Graphs and Eulerian tours

11.11.2021 (last taught)

David Steurer

8.12.2020 (edited)

Contents

In this lecture, we introduce the notion of a graph, which is of central importance for many branches of computer science as well as other scientific disciplines. Many questions benefit from being viewed through the lens of graphs. As a first example, we discuss a famous mathematical and algorithmic problem, which we call `Eulerian-Tour`. We use this example to illustrate and motivate some basic concepts in the context of graphs.

1 Seven bridges of Königsberg

Leonhard Euler (mathematical giant, born 1707 in Basel) solved the following mathematical puzzle (fig. 1), called "Seven Bridges of Königsberg":

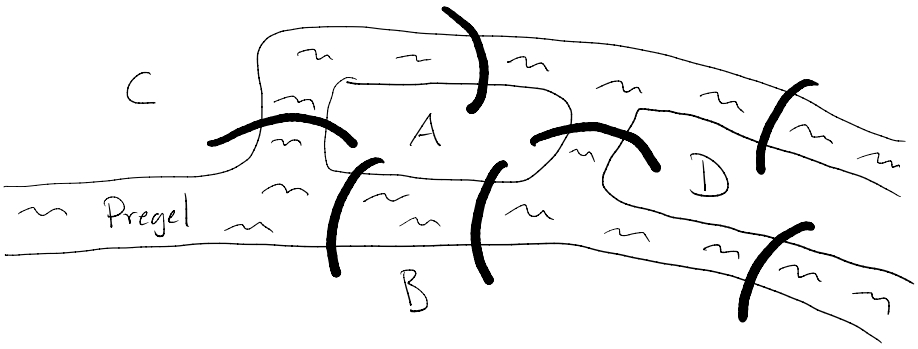
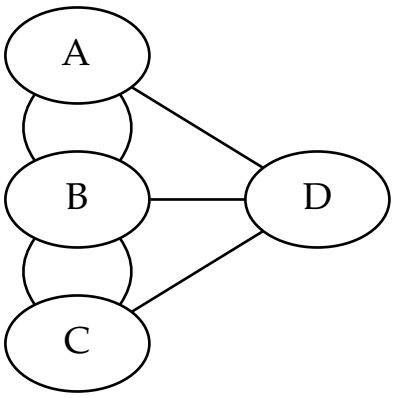


Figure 1: Seven Bridges of Königsberg

> Devise a walk through the city of Königsberg that crosses each of its seven bridges over the Pregel River exactly once.

Euler proved that such a walk does not exist. As part of his proof, he proposed the following abstract representation, which we call *graph* nowadays, for the city parts and the bridges. This representation captures all aspects of the city and bridges relevant for the puzzle.



Here, the four parts of the city separated by the river are represented by four *vertices* labeled A, B, C, and D. The seven bridges over the river are represented by *edges* that connect pairs of vertices.

The puzzle asks whether it is possible to walk along the edges of the above graph in such a way that each edge is traversed exactly once. We call such a walk *Eulerian*.

To answer the question whether a Eulerian walk exists, it turns out that vertex degrees play an important role, where the *degree* of a vertex is defined to be the number of edges that touch the vertex.

The following table contains the degrees of the vertices in the graph representing the bridges of Königsberg.

A	B	C	D
3	5	3	3

A vertex with degree 0 is called *isolated*.

Based on these vertex degrees, the following claim demontrates that the above graph doesn't have an Eulerian walk.

Claim: If there exists a Eulerian walk, then all but at most two vertex degrees must be even.

Proof: Suppose a Eulerian walk W exists. Let v be any vertex not at the start or end of W. Since v is not at the start or end of W, it experiences the same number

of arrivals as departures in W. Since W is Eulerian, the sum of the number of arrivals and the number of departures at v has to equal the degree of v. Hence, the degree of v is equal to twice the number of arrivals, which means that the degree is even. □

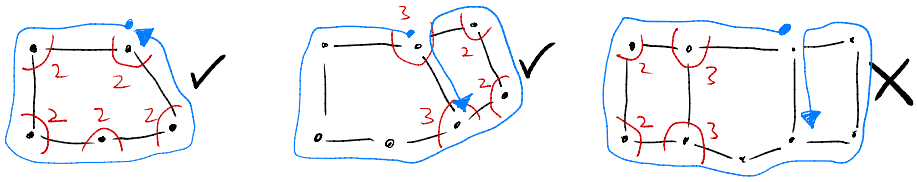
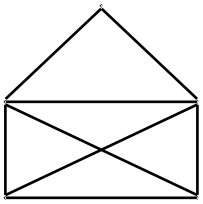


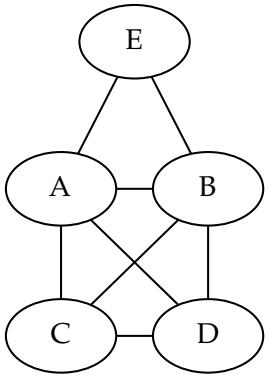
Figure 2: three example graphs with zero, two, and four odd-degree vertices

2 House of Santa Claus

Eulerian walks are connected also to an old drawing and rhyming puzzle for children (which appears to be known mostly in German-speaking regions). The puzzle is to draw the following figure in eight strokes—one for each syllable of the rhyme "Das ist das Haus vom Ni-ko-laus"—without lifting the pen.



Observation: A sequence of strokes to draw this figure without lifting exactly corresponds to a Eulerian walk in the following graph (obtained by placing a vertex at each corner of the drawing).



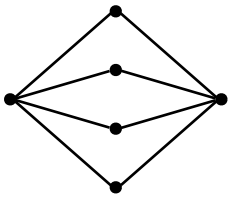
This graph indeed has a Eulerian walk: C,A,E,B,D,A,B,C,D.

3 Hamiltonian path

A Hamiltoninan path in a graph is defined to be a walk that visits every vertex exactly once. This definition is syntactically similar to the definition of Eulerian walks. Only the role of edges and vertices is interchanged.

Despite the similar definition, it turns out that in general it is much more difficult to reason about Hamiltonian paths than about Eulerian walks.

The following graph is an example of a graph that has a Eulerian walk but not a Hamiltonian walk.



4 Algorithms for Hamiltonian paths and Eulerian walks

Suppose we are given a graph and the goal is to compute a Eulerian walk or a Hamiltonian path if it exists.

A naive algorithm is *brute-force search*: we consider all possible orderings of edges or vertices. However, the running time of this approach is at least exponential. Indeed, for a graph with n vertices and m edges, there are $n!$ possible orderings of vertices and $m!$ possible orderings of edges.

For Eulerian walks, as we will see in this lecture, it is possible to avoid brute-force search and compute a Eulerian walk in time $O(n + m)$ if it exists.

In contrast, for Hamiltonian paths, brute-force search appears to be unavoidable. The famous $P \neq NP$ conjecture turns out to be equivalent to conjecturing that there is no polynomial-time algorithm for computing Hamiltonian paths.

5 Graphs

Before proceeding, we step back to discuss the broader significance of graphs. We also give a mathematical definition of graphs and introduce some terminology that is commonly used in the context of graphs.

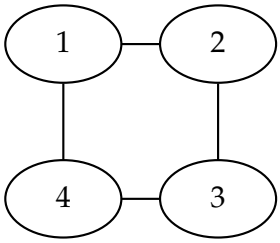
Networks are ubiquitous in many different human endeavors:

- *computer networks* (like the internet): computing devices are connected to each other by data links
- *social networks:* people are connected to each other by their social relationships
- *transportation networks:* cities are connected to each other by streets and train tracks
- *neural networks* (both artifical and natural ones): neurons are connected to each other at synapses

Graphs are mathematical models for networks that highlight some common structure shared by all networks.

Definition: A *graph* $G = (V, E)$ consists of a finite vertex set V and finite edge set E such that each edge $e \in E$ is an unordered pair $e = \{u, v\}$ of distinct vertices $u, v \in V$. Typically, we require that V is not the empty set.

It is often convenient to choose V to be a set of consecutive natural number.



In the above example, the vertex set is $\{1, 2, 3, 4\}$ and the edge set consists of the following edges,

$$\{1, 2\}, \{2, 3\}, \{3, 4\}, \{1, 4\}\,.$$

Definition: If a graph $G = (V, E)$ contains an edge $e = \{u, v\} \in E$, we call the vertices u, v *adjacent* in G and we say e is *incident* to u and v. The degree of a vertex u, denote $\deg(u)$, is the number of edges that u is incident to in G.

Instead of writing $e = \{u, v\}$, we often use the shorthand $e = uv$.

Handshake lemma: For every graph $G = (V, E)$,

$$\sum_{v \in V} \deg(v) = 2|E|\,.$$

Proof: Suppose every vertex distributes one coin to each of its incident edges. Then, every edge receives two coin in total, one coin from each of its endpoints. Thus, the sum of the vertex degrees has to equal twice the number of edges. □

Definition: A *walk* of *length* ℓ is a sequence of $\ell + 1$ vertices $v_0, \ldots, v_\ell$ such that consecutive ones are adjacent, i.e.,

To this walk, we also associate the sequence of ℓ edges $\{v_0, v_1\}, \ldots, \{v_{\ell-1}, v_\ell\}$. Edges and vertices are allowed to be repeated multiple times in a walk. *Note:* The usual convention is that the length of a walk refers to its number of edges (as opposed to its number of vertices).

For a walk $W = (v_0, \ldots, v_\ell)$, we define the degree $\deg_W(u)$ of a vertex u to be the number of edges in W incident to u (with repetitions counted),

$$\deg_W(u) = |\{i \in \{0, \ldots, \ell - 1\} \mid u \in \{v_i, v_{i+1}\}\}| .$$

Note: If W contains the same edge multiple times, then $\deg_W(u)$ could be larger than $\deg(u)$.

A walk is *closed*, if starts and ends with the same vertex $v_0 = v_\ell$. A *path* is a walk without repeated vertices.

Claim: A walk $W = (v_0, \ldots, v_\ell)$ is closed if and only if $\deg_W(v_\ell)$ is even.

Proof: Each occurrence of v_ℓ in the inner part $v_1, \ldots, v_{\ell-1}$ of W increases $\deg_W(v_\ell)$ by 2 and thus doesn't affect the parity of $\deg_W(v_\ell)$. Hence, $\deg_W(v_\ell)$ is even if and only if v_ℓ occurs twice at the two ends of W, which means that $v_0 = v_\ell$. □

We say that vertex u *reaches* vertex v if there exists a walk starting in u and ending in v. This relation satisfies all properties of an equivalence relation: symmetry, reflexivity, and transitivity.

The *connected component* of a vertex u is the set of all vertices that it can reach. In other words, the connected component of u is the equivalence class of the reachability relation that contains u. A graph is *connected* if every vertex u reaches every other vertex v. In other words, a graph is connected if it has only one connected component.

A *Eulerian tour* is a closed walk that visits every edge exactly once.

6 Characterization of Eulerian tours

The following theorem shows that the only possible obstruction for a Eulerian tour is a vertex of odd degree.

Theorem: A connected graph has a Eulerian tour if and only if all vertex degrees are even.[1]

One direction of the statement of the theorem is easy to prove. In a closed walk W, all degrees $\deg_W(u)$ in W are even. If W is a Eulerian, the degrees in W are the same as the degrees in the graph, i.e., $\deg_W(u) = \deg(u)$ for all $u \in V$.

To prove the other direction, we exhibit an algorithm that outputs a Eulerian tour in a connected graph whenever all vertex degrees are even.

Note: A similar theorem holds for Eulerian (non-closed) walks in connected graphs. We just need to account for up to two vertices with odd degrees. We can reduce questions about Eulerian walks to questions about Eulerian tours

[1] A graph with a Eulerian tour is not necessarily connected. However, all edges need to be in the same connected component. In other words, the graph needs to be connected after removing isolated vertices.

by adding a new vertex adjacent to the two odd-degree vertices if needed (see fig. 3).

Figure 3: Reducing Eulerian walks to Eulerian tours: after adding the red vertex and edges all degrees are even

7 Iterative approach

Instead of aiming at directly computing a Eulerian tour, we first consider simpler but related tasks.

For example, we could start by computing any closed walk (of length larger than 0). Then, we could try to compute a list of closed walks $Z_1, \ldots, Z_k$ such that each edge appears exactly once in the walks.

Indeed, if we can find such a list of closed walks for a connected graph, we can obtain a Eulerian tour by merging the closed walks as illustrated in the example fig. 4. We can always merge two closed walks that share a vertex to a single closed walk that uses the same edges. In a connected graph, we can keep merging in this way until only one closed walk remains that uses every edge exactly once.

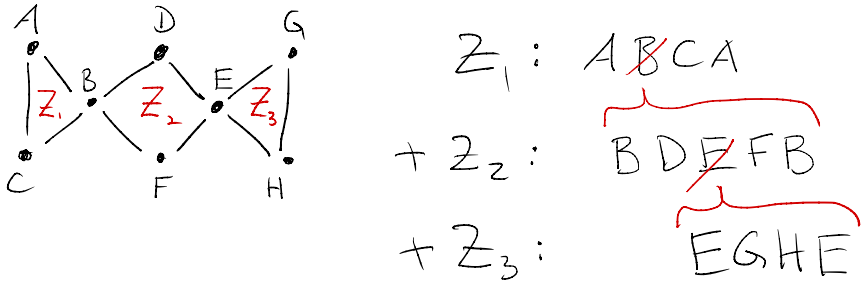


Figure 4: merging closed walks to a Eulerian tour

To summarize, we consider the following approach for computing a Eulerian tour:

- iteratively compute closed walks $Z_1, \ldots, Z_k$ until every edge ised exactly once
- merge the closed walks to a Eulerian tour

In the following, we show how to carry out the first step for graphs without odd degree vertices.

8 Walking in graphs

We consider the following recursive procedure for finding a maximal walk[2] starting in a vertex u without repeating edges.

`Walk`(u):

- If there exists an unmarked edge $uv \in E$ incident to u:
  - Mark the edge uv.
  - Run `Walk`(v).

We take note of a few properties of this procedure:

1. The procedure `Walk`(u) marks a walk W starting in u.
2. Every edge gets marked at most once.
3. The walk W ends in a vertex v such that all edges incident to v are marked.

The second property holds because we never undo markings and we check that an edge is unmarked before marking it. The reason the third property holds is that in the last call of `Walk`(v) we didn't mark any edge incident to v, which means that all edges incident to v were marked at this point.

In the previous example (fig. 4), suppose we start with no edges marked and run the `Walk` procedure first on B and then on E (without undoing the markings made by `Walk`(B)). Then, we mark the following walks, assuming we process the edges in alphabetical order:

procedure call	marked walk
`Walk`(B):	B A C B D E F B
`Walk`(E):	E G H E

In order to analyze the `Walk` procedure we propose the following property as an invariant:

> `ALL-EVEN`: every vertex is incident to an even number of unmarked edges

If all edges are unmarked and all vertex degrees are even, then `ALL-EVEN` holds.

Claim: If `ALL-EVEN` holds before running `Walk`(u), then it also holds after `Walk`(u) has finished. Furthermore, the walk W marked by `Walk`(u) is closed.

[2]Here, maximal means that the walk cannot be extended further. We don't require the walk to have maximum length.

Proof: It is enough to prove that W is closed. (If W is closed, then $\deg_W(w)$ is even for all vertices $w \in V$, which means that `ALL-EVEN` also holds after `Walk`(u).)

Suppose W ends in vertex v. In order to prove that W is closed, we need to show that $\deg_W(v)$ is even. (Recall the claim we proved about closed walks in sec. 5).

We know that before running `Walk`(u), vertex v had an even number of unmarked edges (because we assume `ALL-EVEN` held at that time). After running `Walk`(u), vertex v has no unmarked edges (the third property of the `Walk` procedure we noted). Hence, $\deg_W(v)$, the number of edges incident to v marked by `Walk`(u), is even. □

The claim implies that we can partition the edges of any graph without odd degree vertices into a collection of closed walks by repeatingly running the `Walk` routing on vertices with unmarked edges until all edges are marked.

9 Fast algorithm

While the approach we have discussed so far allows us to compute a Eulerian tour in polynomial[3] time, we need additional ideas in order to achieve linear running time.

As a starting point, we consider the following question: After one call of the `Walk` routine has finished, which vertex should we choose as the start for the next call of the `Walk` routine?

The idea is to backtrack in the walks we have computed so far. For example, suppose in fig. 4 we run `Walk`(A) and mark the walk A B C A. Then, we backtrack in this walk two steps, A C B, and arrive at vertex B with unmarked edges incident to it. We run `Walk`(B) and mark the walk B D E F B. We backtrack two steps, B F B, and arrive at vertex E. Finally, we run `Walk`(E) and mark the walk E G H E. At this point all edges are marked.

To implement the idea of backtracking along the walks we have already found, we will use the following recursive procedure to build up a list Z, which will eventually contain a Eulerian tour:

`Euler-Walk`(u):

- For the edges $uv \in E$ incident to u:
  - If the edge uv is not yet marked, mark the edge uv and run `Euler-Walk`(v).
- Append u to the end of the list Z.

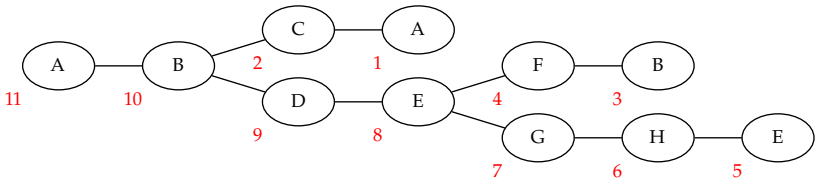
Using `Euler-Walk` as a subroutine, the following procedure find a Eulerian tour in a connected graph G:

[3]Here, we are not referring to any specific running times (like quadratic, cubic, and so on). As we will see later, the specific running times will depend on the kind of data structures we use to represent graphs.

`Euler(G)`:

- Initalize Z as an empty list and all vertices to be unmarked.
- Run `Euler-Walk`(u_0) for an arbitrary[4] vertex $u_0 \in V$.
- Output the final content of the list Z.

Recursion tree of the procedure `Euler-Walk` applied to vertex A for the example graph in fig. 4. The red numbers indicate the order in which vertices are appended to the list Z.



The content of the final list Z for this example:

1	2	3	4	5	6	7	8	9	10	11
A	C	B	F	E	H	G	E	D	B	A

10 Running time with adjacency matrix

In order to analyze the running time of `Euler`(G), we need to specify what data structure we use to represent the graph G.

Let $G = (V, E)$ be a graph with $n \geq 1$ vertices $V = \{1, \ldots, n\}$ and $m \geq 1$ edges.

Definition: The *adjacency matrix* $A = (A_{ij})_{i,j \in V}$ of G is defined as

$$A_{ij} = \begin{cases} 1 & \text{if } \{i, j\} \in E\,, \\ 0 & \text{otherwise.} \end{cases}$$

For example, the graph in fig. 5 has the following adjacency matrix (with the red entries of the matrix corresponding to the red edge in the graph):

[4] For a connected graph, we can choose the start vertex u_0 to be arbitrary. If we allow the underlying graph to have isolated vertices, we should choose u_0 to be a non-isolated vertex if one exists.

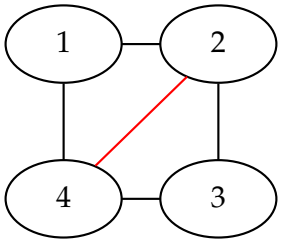


Figure 5: Example graph with adjacency matrix (1)

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \tag{1}$$

Running times of basic operations on adjacency matrices:

1. Given two vertices $u, v \in V$, we can test if $uv \in E$ in time $O(1)$.
2. Given a vertex $u \in V$, we can enumerate the neighbors of u in time $O(n)$.

Claim: `Euler`(G) can be implemented to have running time $O(n \cdot m)$ if G is represented by its adjacency matrix.

Proof: It suffices to bound the time for `Euler-Walk`(u_0). Each recursive call of `Euler-Walk` is associated with with an edge in G marked by the algorithm.

Hence, the total number of calls of `Euler-Walk` is at most[5] $m + 1$ (at most m recursive calls in addition to one initial call for u_0).

For any individual call of `Euler-Walk`, the number of elementary operations carried out while this call is active (i.e., not counting operations carried out inside nested recursive calls) is $O(n)$.

Hence, the total running time is $(m + 1) \cdot O(n) = O(n \cdot m)$. □

[5]Indeed, if the graph is connected and has a Eulerian tour, then the number of calls of `Euler-Walk`is equal to $m + 1$.

11 Running time with adjacency lists

The main weakness of representing graphs by their adjacency matrix is that enumerating the neighbors of a vertex takes time $\Theta(n)$, even when the number of neighbors is much small than n.

A better representation of a graph are its adjacency lists. For example, the graph in fig. 5 has the following adjacency list representation:

1	2	3	4
(2,4)	(3,1,4)	(4,2)	(1,3,2)

Let $G = (V, E)$ be a graph with $n \geq 1$ vertices $V = \{1, \ldots, n\}$ and $m \geq 1$ edges.

Definition: The adjacency list representation of a graph G is an n-dimensional array A such that $A[i]$ is a list of all neighbors of vertex i in G (in arbitrary order).

Running times of basic operations on adjacency list representations:

1. Given two vertices $u, v \in V$, we can test if $uv \in E$ in time $O(1 + \min\{\deg(u), \deg(v)\})$.
2. Given a vertex $u \in V$, we can enumerate the neighbors of v in time $O(1 + \deg(u))$.

Per our convention, $O(\cdot)$ hides only multiplicative constants. We make the additive constant 1 explicit here to account for the possibility that the degree of one of the vertices is 0. The (proof of) following claim illustrates that this additive constant 1 is useful to reason about running times.

Claim: The running time to enumerate all edges of G given its adjacency list representation is $O(n + m)$.

Proof: Using this fact that enumerating the neighbors of a single vertex u takes time $O(1 + \deg(u))$, the total running time is

$$O\left(\sum_{u \in V}\Big(1 + \deg(u)\Big)\right).$$

Using the handshake lemma, we can compute the sum

$$\sum_{u \in V}(1 + \deg(u)) = n + \sum_{u \in V} \deg(u) = n + 2m \,. \square$$

Note that without the additive 1 term, we might have been tempted to conclude a running time bound of $O(m)$. However, this bound is false for graphs that have much fewer edges than vertices (which happens if there are many isolated vertices).

Claim: `Euler`(G) can be implemented to have running time $O(n + m)$ for the adjacency list representation A of G.

Proof: In order to implement `Euler-Walk`(u), we enumerate the edges incident to u by traversing the adjacency list $A[u]$. For every unmarked edge uv, we encounter we start a recursive call for vertex u. Furthermore, after we enumerate an incident edge (be it marked or unmarked), we remove it from the adjacency list in time $O(1)$.

To bound the running time of `Euler-Walk`(u_0) for this implementation, we note that we can charge[6] every operation to an edge of G such that each edge gets charged with only $O(1)$ operations. The only exceptions are $O(1)$ operations in `Euler-Walk`(u_0) that are carried out even if u_0 has no incident edge.

Hence, the total running time of `Euler-Walk`(u_0) is $O(m + 1)$.

The initialization phase of `Euler`(G) takes time $O(n + m)$ in order to set up the data structure to support marking edges.

Thus, the total running itme of `Euler`(G) is $O(n + m)$.

12 Correctness

In this section, we will prove the correctness of the algorithm developed in the previous section.

Lemma: For a connected graph G without odd degree vertices, Euler(G) computes a Eulerian tour.

To prove this lemma, we imagine a tortoise and hare[7] using the recursion tree T of `Euler-Walk`(u_0) to walk in the graph G.

First, we note that each edge of G is represented by precisely one edge in T. Since `Euler-Walk` traverses an edge only if it hasen't been traversed before, it follows that no edge of G can appear more than once in G. Using the assumption that G is connected and the fact that `Euler-Walk` finishes only if all edges of a vertex have been traversed, it also follows that each edge of G appears at least once in G.

Next, we describe how the hare and tortoise move in the recursion tree T.

[6] More details: let's look at all `Euler-Walk` calls that we make during `Euler`(G). Every elementary operation of the algorithm happens for one of these calls (except for the operations during the initialization phase of `Euler`(G). Let m_i be the number of edges we enumerate during the i-th call of `Euler-Walk`. The number of operations for the i-th call of `Euler-Walk` is $O(1 + m_i)$. Each edge appears twice in the adjacency list representation of G. Since we delete an edge after we enumerate it in an adjacency list, we have $\sum_i m_i \leq 2m$. The number of `Euler-Walk` calls is at most $m + 1$. Hence, $\sum_i (1 + m_i) \leq 1 + 3m$. Hence, the total running time for the `Euler-Walk` calls is $O(m + 1)$.

[7] See Aesop's Fable "The Tortoise and the Hare". For our proof, the two protagonists give us a non-recursive way to think about the behavior of the `Euler-Walk` procedure.

The hare walks in T from the root to the first[8] leaf, from the first leaf to the second leaf, and so on. Finally, it walks from the last leaf back to the root. In this way, the hare traverses every edge of T precisely twice, once in the direction toward the root and once in the direction away from the root. The hare's movement follows the recursive calls of the `Euler-Walk` procedure. Moving down the tree (away from the root) corresponds to starting a new recursive call. Moving up the tree (toward the root) corresponds to finishing a recursive call.

The tortoise is waiting at the first leaf of T. When the hare arrives there, the tortoise walks toward the root together with the hare until they encounter an ancestor of the second leaf for the first time. At this point, the tortoise jumps[9] directly to the second leaf of T. Here, the tortoise waits again for the hare to arrive. The tortoise continues moving in this way until it reaches the final leaf of T. Here, it waits again until the hare arrives and then walks together with hare from the final leaf all the way to the root. In this way, the tortoise traverses each edge of the tree precisely once (in the direction toward the root). Since moving up the tree corresponds to finishing the recursive call of `Euler-Walk` for the lower vertex, the sequence of vertices visited by the tortoise, ignoring the vertices it jumped from, equals the content of the final list Z computed by `Euler`.

Since each vertex and each edge of T corresponds to vertices and edge in the graph G, the movements of the tortoise and hare correspond in a canonical way to movements in the graph. Since the movement of the hare in T forms a closed walk, its movement in G also forms a closed walk (starting and ending at vertex u_0). The following claim shows that, somewhat surprisingly, also the movement of the tortoise in G forms a closed walk (starting and ending at u_0).

Claim: The root and the first leaf of T correspond to the same vertex in G. Furthermore, for every leaf i besides the first leaf, the lowest common ancestor of leaf $i-1$ and leaf i corresponds to the same vertex in G as leaf i.

This claim shows that the jumps of the tortoise in T translate to staying put in G. Consequently, the tortoise moves along a closed walk in G, whose vertex sequence is equal to the final list Z computed by `Euler`. As noted before, there is a one-to-one correspondence between the edges in G and in T. Since the tortoise traverses each edge in T exactly once, its closed walk in G is a Eulerian tour.

Hence, to establish the lemma, it remains to prove the claim.

Proof: In order to prove this claim, we use the fact that the `Euler-Walk` procedure behaves the same[10] as the `Walk` procedure from sec. 8 except for branching. We

[8] The first leaf corresponds to the first recursive call of `Euler-Walk` that has finished. In general, we order the leafs of T according to the time that the respective recursive calls of `Euler-Walk` has finished.

[9] Although these jumps may not seem tortoise-like, it turns out that, when viewed in the graph G, these jumps actually correspond to staying put.

[10] In order to get a direct correspondence between `Euler-Walk` and `Walk`, we shall assume that both procedures process unmarked edges incident to a vertex in the same order (e.g., according to the alphabetical order of the labels of the neighbors).

will show that the execution of `Euler-Walk`(u_0) can be viewed as a sequence of executions of the `Walk` procedure.

As part of the proof of the above claim, we will show the following statement by induction on $i \geq 1$:

> $P(i)$: If we mark all edges in G visited by the hare until reaching leaf i of T, then `ALL-EVEN` is satisfied (see sec. 8).

Consider the path in T from its root to its first leaf. This path also corresponds[11] to the (non-branching) recursion tree of `Walk`(u_0) initialized with all edges of G unmarked. According to the claim in sec. 8, `Walk`(u_0) marks a closed walk starting at u_0 and ending at u_0. (Here, we use that the property `ALL-EVEN` is satisfied in G when all edges are unmarked.) It follows that the first leaf of T corresponds to the vertex u_0 in G (establishing the first part of the claim).

It also follows that $P(1)$ holds because the `Walk` procedure maintains the `ALL-EVEN` invariant.

We are to show that $P(i-1)$ implies $P(i)$ for $i \geq 2$. Mark all edges in G that appears on the paths from the root of T to the first $i-1$ leafs of T. By the induction hypothesis, $P(i-1)$ holds, which means that `ALL-EVEN` is satisfied. Consider the lower lowest common ancestor A_{i-1} of leaf $i-1$ and leaf i. Suppose A_{i-1} corresponds to vertex u_{i-1} in G. Then, the (non-branching) recursion tree of `Walk`(u_{i-1}) is the same[12] as the path from A_{i-1} to leaf i in T. By the analysis of `Walk` in sec. 8, this path corresponds to a closed walk in G that starts and ends in u_{i-1} (establishing the claim for leaf i). It also follows that $P(i)$ holds because the `Walk` procedure maintains the `ALL-EVEN` invariant. □

[11] In order to get a direct correspondence between `Euler-Walk` and `Walk`, we shall assume that both procedures process unmarked edges incident to a vertex in the same order (e.g., according to the alphabetical order of the labels of the neighbors).

[12] The recursive call of `Euler-Walk` represented by A_{i-1} in T initiates several recursive calls of `Euler-Walk`. Alternatively, we can simulate this branching by calling the `Walk` procedure several times from this vertex.