Fast Assessment of Eulerian Trails

Alessio Conte[1], Roberto Grossi[1], Grigorios Loukides[2], Nadia Pisanti[1], Solon P. Pissis[3], and Giulia Punzi[1]

[1]Università di Pisa, Pisa, Italy, {conte,grossi,pisanti}@di.unipi.it, giulia.punzi@phd.unipi.it
[2]King's College London, London, UK, grigorios.loukides@kcl.ac.uk
[3]CWI and Vrije Universiteit, Amsterdam, The Netherlands, solon.pissis@cwi.nl

Abstract

Given a directed multigraph $G = (V, E)$, with $|V| = n$ nodes and $|E| = m$ edges, and an integer z, we are asked to assess whether the number $\#ET(G)$ of node-distinct Eulerian trails of G is at least z; two trails are called *node-distinct* if their *node* sequences are different. This problem has been very recently posed by Bernardini et al. [ALENEX 2020]. It can be solved in $\mathcal{O}(n^\omega)$ arithmetic operations by applying the well-known BEST theorem, where $\omega < 2.373$ denotes the matrix multiplication exponent. The algorithmic challenge is: *Can we solve this problem faster for certain values of m and z?* Namely, we want to design a combinatorial algorithm for assessing $\#ET(G) \geq z$ that does not resort to the BEST theorem and has a cost that is predictably bounded as a function of m and z. We address this challenge by providing an algorithm requiring $\mathcal{O}(m \cdot \min\{z, \#ET(G)\})$ time. This gives a time-optimal algorithm for $z = \mathcal{O}(1)$ or for $\#ET(G) = \mathcal{O}(1)$. The impact of this theoretical result on real-world graphs is striking: a simple implementation of our algorithm takes tens of seconds to assess $\#ET(G) \geq z$, whereas a highly-optimized implementation of the BEST theorem requires more than 12 hours.

The most surprising consequence of our techniques for directed graphs is that they extend straightforwardly to undirected graphs, for which the underlying counting problem is #P-complete [Brightwell and Winkler, ALENEX/ANALCO 2005]: we provide an algorithm for assessing $\#ET(G) \geq z$ in an undirected multigraph G requiring time polynomial in m and in z.

Bernardini et al. reduced the following problem on strings to the aforementioned assessment problem for directed graphs: Given a string T and an integer z, find the largest d such that there exist at least z strings having the same multiplicities of length-d substrings as T. The authors proposed an $\mathcal{O}(|T|^\omega \log d)$-time algorithm to solve this problem by applying the BEST theorem to compute the largest d for which $\#ET(G) \geq z$, where G is the order-d de Bruijn graph of T. Our algorithm for directed graphs can directly solve the same problem in $\mathcal{O}(|T| z \log d)$ time.

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 cycle is an Eulerian trail that starts and ends on the same node in V. The perhaps most fundamental algorithmic question related to Eulerian trails is whether we can efficiently identify one of them. Hierholzer's paper [6] can be employed for this purpose to get a linear-time algorithm. A related question is counting Eulerian trails: answering this is $\#P$-complete for undirected graphs [7], while for directed graphs the number of Eulerian trails can be computed in polynomial time using the BEST theorem [28], named after de Bruijn, van Aardenne-Ehrenfest, Smith and Tutte.

Two trails are called *node-distinct* if their *node* sequences are different. Bernardini et al. posed the following fundamental problem in [4], which surprisingly had not been previously asked to the best of our knowledge: Given a directed multigraph $G = (V, E)$, with $|V| = n$ nodes and $|E| = m$ edges, and a positive integer z, assess whether the number $\#ET(G)$ of node-distinct Eulerian trails of G is at least z.[1]

This problem can be solved in $\mathcal{O}(n^\omega)$ arithmetic operations as follows [4, 19], where $\omega < 2.373$ denotes the matrix multiplication exponent [13, 30]. (The underlying assumption is that G is Eulerian,[2] that is, the indegree equals the outdegree in each node, possibly except for the source and the target of the trail.) Let $A = (a_{uv})$ be the adjacency matrix of G allowing both $a_{uv} > 1$ (multi-edges) and $a_{uu} > 0$ (self-loops), and let r_u denote the maximum between the indegree and the outdegree of u, where the edges are counted with multiplicity. We can apply the BEST theorem [7] using its formulation for directed multigraphs,

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

where $L = (l_{uv})$ is the $n \times n$ matrix with $l_{uu} = r_u - a_{uu}$ and $l_{uv} = -a_{uv}$. We can single out two factors in Equation (1), the determinant $\det L$ and the ratio $F = \prod_{u \in V} (r_u - 1)! (\prod_{(u,v) \in E} a_{uv}!)^{-1}$ of factorials. We can assess $\#ET(G) \geq z$ when $F \geq z$; otherwise, we need to check if $\det L \geq z/F$. Unfortunately, there is no guarantee that $F \geq z$, and even worse, we have that $F < 1$ for some instances, thus resorting to $\mathcal{O}(n^\omega)$ arithmetic operations for $\det L$. This makes a difference of several orders of magnitude as these arithmetic operations can be expensive. On typical instances in experiments, the computation of F takes a few seconds versus several hours taken to compute $\det L$ with state-of-the-art (sparse) matrix multiplication libraries and other tricks.

Motivated by the cost of applying the BEST theorem, and, in particular, by the fact that it requires matrix multiplication, we address the following algorithmic challenge [4, Final Remarks]: how to design a combinatorial algorithm for assessing $\#ET(G) \geq z$ that does not resort to the BEST theorem and has a cost that is predictably bounded as a function of m and z. This challenge is well-founded, as we do not need to count $\#ET(G)$ to address it.

Our Results and Techniques We first present a simple and non-trivial algorithm to facilitate the reader's comprehension. The main idea consists in 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. This lower bound is then progressively refined by considering any arbitrarily chosen component, and improving its contribution by employing some novel structural properties of strongly connected components of Eulerian graphs. Our method conceptually provides a recursive enumeration approach whose calls list the first z node-distinct Eulerian trails in $\Theta(m^2 \cdot \min\{z, \#ET(G)\})$ time. However, we

[1] Counting node-distinct Eulerian trails is more challenging than counting all Eulerian trails; see Section 2.

[2] If G is not Eulerian, the answer is trivially negative for any non-zero z.

improve upon that, as our lower-bound driven algorithm does not necessarily perform all the recursive calls to assess whether or not $\#ET(G) \geq z$. We also provide a proof-of-concept implementation of our algorithm merely to show the impact of our technique: it can be significantly faster than applying the BEST theorem on real-world graphs, even for relatively large z values.

The above algorithm may require quadratic time per Eulerian trail because each call might require $\mathcal{O}(m)$ time. We refine it to bring its complexity down by 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 strongly connected components change when we start removing edges (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 $\mathcal{O}(m)$ unsuccessful steps. We thus reduce the time by a factor of m, which gives a time-optimal algorithm for $z = \mathcal{O}(1)$ or for $\#ET(G) = \mathcal{O}(1)$. Our result is formalized as follows.[3]

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 $\mathcal{O}(m \cdot \min\{z, \#ET(G)\})$ time.*

The most surprising consequence of our techniques for directed graphs is that they extend straightforwardly to undirected graphs, for which the underlying counting problem is #P-complete:

Theorem 2. *Given an undirected multigraph $G = (V, E)$, with $|E| = m$, and an integer z, assessing $\#ET(G) \geq z$ can be done in time polynomial in m and in z.*

Let us remark that our algorithms can potentially run asymptotically faster than the worst-case bounds given above. Under suitable assumptions and values of z, it is possible that they run in less than Constant Amortized Time (CAT) per solution (cf. [25]). In principle, this implies that, under suitable assumptions, assessment might be intrinsically more efficient than counting.

Applications to String Algorithms Text indexing is a fundamental and well-studied problem in computer science [3, 8, 10–12, 14, 16–18, 20, 21, 29]. String processing applications (see [2, 15] for comprehensive reviews) require fast access to the content (the substrings) of the input string. These applications rely on indexing data structures, which typically arrange the string suffixes lexicographically in an ordered tree [29] or in an ordered array [20].

Bernardini et al. [4] introduced the concept of reverse-safe data structures. A data structure is called *z-reverse-safe* (shortly, z-RSDS) if there exist at least z datasets with the same set of answers as the ones encoded by the data structure. The ultimate aim of an RSDS is to make the reconstruction of the input dataset sufficiently unlikely in order to protect data privacy but, at the same time, to store as many answers to useful queries as possible in order to support applications. In addition, the RSDS should be constructed efficiently and have size close to the size of the original dataset it encodes.

Bernardini et al. also proposed an algorithm which constructs a z-RSDS for indexing a string T of length $|T|$, for an integer $z > 1$, that has size $\mathcal{O}(|T|)$ words and answers decision and counting pattern matching queries of length at most d optimally, where d is maximal for any z-RSDS. Their construction algorithm requires $\mathcal{O}(|T|^\omega \log d)$ time, and it essentially boils down to computing the largest $d \in [1, |T|]$ for which the number $\#ET(G)$ of node-distinct Eulerian trails in the order-d de Bruijn graph (dBG) G of T is at least z using the BEST theorem. An order-d dBG over T has a node for every distinct length-$(d-1)$ substring of T and an edge connecting two nodes if the corresponding substrings occur successively in T. Thus, $\#ET(G)$ corresponds to the number of distinct strings having the same multiplicities of length-d substrings as T. See Figure 1 for an

[3] We assume throughout that basic arithmetic operations take constant time, which is the case when $z = \mathcal{O}(\text{poly}(m))$.

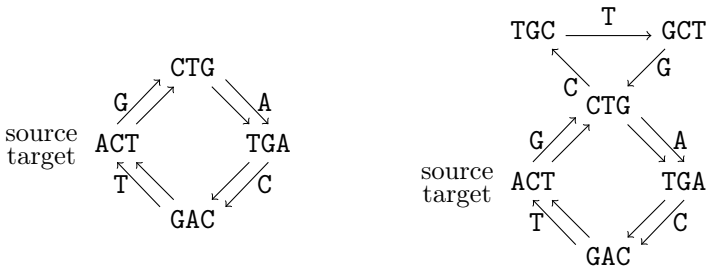


Figure 1: Left: de Bruijn (multi)graph G_1 with 2^4 Eulerian trails, all corresponding to string ACTGACTGACT, thus $\#ET(G_1) = 1$ as there is only one node-distinct Eulerian trail. Right: de Bruijn graph G_2 with 2^5 Eulerian trails but $\#ET(G_2) = 2$, i.e. ACTGCTGACTGACT and ACTGACTGCTGACT.

example: even if more than $\#ET(\cdot)$ trails exist, some of these trails produce *the same sequence of nodes*, and thus *the same string*.

By plugging Theorem 1 in the construction algorithm of [4], we obtain an $\mathcal{O}(|T|z \log d)$-time algorithm for constructing the same z-RSDS for string T. This is because the number m of edges in the order-d dBG of string T is exactly $|T| - d + 1 = \mathcal{O}(|T|)$. We can formally state this result, which gives a nearly linear-time algorithm for $z = \mathcal{O}(1)$, as follows.

Theorem 3. *Given a string T over an integer alphabet $\Sigma = \{1, 2, \ldots, |T|^{\mathcal{O}(1)}\}$ and an integer z, there exists an $\mathcal{O}(|T|z \log d)$-time algorithm to construct an $\mathcal{O}(|T|)$-sized z-RSDS over T for a maximal d that answers decision and counting pattern matching queries, for any pattern P of length $|P| \leq d$, in the optimal $\mathcal{O}(|P|)$ time per query.*

Let us add that Theorem 1 applied to dBGs can find interesting uses in bioinformatics, as dBGs are a common tool to represent genome sequencing data (cf. [24]) and are huge in size. Therein, the (number of) node-distinct Eulerian trails between a given source and target node would correspond to (the number of) alternative sequences in between these nodes. In RNA-Seq data, assessing whether $\#ET(\cdot) \geq z$ between a pair of nodes corresponding to suitable exons, can be useful for isoforms quantification [23]. In genome assembly (cf. [22]), assessing whether $\#ET(\cdot) \geq z$ between a pair of nodes is a useful task for understanding the repetitive structure of the genome, which is the main obstacle for good quality assembly; e.g., in [19] counting is performed using the BEST theorem.

Paper Organization Section 2 introduces the basic definitions and notation used throughout. In Section 3, we prove the combinatorial properties of Eulerian graphs which form the basis of our techniques. In Section 4, we present the simple $\mathcal{O}(m^2 \cdot \min\{z, \#ET(G)\})$-time algorithm. This algorithm is refined to our main result described in Section 5. We extend our techniques to undirected graphs in Section 6. Any materials omitted from Sections 3, 4, and 5 are included in Sections 7 and 8. Section 9 presents proof-of-concept experimental results using real-world de Bruijn graphs showcasing the importance of our main result for directed graphs. The paper is concluded in Section 10.

2 Definitions and Notation

Consider a directed graph $G = (V, E)$ with multi-edges and self-loops, and let $|V| = n, |E| = m$, where the edges are counted with multiplicity. A *trail* over G is a sequence of adjacent distinct edges. Two trails are *node-distinct* if their *node* sequences are different. An *Eulerian trail* of G is a

trail that traverses every edge exactly once. We consider *node-distinct* Eulerian trails.[4] 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 outdegree(u) − 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. A *strongly connected component* (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: it has at least one node and there is a trail between every pair of nodes.

Definition 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 (i) $\Delta(s) = 1$, $\Delta(t) = -1$, and $\Delta(u) = 0$ for all $u \in V \setminus \{s, t\}$; or (ii) $\Delta(u) = 0$ for all $u \in V$. In Case (i), G has an *Eulerian trail from s to t*. In Case (ii), G has an *Eulerian cycle*: an Eulerian trail that starts and ends on $s = t$.

3 Structure and Properties of G_{SCC}

The SCCs of a directed Eulerian graph G induce a directed acyclic graph G_{SCC}. Considering this graph, we derive some non-trivial and useful properties, upon which we will heavily rely to design our algorithms for assessing the number of node-distinct Eulerian trails. The omitted proofs can be found in Section 7. 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 $s_0 = s, t_k = t$.*

It follows from Lemma 1 that each 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.

Lemma 2. *Let G be a Eulerian graph with SCCs $C_0, \ldots, C_k$. Then $ET(G) = \prod_{i=0}^{k} ET(C_i)$, where $\prod$ denotes the cartesian product. It follows that 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 3. *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.*

[4] Traditionally, two trails are distinct if their *edge* sequences are different. It is more challenging to consider node-distinct Eulerian trails as they spell different strings in a dBG (see Figure 1), whereas some Eulerian trails might correspond to the same string. 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 size of the graph by $\mathcal{O}(m)$, and the node-distinct Eulerian trails of this new graph are exactly the edge-distinct Eulerian trails of the original graph.

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 Definition 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 Definition 1.

Suppose that $s \neq t$: we are in case (i) of Definition 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 definition.

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 cycle, 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 4. *Let C_i be any SCC of an Eulerian graph with source s_i. Then:*

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

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 new source.

Note the subtle point in the statement of Corollary 4, 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 of Eulerian Trails in Directed Graphs

We first present AssessET, a simple non-trivial algorithm for assessing the number of node-distinct Eulerian trails on a given directed graph, which will be refined in Section 5.

AssessET takes the following parameters as input: first, a weakly connected Eulerian graph $G = (V, E)$ with source s and target t; second, a positive integer threshold z, and third, 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, we place two requirements: $lb(\cdot)$ must be computable in $\mathcal{O}(m)$ time and $lb(\cdot) \geq 1$ must hold.

Given the parameters above, AssessET is able to discern whether $\#ET(G) \geq z$ or not. Specifically, in what follows, we show the following proposition, whose proof is in Section 8.

Proposition 1. *Given graph G, nodes s and t, integer z, and function $lb(\cdot)$,* AssessET *assesses $\#ET(G) \geq z$ in $\mathcal{O}(m^2 \cdot \min\{z, \#ET(G)\})$ time using $\mathcal{O}(mz)$ 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 Lemmas 2 and 3, and Corollary 4, 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 Lemma 2. 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 3 and its Corollary 4. 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: $lb_{ET}(G) = \#ET(G)$.

Regarding the choice of the lower bound function $lb(\cdot)$, the requirements are trivially satisfied by the constant function $lb(\cdot) \equiv 1$. However, we use a better lower bound function, which is given by Lemma 4 (the proof is in Section 8 and experiments comparing the two bounds are in Section 9).

Lemma 4. *For any Eulerian graph G, the function*

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

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

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, and it requires $\mathcal{O}(m)$ time to achieve this (specifically, we make use of [27]).

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 $\langle C_0^j, \ldots, C_{h_j}^j \rangle$ of 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 3; 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 $\mathcal{O}(1)$ 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 $\mathcal{O}(1)$ 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:

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:

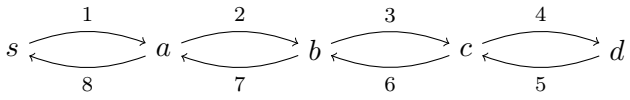


Figure 2: This graph has a single node-distinct Eulerian trail from s to itself, even though all nodes except for s and d are branching.

(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 guess a 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 non-empty, 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 Step 2, then $lb_{ET} \geq z$ and we output YES.

When $lb(\cdot)$ always returns 1, AssessET makes $\mathcal{O}(mz)$ calls to compute the SCCs, of $\mathcal{O}(m)$ time each, as it essentially lists z Eulerian trails one by one. However, when $lb(\cdot) > 1$, a lot of these calls are avoided, as we multiply the lower bounds. Thus, the algorithm becomes significantly faster in practice; see Section 9, where we show that indeed a lot of these calls are avoided on real-world graphs. A formal description of AssessET, given by its pseudocode (Algorithm 1), is provided in Section 8.

5 Improved Assessment Algorithm

We may think of AssessET as a recursive computation (handled explicitly with pop/push on a stack) having the drawback that it makes $\mathcal{O}(mz)$ recursive calls. To try and speed up the process, one could resort to existing decremental SCC algorithms [5], although these tend to add (poly)logarithmic factors, and do not immediately yield improvements unless further amortization is suitably designed.

We use a different approach, reducing the number of calls to $\mathcal{O}(z)$ 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 an efficient combinatorial procedure.

5.1 Introducing Function BranchingSource

Consider the SCC C_i chosen in AssessET, and its source s_i. 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, we have at least two calls by Lemma 3. The issue comes when s_i has just one out-neighbor, as illustrated in Figure 2: 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 3 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 2, 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 $\mathcal{F}$. 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 $\mathcal{O}(m^2)$ time as it simulates what AssessET does until a branching source is found. The challenge is to implement it in $\mathcal{O}(m)$ time. Armed with that, we can modify AssessET and get ImprovedAssessET, where we guarantee in $\mathcal{O}(m)$ time that *the source s_i is always branching*. A formal description of ImprovedAssessET, given by its pseudocode (Algorithm 2), is provided in Section 8. The modification is just few lines, once BranchingSource is available, so we do not provide a detailed description of the pseudocode.

5.2 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 $\mathcal{O}(|E(C)|)$ time, and traverse T asking at each node u whether there is an alternative trail T' branching at u.

Swap Edges

Definition 2. 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 let (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 2), for *any* trail $T \in ET(C)$.

Lemma 5. *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 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; on the other hand, 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 5, BranchingSource examines each $C \in f$: it can test whether C is trivial ($\#ET(C) = 1$), or it can find 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 of SCCs $\mathcal{C} = \text{ComputeSCC}(C \setminus T_u)$. Note that u becomes the source of an SCC in $\mathcal{C}$ and, moreover, 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. Since 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 the following lemma.

Lemma 6. *Given any tuple f in $\mathcal{F}$ and the set of swap edges in the SCCs of f, the function* BranchingSource *takes $\mathcal{O}(m)$ 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 3. Since every swap edge generates *at least one new Eulerian trail*, if we can find all swap edges in $\mathcal{O}(m)$ 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 Figure 3b, 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.

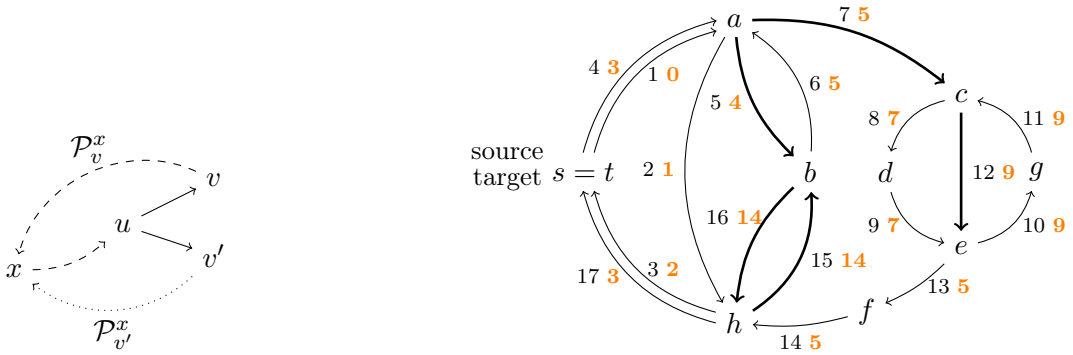
Lemma 7. *Let C be an SCC, and let T with prefix $T_u \cdot (u, v)$ be one of its Eulerian trails. Edge (u, v'), for $v' \neq v$, is a swap edge if and only if there is a trail from v' to u (i.e., u is reachable from v') in $C \setminus T_u$.*

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 $x = u$.
- Otherwise, some edges needed for v' to reach u have been used by $\mathcal{P}^u_v$. Consider the first edge (in the traversal order) of $\mathcal{P}^u_v$ that shares its head with an edge of T'', and let x be their head (see Figure 3a).

Let $\mathcal{P}^x_v$ be the prefix of trail $\mathcal{P}^u_v$ up to the first occurrence of node x. Similarly, let $\mathcal{P}^x_{v'}$ be the prefix of T'' up to the first occurrence of x. The Eulerian trail T will then be of the form $T_u \cdot (u, v) \cdot \mathcal{P}^x_v \cdot \mathcal{P}^u_v \setminus \mathcal{P}^x_v \cdot (u, v') \cdot \mathcal{P}^x_{v'} \cdot T'' \setminus P^x_{v'}$. Consider now $T' = T_u \cdot (u, v') \cdot \mathcal{P}^x_{v'} \cdot \mathcal{P}^u_v \setminus \mathcal{P}^x_v \cdot (u, v) \cdot \mathcal{P}^x_v \cdot T'' \setminus P^x_{v'}$, obtained from T by swapping $(u, v) \cdot \mathcal{P}^x_v$ and $(u, v') \cdot \mathcal{P}^x_{v'}$, 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 of Figure 3b: (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 8), comparing these integers allows us to check if a given edge is a swap edge in constant time.



(a) (b)

Figure 3: **(a)** Setting in which T'' (dotted) intersects $\mathcal{P}_v^u$ (dashed), and the choice of x. **(b)** Example of an Eulerian graph with source and target $s = t$. The black (left) numbers on the edges are the Eulerian trail numbers *etn* for a given trail T; the bold orange (right) ones are the disconnecting indices *di*s. Swap edges are in bold.

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 6 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 4. Given an edge $(u, v') \in E(C)$, its *disconnecting index* is

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

Figure 3b illustrates an example where the following property can be checked by inspection.

Lemma 8. *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 7, 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 4, 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. □

We now show how to compute the disconnecting indices of an SCC C in $\mathcal{O}(|E(C)|)$ time.

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

We proceed by reconstructing T backwards. That is, we conceptually start from an empty graph, and we add edges from T, one edge 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, denoted by BRIDGES: intuitively this will contain 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, prove its correctness, and show that all invariants hold.

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 yet (i.e., $tr(u) = false$), we add (u, v) to BRIDGES and set $tr(u) = true$. If $\ell = m$, then 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 9. *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 $\mathcal{O}(|E(C)|)$ time and space.*

We arrive at the following 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 $\mathcal{O}(m \cdot \min\{z, \#ET(G)\})$ time.*

6 Assessment of Eulerian Trails in Undirected Graphs

In this section, we show how our assessment technique can be extended to the case of undirected graphs. To this end, let $G = (V, E)$ be an undirected multigraph, and let z be a positive integer. Analogously as before, by $\#ET(G)$ we denote the number of node-distinct Eulerian trails of G: we want to assess whether this number is at least z.

6.1 Definitions and Notation

Since G is undirected, we have no distinction between out- and in-degree, as $(u, v) = (v, u) \in E$. Given a node $u \in V$, we thus define its *degree* $\delta(u)$ as the number of edges of the form (u, v), and we define its *neighbors* as $N(u) = \{v \in V | (u, v) \in E\}$. Once again we employ the notation $N_C(u)$ for $N(u) \cap C$ with $C \subseteq V$.

First, recall that an undirected graph is Eulerian with source s and target t if it is connected (there is a path between any pair of vertices), and either (i) $\delta(s)$ and $\delta(t)$ are odd, and $\delta(u)$ is even for $u \in V \setminus \{s, t\}$; or (ii) $\delta(u)$ is even for all $u \in V$. In the first case, G has an Eulerian trail from s to t, while in the second it has an Eulerian cycle, just like in the directed case. Note that, given a directed Eulerian graph, removing directions on the edges yields an undirected Eulerian graph.

In what follows, we show that AssessET can be extended to undirected graphs. To this end, we retrace the steps we performed for the directed case, applying suitable changes when necessary.

6.2 Structural Properties

As with directed graphs, we begin by studying the structural properties of the graph, and what they entail. In the undirected case, strong connectivity naturally generalizes to *2-edge-connectivity*. A connected undirected graph G is called *2-edge-connected* if the removal of any edge does not disconnect the graph: $G \setminus \{e\}$ is connected, for each $e \in E$. On the contrary, an edge whose removal increases the number of connected components of G is called a *bridge*. Undirected bridges will serve the same purpose as that of directed ones in AssessET. The maximal 2-edge-connected subgraphs of G are called *2-edge-connected components* (2ECCs), and they can be easily obtained in linear time in the size of G [26]. These components generalize SCCs to the undirected case: indeed, our first result shows that the 2ECCs of an Eulerian graph form a chain-like structure.

Lemma 10. *Let G be an undirected Eulerian graph, with 2ECCs $C_0, \dots, C_k$, source $s \in C_0$ and target $t \in C_k$. Then, each C_i is Eulerian with source s_i and target t_i.*

Furthermore, G has a chain structure*, i.e. it is of the form $C_0 - C_1 - \dots - C_k$, where the single edge $e_i = (t_i, s_{i+1})$ between is a bridge between C_i and C_{i+1}, and $s = s_0, t = t_k$.*

Proof. Let $C_0, \dots, C_k$ be the 2ECCs of G, with $s \in C_0$ and $t \in C_k$. First, note that, by definition, all edges e_i in $G \setminus C_0, \dots, C_k$ must be bridges. Thus, the chain structure will immediately imply that the edges connecting the C_i's must be bridges. Let us start with a preliminary remark: given C_i, and a bridge $e = (u, v)$ with $u \in C_i$ and $v \in C_j \neq C_i$, there cannot be any trail in $G \setminus e$ from v to a node of C_i. Indeed, if there was such a trail, then e would not be a bridge, since its removal would not disconnect $C_i \cup C_j$, and hence it would not disconnect G. We will now proceed to prove the chain structure, and then we will conclude the proof by showing that each component is Eulerian.

First, let us consider C_i with $i \neq 0, k$, and let us show that there exist exactly two edges (bridges) $e_{i-1} = (u, s_i), e_i = (t_i, v) \in E$ such that $s_i, t_i \in C_i$ and $u, v \notin C_i$; hence, only nodes s_i and t_i (possibly $s_i = t_i$) of C_i have a neighbor outside of C_i: one connecting with C_{i-1}, and one with C_{i+1}. We first show that there are at least two such edges. We know that we have at least one Eulerian trail from $s \in C_0$ to $t \in C_k$; since $i \neq 0, k$, during the traversal we must have one edge entering C_i for the first time, and one edge leaving for the last time, to be able to reach t. In order to show that the edges are exactly two, assume by contradiction that, next to $e_{i-1} = (u, s_i)$ and $e_i = (t_i, v)$, there is at least another edge $f = (x, y)$ in E, with $x \in C_i$ and $y \notin C_i$, such that $f \neq e_{i-1}, e_i$. By hypothesis, there exists an Eulerian trail of G from s to t. Since $i \neq 0$, one of these edges will be traversed by the trail when arriving for the first time at a node of C_i; wlog, let e_{i-1} be this edge. The trail will then need to traverse both e_i, f. Again wlog, assume that it traverses e_i first; then, to be able to traverse f, we must have a trail from $v \notin C_i$ back to $x \in C_i$ which does not use the edges of the Eulerian trail so far (which contain e_i). This contradicts the remark above. Note that the two edges $(u, s_i), (t_i, v) \in E$ such that $s_i, t_i \in C_i$ have $u \in C_j, v \in C_h$ with $j \neq h$ (that is, they lead to two different components), otherwise they would not be bridges.

To conclude the proof of the chain structure, it is left to show that for $i = 0, k$ there is exactly one bridging edge with one endpoint in C_i, and the other outside. Consider first $i = 0$; by contradiction

let $e_i = (t_i, v)$ and $f = (x, y)$ be two distinct edges with $t_i, x \in C_i$ and $v, y \notin C_i$. All Eulerian trails start in C_i since it contains the source s; consider a specific Eulerian trail, and let e_i be the first edge traversed of the two. Then, to be able to traverse f, there needs to be a trail from v to $x \in C_i$ that does not use e_i, a contradiction. Finally, consider $i = k$, and let again by contradiction $e_{i-1} = (u, s_i)$ and $f = (x, y)$ be two distinct edges with $s_i, x \in C_i$ and $u, y \notin C_i$. Symmetrically, all Eulerian trails end in C_i, since it contains the target t. Consider a specific Eulerian trail, and let e_{i-1} be the first edge traversed of the two (representing also the first time the trail reaches a node of C_i). Then again, to be able to traverse f and then reach the target $t \in C_i$, there needs to be a trail from $y \notin C_i$ to t that does not use edge f, a contradiction.

For each i, let $e_i = (t_i, s_{i+1})$ be the bridge connecting C_i and C_{i+1}, and let $s_0 = s, t_0 = t$. We are now ready to prove that each C_i is Eulerian with source s_i and target t_i. Consider first C_i for $i \neq 0, k$; let us denote with $\delta_i(u)$ the degree of node u in the induced subgraph $G[C_i]$. Since G is Eulerian and $s, t \notin C_i$, we have that $\delta(u)$ is even for all $u \in V(C_i)$. In $G[C_i]$ we have removed the bridges, thus the only nodes whose degree changes are the extremes of the two bridges touching C_i, s_i and t_i. A case study is needed:

- $s_i \neq t_i$: $\delta_i(s_i) = \delta(s_i) - 1 = \text{even}-1$, $\delta_i(t_i) = \delta(t_i) - 1 = \text{even}-1$, while all others stay the same. Thus, $\delta_i(s_i)$ and $\delta_i(t_i)$ are odd, while $\delta_i(u) = \delta(u)$ for all $u \in C_i \setminus \{s_i, t_i\}$ are even.
- $s_i = t_i$: $\delta_i(s_i) = \delta_i(t_i) = \delta(s_i) - 2 = \text{even}-2$, while all others stay the same. Thus, all nodes have even degree in $G[C_i]$.

The study for C_0, C_k is analogous, noting that if $s, t \in C_0$ then $C_0 = G$ is Eulerian. □

We call the above structure the *chain decomposition* of G. Each Eulerian trail of G must traverse the entire sequence $C_0, \ldots, C_i$ before crossing the bridge $e_{i+1} = (t_i, s_{i+1})$. This is because, for each component, only one of the two bridges leads to $s \in C_0$, and only the other to $t \in C_k$. Consequently, we have the following lemma, similar to Lemma 2.

Lemma 11. *Let G be an Eulerian graph with chain decomposition $C_0, \ldots, C_k$. Then $ET(G) = \prod_{i=0}^{k} ET(C_i)$, where $\prod$ denotes the cartesian product. It follows that the number of trails of G is the product of the number of trails of its SCCs.*

Given the chain decomposition, we can thus focus on an individual component C_i or, equivalently, assume, wlog, that the Eulerian graph is 2-edge-connected. The following lemma parallels Lemma 3, stating that each non-bridge edge towards a distinct neighbor gives us a choice for a different Eulerian trail.

Lemma 12. *Let C be a 2-edge-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 degree of nodes in C'. Note that, by definition, the 2-edge-connectivity of C implies that C' is connected. We thus only have to check the parity of the degrees.

Suppose that $s = t$, and all nodes have then even degree. Then, $\delta'(s) = \delta(s) - 1$ is odd, but also $\delta'(u) = \delta(u) + 1$ is odd. All other nodes remain with even degree. This proves that C' is Eulerian.

Suppose that $s \neq t$; then only s, t have odd degree, while all other nodes have even degree. Here we consider three cases.

Case (a): $u \neq t, s$. We have $\delta'(s) = \delta(s) - 1$ even, $\delta'(t) = \delta(t)$ odd, while $\delta'(u) = \delta(u) + 1$ becomes odd. All other nodes stay unchanged; leading us to have an Eulerian graph.

Case (b): $u = t$. We have $\delta'(s) = \delta(s) - 1$ even, $\delta'(u) = \delta(u) + 1$ even, and all other nodes unchanged. Furthermore, C' is connected. These two hypotheses directly imply that there is an Eulerian cycle.

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. □

Finally, from Lemma 12 we derive the final corollary, similar to Corollary 4.

Corollary 5. *Let C_i be any component of an Eulerian graph chain decomposition with source s_i. Then:*

$$ET(C_i) = \bigcup_{u \in N_{C_i}(s_i)} ET(C_i \setminus (s_i, u)).$$

Therefore, 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 new source.

Once again, as in the directed case, 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, because of the chain structure.

6.3 Assessment Algorithm

Given the properties introduced in the previous section, we can outline an assessment algorithm AssessUndirectedET for the undirected case which retraces AssessET. The main difference is that, instead of computing the SCCs with ComputeSCC, we compute the chain decomposition of G with the function ChainDecomp. The described modification does not affect the total complexity, as ChainDecomp can be performed in linear time. We obtain the following proposition:

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

Note that, at any moment of the computation, we can see G as a *mixed graph*. A mixed graph is a graph in which some edges have directions, and some do not. In our algorithm, we start from an undirected graph. Each time we consider a neighbor u of $s_i \in C_i$, we can assign the outgoing direction $s_i \to u$ to the edge (s_i, u) in the (implicit) recursion where we choose that edge as the next one. Note that this does not cause any ambiguity: the fact that we assigned a direction to the edge in some recursion subtree does not preclude us from assigning it the opposite direction in some other subtree.

As in the directed case, we can employ the trivial lower bound function $lb(\cdot) \equiv 1$, or we can use a more refined lower bound:

$$lb(G) = 1 + \sum_{v \in V(G) : |N(v)| \geq 4} (|N_G(v)| - 3).$$

This bound differs from that of Equation 2, as we no longer have a distinction between in- and out-edges.

The reader can probably share the intuition that AssessUndirectedET can also be refined to achieve $\mathcal{O}(m \cdot \min\{z, \#ET(G)\})$ time in a similar way to ImprovedAssessET; this improvement is not detailed here as it would not add any further algorithmic insight.

We arrive at the following result.

Theorem 2. *Given an undirected multigraph $G = (V, E)$, with $|E| = m$, and an integer z, assessing $\#ET(G) \geq z$ can be done in time polynomial in m and in z.*

7 Proofs Omitted from G_{SCC} Structure and Properties

Proof of Lemma 1

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 $v \in C_k$, 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 Definition 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 Definition 1.
3. Case $t \in C_i = C_k$, $s \notin C_k$ is symmetric to case 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 Definition 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 Definition 1.

□

Proof of Lemma 2

Proof. This follows immediately from the chain structure of G_{SCC}: every Eulerian trail of G must start in $s \in C_0$, end in $t \in C_k$, and for $0 \leq i \leq k-1$, traverses an Eulerian trail of the whole C_i before moving to C_{i+1}. □

Proof of Corollary 4

Proof. This follows from Lemma 3 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. □

8 Material Omitted from Assessment Algorithms

Proof of Lemma 4

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 $G' = G \setminus T'$. 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 3, 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 Equation 2. □

Algorithm AssessET

We provide a step-by-step description of AssessET in Algorithm 1, and its analysis.

Algorithm 1 (AssessET)

procedure AssessET($G = (V, E)$, z, $lb(\cdot)$)
  $C_0, \ldots, C_k \leftarrow$ ComputeSCC(G) ▷ Only considers non-trivial SCCs
  $f \leftarrow \langle C_0, \ldots, C_k \rangle$
  if f is empty **then** $lb_{ET} \leftarrow 1$
  else Stack.push(f) ▷ Initialization
    $lb_{ET} \leftarrow \prod_{j=0}^{k} lb(C_j)$
  while $lb_{ET} < z$ **do**
    if Stack.IsEmpty() **then Output NO**
    $f \leftarrow$ Stack.pop()
    $lb_{ET} \leftarrow lb_{ET} - lb_{ET}(f)$ ▷ $lb_{ET}(f) = \prod_{i=1}^{|f|} lb(f[i])$
    Guess i; let $C_i = f[i]$ and s_i be its source ▷ $f[i]$ is the i-th SCC of f
    Remove C_i from f
    for all $u \in N^+_{C_i}(s_i)$ **do**
      $\mathcal{C} \leftarrow$ ComputeSCC($C_i \setminus (s_i, u)$)
      if $f \cdot \mathcal{C}$ is not empty **then** ▷ $f \cdot \mathcal{C}$: f with each SCC of $\mathcal{C}$ appended
        Stack.push($f \cdot \mathcal{C}$)
        $lb_{ET} \leftarrow lb_{ET} + lb_{ET}(f \cdot \mathcal{C})$
      else $lb_{ET} \leftarrow lb_{ET} + 1$
  Output YES

Correctness The correctness of AssessET follows from Lemmas 2 and 3 and Corollary 4.

Proof of Proposition 1

Proof. We now analyze the time and space complexity of AssessET, which will allow us to complete the proof of Proposition 1.

We aim at showing that growing Stack to size S takes $\mathcal{O}(m^2S)$ time.[5] 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 $\mathcal{O}(m^2S)$ 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 (see Section 9).

First of all, let us remember that each push and pop operation on Stack requires $\mathcal{O}(1)$ time, and that our $lb(\cdot)$ function can always be computed in time linear in m. Finally, remember that also ComputeSCC requires $\mathcal{O}(m)$ time.

We are now ready for a detailed analysis. By the discussion above, every line up to Line 6 requires $\mathcal{O}(m)$ time. The while loop in Line 7 iterates until our bound lb_{ET} reaches z, and every operation up to Line 13 takes either $\mathcal{O}(1)$ or $\mathcal{O}(m)$ 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 $\mathcal{O}(m)$ time. Then we perform another set of $\mathcal{O}(1)$- or $\mathcal{O}(m)$-time operations, until we exit the loops. Thus, the for loop of Line 13 requires $\mathcal{O}(m \cdot |N^+_{C_i}(s_i)|)$ 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 $\mathcal{O}(m)$ time for each new element. However, if $|N^+_{C_i}(s_i)| = 1$, we have popped f, and spent $\mathcal{O}(m)$ 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 $\mathcal{O}(m)$ time and does not increase the size of Stack, only $\mathcal{O}(h)$ times. In turn, since $h \leq m$, the total time arising from this bad case on a Stack with S elements is $\mathcal{O}(m^2S)$, which implies our claimed bound.

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

Algorithm ImprovedAssessET

We provide a step-by-step description of ImprovedAssessET in Algorithm 2. Here is the difference with AssessET: if we replace lines 11–14 in Algorithm 2 with the instructions

> Guess i; let $C_i = f[i]$ and s_i be its source

we obtain the pseudocode of Algorithm 1. In lines 11–14 in Algorithm 2, after extracting f from Stack, ImprovedAssessET updates f with BranchingSource. If f is empty, it increments lb_{ET} as we have one node-distinct Eulerian trail. Otherwise, it proceeds as AssessET, except that C_i chosen among the SCCs of the updated f having branching source.

[5] This includes the case where we implicitly represent an empty tuple by increasing the bound lb_{ET} (Line 18).

Algorithm 2 (ImprovedAssessET)

```
1: procedure ImprovedAssessET(G = (V, E), z, 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:        f ←BranchingSource(f)
12:        if f is empty then lb_ET ← lb_ET + 1
13:        else
14:            Guess i with branching source s_i; let C_i = f[i]   ▷ f[i] is the i-th SCC of f
15:            Remove C_i from f
16:            for all u ∈ N^+_{C_i}(s_i) do
17:                C ← ComputeSCC(C_i \ (s_i, u))
18:                if f · C is not empty then            ▷ f · C: f with each SCC of C appended
19:                    Stack.push(f · C)
20:                    lb_ET ← lb_ET + lb_ET(f · C)
21:                else lb_ET ← lb_ET + 1
22:    Output YES
```

Proof of Lemma 9

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, its SCCs will have a chain structure, according to Lemma 1. 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 cycle (which may have node repetitions) 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 $\mathcal{O}(|E(C)|)$ 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 $\mathcal{O}(|E(C)|)$ time. Space is also $\mathcal{O}(|E(C)|)$ words of memory, since we only store the graph, the stack Bridges, and, for each node, tr and the maximum etn. □

9 Experimental Results: Proof of Concept

We have implemented algorithm AssessET in `C++`. In what follows, we compare our implementation in terms of runtime efficiency to the application of the BEST theorem for assessing the number of node-distinct Eulerian trails in directed multigraphs (see Equation 1). For the BEST theorem, we used a `C++` implementation, obtained by the authors of [4], which we denote here by BestET. BestET first computes the ratio $F = \prod_{u \in V}(r_u - 1)!(\prod_{(u,v)\in E} a_{uv}!)^{-1}$ of factorials. It gives a positive answer when $F \geq z$; otherwise, it must compute the determinant $\det L$. For the computation of $\det L$, it makes use of the Sparse LU decomposition function of the highly optimized open-source Eigen library (v. 3.3.7) [1], which is based on the algorithm of [9].

In order to generate realistic directed multigraphs, we constructed dBGs of different orders d over a given input string. Specifically, we have used the full sequence of the Y chromosome of the human genome (version GRCh37) whose length is $59,373,566$. All experiments ran on a Desktop PC with an Intel Xeon E5-2640 at 2.66GHz and 128GB RAM.

The results are depicted in Table 1 (fixed $z = 10^5$ and varying d) and Table 2 (fixed $d = 50$ and varying z). Note that we chose $d = 50$ to ensure that these graph instances are sparse and $\det L$ must be computed. Column W denotes the number of times the while loop in Line 7 of AssessET is executed. Column S denotes the number of times Stack.push$(\cdot)$ in Line 16 of AssessET is executed. In column $\det L$, the notation ✓ denotes that $\det L$ had to be computed.

The main observation is that W and S are very small in many instances of both Tables 1 and 2. Thus, AssessET assesses the number of node-distinct Eulerian trails by only few executions of the while loop, being competitive with or faster than BestET. Specifically, Table 1 shows that AssessET is comparable to BestET for dense graph instances and orders of magnitude faster for the sparser instances in which $\det L$ had to be computed. Table 2 shows that AssessET is faster than BestET for sparse graphs for relatively small values of z ($z \leq 1,000$). It also shows that, even for relatively large values of z ($z \leq 7,500$), AssessET is still competitive.

d	$\|V\|$	$\|E\|$	W	S	AssessET Runtime	BestET Runtime	det L
8	16,474	59,373,559	1	4	72.6s	52.3s	✗
10	258,882	59,373,557	0	0	22.5s	53.1s	✗
12	2,938,568	59,373,555	0	0	28.2s	85.4s	✗
14	12,490,769	59,373,553	0	0	36.7s	60.5s	✗
16	17,777,284	59,373,551	0	0	33.5s	$>$ **12h**	✓
18	19,303,644	59,373,549	1	4	199.9s	$>$ **12h**	✓

Table 1: AssessET vs. BestET for $z = 10^5$ and varying d.

z	$\|V\|$	$\|E\|$	W	S	AssessET Runtime	BestET Runtime	det L
10	23,333,491	59,373,517	0	0	32.7s	122.4s	✓
100	23,333,491	59,373,517	0	0	32.7s	122.4s	✓
1,000	23,333,491	59,373,517	0	0	39.3s	122.4s	✓
2,500	23,333,491	59,373,517	1	4	204.7s	122.4s	✓
5,000	23,333,491	59,373,517	1	4	204s	122.4s	✓
7,500	23,333,491	59,373,517	2	6	295.1s	122.4s	✓

Table 2: AssessET vs. BestET for $d = 50$ and varying z.

In Figure 4 we also show that the lb_{ET} function we employ in AssessET grows much faster than the constant function $lb(\cdot) = 1$, as the parameter W grows; we used $d = 8$ and $z = 10^7$.

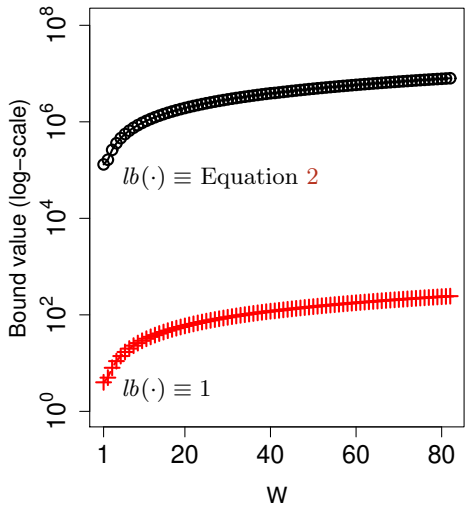


Figure 4: Value of lb_{ET} (in log-scale) using the trivial $lb(\cdot) \equiv 1$ and $lb(\cdot) \equiv$ Equation 2, vs. parameter W for $d = 8$ and $z = 10^7$.

Summarizing, these experimental results showcase the importance of our theoretical findings.

10 Final Remarks

We considered the problem of assessing the number of node-distinct Eulerian trails of a directed multigraph better than with the BEST theorem. In particular, we wanted to make this assessment without resorting to the expensive matrix multiplication, and by taking a cost that is predictably bounded as a function of m and z.

We achieved this goal by developing an algorithm that assesses whether a directed multigraph with m edges has at least z node-distinct Eulerian trails in $\mathcal{O}(mz)$ time. Other than being easily extensible to the edge-distinct case, our algorithm has an attractive property: its number of $\mathcal{O}(m)$-time steps is z in the worst case, but can be significantly smaller in practice, thanks to suitable lower bounding techniques. This property means that our assessment algorithm can potentially run in less than CAT per solution on favorable instances.

The most surprising consequence of our techniques for directed graphs is that they extend straightforwardly to undirected graphs, for which the underlying counting problem is #P-complete. We provided an algorithm that assesses whether an undirected multigraph with m edges has at least z node-distinct Eulerian trails requiring time polynomial in m and in z. This implies that assessment could, under suitable assumptions, be intrinsically more efficient than exact counting, since no #P-complete problem allows for exact counting in polynomial time unless P is equal to NP.

Moreover, our main result for directed graphs yields a nearly linear-time algorithm for constructing a text indexing z-RSDS when $z = \mathcal{O}(1)$.

Acknowledgments We wish to thank Luca Versari (Università di Pisa and Google Zurich) for useful discussions on a previous version of our assessment algorithm for directed graphs.

References

[1] Eigen library. http://eigen.tuxfamily.org, July 2020.

[2] Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and S. Muthukrishnan. 40 years of suffix trees. *Commun. ACM*, 59(4):66–73, 2016.

[3] Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 148–193. ACM, 2014.

[4] Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe data structures for text indexing. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 199–213. SIAM, 2020.

[5] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. 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 2019, page 365–376, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Norman L. Biggs, E. Keith Lloyd, and Robin J. Wilson. *Graph Theory 1736-1936*. Clarendon Press, 1976.

[7] Graham R. Brightwell and Peter Winkler. Counting Eulerian circuits is #P-complete. In Camil Demetrescu, Robert Sedgewick, and Roberto Tamassia, editors, *Proceedings of the Seventh*

Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005, pages 259–262. SIAM, 2005.

[8] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, 72(2):450–466, 2015.

[9] James Weldon Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis Applications*, 20(3):720–755, 1999.

[10] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997.

[11] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[12] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

[13] François Le Gall. Powers of tensors and fast matrix multiplication. In Katsusuke Nabeshima, Kosaku Nagasaka, Franz Winkler, and Ágnes Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014.

[14] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[15] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences–Computer Science and Computational Biology.* Cambridge University Press, 1997.

[16] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009.

[17] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[18] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 756–767. ACM, 2019.

[19] Carl Kingsford, Michael C. Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinform.*, 11:21, 2010.

[20] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[21] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 408–424. SIAM, 2017.

[22] Eugene W. Myers. Toward simplifying and accurately formulating fragment assembly. *J. Comput. Biol.*, 2(2):275–290, 1995.

[23] Robert Patro, Stephen M. Mount, and Carl Kingsford. Sailfish: Alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. *Nature Biotechnology*, 32:462–464, 2014.

[24] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[25] Frank Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

[26] Jens M Schmidt. A simple test on 2-vertex-and 2-edge-connectivity. *Information Processing Letters*, 113(7):241–244, 2013.

[27] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[28] Tatyana van Aardenne-Ehrenfest and Nicolaas G. de Bruijn. *Circuits and Trees in Oriented Linear Graphs*, pages 149–163. Birkhäuser Boston, Boston, MA, 1987.

[29] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973.

[30] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In Howard J. Karloff and Toniann Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 887–898. ACM, 2012.