


SUBGRAPH ISOMORPHISM - ALGORITHM ANALYSIS PROJECT

FINAL REPORT

 **Santiago Martínez Loaiza**
s.martinezl234@uniandes.edu.co
202510729

 **Estefania Laverde**
e.laverdeb@uniandes.edu.co
201922512

 **Pablo Ortega**
p.ortegac@uniandes.edu.co
202021700

May 7, 2025

1 Problem Definition and Modeling

The subgraph isomorphism problem is a fundamental question in computer science and discrete mathematics, with significant applications in domains such as bioinformatics, cybersecurity, social network analysis and pattern recognition. It is a topic of continuous and substantial research interest due to its NP-completeness. Since the problem is computationally complex, finding efficient algorithms and heuristics for practical instances remains a central challenge in the mentioned areas of knowledge. Formally, the problem is described as follows.

Definition: let $S = (V_S, E_S, L_S)$ and $G = (V_G, E_G, L_G)$ be two simple, undirected graphs, where $|V_S| \leq |V_G|$, composed of the sets of vertices and edges and the function L that assigns labels to each vertex. The graph S is isomorphic to a subgraph of G if there exists an injective function $m : V_S \rightarrow V_G$ such that

- **m preserves edges:** $\forall (u, v) \in E_S \rightarrow (m(u), m(v)) \in E_G$.
- **m preserves labels:** $\forall u \in V_S \rightarrow L_S(u) = L_G(m(u))$

In the general case, all vertices of S and G are assigned the same label. For example, consider the graphs in Figure 1. The graph S is isomorphic to a subgraph of G under the function $m : V_S \rightarrow V_G$ such that $m(a) = c$, $m(b) = f$, $m(d) = d$, $m(c) = a$ and $m(e) = e$.

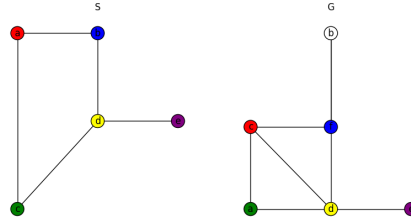


Figure 1: Simple and undirected example graphs.

The input and output values necessary to formalize the algorithms are shown in Table 1.

1.1 Preconditions and Postconditions

The formal specification for subgraph isomorphism requires two key preconditions.

- The patterns graph S and the target graph G must be valid graphs, i.e., no loops, multiple edges and it must be undirected:

$E \subseteq \{\{u, v\} | u, v \in V, u \neq v\}$ is a **set** of edges, where each edge is an unordered pair of distinct vertices.

I/O	Description	Type	Description
Input	G	(V_G, E_G, L_G)	Simple undirected graph, where V_G is the set of vertices, E_G the set of edges and L_G the function that assigns a label to each vertex.
Input	S	(V_S, E_S, L_S)	Simple undirected graph, where V_S is the set of vertices, E_S the set of edges and L_S the function that assigns a label to each vertex.
Output	m	Mapping (python dictionary)	Mapping containing the associations between the vertices of S and the vertices of G if a subgraph isomorphism is found. If no such isomorphism exists, m is an empty mapping.

Table 1: Formal specification of inputs and outputs

- The amount of vertices in S must be less or equal than the vertices in G : $|V_S| \leq |V_G|$.

Upon algorithm completion, two postconditions exist ensuring that it correctly solves the subgraph isomorphism problem.

- If m is not empty, m preserves edges and labels as described in the formal definition:

$$m \neq \emptyset \iff \forall u \in V_S, L_S(u) = L_G(m(u)) \text{ and } \forall \{u, v\} \in E_S, \{m(u), m(v)\} \in E_G.$$

- If the mapping m is empty, this indicates that no subgraph of G is isomorphic to S :

$$m = \emptyset \iff \nexists \text{ injective } m : E_S \rightarrow E_G \text{ preserving labels and edges as above.}$$

2 Subgraph isomorphism applications in Bioinformatics, Cybersecurity and Computer Vision

The subgraph isomorphism problem plays a central role in several scientific and technological domains, as it provides a natural framework for detecting structured patterns within large and complex datasets. Among its most impactful applications are chemical and biological structure identification and cybersecurity, where understanding hidden or recurrent substructures is critical for advancing both research and practical solutions.

In computational chemistry and bioinformatics, molecules and biological systems are naturally modeled as labeled graphs. In these representations, nodes correspond to atoms or amino acid residues, edges represent chemical bonds or biological interactions, and labels capture properties such as element type or bond order. Identifying whether a specific molecular substructure is present within a larger molecule, or finding functional motifs within a protein interaction network, is essential for tasks such as drug discovery, molecular design, and understanding biological function [5]. The subgraph isomorphism problem formalizes these questions by asking whether the structure of interest can be mapped into a larger molecular graph, preserving both adjacency constraints and labeling. Solving this problem enables researchers to efficiently screen large molecular databases for compounds with desired properties, greatly accelerating scientific discovery.

In cybersecurity, networks of computers, users, and data flows are frequently modeled as graphs, where nodes represent hosts, servers, or processes, and edges represent communications or transactions. Attack patterns, malware behaviors, and abnormal network activities can be characterized by distinctive graph structures, making subgraph isomorphism a useful tool for intrusion detection and threat analysis [1]. By formulating known attack signatures as query subgraphs, cybersecurity systems can search for their presence within real-time or historical network data, helping to detect and respond to threats before any damage occurs. This approach allows for the identification of complex, multi-stage attacks that may not be recognizable through simpler, rule-based methods.

Beyond chemistry and cybersecurity, the subgraph isomorphism problem also finds important applications in areas such as computer vision, where it supports object recognition by matching feature graphs; in social network analysis, for detecting communities or information flows; and in pattern recognition tasks across a variety of scientific disciplines. These broad applications underscore the importance of developing efficient and scalable methods for solving subgraph isomorphism.

3 VF2++ algorithm

The VF2++ algorithm, proposed by Alpár Jüttner and Péter Madarasi in 2018 [4], is a significant improvement of the classical VF2 algorithm [2] to solve the subgraph isomorphism problem. The original VF2 algorithm incrementally constructs a partial mapping between the nodes of the pattern and the target graphs, expanding it step by step while verifying consistency and cutting rules at each step. This backtracking search is effective, but can be computationally expensive due to the large search space. Notably, the search space can be conceptualized as a rooted tree, where each node corresponds to a partial injective mapping of vertices from the pattern graph S to the target graph G . The root of the tree represents the empty mapping, and each child node extends its parent by adding a new correspondence between two vertices. The leaves on the tree simulate either complete mappings that are valid isomorphisms or partial mappings that are pruned (not considered further) by consistency or feasibility checks.

VF2++ introduces two very important optimizations to improve the performance of VF2. First, it establishes an optimized matching of the vertices in S by analyzing the structural properties and node labeling of the graphs, which reduces the search space by prioritizing matches that are more likely to succeed, thereby pruning infeasible branches earlier in the search. Second, more aggressive and effective pruning is made possible by the additional feasibility rules that the VF2++ algorithm proposes.

The algorithm process begins by checking preconditions to rule out incompatible instances. If the conditions are satisfied, a node ordering of the nodes in the pattern graph is defined, based on the structural and label heuristics. With the ordering and candidate sets initialized, the main matching function begins; at each recursive call, the algorithm selects the next unmatched pattern node and attempts to pair it with a candidate node. The pairing is subjected to a feasibility check, including consistency and cutting rules. If a pair is deemed infeasible, the algorithm backtracks the state of the mapping and explores an alternative branch. If no candidates remain at a decision point, the branch is no further examined. Otherwise, the pair is added to the mapping and the state of it is updated. If the mapping has every node in S , the graphs are isomorphic. If it is not complete, candidates are recalculated based on the partial mapping and the frontier sets, and the search continues.

The described workflow is visually summarized in the following diagram 2, which provides a high-level overview of the VF2++ algorithm. The diagram illustrates the sequence of steps, including precondition checking, node ordering and candidate selection, feasibility testing, backtracking and mapping completion. For further clarity, the subsequent discussion will reference specific components of the diagram to explain how candidate pair selection and feasibility rules interact.

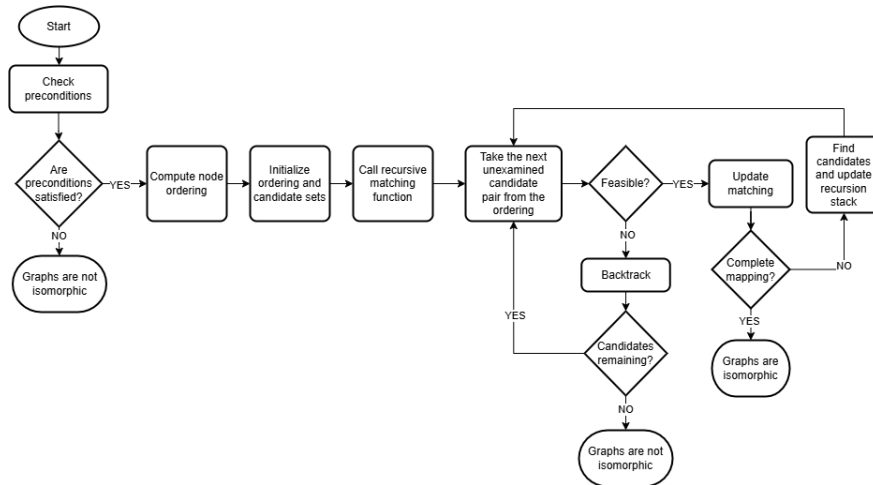


Figure 2: Diagram of the VF2++ algorithm.

3.1 Matching order

The purpose of the matching order is to define the sequence in which pattern nodes are considered for matching, prioritizing those that are more likely to constrain the search. In the implementation by Alpár et al.[4], a breadth-first search (BFS) tree is built for each maximal connected component in the pattern graph. Nodes are ordered primarily by ascending frequency of their labels (rarest labels first), and secondarily by descending node degree on the breadth-first search tree. This way, unfeasible mappings are pruned earlier.

3.2 Feasibility

Feasibility checks are composed of two types of rules: consistency rules, which verify the local validity of a candidate pair, and cutting rules, which anticipate future violations and prune infeasible paths in advance. VF2++ builds upon the classical VF2 rules by enhancing both types, particularly through more aggressive forward-checking mechanisms that exploit the structure and labeling of the graphs.

3.2.1 Consistency

The consistency function varies depending on the specific isomorphism problem: graph isomorphism, induced subgraph isomorphism, or subgraph isomorphism. In this section, we focus specifically on the consistency function for the subgraph isomorphism problem.

Algorithm 1 CHECK_EXTENSION_CONSISTENCY

```

1: function CHECK_EXTENSION_CONSISTENCY( $u, v, f, S, G$ )  $\triangleright f$ : current partial mapping from nodes of  $S$  ( $V(S)$ )
   to nodes of  $G$  ( $V(G)$ )
2:    $V_1 \leftarrow \text{dom}(f)$   $\triangleright$  currently mapped nodes of  $S$ 
3:   for all  $u' \in \text{neigh}(S, u) \cap V_1$  do
4:     if  $f(u') \notin \text{neigh}(G, v)$  then
5:       return false
6:     end if
7:   end for
8:   return true
9: end function

```

The CHECK_EXTENSION_CONSISTENCY function enforces edge-consistency for a proposed mapping ($u \mapsto v$). It iterates over every neighbor u' of node u in the pattern graph S that is already mapped (in the domain of f), and verifies that their corresponding images $f(u')$ are adjacent to node v in the target graph G . If any required adjacency is absent, the extension is rejected; otherwise, it is accepted as consistent.

3.2.2 Cutting

Next, we detail the cutting function specific to the isomorphism problem. We make use of a helper function GAMMA, defined below, which collects all neighbors of a given node u with a specified label ℓ in graph H .

Algorithm 2 GAMMA — Retrieve neighbors of a given label

```

1: function GAMMA( $\ell, u, H$ )  $\triangleright H$  is a graph with adjacency lists neighs and node labels labels
2:    $A \leftarrow \emptyset$ 
3:   for all  $w \in H.\text{neighs}[u]$  do
4:     if  $H.\text{labels}[w] = \ell$  then
5:        $A \leftarrow A \cup \{w\}$ 
6:     end if
7:   end for
8:   return  $A$ 
9: end function

```

The cutting function leverages two additional concepts: $T_1(f)$, the set of nodes in graph S adjacent to nodes already mapped (the “frontier” in S), and $T_2(f)$, the analogous set in graph G (the “frontier” in G). This function acts as a “look-ahead” mechanism that anticipates whether a partial mapping can be successfully extended in the future.

Algorithm 3 CHECK_EXTENSION_CUT — VF2++ cutting test for a proposed extension

```

1: function CHECK_EXTENSION_CUT( $u, v, f, S, G$ )                                ▷  $f$ : current partial mapping from  $S$  to  $G$ 
2:    $\mathcal{L} \leftarrow \text{labels}(S) \cup \text{labels}(G)$ 
3:   for all  $\ell \in \mathcal{L}$  do
4:      $r \leftarrow |S.\gamma(\ell, u) \cap S.T_1(f)|$                                 ▷ required neighbors of label  $\ell$ 
5:      $a \leftarrow |G.\gamma(\ell, v) \cap G.T_2(f)|$                                 ▷ available neighbors of label  $\ell$ 
6:     if  $a < r \vee S.\text{label}(u) \neq G.\text{label}(v)$  then
7:       return true                                                        ▷ extension must be cut
8:     end if
9:   end for
10:  return false                                                            ▷ no cutting condition violated
11: end function

```

This cutting test ensures two conditions for the tentative mapping ($u \mapsto v$): firstly, that nodes u and v have identical labels, and secondly, that for every label ℓ , the number of neighbors of u in the frontier of S ($T_1(f)$) does not exceed the corresponding available neighbors of v in the frontier of G ($T_2(f)$). If either condition fails for any label ℓ , the extension is immediately pruned.

3.3 Correctness and complexity analysis

3.3.1 Correctness

The correctness of VF2++ relies on the structure of the mappings it constructs. The algorithm ensures that only mappings that fully satisfy the subgraph isomorphism constraints, called consistent complete mappings, can be produced by extending consistent partial mappings. Additionally, it guarantees that all consistent complete mappings are reachable by extending partial mappings in a valid way.

Although the same complete mapping may be formed through multiple different sequences of node pairings, the correctness is unaffected because each step preserves consistency. The order in which nodes are added does not alter the final result, as long as consistency is maintained throughout the process.

3.3.2 Computational complexity

In this section, we will address the computational complexity of the VF2++ algorithm, considering both the best and the worst-case scenarios.

We begin by outlining the core operations that VF2++ performs while searching for an embedding of the pattern graph S to the target graph G :

- **Pre-processing (matching-order computation)**
A BFS is used to rank the nodes, and sort by (degree, label). Complexity: $O(|V_S| + |E_S|)$.
- **Consistency check**
 - Label & degree equality $O(1)$
 - Already-mapped neighbor test $O(\min\{\deg_S(u), \deg_G(v)\})$.
- **Cutting**
Ensure every still-unmatched neighbor of u keeps at least one feasible candidate after mapping $u \mapsto v$. Complexity: $O(|E_G|)$.

To calculate the complexity of the whole algorithm, consider the size of the search tree. Let $n = |V_G|$ and $m = |V_S| \leq n$. At depth k the branching factor is at most $n - k$ (the number of unused target vertices). Hence the total number of states explored is bounded by the number of injective mappings

$$N_{\text{states}} \leq n(n-1) \cdots (n-m+1) = \frac{n!}{(n-m)!}.$$

Best-case complexity. The best-case scenario occurs when, at each depth, pruning effectively filters out all candidate matches except exactly one. Under such conditions, the search tree degenerates into a single linear path of depth m , at each step running the consistency and cutting checks. Thus, the complexity is:

$$T_{\text{best}} = O(|V_S| + |E_S|) + O(m|E_G|) = O(|V_S| |E_G|).$$

Worst-case complexity. In the worst-case scenario, pruning fails to eliminate any candidates due to symmetric, dense graph structures. The algorithm explores the entire state space. Yielding:

$$T_{\text{worst}} = O\left(\frac{|V_G|!}{(|V_G| - |V_S|)!}\right)$$

4 Proposed solution algorithm

In this section, we present an algorithm designed to solve the subgraph isomorphism problem. We begin by acknowledging that, in principle, any vertex in the pattern graph S could be potentially mapped to any vertex in G . This observation raises a crucial question: how can we reduce as much as possible the search space using vertices' characteristics?

To address this, we exploit local structural properties of the graphs that are preserved under isomorphisms and attach them to vertices in an attempt to filter the mapping candidates based on them. The first filtering criterion applied is the **degree** of each vertex, since a vertex $u \in S$ can only be mapped to a vertex $v \in G$ if $\deg(u) \leq \deg(v)$. Otherwise, it would be impossible to preserve all adjacencies in the mapping. Next, we refine our filtering by considering the surroundings of each vertex. In a similar way to the first filter, we use the **sum of the degrees** of each vertex's neighbors. If the sum of the neighbor degrees for u in S exceeds that for v in G , u cannot be mapped to v , as the local neighborhood structure would not be preserved. Finally, we incorporate geometric structures into our filtering process, such as the **number of triangles** in which each vertex participates. Since the presence of cyclic substructures is preserved under isomorphisms, calculating and comparing these values provides yet another mean to eliminate infeasible mappings.

Each one of the mentioned vertex characteristics collectively forms what we define as the *fingerprint* of a vertex. This fingerprint serves as a descriptor of the local structure around each node. Formally, we define the fingerprint of a vertex as follows.

Definition: let P be a graph and $v \in V_P$ a vertex of P . The *fingerprint* of v denoted as $fp(v)$ is the tuple containing the following information:

- **Degree:** $\deg(v) = |\{u \in V_P : (v, u) \in E_P\}|$.
- **Sum of neighbor's degrees:** $\sum_{v \in N(u)} \deg(v)$ where $N(u)$ denotes the neighbors of u .
- **Adjacent triangles:** $\sum_{\substack{v, w \in N(u) \\ (v, w) \in E}} 1$.

Using the computed fingerprints, we filter the candidates for each vertex in S progressively, starting from the degree filter up to the adjacent triangles filter. This process can be formalized as follows.

Definition: Let $fp(u)$ and $fp(v)$ denote the fingerprints of vertex $u \in S$ and vertex $v \in G$ respectively, where each fingerprint is a tuple of structural invariants. We define the set of candidate mappings for u as:

$$C(u) = \{v \in V_G : \text{label}(u) = \text{label}(v) \wedge fp(u)_i \leq fp(v)_i \forall i\}$$

where $fp(u)_i$ and $fp(v)_i$ denote the i -th component of the fingerprint.

Therefore, only the vertices in G that satisfy all the conditions remain candidates for $u \in S$.

Once the candidate sets for each node in S have been computed, the proposed algorithm proceeds to order the vertices of S according to the *rarity* of their candidates, i.e., by the number of candidates available for each node. The *rarest* nodes are those with the fewest candidates and are placed first in the ordering. This strategy is motivated by the observation that vertices with distinctive structural properties, and thus, fewer potential matches in G , are more restrictive, and attempting to map them early in the search process can lead to finding earlier infeasible mappings, reducing then the overall search space. Formally, we define an ordering as follows.

Definition: let u_i be the i -th vertex in S and $C(u_i)$ denote the set of candidates for the vertex u_i . The ordering is defined as u_1, u_2, \dots, u_n the vertices of S such that $|C(u_1)| \leq |C(u_2)| \leq \dots \leq |C(u_n)|$

The algorithm proceeds to the core search phase. The first verification we must check is that every vertex in S has at least one candidate. Otherwise, there is no possible way to find a mapping. After the verification, in this stage,

it systematically explores possible mappings starting in the defined vertex order and using a recursive backtracking approach. The search attempts to assign to each vertex in S a feasible candidate with the current state of the mapping. The candidate is feasible under the current state of the mapping if

- The candidate hasn't been mapped previously, since the mapping is an injective function.
- For every vertex that has been mapped previously, the adjacencies under the mapping hold.

If no feasible extension is possible, the algorithm backtracks, undoing the last assignment in the mapping and trying the next candidate. This recursive search continues until either every vertex in S has been assigned a corresponding node in G or if all possibilities are exhausted. Notice that, contrary to the VF2++ implementation, candidate sets are computed only once at the start of the algorithm. A summary of the process is illustrated in the flowchart shown in Figure 3.

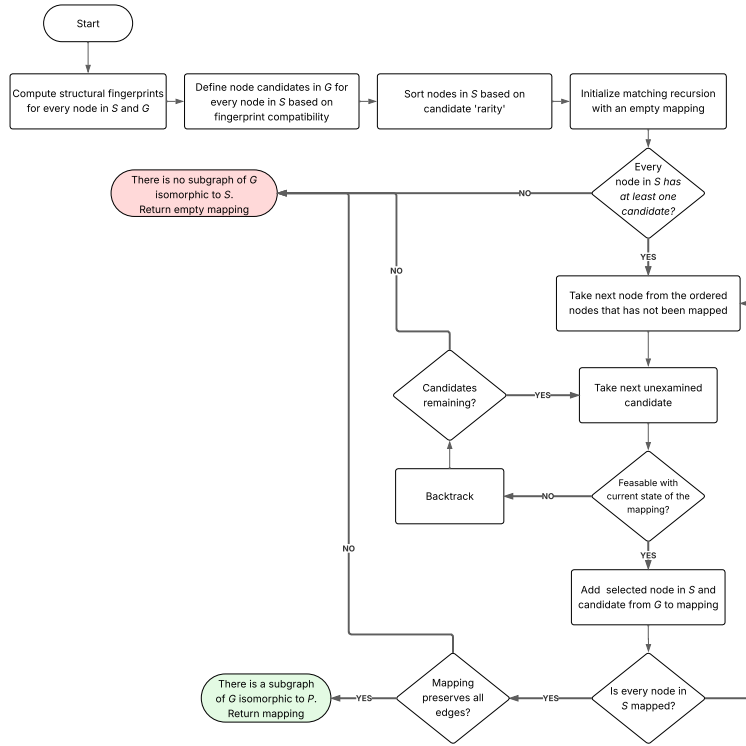


Figure 3: Diagram of the proposed algorithm.

4.1 Correctness and complexity analysis

4.1.1 Correctness

The correctness of the proposed algorithm for subgraph isomorphism relies on the systematic exploration of all feasible injective mappings from the pattern graph S to the target graph G , subject to both label and structural constraints. The candidate filtering phase ensures that only the vertices in G with similar structural fingerprints are compatible with a given node in S , and the characteristics are defined based on invariants under isomorphisms to ensure that no admissible isomorphism map is excluded. The core search procedure employs a backtracking strategy, recursively attempting to extend partial mappings, and at each step, the algorithm ensures injectivity and adjacency preservation. This ensures that the recursion traverses the entire space of admissible mappings, and that if a subgraph isomorphism exists, it will be found. The solution is also verified upon completion to guarantee that all constraints are satisfied.

4.1.2 Computational complexity

In terms of computational complexity, we divide the analysis in three main phases: fingerprint calculation, candidate filtering and backtracking search.

Fingerprint calculation: for each node in S and G , the algorithm computes its fingerprint, which involves:

- Computing the degree of each node: this is $O(1)$ per node, $O(|V_S| + |V_G|)$ total.
- Computing the sum of the degrees of neighbors: iterating over all edges, $O(|E_S| + |E_G|)$ total.
- Computing the number of triangles per node: for each node, this requires checking all pairs of neighbors, which is $O(d^2)$ per node, where d is the maximum degree in S or G .

Thus, the overall complexity for fingerprint calculation is $O((|V_S| + |V_G|)d^2)$, where d is the maximum degree in either graph.

Candidate filtering: for each node in S , the algorithm compares its fingerprint to every node in G to determine the candidate set. Each comparison is $O(1)$ since the fingerprint has a fixed number of features, so the total complexity is $O(|V_S| \cdot |V_G|)$.

Backtracking Search: The backtracking search analysis is identical to the analysis on VF2++. The search explores all injective mappings that are consistent with the candidate sets and adjacency constraints. In the worst case, if G has N nodes and S has n nodes, the number of possible injective mappings is $N!/(N - n)!$.

On the whole, the algorithm exhibits exponential worst-case complexity, which is expected for the NP-complete subgraph isomorphism problem. Nonetheless, practical runtime is improved by candidate filtering and vertex ordering based on fingerprint filters.

4.2 Fingerprint considerations

Structural fingerprints enable rapid elimination of incompatible candidates, thus reducing the computational burden of the search phase. An important note on these types of characteristics is the extensibility of them: depending on the application domain, additional or alternative structural features can be incorporated to capture more complex patterns. For example, in bioinformatics, fingerprints could be augmented to detect the presence of a specific functional group, such as benzene rings (6-cycle), allowing the algorithm to filter candidates based not only on simple graph invariants but also on chemically meaningful substructures. This adaptability ensures that the filtering process remains both relevant and efficient as the problem evolves.

5 Experimental results

5.1 Experimental Setup and Methodology

The conducted experiments aim to comprehensively compare three subgraph isomorphism algorithms: the VF2 algorithm implementation available in the python library NetworkX, our implementation of VF2++, and our proposed fingerprint-based backtracking solution. We must mention that the VF2++ algorithm implementation of NetworkX solves the induced subgraph isomorphism problem, which is, while closely related to the subgraph isomorphism problem (induced subgraph isomorphism implies subgraph isomorphism), not the same problem and has different mechanisms. However, the VF2 algorithm that solves the subgraph isomorphism problem and is a precursor of VF2++ is available and serves as a comparison standard for time and memory usage, as well as algorithm correctness. For evaluation, we employed three distinct datasets:

- **Mutag dataset:** Mutag is a widely-used bioinformatics dataset, that consist of 188 nitro compounds that are labeled regarding whether they have a mutagenic effect on a bacterium. The dataset represents chemical compounds as vertex-labeled graphs, where the vertex represents an atom and the edges are chemical bonds, making it ideal to test graph related algorithms[3]. A key characteristic of said dataset is the possibility of having multiple chemical bonds (up to 4 between any pair of atoms). We modeled this bonds by introducing additional intermediate nodes with a common label that connect the original vertices and leaves one single original edge, eliminating multiple edges. This modeling approach creates triangular structures in the graph, which provide a practical use of the proposed fingerprint in our algorithm.
- **Proteins dataset:** the Proteins dataset is also used as a benchmark in bioinformatics, consisting of 1113 protein structures. In this dataset, each protein is represented as a graph whose nodes correspond to amino acids, and the edges connect amino acids that are within 6 Angstroms of each other, capturing spatial proximity in the proteins's 3D structure [3]. Since each edge in these graphs is unique, the dataset is used directly as it is, without the need for additional preprocessing.

- **Random Dense dataset:** it is important to mention that the previously shown datasets contain sparse/medium sparse graphs. As the experiments intend to compare overall computational complexity, a synthetically generated set of unlabeled dense graphs was created to evaluate the performance under such challenging structural conditions. The dataset contains a set of 15 randomly generated dense graphs. These graphs are generated using a density parameter d in $[0, 1]$. In a graph of n nodes the maximum amount of edges is $n(n-1)/2$. When generating a random dense graph, we take a random float in $[0.95, 1]$ and call it d , then we generate $(n^d)(n^d - 1)/2$ random edges for the graph. Observe that for $d = 0$ we get 0 edges and for $d = 1$ we get all edges, so taking d from a distribution skewed towards 1 allows for the generation of random dense graphs with a specified asymptotic number of edges.

The general statistics of each dataset are described in Table 2

Table 2: Statistics of the datasets.

	Min Vertices	Min Edges	Max Vertices	Max Edges	Avg (SD) Vertices	Avg (SD) Edges	Avg (SD) Degree	Total Labels	Avg (SD) Labels
Mutag									
Small Sparse	10	20	28	66	17.9 (4.58)	39.6 (11.37)	4.38 (1.47)	7	3.12 (0.34)
Proteins									
Medium Sparse	4	5	620	1049	39.1 (45.76)	72.8 (84.60)	3.73 (0.88)	3	2.00 (0.37)
RANDOM									
Small Dense	9	33	11	54	10.0 (0.77)	42.5 (6.79)	8.45 (0.54)	1	1.00 (0.00)

Statistics of the number of vertices and number of edges. These describe the minimum, maximum and average number of vertices and edges in the dataset. Total Labels is the total number of labels in the dataset. Avg Label is the average number of labels per graph. Standard deviations are reported in parentheses.

The comparison focuses on three key performance metrics: **execution time**, **memory consumption** and the **number of recursive calls** or state explorations required to find isomorphisms. The last one is only computed for the VF2++ implementation and the proposed algorithm.

5.2 Results

We present the results for each dataset, where we ran all three algorithms on pairs of graphs from each collection. For every test case, we recorded the following metrics: the number of nodes and edges in both graphs S and G , the execution time, the peak memory usage, the number of recursive calls, and whether a mapping was found, to ensure consistency across implementations. After obtaining these results, we computed summary statistics for time and memory usage, as well as recursive calls, that include the mean, median, standard deviation, minimum, and maximum values. Additionally, we calculated the mapping success rate, which should be identical for all three algorithms. The results are shown in Tables 3, 4 and 5

Table 3: Performance comparison on the Mutag dataset

Algorithm	Time Mean (s)	Time Median (s)	Time Std (s)	Time Min (s)	Time Max (s)	Memory Mean (bytes)	Memory Std (bytes)	Memory Min (bytes)	Memory Max (bytes)	Recursive Calls Mean	Recursive Calls Std	Recursive Calls Min	Recursive Calls Max	Mapping Found Rate (%)
NetworkX VF2	0.0425	0.0095	0.2026	0.0000	4.3194	56674.4	13683.8	28104	108632	nan	nan	nan	nan	0.8
Our VF2++	0.6374	0.1978	3.5177	0.0000	76.5738	44415.7	8839.2	25472	74736	1208.68	7478.06	6	165277	0.8
Fingerprints + Backtracking	0.0094	0.0020	0.0600	0.0000	1.3136	58895.7	20087.9	25456	145984	197.68	1915.71	1	40152	0.8

Summary statistics for execution time (in seconds), memory usage (in bytes), and recursive calls across all algorithms on the Mutag dataset. The mapping found rate indicates the proportion of test cases for which a mapping was found.

Table 4: Performance comparison on the Proteins dataset

Algorithm	Time Mean (s)	Time Median (s)	Time Std (s)	Time Min (s)	Time Max (s)	Memory Mean (bytes)	Memory Std (bytes)	Memory Min (bytes)	Memory Max (bytes)	Recursive Calls Mean	Recursive Calls Std	Recursive Calls Min	Recursive Calls Max	Mapping Found Rate (%)
NetworkX VF2	0.0047	0.0021	0.0055	0.0000	0.0205	48812.0	19620.1	16736	118584	nan	nan	nan	nan	0.0
Our VF2++	0.0331	0.0151	0.1005	0.0000	1.0017	37407.0	14836.8	13512	86192	64.97	188.72	3	1853	0.0
Fingerprints + Backtracking	0.0039	0.0000	0.0062	0.0000	0.0319	53627.2	33119.3	12776	220472	12.00	83.15	1	990	0.0

Summary statistics for execution time (in seconds), memory usage (in bytes), and recursive calls across all algorithms on the Proteins dataset. The mapping found rate indicates the proportion of test cases for which a mapping was found.

Table 5: Performance comparison on the Random Dense dataset

Algorithm	Time Mean (s)	Time Median (s)	Time Std (s)	Time Min (s)	Time Max (s)	Memory Mean (bytes)	Memory Std (bytes)	Memory Min (bytes)	Memory Max (bytes)	Recursive Calls Mean	Recursive Calls Std	Recursive Calls Min	Recursive Calls Max	Mapping Found Rate (%)
NetworkX VF2	3.43067	0.0132194	10.854	0	80.7468	25606.1	4527.44	18680	37728	nan	nan	nan	nan	18.1818
Our VF2++	2.56039	0.00558436	8.24226	0	42.2448	28221.3	5466.26	22280	54992	13914.1	44102	10	205511	38.8889
Fingerprints + Backtracking	0.0440438	0	0.290268	0	2.68829	45676.5	5643.64	35119	60143	2181.28	13897.6	1	126260	38.8889

Summary statistics for execution time (in seconds), memory usage (in bytes), recursive calls, and mapping success rate on the Random Dense dataset.

5.3 Analysis of results

The results presented in the tables provide a detailed performance comparison, and several key observations and explanations can be drawn from them. First of all, on the Mutag dataset, the fingerprints and backtracking approach demonstrate the fastest average median execution times, with a low standard deviation indicating that most executions run on a similar time. The VF2++ implementation exhibits a greater execution time with a large standard deviation, pointing towards a significant variance of times depending on the graphs and indicating that minimum and maximum times are possibly not outliers. This results could be due to lacking additional optimizations on top of the time taken to compute in every recursion the matching order. However, memory-wise, the VF2 algorithm in the library NetworkX exhibits the lowest values. As expected, in this category, the fingerprint computation takes a significant amount of memory compared to the other algorithms. In terms of recursive calls, the fingerprint approach shows significantly lower values compared to the implementation of VF2++, which could indicate that the candidate filtering is indeed effectively reducing the search space. Finally, the mapping found rate is indicating the correctness of the algorithms, empirically confirming that the different pruning and filtering strategies do not exclude valid isomorphisms. This results were expected given the preprocessing of the dataset.

Surprisingly, on the Proteins dataset, the results are very similar to the ones on Mutag. It was expected that, since no modifications were made to the edges of the graphs, fingerprint structures wouldn't be as efficient as the "look-ahead" method on VF2++. Nonetheless, the results show a very similar time execution between VF2 and our proposed algorithms. Taking into account the data on recursive calls, we can confirm that the strategy to filter candidates using fingerprints is very effective. Finally, the performance on the Random Dense dataset shows an improvement comparing VF2 and the implementation of the discussed algorithms, with a great time reduction using fingerprints. Notice also that the mean on recursive calls compared to the other datasets is much greater, indicating that the size of the search space is much greater in dense graphs, even if the amount of vertices is not as big as the ones in Mutag or Proteins. We can also notice in the results that the mapping found rate between VF2++, fingerprints, and VF2 are not the same. Upon further investigation, we examined some of the graphs with dissimilarities and found that the mappings produced by our implementations were, in fact, correct. However, to determine the cause, the VF2 algorithm and the NetworkX implementation must be studied, since the implementation of VF2 could additionally define timeouts or specific undiscussed heuristics.

Overall, the experiments show that the integration of fingerprint-based candidate filtering offers a promising trade-off between execution time and use of memory, specially in denser graphs. While our VF2++ implementation improves VF2 in dense cases, it still suffers from high variance in execution time and recursive depth, probably due to lack of further optimizations. The inconsistency in the mapping found rate for the Random Dense dataset could be due to poor performance of VF2 in dense graphs and additional constraints like timeouts on the implementation. The code of the project can be found in this repository.

References

- [1] George Chin, Sutanay Choudhury, John Feo, and Lawrence Holder. Predicting and detecting emerging cyberattack patterns using streamworks. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, CISR '14, page 93–96, New York, NY, USA, 2014. Association for Computing Machinery.
- [2] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.
- [3] Saiful Islam, Md. Nahid Hasan, and Pitambar Khanra. A structural feature-based approach for comprehensive graph classification, 2024.
- [4] Alpár Jüttner and Péter Madarasi. Vf2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018. Computational Advances in Combinatorial Optimization.
- [5] Sebastian Keller, Pauli Miettinen, and Olga V. Kalinina. Frequent subgraph mining for biologically meaningful structural motifs. *bioRxiv*, 2020.