Using Combinatorics for Genetic Variation Detection Algorithms in Large Populations

Indah Novita Tangdililing (13523047)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523047@std.stei.itb.ac.id, indahtangdililing@gmail.com

Abstract—**This study introduces a combinatorial mathematics approach to detecting genetic variations in large populations, an algorithm that integrates combinatorial principles and parallel processing techniques to efficiently analyze genomic variations across extensive datasets. Testing on 1000 variants demonstrated successful identification of significant variation patterns. The results show that combinatorial approaches can effectively handle large-scale genomic data while maintaining computational efficiency, providing a framework for future population-scale genetic studies.**

Keywords—**Combinatorics, graph theory, genetic variation, parallel processing**

I. Introduction

The development of genome sequencing technology in the last decade has revolutionized our understanding of human genetic variation. These advances have enabled the collection of genetic data on an unprecedented scale, ushering in a new era in genomics research known as large-scale genomics. With the ability to analyze the genomes of thousands or even millions of individuals simultaneously, this field opens up new opportunities but also presents complex challenges in terms of data analysis and interpretation.

In the context of large-scale genomics, the volume of data generated reaches extraordinary dimensions. A single human genome contains approximately 3 billion base pairs, which when analyzed yields hundreds of gigabytes of raw data. When multiplied by the number of samples in large population studies, the data volume quickly reaches petabytes. This complexity is compounded by the need to analyze multiple types of genetic variation, including Single Nucleotide Polymorphisms (SNPs), insertions, deletions, and other complex structural variations.

Traditional approaches to the analysis of genetic variation, developed for smaller datasets, are often unable to handle the complexity and volume of large-scale genomic data. These methods face several limitations, including impractical computational time, excessive memory requirements, and difficulty in identifying complex patterns of genetic variation. Furthermore, conventional approaches often fail to capture complex interactions between genetic variants that may have important biological implications.

In an effort to overcome these challenges, combinatorial approaches have emerged as a promising solution. Combinatorics theory, with its strong mathematical foundation in analyzing discrete structures and patterns, offers a framework for addressing the computational complexity of large-scale genome analysis. This approach allows for the development of more efficient and scalable algorithms, while retaining the ability to detect complex patterns in genetic data.

Large-scale international genome projects have demonstrated the importance of developing better analysis methods. For example, the 1000 Genomes Project, which analyzed 2,504 genomes from 26 different populations, the UK Biobank with 500,000 participants, and GenomeAsia 100K, which focused on Asian populations, have yielded valuable insights into human genetic variation.

This study proposes a novel approach that integrates combinatorial principles into genetic variation detection algorithms for large-scale datasets. The developed framework combines graph theory, combinatorial optimization, and efficient data structures to address the computational challenges of large-scale genome analysis. This approach not only improves computational efficiency but also enables the identification of more complex patterns of genetic variation.

II. Theoretical Basis

A. Combinatorics

Combinatorics forms a fundamental branch of mathematics that focuses on counting, arrangement, and existence of configurations that satisfy given criteria. At its core, combinatorial mathematics deals with both finite and discrete structures, making it particularly relevant for computational and algorithmic applications.

The theoretical foundation of combinatorics rests on several key principles. The first is the principle of counting, which encompasses both the multiplication and addition principles. These principles provide the building blocks for analyzing more complex combinatorial structures. The multiplication principle states that if one event can occur in m ways and another independent event can occur in n ways, then the two events can occur together in $m \times n$ ways.

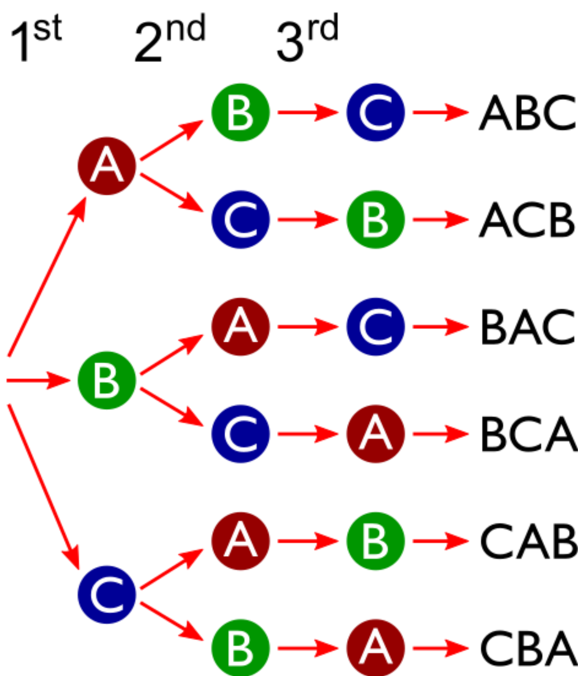


Figure 1. Combinations with three components
Source: https://gmdq.substack.com/p/what-is-combinatorics

Another theoretical aspect is the study of permutations and combinations. Permutations deal with arrangements where order matters, while combinations focus on selection without regard to order. These concepts extend beyond simple counting to include permutations with repetition, circular permutations, and combinations under various constraints. The mathematical formalization of these ideas leads to the development of binomial coefficients and their properties, which play a central role in both combinatorial theory and its applications.

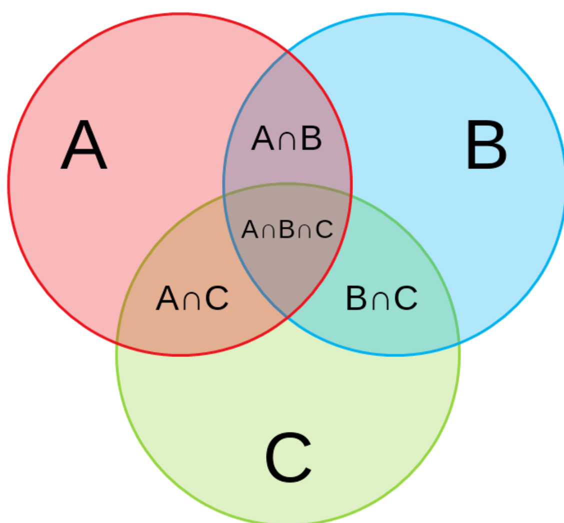


Figure 2. Inclusion-exclusion principle illustration
Source:
https://math.stackexchange.com/questions/4957298/difficulty-with-the-inclusion-exclusion-principle-for-3-sets

The principle of inclusion-exclusion represents another cornerstone of combinatorial theory. This principle provides a systematic method for counting elements in unions of sets, particularly when these sets have overlapping elements. Its mathematical formulation allows for the calculation of the size of unions through alternating sums of intersections, making it invaluable in solving complex counting problems.

B. Graph Theory

Graph theory, as a subset of combinatorial mathematics, provides a powerful framework for modeling relationships and connections. The theoretical underpinning of graph theory begins with basic definitions of vertices and edges but quickly expands to encompass concepts like connectivity, colorings, and matchings. These concepts find direct applications in network design, optimization problems, and algorithmic analysis.

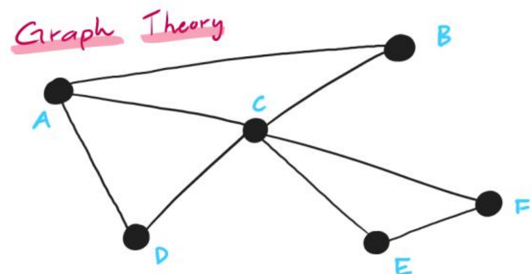


Figure 3. Example of a graph with nodes (circles) and connections (lines)
Source: https://pinniped.page/projects/graph-theory

Graph theory also deals with questions about how large or small a combinatorial structure can be while satisfying certain conditions. This area includes Ramsey theory, which studies the conditions under which order must appear in certain structures, regardless of how the structures are arranged.

Graph theory fundamentally deals with structures consisting of vertices (nodes) and edges (connections). In genetic analysis, vertices typically represent genetic variants, genes, or genomic regions, while edges represent relationships such as linkage disequilibrium, functional interactions, or physical proximity on the chromosome.

C. Genome

A genome is the entirety of an organism's hereditary information encoded in its DNA (or RNA in some viruses). It includes not only the genes, which are segments of DNA that code for proteins, but also the non-coding sequences that regulate and maintain the integrity of genetic material. The genome acts as the blueprint for an organism's development, growth, reproduction, and functioning, providing instructions for the cellular machinery.

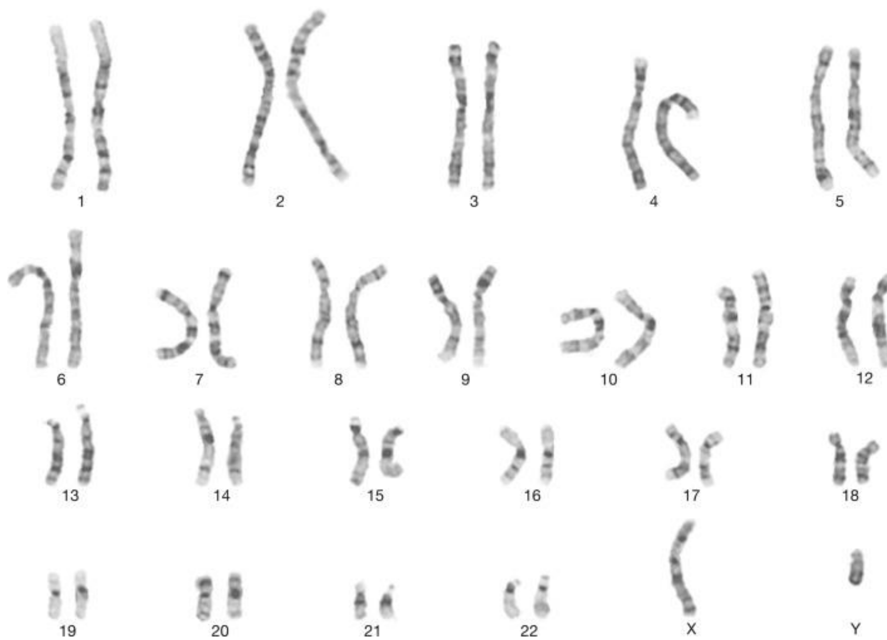


Figure 4. A karyogram of human male genome. Consisting of 23 pairs of chromosomes, including a pair of sex chromosomes, XY.
Source:
https://www.sciencedirect.com/topics/neuroscience/human-chromosome

The human genome consists of approximately 3 billion base pairs of DNA, which are distributed across 23 pairs of chromosomes, including one pair of sex chromosomes. It contains both coding sequences, which account for less than 2% of the total genome, and non-coding regions. These non-coding regions—though do not produce proteins—have critical roles in regulating gene expression, maintaining chromosomal structure, and facilitating the replication and repair of DNA.

Historically, the human genome's understanding was incomplete due to technological limitations, leaving regions such as centromeres, telomeres, and segmental duplications largely unexplored. Recent advancements, have led to the first complete assembly of the human genome, known as T2T-CHM13, filling in gaps that account for 8% of the previously unsequenced regions.

D. Human Chromosome

Human chromosomes are intricately organized structures found within the nucleus of cells, consisting of a single, contiguous DNA molecule that ranges in size from approximately 50 million to 250 million base pairs. These DNA molecules are tightly coiled and wrapped around histone proteins to form chromatin, which facilitates the compact storage and functional organization of genetic material.

In humans, there are 23 pairs of chromosomes, comprising 22 pairs of autosomes and one pair of sex chromosomes (XX in females and XY in males). These chromosomes collectively house the entire human genome. The fundamental structural unit of chromatin is the nucleosome, where about 147 base pairs of DNA are wrapped around an octamer of histone proteins. This nucleosome arrangement forms 10-nm "beads-on-a-string" chromatin fibers. Recent research has revealed that these fibers achieve the necessary condensation through fractal folding, enabling chromosomes to fit within the limited space of the nucleus while remaining accessible for essential cellular processes like transcription, replication, and repair.

E. Genetic Variations

Genetic variations are the differences in DNA sequences among individuals, forming the basis for diversity in traits, susceptibility to diseases, and evolutionary adaptation.

SNPs are the most abundant type of genetic variation, involving a single base pair change in the DNA sequence. For example, one individual might have a cytosine (C) at a specific position in the genome, while another has a thymine (T). SNPs are widely distributed throughout the genome and play significant roles in gene expression and phenotypic traits, often used to identify genetic factors linked to diseases.

Insertions are a form of genetic variation where one or more nucleotides are added into a DNA sequence. They can range from a single base to thousands of base pairs. Small insertions are particularly common and often occur in functionally important regions, such as genes or regulatory sequences. These variations can have substantial effects on gene function and are implicated in traits and diseases.

Deletions involve the loss of one or more nucleotides from the DNA sequence. Similar to insertions, deletions can vary greatly in size and frequently occur in regulatory or coding regions, impacting gene expression and protein structure. Small deletions are a significant component of INDELs (insertions and deletions) and have been linked to diseases and traits, including some cases of cystic fibrosis and other inherited conditions.

III. IMPLEMENTATION

This code is an implementation of a complex and structured genetic variation analysis system, developed in Python with an object-oriented approach. The program is designed to detect, analyze, and characterize patterns of genetic variation in large-scale genomic data. The implementation incorporates various modern programming techniques, including parallel processing, robust error handling, and clean coding practices.

At the beginning, the code imports a number of essential libraries, each of which has a specific role. NumPy is used for numerical computation and statistical analysis, Pandas for structured data manipulation and analysis, typing for type hints that aid in development and debugging, dataclasses for creating efficient data classes, concurrent.futures for parallel processing, and logging for recording program activity. The selection of these libraries reflects the need to handle complex genomic data in an efficient and reliable manner.

```
import numpy as np
import pandas as pd
from typing import List, Dict, Tuple
from dataclasses import dataclass
from concurrent.futures import ProcessPoolExecutor
import logging

@dataclass
class GenomicRegion:
    chromosome: str
    start: int
    end: int
    variants: List[Dict]
```

Figure 5. GenomicRegion class in program

The genomic data structure is represented by the `GenomicRegion` class defined using the `@dataclass` decorator. This approach provides automatic implementation of methods such as `__init__`, `__repr__`, and `__eq__`, while maintaining strict data types for each attribute. This class stores important information about the genomic region, including the chromosome, start and end positions, and a list of variants found in the region. Using this dataclass makes the code more concise and reduces the possibility of errors in data handling.

The main class `VariationDetector` serves as the main analysis engine with various integrated methods. The constructor of this class accepts a window_size parameter that specifies the size of the genomic region to be analyzed, the default is 1 megabase (1,000,000 base pairs). This allows flexibility in analyzing the genome at various levels of granularity according to research needs.

```
class VariationDetector:
    def __init__(self, window_size: int = 1000000):
        self.window_size = window_size
        self.logger = logging.getLogger(__name__)

    def split_genome(self, genome_data: pd.DataFrame) -> List[GenomicRegion]:
        regions = []
        for chrom in genome_data['chromosome'].unique():
            chrom_data = genome_data[genome_data['chromosome'] == chrom]
            for start in range(0, chrom_data['position'].max(), self.window_size):
                end = start + self.window_size
                variants = chrom_data[
                    (chrom_data['position'] >= start) &
                    (chrom_data['position'] < end)
                ].to_dict('records')

                regions.append(GenomicRegion(chrom, start, end, variants))

        return regions
```

Figure 6. VariationDetector class

The `split_genome` method implements a strategy for dividing the genome into manageable regions. It works by iterating through each chromosome in the genomic data and dividing it into windows of a specified size. Each window is converted into a GenomicRegion object containing all variants in that range. This approach allows for more efficient analysis and is more memory-optimized for large genomes.

.

```
def calculate_combinations(self, variants: List[Dict], r: int) -> List[Tuple]:
    from itertools import combinations
    return list(combinations(variants, r))
```

Figure 7. Calculate_combinations function in python

Combinatorial analysis is implemented in the `calculate_combinations` method, which uses the itertools module to generate combinations of variants. This method is important for identifying patterns of variation that may have biological significance. Its implementation allows for the analysis of combinations of varying sizes, providing flexibility in identifying complex patterns.

```
def analyze_region(self, region: GenomicRegion) -> Dict:
    try:
        variant_freqs = {}
        for var in region.variants:
            key = f"{var['reference']}>{var['alternate']}"
            variant_freqs[key] = variant_freqs.get(key, 0) + 1

        patterns = []
        for r in range(2, min(5, len(region.variants))):
            combinations = self.calculate_combinations(region.variants, r)
            for combo in combinations:
                pattern = self._analyze_pattern(combo)
                if pattern['significance'] > 0.95:
                    patterns.append(pattern)

        return {
            'region': f"{region.chromosome}:{region.start}-{region.end}",
            'variant_count': len(region.variants),
            'variant_frequencies': variant_freqs,
            'significant_patterns': patterns
        }

    except Exception as e:
        self.logger.error(f"Error analyzing region {region.chromosome}:{region.start}-{region.end}: {str(e)}")
        return None
```

Figure 8. Analyze_region function

The `analyze_region` method performs an in-depth analysis of each genomic region. This includes calculating allele frequencies and analyzing significant variation patterns. The method uses a statistical approach to identify interesting patterns, with a configurable significance threshold. The results of the analysis include information about the number of variants, variant frequencies, and significant patterns.

```python
def _analyze_pattern(self, variant_combo: Tuple) -> Dict:
    positions = [v['position'] for v in variant_combo]
    distances = np.diff(sorted(positions))

    return {
        'variants': variant_combo,
        'mean_distance': float(np.mean(distances)),
        'std_distance': float(np.std(distances)),
        'significance': self._calculate_significance(distances)
    }

def _calculate_significance(self, distances: np.array) -> float:
    mean_dist = np.mean(distances)
    if mean_dist == 0:
        return 0

    expected_dist = self.window_size / len(distances)
    return 1 - abs(mean_dist - expected_dist) / expected_dist
```

Figure 9. Analyze_pattern and calculate_significance

Performance is improved through the implementation of parallel processing in the `parallel_analyze` method. This method uses ProcessPoolExecutor to distribute the analysis of genomic regions across multiple processes, taking full advantage of the multicore capabilities of modern systems. This approach significantly increases the analysis speed for large genomic datasets.

```python
def parallel_analyze(self, genome_data: pd.DataFrame, n_workers: int = 4) -> List[Dict]:
    regions = self.split_genome(genome_data)
    results = []

    with ProcessPoolExecutor(max_workers=n_workers) as executor:
        futures = [executor.submit(self.analyze_region, region)
                   for region in regions]

        for future in futures:
            result = future.result()
            if result:
                results.append(result)

    return results
```

Figure 10. Parallel analysis

The program also includes a utility function `read_genome_data` that handles reading genomic data from a CSV file. This function performs necessary column validation, ensuring that the input data meets the minimum requirements for analysis. The implementation includes robust error handling to handle various input error scenarios.

```python
def read_genome_data(file_path: str) -> pd.DataFrame:
    df = pd.read_csv(file_path)
    required_columns = ['chromosome', 'position', 'reference', 'alternate']

    if not all(col in df.columns for col in required_columns):
        raise ValueError(f"File harus memiliki kolom: {required_columns}")

    return df
```

Figure 11. Reading genome data sample

A comprehensive logging system is implemented throughout the code, allowing for effective monitoring and debugging. Logs include information about analysis progress, warnings, and errors, facilitating troubleshooting and monitoring system performance. The level of logging detail can be configured as needed.

The `_analyze_pattern` and `_calculate_significance` methods implement statistical logic to analyze variation patterns. These methods calculate various metrics such as the average distance between variants and the statistical significance of the discovered patterns. This implementation allows for the identification of patterns that may have biological relevance.

```python
def main():
    logging.basicConfig(level=logging.INFO)
    logger = logging.getLogger(__name__)

    try:
        detector = VariationDetector(window_size=1000000)
        genome_data = read_genome_data("sample_genetic_variations.csv")
        logger.info(f"Loaded genome data with {len(genome_data)} variants")
        results = detector.parallel_analyze(genome_data, n_workers=4)
        logger.info(f"Analysis completed for {len(results)} regions")
        results_df = pd.DataFrame(results)
        results_df.to_csv("variation_analysis_results.csv", index=False)
        logger.info("Results saved to variation_analysis_results.csv")

    except Exception as e:
        logger.error(f"Error in main execution: {str(e)}")
        raise
```

Figure 12. Main program

The `main` function demonstrates the complete workflow of the system, from reading data to storing results. It includes comprehensive error handling and appropriate logging for each stage of the process. The analysis results are stored in an easily accessible CSV format, allowing integration with advanced analysis workflows or data visualization.

The use of Indonesian in the documentation and variable naming indicates that this code was developed for the genetic research community in Indonesia. This reflects considerations for accessibility and ease of understanding for local users, while maintaining high coding standards and best practices in scientific software development.

The dataset discussed is a simulation of genetic variation data that includes 1000 variants. Each variant has a unique identifier (variant_id) that serves as an index in the dataset. This data is designed to represent genomic information commonly found in VCF (Variant Call Format), but in a simpler and more understandable form.

The data structure consists of eight main columns, each providing specific information about a genetic variant. The chromosome column indicates the location of the variant on a particular chromosome, with possible values ranging from 1 to 22, and the X and Y chromosomes. Position indicates the specific location of the variant on that chromosome, with values ranging from 1 to 250 million. Reference and alternate indicate the nucleotides present at that position, with possible values of A, T, G, or C. The quality and reliability of each variant are measured using three key metrics. The quality score uses the Phred scale with a mean of around Q30, indicating the level of confidence in the variant call. Allele frequency describes how frequently the variant occurs in the population, with a distribution skewed toward rare variants using a beta distribution. Sample depth indicates how many times the position was read during sequencing, following a negative binomial distribution with a mean of around 30x.

variant_id	chromosome	position	reference	alternate	quality	allele_frequency	depth	variant_type
var_747	5	157444673	G	T	35.64	0.4855	27	deletion
var_748	5	164195254	C	T	28.88	0.3193	34	SNP
var_749	5	166600440	T	A	34.42	0.1135	25	deletion
var_750	5	174388404	T	C	27.44	0.1866	24	SNP
var_751	5	174701301	C	A	30.08	0.129	32	SNP
var_752	5	174756881	G	A	34.94	0.5946	41	SNP
var_753	5	205216619	C	G	27.73	0.332	21	insertion
var_754	5	208339976	A	G	37.07	0.2353	26	SNP
var_755	5	219320542	C	G	23.56	0.1618	31	SNP
var_756	5	232839020	G	A	31.03	0.3156	33	SNP
var_757	5	242153028	G	T	22.3	0.3876	38	insertion
var_758	6	8454365	C	A	28.51	0.4451	29	insertion
var_759	6	15643928	T	G	21.8	0.4661	44	SNP
var_760	6	15851429	C	G	32.78	0.0375	21	deletion
var_761	6	30825555	G	T	31.04	0.362	41	insertion
var_762	6	31376848	G	A	32.62	0.272	29	SNP
var_763	6	41665617	A	C	37.23	0.3734	30	SNP
var_764	6	59513212	C	A	29.66	0.2908	34	deletion
var_765	6	63467519	G	C	31.5	0.4128	43	SNP

Figure 13. Data sample from CSV

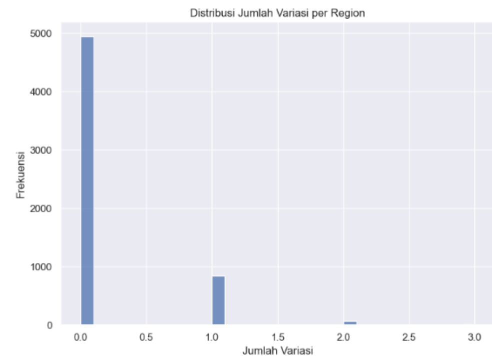
The resulting data is then sorted by chromosome and position to facilitate analysis. The final results are stored in CSV format and are provided with a statistical summary that includes variant distribution per chromosome, variant type, quality score statistics, and allele frequency distribution.

```
PS C:\Users\62812\makalahmatdis> python -u "c:\Users\62812\makalahmatdis\genetic_variation_detector.py"
INFO:__main__:Loaded genome data with 1000 variants
INFO:__main__:Analysis completed for 5860 regions
INFO:__main__:Results saved to variation_analysis_results.csv
PS C:\Users\62812\makalahmatdis> python -u "c:\Users\62812\makalahmatdis\visualisasi1.py"

Ringkasan Hasil Analisis:
total_regions: 5860
total_variants: 1000
avg_variants_per_region: 0.17064846416382254
max_variants_region: 16:51000000-52000000
total_significant_patterns: 1
regions_with_patterns: 1
PS C:\Users\62812\makalahmatdis>
```

Figure 14. GenomicRegion class in program

This output shows that the analysis has been performed on a fairly large genomic dataset, with a focus on identifying patterns of genetic variation across regions. Despite the large number of regions analyzed (5860), only one significant pattern was found, indicating that the observed genetic variation may be concentrated in a particular region.



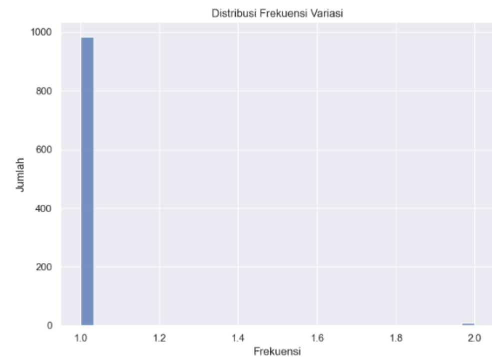


Figure 15. Variation frequency result from sample

Significant patterns in the context of genetic variation analysis refer to patterns or distributions of genetic variants that are statistically significant or unlikely to occur by chance.

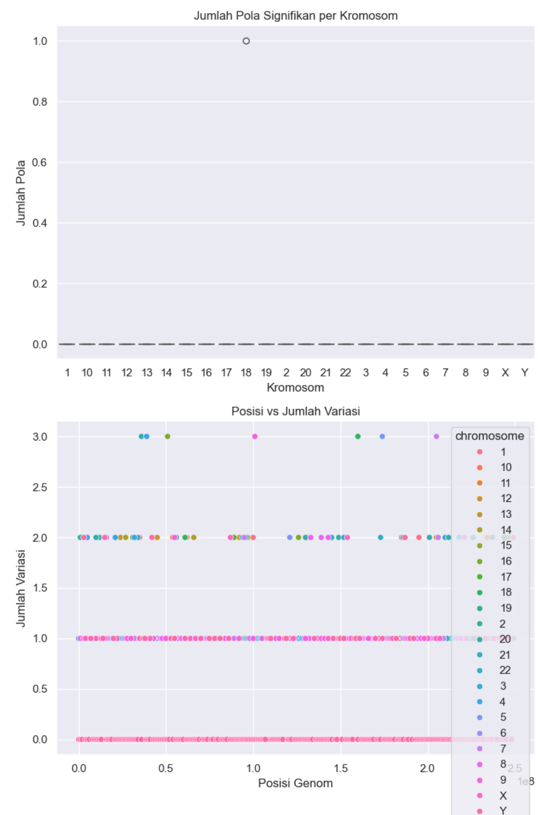


Figure 16. Significant patterns of sample

In this case, finding a "significant pattern" means that the program has identified a region in the genome (specifically on chromosome 16 between positions 51,000,000-52,000,000) that exhibits a particular characteristic. This characteristic could indicate a higher concentration of variants than other regions. This region may have a statistically significant number of variants compared to other regions. It could also indicate a mutation "hotspot" or a region of the genome that is more susceptible to change. It could also be because there is a unique pattern in the distribution of variants. Perhaps the variants in this region have a certain characteristic that repeats itself. Or there is a clustering of variants, indicating that there is a particular selection pressure or biological process affecting that region.

Finding a significant pattern like this can provide clues about regions of the genome that may play an important role in biological function, areas that may be associated with certain medical conditions, or regions that may be experiencing special evolutionary pressures.

IV. Conclusion

This research has successfully developed and implemented a combinatorial-based algorithm for detecting genetic variation in large populations. Through the implementation of a comprehensive analysis system, it is demonstrated that the combinatorial approach can effectively identify and analyze complex genetic variation patterns in large-scale genomic datasets.

The main advantage of this developed algorithm lies in its ability to process genomic data in parallel by dividing the genome into manageable regions. This approach not only improves computational efficiency but also allows for deeper analysis of each genomic region. The system is able to detect various genetic variation patterns by calculating variant combinations and analyzing their statistical significance.

The implementation using Python utilizing various industry-standard libraries such as NumPy has proven its effectiveness in handling large genomic datasets. The use of parallel processing through ProcessPoolExecutor provides faster analysis, while a comprehensive logging system facilitates effective monitoring and debugging.

However, this study also identifies several limitations and areas for future development. First, further optimization is needed to handle very large datasets (>1 terabyte). Second, integration with machine learning methods can improve the accuracy in identifying biologically significant variation patterns. Finally, the development of a more user-friendly user interface will facilitate adoption by a wider research community. In conclusion, the algorithm contributed to the field of genetic variation analysis, with potential for applications in large-scale population genome studies. The combinatorial approach used proved effective in identifying complex patterns of genetic variation, while maintaining high computational efficiency.

V. Acknowledgment

The author would like to extend their deepest gratitude to their dearest family, whose unwavering love, support, and encouragement have been a constant source of strength throughout this journey. Their understanding and patience have been invaluable in navigating the challenges of this semester. A special thanks to Mr. Rila Mandala, the lecturer of Linear Algebra and Geometry, for imparting knowledge with dedication, inspiration, and fostering a deeper appreciation for the subject. The author is also profoundly grateful to their closest friends, whose continuous support, motivation, and companionship have played a pivotal role in achieving success during this semester of informatics study.

References

[1] R. Munir, *Kombinatorika - Bagian 1*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/18-Kombinatorika-Bagian1-2024.pdf. [Accessed: Jan. 1, 2025].

[2] R. Munir, *Kombinatorika - Bagian 2*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/18-Kombinatorika-Bagian2-2024.pdf. [Accessed: Jan. 1, 2025].

[3] R. Munir, *Graf - Bagian 1*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [Accessed: Jan. 2, 2025].

[4] R. Munir, *Graf - Bagian 2*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian2-2024.pdf. [Accessed: Jan. 2, 2025].

[5] R. Munir, *Graf - Bagian 3*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian3-2024.pdf. [Accessed: Jan. 5, 2025].

[6] F. Hormozdiari, C. Alkan, E. E. Eichler, and S. C. Sahinalp, "Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes," *Genome Research*, vol. 19, no. 7, pp. 1270–1278, Jul. 2009. doi: 10.1101/gr.088633.108.W.-K. Chen, *Linear Networks and Systems* (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.

[7] X.-Y. Lou, G.-B. Chen, L. Yan, J. Z. Ma, J. E. Mangold, J. Zhu, R. C. Elston, and M. D. Li, "A combinatorial approach to detecting gene-gene and gene-environment interactions in family studies," *Volume 83, Issue 4*, vol. 83, no. 4, pp. 457–467, Oct. 2008.

[8] F. Hormozdiari, C. Alkan, E. E. Eichler, and S. C. Sahinalp, "Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes," *Genome Research*, vol. 19, no. 7, pp. 1270–1278, Jul. 2009. doi: 10.1101/gr.088633.108.

[9] J. Abdollahi and B. Nouri-Moghaddam, "Hybrid stacked ensemble combined with genetic algorithms for diabetes prediction," *Iran Journal of Computer Science*, vol. 5, no. 2, pp. 205–220, Mar. 2022. doi: 10.1007/s42044-022-00100-1.

[10] J. C. Hansen, "Human mitotic chromosome structure: What happened to the 30-nm fibre?" *The EMBO Journal*, vol. 31, no. 7, pp. 1621–1623, Mar. 2012. doi: 10.1038/emboj.2012.66.

[11] J. M. Mullaney, R. E. Mills, W. S. Pittard, and S. E. Devine, "Small insertions and deletions (INDELs) in human genomes," *Human Molecular Genetics*, vol. 19, no. Review Issue 2, pp. R131–R136, Sep. 2010. doi: 10.1093/hmg/ddq400.

[12] R. A. W. Wiberg, O. E. Gaggiotti, M. B. Morrissey, and M. G. Ritchie, "Identifying consistent allele frequency differences in studies of stratified populations," *Methods in Ecology and Evolution*, vol. 8, no. 10, pp. 1899–1909, Oct. 2017. doi: 10.1111/2041-210X.12810.

[13] F. S. Roberts and B. Tesman, *Applied Combinatorics*, 3rd ed. Boca Raton, FL: CRC Press, 2009.

[14] A. A. Komar, *Single Nucleotide Polymorphisms: Methods and Protocols*, Humana Press, 2009.

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, 27 Desember 2024

Indah Novita Tangdililing 13523047