Naive vertex classification and the Cardon-Crochemore algorithm

Dario Moschetti

August 2025

1 Introduction

In mathematics and computer science, graphs are a fundamental structure used in many context to represent objects and their relationships. A graph is a set of objects, called nodes or vertices, which are in relation with each other, through edges.

Since graphs can represent a wide variety of objects and relationships, they can be used to model numerous problems. We had the chance to see many examples of graphs in the meetings held throughout the year: the web graph, social networks, flow networks and many more. These are just a few examples of the broad range of graph applications, but they give an idea of how powerful and versatile graphs are, and how important it is to understand how to use, interpret and manipulate them.

For this reason, the study of graphs is a rich field with its own name: graph theory. In this article, we will focus on one specific aspect of graph theory, the problem of determining whether two graphs are isomorphic or not (namely, essentially, the same graph up to vertex renaming), which has several useful applications.

1.1 Motivation

To understand the importance of the problem we are studying, we begin with a real-world example of its application.

Consider the digital representation of a molecule, using a graph. The nodes represent atoms, and the edges represent bonds between them. This is a common way to represent molecules because it captures the structure of the molecule through the edges, the types of atoms through the node labels, and the types of bonds through the edge labels.

Now imagine we have a database of molecules, and we want to check whether a given molecule is present in the database. Since molecules are represented as graphs, the problem reduces to checking whether a given graph (representing the molecule) already exists in the database, comparing it with the stored graphs.

This is a problem of distinguishing graphs, as we need to determine whether two graphs are identical or not.

This problem has other applications as well, which we will explore later, making it interesting not only from a theoretical perspective but also from a practical one.

1.2 Distinguish graphs

To understand what it means for two graphs to be the same, we start from the definition of a graph. A graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E, which are pairs of nodes. So V is a set, and E is a binary relation in the mathematical sense, defined on the set V. This definition fits particularly well with the directed graphs, where the edges are ordered pairs of nodes, but it also works for undirected graphs by assuming that the relation is symmetric. In this sense, two graphs are the same if they represent the same relation on the same set of objects, that is, if they have the same edges between nodes. So, given two graphs, we can check if they are the same by checking if the they have the same nodes and if each node is connected (in relation) to the same nodes in both graphs. The set of nodes connected to a node is called its neighborhood (we will see that the definition may change a little bit when we consider directed graphs), so two graphs are the same if each node has the same neighborhood in both graphs. Let's illustrate this with an example.

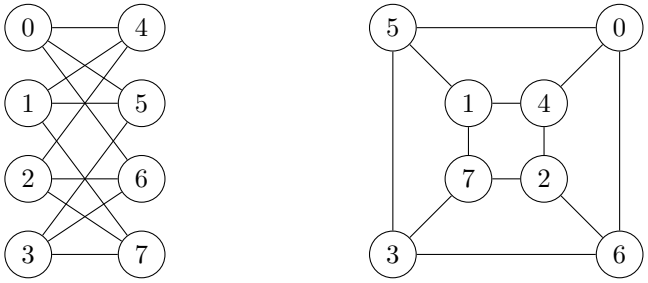


Figure 1: Two graphs

2 The graph isomorphism problem

In the previous section, we introduced a definition of when two graphs are the same, along with a routine for testing it, assuming that the nodes in the two graphs could be identified by their labels.

However, in most cases, we use graphs to represent connections between generic objects, not specific ones. For example, in a social network, we may not be interested in distinguishing users by their profile information, but rather in understanding the overall structure of the network, such as the connections

between generic users. This idea requires a more powerful use of graphs, allowing us to represent relationships without being tied to specific identities.

This application, however, requires us to reconsider our definition of when two graphs are the same, specifically, one that does not rely on node labels.

2.1 Graph isomorphism

This brings us to the concept of graph isomorphism, a more general definition of graph equivalence that we already discussed informally in section 1.2.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there exists a bijection $f : V_1 \to V_2$ such that for every pair of nodes $u, v \in V_1$, $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$.

In other words, two graphs are isomorphic if there is a way to relabel the nodes so that the resulting graphs are identical according to the earlier definition.

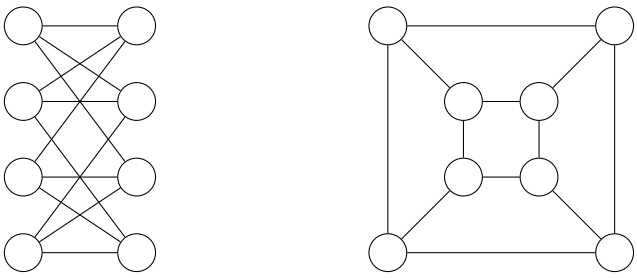


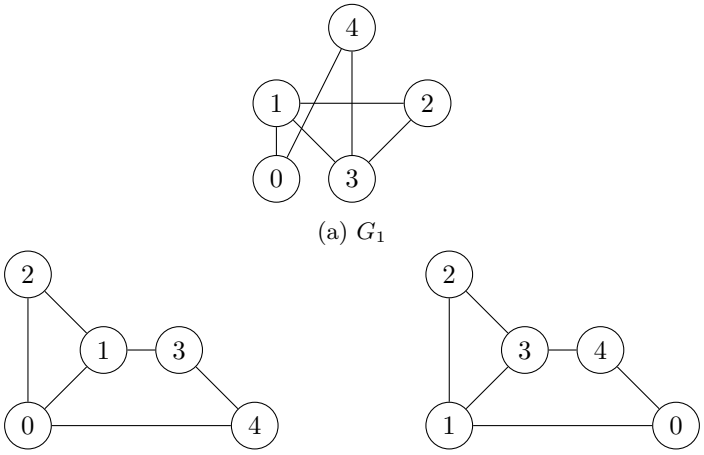
Figure 2: Two isomorphic graphs

In the previous example, the two graphs are isomorphic because we can relabel their nodes to make them identical (as shown in Figure 1). Thus, the notion of graph isomorphism captures the idea of two graphs having the same structure, regardless of the node labels.

2.2 The problem

Deciding whether two graphs are isomorphic involves determining whether there exists at least one valid labeling of the nodes that makes the two graphs identical. This is a more complex task than simply comparing neighborhoods, as it also requires choosing an appropriate labeling, unlike the previous case, where the labels were fixed.

What makes the problem hard is the fact that a labeling being not valid does not implies that the graphs are not isomorphic. In fact, an exponential number of labelings exist, but not all will reveal the isomorphism. Let us illustrate this with an example.



(a) G_1

(b) G_2 - Labeling 1

(c) G_2 - Labeling 2

Figure 3: Different labelings can highlight or hide the isomorphism

In this example, we want to determine whether G_1 and G_2 are isomorphic as unlabeled graphs. If we label the nodes of G_2 as shown in Figure 3 (b), the two graphs do not appear identical, for instance, the neighborhood of node 2 is different. However, if we relabel the nodes as shown in Figure 3 (c), the graphs become identical, revealing that they are indeed isomorphic.

The number of possible labelings of a graph with n nodes is $n!$, making it impractical to check all of them by brute force. This implies that we need to look for more efficient methods to solve the graph isomorphism decision problem.

2.3 The color refinement heuristic

Since we are trying to find a bijection between the nodes of two graphs that preserves neighborhoods, we want to pair nodes that have "similar" neighborhoods in their respective graphs. For example, two nodes can only correspond if they have the same degree. Therefore, for two graphs to be isomorphic, the number of nodes with a given degree must be the same in both graphs. Otherwise, any labeling would contain at least one pair of nodes with different neighborhood sizes, proving that the graphs are not isomorphic.

As a first step, we can assign a color (or label) to each node based on its degree and then verify that the two graphs have the same number of nodes for each color. However, this approach alone is not sufficient to distinguish all non-isomorphic graphs, since many such graphs share identical degree distributions. This limitation arises because we are only considering the size of each neighborhood, not its structure.

To improve this, we can refine the characterization of nodes by also considering the colors of their neighbors. The idea is to iteratively update node colors based on their current color and the multiset of colors in their neighborhood.

Repeating this process refines the coloring at each step, eventually providing a more precise representation of the graph's structure which can highlight differences between non-isomorphic graphs.

At the end of each iteration, the color distributions of the two graphs must match; otherwise, the graphs cannot be isomorphic for the same reason as before. This procedure is known as **color refinement**, or the 1-dimensional Weisfeiler–Lehman (WL) test, and is a common approach to tackle the graph isomorphism problem.

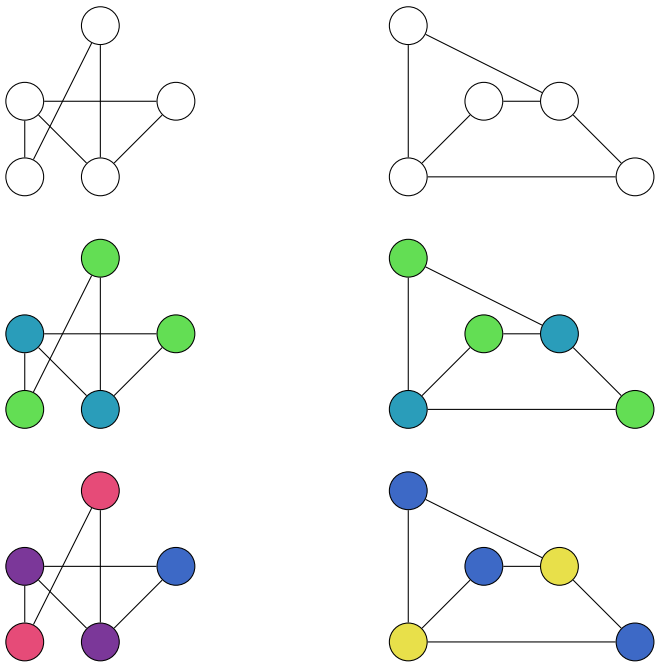


Figure 4: Colors in the two graphs are different, hence the graphs are not isomorphic

The algorithm terminates either when a difference between the graphs is detected or, as we will see, when the refinement stabilizes.

2.4 Edge labels

The algorithm can also be modified to take edge labels into account, increasing its expressiveness. For instance we can model weighted edges or, as in the molecule example, represent different bond types with distinct labels. It also applies naturally to directed graphs and can incorporate both out- and in-neighborhood information if desired.

The modification is simple: the neighborhood of a node is no longer represented by the multiset of colors of its neighbors, but by the multiset of pairs

(edge label - node color). Conceptually, we keep separate neighbor counts for each edge label and apply the refinement rule per label.

3 Naive vertex classification

From now on, we will focus on directed graphs, considering only the out-neighborhood of each node when defining its color. Although this choice sacrifices some expressiveness by not taking in-neighborhoods into account, it simplifies the explanation of the algorithm and can be easily adapted to include them if needed. In particular, although we restrict attention to out-neighborhoods, we can easily include in-neighborhood information by reinterpreting all incident edges as part of the out-neighborhood. To preserve their original orientation, each edge is then labeled according to its role: "out" if it was originally outgoing, and "in" if it was originally incoming. In this way, every edge belongs to the out-neighborhood of both its endpoints, but with opposite labels.

More formally, we would like to define an equivalence relation R on the set of nodes V, such that

$$(x, y) \in R \Rightarrow \forall C \ |N(x) \cap C| = |N(y) \cap C| \tag{1}$$

where $N(x)$ is the out-neighborhood of node x, and C is a color class of R.

The refinement rule of the color refinement algorithm enforces this condition: if two nodes x and y have a different number of neighbors in some color class C, that is, if their neighborhood multisets differ, then they are separated into different classes and are no longer in relation.

It is important to note that the coloring defined in this way is not merely a tool for distinguishing non-isomorphic graphs, but a relation that captures structural similarity between nodes. As we will see, it has applications beyond graph isomorphism testing.

A relation that satisfies the above condition is said to be **regular**. Proving that such a relation always exists is trivial: place every node in its own class so that no two nodes can violate the condition. This also explains why the algorithm always converges. At each step, the current relation is either regular or not. If it is regular, no further refinement is needed, the relation remains unchanged, and the procedure stops. If it is not regular, the algorithm refines the relation by separating nodes that violate the regularity condition. This can only happen a finite number of times because the number of nodes is finite, and it eventually leads to a regular relation.

3.1 Coarsest regular congruence

We showed that finding a regular relation is not hard, but when multiple regular relations can be defined on the set of nodes, which one is most interesting?

Such a relation is called the **coarsest regular congruence** and has three properties. We want a relation R that is:

- **Regular**: it satisfies the regularity condition defined above.
- **Coarsest**: it has the fewest classes possible.
- **A refinement of a given relation S.**

3.1.1 Refinement

Relation R is a **refinement** of relation S if:

$$(x, y) \in R \Rightarrow (x, y) \in S \tag{2}$$

That is, if two nodes are in relation according to R, they are also in relation according to S; equivalently, if two nodes are in different classes according to S, then they remain in different classes in the refinement R.

Nodes in the same class at some point may be separated later, but nodes in different classes will never be put in the same class in the future.

In the examples so far, color refinement starts with all nodes having the same color, and we gradually distinguish them exploiting the graph structure. However, we may want to encode prior knowledge about the nodes. Treat this prior knowledge as labels and start from an initial coloring S that groups nodes by label. In that case, nodes with different labels must never be merged, which the refinement property guarantees.

Returning to the coarsest regular congruence with the three properties defined above, it can be proved that it always exists and is unique [6], and the problem of finding it is known as **naive vertex classification**.

4 The Cardon-Crochemore Algorithm

It can be proved that the color refinement procedure described above converges to the coarsest regular congruence, thereby solving the naive vertex classification problem.

To assess efficiency, we analyze how quickly it solves naive vertex classification; in particular, we study its time complexity.

Given a graph G with n nodes and m edges, the procedure takes at most $n - 1$ iterations to converge, since at each step at least one class is split and classes never merge. This bound is tight and there exist infinite many graphs that realize it [10]. Each iteration requires checking the neighborhood of each node, which takes $O(m)$. Therefore, the overall time complexity of the color refinement algorithm is $O(nm)$.

Cardon and Crochemore proposed a more efficient algorithm to solve naive vertex classification, with time complexity $O((n + m) \log n)$.

4.1 Improvement ideas

The authors exploit two main ideas to achieve this improvement. The first idea is that, since the color of a node depends on the colors of its successors, a class

can be split only if at least one successor of some node in the class changed color in the previous iteration (that is, the successor class was split); otherwise the class cannot become irregular. Therefore, if all successors of the nodes in a class did not change color in the previous iteration, the class is regular and we can skip it in the current iteration (we do not check the neighborhoods of its nodes). At each iteration we track the classes that were split, and in the next iteration we check only those classes that contain at least one predecessor of a node from the split classes. In addition, we do not need to check all the neighbors of a node, but only those in the split classes, since the others cannot change the node color.

The second idea is that it is not necessary to inspect every node in a class to decide whether and how that class should be split.

4.1.1 Lazy checking

We call this second idea **lazy checking**. A small example clarifies it.

Consider two classes A and B, where every node in A has k successors in B. Suppose B is split into two classes C and D. Class A may now be irregular: each node can have j successors in C and $k - j$ in D, and nodes with different j must be separated. However, we do not need to check all nodes in A. It is enough to examine the predecessors of one of C or D and count their edges to that subclass; the count to the other subclass follows by difference. To minimize work, choose the smaller of C and D.

Thus, whenever a class splits in two, we process only the smaller new subclass, which we call significant, and ignore the other, knowing we can obtain its counts by subtraction.

In general, a class can split into more than two subclasses. By the same reasoning, we ignore the largest new subclass (or one of the largest, in case of ties) and consider all the others, that are said **significant**.

This idea was first introduced by Hopcroft (1971) for state minimization in deterministic finite automata [9], and later generalized to handle splits into more than two subclasses by Cardon–Crochemore.

4.2 The algorithm

Let us summarize the Cardon–Crochemore algorithm. As noted earlier, to decide how to split a class we do not need to inspect the entire neighborhood of its nodes, but only the neighbors that lie in significant classes, which are sufficient to reveal any irregularities. The algorithm maintains a stack ρ of significant classes, initially containing all classes of the initial coloring I. At each iteration, it pops all the classes from ρ and uses them to characterize the predecessor nodes. For each such node, it builds a compact record of the relevant part of its neighborhood. It then refines the partition by assigning a new class to any node whose record differs from others in its current class. Newly created significant classes are pushed onto ρ. The process repeats until ρ is empty.

5 Implementation and complexity

5.1 Implementation

Algorithm 1 Cardon-Crochemore NVC

Input: Graph $G = (V, E)$, initial partition I
Output: Coarsest regular congruence of G refining I

1: $R \leftarrow I$; $C \leftarrow R$-classes; $\rho \leftarrow$ indices of R-classes
2: $\tilde{E} \leftarrow \{(u, v) \mid (v, u) \in E\}$
3: **repeat**
4: **while** $\rho \neq \emptyset$ **do**
5: $r \leftarrow \text{pop}(\rho)$
6: **for** $v \in C_r$ **do**
7: **for** $w \in \tilde{E}(v)$ **do**
8: Append r to $L(w)$
9: $LC \leftarrow \{i \mid \exists v \in C_i : L(v) \neq \emptyset\}$
10: **for** $i \in LC$ **do** $n(i) \leftarrow |\{v \in C_i : L(v) \neq \emptyset\}|$
11: SORT(L) lexicographically
12: **while** $L \neq \emptyset$ **do**
13: Extract first (v, S) from L
14: $i \leftarrow R(v)$
15: **if** $n(i) \neq 0$ **then**
16: $T(i) \leftarrow S$; $B(i) \leftarrow \{i\}$
17: **if** $n(i) \neq |C_i|$ **then** push new index in $B(i)$
18: $n(i) \leftarrow 0$
19: **if** $T(i) \neq S$ **then** push new index in $B(i)$
20: $r \leftarrow \text{top}(B(i))$; move v from C_i to C_r; $R(v) \leftarrow r$
21: **while** $LC \neq 0$ **do**
22: Extract i from LC
23: **for** $c \in B(i) \setminus \{i\}$ **do** push c in ρ
24: $r \leftarrow \arg\max_{c \in B(i)} |C_c|$ (if more than one, just pick the first)
25: **if** $r \neq i$ **then** update R and C in order to swap C_r and C_i
26: **until** $\rho = \emptyset$
27: **return** R

The algorithm is quite technical, so it is worth providing some explanation. R is the relation we are computing on the graph. It is represented as an array where each entry stores the index of the color class of the corresponding vertex. In lines 4–8, the algorithm populates the dictionary L. For each predecessor w of a node belonging to a significant class, the algorithm associates w with the list of colors of its successors in significant classes. This information is sufficient to distinguish w from the other nodes in its class. The lists are then sorted lexicographically, using a specialized sorting procedure that reduces the

algorithm's complexity by exploiting structural properties of the values that can appear in the lists. In lines 12–20, the algorithm splits the nodes according to the lists in L and creates all necessary subclasses. Finally, in lines 21–25, the indexes of the newly created subclasses are added to ρ for the next iteration, while the largest class is excluded from the set of significant classes.

5.2 Complexity

The authors prove that in each iteration, the total cost is proportional to

$$\sum_{C \in \rho} |\{(u, v) \in E \mid v \in C\}|$$

where ρ is the stack of significant classes, and E is the set of edges. Thus, the total cost of the algorithm is proportional to

$$\sum_{i=0}^{N} \sum_{C \in \rho_i} |\{(u, v) \in E \mid v \in C\}|$$

where N is the number of iterations, at most $n - 1$, with $n = |V|$.

We can refine the analysis. Let $i > 0$ and consider a class $C \in \rho_i$. Since C is significant, it must have been created by splitting some class C' in the previous iteration, and by definition it is not the largest new subclass (or not the only one). Hence $|C| \leq |C'|/2$. Therefore, each node can belong to a significant class at most $\log_2 n$ times.

In the worst case, the running time is

$$\log_2 n \sum_{v \in V} \big|\{(u, v) \in E\}\big| = \log_2 n \cdot m = O(m \log n).$$

Thus, the time complexity of the Cardon–Crochemore algorithm is $O(m \log n)$.

A more precise analysis that takes implementation details into account yields $O((n + m) \log n)$.

5.3 Experimental results

We conducted experiments to compare the Cardon–Crochemore algorithm with the naive implementation of color refinement, using adversarial graphs designed to expose the naive algorithm's inefficiencies. Specifically, we built graphs with n nodes where half form a path (chain) and the other half form a clique, connected by a single edge. This handcrafted instance is hard for the naive algorithm: the chain propagates colors over many iterations (in the order of $|V|$), and in each of those iterations the number of edges to inspect is large because of the clique. On the other hand, the Cardon–Crochemore algorithm runs for a similar number of iterations, but inspects far fewer edges overall. All edges inside the clique are examined only once, while in the remaining iterations it touches essentially

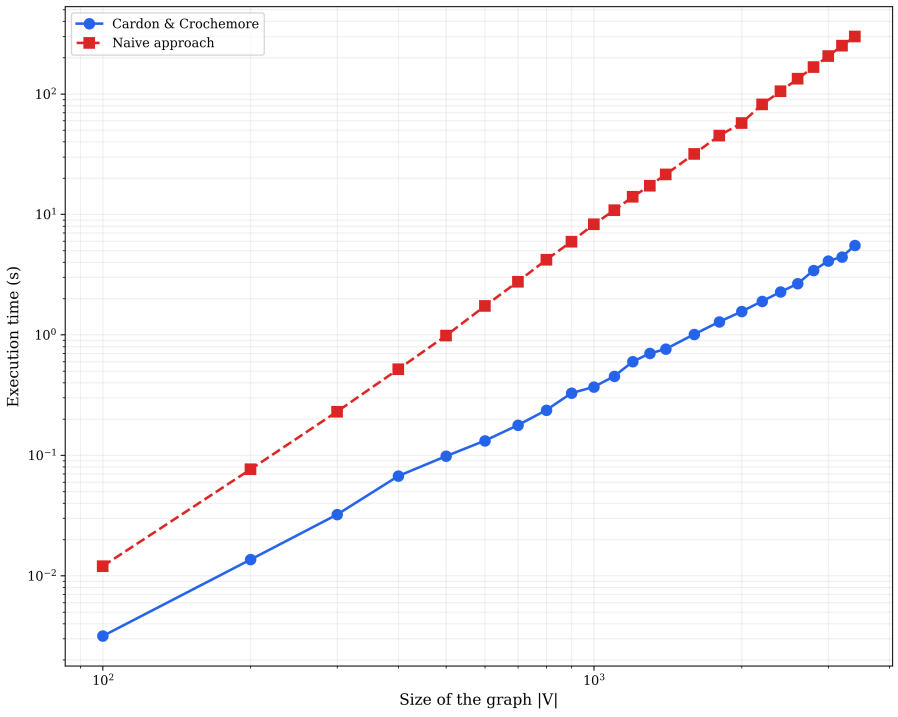
only the chain edges, reducing the cost of every iteration after the first to $O(1)$ instead of $O(n^2)$.

For the Cardon–Crochemore algorithm, we implemented a Python version based on the original paper. For the naive one, we used the implementation provided by the NetworkX Python library [8]. The NetworkX function sets the maximum number of iterations to 3 by default, as this is often sufficient for practical isomorphism tests, however, we forced it to run until convergence. All the code is available on GitHub [1].

Size	200	400	800	1200	1600	2000	3400
Cardon-Crochemore (ms)	13.62	67.31	236.84	598.95	1009.29	1561.85	5510.94
Naive Approach (ms)	76.43	518.95	4189.14	13 974.32	31 696.16	57 324.42	299 402.57
Speedup Ratio	5.61	7.71	17.69	23.33	31.40	36.70	54.33

Table 1: Performance comparison between the Cardon–Crochemore algorithm and the naive approach

Table 1 reports running times for both algorithms on the same instances. The Cardon–Crochemore algorithm is faster in every case, and the gap widens with graph size, in line with the theoretical analysis showing its asymptotic advantage over the naive method.



[1] `https://github.com/moschettid/nvc-study`

To visualize the results, the plot shows execution time versus graph size on a log–log scale. The Cardon–Crochemore algorithm is the blue line and the naive approach is the red line. The near-linear trends suggest polynomial-time behavior for both algorithms. The smaller slope of the blue line reflects the asymptotic advantage of Cardon–Crochemore over the naive approach ($O((n + m) \log n)$ versus $O(nm)$).

5.4 Adoption and overlooked potential

Another algorithm that solves the naive vertex classification problem with the same time complexity as Cardon–Crochemore was later proposed by Paige and Tarjan [13]. They describe their method as "somewhat simpler", particularly in how it handles refinements.

Despite these two algorithms having better asymptotic complexity than naive color refinement, and our constructed instances where they outperform it, they are not widely adopted, in particular in the context of graph isomorphism testing, which is the main application of the color refinement routine. For example, in NetworkX the naive color refinement is the only available option for this technique. It assumes that only a few graphs will trigger the naive algorithm's worst-case behavior, and to keep performance acceptable in those cases, it caps the number of iterations at 3. Although this may not reach a stable partition, it often suffices to distinguish most graphs.

Furthermore, the theoretical advantage is primarily for certain graph families and is asymptotic. We have not systematically studied performance on random graphs, where the asymptotic complexities are similar and constant factors and cache efficiency may play a larger role in determining which approach is faster.

We were unable to find pre-existing implementations of the Cardon–Crochemore algorithm, so we implemented our own in order to conduct our experiments. This further suggests that the method is not widely adopted, despite its theoretical advantages.

6 Limitations and Theoretical Boundaries

6.1 Expressiveness

We have seen that the color refinement routine can be a powerful tool for distinguishing non-isomorphic graphs, but it is not complete. There exist graphs that it does not distinguish, meaning it cannot produce a coloring that reveals their differences, even at the stable partition. For instance, starting from a uniform initial coloring, any d-regular graph remains monochromatic under color refinement. Thus any two d-regular graphs with the same number of vertices are indistinguishable by color refinement. For example, a 6-cycle and the disjoint union of two triangles. More generally, there are infinitely many pairs of non-isomorphic graphs that the routine cannot distinguish, even after stabilization.

On the other hand, the algorithm performs very well on most graphs. Random graphs are identified almost surely [3, 1]. In particular, as n grows, the fraction of n-vertex graphs it does not identify decreases exponentially [4]. A smoothed analysis [1] further shows that adding or removing only $O(n \log n)$ random edges to any fixed graph makes it identifiable with high probability, indicating that hard instances are fragile under small random perturbations.

These results make color refinement an effective tool in graph isomorphism testing. In particular, its quasi-linear running time in practice $O((n+m) \log n)$ is far better than the worst-case bounds for exact isomorphism.

In fact, the best-known algorithm for solving the graph isomorphism problem is Babai's algorithm [2], introduced in 2015, which runs in quasi-polynomial time, that is, $O(n^{O(\log n)})$, making it practically infeasible for very large graphs.

As of today, the graph isomorphism problem is neither known to be NP-complete nor known to be in P, leaving open whether it is tractable. A complexity class GI has been defined to contain the problems that reduce to Graph Isomorphism. If Graph Isomorphism were NP-complete, then together with Babai's quasi-polynomial-time algorithm it would imply that every problem in NP admits a quasi-polynomial-time algorithm.

6.2 Time complexity bound

We presented two algorithms and cited a third for solving the naive vertex classification problem: the first runs in $O(nm)$ time, and the other two in $O((n+m) \log n)$. More recently, it was shown that no algorithm can solve naive vertex classification faster than $\Omega((n+m) \log n)$ under some assumption on the type of algorithm that captures all known color-refinement algorithms [5]. In particular, the lower bound holds for all algorithms that use partition refinement operations.

7 Applications

Both the Graph Isomorphism problem and the naive vertex classification problem are fundamental in graph theory and have applications in computer science.

Graphs represent a wide range of structures, from social networks to chemical compounds, and identifying structural similarity is important. For example, in chemistry, molecules can be represented as graphs, and checking whether a molecule is in a database corresponds to testing whether the associated graph is isomorphic to one already stored. One of the earliest appearances of the naive vertex classification problem was in this context, where the goal was to assign a unique label to each molecule to enable efficient membership queries in a database [11]. This task is called graph canonization, and it is closely related to the graph isomorphism problem. Naive vertex classification performs well in practice, as noted earlier. The same idea applies in any setting where the objects to be stored can be represented as graphs. In network analysis, it is also useful to find structurally equivalent networks for optimization purposes.

However, in many such cases users are more interested in similarity detection or approximate matching between graphs rather than exact isomorphism, which helps explain the popularity of these heuristics. Exact graph isomorphism requires a perfect structural match, making it a harder problem and one often regarded as more theoretical than directly applied.

Naive vertex classification is employed in all the aforementioned fields as a graph isomorphism test heuristic, and also in other contexts such as linear programming and graph kernels. In linear programming it can serve as a preprocessing routine to find similar elements (for instance, rows of the constraint matrix) and merge them, reducing instance size [7]. In graph kernels, each iteration's color class counts form a feature vector that characterizes the graph and such vectors are used to define similarity measures between graphs [14].

Furthermore, naive vertex classification has proven useful in understanding the expressive power of graph neural networks (GNNs). It has been shown that message-passing GNNs are at most as expressive as the color refinement procedure: they cannot distinguish nodes that 1-WL does not separate [12, 15].

8 Future work

There are several directions for future work.

First, the practical performance of the Cardon–Crochemore algorithm could be explored further. Its behavior should be compared with the naive approach on multiple families of random graphs to obtain average-case (not only adversarial) evidence. It would also be useful to compare it with the Paige–Tarjan algorithm, which has the same asymptotic complexity and may offer better constant factors.

It may be interesting searching for implementation improvements and exploring a parallel or cache-oblivious version of the algorithm in order to improve efficiency.

Another direction is a dynamic version of the algorithm that handles graph updates (node/edge insertions and deletions) without recomputing from scratch.

On the theoretical side, it would be interesting to characterize graph families on which Cardon–Crochemore consistently outperforms the naive version. The adversarial graphs in 5.3 were handcrafted, and a systematic study might reveal families that widen the gap. Paper [10] presents families with a maximal number of iterations but few operations overall (very sparse graphs). Investigating this density versus iteration-count trade-off could yield sharper bounds or heuristics.

Also a formal analysis of the performance gap between color refinement and Cardon–Crochemore on probabilistic graph classes could provide new information about the expected-case complexity.

A more ambitious (and likely difficult) direction is to design a correct algorithm for naive vertex classification that is not based on partition refinement, potentially bypassing the lower bound in 6.2.

References

[1] Anastos, M., Kwan, M., and Moore, B. Smoothed analysis for graph isomorphism. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2025), STOC '25, Association for Computing Machinery, p. 2098–2106.

[2] Babai, L. Graph isomorphism in quasipolynomial time, 2015.

[3] Babai, L., Erdo˝s, P., and Selkow, S. M. Random graph isomorphism. *SIAM Journal on Computing 9*, 3 (1980), 628–635.

[4] Babai, L., and Kucera, L. Canonical labelling of graphs in linear average time. *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)* (1979), 39–46.

[5] Berkholz, C., Bonsma, P., and Grohe, M. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theor. Comp. Sys. 60*, 4 (May 2017), 581–614.

[6] Cardon, A., and Crochemore, M. Partitioning a graph in o(|a|log2|v|). *Theoretical Computer Science 19*, 1 (1982), 85–98.

[7] Grohe, M., Kersting, K., Mladenov, M., and Selman, E. Dimension reduction via colour refinement. In *Algorithms - ESA 2014* (Berlin, Heidelberg, 2014), A. S. Schulz and D. Wagner, Eds., Springer Berlin Heidelberg, pp. 505–516.

[8] Hagberg, A. A., Schult, D. A., and Swart, P. J. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference* (Pasadena, CA USA, 2008), G. Varoquaux, T. Vaught, and J. Millman, Eds., pp. 11 – 15.

[9] Hopcroft, J. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Academic Press, 1971, pp. 189–196.

[10] Kiefer, S., and McKay, B. D. The iteration number of colour refinement, 2020.

[11] Morgan, H. L. The generation of a unique machine description for chemical structures-a technique developed at chemical abstracts service. *Journal of Chemical Documentation 5* (1965), 107–113.

[12] Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. Weisfeiler and leman go neural: higher-order graph neural networks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (2019), AAAI'19/IAAI'19/EAAI'19, AAAI Press.

[13] Paige, R., and Tarjan, R. E. Three partition refinement algorithms. *SIAM Journal on Computing 16*, 6 (1987), 973–989.

[14] Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., and Borgwardt, K. M. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res. 12*, null (Nov. 2011), 2539–2561.

[15] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? *ArXiv abs/1810.00826* (2018).