

Optimal Enumeration of Eulerian Trails in Directed Graphs

Ben Bals  

CWI, Amsterdam, The Netherlands
Vrije Universiteit Amsterdam, The Netherlands

Solon P. Pissis  

The Cyprus Institute, Nicosia, Cyprus

Matei Tinca  

Vrije Universiteit Amsterdam, The Netherlands

Abstract

The BEST theorem, due to de Bruijn, van Aardenne-Ehrenfest, Smith, and Tutte, is a classical tool from graph theory that links the Eulerian trails in a directed graph $G = (V, E)$ with the arborescences in G . In particular, one can use the BEST theorem to count the Eulerian trails in G in polynomial time. For enumerating the Eulerian trails in G , one could naturally resort to first enumerating the arborescences in G and then exploiting the insight of the BEST theorem to enumerate the Eulerian trails in G : every arborescence in G corresponds to at least one Eulerian trail in G . For over two decades, the fastest algorithm for enumerating arborescences in G took $\mathcal{O}(m \log n + n + z_A \log^2 n)$ time, where $n = |V|$, $m = |E|$, and z_A is the number of arborescences in G [Uno, ISAAC 1998]. Since Uno’s algorithm does not lead to an optimal enumeration of Eulerian trails in directed graphs, we were motivated to develop a direct algorithm for this problem.¹

Our central contribution is a remarkably simple algorithm to *directly* enumerate the z_T Eulerian trails in G in the *optimal* $\mathcal{O}(m + z_T)$ time. As a consequence, our result improves on an implementation of the BEST theorem for counting Eulerian trails in G when $z_T = o(n^2)$, and also unconditionally improves the combinatorial $\mathcal{O}(m \cdot z_T)$ -time algorithm of Conte et al. [TKDD 2026] for the same task. Moreover, we show that, with some care, our algorithm can be extended to enumerate Eulerian trails in directed multigraphs in optimal time, enabling applications in bioinformatics and data privacy.

2012 ACM Subject Classification Mathematics of computing $\rightarrow$ Graph algorithms; Mathematics of computing $\rightarrow$ Enumeration

Keywords and phrases graph algorithms, enumeration algorithms, Eulerian trails, Eulerian paths

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.60

Related Version *Full Version:* <https://arxiv.org/abs/2603.12894>

Acknowledgements We thank Jeroen op de Beek for carefully reading an earlier version of this manuscript and for providing valuable feedback.

1 Introduction

Let $G = (V, E)$ be a graph on $n = |V|$ nodes and $m = |E|$ edges. Unless explicitly stated otherwise, we will assume throughout that G is simple and connected when G is undirected, or simple and weakly connected when G is directed. A *trail* in G is a walk in G with no repeated edge. An *Eulerian trail* is a trail that uses each graph edge exactly once. If G

¹ Independently of and concurrently with our work, the running time to enumerate arborescences was improved to the optimal $\mathcal{O}(m + z_A)$ time [11] using a rather intricate technique. We believe a *direct* and *simple* algorithm to enumerate Eulerian trails in directed graphs remains of interest.



has an Eulerian trail, we will call G *Eulerian*. An *Eulerian cycle*, or *Eulerian circuit*, is an Eulerian trail that starts and ends at the same node in G ; that is, it is a graph cycle that uses each edge exactly once.

Euler showed that an undirected graph has an Eulerian cycle if and only if it has no nodes of odd degree. Similarly, an undirected graph has an Eulerian trail if and only if exactly zero or two nodes have odd degree. A directed graph has an Eulerian cycle if and only if for every node $u \in V$ the in-degree $\delta^-(u)$ is equal to the out-degree $\delta^+(u)$. Similarly, a directed graph has an Eulerian trail if and only if for at most one node $u \in V$, $\delta^+(u) - \delta^-(u) = 1$, and for at most one node v , $\delta^-(v) - \delta^+(v) = 1$, and every other node has equal in-degree and out-degree. Thus, *deciding* whether an undirected or directed graph G has an Eulerian trail can be done in $\mathcal{O}(m)$ time. If G is Eulerian, constructing a *witness* Eulerian trail can be done in $\mathcal{O}(m)$ time for both undirected and directed graphs using the algorithm by Hierholzer [13]. In fact, Hierholzer discovered this algorithm before his death in 1871, and it was published posthumously in 1873.

Eulerian trails are arguably among the most basic combinatorial objects. Most undergraduate or graduate courses on graph algorithms provide the above introduction to Eulerian graphs. Motivated by the crossing of the iconic Seven Bridges of Königsberg, this topic often appears in the first lecture. As with many other combinatorial objects, the two most basic remaining questions are *counting* and *enumeration*.

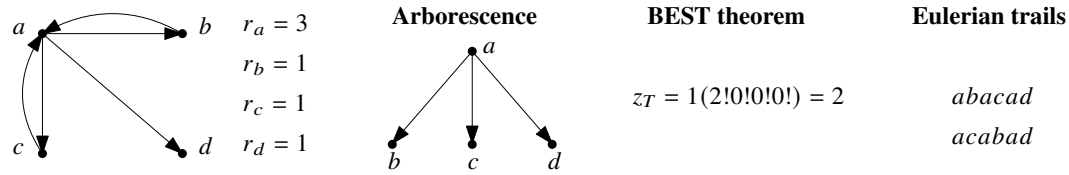
For undirected graphs, Brightwell and Winkler [7] showed that counting Eulerian trails is $\#P$ -complete. Kurita and Wasa [16] showed that enumerating all z_T Eulerian trails in G can be done in the optimal $\mathcal{O}(m + z_T)$ time.

For directed graphs, counting Eulerian trails can be done in polynomial time using the BEST theorem [20], named after de Bruijn, van Aardenne-Ehrenfest, Smith, and Tutte. The BEST theorem relies on the close relationship between Eulerian trails and arborescences. A directed graph is called an *arborescence* if, from a given node s known as the root node, there is exactly one elementary path (meaning without node repetition) from s to every other graph node. Let us fix the source and target nodes of the Eulerian trails in graph G , and denote them by s and t , respectively. Further, let $r_u := \delta^+(u)$ for $u \neq t$, and $r_t := \delta^+(t) + 1$. The BEST theorem can be stated as follows (see also Figure 1):

$$z_T = z_A \cdot \left(\prod_{u \in V} (r_u - 1)! \right), \quad (1)$$

where z_T is the number of Eulerian trails that start at node s , and z_A is the number of arborescences rooted at node s . Thus, z_T is obtained by multiplying z_A by the number of permutations of the edges outgoing from each node ($\prod_{u \in V} (r_u - 1)!$). One can compute z_A as a determinant [6] by Kirchhoff's theorem [17]. Since computing a determinant is equivalent to matrix multiplication [8], counting Eulerian trails in G requires $\mathcal{O}(\text{poly}(m))$ time. However, as matrix multiplication requires $\Omega(n^2)$ time and space, and since the lower-upper (LU) decomposition of the underlying Laplacian matrix is susceptible to numerical issues [10, 12, 14], Conte, Grossi, Loukides, Pisanti, Pissis, and Punzi [9] developed an alternative, combinatorial algorithm for counting z_T exactly in $\mathcal{O}(m \cdot z_T)$ time; in fact, their goal was to *assess* whether $z_T \geq z$, for some input parameter z .

For enumerating Eulerian trails in directed graphs, one could naturally resort to first enumerating the arborescences in G and then exploiting the insight of the BEST theorem to enumerate the Eulerian trails in G : every arborescence corresponds to at least one Eulerian trail. For over two decades, the fastest algorithm for enumerating arborescences in G took $\mathcal{O}(m \log n + n + z_A \log^2 n)$ time, where $n = |V|$, $m = |E|$, and z_A is the number



■ **Figure 1** A directed graph, the arborescence rooted at node a , and the two Eulerian trails starting at a .

■ **Table 1** Complexity landscape of computing Eulerian trails in undirected and directed graphs.

	Undirected		Directed	
	Bound	Reference	Bound	Reference
Decision	$\mathcal{O}(m)$ time	Euler's theorem	$\mathcal{O}(m)$ time	Euler's theorem
Witness	$\mathcal{O}(m)$ time	[13]	$\mathcal{O}(m)$ time	[13]
Counting	#P-complete	[7]	$\mathcal{O}(\text{poly}(m))$ time	BEST theorem
Enumeration	$\mathcal{O}(m + z_T)$ time	[16]	$\mathcal{O}(m + z_T)$ time	[This paper]

of arborescences in G [18]. Since Uno's algorithm does not lead to an optimal enumeration of Eulerian trails in directed graphs, we were motivated to develop a direct algorithm for this problem. Interestingly, the algorithm by Conte et al. can be lifted to an enumeration algorithm with some effort [9, Section 5.1], but it still requires $\Theta(m \cdot z_T)$ time.

Our central contribution is a remarkably simple algorithm to *directly* enumerate Eulerian trails in directed graphs in the *optimal* $\mathcal{O}(m + z_T)$ time.² Thus, we complete the complexity landscape of computing Eulerian trails in undirected and directed graphs; see Table 1. Furthermore, our algorithm has the following important consequences:

- It improves on an implementation of the BEST theorem for counting Eulerian trails in directed graphs when $z_T = o(n^2)$ – only to *construct* the underlying Laplacian matrix, $\Omega(n^2)$ time is required.
- It improves the combinatorial $\mathcal{O}(m \cdot z_T)$ -time algorithm of Conte et al. [9] for counting Eulerian trails in directed graphs. Like the algorithm of Conte et al., our algorithm is also combinatorial.

The algorithm by Kurita and Wasa [16] for the optimal enumeration of Eulerian trails in undirected graphs is based on *reverse search*, a widely-used technique that was introduced in 1996 by Avis and Fukuda [2]. We use the same overall approach for directed graphs, but introduce combinatorial insights to avoid the explicit connectivity computation at every step. In particular, in the undirected case, one crucially needs to track bridges (edges between 2-connected components). In the directed case, a similar role is played by crossing edges (edges between strongly connected components). However, there is strong evidence from fine-grained complexity [1] that we cannot dynamically keep track of the latter efficiently enough for our purposes. In response, we develop combinatorial insights showing that tracking an easier-to-detect subset of the crossing edges is sufficient in the context of Eulerian trail enumeration. This overcomes the challenges in adapting the algorithm in [16] to directed graphs. Additionally, this avoids the intricate *push-out amortization* technique, which was

² An implementation of this algorithm written by Jeroen op de Beek is made available at <https://github.com/Jeroenodb/Optimal-Enumeration-of-Euler-Trails>.

introduced by Uno [19] in 2015 and used by Kurita and Wasa in [16], making our algorithm both simple and self-contained. Crucially, avoiding this amortization tool allows us to output a desired number $z \leq z_T$ of Eulerian trails in the optimal $\mathcal{O}(m + z)$ time.

Since outputting z Eulerian trails explicitly requires $\Omega(mz)$ space and time, we output a *compressed* representation: a rooted tree, in which each node represents a prefix of an Eulerian trail, and some inner nodes having a single child may be omitted, and are thus implicit. These implicit nodes represent prefixes of Eulerian trails that can be extended only by a unique edge of the input graph, making the decompression of the output natural – in fact, the same tree representation has been used by Kurita and Wasa [16].³

In addition to the rich history of theoretical interest in this problem, Eulerian trails have direct applications in bioinformatics [15] and data privacy [3]. These applications typically operate on directed multigraphs, which motivated us to extend our algorithm to this setting, thus opening the possibility of boosting the efficiency of these applications compared to the existing algebraic computations that are based on the BEST theorem (see [9] for details). Indeed, our direct approach extends to directed multigraphs with little additional effort.

Technical Overview and Paper Organization

The State Tree. Assume a directed graph G has $z_T > 0$ Eulerian trails starting at some fixed node v_0 (this restriction to a single start node can easily be removed, but it is conceptually easier). Consider the set of prefixes of these trails. They can be arranged into a tree where there is an edge between two prefixes if one extends the other by a single edge. Note that the set of leaves is exactly the set of Eulerian trails. See Figure 2 for an example. Since every Eulerian trail has length m , this tree has size $\mathcal{O}(m \cdot z_T)$. Since we construct this tree by recursively exploring it, the current node in the tree encodes the state our exploration algorithm is in. We refer to this tree as the *state tree*, a node as a *state*, and an edge as a *state transition* to avoid any confusion with the nodes and edges from G . See Section 3.

A Simple $\mathcal{O}(m \cdot z_T)$ -time Enumeration Algorithm. A first observation is that this full state tree can be constructed by recursively completing prefixes using Hierholzer’s algorithm for finding an Eulerian trail. Given some inner state (i.e., a proper prefix of an Eulerian trail), a single call to Hierholzer’s algorithm completes the prefix into an Eulerian trail: it creates a path to a leaf state from that inner state. Exploiting a simple fact, we can detect which of the newly created inner states are branching, meaning their prefixes can be completed to a different Eulerian trail. All of these branching states can then be explored recursively in the same fashion. This already yields an $\mathcal{O}(m \cdot z_T)$ -time algorithm, matching the state-of-the-art algorithm by Conte et al. [9], but with a much simpler algorithm. See Section 4.

Compression: Avoiding Repetition in the State Tree. The core idea that enables us to obtain a time-optimal enumeration algorithm is that Eulerian trails are repetitive in a way that can be exploited to obtain a more compact representation. States in the state tree with only one child correspond to prefixes that can be extended in exactly one way. So, in some sense, they do not encode useful information. One can observe this in Figure 2. Thus, our goal is to compress the state tree so that every state has at least two child states (in which case, we call it *branching*). Unfortunately, we cannot quite achieve this. Instead, we

³ This tree representation is standard in enumeration algorithms. Compare it to how the suffix tree of a string is obtained from the trie of its suffixes [21].

settle for a slightly weaker result showing that, with the right compression, there can be only a constant number of non-branching states in a row. This is enough to establish that the compressed state tree has size $\mathcal{O}(z_T)$. This gives the desired output. See Section 5.

Our Compression Rules. We propose two compression rules. The first is simple: if a node has exactly one out-edge, then on every Eulerian trail, that edge must be used as soon as that node is reached. Thus, we can contract the node with its out-neighbor without changing the number of Eulerian trails. See Section 5.1.

The second compression rule exploits that if there is a node that has exactly one out-edge to its own strongly connected component (SCC) and exactly one edge to a different SCC, the edge to the same SCC must be used *first* on all Eulerian trails. Thus, the node can be contracted with its out-neighbor from the same SCC. Unfortunately, exhaustively applying this rule after every state transition seems difficult (and there is evidence from fine-grained complexity to support this). To overcome this difficulty, we propose a weaker (but more technical) rule. See Section 5.2. This compression rule suffices to bound the size of the (compressed) state tree by $\mathcal{O}(z_T)$. See Section 5.4.

Efficient Exploration. Our time-optimal algorithm constructs the state tree similarly to the $\mathcal{O}(m \cdot z_T)$ -time algorithm outlined above. See Section 6. There is an extra algorithmic challenge: we need to maintain what we call the *compressed remaining graph*: while exploring the state tree we want to have access to G after the Eulerian trail prefix corresponding to the current state has been removed. We also want this remaining graph to be exhaustively compressed using our compression rules. By the simplicity of the compression rules, one compression is easy to execute, so the important fact here is that compressions cannot cascade so that many compressions become necessary at one time. See Section 6.1. As in the $\mathcal{O}(m \cdot z_T)$ -time algorithm, we will iteratively use Hierholzer’s algorithm to extend the partially constructed state tree. See Section 6.2.

Putting It All Together. We combine the above ingredients to form our algorithm. See Section 6.3. Note that, if we want to enumerate z Eulerian trails in $\mathcal{O}(m + z)$ time, where z is an arbitrary input parameter, we can terminate the construction of the state tree early.

The remainder of this paper is organized as follows. We start in Section 2 with the necessary definitions and notation. We conclude this paper in Section 7. Additionally, in the full version, we show how our algorithm for simple graphs can be extended, with some care, to enumerate Eulerian trails in directed multigraphs in optimal time.

2 Preliminaries

A *strongly connected component* (SCC) in a directed graph G is a maximal subgraph in which every node can reach every other node using a directed path. The SCCs in G form a partition of the set of nodes. Thus, without any ambiguity, for any node v in G , we write $\text{SCC}(v)$ to refer to the SCC of v in G .

A *crossing* in a directed graph G is an edge between two different SCCs. This is consistent with the usage for any edge between two different parts of a partition. In our case, the partition will always be given by the SCCs in G .⁴

⁴ Sometimes, this is called a (weak) bridge. However, since the term bridge is used inconsistently in directed graphs, we use the term *crossing* to avoid any confusion. Additionally, intuitions about undirected bridges do not carry over. For our purposes, they can even be counterproductive.

A *contraction* of two nodes v and w in a directed graph G modifies the graph as follows. The nodes v and w are removed and a new node vw is added. This new node has all in-neighbors from v and w as in-neighbors and all out-neighbors from v and w as out-neighbors. We create a self-loop on the new node vw if and only if both edges $v \rightarrow w$ and $w \rightarrow v$ exist. This choice is motivated by our application.

We write z for the target number of Eulerian trails to enumerate in a directed graph G ; and we write $z_T(G)$ for the total number of Eulerian trails in G . The graph G may be omitted if it is clear from the context.

3 The State Tree

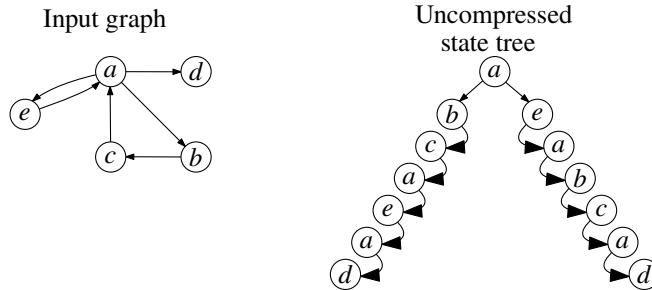
The state tree, as it is commonly used in the reverse search framework, is both our core data structure and the compressed output of our main algorithm. This is the same basic data structure used in the algorithm by Kurita and Wasa for undirected graphs [16]. By definition, explicitly outputting z Eulerian trails requires $\Omega(mz)$ time and space. Our main contribution is to construct a tree of size $\mathcal{O}(z_T)$, where every leaf corresponds to one of the z_T Eulerian trails in G . We also show how to explore a subtree of the state tree having z leaves in $\mathcal{O}(m + z)$ time.

In the rest of the paper, we assume the input graph $G = (V, E)$ is a simple and weakly connected directed graph with at least one Eulerian trail. As mentioned in Section 1, the latter can be verified in $\mathcal{O}(m)$ time using Euler's theorem. A witness can then be found in $\mathcal{O}(m)$ time using Hierholzer's algorithm [13].

Recall that either there is a unique source node s for all Eulerian trails (i.e., a node $s := v$ with $\delta^+(v) = \delta^-(v) + 1$) or the graph contains an Eulerian cycle. If there is an Eulerian cycle, all Eulerian trails can be rotated to start with an arbitrary fixed node. Henceforth, we assume a fixed source node v_0 .

We define the *uncompressed state tree* of G , denoted by T^u , as a rooted tree, where each node of the tree represents a prefix of an Eulerian trail starting at v_0 . See Figure 2 for an example. We add an edge from a tree node s_1 to another tree node s_2 if s_1 corresponds to the prefix obtained by removing the last edge from the prefix corresponding to s_2 . In the rest of the paper, to avoid confusion, we reserve the terms *node* and *edge* for the input graph G , and use *state* and *transition* for the (uncompressed) state tree. For a state s , we write P_s for the prefix (of an Eulerian trail) corresponding to s . The following observation is then trivial.

► **Observation 1.** *The size of T^u constructed over an m -edge graph $G = (V, E)$ is in $\mathcal{O}(m \cdot z_T)$.*



■ **Figure 2** A directed graph and the uncompressed state tree encoding the two Eulerian trails starting at a . For readability, each state is only labeled with the last node on the prefix.

4 Warm-up: A Simple $\mathcal{O}(m \cdot z_T)$ -time Enumeration Algorithm

As a warm-up, we describe a slower, $\mathcal{O}(m \cdot z_T)$ -time version of our main algorithm. We outline our algorithmic strategy and provide an intuition on why even this simple version ties the state-of-the-art algorithm by Conte et al. [9]. The full proof for this version is omitted, as we later prove the faster version in detail. The rest of the paper refines this simple algorithm to achieve the optimal $\mathcal{O}(m + z_T)$ -time complexity.

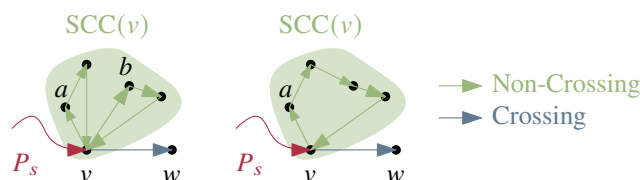
We construct the state tree in a DFS fashion. Given a prefix of an Eulerian trail in G , we can leverage Hierholzer's algorithm [13] to complete it into a full Eulerian trail in G . In this way, given an inner state s (potentially the initial state), we construct a path from s to a new leaf. We repeat this until we have constructed a part of the state tree with z leaves (or have explored the entire state tree).

The remaining challenge is to find out which of the states on the path from s to a leaf, constructed by Hierholzer's algorithm, are *branching* (i.e., have more than one child). At these branching states, we must call Hierholzer's algorithm again with a modified prefix to construct the subsequent set of states.

Recall that a crossing is an edge between two SCCs. This gives Lemma 2, where by G'_s we denote the *remaining graph* after deleting the prefix P_s of an Eulerian trail from G and then removing isolated nodes.

► **Lemma 2.** *A state s of T^u is branching if and only if the last node on P_s has at least two out-edges that are not crossings in G'_s .*

Intuitively, only non-crossing out-edges correspond to real choices because: (1) non-crossing edges must be used before crossing edges; (2) there can be at most one crossing edge at each node; and (3) any non-crossing edge is a valid extension of a prefix. See Figure 3 for an illustration.



■ **Figure 3** On the left, P_s can be extended both by $v \rightarrow a$ and by $v \rightarrow b$ since both of these out-edges are non-crossing. On the right, P_s can be extended only by $v \rightarrow a$. If instead we choose the crossing edge $v \rightarrow w$ first, the other edges from $\text{SCC}(v)$ would not be reachable anymore.

Proof of Lemma 2. Let us prove that (1), (2), and (3) are true. The lemma then follows.

For (1), assume v is the last node on P_s and that, in G'_s , it has both a crossing out-edge $v \rightarrow w$ and a non-crossing out-edge $v \rightarrow a$. Assume for contradiction that $P_s \cdot (v \rightarrow w)$ is a prefix of an Eulerian trail. Then there is some way to extend it such that $v \rightarrow a$ later appears on it. This means that w lies on a cycle from v to v , and therefore w is in $\text{SCC}(v)$. So, $v \rightarrow w$ was not crossing.

For (2), the argument is similar. Assume for contradiction that v has two crossing out-edges in G'_s , $v \rightarrow w$ and $v \rightarrow w'$. An Eulerian trail must traverse every edge in G'_s exactly once. If the trail leaves the SCC of v via $v \rightarrow w$, it can never return to v because, by definition, a crossing connects to a different SCC with no path back. This means the edge $v \rightarrow w'$ remains untraversed, which is a contradiction.

For (3), removing a non-crossing out-edge adjacent to the last node on P_s (a) maintains Euler's degree criterion and (b) leaves the graph weakly connected. By Euler's theorem, this guarantees the existence of an Eulerian trail in G'_s , and thus the extended prefix is valid. ◀

With Lemma 2, we can efficiently detect branching states. Crucially, we do not need to maintain the complete set of crossing edges (which might be difficult as we explain in Section 5.2). Instead, given a state s , we only need to know which out-edges are crossing at the last node on P_s . As it turns out, this is easy.

► **Lemma 3.** *Given a graph H and an Eulerian trail $R = e_1, \dots, e_k$ in H , there is an algorithm running in $\mathcal{O}(|R|)$ time that determines which of the edges e_i are crossings in $H \setminus \{e_1, \dots, e_{i-1}\}$.*

Proof. The crossings are precisely the last out-edges at each node, with respect to their order in R – except if R is a cycle, in which case the last out-edge at the end node of the cycle is not a crossing. The last out-edge from v must be a crossing because all other out-edges from v have been used, and to complete the trail, the algorithm must leave the current SCC (unless the trail is about to end). A non-last out-edge $v \rightarrow w$ cannot be a crossing because the trail must eventually return to v to use its other out-edges. The last edge of a cycle is: it is the last edge, but it returns to the start node, which is in the same SCC. ◀

This enables us to detect the branching states on a path from an inner state to a leaf in time linear in the size of that path. For a path from an inner state s_1 to a leaf s_2 , we apply Lemma 3 to $H := G'_{s_1}$ and $R := P_{s_2} - P_{s_1}$. Combining this insight with the DFS strategy based on Hierholzer's algorithm shows that we can construct the state tree in time linear in the number of states we construct. This allows us to avoid the application of the push-out amortization technique. Together with Observation 1, which bounds the size of the (uncompressed) state tree, this gives the following result.

► **Theorem 4.** *There is an algorithm that, for any directed m -edge graph $G = (V, E)$ and any parameter $z \in \mathbb{N}$, computes z Eulerian trails in G (or all z_T of them if $z_T \leq z$) in $\mathcal{O}(m \cdot \min(z, z_T))$ time.*

We recovered the state-of-the-art result [9] with a simpler algorithm. Although the algorithm in [9] is for assessing z_T , it can be lifted to enumerate the trails with some effort [9, Section 5.1], and it also works for directed multigraphs.

5 Compression: Avoiding Repetition in the State Tree

Notice that T^u can be highly repetitive. For example, in Figure 2, the cycle $a \rightarrow b \rightarrow c \rightarrow a$ in the input graph appears in both Eulerian trails from node a to node d , so the left and right subtrees in the state tree share duplicate information. We aim to compress T^u such that there are only branching states (states with at least two children) and leaves. We then rely on the following standard fact to bound the number of branching states in terms of the number of leaves (and thus in terms of the number of Eulerian trails).

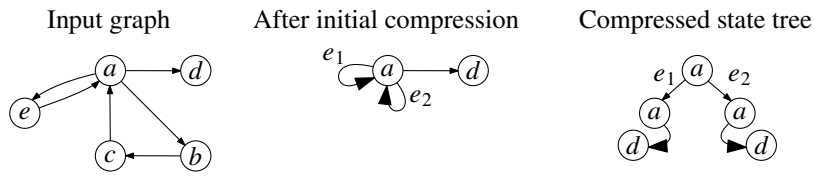
► **Fact 5.** *A rooted tree with k leaves has at most k branching states.*

Unfortunately, having only branching states and leaves is not (efficiently) attainable. Instead, we settle for the more modest goal of having only a constant number of non-branching inner states per branching state. Concretely, we bound the number of non-branching states on a path between two branching states. For our compression, we take a closer look at when non-branching states can occur. If s_1 is non-branching (with child state s_2), we call the transition $s_1 \rightarrow s_2$ *forced* because it is the only way to extend the prefix P_{s_1} .

5.1 Compression Rule 1: Only One Out-edge

The simplest compression case concerns nodes with out-degree one. Given a state s , if there is a node v in the remaining graph G'_s with out-degree one, it is clear that any completion of the prefix P_s must use this out-edge as soon as it reaches v . Note that for any state s , the last node on P_s (or just *last node*, for short) is the node where the next edge has to depart from. Thus, if w is the only out-neighbor of v , then in the subtree of the state tree rooted at s , any state s_1 with current node v has a single child state s_2 with current node w . Thus, the transition $s_1 \rightarrow s_2$ is forced. For an example, recall Figure 2. There, nodes b , c , and e have only a single out-edge, resulting in forced transitions.

► **Compression Rule 1.** *For any state s , if in the remaining graph G'_s , there is a node v with out-degree one, contract that node and its only out-neighbor.*



■ **Figure 4** The graph from Figure 2 after Compression Rule 1 has been applied exhaustively, together with the compressed state tree. Compare this smaller state tree to the one from Figure 2.

Since contractions are associative and commutative (up to isomorphism), we do not need to specify their order. Note that after applying this compression rule, the graph may not be simple anymore; we allow each node to have one or more self-loops. To visualize the effect of this rule, see Figure 4. The next lemma proves its soundness.

► **Lemma 6.** *For any state s , there is a bijection between Eulerian trails in the remaining G'_s and the same graph after the application of Compression Rule 1.*

Proof. The claimed bijection f is given by the contractions performed. To see the claim, take any edge $v \rightarrow w$ that is contracted by Compression Rule 1. Then the out-degree of v in G'_s is one and thus in any Eulerian trail in G'_s , v must be directly followed by w . This proves that f is injective. To see that f is surjective, observe that any Eulerian trail in the compressed graph can be extended to an Eulerian trail in the uncompressed graph by undoing the contractions. ◀

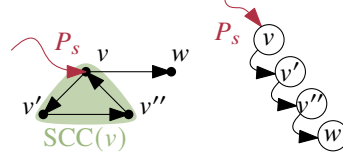
In our algorithm, we will explore the state tree while efficiently maintaining the compressed remaining graph for the current state. More formally, we have the following lemma.

► **Lemma 7.** *Let $s_1 \rightarrow s_2$ be a transition in the state tree that appends the edge $v \rightarrow w$ to P_{s_1} . Assume G'_{s_1} is fully compressed with Compression Rule 1. When generating G'_{s_2} , Compression Rule 1 needs to be applied at most once (specifically at v).*

Proof. Since G'_{s_2} is obtained from G'_{s_1} by deleting the edge $v \rightarrow w$ and all isolated nodes, the only node at which the out-degree decreases (and thus may become one) is v . ◀

5.2 Compression Rule 2: Crossings

By the definition of crossing, a prefix of an Eulerian trail that ends at a node v that has a crossing $v \rightarrow w$ must first be extended with the remaining edges in the SCC of v before using the crossing. Indeed, this follows because after the crossing, the trail cannot return to the SCC of v , and therefore cannot use the remaining edges within that component. See Figure 5 for an example. This leads to the following observation.



■ **Figure 5** For a state s in the state tree, P_s is some prefix of an Eulerian trail. On the left, notice that $v \rightarrow w$ is a crossing. Thus, P_s can only be extended by first using the edges from $\text{SCC}(v)$ and then using $v \rightarrow w$. In the state tree, displayed on the right, the state s therefore only has a single child state.

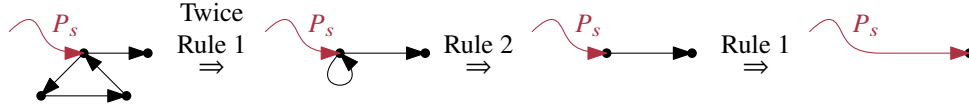
► **Observation 8** (Candidate Compression Rule). *Let s be a state and let v be the last node on its prefix P_s . If v has at most one out-edge that is not a crossing, s has only one child state.*

We would like to turn Observation 8 into a compression rule. This would even be a stronger version of Compression Rule 1 and would immediately give us that the compressed state tree contains only a few non-branching states. For our algorithm, it is crucial that after each state transition, we can efficiently maintain the compressed remaining graph. Unfortunately, this seems difficult for Observation 8.⁵ Instead, we will use the following.

► **Compression Rule 2.** *Let s be any state. If there is a node v in G'_s with only one self-loop, exactly one other out-edge, and exactly one other in-edge, remove the self-loop.*

If v is the last node on P_s , we remove the self-loop only if v has no other in-edge.

At first, this may appear like a restrictive special case (and it is), but we will see that (together with Compression Rule 1) it guarantees that in a compressed graph, instances where Observation 8 applies are rare (and do not occur in a row). See Figure 6 for an example of how these rules act together.



■ **Figure 6** Recall the graph from Figure 5. We can see that together Compression Rules 1 and 2 compress this graph so that no forced state transition remains.

Before we proceed with proving the size bound, we give Lemma 9, which proves that Compression Rule 2 is sound.

► **Lemma 9.** *For any state s , there is a bijection between Eulerian trails in the remaining G'_s and the same graph after the application of Compression Rule 2.*

Proof. The claimed bijection f is given by the self-loop removals. For simplicity, assume a single self-loop $v \rightarrow v$ was removed. The result then extends to multiple rule applications by chaining the resulting bijections.

⁵ We effectively want a decremental algorithm maintaining strongly connected components with undo support. Unfortunately, the best algorithms give only amortized guarantees for the decremental case and any fast enough algorithm with worst-case guarantees would violate a conditional lower bound. Technically, these general lower bounds do not rule out an algorithm for the graph class of Eulerian graphs, but there is no known decremental algorithm amenable to the undo support we would require. See [5, 4] for the currently best decremental SCC algorithms and [1] for the conditional lower bounds.

For injectivity, assume that we have two Eulerian trails $T_1 \neq T_2$ with $f(T_1) = f(T_2)$. The difference between T_1 and T_2 cannot be due to the self-loop: its position is *fixed* because the out-edge at v must be a crossing. This holds as the condition on the number of in-edges at v means that v cannot be visited again after using the out-edge. Thus, the self-loop must be used directly before the out-edge. Also, the self-loop must be traversed exactly once in both trails. The difference must therefore lie in a different sequence of edges, which would mean that $f(T_1) \neq f(T_2)$, contradicting our initial assumption. Thus, $T_1 \neq T_2$ is false.

For surjectivity, again, observe that any Eulerian trail in the compressed graph can be extended to an Eulerian trail in the uncompressed graph by undoing the self-loop removal. Let T' be any Eulerian trail in the compressed graph. Then, since v has an in-edge in the compressed graph by the condition of Compression Rule 2, the trail T' must pass through v . By inserting the self-loop $v \rightarrow v$ at this position in T' , we can construct an Eulerian trail for the uncompressed graph because the degrees of v remain balanced. ◀

There is exactly one structural scenario where applications of both rules can cascade into each other, but a slight adjustment to the graph representation can easily avoid this.

5.3 Handling Induced Two-Way Paths

In particular, a cascade can occur along an *induced two-way path*: a sequence of nodes and edges $v_1 \rightleftarrows \dots \rightleftarrows v_k$, where the inner nodes $v_2, \dots, v_{k-1}$ have exactly in-degree 1 and out-degree 1 in both directions. We represent such a path by replacing the inner nodes with a special meta-node annotated with the length k (i.e., a constant-size representation). If an induced two-way path becomes disconnected from the rest of the graph at one of its endpoints (i.e., only one of v_1, v_k has an edge other than those of the induced two-way path), the entire path can be immediately contracted into a self-loop on the remaining connected endpoint. When running Hierholzer's algorithm, if an induced two-way path is entered from both ends, we can easily generate all branching states corresponding to the valid prefix and suffix pairs of that path. If the path is only entered from one end, all state transitions traversing the implicitly represented inner nodes are non-branching and can be safely omitted from the explicit state tree. We can handle these updates efficiently.

► **Lemma 10.** *Let $s_1 \rightarrow s_2$ be a transition in the state tree that appends the edge $v \rightarrow w$ to P_{s_1} . Then, any induced two-way paths can be updated in $\mathcal{O}(1)$ time, and Compression Rule 1 will be applied at most once while updating the remaining graph.*

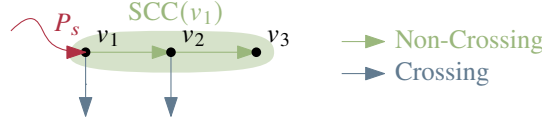
Proof. First, observe that updating the induced two-way paths is easy. The only things that can happen are that v leaves one, enters one, or two of them merge at v . All of these things can be checked and performed in $\mathcal{O}(1)$ time. As Compression Rule 1 was not applicable at w before, the only case in which it can become applicable is if it now cascades through an induced two-way path that is only connected at one end. This follows as an induced two-way path is the only case where contracting an edge $u \rightarrow v$ not only reduces the in-degree of v , but also its out-degree (after an application of Compression Rule 2), making Compression Rule 1 applicable at v . The induced property then follows from the degree-criterion for Eulerian graphs. By the checks we just performed, it would have been turned into a self-loop. Thus, no cascading happens. ◀

5.4 The Size of the Compressed Graph with Rules 1 and 2

The core result that underlies our algorithm is that since in each state we compress the remaining graph using Compression Rules 1 and 2, the resulting graph has few non-branching nodes. This follows from Lemma 11.

► **Lemma 11.** *Assume that in the state tree, Compression Rules 1 and 2 are applied exhaustively after each transition. Then there are never two forced state transitions in a row (i.e., there is no path of length two of forced transitions).*

Proof. Assume to the contrary that s_1, s_2, s_3 are three states such that the two transitions $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_3$ are forced. Then s_1 and s_2 only have a single child state each. For $i \in [1, 3]$, write v_i for the last node on P_{s_i} . So, by assumption, appending first v_2 and then v_3 to P_{s_1} is the only way to extend P_{s_1} keeping it a prefix of an Eulerian trail. Let our analysis proceed in G'_{s_1} . This graph is Eulerian because removing a prefix of an Eulerian trail does not change that property. Nodes v_1 and v_2 have out-degree at least two as otherwise Compression Rule 1 would be applicable. Notice that they have at most one non-crossing out-edge as otherwise the states would be branching. Thus, we can conclude that v_1 and v_2 each have exactly one non-crossing out-edge and one crossing out-edge. The edges $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_3$ must be the non-crossing edges as those must appear before the crossings on any Eulerian trail.



■ **Figure 7** The contradictory situation created in the proof of Lemma 11. Here, the only way to extend P_s is by making two forced choices in a row (adding $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_3$). In the proof we argue why this cannot happen if G is Eulerian and compressed using Compression Rules 1 and 2. In particular, no SCC can have two outgoing crossings.

None of these edges can be self-loops as otherwise Compression Rule 2 would have been applicable. Thus, v_1 and v_2 are distinct nodes. Since they are connected by a non-crossing edge, they must be in the same SCC. This SCC must now have two outgoing crossings. This can never occur in any Eulerian graph (also see [9, Lemma 1] for a more general version of this observation). See Figure 7 for an illustration. ◀

This lemma, together with Fact 5, yields the following main result.

► **Theorem 12.** *The state tree compressed using Compression Rules 1 and 2 has size $\mathcal{O}(z_T)$. In particular, any state with z leaf-descendants has $\mathcal{O}(m + z)$ ancestors and descendants.*

6 Enumeration Algorithm

We have established that the compressed state tree is small enough. Our next step is to show how to efficiently construct it via a DFS exploration, thereby enumerating all Eulerian trails. This approach requires two main ingredients: (1) an efficient way to maintain the compressed remaining graph for the current state; and (2) a method to extend the partial state tree. In the usual DFS fashion, we would add new states one-by-one, but we need to be more clever and batch some operations. Concretely, we will extend the state tree by adding a full path of states from the current state to a new leaf of the state tree at a time. Crucially, we make sure that the remaining graph for each state is always exhaustively compressed.

6.1 Keeping the Remaining Graph Updated and Compressed

The main observation towards efficiently computing a compressed state tree is that removing a single edge only triggers a constant number of compression operations. In particular, compression operations do not cascade, that is, a single compression does not trigger a chain reaction that creates many new compressions. Lemma 13 formalizes this.

► **Lemma 13.** *Given a graph G , exhaustively compressed by Compression Rules 1 and 2, and an edge e , we can compute $G - e$, after Compression Rules 1 and 2 have been applied exhaustively, in $\mathcal{O}(1)$ time.*

Proof. Let $e = v \rightarrow w$. Removing e decreases the out-degree of exactly one node, namely v . Thus, Compression Rule 1 can become applicable only at v (or nowhere). All resulting applications of the rule are handled in $\mathcal{O}(1)$ time by Lemmas 7 and 10.

Similarly, Compression Rule 2 can become applicable only at v or w . At each of these, it can only be applied once (since by the condition of Compression Rule 2 they have exactly one self-loop). At each of these nodes, Compression Rule 1 can become applicable, but nowhere else. As argued above, potentially applying Compression Rule 1 does not cause further compressions. Finally, note that checking the conditions of the rules at v and w , the contraction of two nodes and the removal of a self-loop can be implemented in $\mathcal{O}(1)$ time. ◀

This implies that the state tree can be explored in a DFS fashion while keeping an up-to-date compressed version of the remaining graph with constant overhead per state transition. Note that by storing the performed compression operations on a stack, they can be undone in the same constant overhead per state transition while the DFS exploration backtracks. Additionally, since by the above arguments the compression does not cascade, the following holds.

► **Corollary 14.** *We can exhaustively compress any directed m -edge graph in $\mathcal{O}(m)$ time.*

6.2 Exploring Using Hierholzer's Algorithm

As in Section 4, we will iteratively use Hierholzer's algorithm to extend a partially constructed state tree. However, this time the state tree will be compressed using Compression Rules 1 and 2. Essentially, we will perform a DFS, where Hierholzer's algorithm will serve as the subroutine that adds a new path from a not-fully-explored inner state to a new leaf. To obtain a time-optimal algorithm, we must analyze the cost of these calls more carefully than in Section 4, where their cost was linear in the number of states in the uncompressed tree. Instead, we will show that the cost is linear in the number of branching states created.

► **Lemma 15.** *Given a partially constructed compressed state tree and a branching inner state s , where one subtree has not been constructed, we can construct a new state-path R from s to a leaf in $\mathcal{O}(|R|)$ time. Here, $|R|$ is the number of newly created states in the compressed state tree.*

Proof. Note that R corresponds to an Eulerian trail in the remaining graph G'_s . Hierholzer's algorithm finds this specific trail. Our goal is to show that the lengths of this trail and R differ only by at most a constant factor. This allows us to conclude that running Hierholzer's algorithm on G'_s takes $\mathcal{O}(|R|)$ time. Informally, we create a number of states that is worth the running time we spent. For this, we show that not too much of the trail is removed by compression.

We classify all edges in the Eulerian trail computed by Hierholzer's algorithm in G'_s into three categories: (1) those corresponding to branching states; (2) those corresponding to non-branching states that also exist in the compressed state tree; and (3) those corresponding to non-branching states that are removed by compression. Clearly, states in categories (1) and (2) are bounded by $|R|$.

Since G'_s is exhaustively compressed, any states in category (3) must become compressible by the removal of some (earlier) edge on R . In particular, as argued in the proof of Lemma 13, an edge in category (1) or (2) can cause only a constant number of possible compressions

later on R . In the same proof, we also show that each compression that is caused in this fashion does not cascade. As we start off with an exhaustively compressed graph, this is the only way new applications of the compression rules can become possible. Thus, the number of states in category (3) is bounded by the number of states in categories (1) and (2).

In total, the Eulerian trail constructed has length $\mathcal{O}(|R|)$ and thus the execution of Hierholzer's algorithm takes $\mathcal{O}(|R|)$ time, yielding the desired result. $\blacktriangleleft$

On a technical note, we need to be a bit careful about picking the state s at which to add a new path to a leaf. Recall that we may want to output a desired number $z \leq z_T$ of Eulerian trails in the optimal $\mathcal{O}(m + z)$ time.

Formally, if $z \leq z_T$, we can construct a subtree with z leaves and $\mathcal{O}(m + z)$ inner states by adding the new inner state to leaf paths in a DFS fashion; that is, processing a branching state with missing children only after all its already constructed children have no unconstructed children themselves.

6.3 Summary of the Algorithm

Despite proving many technical statements – like bounding the state tree size, the soundness of compression rules, and detailing efficient subroutines – the final algorithm is simple to state. See Algorithm 1.

■ **Algorithm 1** Computing a subtree of the compressed state tree with z leaves.

Require: $G = (V, E)$ is weakly connected and Eulerian with source v_0 , and $z > 0$ is an integer parameter.

- 1: $T \leftarrow \{(v_0, [])\}$ $\triangleright$ State tree only containing the empty prefix at v_0
 - 2: Exhaustively compress G $\triangleright$ Corollary 14
 - 3: **while** T has fewer than z leaves **do**
 - 4: Take a branching state s with an unexplored subtree (if there is none, break)
 - 5: Add a path R from s to a new leaf using Hierholzer's algorithm $\triangleright$ Lemma 15
 - 6: Compress R $\triangleright$ Lemma 13
 - 7: Mark the branching states with unexplored subtrees on R $\triangleright$ Lemma 3
 - 8: **Output** T
-

We also give the properties of this algorithm as a formal statement, summarizing our main contribution.

► **Theorem 16.** *For any directed m -edge graph $G = (V, E)$ and any parameter $z \in \mathbb{N}$, Algorithm 1 computes a compressed representation of z Eulerian trails in G (or all z_T of them if $z_T \leq z$) in $\mathcal{O}(m + \min(z, z_T))$ time.*

We stress that the parameterization by $z \in \mathbb{N}$ in Theorem 16, in conjunction with the $\mathcal{O}(m + \min(z, z_T))$ bound, can be useful in applications [3, 9]. In particular, these properties of our algorithm allow one to assess whether $G = (V, E)$ has at least $z \leq z_T$ Eulerian trails in $\mathcal{O}(m + z)$ time, which is useful for real-world graphs with very large z_T .

7 Final Remarks

We have developed a remarkably simple and optimal algorithm for enumerating Eulerian trails in directed graphs, essentially completing the complexity landscape of this fundamental topic. As a consequence, our result improves on an implementation of the BEST theorem for counting Eulerian trails when $z_T = o(n^2)$, and, in addition, it unconditionally improves the combinatorial $\mathcal{O}(m \cdot z_T)$ -time counting algorithm of Conte et al. [9] for the same task.

Beyond its theoretical or practical impact, our paper serves as a useful educational resource for graph algorithms classes, demonstrating how combinatorial insight yields *simple* and *optimal* solutions.

References

- 1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.
- 2 David Avis and Komei Fukuda. Reverse search for enumeration. *Discret. Appl. Math.*, 65(1-3):21–46, 1996. doi:10.1016/0166-218X(95)00026-N.
- 3 Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe text indexing. *ACM J. Exp. Algorithmics*, 26:1.10:1–1.10:26, 2021. doi:10.1145/3461698.
- 4 Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, SCC, and shortest paths via directed expanders and congestion balancing. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1123–1134. IEEE, 2020. doi:10.1109/FOCS46700.2020.00108.
- 5 Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. Decremental strongly connected components and single-source reachability in near-linear time. *SIAM J. Comput.*, 52(on):128–155, 2019. doi:10.1137/20M1312149.
- 6 Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer, 2002. doi:10.1007/978-1-4612-0619-4.
- 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. URL: <http://www.siam.org/meetings/analco05/papers/09grbrightwell.pdf>.
- 8 James R. Bunch and John E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974. URL: <http://www.jstor.org/stable/2005828>.
- 9 Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giulia Punzi. Fast assessment of eulerian trails in graphs with applications. *ACM Trans. Knowl. Discov. Data*, 20(1), December 2025. doi:10.1145/3771997.
- 10 Leslie V. Foster. Gaussian elimination with partial pivoting can fail in practice. *SIAM J. Matrix Anal. Appl.*, 15(4):1354–1362, 1994. doi:10.1137/s0895479892239755.
- 11 Paweł Gawrychowski and Marcin Knapik. Enumerating all directed spanning trees in optimal time, 2026. doi:10.48550/arXiv.2603.11763.
- 12 Gene H. Golub and Charles F. Van Loan. *Matrix Computations, Third Edition*. Johns Hopkins University Press, 1996.
- 13 Carl Hierholzer. Ueber die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873. doi:10.1007/BF01442866.
- 14 Nicholas J. Higham. *Accuracy and stability of numerical algorithms, Second Edition*. SIAM, 2002. doi:10.1137/1.9780898718027.
- 15 Carl Kingsford, Michael C. Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinform.*, 11:21, 2010. doi:10.1186/1471-2105-11-21.
- 16 Kazuhiro Kurita and Kunihiro Wasa. Constant amortized time enumeration of Eulerian trails. *Theor. Comput. Sci.*, 923:1–12, 2022. doi:10.1016/j.tcs.2022.04.048.

- 17 William Thomas Tutte. The dissection of equilateral triangles into equilateral triangles. *The American Mathematical Monthly*, 55(4):249–250, 1948.
- 18 Takeaki Uno. A new approach for speeding up enumeration algorithms. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *Algorithms and Computation, 9th International Symposium, ISAAC '98, Taejon, Korea, December 14-16, 1998, Proceedings*, volume 1533 of *Lecture Notes in Computer Science*, pages 287–296. Springer, 1998. doi:10.1007/3-540-49381-6_31.
- 19 Takeaki Uno. Constant time enumeration by amortization. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures - 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*, volume 9214 of *Lecture Notes in Computer Science*, pages 593–605. Springer, 2015. doi:10.1007/978-3-319-21840-3_49.
- 20 Tatyana van Aardenne-Ehrenfest and Nicolaas G. de Bruijn. *Circuits and Trees in Oriented Linear Graphs*, pages 149–163. Birkhäuser Boston, Boston, MA, 1987. doi:10.1007/978-0-8176-4842-8_12.
- 21 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. doi:10.1109/SWAT.1973.13.