u$ that shares its head with an edge of T'', and let x be their head (see the left of Figure 8).

Let $\mathcal{P}_v^x$ be the prefix of trail $\mathcal{P}_v^u$ up to the first occurrence of node x. Similarly, let $\mathcal{P}_{v'}^x$ be the prefix of T'' up to the first occurrence of x. The Eulerian trail T will then be of the form $T_u \cdot (u,v) \cdot \mathcal{P}_v^x \cdot \mathcal{P}_v^u \setminus \mathcal{P}_v^x \cdot (u,v') \cdot \mathcal{P}_{v'}^x \cdot T'' \setminus \mathcal{P}_{v'}^x$. Consider now $T' = T_u \cdot (u,v') \cdot \mathcal{P}_{v'}^x \cdot \mathcal{P}_v^u \setminus \mathcal{P}_v^x \cdot (u,v) \cdot \mathcal{P}_v^x \cdot T'' \setminus \mathcal{P}_{v'}^x$, obtained from T by swapping $(u,v) \cdot \mathcal{P}_v^x$ and $(u,v') \cdot \mathcal{P}_{v'}^x$, both of which are trails from u to x. By construction, T' is an Eulerian trail of C, equal to T up to the first occurrence of node u, and with (u,v') as the next edge. That is, (u,v') is a swap edge. □

In order to find the swap edges, we need to traverse T in reverse order and assign each edge $e \in E(C)$ two integers, as illustrated in the example in the right of Figure 8: (i) the *Eulerian trail numbering* $etn(e)$, which represents the position of e inside T and is immediate to compute, and (ii) the *disconnecting index* $di(e)$, which is discussed in the next paragraph as its computation is a bit more involved. As we will see (Lemma 7), comparing these integers allows us to check if a given edge is a swap edge in constant time.

Disconnecting Indices. We introduce the notion of disconnecting index relatively to a given trail $T \in ET(C)$, according to the following rationale. We observe that Lemma 5 characterizes a swap edge (u,v') by stating that u and v' must belong to the same SCC after T_u is removed from C. Suppose that we want to traverse T to discover the swap edges. Equivalently, we take the edges according to their *etn* order in T. Fix any edge (u,v'). At the beginning, u and v' are in the same SCC C. Next, we start to conceptually remove, from C, the edges traversed by an increasingly long prefix of T: how long will u and v' stay in the same SCC? In this scenario, the disconnecting index of (u,v') corresponds to the maximum *etn* (hence prefix of T) for which u and v' will stay in the same SCC, i.e., removing any prefix of T longer than this one from C disconnects v' from u.

For any $\ell \in [0,m]$, we denote by $T_{\leq \ell}$ the prefix T_u of T such that $|T_u| = \ell$. When $\ell = 0$, it is the empty prefix; when $\ell = m$, it is T itself.

Definition 6.2. Given an edge $(u,v') \in E(C)$, its *disconnecting index* $di(u,v')$ is given by:

$$\max \{0 \leq \ell < etn(u,v') \mid u, v' \text{ are inside an SCC of } C \setminus T_{\leq \ell}\}.$$

Figure 8 (right) illustrates an example where the following property can be checked by inspection.

Lemma 7. *For any edge $(u, v') \in E(C)$, we have that (u, v') is a swap edge for a given trail if and only if $di(u, v') \geq etn(u, v) - 1$ for some $v \neq v'$.*

Proof. ($\Rightarrow$) Let (u, v') be a swap edge, and let (u, v) be the edge traversed by T after T_u, with $etn(u, v) = p + 1$ and $|T_u| = p$. Trivially, (u, v') constitutes a trail from u to v' in $C \setminus T_u$. By Lemma 6, there is also a trail from v' to u in $C \setminus T_u$. Therefore, u and v' are in the same SCC of $C \setminus T_u = C \setminus T_{\leq p}$, which implies $di(u, v') \geq p = etn(u, v) - 1$.

($\Leftarrow$) Let us now assume that $di(u, v') \geq etn(u, v) - 1$ for some (u, v) with $v \neq v'$. Wlog we can consider (u, v) with minimum *etn*. By Definition 6.2, u and v' are in the same SCC of $C \setminus T_{\leq p} = C \setminus T_u$ where $p = etn(u, v) - 1$. Thus v' reaches u in this subgraph, that is, (u, v') is a swap edge. □

Linear-Time Computation of Disconnecting Indices. Consider an SCC C from $f \in \mathcal{F}$, and any arbitrary trail $T \in ET(C)$ (computable in $O(|E(C)|)$ time). Assign the Eulerian trail numbering $etn(e)$ to each edge $e \in E(C)$. We discuss how to assign the disconnecting index $di(e)$ to each edge e in $O(|E(C)|)$ time.

We proceed by reconstructing T backwards. That is, we conceptually start from an empty graph, and we add edges from T, one at a time from last to first (i.e., in decreasing order of their *etn* values), until all edges from T are added back obtaining again the SCC C. During this task, along with disconnecting indices, we also assign a *flag* $tr(u) = true$ to the nodes u touched by the edges that have been added. We keep a stack, Bridges, for the edges that have been added but do not yet have a disconnecting index, i.e., they are not in an SCC of the current partial graph. More formally, we will guarantee the following invariants:

I1 The edges in Bridges have increasing *etn* values, starting from the top.

I2 The edges in Bridges are all and only the bridging edges of the current graph.

I3 Given any two consecutive edges e, e' in Bridges, the edges with *etn* values in $[etn(e) + 1, etn(e') - 1]$ (which, observe, are not in Bridges) make up an SCC of the current graph.

I4 The flag $tr(u)$ is *true* if and only if u is incident to an edge of the current graph.

We describe the algorithm and prove its correctness and all invariants.

For $\ell = m, m-1, \ldots, 1$, step ℓ adds back to the current graph the edge (u, v) such that $etn(u, v) = \ell$. Let u be the tail and v be the head of the edge.

- If the tail u has not been explored (i.e., $tr(u) = false$), we add (u, v) to Bridges and set $tr(u) = true$. If $\ell = m$, v is the last node of the trail and we also set $tr(v) = true$.
- Otherwise, u has been traversed before, and there must be at least an edge incoming in u in our current graph; let (z, u) be the one such edge with highest *etn* value, say, $etn(z, u) = x$. We assign $di(u, v) = etn(u, v) - 1$, and pop all edges e from Bridges such that $etn(e) \leq x$, assigning $di(e) = etn(u, v) - 1$ to all of these too.

Lemma 8. *Given an SCC C from f in $\mathcal{F}$, and any arbitrary trail $T \in ET(C)$, the disconnecting indices of T can be computed in $O(|E(C)|)$ time and space.*

Proof. The proof will consist in showing that all invariants hold at all times, and all assigned $di(\cdot)$ are correct.

All invariants trivially hold at the step $\ell = m$ of the computation, as the graph is composed of a single edge (that is bridging); all $tr(\cdot)$ are initialized to *false*.

As edges are taken into account in decreasing order of their *etn* value and only added once to Bridges, invariant I1 always holds. I4 also trivially holds, as the head of the edge at step ℓ is already the tail of the edge at step $\ell + 1$, which has been already considered (except in the case $\ell = m$, which is handled ad-hoc). Furthermore, since the graph is always Eulerian, by Lemma 1 its SCCs

will have a chain structure. Therefore, the edges between bridges of the current graph necessarily form SCCs. That is, I2 directly implies I3. In the rest, we thus only need to focus on I2.

When edge (u, v) is added at step ℓ, we have two cases.

- If the tail u has not been traversed yet, it has no other incoming or outgoing edge in the graph so far; thus u must be the source and (u, v) a bridge. No $di(\cdot)$ is assigned, and I2 is still satisfied.
- Otherwise, u has been traversed before. We need to show that the popped edges are exactly the ones that are part of the same SCC as node u, but that have not been previously popped. First, note that by adding edge (u, v), we just closed a directed circuit composed of the edges from (u, v) to (z, u) in T, where (z, u) has maximum *etn* value x among incoming edges in u. Thus, the popped edges are surely in the same SCC as u. Furthermore, all these edges were bridging edges until now (by I2). Thus, since we are traversing the edges of T backwards, their disconnecting index must be precisely $etn(u, v) - 1$ as it is largest.
  Vice versa, let us now prove that there are no other edges of the stack in the SCC of u. By contradiction, let $e \in$ BRIDGES be such an edge, with $etn(e) > x$. That is, in the current graph there are both a trail from u to the tail of e, and from the head of e to u. Since $etn(e) > x \geq \ell + 1$, edge e appears after edge (z, u) in trail T, and thus u reached the tail of e also at the previous step of the computation $(\ell + 1)$. As (u, v) cannot help e to reach u, the head of e also reached u at the previous step. This leads to a contradiction: e would not have been a bridging edge at the previous step, violating invariant I2. We thus remove from Bridges exactly the edges that start belonging to a non-trivial SCC, assigning them their correct disconnecting indices. The edges left in the stack must be the bridges of the current graph, preserving I2.

Finally, let us discuss the complexity: all operations concerning the $tr(\cdot)$ values, the stack, and $di(\cdot)$ values take constant time each, and thus $O(|E(C)|)$ time in total. The identification of the edge (z, u) of maximum *etn* among incoming edges in u can be managed by storing, in each node with $tr(\cdot) = true$, the maximum *etn* value among its incoming edges. Note that this assignment is only performed once, as edges are traversed in decreasing *etn* values, thus we just record the first edge incoming in u considered in the backwards traversal. The whole cost amounts to $O(|E(C)|)$ time. Space is also $O(|E(C)|)$ words of memory, since we only store the graph and, for each node, *tr* and the maximum *etn*. □

We thus arrive at our main result.

Theorem 1. *Given a directed multigraph $G = (V, E)$, with $|E| = m$, and an integer z, assessing $\#ET(G) \geq z$ can be done in $O(m \cdot \min\{z, \#ET(G)\})$ time using $O(mz)$ space.*

Other than being easily extensible to the *edge-distinct* case, the algorithm underlying Theorem 1 has an attractive property: its number of $O(m)$-time steps is z in the worst case, but can be significantly smaller, thanks to suitable lower bounding techniques. This property means that our assessment algorithm can potentially run in less than constant amortized time per solution on favorable instances.

7 Experimental Evaluation

Setup and Datasets. We evaluated AssessET with the Frontier data structure (Section 4), referred to as AF, and with the tree-like data structure (Section 5), referred to as AT. To achieve a good compromise between theory and practice, both our algorithms employ the *chain-compression* strategy described in Section 6.1, but they do not employ the *branching source* computation of Section 6.2. Indeed, while the latter one is necessary to obtain the claimed theoretical bound, its linear cost in every recursive call is significant, while its benefit may only occur in rare cases.

Table 1. Datasets Characteristics

Dataset	Length n	Alphabet size $\|\Sigma\|$
XML	50,000,000	96
SOURCES	50,000,000	102
ENGLISH	50,000,000	236
DNA	50,000,000	16
PROTEINS	50,000,000	25
PITCHES	69,904,636	64

We compared our two implementations of AF and AT to BEST, the application of the BEST theorem, all for assessing the number of node-distinct Eulerian trails in directed multigraphs.

In all three implementations, we first compute the ratio $F = \prod_{u \in V}(r_u - 1)!(\prod_{(u,v)\in E} a_{uv}!)^{-1}$ of factorials (see Equation (1)) and give a positive answer when $F \geq z$. Otherwise, we proceed with the workflow of our algorithms, or in the case of BEST, we compute $\det L$ and combine its value with F. The BEST implementation makes use of the Sparse LU decomposition function of the open-source Eigen library (v. 3.3.7) [29], which is based on the algorithm of [46], to compute $\det L$.

To obtain realistic directed multigraphs, which are also *Eulerian*, we used all six datasets from the popular Pizza & Chilli corpus [47]. These are string datasets that come from different domains and have alphabets of different sizes. Their characteristics are summarized in Table 1. The first five of these datasets are ASCII-encoded strings, and the last one had to be converted to such a string. From each string S, we then constructed a dBG that was given as input to the ETA problem. Indeed, since the dBG came directly from a single string S, it was Eulerian. One could make a non-Eulerian graph Eulerian (as described in *A4*) and use it as well, but we did not use non-Eulerian graphs, as the efficiency of our algorithms does not depend on the dataset domain. Moreover, note that any Eulerian graph can be easily transformed to a dBG by setting $\Sigma := V$.

We have implemented AF and AT in C++. Our code is available at https://github.com/gloukides/eta/. For BEST, we used the C++ implementation of [5]. All experiments ran on an AMD EPYC 7282 @ 2.8 GHz CPU with 512 GB RAM.

Comparison to BEST. After ensuring that all implementations give always the same answer, we have turned our focus on efficiency, examining the impact of parameters.

Impact of d. We examined the impact of d, i.e., the order of the dBG which is constructed from a string dataset (see Section 1). A larger d increases the number of nodes of the dBG and very slightly decreases the number of edges, which is $|S| - d + 1$, where $|S|$ is the string length. Thus, the graph becomes sparser and sparser, as d increases: the number of repeating substrings of length-d in S decreases as d increases.

Figure 9 shows the results for runtime for varying d on all the datasets. As can be seen, our algorithms are *up to more than two orders of magnitude faster* than BEST, while AF and AT is *more than 20 and 26.7 times faster*, respectively on average (over all datasets). The difference is larger for small values of d (e.g., $d = 32$ which correspond to dense graphs). This is because the input dBGs are denser and the implementation of BEST exploits Sparse LU decomposition, which is faster for sparse graphs but less efficient for dense graphs. On the other hand, our algorithms are not substantially affected by d. For example, even for the largest $d = 1{,}024$, AF and AT were 4 and 5.5 times faster than BEST, respectively. Note also that AT outperforms AF by 3.2 times on average. Furthermore, there are two "outlier" cases (Figure 9(c) for $d = 512$ and $d = 1{,}024$), where the running time of AF surpasses not only that of AT but also that of BEST, due to a significant duplication of tuples in the frontier data structure. However, the performance of AT remains consistently faster than BEST, as it does not duplicate SCCs (see Remark 2). This highlights the practical benefit of AT.

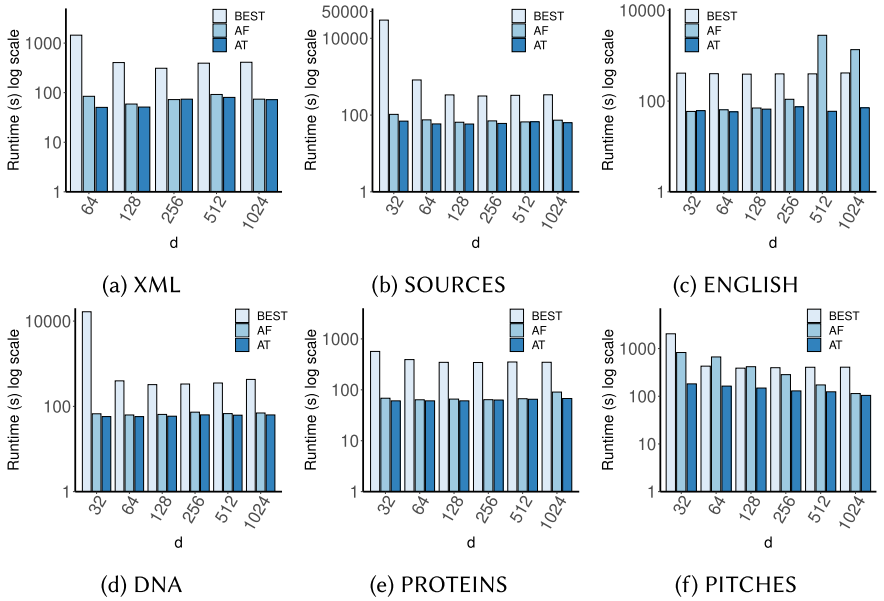


Fig. 9. Runtime vs. d for $z = 1{,}000$. We omit the result for $d = 32$ from (a) because all algorithms answer YES, based on the ratio of BEST theorem, which is very fast.

Figure 10 shows the size (number of edges) of dBG after chain compression performed by AF and AT, for the experiments of Figure 9, as well as the number of iterations performed by these algorithms. The compressed dBG has a much smaller size than the input dBG; the latter contains about 50 million edges for all datasets except PITCHES and about 70 million edges for PITCHES. Also, the size of the compressed dBG decreases as d increases, since the degree of nodes decreases and the paths get longer and are compressed. Interestingly, the number of iterations performed by our algorithms is in most cases much smaller than the size of the compressed dBG and in any case much smaller than mz. This is encouraging, as it shows that the quadratic in m worst-case bounds of our simple algorithms are quite pessimistic, and it justifies the use of these algorithms.

Impact of z. We examined the impact of z, i.e., the threshold in ETA. Clearly, a larger z does not affect Best, since the BEST theorem counts all Eulerian trails irrespectively of z, but it affects our algorithms as it generally increases the number of iterations they perform.

Figure 11 shows the results for varying z for all the datasets. As can be seen, our algorithms are *up to 6 times faster* than Best, while AF and AT are *3.5 and 4.4 times faster* on average, respectively (over all datasets). The difference is generally smaller for large values of z, as the number of iterations performed by our algorithms increases. Again, AT outperforms AF by 7.7 times on average, and there are cases in which AT performs much better. There are four cases (see Figure 11(c) and (d) for $z \geq 10^4$, and Figure 11(f) for $z = 10^5$) in which AF performs worse than Best due to the large number of iterations it performed. For example, in Figure 11(d) for $z = 10^4$, AF performed 932,197 iterations, but for $z = 10^5$ the number of iterations increased to 8,540,177. On the other hand, the number of iterations of AT was much smaller (e.g., in Figure 11(d), it was 2,490 for $z = 10^4$ and 24,978 for $z = 10^5$). It is encouraging to see that even with large z values, such as $z = 10^5$, our algorithms perform so well.

Impact of $|S|$. Finally, we examined the impact of $|S|$, i.e., the length of string S from which the input dBG to ETA is constructed. Note that $m = |S| - d + 1$, which counts also multiplicities. Thus,

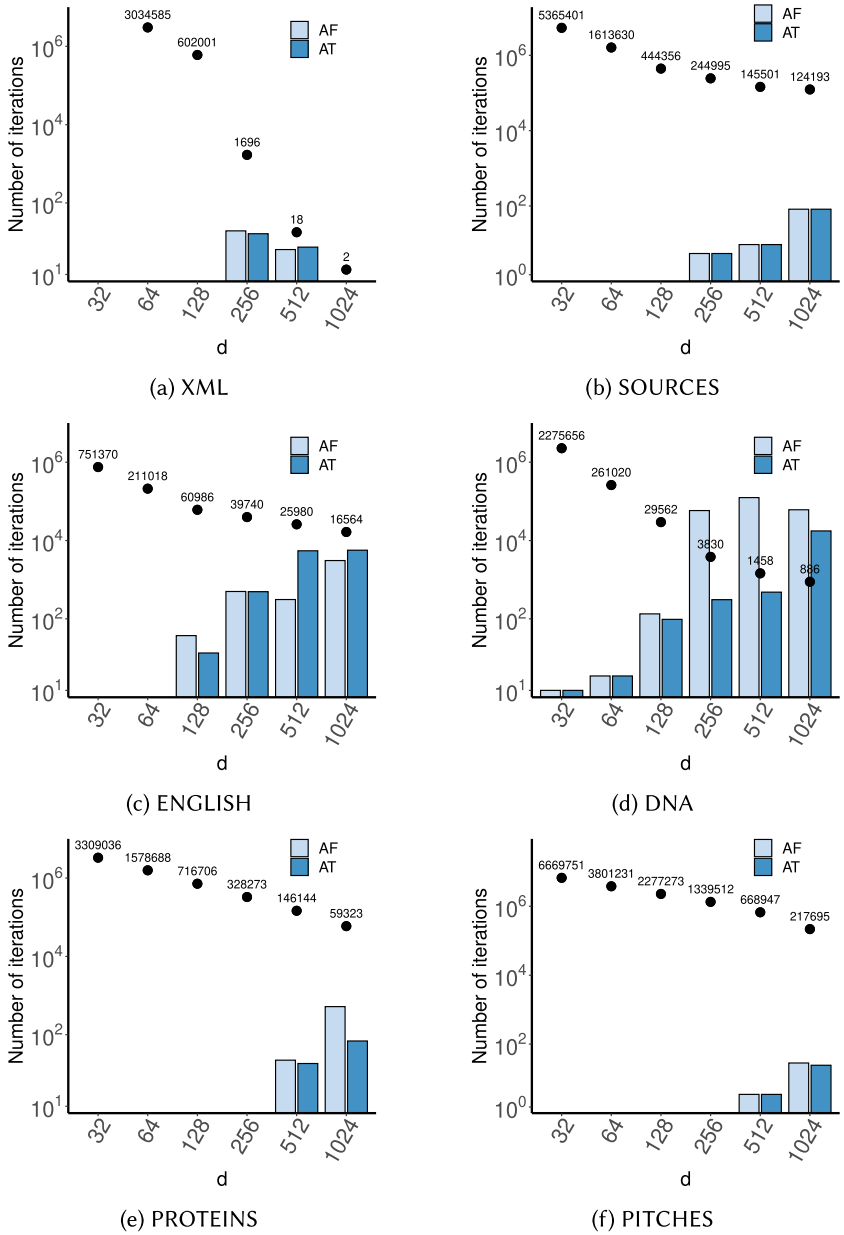


Fig. 10. Number of iterations vs. d. Each dotted line corresponds to m *after* chain compression. In (a) both algorithms simply calculate $F \geq z$ and return a positive answer, for $d = 32$. When there are no bars, the algorithms do not perform any iterations because the minimum lower bound is at least z.

a larger string length $|S|$ increases the number of nodes and edges of the dBG, and therefore the runtime of both Best and our algorithms also increases.

Figure 12 shows the results for runtime for varying $|S|$ on all datasets. As can be seen, our algorithms are *up to 6.5 times faster* than Best, while on average (over all datasets), AF is over 3.9

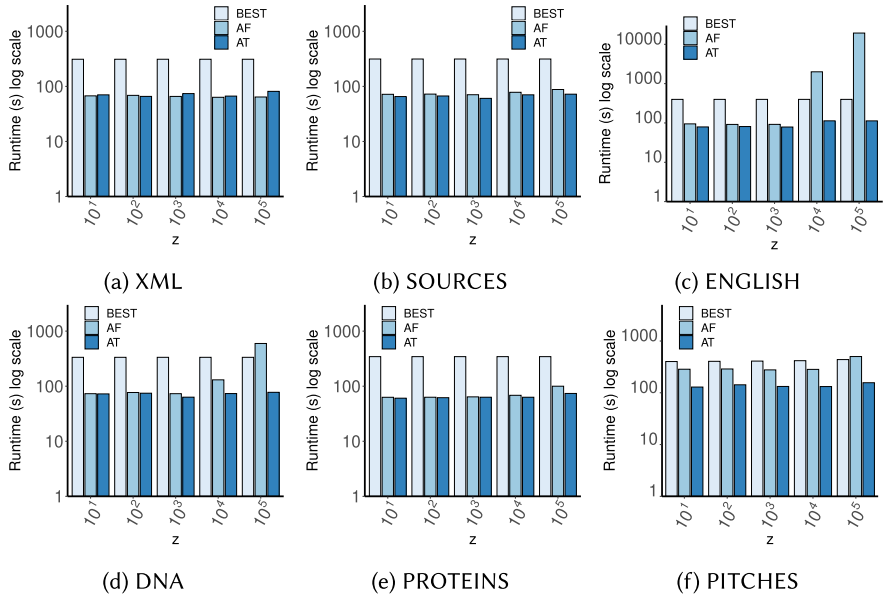


Fig. 11. Runtime vs. z for $d = 256$.

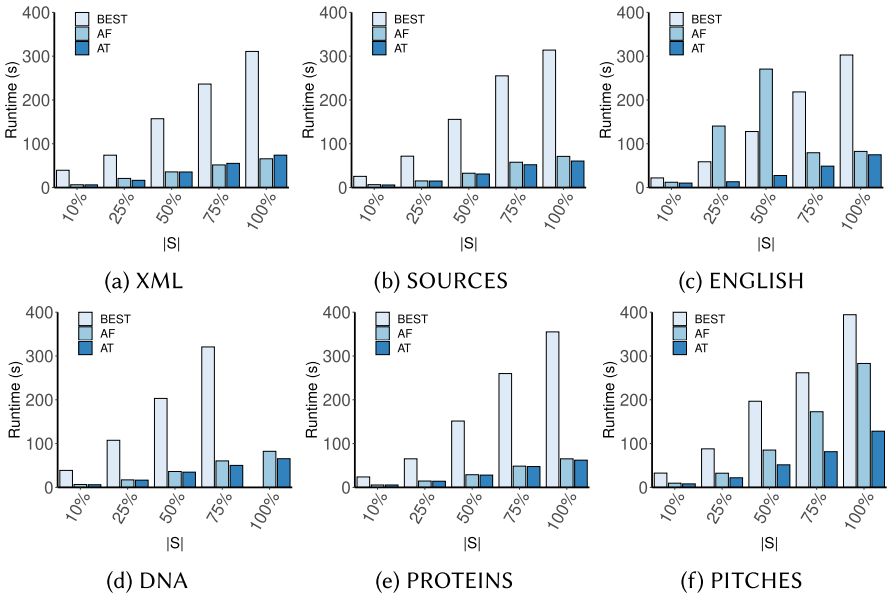


Fig. 12. Runtime vs. $|S|$ for $z = 1{,}000$ and $d = 256$.

and AT is over 4.8 times faster, respectively. Furthermore, the difference generally becomes larger as $|S|$ increases. Again, AT was faster than AF by up to 12.7 times and by 2 times on average (over all datasets). As expected, both AT and AF scale much better than their $O(m^2 \min\{z, \#ET(G)\})$-time bound. In particular, it is encouraging to see that AT scales near-linearly with m.

Table 2. Runtime Comparison for Counting $\#ET(G)$

(a) XML

Algorithm	$z = \#ET(G) = 74{,}880$ and $d = 280$	$z = \#ET(G) = 25{,}897{,}000$ and $d = 267$
Best	1,481	1,369
AF	514	42,189
AT	383	436

(b) SOURCES

Algorithm	$z = \#ET(G) = 16{,}818$ and $d = 16{,}384$	$z = \#ET(G) = 153{,}475{,}000$ and $d = 14{,}500$
Best	1,244	1,349
AF	406	$>$ 43,200
AT	391	$>$ 43,200

(c) ENGLISH

Algorithm	$z = \#ET(G) = 131{,}072$ and $d = 35{,}000$	$z = \#ET(G) = 4{,}194{,}300$ and $d = 33{,}000$
Best	1,3331	1,329
AF	498	8,024
AT	313	318

(d) DNA

Algorithm	$z = \#ET(G) = 16{,}818$ and $d = 16{,}384$	$z = \#ET(G) = 153{,}475{,}000$ and $d = 14{,}500$
Best	1,682	1,683
AF	320	3,901
AT	326	717

(e) PROTEINS

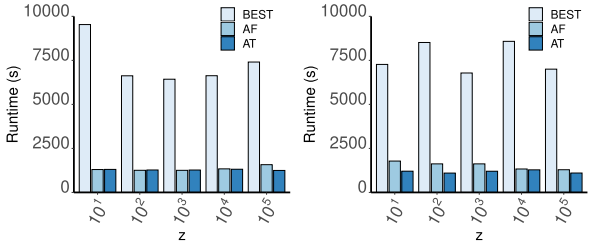
Algorithm	$z = \#ET(G) = 1{,}728$ and $d = 8{,}750$	$z = \#ET(G) = 33{,}984{,}000$ and $d = 6{,}554$
Best	1,314	1,429
AF	9,236	$>$ 43,200
AT	321	338

(f) PITCHES

Algorithm	$z = \#ET(G) = 6{,}244$ and $d = 9{,}830$	$z = \#ET(G) = 2{,}352{,}380$ and $d = 8{,}192$
Best	1,936	1,942
AF	486	12,216
AT	483	6,663

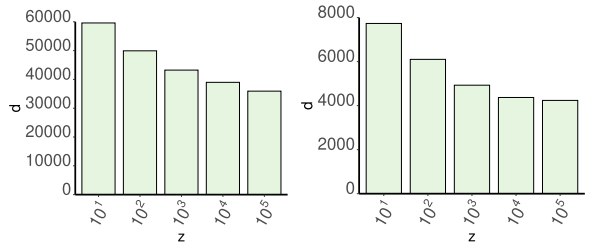
The reported timings are in seconds; since some instances did not finish within 12 hours, we have marked them with $>$ 43,200, the corresponding amount in seconds.

Counting $\#ET(G)$. As we have shown, AT and AF are more efficient than Best for assessing the $\#ET(G)$. One may wonder, however, how AT and AF compare to Best for counting $\#ET(G)$. For this, we first used Best with some d value to obtain $\#ET(G)$ and then we applied our algorithms with the same d value and $z :=$ MAX_INT – recall the multiplicative $\min\{z, \#ET(G)\}$ factor in the time complexity of our algorithms. The results in Table 2 show that AT and AF were faster than Best (with AT being faster than AF) when $\#ET(G)$ was small, and in some cases, AT and AF were slower when $\#ET(G)$ was large. These results confirm our theoretical findings, and show that AT and AF can be useful for counting in instances where $\#ET(G)$ is expected to be small.



(a) ENGLISH (b) DNA

Fig. 13. Speeding up the approach of [28]: runtime vs. z.



(a) ENGLISH (b) DNA

Fig. 14. Maximal d found by the approach of [28] vs. z.

Case Study: Defending Against Reconstruction Attacks. We show that both AF and AT can substantially speed up the defense approach in [28], if they are used instead of Best. Recall that the approach in [28] does not need to count $\#ET(G)$, which is the output of Best: it suffices to assess whether $\#ET(G) \geq z$, which is exactly what our algorithms do. Recall also that the approach in [28] aims to find the maximal d for which $\#ET(G) \geq z$, which implies that our algorithms or Best are applied a large number of times with different candidate values for the maximal d and thus the efficiency benefit they bring accumulates.

In these experiments, we used the values for z from [28] and the ENGLISH and DNA datasets, as they come from domains where reconstruction attacks are likely. Figure 13 shows the total time spent for counting $\#E(G)$ by Best (respectively, assessing $\#E(G) \geq z$ by our algorithms), for varying z. As can be seen, our algorithms brought substantial time savings, as the total time they need is up to 7.7 smaller compared to the time needed by Best. Also, the time needed by AF and AT is lower by 5.3 and 6.1 times on average (over all datasets). Figure 14 shows the maximal d values found by the approach of [28] for varying z. Although these values are clearly the same for both our algorithms and Best, for a given z, we show them to justify that the z values we used in all the experiments allow the approach of [28] to incur no accuracy loss for pattern matching queries on very long (i.e., up to length-d) substrings, while providing privacy against dataset reconstruction (i.e., probability of data reconstruction of no more than $\frac{1}{z}$).

8 Conclusion and Future Work

We considered the problem of assessing the number of Eulerian trails in a directed multigraph, which underlies many applications in domains such as data privacy and computational biology. In these applications, the BEST theorem is typically used to count, instead of assess, the number of Eulerian trails, which can be inefficient for large graphs. In response, we proposed the AssessET algorithm for solving the assessment problem and the TreeAssessET algorithm, which employs a novel tree

data structure to reduce the number of iterations in comparison to AssessET. We enhanced the practical performance of our algorithms with a chain-compression strategy and presented a way to bring the time complexity of both algorithms down to $O(mz)$ based on combinatorial insight.

Our experiments show that AssessET and TreeAssessET are up to two orders of magnitude faster than the algorithm applying the BEST theorem, perform much fewer than mz iterations in most cases, and scale near-linearly with m. Our experiments also show that the proposed algorithms bring substantial efficiency benefits in a data privacy application (see A1 in Introduction).

Open Problems. In our view, the main open questions that stem from our work are as follows:

—Can the $O(mz)$-time bound be improved for the assessment problem in general directed graphs?
—Can the $O(mz)$-time bound be improved for the assessment problem in dBGs?

Acknowledgments

We would like to thank Luca Versari for useful discussions on a previous version of the main algorithm and Paweł Gawrychowski for bringing to our attention an algorithm for assessing the number of Eulerian trails via enumerating arborescences.

References

[1] C. Hierholzer and C. Wiener. 1873. Über die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen* 6, 1 (1873), 30–32.
[2] N. L. Biggs, E. K. Lloyd, and R. J. Wilson. 1976. *Graph Theory*, 1736-1936. Clarendon Press.
[3] G. R. Brightwell and P. Winkler. 2005. Counting Eulerian circuits is #P-complete. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the 2nd Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO '05)*. SIAM, Vancouver, BC, Canada, 259–262.
[4] T. van Aardenne-Ehrenfest and N. G. de Bruijn. 1987. Circuits and trees in oriented linear graphs. In: *Classic Papers in Combinatorics*. Branko Grünbaum and G. C. Shephard (Eds.), Springer, Boston, MA, 149–163.
[5] G. Bernardini, H. Chen, G. Fici, G. Loukides, and S. P. Pissis. 2020. Reverse-safe data structures for text indexing. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX '20)*. SIAM, 199–213.
[6] C. Kingsford, M. C. Schatz, and M. Pop. 2010. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics* 11, (2010), 21.
[7] R. Patro, S. M. Mount, and C. Kingsford. 2014. Sailfish: Alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. *Nature Biotechnology* 32 (2014), 462–464.
[8] G. Punzi, A. Conte, R. Grossi, and A. Marino. 2023. An efficient algorithm for assessing the number of ST-paths in large graphs. In *Proceedings of the 2023 SIAM International Conference on Data Mining (SDM '23)*. Shashi Shekhar, Zhe Zhou, Yizhou Chiang, and Gregor Stiglic (Eds.), SIAM, 289–297.
[9] Z. Lai, Y. Peng, S. Yang, X. Lin, and W. Zhang. 2021. PEFP: Efficient k-hop constrained s-t simple path enumeration on FPGA. In *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE '21)*. IEEE, 1320–1331.
[10] J. Cheng, L. Zhu, Y. Ke, and S. Chu. 2012. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*. ACM, 1240–1248.
[11] S. Jain and C. Seshadhri. 2017. A fast and provable method for estimating clique counts using Turán's theorem. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. ACM, 441–449.
[12] A. Conte, D. Firmani, C. Mordente, M. Patrignani, and R. Torlone. 2017. Fast enumeration of large k-plexes. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. ACM, New York, NY, 115–124.
[13] Q. Dai, R.-H. Li, H. Qin, M. Liao, and G. Wang. 2022. Scaling up maximal k-plex enumeration. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM '22)*. ACM, New York, NY, 345–354.
[14] R. Kumar and T. Calders. 2018. 2SCENT: An efficient algorithm to enumerate all simple temporal cycles. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1441–1453.
[15] M. Al Hasan and V. S. Dave. 2018. Triangle counting in large networks: A review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 2 (2018), e1226.
[16] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network motifs: Simple building blocks of complex networks. *Science (New York, N.Y.)* 298, 5594 (2002), 824–827.

[17] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. 2007. Approximating betweenness centrality. In *Proceedings of the 4th International Workshop on Algorithms and Models for the Web-Graph (WAW '07)*. A. Bonato and F. R. K. Chung (Eds.), Lecture Notes in Computer Science, Vol. 4863, Springer-Verlag, Berlin, Heidelberg, 129–137.
[18] S. Bugedo, C. Riveros, and J. Salas. 2024. A family of centrality measures for graph data based on subgraphs. *ACM Transactions on Database Systems* 49, 3 (2024), 1–45.
[19] X. Chen and J. C. S. Lui. 2018. Mining graphlet counts in online social networks. *ACM Transactions on Knowledge Discovery from Data* 12, 4 (2018), Article 41, 1–38.
[20] L. D. Stefani, E. Terolli, and E. Upfal. 2020. Tiered sampling: An efficient method for counting sparse motifs in massive graph streams. *ACM Transactions on Knowledge Discovery from Data* 15, 5 (2021), Article 79, 1–52.
[21] K. Shin, E. Lee, J. Oh, M. Hammoud, and C. Faloutsos. 2021. CoCoS: Fast and accurate distributed triangle counting in graph streams. *ACM Transactions on Knowledge Discovery from Data* 15, 3 (2021), Article 38, 1–30.
[22] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*. ACM, New York, NY, 16–24.
[23] S. Arifuzzaman, M. Khan, and M. V. Marathe. 2020. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Transactions on Knowledge Discovery from Data* 14, 1 (2020), Article 5, 1–34.
[24] V. V. Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC '12)*. ACM, New York, NY, 887–898.
[25] F. L. Gall. 2014. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC '14)*. ACM, New York, NY, 296–303.
[26] J. Alman and V. V. Williams. 2021. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA '21)*. SIAM, 522–539.
[27] J. P. Hutchinson and H. S. Wilf. 1975. On Eulerian circuits and words with prescribed adjacency patterns. *Journal of Combinatorial Theory, Series A* 18, 1 (1975), 80–87.
[28] G. Bernardini, H. Chen, G. Fici, G. Loukides, and S. P. Pissis. 2021. Reverse-safe text indexing. *ACM Journal of Experimental Algorithmics* 26 (2021), Article 1.10, 1–26.
[29] Eigen library. 2020. Eigen: C++ Template Library for Linear Algebra. Retrieved from http://eigen.tuxfamily.org
[30] A. Conte, G. Loukides, N. Pisanti, S. P. Pissis, and G. Punzi. 2021. Beyond the BEST theorem: Fast assessment of Eulerian trails. In *Proceedings of the 23rd International Symposium on Fundamentals of Computation Theory (FCT '21)*. E. Bampis and A. Pagourtzis (Eds.), Lecture Notes in Computer Science, Vol. 12867, Springer-Verlag, Berlin, 162–175.
[31] H. N. Gabow and E. W. Myers. 1978. Finding all spanning trees of directed and undirected graphs. *SIAM Journal on Computing* 7, 3 (1978), 280–287.
[32] T. Uno. 1999. A new approach for speeding up enumeration algorithms and its application for matroid bases. In *Proceedings of the 5th Annual International Conference on Computing and Combinatorics (COCOON '99)*. T. Asano (Eds.), Lecture Notes in Computer Science, Vol. 1627, Springer, Tokyo, Japan, 349–359.
[33] G. Bernardini, A. Conte, G. Gourdel, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Punzi, L. Stougie, and M. Sweering. 2023. Hide and mine in strings: Hardness, algorithms, and experiments. *IEEE Transactions on Knowledge and Data Engineering* 35, 6 (2023), 5948–5963.
[34] G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Rosone, and M. Sweering. 2021. Combinatorial algorithms for string sanitization. *ACM Transactions on Knowledge Discovery from Data* 15, 1 (2021), Article 8, 1–34.
[35] P. A. Pevzner, H. Tang, and M. S. Waterman. 2001. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences* 98, 17 (2001), 9748–9753.
[36] E. W. Myers. 1995. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology* 2, 2 (1995), 275–290.
[37] O. Kupferman and G. Vardi. 2016. Eulerian paths with regular constraints. In *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS '16)*. Leibniz International Proceedings in Informatics (LIPIcs), Vol. 58, Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
[38] S. Hannenhalli, W. Feldman, H. F. Lewis, S. S. Skiena, and P. A. Pevzner. 1996. Positional sequencing by hybridization. *Computer Applications in the Biosciences* 12, 1 (1996), 19–24.
[39] A. Ben-Dor, I. Pe'er, R. Shamir, and R. Sharan. 2002. On the complexity of positional sequencing by hybridization. *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology* 8, 4 (2002), 361–371. DOI:
[40] H. Maserrat, J. Pei. 2010. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*. ACM, New York, NY, 533–542.
[41] F. Dorn, H. Moser, R. Niedermeier, and M. Weller. 2013. Efficient algorithms for Eulerian extension and rural postman. *SIAM Journal on Discrete Mathematics* 27, 1 (2013), 75–94.

[42] G. Bernardini, H. Chen, G. Loukides, S. P. Pissis, L. Stougie, and M. Sweering. 2022. Making de Bruijn graphs Eulerian. In *Proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM '22)*. Leibniz International Proceedings in Informatics (LIPIcs), Vol. 223, Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
[43] G. Navarro. 2016. *Compact Data Structures: A Practical Approach* (1st ed.). Cambridge University Press, Cambridge, UK.
[44] R. E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1, 2 (1972), 146–160.
[45] A. Bernstein, M. Probst, and C. Wulff-Nilsen. 2019. Decremental strongly-connected components and single-source reachability in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC '19)*. ACM, New York, NY, 365–376.
[46] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. 1999. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications* 20, 3 (1999), 720–755.
[47] P. Ferragina and G. Navarro. *Pizza & Chili Corpus—Compressed Indexes and Their Testbeds*. Retrieved from http://pizzachili.dcc.uchile.cl/texts.html

Received 20 August 2024; revised 1 July 2025; accepted 7 October 2025