

Hierholzer's Algorithm

Der Algorithmus von Hierholzer

Michael Emmerich

University of Jyväskylä

17 October 2026 / 17. Oktober 2026

Euler trails, cycle splicing, proof and implementation

Eulerzüge, Einfügen geschlossener Kantenzüge, Beweis und Implementierung

English

An Euler trail traverses every edge of a graph exactly once. Hierholzer's algorithm constructs such a trail by joining smaller trails at shared vertices. The Haus vom Nikolaus illustrates the construction; parity supplies a short correctness proof. An explicit stack gives a linear-time implementation without repeated edges in the resulting drawing.

1. The problem and its precise conditions

Let $G = (V, E)$ be a finite undirected graph with at least one edge. Parallel edges and loops are allowed. Parallel edges have distinct identities, even if their endpoints coincide; a loop contributes two to its vertex's degree. Assume that all vertices incident with an edge belong to one connected component. Isolated vertices have no role in traversing edges and are ignored here.

A walk is a sequence of incident vertices and edges. A trail is a walk that never repeats an edge; vertices may repeat. An Euler trail uses every edge exactly once. It is an Euler circuit if its endpoints coincide, and an open Euler trail otherwise. Thus a drawing in one stroke concerns edges, not a rule that each vertex may be visited only once.

An Euler circuit exists precisely when every vertex has even degree. An open Euler trail exists precisely when exactly two vertices have odd degree; these two vertices must be its endpoints. Together with connectivity, these are necessary and sufficient conditions. The construction below proves sufficiency. [1]

2. Construct a trail, then splice in closed trails

Choose an odd-degree vertex as the start if there are two; otherwise choose any vertex incident with an edge. Follow any unused incident edge, marking its identity as used, until the current vertex has no unused incident edge. The choices need not avoid bridges. This first maximal trail ends at the other odd vertex, or at its start when all degrees are even.

If unused edges remain, find an occurrence of a vertex x on the current trail that is incident with an unused edge. Starting at x , follow unused edges until stuck. This new trail is closed:

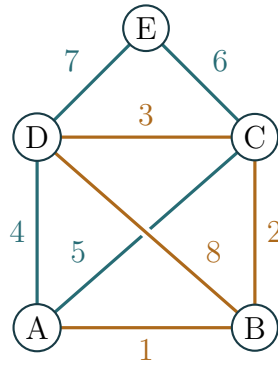
it returns to x . Insert it at that occurrence of x in the existing trail, then continue with the old suffix. Repeat until every edge has been included.

This is often called cycle splicing. More precisely, the inserted piece may be a closed trail that revisits vertices, rather than a simple cycle. Because it uses only previously unused edges, the enlarged result remains a trail. No edge is doubled.

An equivalent reduction handles two odd vertices s and t by adding one uniquely marked auxiliary edge between them. All degrees are now even. Compute a circuit, rotate its cyclic edge sequence so that the auxiliary edge is at the cut, and remove that edge. The remainder is an open Euler trail between s and t . This artificial edge is only a computational device; it is not drawn in the original graph.

$$P = P_1 \cdot x \cdot P_2, \quad C = x \cdots x, \quad P' = P_1 \cdot C \cdot P_2.$$

3. Example: the Haus vom Nikolaus



The eight edge numbers give the final trail. The crossing is not a vertex.

Label the bottom-left corner A, bottom-right B, top-right C, top-left D, and roof tip E. The eight undirected edges are AB, BC, CD, DA, AC, BD, DE, and EC. The crossing of the two diagonals is not an additional vertex. Hence the degrees are 3, 3, 4, 4, and 2 at A, B, C, D, and E respectively. The drawing therefore has an open Euler trail from A to B, or in reverse.

One possible first maximal trail is A-B-C-D-B. It stops at B while four edges remain. At D, insert the unused closed trail D-A-C-E-D. The resulting Euler trail is A-B-C-D-A-C-E-D-B. It has eight edge steps and nine vertex occurrences; the four repeated vertex visits are entirely legitimate.

The initial traversal order need not equal the final drawing order: inserting the closed trail places it before the last step D-B. A program may construct the complete sequence first and animate that sequence afterwards.

4. A short correctness proof

Necessity. At every internal occurrence of a vertex, an arrival is paired with a departure. Consequently the number of used incidences is even there. In a closed trail this also holds at the starting vertex. An open trail leaves one unpaired incidence at each of its two distinct endpoints. An Euler trail can therefore have zero or two odd-degree vertices, with the stated endpoints.

No premature stranding. First suppose all degrees are even. At any current endpoint other than the starting vertex, the trail has used an odd number of incidences. An even total degree then leaves at least one unused incidence, so the trail can continue. A maximal trail can stop only at its start. With two odd vertices s and t , start at s . The same parity argument rules out stopping at an even-degree vertex; it also rules out stopping back at s , where a closed return has used an even number of incidences but the total degree is odd. Therefore the first maximal trail stops at t .

Residual parity and splicing. Removing the edges of this first trail leaves even degree at every vertex: internal visits remove incidences in pairs, and the two odd endpoints each lose an odd number. Every later maximal trail in the unused edges is consequently closed. Splicing it into the current trail preserves adjacency and the endpoints, and cannot repeat an edge because the two edge sets are disjoint.

Progress and completeness. If an unused edge remains, connectivity guarantees a vertex of the current trail incident with an unused edge. If the unused edge already meets a visited vertex, the claim is immediate. Otherwise take a path from the current trail towards it; the first edge leaving the visited vertices is unused and provides an attachment point. Thus another splice is possible. Every splice uses at least one new edge, and there are only finitely many edges. The process terminates with a trail using every edge exactly once. This proves sufficiency as well as correctness.

5. An explicit stack implements the splicing

An explicit stack stores vertex occurrences, together with the edge ID by which each occurrence was reached. When the top vertex has an unused edge, mark that edge and push its other endpoint. When it has none, pop the occurrence into a reverse output list. The final route is the reversal of that list. Used flags are shared by the two adjacency entries of each undirected edge, including a loop's two entries.

Popping records a completed suffix; it does not traverse an edge again. If a previous occurrence still has unused incident edges, the stack explores that remaining closed excursion before finalising the occurrence. Reversing the output therefore inserts the excursion at its shared vertex: exactly the splicing operation proved correct above. Starting at an odd vertex handles the open case directly; no auxiliary edge is required.

```

Choose a valid start  $s$ ; mark every edge unused.
 $S := [(s, \text{no incoming edge})]$ ;  $R := \text{empty list}$ .
while  $S$  is not empty:
     $v := \text{the vertex at the top of } S$ 
    discard already-used adjacency entries at  $v$ 
    if an unused edge  $e = \{v, w\}$  is available:
        mark  $e$  used; push  $(w, e)$  onto  $S$ 
    else:
        append  $\text{pop}(S)$  to  $R$ 
reverse  $R$ 
read its vertices and incoming edge IDs as the trail
reject if the number of output edges is not  $|E|$ 

```

6. Running time, validation, and reproducibility

Store adjacency lists with edge IDs and remove entries from their ends, or advance a cursor per vertex. Each of the $2|E|$ adjacency entries is inspected and discarded at most once. Each edge causes one push, and there are $|E| + 1$ pops. Constructing the lists, checking degrees, reversing the output, and optional connectivity checking all fit within $O(|V| + |E|)$ time and $O(|V| + |E|)$ space. For arbitrary Python hashable labels, these bounds assume constant-time dictionary operations on average.

Naively scanning the full growing trail after each splice, or repeatedly copying long lists, can lose the linear bound. The explicit stack avoids these operations and Python's recursion-depth limit. The implementation in the appendix rejects invalid parity and starting vertices. Its final edge count detects disconnected edge-containing components; empty input returns an empty trail. It preserves parallel-edge IDs and counts loops correctly.

For an independent implementation, NetworkX provides `eulerian_path` and `eulerian_circuit`, with edge keys for multigraphs. Its documented convention treats a graph containing both edges and isolated vertices as non-Eulerian, whereas our edge-list convention ignores isolated vertices. Remove isolates before comparing the conventions. [2]

7. What this does - and does not - model

Hierholzer's algorithm solves an exact edge-traversal problem on a fixed graph; it does not by itself model how a biological spider builds a web. A physical spider may revisit silk, move between construction sites, or attach new threads. Those movements must be distinguished from the final Euler trail and from stack bookkeeping.

If a connected web graph has more than two odd-degree vertices, a one-stroke drawing of all its edges without repetition is impossible. Repeated edges can be introduced as a separate graph-augmentation step; then Hierholzer runs on the augmented multigraph. Choosing economical repetitions is a route-inspection problem, not a necessary part of Hierholzer's algorithm. The Haus vom Nikolaus needs no such augmentation.

8. Historical note

The 1873 article is headed with Carl Hierholzer's name and identifies Chr. Wiener as its communicator. Wiener's introductory footnote states that Hierholzer had presented the investigation to friends and that, in the absence of a written record, Wiener reconstructed it from memory with Lüroth's assistance. It is therefore more accurate to describe a posthumously reconstructed account of Hierholzer's argument than to claim that he personally wrote the published paper. [1]

Deutsch

Ein Eulerweg durchläuft jede Kante eines Graphen genau einmal. Hierholzers Algorithmus konstruiert einen solchen Weg, indem er kleinere Kantenzüge an gemeinsamen Knoten zusammenfügt. Das Haus vom Nikolaus veranschaulicht die Konstruktion; ein Paritätsargument liefert einen kurzen Korrektheitsbeweis. Mit einem expliziten Stapel entsteht eine Implementierung mit linearer Laufzeit, deren fertiger Zeichenweg keine Kante wiederholt.

1. Das Problem und seine genauen Voraussetzungen

Sei $G = (V, E)$ ein endlicher ungerichteter Graph mit mindestens einer Kante. Parallele Kanten und Schleifen sind erlaubt. Parallele Kanten haben verschiedene Identitäten, auch wenn ihre Endpunkte übereinstimmen; eine Schleife zählt beim Grad ihres Knotens doppelt. Alle Knoten, die mit einer Kante inzident sind, sollen zu einer einzigen Zusammenhangskomponente gehören. Isolierte Knoten spielen beim Durchlaufen der Kanten keine Rolle und werden hier ignoriert.

Ein Kantenzug ist eine Folge aufeinanderfolgender Kanten mit passenden Endknoten. Hier betrachten wir Kantenzüge ohne wiederholte Kante; Knoten dürfen mehrfach vorkommen. Ein Eulerweg benutzt jede Kante genau einmal. Stimmen seine Endpunkte überein, heißt er Eulerkreis, andernfalls offener Eulerweg. Beim Zeichnen in einem Zug geht es also um die Kanten, nicht darum, jeden Knoten nur einmal zu besuchen.

Ein Eulerkreis existiert genau dann, wenn alle Knotengrade gerade sind. Ein offener Eulerweg existiert genau dann, wenn genau zwei Knoten ungeraden Grad haben; diese müssen seine Endpunkte sein. Zusammen mit dem Zusammenhang sind diese Bedingungen notwendig und hinreichend. Die folgende Konstruktion beweist die Hinrichtung von den Bedingungen zur Existenz. [1]

2. Einen Kantenzug bilden und geschlossene Züge einfügen

Gibt es zwei Knoten ungeraden Grades, wähle einen davon als Start. Andernfalls beginne an einem beliebigen Knoten mit mindestens einer Kante. Folge jeweils irgendeiner noch unbenutzten Kante, markiere ihre Identität als benutzt und fahre fort, bis am aktuellen Knoten keine unbenutzte Kante mehr verfügbar ist. Brücken müssen bei der Auswahl nicht vermieden werden. Dieser erste maximale Kantenzug endet am anderen ungeraden Knoten beziehungsweise am Start, falls alle Grade gerade sind.

Sind noch Kanten unbenutzt, suche ein Vorkommen eines Knotens x im bisherigen Kantenzug, an dem eine solche Kante anliegt. Beginne dort einen neuen Zug aus unbenutzten Kanten und gehe weiter, bis keine Fortsetzung möglich ist. Dieser Zug ist geschlossen: Er endet wieder bei x . Füge ihn an dem gewählten Vorkommen von x in den bisherigen Zug ein und setze danach dessen alten Rest fort. Wiederhole dies, bis alle Kanten enthalten sind.

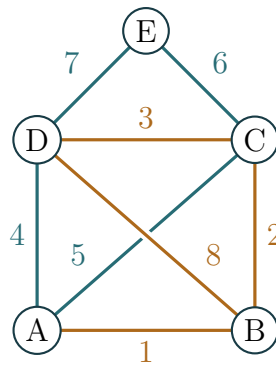
Oft spricht man vom Einfügen von Zyklen. Genauer kann das eingefügte Stück ein geschlossener Kantenzug mit mehrfach besuchten Knoten sein; es muss kein einfacher Zyklus sein. Da es ausschließlich vorher unbenutzte Kanten enthält, bleibt der zusammengesetzte Zug kantenwiederholungsfrei. Es wird keine Kante verdoppelt.

Eine gleichwertige Reduktion ergänzt bei zwei ungeraden Knoten s und t eine eindeutig markierte Hilfskante zwischen ihnen. Nun sind alle Grade gerade. Berechne einen Eulerkreis,

drehe seine zyklische Kantenfolge so, dass an der Hilfskante aufgeschnitten wird, und entferne diese Kante. Der Rest ist ein offener Eulerweg zwischen s und t . Die Hilfskante dient nur der Berechnung; sie wird im ursprünglichen Graphen nicht gezeichnet.

$$P = P_1 \cdot x \cdot P_2, \quad C = x \cdots x, \quad P' = P_1 \cdot C \cdot P_2.$$

3. Beispiel: das Haus vom Nikolaus



Die acht Kantennummern geben den fertigen Eulerzug an. Die Kreuzung ist kein Knoten.

Bezeichne die untere linke Ecke mit A, die untere rechte mit B, die obere rechte mit C, die obere linke mit D und die Dachspitze mit E. Die acht ungerichteten Kanten sind AB, BC, CD, DA, AC, BD, DE und EC. Der Kreuzungspunkt der beiden Diagonalen ist kein zusätzlicher Knoten. Die Grade von A, B, C, D und E sind daher 3, 3, 4, 4 und 2. Somit besitzt die Zeichnung einen offenen Eulerweg von A nach B oder umgekehrt.

Ein möglicher erster maximaler Kantenzug ist A-B-C-D-B. Er endet bei B, obwohl noch vier Kanten fehlen. Bei D wird der unbenutzte geschlossene Zug D-A-C-E-D eingefügt. Daraus entsteht der Eulerweg A-B-C-D-A-C-E-D-B. Er besteht aus acht Kantenschritten und neun Knotenvorkommen; die vier zusätzlichen Besuche bereits bekannter Knoten sind erlaubt.

Die anfängliche Erkundungsreihenfolge muss nicht der endgültigen Zeichenreihenfolge entsprechen: Durch das Einfügen liegt der geschlossene Zug vor dem letzten Schritt D-B. Ein Programm kann zunächst die vollständige Folge berechnen und erst danach diese Folge animieren.

4. Ein kurzer Korrektheitsbeweis

Notwendigkeit. Bei jedem inneren Besuch eines Knotens bilden Ankunft und Abgang ein Paar. Die Zahl der dabei benutzten Inzidenzen ist deshalb gerade. Bei einem geschlossenen Zug gilt dies auch am Startknoten. Bei einem offenen Zug bleibt an jedem seiner beiden verschiedenen Endpunkte eine ungepaarte Inzidenz übrig. Ein Eulerweg kann daher nur null oder zwei Knoten ungeraden Grades haben, mit den angegebenen Endpunkten.

Kein vorzeitiges Steckenbleiben. Zunächst seien alle Grade gerade. An einem aktuellen Endknoten, der nicht der Startknoten ist, hat der bisherige Zug eine ungerade Zahl von Inzidenzen benutzt. Ein gerader Gesamtgrad lässt dann mindestens eine unbenutzte Inzidenz übrig; der Zug kann fortgesetzt werden. Ein maximaler Zug endet daher am Start. Gibt es genau zwei ungerade Knoten s und t , beginne bei s . Dasselbe Paritätsargument schließt ein

Ende an einem geraden Knoten aus. Auch bei s ist kein Ende möglich: Bei einer Rückkehr wurden dort gerade viele Inzidenzen benutzt, sein Gesamtgrad ist aber ungerade. Der erste maximale Zug endet folglich bei t .

Restparität und Einfügen. Nach Entfernen der Kanten des ersten Zuges sind alle verbleibenden Knotengrade gerade: Innere Besuche entfernen Inzidenzen paarweise, und die beiden ungeraden Endpunkte verlieren jeweils ungerade viele. Jeder spätere maximale Zug aus unbenutzten Kanten ist somit geschlossen. Sein Einfügen erhält den Zusammenhang der Kantenfolge und ihre Endpunkte. Kantenwiederholungen entstehen nicht, weil die beiden Kantenmengen disjunkt sind.

Fortschritt und Vollständigkeit. Bleibt eine Kante unbenutzt, so garantiert der Zusammenhang einen Knoten des bisherigen Zuges mit einer unbenutzten inzidenten Kante. Liegt die unbenutzte Kante bereits an einem besuchten Knoten, ist dies unmittelbar klar. Andernfalls betrachte einen Weg vom bisherigen Zug zu dieser Kante: Die erste Kante, die die besuchten Knoten verlässt, ist unbenutzt und liefert eine Anschlussstelle. Ein weiteres Einfügen ist daher möglich. Jeder solche Schritt benutzt mindestens eine neue Kante; insgesamt gibt es nur endlich viele. Das Verfahren endet somit mit einem Zug, der jede Kante genau einmal enthält. Das beweist sowohl die hinreichende Bedingung als auch die Korrektheit.

5. Ein expliziter Stapel übernimmt das Einfügen

Ein expliziter Stapel speichert Knotenvorkommen zusammen mit der Kennung der jeweils benutzten Eingangskante. Hat der oberste Knoten eine unbenutzte Kante, wird diese markiert und ihr anderer Endknoten auf den Stapel gelegt. Andernfalls wird das oberste Vorkommen entfernt und an eine rückwärts aufgebaute Ausgabeliste angehängt. Die endgültige Route ist die Umkehrung dieser Liste. Die beiden Adjazenzeinträge einer ungerichteten Kante teilen sich dieselbe Benutzt-Markierung; das gilt auch für die beiden Einträge einer Schleife.

Das Entfernen vom Stapel hält einen fertigen Restabschnitt fest; es bedeutet kein erneutes Durchlaufen einer Kante. Sind bei einem früheren Vorkommen noch Kanten unbenutzt, erkundet der Stapel den dort verbleibenden geschlossenen Ausflug, bevor dieses Vorkommen endgültig ausgegeben wird. Nach Umkehrung der Ausgabe steht der Ausflug genau an seinem gemeinsamen Knoten: Das ist die oben bewiesene Einfügeoperation. Mit einem ungeraden Startknoten wird der offene Fall unmittelbar behandelt; eine Hilfskante ist nicht erforderlich.

```

Wähle einen zulässigen Start  $s$ ; markiere alle Kanten unbenutzt.
 $S := [(s, \text{keine Eingangskante})]$ ;  $R :=$  leere Liste.
solange  $S$  nicht leer ist:
     $v :=$  Knoten an der Spitze von  $S$ 
    entferne bereits benutzte Adjazenzeinträge bei  $v$ 
    falls eine unbenutzte Kante  $e = \{v, w\}$  verfügbar ist:
        markiere  $e$  benutzt; lege  $(w, e)$  auf  $S$ 
    sonst:
        entferne die Spitze von  $S$  und hänge sie an  $R$  an
kehre  $R$  um
lies Knoten und Eingangskanten-Kennungen als Kantenzug ab
weise zurück, falls die Zahl der Ausgabekanten nicht  $|E|$  ist

```

6. Laufzeit, Prüfung und Reproduzierbarkeit

Speichere Adjazenzlisten mit Kantenkennungen und entferne Einträge an ihren Enden, oder verwende je Knoten einen fortlaufenden Index. Jeder der $2|E|$ Adjazenzeinträge wird höchstens einmal geprüft und entfernt. Jede Kante verursacht ein Auflegen; insgesamt gibt es $|E| + 1$ Entnahmen vom Stapel. Aufbau der Listen, Gradprüfung, Umkehrung der Ausgabe und eine optionale Zusammenhangsprüfung benötigen insgesamt $O(|V| + |E|)$ Zeit und $O(|V| + |E|)$ Speicher. Für beliebige hashbare Python-Knotenbezeichnungen setzt diese Aussage im Mittel konstante Kosten der Wörterbuchoperationen voraus.

Wer nach jedem Einfügen den gesamten wachsenden Zug durchsucht oder lange Listen kopiert, kann die lineare Laufzeit verlieren. Der explizite Stapel vermeidet dies und die Rekursionstiefengrenze von Python. Die Implementierung im Anhang weist unzulässige Paritäten und Startknoten zurück. Ihre abschließende Kantenzählung erkennt getrennte Komponenten mit Kanten; eine leere Eingabe ergibt einen leeren Zug. Parallele Kanten behalten ihre Kennungen, und Schleifen werden korrekt gezählt.

Als unabhängige Implementierung bietet NetworkX `eulerian_path` und `eulerian_circuit` mit Kantenschlüsseln für Multigraphen. Nach der dokumentierten NetworkX-Konvention verhindern isolierte Knoten einen Eulerweg, wenn sie den Gesamtgraphen unzusammenhängend machen. Unsere Kantenlisten-Konvention ignoriert sie. Zum Vergleich beider Konventionen sind isolierte Knoten vorher zu entfernen. [2]

7. Was hier modelliert wird - und was nicht

Hierholzers Algorithmus löst ein genau bestimmtes Kantendurchlaufproblem auf einem feststehenden Graphen; er modelliert für sich genommen nicht den biologischen Netzbau einer Spinne. Eine reale Spinne kann vorhandene Fäden mehrfach benutzen, zwischen Baustellen wechseln oder neue Fäden anbringen. Solche Bewegungen sind vom fertigen Eulerweg und von der Stapelverwaltung zu unterscheiden.

Hat ein zusammenhängender Netzgraph mehr als zwei ungerade Knoten, ist eine Zeichnung aller Kanten in einem Zug ohne Wiederholung unmöglich. Wiederholte Kanten können in einem gesonderten Erweiterungsschritt hinzugefügt werden; anschließend läuft Hierholzer auf dem erweiterten Multigraphen. Eine wirtschaftliche Wahl dieser Wiederholungen gehört zur Routenplanung, nicht zu den notwendigen Schritten von Hierholzers Algorithmus. Beim Haus vom Nikolaus ist eine solche Erweiterung nicht nötig.

8. Historische Anmerkung

Der Artikel von 1873 trägt Carl Hierholzers Namen und nennt Chr. Wiener als denjenigen, der die Untersuchung mitteilte. In der einleitenden Fußnote berichtet Wiener, Hierholzer habe seine Überlegungen Freunden vorgetragen. Da keine schriftliche Aufzeichnung vorlag, habe Wiener die Darstellung mit Unterstützung von Lüröth aus dem Gedächtnis rekonstruiert. Es ist deshalb genauer, von einer nach Hierholzers Tod rekonstruierten Darstellung seines Arguments zu sprechen, als ihm das persönliche Verfassen des veröffentlichten Artikels zuzuschreiben. [1]

Python implementation / Python-Implementierung

This complete Python 3 program needs only the standard library. Edge IDs are positions in the input list; the result records traversal direction and edge identity.

Dieses vollständige Python-3-Programm benötigt nur die Standardbibliothek. Kantennummern entsprechen den Positionen in der Eingabeliste; das Ergebnis hält Durchlaufrichtung und Kantenidentität fest.

```

"""Hierholzer for undirected edge lists.

Parallel edges and loops are allowed.
"""

_AUTO = object()

def euler_trail(edges, start=_AUTO):
    """Return (u, v, edge_id) triples in traversal order.

    Edge IDs are input positions. Isolated vertices are omitted.
    Empty input returns []. Vertex labels must be hashable.
    """
    edges = list(edges)
    if not edges:
        return []
    adj = {}
    for eid, (u, v) in enumerate(edges):
        adj.setdefault(u, []).append((v, eid))
        adj.setdefault(v, []).append((u, eid))
    odd = [u for u, row in adj.items() if len(row) % 2]
    if len(odd) not in (0, 2):
        raise ValueError("Expected zero or two odd-degree vertices")
    if start is _AUTO:
        start = odd[0] if odd else next(iter(adj))
    if start not in adj or (odd and start not in odd):
        raise ValueError("Invalid starting vertex")
    for row in adj.values():
        row.reverse() # Pop from the end in input order.
    used = [False] * len(edges)
    stack = [(start, None)]
    reverse_vertices, reverse_ids = [], []
    while stack:
        u = stack[-1][0]
        while adj[u] and used[adj[u][-1][1]]:
            adj[u].pop()
        if adj[u]:
            v, eid = adj[u].pop()
            used[eid] = True

```

```

        stack.append((v, eid))
    else:
        v, eid = stack.pop()
        reverse_vertices.append(v)
        if eid is not None:
            reverse_ids.append(eid)
    if len(reverse_ids) != len(edges):
        raise ValueError("The non-isolated part is disconnected")
    vertices = reverse_vertices[::-1]
    ids = reverse_ids[::-1]
    return list(zip(vertices, vertices[1:], ids))

HOUSE = [("A", "B"), ("B", "C"), ("C", "D"), ("D", "B"),
          ("D", "A"), ("A", "C"), ("C", "E"), ("E", "D")]

if __name__ == "__main__":
    tour = euler_trail(HOUSE, "A")
    print("-".join([tour[0][0]] + [v for _, v, _ in tour]))

```

Expected output / Erwartete Ausgabe: A-B-C-D-A-C-E-D-B.

References / Literatur

- [1] C. Hierholzer (communicated by / mitgeteilt von Chr. Wiener). *Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren*. Mathematische Annalen, 6, 30–32, 1873. doi:10.1007/BF01442866. Original scan / Originalscan: <https://zenodo.org/records/1447429>.
- [2] NetworkX developers. *Eulerian algorithms: source code and documentation*. https://networkx.org/documentation/stable/_modules/networkx/algorithms/euler.html. Accessed / Abgerufen: 17 October / Oktober 2026.