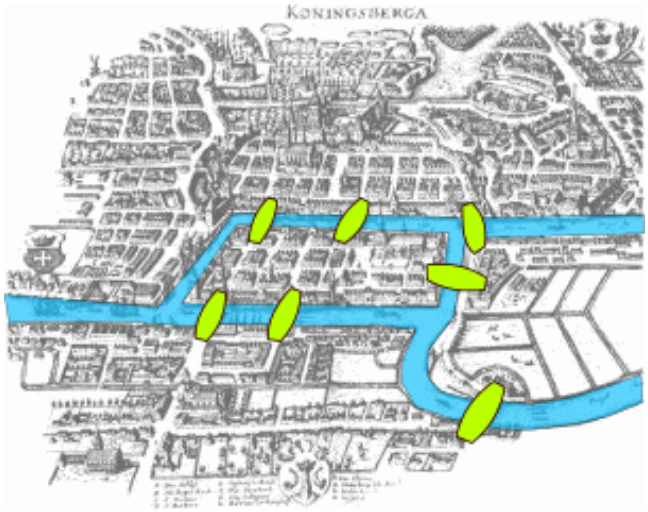
# Simple Graph and Connectivity

## Introduction

Euler's Theorem is a very neat and classic theorem. However, it cannot be easily formalised by Lean yet. In this naive attempt, a simplified version of "The bridges of Konigsburg" is formalised (and proved). A more generalised theorem, Euler's theorem, is also formalised, with a bunch of auxiliary definitions and theorems related to connectivity being formalised as well.




## Definitions, Corollaries and Theorems

Basic graph theory definitions are omitted here. Note that all definitions that is not included in `mathlib`, to be specific, `combinatorics.simple_graph`, will be defined here.

Only undirected simple graph is discussed here as this is the only "sophisticated" type of graph being explored by `mathlib`, although the discussion below can be easily extended to other types like directed graph or more general ones.

### Euler Circuit

**Definition.** Euler Circuit (`is_euler_circuit`).

Sometimes written as "Euler Cycle". For consistency with mathlib, "circuit" is preferred. A circuit $p$ in a graph $G$ is a **Euler circuit** of graph $G$ if this circuit contains all edges in this graph. Formally,

$$\forall e, e \in G.\,\text{edge\_set} \implies e \in p.\,\text{edges} \tag{6}$$

(`is_eulerian`). It is also said that a graph $G$ is **Eulerian** if it contains a Euler circuit. i.e., there is a $p$ walk in $G$ s.t. $p$ is a Euler circuit.

# Euler's Theorem

**Theorem.** Euler's Theorem. (`is_eulerian_iff_all_even_degree`)

"A *connected graph* has an *Euler circuit* if and only if every vertex has even degree."

Noted that `connected graph` is not defined here yet. A series of necessary definitions will be introduced "backwards" below:

# Connected

**Definition.** Connected.

A graph is **connected** if every pair of vertices ($V$) in the graph $G$ is **reachable** to each other.

$$\forall v\ w : V, G.\ \text{reachable}(v, w) \tag{7}$$

**Lemma.** A complete graph is connected (`complete_graph_is_connected`).

Since given any two vertices $v\ w$ in a complete graph $G$, either $v = w$ or $G.\ \text{adj}\ \text{v}\ \text{w}$, by the fact that a walk $p$ can be constructed by i) from same vertex to itself, or ii) from $v$ to $w$ given they are adjacent to each other, $v$ and $w$ will be reachable to each other.

# Reachable

**Definition.** Reachable.

For any two vertices $v\ w$ in graph $G$, to say $v$ is **reachable** to $w$ (or vice versa, since $G$ is undirected), is to say that there is a **walk** (defined in `.simple_graph.connectivity`) from $v$ to $w$,

$$\exists p : G.\ \text{walk}\ \text{v}\ \text{w} \tag{8}$$

**Corollary.** *Reachable* is a equivalence relation.

This is simple to prove. Proofs are given in the coursework files, as

- Reflexivity: `reachable_self`
- Symmetry: `reachable_symm`
- Transitivity: `reachable_trans`

**Note.** The equivalence class in a graph $G$ respect to the binary relation *reachable* is usually called **Connected Components**, where all vertices in a connected components are reachable to each other.

## Existence of Walk from Sub-walk

**Lemma.** A tail part of a walk is still a walk.

This is trivial by the definition of a walk.

**Theorem.** Existence of a walk between two vertices on a same walk.

$$\forall v\ w : V, \exists p : G.\ \text{walk}\ \_\ \_, v, w \in p \implies \exists G.\ \text{walk}\ \text{v}\ \text{w} \tag{9}$$

Pick two vertices $v\ w$.

WLOG, suppose $p$ exists and $v, w \in p$, where $v$ is visited before $w$ is visited in some section of $p$ (i.e., $w$ is closer to `walk.nil`).

We can obtain a walk by dropping all sections before this occurrence of $v$ and then taking all sections until (including) this occurrence of $w$, thus having a walk from $v$ to $w$.

**Corollary.** All vertices on a walk are reachable to each other.

As from the above theorem, we know that for all vertices on a walk, there would be a walk between them; thus, by the definition of *reachable*, they will be reachable to each other.

## Existence of trail / path between reachable vertices

**Theorem.** Existence of trail / path between reachable vertices.

$$\begin{aligned} &\forall v\ w : V, G.\ \text{reachable v w} \implies \exists p : G.\ \text{walk v w}, p.\ \text{is\_trail} \\ &\forall v\ w : V, G.\ \text{reachable v w} \implies \exists p : G.\ \text{walk v w}, p.\ \text{is\_path} \end{aligned} \tag{10}$$

This can be proved easily since the existence of a walk between $v$ and $w$ implies the existence of a path between them (in a undirected graph). Furthermore, all paths are trails, and thus this path would be an example of such a trail between them.

**Proof.** The existence of a path from $v$ to $w$ when $v$ is reachable to $w$.

An algorithm of constructing such a path from $v$ to $w$ will be provided here, and thus to prove the existence.

1. By the definition of reachable, we know there there must be a walk `p` from `v` to `w`.
2. Check if `p` is a path. If so, this is the one we need; otherwise, proceed:
    1. Check if `v` is in the walk `q`. If so, drop all edges in walk `p` from this occurrence of `v` and the last occurrence of `v`, and concat the remaining, set it as the new `p`. Example:
        1. Suppose `p := v - x - a - b - v - z - v - c - w`
        2. Drop all edges in between means to `p := v - c -` with `x - a - b - v - z - v -` being dropped.
    2. `v` should be unique in `p` so far.
    3. Check if `p` is a path. If so, break; otherwise, iteratively convert the remaining parts of the walk `p` as follows:
    4. Separate walk `p` by two parts: the first edge from `v` to a mid point `x`, and the walk `q` from `x` to `w`. This can always be done as `p` must contains at least two edges for now.
        1. If `p` contains less than two edges, `p` must be a path as it does not have repeated edges nor repeated vertices
    5. Repeat step i. to iv. for `q`.

The validness of this algorithm is based on the fact that for all walk `p`, if vertices `x` `y` `z` are involved such that edges `x - y` `x - z` are both involved in a way that forming a walk `·1· - x - y - ·2· - x - z - ·3·`, then there will definitely be a walk passing `x` one less time than `p`, by not going to `y` at first but to `z` directly, thus forming a valid walk `·1· - x - z - ·3·`. Since the validity of walk can be defined iteratively, and `·1· - x` `x - z` `z - ·3·` are all valid walks, concat them together will also yield a valid walk.

The correctness of this algorithm is built on the fact that in each iteration of 2.v., as 2.ii. states, the whole walk will have only one occurrence of the current vertex `v`, and thus having no repetitive vertices. Since all vertices in this walk `p` will be checked until `p` is a path, this elimination is exhaustive, and thus a path will be yielded.

Further, this algorithm has only finite steps if walk `p` is finite, as in each iteration of 2.v., the number of repetitive vertices in total in the walk will be reduced by a positive amount. In fact, the number of iterations required will be no more than the total vertices involved in this walk `p`.

■

Although this proof can be constructed with clear steps, it is hard (and I failed) to formalise them in Lean. The issue relies on the inductive dependent types.

Note that `simple_graph.walk` is dependent on the type of vertices `u v : V` where `V : Type u`, `universe u`. `u` `v` enforced two endpoints of the walk. Since `walk` is defined inductively like a `list`, with each two connected sections always sharing a common vertex in between, it becomes hard to construct a new walk, or re-wire the walk to replace / remove some sections in between, just because the `def` function signature would enforce a type that the recurse step will be hard to follow.

# The Proof of Euler's Theorem

There are two parts of this theorem:

1. all vertices in Eulerian graph have even degrees
2. a connected graph with all its vertices having even degrees is a Eulerian graph

Note that the connectivity of the graph is a pre-requisite, rather than a result of " $\Longleftarrow$ ". If we consider a subgraph with all vertices having a positive degree, then this subgraph of a Eulerian graph would be connected.

## Eulerian Graph is Connected (Non-Zero Degree Vertices Only)

1. Vertices of a non-zero degree must have an edge connecting it
2. A Eulerian graph has a circuit containing all edges
    1. Due to 1., this circuit contains all vertices
3. A circuit is a walk, thus circuit in 2. is a walk
4. All vertices are *reachable* if they are contained by a same walk
5. The walk in 2. is such a walk that contains all vertices. Thus by 4., all vertices in this graph are reachable to each other
6. By definition of *connected*, this graph is connected.

## All Vertices in Eulerian Graph have Even Degrees

Since the graph is Eulerian, it has a circuit $p$ passing all its edges, thus all its vertices (and since the graph is connected, no vertex will have a degree of zero).

Pick an arbitrary vertex $v$.

Suppose the Eulerian circuit $p$ visits $v$ $k$ times. Each time it visits $v$, it must come from one other vertex via one edge, and leaves to another, thus adds up 2 degree. Hence, $v$ will have a degree of $2k$. Note that these 2 edges must be distinct and not used before, as in a Euler circuit $p$, all edges are visited exactly once.

## Graph with All Vertices of Even Degrees is Eulerian

> omitted for now.

# Reflection

Clearly I am not capable to do such a project yet, as `simple_graph` module involves a lot of type manipulations, jumping between Type, Sort and values. Proofs also involves many computations which is hard to be done solely by manipulating terms without constructing a specific thing that satisfy the condition. With these being said, `simple_graph` is far from what is being covered in the course, and thus a lot of extra efforts must be put to generate more meaningful results, which is definitely not enough in my naive attempt.

Many partial proofs are left in the document with my attempts, involving `sorry`s to make the file compile. If possible, I will explore further in this field in future coursework to prove them thoroughly. As future works, I hope Lean can be used to prove correctness or other properties of classical algorithms formally.