1 Problem statement

1.1 Notation

Given a pedigree, represented as a directed acyclic graph $P = \{H, F\}$, where H represents haploid individuals or ploids (vertices in the graph) and F are genealogical filiations (edges in the graph), and a gene (coalescent) tree $C = \{V(C), E(C)\}$ that is 'extracted' from P, find a mapping $M : V(C) \to H$ such that $M(V(C))$ is consistent with P. The definition of being consistent is given by the problem formulation.

1.2 Formulation 0: Vertex only, oracle

(Not sure whether we need this formulation)

In this version of the problem, there exists a map $O : V(C) \to H$ that is the only consistent function between C and P.

Consistency condition: $M = O$.

1.3 Formulation 1: Vertex only

Definition 1 *In a directed graph D, a vertex $v \in V(D)$ is a descendant of a vertex $u \in V(D)$ if there exists a simple directed path $ua_1 \dots a_n v$.*

By $\text{desc}_D(v)$, let's denote the set of all the descendants of v in a directed graph D.

Definition 2 *(Incorrect)*

Let D be a directed graph and $A = \{a_1, \dots, a_n\}$ be a set of vertices in $V(D)$. A vertex $v \in V(D)$ is said to a genealogical common ancestor of A if:

- $A \subset \text{desc}(v)$,

- $\forall u \in \text{desc}(v) : A \not\subset \text{desc}(u)$.

Definition 3 *(Correct)*
Let D be a directed graph, $A = \{a_1, \ldots, a_n\}$ be a set of vertices in $V(D)$, and $v \in V(D)$. Put $C = \{u \in V(D) | A \subset \text{desc}(u)\}$. The vertex $v \in V(D)$ is said to a genealogical common ancestor of A if:

- $v \in C$,
- $A \subset \text{desc}_{D'}(v)$ *where* $D' = D - (C \setminus \{v\})$.

Consistency condition: $\forall v \in V(C)$ with $N_C(u) \neq \emptyset$ it holds that $M(v)$ is a genealogical common ancestor of $M(N_C(u))$.

1.4 Formulation 2: Vertex-only with path-specific output

2 Preprocessing step

The preprocessing step consists of three steps:

2.1 Reading the pedigree

During this step, we simply read the pedigree from a file and save the information about the graph. The time complexity of this step is $O(n)$ where n is the number of vertices in the graph.

2.2 Calculating the levels and descendants lists from the probands

Next, we calculate the levels of the graph and the descendant list for every vertex in the graph. Every vertex has to know its descendants be-

cause we need this information to discard graphical common ancestors that are not genealogical common ancestors later in the algorithm.

Generally speaking, having descendants for every vertex can require a lot of memory. According to my calculations, we should still be able to store it for large graphs. There are different more complicated approaches that can take a little bit more time but save a lot of memory. Since the preprocessing step is done only once, we can afford to spend more time during this step to lower the memory requirements.

In the "path aware" approach, the descendant list also stores the path to every descendant.

2.3 Preprocessing the common ancestors map

Starting with the founders (whose map entries has been already calculated), the algorithm fills the common ancestors map for all the other vertices going level by level.

2.4 Result

In the end, we have a three-dimensional matrix (a dictionary of dictionaries of dictionaries) that returns a common ancestry object for a triple (a, b, c) where c is a common ancestor of a and b.

The common ancestry object contains all the paths by which a and b can reach c.

```
class PathCommonAncestry:

        def __init__(self, common_ancestor: int,
                first_vertex: int, second_vertex: int,
                first_vertex_paths: [Path],
                second_vertex_paths: [Path]):
                self.common_ancestor = common_ancestor
                self.first_vertex = first_vertex
```

```
        self.second_vertex = second_vertex
        self.first_vertex_paths = first_vertex_paths
        self.second_vertex_paths = second_vertex_paths
```

3 Alignment

```
class PathAligner:

def align():
# Get current partial mapping
partial_mapping = get_partial_mapping()
iterator = get_vertex_iterator()
while not iterator.finished():
        # Vertex in the coalescence tree that has not been mapped
        vertex_to_map = iterator.next()
        # Get the candidates for the vertex. If vertex_to_map has only
        # one child (no coalescence), return None
        candidates = get_candidates_for_vertex(vertex_to_map)
        if candidates is None:
                partial_mapping[vertex_to_map] = None
                continue
        result = []
        for candidate in candidates:
                if verify_assignment(vertex_to_map, candidate):
                        result.append(candidate)
        # If there are no candidates whose assignment are consistent,
        # trace back to the previous decision
        if not result:
                return None
        while length(result) > 0:
                candidate_to_try = result.pop()
                aligner = PathAligner(self)
                alignment_result = aligner.align()
                if alignment_result is not None:
                        return alignment_result
        # None of the assignment worked, return None
        return None
```