Alignment approach

```
In [58]: import pandas as pd
         import numpy as np
         import matplotlib.pyplot as plt
         import pyranges
         from collections import namedtuple
         from Bio import Align
         from Bio.Align import substitution_matrices
         from Bio import SeqIO
         from Bio.Seq import Seq
```

```
In [55]: path_maternal="data/shared_contigs_maternal_coordinates.tsv"
         path_paternal="data/shared_contigs_paternal_coordinates.tsv"

         path_PAN028_index = "data/assembly.v1.0.PAN028.diploid-002.fa.fai"

         path_grandmother_genome = "data/assembly.v1.0.PAN010.diploid-006.fa"
         path_grandfather_genome = "data/assembly.v1.0.PAN011.diploid-005.fa"
         path_mother_genome = "data/assembly.v1.0.PAN027.diploid-003.fa"
         path_proband_genome = "data/assembly.v1.0.PAN028.diploid-002.fa"

         path_grandmother_RM = "data/assembly.v1.0.PAN010.diploid.RM.bed"
         path_grandfather_RM = "data/assembly.v1.0.PAN011.diploid.RM.bed"
         path_mother_RM = "data/assembly.v1.0.PAN027.diploid.RM.bed"
         path_proband_RM = "data/assembly.v1.0.PAN028.diploid.RM.bed"
```

```
In [4]: df_maternal = pd.read_csv(path_maternal, sep="\t", header=None, names=["Contig", "Grandparent", "Mother", "Proband"])

        # Load repeat annotations as PyRanges objects
        rm_grandfather = pyranges.readers.read_bed(path_grandfather_RM)
        rm_grandmother = pyranges.readers.read_bed(path_grandmother_RM)
        rm_mother = pyranges.readers.read_bed(path_mother_RM)
        rm_proband = pyranges.readers.read_bed(path_proband_RM)
```

```
In [ ]: def split_info(info): # PAN010.chr1.haplotype1:142532907-142550224
            if pd.isna(info):
                return pd.Series([None, None, None])
            else:
                info = info.split(":")
                chr = info[0]
                positions = info[1].split("-")
                start = int(positions[0])
                end = int(positions[1])
                return pd.Series([chr, start, end])
        def preprocess_rm(rm_df):
            """
            Get rid of simple repeats
            """
            rm_filt = rm_df[rm_df["ThickEnd"] != "unknown"]


            return rm_filt

        def extract_repeat_annotations(info, rm_bed):
            if pd.isna(info):
                return None
            else:
                chr, start, end = split_info(info)
                rm_filtered = rm_bed[chr, start:end]
                rm_df = rm_filtered.df.sort_values(by="Start")
                rm_preprocessed = preprocess_rm(rm_df)
                rm_list = rm_preprocessed[["Chromosome", "Start", "End", "Name", "Score", "Strand",
                                           "ThickStart", "ThickEnd", "ItemRGB", "BlockCount"]].values.tolist()

                return rm_list

        def repeat_to_token(r):
            RepeatToken = namedtuple("RepeatToken", ["name", "length", "strand", "type", "family"])
            return RepeatToken(r[3], r[4], r[5], r[6], r[7])

        def match_score(token1, token2, match_tolerance=0.1):
            name1, len1, strand1, type1, family1 = token1
            name2, len2, strand2, type2, family2 = token2

            score = 0
            if type1 == type2:
                score += 1
            if family1 == family2:
                score += 1
            if strand1 == strand2:
                score += 1
            else:
                score -= 1
            if name1 == name2:
                score += 2
            if abs(len1 - len2) <= match_tolerance * max(len1, len2):
                score += 1

            return score

        def align_repeat_lists(proband, grandparent, gap_penalty=-2, match_tolerance=0.1):
            # Convert repeat annotations to tokens
            tokens_p = [repeat_to_token(r) for r in proband]
            tokens_g = [repeat_to_token(r) for r in grandparent]

            m = len(tokens_p)
```

```
    n = len(tokens_g)

    # Initialize scoring matrix
    M = np.zeros((m + 1, n + 1), dtype=int)

    # Fill the first row and column with gap penalties
    for i in range(1, m + 1):
        M[i][0] = M[i - 1][0] + gap_penalty
    for j in range(1, n + 1):
        M[0][j] = M[0][j - 1] + gap_penalty

    # Fill in the matrix
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            match = M[i - 1][j - 1] + match_score(tokens_p[i - 1], tokens_g[j - 1], match_tolerance)
            delete = M[i - 1][j] + gap_penalty
            insert = M[i][j - 1] + gap_penalty
            M[i][j] = max(match, delete, insert)

    # Traceback
    aligned_proband = []
    aligned_grandparent = []

    i, j = m, n
    while i > 0 and j > 0:
        current_score = M[i][j]
        diag = M[i - 1][j - 1]
        up = M[i - 1][j]
        left = M[i][j - 1]

        if current_score == diag + match_score(tokens_p[i - 1], tokens_g[j - 1], match_tolerance):
            aligned_proband.insert(0, proband[i - 1])
            aligned_grandparent.insert(0, grandparent[j - 1])
            i -= 1
            j -= 1
        elif current_score == up + gap_penalty:
            aligned_proband.insert(0, proband[i - 1])
            aligned_grandparent.insert(0, None)  # gap in grandparent = insertion
            i -= 1
        else:
            aligned_proband.insert(0, None)  # gap in proband = deletion
            aligned_grandparent.insert(0, grandparent[j - 1])
            j -= 1

    # Add any remaining elements (at edges)
    while i > 0:
        aligned_proband.insert(0, proband[i - 1])
        aligned_grandparent.insert(0, None)
        i -= 1
    while j > 0:
        aligned_proband.insert(0, None)
        aligned_grandparent.insert(0, grandparent[j - 1])
        j -= 1

    return aligned_proband, aligned_grandparent, M

def format_alignment_to_table(alignment):
    aligned_grandparent = alignment[0]
    aligned_proband = alignment[1]

    table = []
    insertions = []
    for i, (p, g) in enumerate(zip(aligned_proband, aligned_grandparent)):
        row = {
            "Proband": p[3] if p else "-",
            "Grandparent": g[3] if g else "-",
            "Type": p[6] if p else g[6] if g else "-",
            "Family": p[7] if p else g[7] if g else "-",
            "Status": "Match" if p and g else "Insertion" if p else "Deletion"
        }
        table.append(row)
        if p and not g:
            ancestor_chromosome = aligned_grandparent[i-1][0] if i > 0 and aligned_grandparent[i - 1] else None
            ancestor_start = aligned_grandparent[i - 1][2] if i > 0 and aligned_grandparent[i - 1] else None
            ancestor_end = aligned_grandparent[i + 1][1] if i < len(aligned_grandparent) - 1 and aligned_grandparent[i + 1] else None
            p.extend([ancestor_chromosome, ancestor_start, ancestor_end])
            insertions.append(p)

    cols = ["Chromosome", "Start", "End", "Name", "Length", "Strand", "Type", "Family",
            "Score", "Index", "AncestorChromosome", "AncestorStart", "AncestorEnd"]
    insertions_df = pd.DataFrame(insertions, columns=cols)
    return pd.DataFrame(table), insertions_df
```

Single-thread

In [ ]:
```
all_insertions_df = pd.DataFrame()
alignment_dict = {}
for row in df_maternal.itertuples(index=False):
    contig, grandparent_info, mother_info, proband_info = row
    grandparent_chr, grandparent_start, grandparent_end = split_info(grandparent_info)
    mother_chr, mother_start, mother_end = split_info(mother_info)
    proband_chr, proband_start, proband_end = split_info(proband_info)

    grandparent_repeats = extract_repeat_annotations(grandparent_info, rm_grandmother)
    mother_repeats = extract_repeat_annotations(mother_info, rm_mother)
    proband_repeats = extract_repeat_annotations(proband_info, rm_proband) if not pd.isna(proband_info) else None

    if len(grandparent_repeats) > 2000:
        n_chunks = len(grandparent_repeats) // 2000 + 1
        chunk_size = len(grandparent_repeats) // n_chunks
        for chunk in range(n_chunks):
            start_id_descendant = max(1, chunk_size * chunk)
            end_id_descendant = min(len(mother_repeats), chunk_size * (chunk + 1))
            mother_repeats_chunk = mother_repeats[start_id_descendant: end_id_descendant]
            grandparent_repeats_chunk = grandparent_repeats[max(1, start_id_descendant-20): min(len(grandparent_repeats), end_id_descendant+20)]
            alignment = align_repeat_lists(grandparent_repeats_chunk, mother_repeats_chunk)
            df, insertions = format_alignment_to_table(alignment)
            insertions["Contig"] = contig
            all_insertions_df = pd.concat([all_insertions_df, insertions], ignore_index=True)
            alignment_dict[f"{contig}_{chunk}"] = df
    else:
        alignment = align_repeat_lists(grandparent_repeats, mother_repeats)
        df, insertions = format_alignment_to_table(alignment)
        insertions["Contig"] = contig
        all_insertions_df = pd.concat([all_insertions_df, insertions], ignore_index=True)
        alignment_dict[contig] = df
all_insertions_df.to_csv("data/insertions_grandmother_mother_single_thread.tsv", sep="\t", index=False)
```

Parallel

In [ ]:

```
from multiprocessing import Pool, cpu_count
from tqdm import tqdm
import pandas as pd

def process_row(row):
    contig, grandparent_info, mother_info, proband_info = row

    grandparent_repeats = extract_repeat_annotations(grandparent_info, rm_grandmother)
    mother_repeats = extract_repeat_annotations(mother_info, rm_mother)
    proband_repeats = extract_repeat_annotations(proband_info, rm_proband) if pd.notna(proband_info) else None

    result = []
    local_alignment_dict = {}

    if len(grandparent_repeats) > 2000:
        n_chunks = len(grandparent_repeats) // 2000 + 1
        chunk_size = len(grandparent_repeats) // n_chunks
        for chunk in range(n_chunks):
            start = max(1, chunk_size * chunk)
            end = min(len(mother_repeats), chunk_size * (chunk + 1))
            mother_chunk = mother_repeats[start:end]
            grandparent_chunk = grandparent_repeats[max(1, start - 20):min(len(grandparent_repeats), end + 20)]
            alignment = align_repeat_lists(grandparent_chunk, mother_chunk)
            df, insertions = format_alignment_to_table(alignment)
            insertions["Contig"] = contig
            result.append(insertions)
            local_alignment_dict[f"{contig}_{chunk}"] = df
    else:
        alignment = align_repeat_lists(grandparent_repeats, mother_repeats)
        df, insertions = format_alignment_to_table(alignment)
        insertions["Contig"] = contig
        result.append(insertions)
        local_alignment_dict[contig] = df

    return pd.concat(result, ignore_index=True), local_alignment_dict

# Parallel execution
if __name__ == "__main__":
    all_insertions_df = pd.DataFrame()
    alignment_dict = {}

    with Pool(cpu_count()) as pool:
        results = list(tqdm(pool.imap(process_row, df_maternal.itertuples(index=False, name=None)), total=len(df_maternal)))

    # Combine results
    all_insertions_df = pd.concat([res[0] for res in results], ignore_index=True)
    for res in results:
        alignment_dict.update(res[1])

    # Save output
    all_insertions_df.to_csv("data/insertions_grandmother_mother.tsv", sep="\t", index=False)
```