# A Walkthrough of Eulerian Circuit Algorithms

## From Fleury's to Space-Efficient Hierholzer

Atisahya Maharjan

GADA Lab

February 4, 2026

# Table of Contents

1. Introduction
2. Fleury's Algorithm
3. Classical Hierholzer
4. Space-Efficient Hierholzer
5. Hypergraphs
6. Open Questions

# What is an Eulerian circuit?

**Definition**

An **Eulerian circuit** is a closed walk that traverses every edge exactly once.

# What is an Eulerian circuit?

**Definition**

An **Eulerian circuit** is a closed walk that traverses every edge exactly once.

**Undirected Graphs**

- Graph is connected (ignoring isolated vertices).
- Every vertex has even degree.

# What is an Eulerian circuit?

**Definition**

An **Eulerian circuit** is a closed walk that traverses every edge exactly once.

**Undirected Graphs**

- Graph is connected (ignoring isolated vertices).
- Every vertex has even degree.

**Directed Graphs**

- All vertices with nonzero degree lie in one strongly connected component.
- For every vertex $v$: $\deg^{+}(v) = \deg^{-}(v)$.

# The Goal

Find a circuit that visits every edge exactly once.

# Time

$O(m)$ (Linear in edges)

# The Challenge

**Time**

$O(m)$ (Linear in edges)

**Space**

$O(n)$ (Linear in vertices)

# The Challenge

**Time**

$O(m)$ (Linear in edges)

**Space**

$O(n)$ (Linear in vertices)

**Why is $O(n)$ space hard?** We cannot store "visited" flags for edges ($m$ bits). We cannot store the output path in memory ($m$ words).

# Table of Contents

1 Introduction

2 Fleury's Algorithm

3 Classical Hierholzer

4 Space-Efficient Hierholzer

5 Hypergraphs

6 Open Questions

# Attempt 1: The Greedy Approach

Fleury's Algorithm (1883) **Fleury1883**

# Fleury's Rule

"Don't cross a bridge unless you have to."

**Definition (Bridge)**

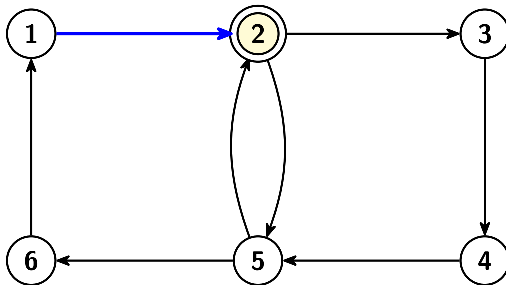
An unused edge $e = uv$ is called a **bridge** (with respect to the current set of unused edges) if, when you remove $e$ from the unused subgraph, that subgraph becomes disconnected. Specifically, $u$ and $v$ are no longer reachable from each other via other unused edges.

# Fleury's Algorithm (Pseudocode) **Fleury1883**

```
Input   : Graph G = (V, E), Start Node u
Output: Eulerian Trail (printed sequence)
while G has edges do
    for each neighbor v of u do
        if |edges from u| == 1 or not IsBridge(u, v) then
            print(u → v);
            Remove edge (u, v) from G;
            u ← v;
            break;
        end
    end
end
```

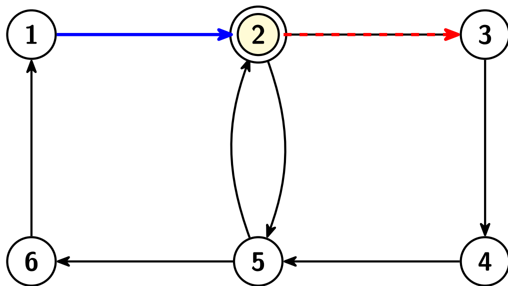
**Algorithm 1:** Fleury's Algorithm

# Fleury's Algorithm



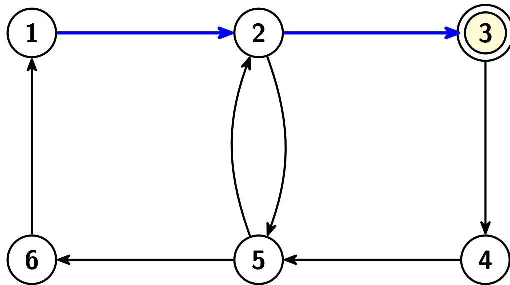

Start at 1. Move $1 \to 2$. **Check:** Is $(1, 2)$ a bridge? No. OK.

# Fleury's Algorithm



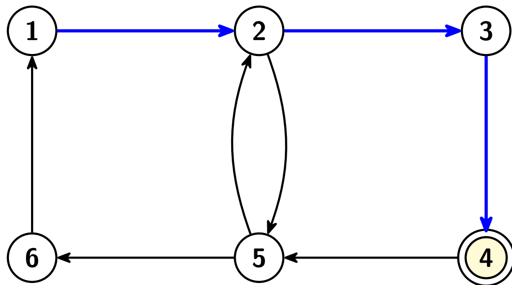

At 2. Options: $2 \to 3$ or $2 \to 5$. Is $2 \to 3$ a bridge? No. Take that edge.

# Fleury's Algorithm



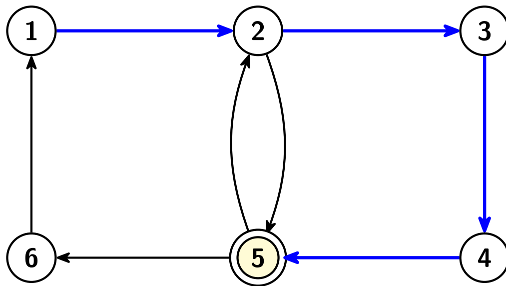

Move 2 $\rightarrow$ 3.

# Fleury's Algorithm



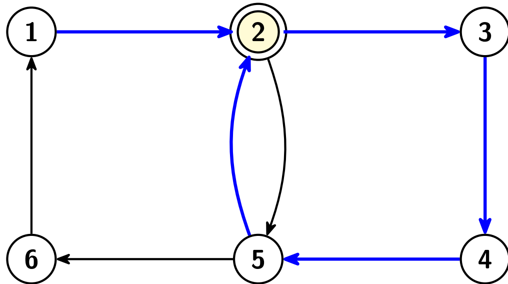

At 3, only option: $3 \to 4$.

# Fleury's Algorithm



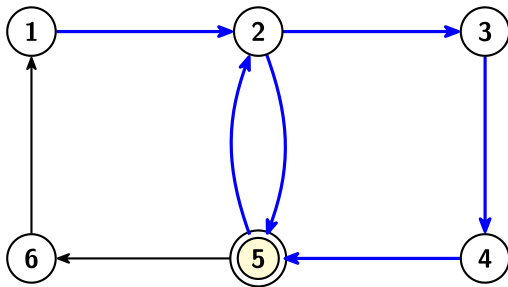

At 4, only option: $4 \to 5$.

# Fleury's Algorithm



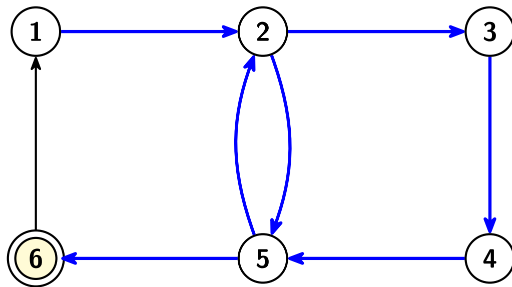

At 5, take $5 \to 2$.

# Fleury's Algorithm



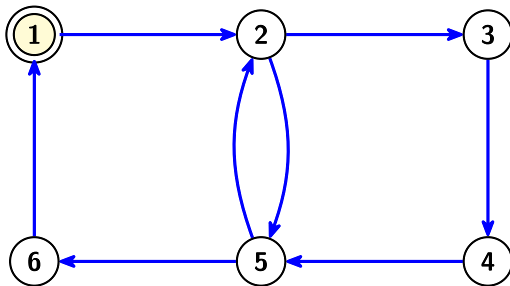

At 2, take $2 \to 5$.

# Fleury's Algorithm




At 5, take $5 \to 6$.

# Fleury's Algorithm



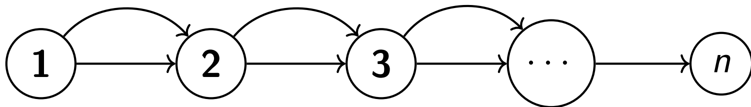

At 6, take $6 \to 1$. **Eulerian circuit complete.**

# Why is Fleury's Algorithm Bad?

Checking "Is this a bridge?" takes $O(m)$ time. Need either DFS or BFS to check connectivity.

# Why is Fleury's Algorithm Bad?

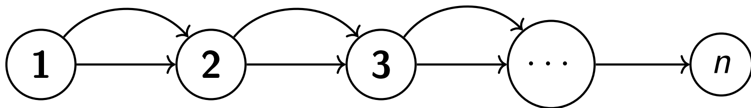
Checking "Is this a bridge?" takes $O(m)$ time. Need either DFS or BFS to check connectivity.




**Bridge detection needed at every vertex!**

# Why is Fleury's Algorithm Bad?

Checking "Is this a bridge?" takes $O(m)$ time. Need either DFS or BFS to check connectivity.




**Bridge detection needed at every vertex!**

Total Time: $\Theta(m \times m) = \Theta(m^2)$.

# Table of Contents

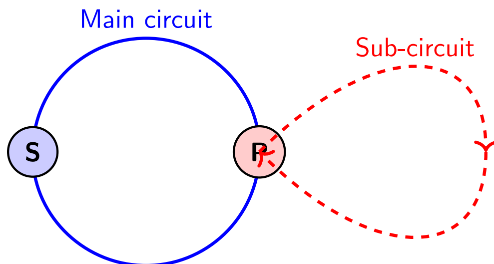

1. Introduction
2. Fleury's Algorithm
3. Classical Hierholzer
4. Space-Efficient Hierholzer
5. Hypergraphs
6. Open Questions

# Attempt 2: Circuit Splicing

Classical Hierholzer (1873) **Hierholzer1873**

# The Strategy: Stitching Detours

**Core Idea:** Walk a path until you return to start. If you missed edges, take a detour from a visited node.




1. Walk Blue Path
$S \to \cdots \to P \to \cdots \to S$

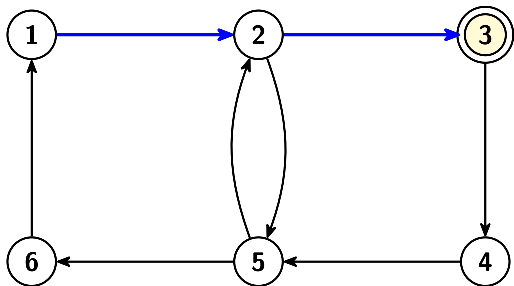
2. Discover Unused Edges at $P$

3. Insert Red Path

# Hierholzer Algorithm (Stack Based) **Hierholzer1873**

```
Input   : Graph G = (V, E), Start Node start
Output: Eulerian Circuit (List)
curr_path.push(start);
circuit ← empty list;
while curr_path is not empty do
    u ← curr_path.top();
    if u has outgoing edges then
        v ← pick first neighbor of u;
        Remove edge (u, v) from G;
        curr_path.push(v);
    else
        /* No edges left:  Backtrack and add to circuit */
        circuit.append(u);
        curr_path.pop();
    end
end
return reverse(circuit);
```

# Classical Hierholzer Algorithm

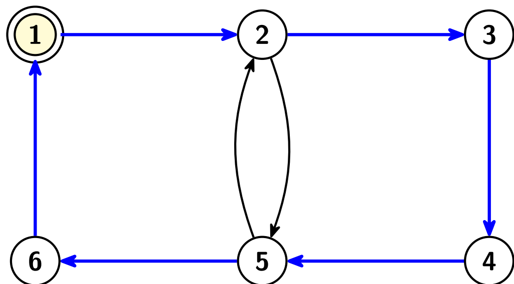



**Stack Memory ($O(m)$)**

- 1
- 2
- **3** $\leftarrow$ Top

**Step 1:** Start at Node 1. Greedily follow edges ($1 \to 2 \to 3 \ldots$) pushing to stack.

# Classical Hierholzer Algorithm

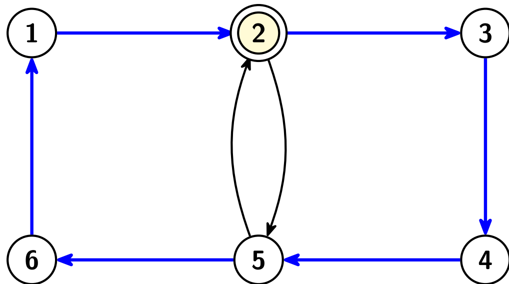



**Stack Memory ($O(m)$)**

- 1
- 2
- 3 .. 6
- **1** (Stuck!)

**Step 2:** Returned to Node 1. Node 1 has no unused edges. **Circuit Detected.**

# Classical Hierholzer Algorithm

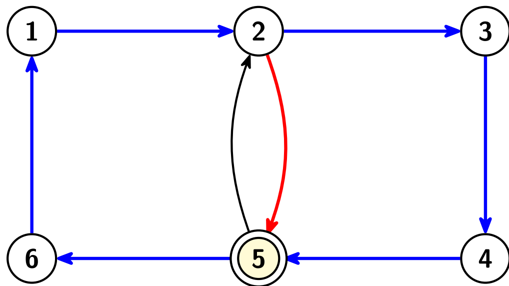



**Stack Memory ($O(m)$)**

- 1
- **2** $\leftarrow$ Check
- 3 .. 6 .. 1

**Step 3:** Scan the path for nodes with unused edges. **Node 2** still has neighbors ($2 \to 5$).

# Classical Hierholzer Algorithm

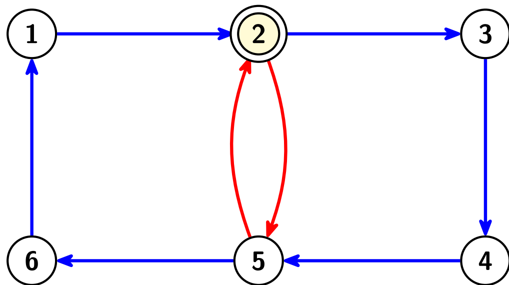



**Stack Memory ($O(m)$)**

- 1
- 2
- $\hookrightarrow$ 5
- 3 .. 6 .. 1

**Step 4:** Start a new detour from Node 2. Push 2 $\rightarrow$ 5 onto the stack at the insertion point.

# Classical Hierholzer Algorithm




**Stack Memory ($O(m)$)**

- 1
- 2
- $\hookrightarrow$ 5
- $\hookrightarrow$ 2 (Closed)
- 3 .. 6 .. 1

**Step 5:** Sub-circuit closed ($5 \rightarrow 2$). Node 2 has no more edges. **Result:** circuits are spliced together.

# The Problem

We are storing the **entire path** in the stack.

# The Problem

We are storing the **entire path** in the stack.

Space: $O(m)$ words.

# The Problem

We are storing the **entire path** in the stack.

Space: $O(m)$ words.

Can we do it with $O(n)$ (just vertices)?

# Table of Contents

1 Introduction

2 Fleury's Algorithm

3 Classical Hierholzer

4 Space-Efficient Hierholzer

5 Hypergraphs

6 Open Questions

# The Solution: Reverse Search

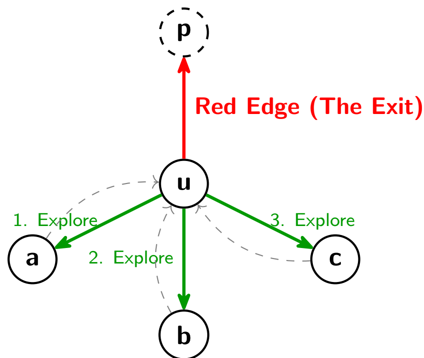
Ismaili Alaoui, Plump & Wild (2026) **IsmailiAlaoui2026**

# The Core Intuition: The “Save for Last” Rule

How do we ensure we visit every sub-circuit without a stack?

# The Core Intuition: The "Save for Last" Rule

How do we ensure we visit every sub-circuit without a stack?




- The **Reverse Search** builds a "Red Edge" pointing home for every node.
- **The Rule:** You are **forbidden** from taking the Red Edge until all Green Edges are gone. This forces the algorithm to fully explore all local sub-circuits before returning to the parent.

```
Input  : Graph G, Start Node u
Output: Stream of Edges
foreach v ∈ V do
    B[v] ← null; next[v] ← 0; vis[v] ← false; skip[v] ← false
end
;
vis[u] ← true;
while true do
    /* Phase 1:  Reverse Search                                    */
    found ← false;
    foreach (v → u) incoming do
        if ¬vis[v] then
            B[v] ← u; vis[v] ← true; u ← v; found ← true; break;
        end
    end
    if found then continue ;
    /* Phase 2:  Forward                                           */
    while next[u] < deg(u) do
        w ← adj[u][next[u]]; next[u]++;
        if w = B[u] then
            skip[u] ← true;
        else
            output(u → w); u ← w; goto Start;
        end
    end
    /* Phase 3:  Backtrack                                         */
    if skip[u] then
        output(u → B[u]); u ← B[u]
    end
    else
        break
    end
end
```

**Algorithm 2:** Space-Efficient Hierholzer

# Introducing the 4 Arrays

Instead of a stack, we maintain state **per vertex**.

- `B[v]` **(Backtrack Pointer)**: Stores the "Parent" vertex that discovered $v$ in the reverse search. This forms a Spanning Tree.

# Introducing the 4 Arrays

Instead of a stack, we maintain state **per vertex**.

- `B[v]` **(Backtrack Pointer)**: Stores the "Parent" vertex that discovered $v$ in the reverse search. This forms a Spanning Tree.
- `next[v]` **(Progress Counter)**: Stores index of the next outgoing edge to process. Will take at most $\log m$ bits.

# Introducing the 4 Arrays

Instead of a stack, we maintain state **per vertex**.

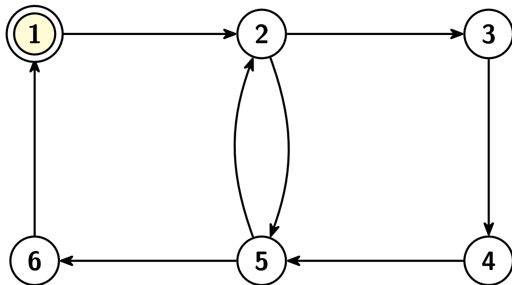
- `B[v]` **(Backtrack Pointer)**: Stores the "Parent" vertex that discovered $v$ in the reverse search. This forms a Spanning Tree.
- `next[v]` **(Progress Counter)**: Stores index of the next outgoing edge to process. Will take at most $\log m$ bits.
- `visited[v]` **(Discovery Flag)**: 1 bit. Ensures we don't visit a node twice during the reverse tree-building phase.

# Introducing the 4 Arrays

Instead of a stack, we maintain state **per vertex**.

- `B[v]` **(Backtrack Pointer)**: Stores the "Parent" vertex that discovered $v$ in the reverse search. This forms a Spanning Tree.
- `next[v]` **(Progress Counter)**: Stores index of the next outgoing edge to process. Will take at most $\log m$ bits.
- `visited[v]` **(Discovery Flag)**: 1 bit. Ensures we don't visit a node twice during the reverse tree-building phase.
- `skipped[v]` **(Constraint Flag)**: 1 bit. Marks if we have temporarily skipped the "Red Edge" (parent edge) to process it last.

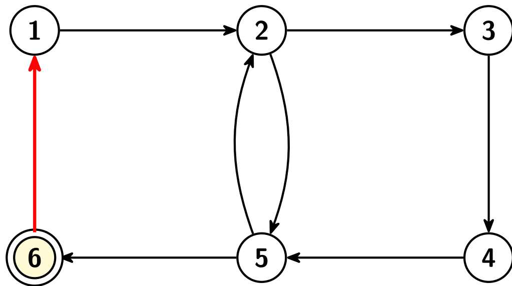
# Space-Efficient Hierholzer Algorithm




Start at Node 1.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | · | · | · | · | · |
| next[v] | 0 | 0 | 0 | 0 | 0 | 0 |
| visited[v] | T | F | F | F | F | F |
| skipped[v] | F | F | F | F | F | F |

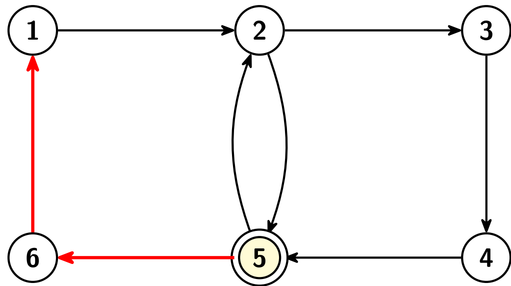
# Space-Efficient Hierholzer Algorithm




Reverse: $6 \to 1$. Set $B[6] = 1$.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | · | · | · | · | 1 |
| next[v] | 0 | 0 | 0 | 0 | 0 | 0 |
| visited[v] | T | F | F | F | F | T |
| skipped[v] | F | F | F | F | F | F |

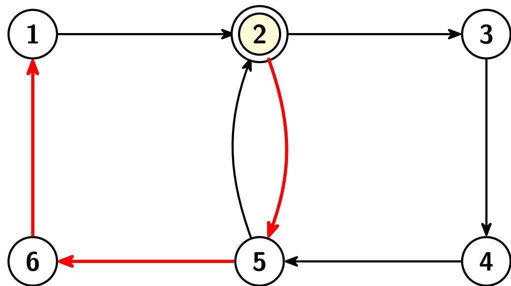
# Space-Efficient Hierholzer Algorithm




Reverse: $5 \to 6$. Set $B[5] = 6$.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | · | · | · | 6 | 1 |
| next[v] | 0 | 0 | 0 | 0 | 0 | 0 |
| visited[v] | T | F | F | F | T | T |
| skipped[v] | F | F | F | F | F | F |

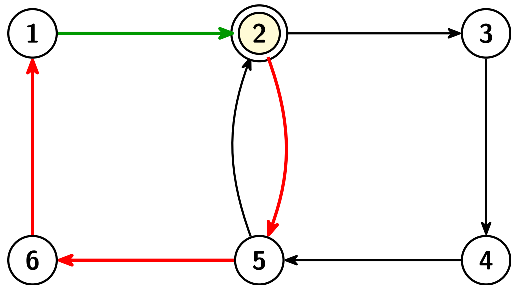
# Space-Efficient Hierholzer Algorithm




Reverse: $2 \to 5$. Set $B[2] = 5$.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | · | · | 6 | 1 |
| next[v] | 0 | 0 | 0 | 0 | 0 | 0 |
| visited[v] | T | T | F | F | T | T |
| skipped[v] | F | F | F | F | F | F |

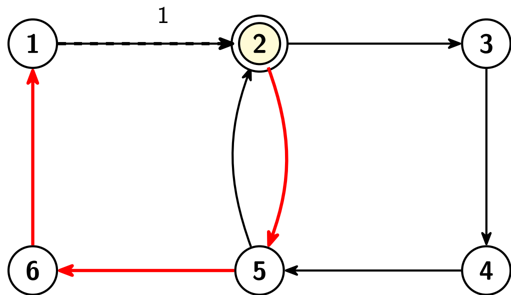
# Space-Efficient Hierholzer Algorithm




Forward: $1 \to 2$. Output 1. Move to 2.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | · | · | 6 | 1 |
| next[v] | 1 | 0 | 0 | 0 | 0 | 0 |
| visited[v] | T | T | F | F | T | T |
| skipped[v] | F | F | F | F | F | F |

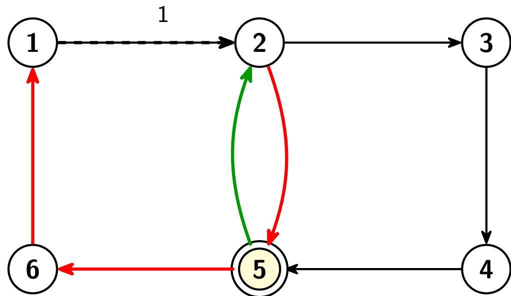
# Space-Efficient Hierholzer Algorithm




At 2: Edge $2 \to 5$ is Parent ($B[2] = 5$). Skip!

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | · | · | 6 | 1 |
| next[v] | 1 | 1 | 0 | 0 | 0 | 0 |
| visited[v] | T | T | F | F | T | T |
| skipped[v] | F | T | F | F | F | F |

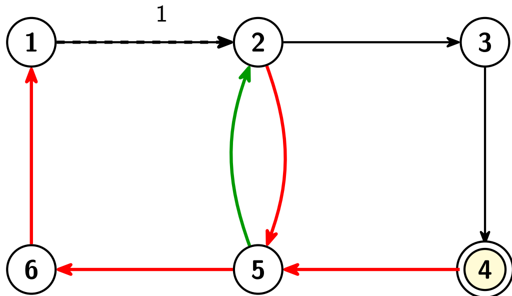
# Space-Efficient Hierholzer Algorithm




At 2: Reverse Search logic triggers for inner circuit.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | · | · | 6 | 1 |
| next[v] | 1 | 1 | 0 | 0 | 0 | 0 |
| visited[v] | T | T | F | F | T | T |
| skipped[v] | F | T | F | F | F | F |

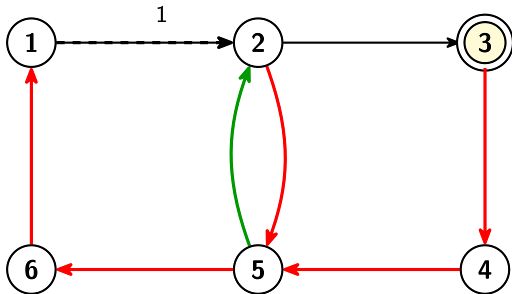
# Space-Efficient Hierholzer Algorithm




Reverse: $4 \to 5$. Set $B[5] = 4$. (Inner Tree)

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | · | · | 4 | 1 |
| next[v] | 1 | 1 | 0 | 0 | 0 | 0 |
| visited[v] | T | T | F | T | T | T |
| skipped[v] | F | T | F | F | F | F |

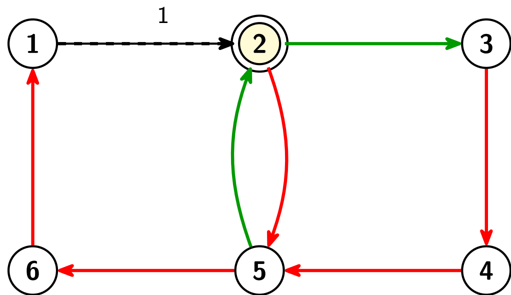
# Space-Efficient Hierholzer Algorithm




Reverse: $3 \to 4$. Set $B[4] = 3$.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **B[v]** | · | 5 | 2 | 3 | 4 | 1 |
| **next[v]** | 1 | 1 | 0 | 0 | 0 | 0 |
| **visited[v]** | T | T | T | T | T | T |
| **skipped[v]** | F | T | F | F | F | F |

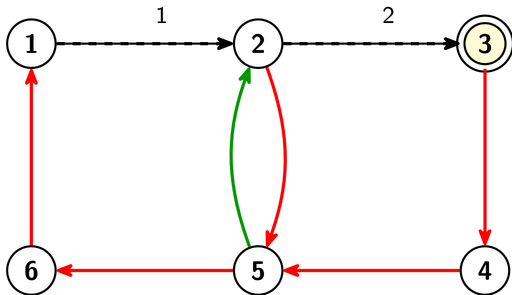
# Space-Efficient Hierholzer Algorithm




Forward: Found 2 $\to$ 3 (Green). Inner circuit Closed.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | 2 | 3 | 4 | 1 |
| next[v] | 1 | 1 | 0 | 0 | 0 | 0 |
| visited[v] | T | T | T | T | T | T |
| skipped[v] | F | T | F | F | F | F |

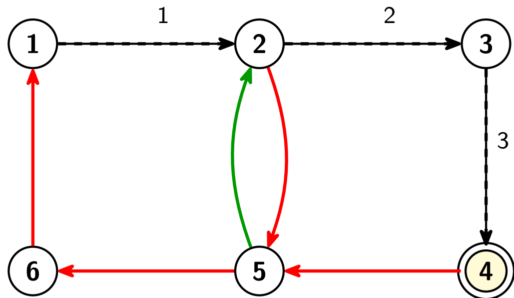
# Space-Efficient Hierholzer Algorithm




Forward: Output 2 → 3 (Output 2). Move to 3.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | 2 | 3 | 4 | 1 |
| next[v] | 1 | 2 | 0 | 0 | 0 | 0 |
| visited[v] | T | T | T | T | T | T |
| skipped[v] | F | T | F | F | F | F |

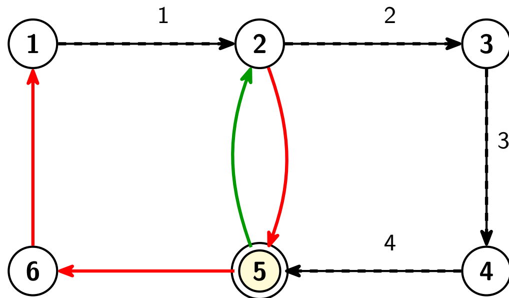
# Space-Efficient Hierholzer Algorithm




Forward: Output $3 \to 4$. Move to 4.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | 2 | 3 | 4 | 1 |
| next[v] | 1 | 2 | 1 | 0 | 0 | 0 |
| visited[v] | T | T | T | T | T | T |
| skipped[v] | F | T | F | F | F | F |

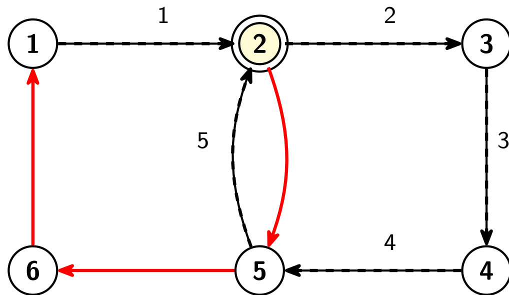
# Space-Efficient Hierholzer Algorithm




Forward: Output 4 $\to$ 5. Move to 5.

| **v** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| **B[v]** | · | 5 | 2 | 3 | 4 | 1 |
| **next[v]** | 1 | 2 | 1 | 1 | 0 | 0 |
| **visited[v]** | T | T | T | T | T | T |
| **skipped[v]** | F | T | F | F | F | F |

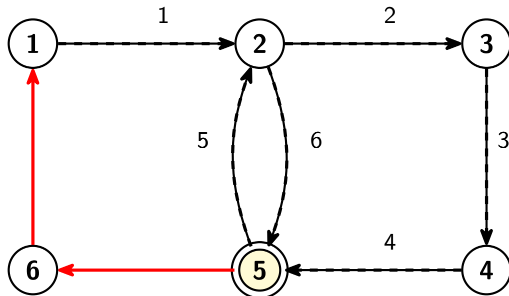
# Space-Efficient Hierholzer Algorithm




Forward: Output 5 $\to$ 2 (Green). Move to 2.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | 2 | 3 | 4 | 1 |
| next[v] | 1 | 2 | 1 | 1 | 1 | 0 |
| visited[v] | T | T | T | T | T | T |
| skipped[v] | F | T | F | F | F | F |

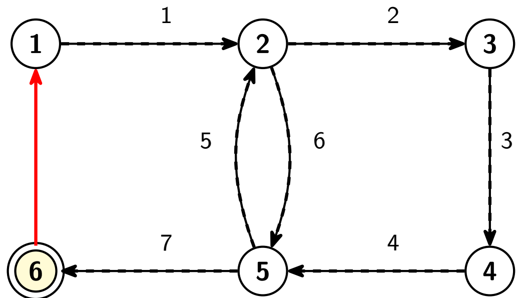
# Space-Efficient Hierholzer Algorithm




Backtrack: Take skipped parent $2 \to 5$.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **B[v]** | · | 5 | 2 | 3 | 4 | 1 |
| **next[v]** | 1 | 2 | 1 | 1 | 1 | 0 |
| **visited[v]** | T | T | T | T | T | T |
| **skipped[v]** | F | T | F | F | F | F |

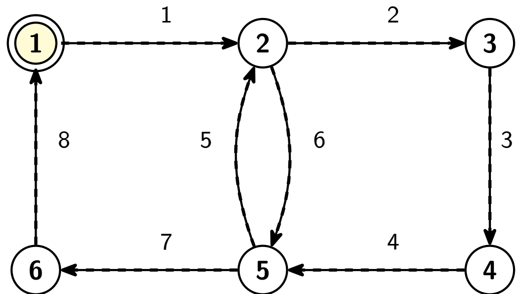
# Space-Efficient Hierholzer Algorithm




Backtrack: Take skipped parent 5 $\rightarrow$ 6.

| **v** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| **B[v]** | · | 5 | 2 | 3 | 4 | 1 |
| **next[v]** | 1 | 2 | 1 | 1 | 2 | 0 |
| **visited[v]** | T | T | T | T | T | T |
| **skipped[v]** | F | T | F | F | T | F |

# Space-Efficient Hierholzer Algorithm




Backtrack: Take skipped parent 6 → 1. Done.

| v | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| B[v] | · | 5 | 2 | 3 | 4 | 1 |
| next[v] | 1 | 2 | 1 | 1 | 2 | 1 |
| visited[v] | T | T | T | T | T | T |
| skipped[v] | F | T | F | F | T | T |

# Intuition: How the Algorithm Works

- We first build the circuit **backwards** along the reverse tree:

$$1 \xleftarrow{\text{reverse}} 6 \leftarrow 5 \leftarrow 2$$

- **Needle vertices** (like 2 here) mark where the forward traversal should resume.
- During forward phase, edges are emitted **parent** $\rightarrow$ **child**:

$$1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 1$$

- We only store $B$ pointers $\rightarrow$ memory $O(n)$, no full stack needed.

*Needle vertices are like "handles" that let us unwind circuits incrementally without storing the full path.*

# Needle Vertex Invariant

**Definition**

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 $d^+(w, \text{Black}) = d^-(w, \text{Black}) + 1$.

There is this invariant of:
At every step of the execution of Space-Efficient-Hierholzer, exactly one of the following statements holds:

- for all $v \in V$, $d^+(v, \text{Black}) = d^-(v, \text{Black})$; or
- there are two vertices $a$ and $b$ such that:
  - $d^+(a, \text{Black}) = d^-(a, \text{Black}) + 1$ 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.

This invariant guarantees that we can always either continue building the reverse search tree or complete a circuit in the forward phase.

# Table of Contents

1 Introduction

2 Fleury's Algorithm

3 Classical Hierholzer

4 Space-Efficient Hierholzer

5 Hypergraphs

6 Open Questions

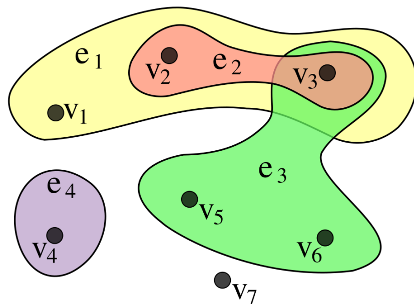
# One possible generalization: Hypergraphs

**Standard Graph:** Edge connects 2 vertices.

**Hypergraph:** Hyperedge connects a *set* of vertices.

### Definition

A **hypergraph** $H = (V, E)$ consists of a set of vertices $V$ and a set of hyperedges $E$, where each hyperedge $e \in E$ is a subset of $V$ (i.e., $e \subseteq V$) that can connect two or more vertices.



# Challenges for Hypergraph Eulerian Cycles

**Problem: No Consensus!**

What is even a **walk** in a hypergraph?

# Challenges for Hypergraph Eulerian Cycles

**Problem: No Consensus!**

What is even a **walk** in a hypergraph?

**Three main schools of thought:**

1. **Vertex sequences:** Walk $= v_0, v_1, \ldots, v_k$ where consecutive vertices share a hyperedge
2. **Alternating sequences:** Walk $= v_0, e_1, v_1, e_2, \ldots$ with $v_{i-1}, v_i \in e_i$
3. **Edge-centric:** Walk defined by hyperedge transitions

# Challenges for Hypergraph Eulerian Cycles

## Problem: No Consensus!

What is even a **walk** in a hypergraph?

**Three main schools of thought:**

1. **Vertex sequences:** Walk $= v_0, v_1, \ldots, v_k$ where consecutive vertices share a hyperedge
2. **Alternating sequences:** Walk $= v_0, e_1, v_1, e_2, \ldots$ with $v_{i-1}, v_i \in e_i$
3. **Edge-centric:** Walk defined by hyperedge transitions

## Complexity

Depending on the definition, determining if a dihypergraph has an Eulerian cycle is **NP-complete**! (**EulerianHypergraph**)

# Restriction: Euler Families in Hypergraphs

## Euler Families

A hypergraph has an **Euler family** if its edges can be covered by disjoint closed walks that jointly traverse each edge once.

**Known complexity results:**

- Existence of an **Euler tour** is NP-complete even for restricted 3-uniform hypergraphs
- Existence of an **Euler family** is **polynomial-time solvable** for general hypergraphs

**Algorithmic idea:** Matching / factor reductions can be used to decide Euler family existence efficiently. **EulerianPropertiesHypergraphs**

# From Euler Families to Traversals

**High-level approach:**

1. **Incidence graph construction:** Transform hypergraph $H$ into its bipartite incidence graph $B(H)$
   - Left: vertices $V$
   - Right: hyperedges $E$
   - $(v, e)$ iff $v \in e$
2. **Pairing / factor selection:** Use matching or $f$-factor techniques on $B(H)$ to select consistent vertex pairings inside hyperedges
3. **Cycle extraction:** The selected pairings induce a family of closed walks covering each hyperedge exactly once

**Remark:** This is still a work in progress; details and space analysis are ongoing. I'm still thinking about the correct formulation of this algorithm/reduction.

# Table of Contents

1 Introduction

2 Fleury's Algorithm

3 Classical Hierholzer

4 Space-Efficient Hierholzer

5 Hypergraphs

6 Open Questions

# Open Questions: Simple Graphs

**For standard graphs:**

1. **Streaming models:** Can we compute Eulerian cycles in streaming with $o(n)$ space?

# Open Questions: Simple Graphs

**For standard graphs:**

1. **Streaming models:** Can we compute Eulerian cycles in streaming with $o(n)$ space?
2. **Unsorted Adjacency Lists:** Can we adapt to unsorted or dynamic adjacency representations?

# Open Questions: Simple Graphs

**For standard graphs:**

1. **Streaming models:** Can we compute Eulerian cycles in streaming with $o(n)$ space?
2. **Unsorted Adjacency Lists:** Can we adapt to unsorted or dynamic adjacency representations?
3. **Undirected Multigraphs:** Can we adapt to undirected multigraphs efficiently?

# Open Questions: Hypergraphs

**For hypergraphs:**

1. **Efficient traversal algorithms:** Can we design space-efficient algorithms to output Euler families in hypergraphs?

# Open Questions: Hypergraphs

**For hypergraphs:**

1. **Efficient traversal algorithms:** Can we design space-efficient algorithms to output Euler families in hypergraphs?
2. **Higher-order structures:** Generalizing to simplicial complexes and directed hypergraphs?

# References I