On Non-deterministic Problem (NP): Finding a Relation Between Sudoku and DNA Sequencing With Graph Coloring

Naufarrel Zhafif Abhista - 13523149
Informatics Major
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523149@std.stei.itb.ac.id
naufarrelzhafifabhista@gmail.com

***Abstract*—The class of Non-deterministic Polynomial-time (NP) problems represents a major frontier in computer science, encompassing a vast number of computationally intensive challenges found in diverse real-world domains. This paper explores the application of a classic NP-complete problem, Graph Coloring, as a versatile tool for understanding and addressing two such challenges: the logic puzzle Sudoku and the bioinformatics problem of DNA sequencing. We demonstrate how Sudoku can be directly transformed into a graph coloring problem, where the puzzle's constraints are perfectly mirrored by the graph's structure, allowing for a solution via algorithms like backtracking. In contrast, for the immense and complex task of DNA fragment assembly, we show that while a direct coloring approach is infeasible due to scale and complexity, the concept of "coloring" serves as a powerful conceptual model. By comparing these two applications, this paper highlights the dual role of graph coloring as both a direct solving mechanism and an abstract modeling tool.**

***Index Terms*—NP-Complete, Graph Coloring, Sudoku, DNA Sequencing, Fragment Assembly, Backtracking, Heuristic, Greedy.**

I. Introduction

A. Background

In the landscape of modern computation, problems are often categorized by the resources required to solve them, with time being the most critical constraint. This has given rise to the field of computational complexity theory, which seeks to classify computational problems according to their inherent difficulty. A computational problem can be understood as a question that a computer can answer, often framed as a decision problem with a "yes" or "no" ("accept" or "reject") output [1].

At the core of this classification lies the distinction between problems that are considered "easy" and those that are "hard." [2] The class P, for Polynomial time, encompasses all decision problems that can be solved by a deterministic algorithm in a duration that is a polynomial function of the size of the input. These problems are deemed computationally "tractable" or "efficiently solvable". Examples include sorting a list or finding the shortest path between two points in a network. In contrast, the class NP, for Non-deterministic Polynomial time, includes decision problems for which a proposed solution, or "certificate," can be verified for correctness in polynomial time. The crucial distinction is between solving a problem and verifying a solution. For instance, finding the prime factors of a very large number is believed to be difficult (not in P), but verifying that a given set of factors is correct is easy (in NP)—one simply has to multiply them together.

B. Problem Statement

This paper investigates two problems from vastly different domains: the popular logic puzzle Sudoku and the fundamental bioinformatics challenge of DNA Fragment Assembly. Sudoku is a well-defined and discrete puzzle with a finite set of rules and a constrained solution space. The objective is to fill a grid with numbers such that each row, column, and sub-grid contains each number exactly once [3]. While the rules are simple, finding a solution can be a non-trivial combinatorial task.

DNA fragment assembly, on the other hand, is a large-scale, data-intensive problem central to modern genomics. High-throughput sequencing technologies cannot read a whole genome at once; instead, they produce millions or billions of short, overlapping DNA fragments called "reads". The computational task is to reassemble these reads into the original, long genomic sequence. This process is complicated by sequencing errors and, most critically, by the presence of repetitive sequences—identical or near-identical stretches of DNA that appear in multiple locations throughout the genome. These repeats create profound ambiguity, making it difficult to determine the correct path of assembly.

Despite their differences in context, scale, and application—one a recreational pastime, the other a cornerstone of biological research—this paper posits that both Sudoku and DNA fragment assembly can be understood and analyzed through the unifying lens of a single, powerful abstraction from theoretical computer science: the Graph Coloring problem.

C. Objectives

The primary objectives of this paper are threefold, aiming to bridge the gap between abstract theory and practical application:

1) To formally define the class of NP-complete problems and establish Graph Coloring as a canonical member of this class, providing a robust theoretical foundation for the subsequent analysis.
2) To demonstrate the direct and literal transformation of a Sudoku puzzle into a Graph Coloring problem, where a valid coloring of the corresponding graph is equivalent to a solution of the puzzle.
3) To explore the more conceptual and metaphorical application of graph coloring principles to model and resolve ambiguities in DNA fragment assembly, particularly those arising from genomic repeats.
4) To synthesize these two case studies, drawing nuanced conclusions about how the inherent computational complexity and, crucially, the scale of a problem dictate the choice between exact, brute-force algorithms and heuristic or problem-remodeling approaches.

By examining these two cases, this paper will illustrate the versatile power of NP-complete problems not only as benchmarks for computational hardness but also as flexible modeling tools for understanding and tackling complex challenges across scientific and recreational domains.

II. Theoritical Foundation

A. P, NP, and NP-Completeness

To understand the connection between Sudoku and DNA sequencing, it is essential to first establish a formal framework for computational complexity. This framework is built upon a hierarchy of classes that categorize problems based on their difficulty.

- P (Polynomial Time): This class contains decision problems that can be solved by a deterministic Turing machine in a time that is a polynomial function of the input size, denoted as $O(n^k)$ for some constant k. These are problems for which efficient algorithms are known to exist.
- NP (Non-deterministic Polynomial Time): This class contains decision problems for which, if the answer is "yes," there exists a proof or "certificate" that can be verified in polynomial time by a deterministic Turing machine. For example, the problem "Does this graph have a Hamiltonian path?" is in NP because if such a path is given (the certificate), one can easily check in polynomial time that it visits every vertex exactly once. All problems in P are also in NP, since if a problem can be solved quickly, its solution can certainly be verified quickly. The central question remains whether $P = NP$
- NP-hard: A problem is classified as NP-hard if every problem in NP can be reduced to it in polynomial time. A reduction is a transformation of an instance of one problem into an instance of another. If problem A reduces to problem B, a solution for B can be used to solve A. Thus, NP-hard problems are at least as difficult as the hardest problems in NP. An NP-hard problem is not necessarily in NP itself; it could be even harder, for instance, a problem for which a solution cannot be verified in polynomial time.
- NP-complete (NPC): A problem is NP-complete if it satisfies two conditions: (1) it is in NP, and (2) it is NP-hard. These problems represent the most difficult problems within the NP class. The discovery of a polynomial-time algorithm for any single NP-complete problem would automatically provide a polynomial-time algorithm for every problem in NP, thereby proving that P=NP.

The existence of NP-complete problems was not obvious until the Cook-Levin theorem (1971) [5], which proved that the Boolean Satisfiability Problem (SAT) is NP-complete. SAT asks whether there exists an assignment of truth values (True/False) to variables in a given Boolean formula that makes the entire formula evaluate to True. The Cook-Levin theorem established SAT as the first NP-complete problem, providing a foundational benchmark from which the NP-completeness of thousands of other problems could be proven through polynomial-time reductions.

B. Graph Coloring

Among the most studied NP-complete problems is the Graph Coloring problem, whose simple statement belies its profound computational complexity.

A graph G is a mathematical structure consisting of a set of vertices V (or nodes) and a set of edges E that connect pairs of vertices. A proper vertex coloring is an assignment of a label, or "color," to each vertex of the graph such that no two adjacent vertices (i.e., vertices connected by an edge) receive the same color. A graph is said to be k-colorable if it can be properly colored with at most k colors. The minimum number of colors required to properly color a graph G is known as its chromatic number, denoted $\chi(G)$. The k-Coloring decision problem asks: given a graph G and an integer k, is G k-colorable? The complexity of this problem famously depends on the value of k.

- 2-Coloring: Determining if a graph can be colored with just two colors is in P. This is equivalent to checking if the graph is bipartite (i.e., its vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to one in the other), which can be done efficiently using algorithms like Breadth-First Search.
- 3-Coloring: In stark contrast, the 3-Coloring problem is NP-complete [6]. This sharp transition illustrates how a seemingly minor change to a problem's constraints can push it across the boundary from tractable to intractable.

III. Methodology

The NP-complete nature of graph coloring implies that no known algorithm can find an optimal coloring for all graphs in polynomial time. This reality forces a choice between two distinct paths: exact algorithms that guarantee a correct solution

but may require exponential time, and heuristic algorithms that run quickly but may produce suboptimal results. The selection of an approach is dictated by the specific requirements of the problem at hand, particularly its scale and the necessity of an exact versus an approximate solution.

A. Backtracking

When an exact solution is non-negotiable and the problem size is manageable, backtracking serves as a fundamental algorithmic technique. It is a refined form of brute-force search that systematically explores the space of all possible solutions. Rather than generating every possible configuration and then testing it, backtracking builds a solution incrementally and abandons a path [4] as soon as it determines that it cannot lead to a valid solution.

The algorithm operates recursively, akin to a depth-first search through a state-space tree where each node represents a partial solution. For the graph coloring problem, the process is as follows:

1) Begin with an uncolored vertex, typically the first in a predefined order (e.g., vertex 0).
2) Iterate through the available colors (e.g., 1 to m). For the current color c, assign it to the vertex.
3) Verify if this assignment is "safe" by checking that no adjacent, already-colored vertices have the color c. This is often implemented in a helper function, is_safe().
4) If the assignment is safe, recursively call the backtracking function for the next uncolored vertex.
5) If the recursive call returns false (meaning it could not find a valid coloring for the rest of the graph from that point), or if no safe color can be found for the current vertex, the algorithm "backtracks." It undoes the current color assignment and tries the next available color in its loop. If all colors have been tried unsuccessfully, the function returns false to its caller, propagating the failure up the recursion stack
6) If the function is called for a vertex beyond the last one (i.e., all vertices have been successfully colored), a solution has been found, and the algorithm terminates, returning true

The key to backtracking's relative efficiency over naive brute force is pruning. By abandoning a search path the moment a constraint is violated, it avoids exploring entire subtrees of invalid solutions. Nonetheless, in the worst-case scenario, the algorithm may still need to explore a significant portion of the solution space. Its time complexity is exponential, typically expressed as $O(m^V)$, where m is the number of colors and V is the number of vertices. The space complexity is determined by the depth of the recursion stack, which is $O(V)$. This makes backtracking suitable for problems like Sudoku, where an exact solution is required and the number of vertices is small and fixed.

B. Heuristics for Feasible Approximations (Greedy)

For large-scale problems where exponential-time algorithms are computationally infeasible, the focus shifts from finding optimal solutions to finding "good enough" solutions quickly. This is the domain of heuristic and approximation algorithms. For graph coloring, the most well-known heuristic is the Greedy Coloring algorithm, also known as sequential coloring.

The mechanism of the greedy algorithm is simple:

1) First, establish a specific sequence or ordering of all vertices in the graph,
2) Process the vertices one by one according to this order. For each vertex v_i, assign it the smallest-indexed color (e.g., the smallest positive integer) that has not already been used by any of its neighbors that appear earlier in the sequence $(v_1, ..., v_{i-1})$.

This algorithm is computationally efficient, running in linear time, $O(V + E)$, where E is the number of edges. However, its performance is critically dependent on the initial vertex ordering. A well-chosen order can lead to an optimal coloring (using $\chi(G)$ colors), while a poor ordering can result in a coloring that uses a number of colors proportional to the number of vertices, which can be far from optimal. Finding the absolute best ordering is, unfortunately, an NP-hard problem in itself.

To address this limitation, various ordering heuristics have been developed to improve the greedy algorithm's performance. One of the most effective is the Smallest Last Ordering, also known as degeneracy ordering. This strategy works by finding the vertex with the lowest degree, removing it from the graph, and adding it to the end of the order. This process is repeated on the remaining subgraph until all vertices are ordered. The degeneracy of a graph, denoted d, is the maximum degree of any vertex at the moment it is removed. When the greedy algorithm is applied to a degeneracy ordering, it is guaranteed to use at most $d + 1$ colors. This provides a valuable and efficiently computable upper bound on the chromatic number, making it a powerful heuristic for large graphs where optimality is secondary to speed and resource constraints.

The choice between an exact algorithm like backtracking and a heuristic like greedy coloring is therefore not merely a matter of performance. It is a strategic decision rooted in the fundamental goals of the problem. A puzzle like Sudoku has a binary standard of success: the solution is either entirely correct or it is useless. This demands an exact algorithm. In contrast, many optimization problems modeled by graph coloring, such as scheduling or resource allocation, exist on a spectrum of quality. A schedule using one extra time slot but generated in seconds may be far more valuable than a perfect schedule that takes centuries to compute. This distinction highlights a practical dimension of NP-completeness: the nature of an acceptable solution fundamentally shapes the algorithmic approach.

IV. Results And Discussion

The theoretical framework of graph coloring, despite its abstract origins, provides a surprisingly potent lens through which to analyze and solve problems in disparate fields. This section explores two case studies: the direct application of

graph coloring to solve the Sudoku puzzle and its more conceptual use as a modeling tool to navigate the complexities of DNA fragment assembly. The contrast between these two applications reveals how the scale and nature of a problem dictate the way we engage with its computational hardness.

A. Sudoku

The process of converting a Sudoku puzzle into a graph is methodical and precise. For a standard 9×9 grid, the mapping is as follows:

1) Vertices: Each of the 81 cells in the Sudoku grid is represented as a unique vertex in the graph. Thus, the graph has $V = 81$ vertices.
2) Edges: An edge is drawn between two vertices if their corresponding cells are constrained by the Sudoku rules. Specifically, two vertices are connected by an edge if the cells they represent are in the same row, the same column, or the same 3×3 sub-grid. This construction results in a highly regular graph where every vertex is connected to 8 other vertices in its row, 8 in its column, and 4 more in its 3×3 block (that are not already counted in the row or column), for a total degree of 20 for each vertex.
3) Colors: The integers that can be placed in the cells, typically 1 through 9, correspond to the set of available "colors".

A valid solution to the Sudoku puzzle is, therefore, equivalent to a proper 9-coloring of this graph. The rule that no two adjacent vertices can have the same color perfectly enforces all the constraints of Sudoku. A puzzle that is presented with some numbers already filled in is known as a pre-coloring extension problem. The vertices corresponding to the given numbers are "pre-colored," and the challenge is to extend this partial coloring to a complete, proper coloring of all 81 vertices using the nine available colors.

To illustrate the solution process clearly, we consider a smaller 4×4 Sudoku puzzle, called a "Shidoku." [7] The rules are analogous: fill a 4×4 grid with numbers from 1 to 4 such that each row, column, and 2×2 sub-grid contains each number exactly once.

1) Graph Modelling: A 4×4 grid is modeled as a graph with 16 vertices, one for each cell. Let's label the cells by their coordinates (r, c), from (0,0) to (3,3). An edge exists between (r_1, c_1) and (r_2, c_2) if $r_1 = r_2, c_1 = c_2$, or $\lfloor r_1/2 \rfloor = \lfloor r_2/2 \rfloor$ and $\lfloor c_1/2 \rfloor = \lfloor c_2/2 \rfloor$ The available colors are the set $\{1,2,3,4\}$.

Consider the following Shidoku puzzle,

1			4
	2	3	
	1		2
4			3

2) Pre-coloring: The given numbers translate to a pre-colored graph. The vertices corresponding to the cells are assigned fixed colors:

- Vertex (0,0) is colored '1'.
- Vertex (0,3) is colored '4'.
- Vertex (1,1) is colored '2'.
- Vertex (1,2) is colored '3'.

And so on for the other given clues.

3) Solving with Backtracking: The backtracking algorithm is now applied to find a valid 4-coloring for the remaining uncolored vertices.

1) Start: The algorithm selects the first empty cell, say (0,1). It needs to assign a color from 1,2,3,4.
2) Attempt Color '1': It tries to color vertex (0,1) with '1'. The is_safe() function checks its neighbors. Vertex (0,0) is a neighbor (same row) and is already colored '1'. This is a conflict.
3) Attempt Color '2': The algorithm tries the next color, '2'. It checks neighbors:
   - Row neighbors: (0,0) is '1', (0,2) is empty, (0,3) is '4'. No conflict.
   - Column neighbors: (1,1) is '2'. This is a conflict.
4) Attempt Color '3': The algorithm tries '3'. It checks all neighbors of (0,1) (in its row, column, and 2×2 block). Let's assume no conflicts are found. The assignment is safe.
   - Vertex (0,1) is temporarily colored '3'.
   - The algorithm recursively calls itself for the next empty cell, say (0,2).
5) Recursive Step and Conflict: The algorithm proceeds, coloring (0,2), (1,0), etc. Let's imagine it reaches a state where it must color vertex (2,2). Its neighbors in the same row, column, and block might already be colored with 1,2,3,4. For example:
   - Row 2 has a '1' and '2'.
   - Column 2 has a '3'.
   - Block 3 (bottom-left) has a '1' and '4'.
   - The neighbors of (2,2) are colored 1,2,3,4. The algorithm tries to color (2,2) with '1', '2', '3', and '4', but is_safe() returns false for all of them.
6) Backtrack: Having exhausted all color options for (2,2), the recursive call for this vertex fails and returns false. The algorithm now backtracks to the vertex that called it, say (2,0). The call for (2,0) had previously assigned a color and moved on. Now that its recursive call has failed, it undoes its assignment and tries the next available color in its own list. If it had assigned '3', it might now try '4'. This process continues, unwinding the recursion and exploring different branches of the search tree until a complete, valid coloring is found.

This step-by-step process of assigning, checking, and backtracking guarantees that if a solution exists, it will be found. For a fixed-size puzzle like Sudoku, this exponential-time algorithm is perfectly practical, demonstrating a direct and successful application of graph coloring to solve a constrained problem.

B. DNA Fragment Assembly (DNA Sequencing)

While graph coloring provides a literal solution for Sudoku, its role in DNA fragment assembly is more abstract and heuristic. The sheer scale and inherent messiness of genomic data make a direct application of k-coloring infeasible. Instead, the concepts of graph theory and coloring are used to model the problem and guide algorithms through its most challenging aspects, particularly the resolution of repetitive sequences.

In practice, many use shotgun sequencing as a method to do a DNA sequencing [8]. Shotgun sequencing is a method which the genome is randomly broken into small DNA fragments. These fragments then are sequenced individually. The goal of shotgun sequencing is to reconstruct a genome by piecing together millions of short, overlapping reads [9]. The primary obstacle is the presence of *repeats*, which are sequences that appear multiple times in the genome. A repeat longer than a read creates ambiguity: if a read ends within a repeat, it is unclear which of the repeat's copies it belongs to, leading to multiple possible paths for assembly and a "tangled" graph.

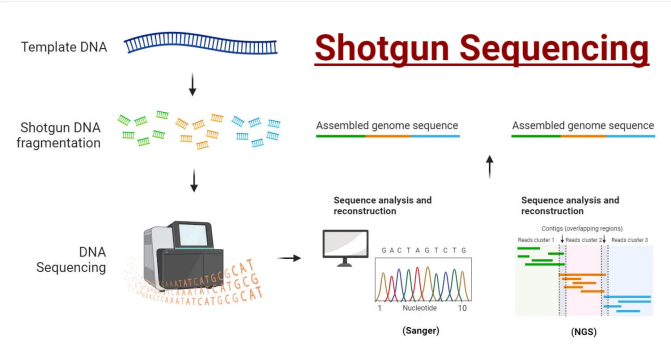

Fig. 1. A Process Of Shotgun Sequencing [8]

The evolution of graph models for this problem illustrates a classic theme in computer science: when faced with an intractable problem, change the representation.

- **Overlap Graph and the Hamiltonian Path:** The most intuitive model represents each DNA read as a vertex. An edge connects two vertices if their corresponding reads overlap significantly. In this overlap graph, the task of reassembling the genome is equivalent to finding a path that visits every vertex (read) exactly once. This is the Hamiltonian Path Problem, which is famously NP-complete. For a genome with millions of reads, this approach is computationally prohibitive.
- **De Bruijn Graph and the Eulerian Path:** A more sophisticated approach breaks reads into smaller, overlapping substrings of a fixed length k, known as k-mers. A de Bruijn graph is then constructed where vertices represent $(k-1)$-mers and a directed edge is drawn between two vertices if they form an observed k-mer. In this representation, repeats are collapsed into single paths or cycles. The assembly problem is transformed into finding a path that traverses every edge exactly once—the Eulerian Path Problem [10]. Unlike the Hamiltonian Path Problem, the Eulerian Path Problem is solvable in linear time, a monumental leap in efficiency. This reformulation cleverly sidesteps the core NP-hard challenge.

Even with the efficient de Bruijn graph model, ambiguity persists. Branching points in the graph, often caused by repeats, represent decision points where the assembly path is uncertain. This is where the concept of "coloring" becomes a powerful heuristic tool. In this context, coloring is not about satisfying local adjacency constraints but about annotating the graph with external, long-range information to resolve local ambiguities.

The "colors" are metadata labels derived from advanced sequencing techniques:

- **Sample of Origin**: In metagenomics (sequencing a community of organisms) or pangenomics (sequencing many individuals of a species), reads from different sources can be assigned distinct "colors." When traversing the graph, if a path enters a complex region using a "blue" edge, the assembler knows to prefer an exit path that is also "blue," helping to separate the genomes of different organisms or individuals [11].
- **DNA Strand Orientation**: DNA is double-stranded, and reads can originate from either the forward or reverse-complement strand. Some graph models use two "colors" to label nodes or edges based on their strand orientation [12]. This helps resolve ambiguities that arise when a sequence overlaps with the reverse complement of another.
- **Linked-Reads and Mate-Pairs**: This is arguably the most powerful application of the coloring concept. Technologies like 10x Genomics or TELL-Seq partition long DNA molecules into droplets and attach a unique molecular barcode (the "color") to all the short reads generated from that single long molecule [13]. If reads with the same barcode are found on two graph edges that are far apart in the graph, it provides strong evidence that these two regions were physically close in the original genome. This long-range information acts as a scaffold, allowing the assembly algorithm to confidently "jump" across a repeat-induced tangle. For example, if an edge entering a repeat is colored "green" and an edge exiting the repeat is also "green," the assembler can infer a direct path, effectively resolving the repeat for that specific instance.

In this paradigm, coloring is a mechanism for integrating orthogonal datasets to disambiguate a graph structure, transforming an under-determined problem into a solvable one.

C. Synthesis

Comparing the application of graph coloring to Sudoku and DNA sequencing reveals a fascinating dichotomy in how a single theoretical concept can be deployed. The differences are driven primarily by the problem's scale, goals, and the nature of its inherent complexity.

In the case of Sudoku, graph coloring is a **direct solution** mechanism. The puzzle is transformed into a canonical graph

coloring problem, an exact algorithm (backtracking) is applied, and the resulting coloring is translated back into a puzzle solution. The problem is NP-complete, but its small, fixed input size ($V = 81$) makes an exponential-time algorithm computationally tractable. The goal is absolute correctness, and an approximate solution is meaningless. Here, NP-completeness is a formal property that is confronted head-on.

In DNA sequencing, graph coloring is a **conceptual modeling** heuristic. The initial, intuitive formulation of the problem as a Hamiltonian Path is NP-complete, but its immense scale (millions of vertices) renders this approach impossible. The scientific community's response was not to build a faster computer but to re-frame the problem into the polynomially solvable Eulerian Path problem using de Bruijn graphs. The concept of "coloring" is then reintroduced not to solve the core problem, but to guide the path-finding algorithm through the remaining points of ambiguity (repeats). The "colors" are not abstract labels to be minimized but are carriers of crucial external data (like barcodes) that provide long-range constraints. The goal is not to find an optimal coloring but to reconstruct the most plausible biological sequence.

This contrast highlights a profound principle in applied computer science: theoretical hardness is a practical barrier whose height depends on the instance size. For small instances, we can afford the computational cost of exactness. For massive instances, we must innovate, either by finding clever reformulations that sidestep the intractability or by using heuristics to guide us to high-quality, albeit not provably optimal, solutions.

V. Conclusion

A. Summary

This paper has explored the multifaceted role of the NP-complete Graph Coloring problem as a unifying framework for understanding two fundamentally different challenges: the recreational logic puzzle of Sudoku and the large-scale bioinformatics problem of DNA fragment assembly. The analysis reveals that the utility of a theoretical concept like graph coloring is not monolithic but is instead highly contingent on the context, scale, and objectives of the application.

For Sudoku, the relationship is direct and literal. The puzzle's constraints map perfectly onto the definition of a proper vertex coloring, allowing the problem to be solved exactly using algorithms designed for this NP-complete task. The small and fixed scale of the puzzle renders an exponential-time algorithm like backtracking computationally feasible, making it a classic example of tackling an NP-complete problem head-on.

For DNA fragment assembly, the relationship is more conceptual and metaphorical. The immense scale of genomic data makes a direct assault on its NP-hard formulation (the Hamiltonian Path problem) impossible. This intractability spurred a crucial innovation: the reformulation of the problem into the polynomially solvable Eulerian Path problem via de Bruijn graphs. Within this new framework, the concept of coloring re-emerges as a powerful heuristic for resolving ambiguities, particularly those caused by genomic repeats. Here, "colors" are not abstract labels to be minimized but are carriers of vital external information, such as linked-read barcodes, that provide long-range constraints to guide the assembly algorithm through tangled regions of the graph.

Ultimately, the connection between the two domains illustrates a tale of direct application versus conceptual modeling. This dichotomy is driven by problem scale and underscores a critical lesson in computational science: the label "NP-complete" is not a final verdict of impossibility but a signpost that directs researchers toward different strategies. For small-scale problems, it may be a challenge to be met with computational power; for large-scale problems, it is a barrier to be circumvented with algorithmic ingenuity and clever modeling.

B. Suggestions

The insights gleaned from this comparative analysis have implications that extend beyond these two specific examples. The P versus NP problem remains one of the most profound unanswered questions in science, and as datasets continue to grow in size and complexity, the need to understand and navigate the boundary of tractability will only become more acute. The strategies of problem reformulation and heuristic guidance, as seen in genomics, will continue to be essential tools for progress.

The future of genomics, in particular, is inextricably linked to graph-based models. The principles of annotating, or "coloring," a graph with external data are foundational to the emerging field of pangenomics. Instead of a single linear reference genome, a pangenome graph can represent the complete genetic diversity of an entire species or population. In these graphs, nodes represent shared sequences, branches represent variations, and "colors" can be used to annotate paths corresponding to specific individuals, populations, or traits. The conceptual tools of graph coloring, once used to solve puzzles and later to resolve repeats in a single genome, are now being scaled up to map the full spectrum of life's genetic variation. This evolution demonstrates the enduring power of abstract computational concepts to provide the language and framework for scientific revolutions.

Acknowledgement

The author would like to express their gratitude to God Almighty (Allah SWT) for the blessings of health and opportunity, without which the completion of this paper would not have been possible.

The author also wishes to extend their sincere thanks to the lecturers of the IF2211 Algorithm Strategies course, especially Mr. Monterico Adrian, S.T., M.T., for his invaluable guidance and for sharing a wealth of knowledge throughout the semester.

Furthermore, profound gratitude is extended to friends who provided support and encouragement throughout this process. A special thanks goes to Kimberly Mahdiya Khairunnisa for her constant encouragement, and to Zulfaqqar Nayaka Athadiansyah, who patiently provided guidance on formatting this paper using LaTeX.

Finally, the author hopes that this paper will be beneficial, both for their own academic development and for the wider community.

YOUTUBE LINK

Can be accessed here: https://youtu.be/7sS4clvjlh8

REFERENCES

[1] Munir, R. (2020). *Teori P, NP, dan NPC (Bagian 1)* [Theory of P, NP, and NPC (Part 1)]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/Teori-P-NP-dan-NPC-(Bagian%201).pdf

[2] Munir, R. (2020). *Teori P, NP, dan NPC (Bagian 2)* [Theory of P, NP, and NPC (Part 2)]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/Teori-P-NP-dan-NPC-(Bagian%202).pdf

[3] Mulroy, C. (2022, August 12). What is Sudoku? How to solve the puzzle and where to play. USA Today. https://www.usatoday.com/story/life/2022/08/12/what-is-sudoku-solve-puzzle/10299742002/

[4] Munir, R. (2020). *Algoritma runut-balik (Backtracking) (Bagian 1)* [Backtracking Algorithm (Part 1)]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/15-Algoritma-backtracking-(2025)-Bagian1.pdf

[5] Singla, A. (2024, May 23). Cook-Levin theorem or Cook's theorem. GeeksforGeeks. https://www.geeksforgeeks.org/cook-levin-theorem-or-cooks-theorem/

[6] GeeksforGeeks. (2024, May 21). 3-coloring is NP-complete. https://www.geeksforgeeks.org/dsa/3-coloring-is-np-complete/

[7] Mastering Sudoku. (n.d.). Shidoku. Accessed on June 25th, 2025, from https://masteringsudoku.com/shidoku/

[8] National Human Genome Research Institute. (2023, August 2). Shotgun sequencing. https://www.genome.gov/genetics-glossary/Shotgun-Sequencing

[9] Quiroz-Ibarra, J. E., Mallén-Fullerton, G. M., & Fernández-Anaya, G. (2017). DNA paired fragment assembly using graph theory. Algorithms, 10(2), 36. https://doi.org/10.3390/a10020036

[10] Pevzner, P. A., Tang, H., & Waterman, M. S. (2001). An Eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences of the United States of America, 98(17), 9748–9753. https://doi.org/10.1073/pnas.171285098

[11] Mustafa, H., Schilken, I., Karasikov, M., Eickhoff, C., Rätsch, G., & Kahles, A. (2019). Dynamic compression schemes for graph coloring. Bioinformatics, 35(3), 407–414. https://doi.org/10.1093/bioinformatics/bty632

[12] Kellis, M., Korf, I., & Edwards, S. (n.d.). 5.3: Genome assembly II-String graph methods. In Computational biology - Genomes, networks, and evolution. LibreTexts. Accessed on June 25th, 2025, dari https://bio.libretexts.org/Bookshelves/Computational_Biology/Book%3A_Computational_Biology_-Genomes_Networks_and_Evolution(Kellis_et_al.)/05%3A_Genome_Assembly_and_Whole-Genome_Alignment/5.03%3A_Genome_Assembly_II-_String_graph_methods

[13] Tolstoganov, I., Pevzner, P. A., & Korobeynikov, A. (2024). SpLitteR: Diploid genome assembly using linked TELL-Seq reads and assembly graphs. PeerJ, 12, e18050. https://doi.org/10.7717/peerj.18050

STATEMENT

I hereby declare that this paper is my original work, and has not been adapted, translated, or plagiarized from any other source.

Naufarrel Zhafif Abhista
13523149