Computing Chromatic Polynomial via Dynamic Programming and Lagrange Interpolation

Jonathan Kris Wicaksono - 13524023
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: jonathankw2005@gmail.com, 13524023@std.stei.itb.ac.id

***Abstract*—Computing the chromatic polynomial $P(G, k)$ asks how many proper ways a graph G can be colored using a palette of k colors. This problem is important in graph theory and combinatorics, but it is computationally difficult because direct enumeration belongs to the family of #P-hard counting problems. This paper proposes an algorithmic strategy that combines Dynamic Programming over vertex subsets with Lagrange Interpolation. The DP step counts the number of valid colorings for the required integer color values by treating each color class as an independent set, while the interpolation step reconstructs the full polynomial from those values.**

***Keywords*—Chromatic Polynomial, Dynamic Programming, Lagrange Interpolation, Graph Theory, Algorithm Complexity**

I. INTRODUCTION

Graph (vertex) coloring problem is a classic and fundamental problem in graph theory, computer science, and combinatorics. The problem is straightforward: given a graph, how can we assign colors to its vertices so that (a) no two adjacent vertices share the same color, and (b) we use as few colors as possible? This search for a single valid assignment has been discussed in depth in my previous comparative study of well-known algorithms [1]. However, in this paper, we will move from a decision problem of finding only one single assignment to that of a more complex problem: given a palette of k colors, how many distinct ways are there to properly color a graph G. This number is known as the chromatic polynomial of G with k colors and is denoted by $P(G, k)$ which will be discussed thoroughly in this paper.

But why do we care about $P(G, k)$ in the first place? Well, it has some well-known applications in real word beyond just theoretical combinatorics. From [2] and [3], the chromatic polynomial is useful for allocation of channels to television stations, construction of timetables, and the field of statistical mechanics (potts model). Not only that, understanding $P(G, k)$ of a graph will reveal structural property of that particular graph.

Despite it's many usage and applications, unfortunately computing the chromatic polynomial is notoriously difficult. The task itself was born in the *#P-hard* complexity class [4]. If $n = |V|$ is the number of vertices, then a naive brute-force enumeration has to test up to k^n assignments, which is computationally intractable.

This paper focuses on improving that computation by combining subset-based dynamic programming and Lagrange Interpolation. The dynamic programming part counts the number of valid colorings for enough integer values of k, and the interpolation part reconstructs the polynomial from those exact values. In this way, we can avoid enumerating every complete coloring assignment directly.

This paper is organized as follows. Section II explains the mathematical foundation of chromatic polynomials, independent sets, subset representation, Lagrange Interpolation, and why DP and interpolation are combined. Section III describes the subset DP algorithm, independent-set precomputation, and the resulting time and space complexity. Section IV reports the experimental setup and measured comparison against naive backtracking. Finally, Section V concludes the paper and discusses the main limitation of the approach.

II. THEORETICAL FOUNDATION

A. Chromatic Polynomials

Let $G = (V, E)$ be an undirected graph with $n = |V|$. A proper k-coloring of G is a function

$$f : V \to \{1, 2, \ldots, k\}$$

such that $f(u) \neq f(v)$ for every edge $(u, v) \in E$. The chromatic polynomial $P(G, k)$ is defined as the number of such proper colorings. In other words,

$$P(G, k) = \big|\{f : V \to \{1, \ldots, k\} \mid \text{for every } (u, v) \in E, \\ f(u) \neq f(v)\}\big|.$$

Although it is written as a counting function, $P(G, k)$ is a polynomial in k with degree n [2]. This is the important property that makes interpolation possible later. Unfortunately, computing the value of this polynomial is still hard. A direct brute force method tries all k^n assignments and checks whether each assignment is valid. This grows too quickly even for a graph that still looks small in a drawing. More formally, computing graph polynomial evaluations is closely related to *#P-hard* counting problems [4].

Some examples make the meaning of the polynomial easier to see. If G has no edges, then every vertex can choose any of the k colors independently, so $P(G, k) = k^n$. If G is a complete graph K_n, then every vertex must use a different color, so

$$P(K_n, k) = k(k-1)(k-2)\cdots(k-n+1).$$

These two examples also show the range of difficulty. An empty graph is trivial because every assignment is valid, while a complete graph is also easy because the structure is very strict. The difficult cases are usually in the middle, where some assignments are valid and many others are not.

A classical way to compute chromatic polynomials is the deletion-contraction recurrence [7]. For an edge e, let $G - e$ be the graph after deleting e, and let G/e be the graph after contracting e. Then

$$P(G, k) = P(G - e, k) - P(G/e, k).$$

This identity is elegant, but a direct recursive implementation can still branch heavily. The approach in this paper takes a different direction. Instead of recursively changing the graph, it keeps the graph fixed and works over subsets of its vertex set.

B. Labeled Colors and Unused Colors

One subtle but important detail is that $P(G, k)$ counts colorings using a palette of k labeled colors, not necessarily colorings that use all k colors. For example, if the available colors are red, blue, and green, a valid coloring that only uses red and blue is still counted by $P(G, 3)$. This is why the DP recurrence later allows the color class I to be empty. If the c-th color is unused, then no vertex is assigned to that color, but the coloring is still a valid coloring using the first c colors.

This also distinguishes the chromatic polynomial from the chromatic number $\chi(G)$. The chromatic number asks for the smallest number of colors needed to color G, while the chromatic polynomial asks how many valid colorings exist for every palette size k. These two concepts are related because $P(G, k) = 0$ for positive integers $k < \chi(G)$, but they answer different questions. In this paper, the target is the counting problem, not only the minimum-color decision problem.

C. Useful Polynomial Properties

Several known properties are useful for checking whether a computed polynomial makes sense [2]. First, for a graph with n vertices, $P(G, k)$ has degree n and leading coefficient 1. This matches the intuition that, for very large k, almost all of the k^n assignments are valid, so the highest-degree term is the same as the empty graph.

Second, if G has at least one vertex, then $P(G, 0) = 0$. There is no way to color a nonempty graph with zero available colors. Third, if G has at least one edge, then $P(G, 1) = 0$, because the two endpoints of that edge cannot receive the same single color.

D. Independent Sets and Subset Representation

A set of vertices $I \subseteq V$ is called an independent set if there is no edge whose two endpoints are both in I. This idea connects naturally with graph coloring. In a proper coloring, every color class must be an independent set, because two adjacent vertices are forbidden from having the same color. Therefore, a proper coloring can be viewed as splitting the vertex set into labeled independent sets, where each label is one available color. Some color labels may be unused, and that is still counted by $P(G, k)$.

To make this operation efficient in a program, every subset of vertices can be represented by a bitmask. If the vertices are ordered as $v_0, v_1, \ldots, v_{n-1}$, then the i-th bit of an integer represents whether v_i is inside the subset. For example, the mask 1011_2 represents the subset $\{v_0, v_1, v_3\}$. This representation makes subset iteration and set difference fast, because the program can use bitwise operations instead of repeatedly constructing ordinary sets.

The graph itself can also be stored with bitmasks. For each vertex v_i, store a mask A_i containing all neighbors of v_i. A subset S is independent if no vertex inside S has another neighbor also inside S. In bitmask terms, this can be checked by verifying that $A_i \& S = 0$ for each selected vertex v_i after removing v_i itself from the set.

E. Lagrange Interpolation

Lagrange Interpolation is a method for reconstructing a polynomial from known values at distinct input points [5]. The explanation in this subsection follows the construction in my online note [6], but is adapted to the chromatic polynomial setting. The general problem is the following, given m points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m),$$

where all x_i values are distinct, construct a polynomial $f(x)$ such that

$$f(x_1) = y_1, \quad f(x_2) = y_2, \quad \ldots, \quad f(x_m) = y_m.$$

There is a unique polynomial of degree at most $m - 1$ satisfying these conditions. Instead of directly solving for all coefficients of $f(x)$, the Lagrange idea is to construct $f(x)$ by building smaller polynomials

$$f_1(x), f_2(x), \ldots, f_m(x).$$

Each $f_i(x)$ is responsible for exactly one point. More precisely, we want

$$f_i(x_j) = \begin{cases} y_i, & x_j = x_i, \\ 0, & x_j \neq x_i. \end{cases}$$

If we can construct all of these f_i polynomials, then the final polynomial is simply

$$f(x) = \sum_{i=1}^{m} f_i(x).$$

This works because, when $x = x_j$, all terms are zero except $f_j(x_j)$, so

$$f(x_j) = 0 + \cdots + 0 + f_j(x_j) + 0 + \cdots + 0 = y_j.$$

Now we only need to solve for each $f_i(x)$. The definition above says that every point x_j with $j \neq i$ must be a root of $f_i(x)$. Therefore, $f_i(x)$ must contain all factors $(x - x_j)$ except the factor $(x - x_i)$:

$$(x - x_1)(x - x_2) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_m).$$

In product notation, the same expression is

$$\prod_{\substack{1\le j\le m\\ j\neq i}} (x - x_j).$$

This already gives zero at every other input point, but it does not necessarily give y_i when $x = x_i$. If we substitute x_i, the value becomes

$$\prod_{\substack{1\le j\le m\\ j\neq i}} (x_i - x_j).$$

So the product must be scaled. To make the value at x_i equal to y_i, multiply it by

$$\frac{y_i}{\prod\limits_{\substack{1\le j\le m\\ j\neq i}} (x_i - x_j)}.$$

Thus,

$$\begin{aligned} f_i(x) &= \left(\prod_{\substack{1\le j\le m\\ j\neq i}} (x - x_j) \right) \frac{y_i}{\prod\limits_{\substack{1\le j\le m\\ j\neq i}} (x_i - x_j)} \\ &= y_i \prod_{\substack{1\le j\le m\\ j\neq i}} \frac{x - x_j}{x_i - x_j}. \end{aligned}$$

Finally, because $f(x)$ is the sum of all $f_i(x)$, we obtain the Lagrange Interpolation formula

$$f(x) = \sum_{i=1}^{m} y_i \prod_{\substack{1\le j\le m\\ j\neq i}} \frac{x - x_j}{x_i - x_j}.$$

This is the core mathematical trick used in this paper. Since $P(G, k)$ is a polynomial of degree n, it is uniquely determined by $n + 1$ values. So, if the algorithm can compute

$$P(G, 1), P(G, 2), \ldots, P(G, n + 1),$$

then the full algebraic form of the chromatic polynomial can be reconstructed. In this way, the program does not need to manipulate the symbolic polynomial during the dynamic programming phase. It only needs exact integer counts at enough color values, and the symbolic reconstruction is postponed to the interpolation step.

The input points used in this paper are consecutive positive integers

$$(1, P(G, 1)), (2, P(G, 2)), \ldots, (n + 1, P(G, n + 1)).$$

This is convenient because each x_i has a direct meaning as a number of available colors. In implementation problems with consecutive points, products in the numerator and denominator can often be organized with prefix and suffix products. In this paper, the more important benefit is simpler, the program can ask for $P(G, 1)$ through $P(G, n + 1)$ and then reconstruct the polynomial exactly. A direct evaluation of the formula takes $\mathcal{O}(m^2)$ product work for m points, but this cost is small compared with the exponential subset DP used to obtain the values.

F. Why Combine DP and Lagrange Interpolation

The combination of DP and Lagrange Interpolation is useful because each part solves a different side of the problem. The dynamic programming algorithm is good at answering a numerical question: for this fixed k, how many valid k-colorings does the graph have? It does this by adding one labeled color class at a time, where each color class must be an independent set. However, DP by itself only gives values such as $P(G, 3)$ or $P(G, 4)$; it does not directly give the symbolic polynomial.

Lagrange Interpolation fills that gap. Since the chromatic polynomial has degree n, the algorithm does not need to discover its coefficients directly. It only needs enough exact evaluations. After DP computes $n + 1$ values, interpolation turns those values into the polynomial coefficients. So the computation is split into two cleaner jobs: DP handles the combinatorial counting, and interpolation handles the algebraic reconstruction.

This is better than enumerating all color assignments because the DP groups many assignments together by subset state. It is also simpler than doing symbolic polynomial operations during every DP transition. The tradeoff is still exponential memory, because the DP table has one entry for each subset of vertices. Therefore, the method does not make the hard problem easy for all graph sizes, but it makes the computation much more practical for the small and medium graphs tested in this paper.

G. Small Worked Example

As a small example, consider the path graph P_3 with vertices v_0, v_1, v_2 and edges (v_0, v_1) and (v_1, v_2). The independent subsets are

$$\emptyset, \{v_0\}, \{v_1\}, \{v_2\}, \{v_0, v_2\}.$$

The subset $\{v_0, v_2\}$ is allowed because the two endpoints are not adjacent. However, $\{v_0, v_1\}$ and $\{v_1, v_2\}$ are not independent sets because each contains an edge.

For this graph, the chromatic polynomial can be reasoned manually. The middle vertex v_1 can choose any of the k colors. Then v_0 has $k-1$ choices because it must be different from v_1, and v_2 also has $k - 1$ choices. The two endpoints are allowed to have the same color. Therefore,

$$P(P_3, k) = k(k - 1)^2 = k^3 - 2k^2 + k.$$

The DP and interpolation approach also reaches the same result in a more mechanical way. Since the graph has $n = 3$ vertices, four values are enough:

$$\begin{aligned} P(P_3, 1) = 0, \quad P(P_3, 2) = 2, \\ P(P_3, 3) = 12, \quad P(P_3, 4) = 36. \end{aligned}$$

Lagrange Interpolation on these four points reconstructs the polynomial $k^3 - 2k^2 + k$.

III. ALGORITHM DESIGN AND METHODOLOGY

A. The Dynamic Programming Strategy

The dynamic programming part counts valid colorings for a fixed number of colors. Let $\mathcal{I}$ be the family of all independent subsets of V. For every subset $S \subseteq V$ and every integer $c \geq 0$, define

$$F_c(S) = \#\{\text{proper colorings of } S \text{ using colors } 1, \ldots, c\}.$$

The base case is

$$F_0(\emptyset) = 1, \qquad F_0(S) = 0 \text{ for } S \neq \emptyset.$$

For the transition, choose which vertices receive the c-th color. That chosen set must be independent. Therefore, for $c \geq 1$,

$$F_c(S) = \sum_{\substack{I \subseteq S \\ I \in \mathcal{I}}} F_{c-1}(S \setminus I).$$

The empty set is allowed as I, because a color may be unused. After this recurrence is evaluated, the number we need is simply

$$P(G, k) = F_k(V).$$

Algorithm 1 Subset DP for $P(G, k)$

```
Precompute all independent subsets of V
dp[∅] ← 1 and dp[S] ← 0 for all S ≠ ∅
for c ← 1 to k do
    next[S] ← 0 for every S ⊆ V
    for each independent subset I do
        for each subset R with R ∩ I = ∅ do
            next[R ∪ I] ← next[R ∪ I] + dp[R]
        end for
    end for
    dp ← next
end for
return dp[V]
```

This loop is the same recurrence written in a more practical direction. Here, R represents the vertices already colored by the previous colors, while I represents the vertices receiving the new color. The program first iterates over independent I, then over subsets disjoint from I, so it does not spend time trying color classes that already contain an edge.

B. Independent Set Precomputation

Before running the DP, the program marks every subset as either independent or not independent. With bitmasks, this can be done by taking one vertex v from a nonempty subset S and writing $R = S \setminus \{v\}$. Then S is independent if and only if R is independent and v has no neighbor inside R.

This precomputation costs $\mathcal{O}(2^n)$ time and stores one Boolean value for each subset. The advantage is that the DP transition only checks a table. This matters because the same independent subset can appear many times across different color layers and different remaining vertex sets.

C. Complexity Analysis and Interpolation

Let $n = |V|$ in this subsection. For one DP layer, the basic subset transition can be viewed as iterating through all ordered pairs (I, S) where $I \subseteq S \subseteq V$. For each vertex, there are three possibilities: it is outside S, inside S but outside I, or inside I. Thus, the number of such pairs is

$$\sum_{S \subseteq V} 2^{|S|} = 3^n.$$

So the time complexity for computing one fixed value $P(G, k)$ is $\mathcal{O}(k3^n)$ after independent-set precomputation. When the goal is to reconstruct the full polynomial, the program records $P(G, 1)$ through $P(G, n+1)$ from the same increasing color loop, so the total DP cost becomes $\mathcal{O}(n3^n)$. This is still exponential, but it improves the naive enumeration cost, where the largest required coloring count alone checks up to $(n+1)^n$ assignments.

The complete method used in this paper is therefore simple,

1. Precompute whether every vertex subset is an independent set.
2. Run the subset DP and record $P(G, i)$ for $i \in \{1, 2, \ldots, n+1\}$.
3. Apply Lagrange Interpolation to those $n+1$ points.
4. Output the coefficients of the chromatic polynomial.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experimental Setup

To test whether the algorithm is not only mathematically correct but also useful in practice, I implemented the benchmark in C++17. The benchmark source and raw measured CSV are stored in the `experiments` directory. The experiment was run on an Apple M4 CPU with 16 GB RAM, macOS 26.5.1, and Apple clang 21.0.0.

The dataset consists of deterministic random undirected graphs. For each graph size, the program uses a fixed seed, so the same graph can be regenerated and tested again. The first experiment is a size sweep with $n = 5, 8, 10, 12, 14, 15$. I used this range because it is enough to show the difference between both approaches while still allowing the naive baseline to finish or hit a controlled timeout during the experiment. The second experiment fixes $n = 12$ and changes the edge density to observe how the number of independent sets affects the DP runtime.

The proposed DP + Lagrange method is compared against a naive backtracking counter. The backtracking method recursively assigns colors to vertices and counts all valid assignments. For each graph, both methods try to obtain the values required for interpolation, namely $P(G, 1)$ through $P(G, n+1)$. The naive method is stopped when it exceeds 2000 ms for one graph, because after that point the comparison is already clear and waiting longer would not add much information.

The measurement includes the total time for DP and interpolation, the time spent only in interpolation, the approximate DP memory usage, the number of independent sets found by

precomputation, and the number of recursive nodes visited by the backtracking baseline.

B. Performance Metrics

TABLE I
GENERATED GRAPH INSTANCES

Case	n	Seed	Requested p	Actual density
size-5	5	13524023	0.40	0.400
size-8	8	13524024	0.40	0.321
size-10	10	13524025	0.45	0.356
size-12	12	13524026	0.45	0.439
size-14	14	13524027	0.50	0.593
size-15	15	13524028	0.50	0.495

Table I records the graph instances used for the size sweep. The requested probability p is the probability used when deciding whether each possible edge exists. The actual density can differ because each graph is generated from independent random choices. I include both values so that the experiment is reproducible and so that the measured result is not interpreted as coming from an idealized density that the generated graph does not actually have.

TABLE II
SIZE SWEEP: DP + LAGRANGE VERSUS NAIVE BACKTRACKING

n	$\|E\|$	Indep.	DP (ms)	Backtracking (ms)	Stop k	Memory (KB)
5	4	13	0.022	0.107	6	0.680
8	9	45	0.098	200.672	9	5.254
10	16	80	0.201	> 2000	9	21.573
12	29	99	0.803	> 2000	7	89.522
14	54	74	3.155	> 2000	7	374.325
15	52	127	8.842	> 2000	7	768.143

For the rows where backtracking timed out, the program also recorded at which color count it stopped. For example, the $n = 10$ graph stopped while computing $P(G, 9)$, while the $n = 12$, $n = 14$, and $n = 15$ size-sweep graphs stopped while computing $P(G, 7)$. For every value that backtracking did finish before timeout, the result was identical to the DP result. This is shown in the `verified_prefix` column of the raw CSV file.

TABLE III
DENSITY SWEEP FOR $n = 12$

Density	$\|E\|$	Indep.	DP (ms)	Backtracking (ms)	Stop k
0.242	16	274	1.234	> 2000	6
0.500	33	81	0.710	> 2000	8
0.773	51	32	0.502	> 2000	9

The density sweep shows why independent sets are important for this algorithm. The sparse $n = 12$ graph has 274 independent subsets and takes 1.234 ms. The dense graph has only 32 independent subsets and takes 0.502 ms. The DP table size is almost the same because 2^n depends only on the number of vertices, but the transition count becomes smaller when the graph is dense enough to forbid many color classes.

TABLE IV
DETAILED COUNTERS FROM THE BENCHMARK

Case	Interp. (ms)	Visited nodes	Status	Verified
size-5	0.018	7,356	completed	yes
size-8	0.051	25,186,659	completed	yes
size-10	0.048	225,472,512	timeout	yes
size-12	0.074	133,881,856	timeout	yes
size-14	0.096	78,630,912	timeout	yes
size-15	0.102	74,129,408	timeout	yes

The interpolation time in Table IV is very small compared with the total DP + Lagrange time. Even for $n = 15$, interpolation takes only 0.102 ms in the measured run. This supports the complexity discussion: for this experiment, the bottleneck is not reconstructing the polynomial from points, but computing the exact point values with subset DP.

The verified column means that every value completed by backtracking before timeout was equal to the corresponding DP value. For timed-out rows, the program cannot verify values after the stopping color count because the baseline did not finish those values. However, the prefix agreement is still useful because both algorithms are very different internally. A mismatch in the prefix would immediately reveal a bug in either the DP recurrence, the independent-set table, or the backtracking baseline.

C. Analytical Discussion

The result shows a very large practical gap. At $n = 8$, backtracking still completes, but it already takes 200.672 ms, while DP + Lagrange takes only 0.098 ms. Starting from $n = 10$, backtracking cannot finish the full interpolation input within the 2000 ms limit. Meanwhile, the DP + Lagrange method still finishes in less than one millisecond for $n = 12$, 3.155 ms for $n = 14$, and 8.842 ms for $n = 15$.

This happens because backtracking still has to count valid assignments explicitly. Even with pruning, it is still fundamentally walking through a huge coloring tree. The DP method does not enumerate complete colorings one by one. It groups many assignments together by using subsets, so the repeated structure of the problem is reused.

The number of recursive nodes visited by backtracking also shows why the timeout happens. In the raw CSV, the $n = 10$ graph visits more than 225 million recursive nodes before stopping. Even the dense $n = 12$ graph still visits more than 111 million nodes before timeout. These numbers are not direct proof of the theoretical worst case, but they explain the practical behavior that backtracking spends its time exploring many partial colorings, while the DP stores aggregate counts.

There is also a tradeoff. The DP table grows as $\mathcal{O}(2^n)$, and this memory growth will eventually become the bottleneck. The experiment table does not push that boundary because the benchmark intentionally uses small graphs and exact 64-bit counts. However, the asymptotic behavior is clear, for larger graphs, especially around the mid-twenties and above depending on the machine and integer representation, memory will become a serious limitation. So the method is much better for time, but it is not free.

Another limitation is that this experiment measures one implementation, not every possible implementation of chromatic-polynomial computation. A highly optimized deletion-contraction solver or a specialized graph library may perform better on certain graph families. Even so, the experiment still satisfies the goal of this paper. It shows that the proposed DP + Lagrange pipeline is correct on the checked values and gives a concrete improvement over a straightforward coloring enumerator.

V. CONCLUSION

This paper shows that combining Dynamic Programming over subsets with Lagrange Interpolation can compute the chromatic polynomial much more efficiently than direct coloring enumeration. The main idea is to count $P(G, k)$ for enough integer values of k by treating every color class as an independent set, then reconstruct the full polynomial from those values.

The theoretical improvement comes from changing the main computation from naive $\mathcal{O}(k^n)$ enumeration into subset-based transitions with $\mathcal{O}(k3^n)$ time for a fixed color value, or $\mathcal{O}(n3^n)$ for the interpolation pipeline from 1 to $n+1$ colors. The memory usage is still exponential, $\mathcal{O}(2^n)$, so this method does not remove the hardness of the problem completely. It simply moves the bottleneck into a form that is much more manageable for small and medium graphs.

The experiment also supports the theory. On deterministic random graphs with $n = 5$ to 15, DP + Lagrange finished in 0.022–8.842 ms, while naive backtracking reached the 2000 ms timeout starting at $n = 10$. The density sweep also showed that denser graphs can be easier for this specific DP transition because they have fewer independent subsets. Therefore, DP + Lagrange is clearly superior in execution time for the tested graphs, although its exponentially growing table remains the main limitation for larger inputs.

APPENDIX

The link to the repository for this paper is provided here:

https://github.com/Joji0/chromatic-polynomial

The link to the video explaining this paper is provided here:

https://youtu.be/lQ1gtua1sEA.

ACKNOWLEDGMENT

The author would like to thank Prof. Dr. Ir. Rinaldi, M.T. for his guidance and lectures in the IF2211 Strategi Algoritma course at Institut Teknologi Bandung, which provided the foundational knowledge required for this research.

REFERENCES

[1] J. K. Wicaksono, "Greedy vs Backtracking: A Comparative Study of Graph Vertex Coloring Algorithms with C++ Implementations," *Makalah IF1220 Matematika Diskrit*, Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025-2/Makalah2025/Makalah-Matdis-2025-IF-ITB%20(30).pdf.

[2] R. C. Read, "An Introduction to Chromatic Polynomials," *Journal of Combinatorial Theory*, vol. 4, pp. 52–71, 1968. [Online]. Available: https://doi.org/10.1016/S0021-9800(68)80087-0.

[3] A. D. Sokal, "Chromatic Polynomials, Potts Models and All That," *arXiv preprint cond-mat/9910503*, 1999. [Online]. Available: https://arxiv.org/pdf/cond-mat/9910503.

[4] F. Bencs, J. Huijben, and G. Regts, "Approximating the Chromatic Polynomial Is as Hard as Computing It Exactly," *arXiv preprint arXiv:2211.13790*, 2022. [Online]. Available: https://arxiv.org/abs/2211.13790.

[5] J.-P. Berrut and L. N. Trefethen, "Barycentric Lagrange Interpolation," *SIAM Review*, vol. 46, no. 3, pp. 501–517, 2004. [Online]. Available: https://people.maths.ox.ac.uk/trefethen/barycentric.pdf.

[6] J. K. Wicaksono, "Lagrange Interpolation," *VoidSolve()*, n.d. [Online]. Available: https://voidsolve.vercel.app/mathematics/polynomials/lagrange-interpolation. Accessed: Jun. 18, 2026.

[7] J. A. Mudrock, "A Deletion-Contraction Relation for the DP Color Function," *arXiv preprint arXiv:2107.08154*, 2021. [Online]. Available: https://arxiv.org/abs/2107.08154.

STATEMENT

Hereby, I declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not a product of plagiarism.

Bandung, 18 June 2026

Jonathan Kris Wicaksono
13524023