# Space-Efficient Hierholzer: Eulerian Cycles in $O(m)$ Time and $O(n)$ Space*

Ziad Ismaili Alaoui† Detlef Plump‡ Sebastian Wild§

**Abstract.** We describe a simple variant of Hierholzer's algorithm that finds an Eulerian cycle in a (multi)graph with $n$ vertices and $m$ edges using $O(n \lg m)$ bits of working memory. This substantially improves the working space compared to standard implementations of Hierholzer's algorithm, which use $O(m \lg n)$ bits of space. Our algorithm runs in linear time, like the classical versions, but avoids an $O(m)$-size stack of vertices or storing information for each edge. To our knowledge, this is the first linear-time algorithm to achieve this space bound, and the method is very easy to implement. The correctness argument, by contrast, is surprisingly subtle; we give a detailed formal proof. The space savings are particularly relevant for dense graphs or multigraphs with large edge multiplicities.

**1 Introduction.** An *Eulerian cycle* in a graph is a closed walk that traverses every edge exactly once. A graph is called *Eulerian* if it contains an Eulerian cycle. Euler's study of the existence of Eulerian cycles and their computation is often cited as the birth of modern graph theory. Apart from historical significance, efficiently computing them has applications in, e.g., genome assembly [7], routing problems [6], and 3D printing [9].

Classic algorithms such as Fleury's algorithm or Hierholzer's algorithm [5], both dating from the late 19th century, are simple to describe and find an Eulerian cycle in any Eulerian graph in polynomial time. Indeed, both consider each edge only once, and Hierholzer's algorithm can be implemented to run in optimal linear time. Being both simple and efficient, the latter is the method of choice in practice. Typical implementations of Hierholzer's algorithm (see below) use $O(m \lg n)$ bits of working memory. (Here and throughout, $\lg = \log_2$.)

In this paper, we present a simple variant of Hierholzer's algorithm that uses only $O(n \lg m)$ bits of working memory and runs in optimal $O(n + m)$ time. For dense graphs or multigraphs with many parallel edges, this is a substantial saving. Note that this space can neither hold the entire input nor the entire output of the problem; we treat the input graph as given in read-only memory and produce the Eulerian cycle *in order* to a write-only stream. In particular, our algorithm would be suitable for an application that directly consumes and uses edges of the Eulerian cycle as they are computed, potentially without ever storing the entire cycle. By *working memory*, we mean the extra space occupied by auxiliary data structures during the computation. Our algorithm relies on an assumption about the graph representation, which is satisfied in typical adjacency-list implementations: for directed graphs, we require that we can iterate over incoming and outgoing edges; for undirected graphs, we (instead) require a consistent ordering of vertices across all adjacency lists.

To our knowledge, we present the first algorithm with working space $O(n \lg m)$ bits that runs in linear time and explicitly produces an Eulerian cycle as its output. Our algorithm is also usable in rule-based models of computation, such as graph-transformation languages like GP 2 [8], which do not natively support stacks or recursion. This is not known to be true for other standard efficient implementations of Eulerian cycle algorithms.

**1.1 Hierholzer's Algorithm.** Hierholzer's algorithm was originally described in 1873 [5], unsurprisingly without detailed information about data structures for efficient execution on a computer. The original algorithm consists of following an arbitrary walk in an Eulerian graph until we get stuck, which can only happen when we closed a cycle. If this cycle has not used all edges yet, starting at such an unused edge, again following an arbitrary walk of unused edges can again only get stuck when closing a cycle; we can then fuse this new cycle into

*Ziad Ismaili Alaoui and Sebastian Wild are supported by EPSRC grant EP/X039447/1.

†University of Liverpool, United Kingdom (ziad.ismaili-alaoui@liverpool.ac.uk).

‡University of York, United Kingdom (detlef.plump@york.ac.uk).

§Philipps-Universität Marburg, Germany, and University of Liverpool, United Kingdom (wild@informatik.uni-marburg.de).

the previous one as a detour before continuing the original cycle. By iterating this procedure, we eventually obtain an Eulerian cycle.[1]

**1.2 Typical Implementations.** Several implementations of Hierholzer's algorithm are in use. Closest to the above description is using a linked list of edges to represent the Eulerian cycle; then one can insert cyclic detours as sketched above at any point in the tour. This approach is used, e.g., in the widely used Java graph library *JGraphT* [2]. It clearly needs $\Theta(m \lg m)$ bits of working memory for storing the linked list of edges.

More amenable to producing the output directly in the correct order is the approach to treat Hierholzer's algorithm as an *edge-centric* depth-first-search. We follow the basic steps of a graph traversal, but mark edges rather than vertices. So we only backtrack from a vertex when all of its incident edges have been traversed. When this happens for the first time, the last edge traversed to this dead end was the edge closing the first cycle of Hierholzer's algorithm, and can thus be output now as the *last* edge of the Eulerian cycle. Iteratively, the same is true for the next dead end vertex.[2] Both the stack and the marking of edges require $\Theta(m \lg m)$ bits of working space in the worst case. A minor twist of this approach, where edges are not marked, but removed from a copy of the input graph, is used, e.g., in *NetworkX* [1], a popular graph framework for Python. The copy, of course, also uses $\Omega(m \lg n)$ bits of space.

**1.3 Our Idea.** Our key idea is the following observation. While an edge-centric DFS is *one* correct implementation of Hierholzer's algorithm, it is a needlessly specific one. Concretely, there is no need to keep the entire constructed walk on a stack; the order in which we add remaining "detour cycles" to the tour is immaterial for Eulerian cycles anyway. The only constraint we have is that we *reserve* the edges of the *original* cycle/traversal to be the last ones to backtrack on, to make sure that we end at the starting vertex and do not omit any edges. For that, it suffices to remember one incoming edge per vertex. The second thing we need to remember is which edges have already been traversed; but for that, we can store for each vertex an iterator in its adjacency list.

**1.4 Directed and Undirected Eulerian Cycles.** Hierholzer's algorithm works equally well for directed and undirected graphs, and it can deal with parallel edges even if these are represented as duplicate entries in the adjacency list. Since the implementation is slightly cleaner to state for directed graphs, in what follows, we consider directed Eulerian multigraphs that are given in adjacency list representation with no further assumptions.

For directed graphs, it is convenient to (logically) reverse directions of edges; then the order of edges produced via backtracking from the edge-centric DFS directly gives the Eulerian cycle in the correct order. For undirected graphs, clearly, we can skip the reversal of edges as the Eulerian cycle can be traversed in both directions.

**1.5 Related Work.** Glazik, Schiemann, and Srivastav [3] present a streaming algorithm for Eulerian cycles. While they also achieve $O(n \lg n)$-space and in a much more restrictive model, it is not clear if their algorithm can be implemented to run in linear total time, and they construct the cycle implicitly via a successor function. Their algorithm is arguably more complicated than Hierholzer's algorithm.

Hagerup, Kammer, and Laudahn [4] proposed an algorithm using $O(n + m)$ time and $O(n + m)$ *bits* of space for computing Eulerian cycles. They do not need strong assumptions regarding the graph data representation, but focus on undirected graphs only. They build a nontrivial dynamic data structure in each vertex that stores which pairs of edges are adjacent in the Eulerian cycle. Their algorithm is considerably more involved than the typical implementations of Hierholzer's algorithm using $O(n + m)$ *words* of space.

**1.6 Outline.** Our paper is organised as follows. Section 2 introduces basic notation. Section 3 presents the Space-Efficient-Hierholzer algorithm. Finally, Section 4 proves correctness, establishes the $O(m)$ running time, and shows that the working memory usage is $O(n \lg m)$.

**2 Preliminaries.** We write $[n]$ for $\{1, 2, \ldots, n\}$. Throughout the paper, $G = (V, E)$ denotes a finite directed multigraph, where $V$ is the set of vertices and $E$ is the multiset of directed edges, with $|V| = n$ and $|E| = m$. Loop edges are allowed. We assume without loss of generality that $V = [n]$. We denote by $d^+(v)$ and $d^-(v)$ the out-degree and in-degree of vertex $v$, respectively. The $i$th out-neighbour of a vertex $v$ is denoted by $\Gamma^+(v, i)$, and its $i$th in-neighbour by $\Gamma^-(v, i)$, assuming arbitrary but fixed orderings of incoming and outgoing edges for each vertex. We concisely write $uv$ for a (directed) edge from $u$ to $v$.

[1] Throughout the paper, we always assume the graphs to be Eulerian, i.e. to contain an Eulerian cycle.

[2] See, e.g., slide 33 of these lecture notes: https://www.wild-inter.net/teaching/ea/09-graph-algorithms.pdf#page=40.

Definition 2.1 (Eulerian, Eulerian cycle). *A directed graph $G$ is* Eulerian *if and only if it is strongly connected*[3] *and the in-degree equals the out-degree at every vertex; i.e. $d^+(v) = d^-(v)$ for all $v \in V$. An* Eulerian cycle *in $G$ is a closed trail that traverses every edge exactly once and returns to its starting vertex.*

A *trail* in a graph is a walk in which all edges are distinct. A *closed trail* is a trail whose first and last vertices coincide.

We assume that basic arithmetic and memory operations on $\lg m$-bit integers take constant time, and that the in- and out-degree functions as well as in- and out-neighbour queries can be evaluated in constant time per operation.

## 3 Space-Efficient Implementation of Hierholzer's Algorithm.

We now describe our algorithm, which computes an Eulerian cycle in a directed graph using only $O(n \lg m)$ bits of working space, where $n$ is the number of vertices and $m$ is the number of edges.

### 3.1 Conceptual Algorithm.

Let $G$ be a directed Eulerian graph, and let $v_0$ be the starting vertex (it may be chosen arbitrarily). Unlike the classical approach, our algorithm traverses the graph by following *incoming* edges rather than outgoing ones; that is, we perform a reverse traversal of the graph. This reversal allows the algorithm to compute the tour incrementally, in the correct order, without needing to store the entire tour and reverse it at the end.

We first describe the conceptual algorithm in terms of edge markings (colouring edges Black, Red, Green, or Dashed) to clarify the traversal and backtracking logic. Our implementation will represent these only implicitly.

Initially, all edges are Black. The algorithm begins at an arbitrary starting vertex $v_0$ and follows Black incoming edges via a reverse traversal. When a vertex $v$ is reached for the first time in this traversal, via a Black edge $e = uv$, we colour $e$ Red. Otherwise, if we traverse an edge $e = uv$ to a vertex $v$ that had already been reached before, we colour $e$ Green.

When we reach a vertex $v$ without Black incoming edges, we backtrack from it via an *outgoing* edge according to the following rule:

1. If $v$ has an outgoing Green edge, backtrack along it;
2. otherwise, if $v \neq v_0$, backtrack via its (unique) Red outgoing edge;
3. otherwise, if $v = v_0$, terminate the algorithm.

Each time we backtrack across an edge (irrespective of whether it was Green or Red before), we mark it Dashed and we output it as the next edge of the Eulerian cycle. In this way, the Eulerian cycle is produced sequentially, in the correct order. Figure 3.1 shows a sample execution of the algorithm. Notice that, at step (m) of the figure, the traversal proceeds to vertex 2 instead of vertex 6; this is due to the backtracking rules, which prioritise Green edges over Red ones. Indeed, if the traversal proceeded to vertex 6 instead, it would get stuck at vertex 1, and the two edges between vertices 2 and 5 would never be counted as part of the computed cycle. The backtracking rules ensure that such a scenario does not happen.

### 3.2 Space-Efficient Implementation.

We now describe our space-efficient implementation of the conceptual algorithm, see Algorithm 3.1. For each vertex $u$, the algorithm uses a counter `next`$[u]$ to keep track of how many of its incoming and outgoing edges have been traversed. The bit vector `visited` stores whether a vertex has been seen before, and `skipped` ensures that the special backtracking edge (the Red outgoing edge) is not used until all other outgoing edges have been used.

When a vertex $v$ is visited for the first time (via an outgoing edge $vu$), the algorithm records $u$ as $v$'s predecessor by setting $B[v] \leftarrow u$; this is equivalent to marking $uv$ Red. Backtracking begins when a vertex $u$ has no unexplored incoming (Black) edges, which occurs when `next`$[u] > d^-(u)$ (i.e. when `next`$[u]$ is out of the bounds of $d^-(u)$). At that point, the algorithm starts following outgoing edges from $u$. To ensure the Red edge is used last, it checks whether the next candidate edge (indexed by `next`$[u] - d^-(u)$) leads to $B[u]$. If so, and if it has not already been skipped, the algorithm skips this edge. This guarantees that all non-Red (i.e. Green) edges are backtracked over

[3] A set of vertices is considered *strongly connected* if, for every pair of vertices $u, v$, there is a path from $u$ to $v$. This condition rules out isolated vertices.

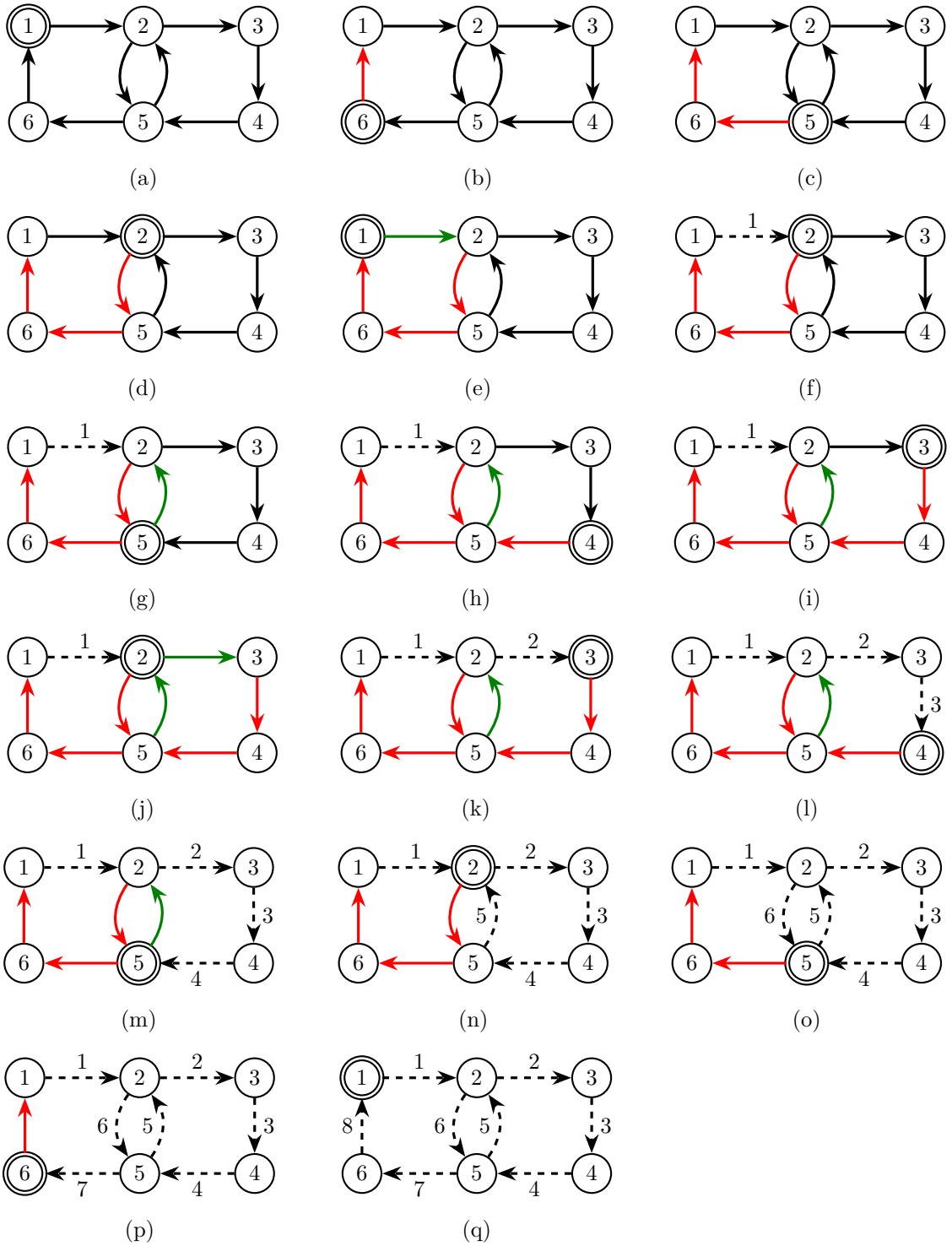


Figure 3.1: Sample execution of Space-Efficient-Hierholzer. For clarity, Dashed edges are labelled in the order in which they are written. The current vertex is represented by double borders.

**Algorithm 3.1** SPACE-EFFICIENT-HIERHOLZER

**Input:** An Eulerian directed graph $G$ with $n$ vertices and $m$ edges.
**Output:** An Eulerian cycle, incrementally written.

```
1:  next[1..n], visited[1..n], skipped[1..n], B[1..n] ← [0]^n
2:  c ← 0
3:  v_0 ← k ∈ [n]                                  ▷ Pick an arbitrary starting vertex.
4:  u ← v_0                                        ▷ Initially, the current vertex is the starting one.
5:  visited[u] ← 1
6:  while c < m do                                 ▷ We loop as long as there are still edges to write.
7:      next[u] ← next[u] + 1
8:      i ← next[u]
9:      if i ≤ d⁻(u) then                          ▷ Reverse traversal phase.
10:         v ← Γ⁻(u, i)
11:         if visited[v] = 0 then
12:             if v ≠ v_0 then
13:                 B[v] ← u
14:             visited[v] ← 1
15:         u ← v
16:     else                                       ▷ Backtracking phase.
17:         i ← next[u] − d⁻(u)
18:         if Γ⁺(u, i) = B[u] and skipped[u] = 0 then
19:             skipped[u] ← 1
20:             next[u] ← next[u] + 1
21:             i ← i + 1
22:         if i > d⁺(u) then
23:             v ← B[u]
24:         else
25:             v ← Γ⁺(u, i)
26:         WRITE(uv)
27:         c ← c + 1
28:         u ← v
```

first.[4] Finally, after all other outgoing edges have been used (i.e. when $\texttt{next}[u] > d^+(u) + d^-(u) = d(u)$), the RED edge becomes the only option and is followed. As the algorithm backtracks from vertices, each traversed edge is immediately written to the output.

**4 Analysis.** We now analyse the correctness and complexity of the algorithm. In Subsection 4.1, we mainly use the high-level abstraction involving edge marks (BLACK, RED, GREEN, and DASHED) to prove that the algorithm outputs a valid Eulerian cycle. In Subsection 4.2, we analyse the running time and space usage of the algorithm by referring directly to the pseudocode in Algorithm 3.1.

**4.1 Proof of Correctness.** For a vertex $v$ and a mark $X$, let $d^+(v, X)$ denote the number of outgoing edges from $v$ marked $X$, and define $d^-(v, X)$ analogously as the number of incoming edges to $v$ marked $X$.

DEFINITION 4.1 (Needle vertex). *Let $u$ denote the current vertex of the traversal. A vertex $w$ is a* needle vertex *if and only if:*

- *$w = u$ and $d^+(w, \text{BLACK}) = d^-(w, \text{BLACK})$; or*
- *$w \neq u$ and $d^+(w, \text{BLACK}) = d^-(w, \text{BLACK}) + 1$.*

[4] Since the graph may contain parallel edges, the target vertex $v = B[u]$ could appear multiple times in the multiset $\{\Gamma^+(u, i) : i \in [d^+(u)]\}$. Hence we use `skipped`-array to only skip the first edge $uv$.

Our first claim and corollary establish a crucial invariant of Space-Efficient-Hierholzer by showing that there is exactly one needle vertex throughout the execution of the algorithm. Our follow-up claim states that backtracking occurs from $u$ only when $u$ is a needle vertex.

Lemma 4.2. *At every step of the execution of Space-Efficient-Hierholzer, exactly one of the following statements holds:*

1. *for all $v \in V$, $d^+(v, \text{Black}) = d^-(v, \text{Black})$;*
2. *there are two vertices $a$ and $b$ s.t.:*
   - *$d^+(a, \text{Black}) = d^-(a, \text{Black}) + 1$ and $d^+(b, \text{Black}) + 1 = d^-(b, \text{Black})$,*
   - *for all $v \in V \setminus \{a, b\}$, $d^+(v, \text{Black}) = d^-(v, \text{Black})$, and*
   - *$b$ is the current vertex.*

*Proof.* We prove this claim by induction on the number of traversed edges. The base case is trivial, as no edges have been traversed. Hence, for all vertices $v \in V$, we have $d^+(v, \text{Black}) = d^-(v, \text{Black})$, and the invariant is satisfied.

Now, suppose the invariant holds after $k$ edge traversals. We consider the $(k+1)$st edge traversal, which must either be a forward step over a Black incoming edge, or a backtracking step over a Red or Green outgoing edge, from some vertex $u$.

*Case 1. Statement 1 holds after $k$ edge traversals.* Let $u$ be the current vertex. By the induction hypothesis, we have $d^+(u, \text{Black}) = d^-(u, \text{Black})$. If $d^+(u, \text{Black}) = 0$, the traversal backtracks via an outgoing Red or Green edge, meaning that neither $d^+(v, \text{Black})$ nor $d^-(v, \text{Black})$ of any vertex $v \in V$ is affected, preserving the invariant. Otherwise, if $d^+(u, \text{Black}) > 0$, the traversal walks forward via one incoming Black edge and marks it either Red or Green. Let $w$ be the new current vertex. Vertex $u$ loses one Black incoming edge, and $w$ loses one Black outgoing edge; thus, we have $d^+(u, \text{Black}) = d^-(u, \text{Black}) + 1$ and $d^+(w, \text{Black}) + 1 = d^-(w, \text{Black})$, the measure of Black edges is not affected for vertices in $V \setminus \{u, w\}$, and $w$ becomes the new current vertex, satisfying the invariant.

*Case 2. Statement 2 holds after $k$ edge traversals.* We then have $u = b$. Since $d^+(b, \text{Black}) + 1 = d^-(b, \text{Black})$, the traversal then proceeds by walking forward towards some vertex $x$. Consider two possible subcases.

*Subcase 2.1. $x = a$.* By entering $a$ from $b$, $b$ loses one Black incoming edge, $a$ loses one Black outgoing edge, and thus we have $d^+(v, \text{Black}) = d^-(v, \text{Black})$ for all $v \in V$.

*Subcase 2.2. $x \neq a$.* Before entering $x$ from $b$, we have $d^+(x, \text{Black}) = d^-(x, \text{Black})$; upon entering it, we have $d^+(x, \text{Black}) + 1 = d^-(x, \text{Black})$, $d^+(b, \text{Black}) = d^-(b, \text{Black})$, $d^+(a, \text{Black}) = d^-(a, \text{Black}) + 1$, and $x$ is the new current vertex. ∎

The following corollary immediately follows from Lemma 4.2 and Definition 4.1.

Corollary 4.3. *At every step of the execution of Space-Efficient-Hierholzer, there is exactly one needle vertex.*

Lemma 4.4. *At every step of the execution of Space-Efficient-Hierholzer, backtracking occurs from $u$ only when $u$ is a needle vertex.*

*Proof.* It is established in Corollary 4.3 that exactly one needle vertex exists in the graph throughout the execution of the algorithm; thus, consider $w$ to be the needle vertex, and for the sake of contradiction, suppose that the algorithm backtracks from $u \neq w$. By Lemma 4.2, we arrive at two distinct cases.

*Case 1. Statement 1 of Lemma 4.2 holds.* Before backtracking, we have $d^+(v, \text{Black}) = d^-(v, \text{Black})$ for all $v \in V$. Since there is exactly one needle vertex, then it must be the current vertex $u$ by Definition 4.1, which is in contradiction with the assumption that the current vertex is not the needle vertex.

*Case 2. Statement 2 of Lemma 4.2 holds.* We then have $d^+(u, \text{Black}) + 1 = d^-(u, \text{Black})$, implying that prior to backtracking from $u$, there existed a Black incoming edge incident on $u$, which is not possible by the definition of the algorithm.

This completes the proof. ∎

Lemma 4.5. *The needle vertex changes only upon backtracking.*

*Proof.* Towards a contradiction, suppose otherwise; that is, assume that a forward step into $v$ makes $v$ the needle vertex. Before this forward step, we had $d^+(v, \textsc{Black}) = d^-(v, \textsc{Black})$. After the step, however, $d^+(v, \textsc{Black}) + 1 = d^-(v, \textsc{Black})$. Since $v$ is now the current vertex and $d^+(v, \textsc{Black}) \neq d^-(v, \textsc{Black})$, this contradicts Definition 4.1. □

*Remark* 4.6. A consequence of Lemmata 4.4 and 4.5 is that backtracking occurs for the first time at $v_0$ (the first needle vertex), and then makes a vertex incident on $v_0$ the new needle vertex. Observe that this process indeed forms a trail of Dashed edges in the graph.

We now show that this trail is closed at termination, i.e. forms a cycle where $v_0$ is included. To do so, it suffices to show that the algorithm terminates and that $v_0$ is the last current vertex.

To prove termination, we fix the cost function $c : M \to \mathbb{N} \cup \{0\}$ s.t. $M = \{\textsc{Black}, \textsc{Red}, \textsc{Green}, \textsc{Dashed}\}$, $c(\textsc{Black}) = 3$, $c(\textsc{Red}) = 2$, $c(\textsc{Green}) = 2$, and $c(\textsc{Dashed}) = 0$.

Lemma 4.7. *The algorithm Space-Efficient-Hierholzer terminates.*

*Proof.* Let $\Phi$ be a potential function defined as $\Phi(G) = \sum_{e \in E} c(m(e))$ for a graph $G = (V, E)$, where $m(e)$ denotes the mark of edge $e$; in other words, $\Phi$ is equal to the total sum of the costs of all edges in $G$. On an input graph $G$, we show that at each step of the traversal, the measure $\Phi(G)$ strictly decreases.

Initially, all edges are Black, hence $\Phi(G) = 3|E|$. Observe that each edge changes its mark over time in a fixed and monotonic order: $\textsc{Black} \to \{\textsc{Red}, \textsc{Green}\} \to \textsc{Dashed}$. Since each step forward or backwards in the traversal alters the mark of an edge, and no Red or Green edge is remarked Green or Red respectively, the measure strictly decreases at each step, and the algorithm eventually terminates as $\Phi(G) \geq 0$. □

Lemma 4.8. *Upon termination of Space-Efficient-Hierholzer, the Dashed edges form a closed trail which includes $v_0$ as an endpoint.*

*Proof.* By Remark 4.6, the algorithm first backtracks from $v_0$. Observe that, by the definition of an Eulerian graph, entering a non-starting vertex via an outgoing Black edge (which is then marked Red or Green) guarantees that there is a corresponding incoming Black edge that will eventually be used to exit the vertex and marked accordingly. Similarly, entering a non-starting vertex via a Red or Green outgoing edge (which is then marked Dashed) ensures that there is a corresponding incoming Red or Green edge that will eventually be used to exit the vertex and also be marked Dashed. Therefore, at every step of the algorithm, if $u$ is the current vertex, the following conditions hold:

- $d^+(v, \textsc{Red}) + d^+(v, \textsc{Green}) = d^-(v, \textsc{Red}) + d^-(v, \textsc{Green})$ for all $v \in V \setminus \{v_0, u\}$, and
- $d^+(u, \textsc{Red}) + d^+(u, \textsc{Green}) = d^-(u, \textsc{Red}) + d^-(u, \textsc{Green}) + 1$ if $u \neq v_0$.

Towards a contradiction, suppose now that the last current vertex is $v \neq v_0$ upon termination. This implies that there was no Red or Green outgoing edge to backtrack through from $v$, hence the termination. However, if $v$ were entered via a Black incoming edge, then, by the observation above, there would have been a Black incoming edge to resume the traversal, a contradiction. If, analogously, $v$ were entered via a Green or Red edge, then there would be at least one Red or Green outgoing edge to backtrack through, a contradiction. Therefore, the last current vertex cannot be in $V \setminus \{v_0\}$, and can only be $v_0$ since the algorithm terminates by Lemma 4.7. □

We now need to show that the closed trail produced by the algorithm includes all edges in the graph; it suffices to show that all edges are Dashed upon termination. First, we show that there are no Black edges at termination, indicating that all edges have been traversed at least once. The following structural property of Eulerian graphs comes in handy.

Lemma 4.9. *Let $G$ be an Eulerian graph, and let $T$ be some arbitrary closed trail in $G = (V, E)$. Then, $G' = (V, E \setminus T)$ consists of multiple (possibly none) vertex-disjoint closed trails.*

*Proof.* Initially, for all $v \in V$, we have $d^+(v) = d^-(v)$. Clearly, removing the trail $T$ preserves that equality. Let $H$ be some arbitrary weakly[5] connected component of $G'$; then, for all vertices $v_H \in H$, we have $d^+(v_H) = d^-(v_H)$. Hence, $H$ is Eulerian and its edges form a closed trail. □

[5] A set of vertices is considered *weakly connected* if there exists a path of edges from every pair of vertices within that set when edge directions are ignored.

Lemma 4.10. *Upon termination of Space-Efficient-Hierholzer, no Black edges remain.*

*Proof.* Let $T$ be the trail of Dashed edges computed by Space-Efficient-Hierholzer by Lemma 4.8; for the sake of contradiction, we split the proof into two distinct assumptions.

*Case 1. A Black edge is incident on a vertex $v \in T$.* If $v$ has a Black incoming edge, this implies that the algorithm backtracked from or terminated at $v$ despite there being a Black incoming edge to traverse, which is a contradiction. If, on the other hand, the Black edge is outgoing from $v$, then by Lemma 4.2, there must also be an incoming Black edge at $v$. Indeed, Statement 1 of the lemma applies, since the current vertex cannot possess incoming Black edges without violating the algorithm's definition (i.e. backtracking would have occurred instead of termination). Therefore, the previous subcase applies.

*Case 2. No Black edge is incident on a vertex $v \in T$.* By Lemma 4.9, we can remove $T$ and obtain multiple Eulerian connected components, each containing a closed trail. Let $H_1, H_2, \ldots, H_k$ be the $k$ vertex-disjoint strongly connected components obtained. Now, define $S_i = V_T \cap H_i$, where $V_T$ is the set of endpoints in $T$; in other words, $S_i$ is the set of all vertices that are part of both the trail and the $i$th connected component. We claim that for each $S_i$ for $i \in [k]$, there is at least one vertex $w \in S_i$ such that $d^+(w, \textsc{Green}) \geq 1$, and $d^+(w, \textsc{Red}) = 0$.

First, observe that the Red edges marked by Space-Efficient-Hierholzer form a tree rooted at $v_0$, the starting vertex; hence $d^+(v, \textsc{Red}) \leq 1$ for any vertex $v$. Pick an arbitrary set $S_i$: using the invariant from the proof of Lemma 4.8, for each $w \in S_i$, we have $d^+(w, \textsc{Red}) + d^+(w, \textsc{Green}) = d^-(w, \textsc{Red}) + d^-(w, \textsc{Green}) > 0$. If $v_0 \in S_i$, then we are done since $d^+(v_0, \textsc{Red}) = 0$ and thus $d^+(v_0, \textsc{Green}) \geq 1$. Now, towards a contradiction, suppose that we have $d^+(w, \textsc{Red}) = 1$ for all $w \in S_i$ (recall that this measure cannot be greater than 1), and $v_0 \notin S_i$. Since $H_i$ does not share vertices with other components in $\biguplus_{j \in [k] \setminus \{i\}} H_j$, there is no path from a vertex in $H_i \setminus S_i$ to $v_0$ that does not include an outgoing edge incident on a vertex in $S_i$. Thus, the assumption entails that the starting vertex (i.e. the root of the tree of Red edges) $v_0 \in H_i \setminus S_i$, which contradicts our assumption that $v_0 \in T$ since $T \cap (H_i \setminus S_i) = \emptyset$.

Now, as we have shown that a vertex $w \in T$ such that $d^+(w, \textsc{Green}) \geq 1$ and $d^+(w, \textsc{Red}) = 0$ exists, we can pick one arbitrarily and consider two distinct cases. If $w = v_0$, then the algorithm terminated at $v_0$ despite $d^+(v_0, \textsc{Green}) \geq 1$, which is impossible by the way the algorithm is defined. Finally, if $w \neq v_0$, then the algorithm backtracked via a Red outgoing edge (later marked Dashed) despite there being a Green outgoing edge to backtrack through, which contradicts the definition of the algorithm.

In either case, we reach a contradiction; therefore, no Black edges remain in the graph. ∎

The following corollary strictly follows from Case 2 of the proof of Lemma 4.10.

Corollary 4.11. *Upon termination of Space-Efficient-Hierholzer, all edges are Dashed.*

*Proof.* For the sake of contradiction, suppose that, at termination, the graph contains Red or Green edges; recall that it has been established in Lemma 4.10 that the graph cannot contain Black edges. Let $T$ be the trail of Dashed edges produced by the algorithm (Lemma 4.8). Then, by the argument of Case 2 of the proof of Lemma 4.10, there must exist some vertex $w \in T$ such that $d^+(w, \textsc{Green}) \geq 1$ and $d^+(w, \textsc{Red}) = 0$, which leads to contradicting the definition of the algorithm. Therefore, all edges are Dashed at termination. ∎

Theorem 4.12. *The algorithm Space-Efficient-Hierholzer correctly computes an Eulerian cycle.*

*Proof.* By Lemma 4.7, the algorithm terminates after a finite number of steps. Lemma 4.8 establishes that, upon termination, the set of Dashed edges forms a closed trail containing the starting vertex $v_0$. Moreover, Lemma 4.5 and Lemma 4.4 together ensure that a new Dashed edge is created exactly when the traversal backtracks from the current needle vertex, and that the next needle vertex is incident to the previously Dashed one, thus producing the trail in a correct order. Finally, Corollary 4.11 implies that all edges are marked Dashed upon termination, which ensures that the closed trail includes every edge exactly once. ∎

This completes the proof of correctness.

**4.2 Space and Time Complexity.** Recall that we assume computing the $i$th in- or out-neighbour of a vertex, or its in- or out-degree, takes $O(1)$ time.

Theorem 4.13. *The algorithm Space-Efficient-Hierholzer (Algorithm 3.1) terminates in time $O(m)$, where $m$ is the number of edges in the input graph.*

*Proof.* Termination follows from Lemma 4.7. Initialising the arrays `next`, `visited`, `skipped` and $B$ takes $O(n)$ time. We claim that the **while** loop (Lines 6-28) iterates $2m$ times. The loop terminates when $c \geq m$; and during an iteration, $c$ is incremented if and only if $\texttt{next}[u] > d^-(u)$. Since the latter is incremented at each iteration of the loop, the **if** statement (Line 9) is satisfied exactly $\sum_{v \in V} d^-(v) = m$ times. Thus, the **else** block (Line 16) is executed $m$ times.

The overall complexity is $O(n + m) \in O(m)$ since, in any Eulerian graph of 2 or more vertices, $n \leq m$. □

THEOREM 4.14. *The algorithm SPACE-EFFICIENT-HIERHOLZER (Algorithm 3.1) uses* $O(n \lg m)$ *bits of working space.*

*Proof.* The arrays `visited` and `skipped` take $O(n)$ bits, as these represent bit vectors. Values in the arrays `next` and $B$ are bounded by $m$, and thus the arrays take $O(n \lg m)$ bits. □

**5 Conclusion.** We presented a simple and memory-efficient algorithm for computing Eulerian cycles in directed graphs. Unlike classical implementations of Hierholzer's algorithm, which require storing all edges or large stacks, our method uses only $O(n \lg m)$ bits of working space by storing a small amount of data per vertex rather than per edge. Despite the reduced memory usage, our algorithm still runs in linear time and produces the Eulerian cycle incrementally, without needing to reverse it at the end. Our approach is especially useful for dense graphs, where the number of edges is much larger than the number of vertices, and in settings where memory is limited. It is also well-suited for environments such as graph rewriting systems that do not support stacks or pointer-based data structures.

Our approach focuses on directed graphs, but the underlying principle can also be applied to undirected graphs. A nuisance not present in directed graphs is that an edge might have been traversed in the opposite direction when we now consider it, so we have to detect this. When the undirected graph is simple (i.e. no parallel edges) and the adjacency lists are sorted (that is, for all $i < j \in [d(v)]$, we have $\Gamma(v, i) < \Gamma(v, j)$ for all $v \in V$), we can determine whether an edge $uv$ incident at $u$ has already been traversed via $\Gamma(v, \texttt{next}[v]) \geq u$.

Extending the algorithm to handle unsorted adjacency lists and undirected multigraphs, while preserving or improving the current time and space complexity, remains open for future work.

**Acknowledgements.** We would like to thank Viktor Zamaraev and Nikhil Mande for the initial discussions on the topic.

**References**

[1] *`eulerian_circuit` in NetworkX.* https://networkx.org/documentation/stable/_modules/networkx/algorithms/euler.html#eulerian_circuit. Accessed on 06/08/2025.

[2] *`HierholzerEulerianCycle.java` in JGraphT.* https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/cycle/HierholzerEulerianCycle.java. Accessed on 06/08/2025.

[3] C. GLAZIK, J. SCHIEMANN, AND A. SRIVASTAV, *A one pass streaming algorithm for finding Euler tours*, Theory of Computing Systems, 67 (2023), pp. 671–693.

[4] T. HAGERUP, F. KAMMER, AND M. LAUDAHN, *Space-efficient euler partition and bipartite edge coloring*, Theoretical Computer Science, 754 (2019), pp. 16–34.

[5] C. HIERHOLZER AND C. WIENER, *Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren*, Mathematische Annalen, 6 (1873), pp. 30–32.

[6] C. S. ORLOFF, *A Fundamental Problem in Vehicle Routing*, Networks, 4 (1974), pp. 35–64.

[7] P. A. PEVZNER, H. TANG, AND M. S. WATERMAN, *An Eulerian path approach to DNA fragment assembly*, Proceedings of the national academy of sciences, 98 (2001), pp. 9748–9753.

[8] D. PLUMP, *The design of GP 2*, in Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011), vol. 82 of Electronic Proceedings in Theoretical Computer Science, 2012, pp. 1–16, https://doi.org/10.4204/EPTCS.82.1.

[9] K. Yamamoto, J. V. S. Luces, K. Shirasu, Y. Hoshikawa, T. Okabe, and Y. Hirata, *A Novel Single-Stroke Path Planning Algorithm for 3D Printers using Continuous Carbon Fiber Reinforced Thermoplastics*, Additive Manufacturing, 55 (2022), p. 102816.