FASTiso: Fast Algorithm on Search state Tree for subgraph ISOmorphism in graphs of any size and density

Wilfried Agbeto[ID] , Camille Coti[ID] , Vladimir Reinharz[ID]

***Abstract*—Subgraph isomorphism is a combinatorial problem that involves finding one or all occurrences of a pattern graph within a target graph. Subgraph isomorphism has numerous applications in fields such as biology, chemistry, social network analysis, and pattern recognition. Although subgraph isomorphism is generally NP-complete, there are polynomial algorithms for special cases of graphs and efficient heuristics for the general case. However, the increasing size and complexity of graphs datasets require continuous improvements to existing methods.**

In this article, we present FASTiso, a novel subgraph isomorphism algorithm for any type of graphs with new heuristics for efficient exploration and pruning of the search state tree. We compared FASTiso to an ensemble of state-of-the-art solvers (VF3, VF3l, RI, Glasgow) on 7 synthetic and real datasets from the literature. The results show a significant reduction in the execution time of FASTiso compared to VF3 and other state-of-the-art algorithms. We also demonstrate that FASTiso is particularly efficient on large and dense graphs, and that it performs better than other solvers on different datasets. In addition to its execution speed, FASTiso has a smaller memory footprint than almost all other algorithms as only RI uses less memory.

The FASTiso code is available at https://gitlab.info.uqam.ca/cbe/fastiso as a C++ implementation, a python module, and integrated in a NetworkX fork. Our implementations support graphs and multigraphs, directed or not, and can include labels on nodes, edges, or both.

***Index Terms*—Graphs, graph matching, graph isomorphism, subgraph isomorphism, monomorphism**

I. Introduction

Graphs are present in many fields, such as biology, chemistry, bioinformatics, social networks, databases, and knowledge discovery. They are used to model data for better analysis and description, which often involves searching for one or all occurrences of a smaller graph (pattern graph) within a larger graph (target graph). This problem is known in the literature as the subgraph isomorphism problem. For example, in biology, molecular structures such as RNA can be represented as graphs, where nodes represent nucleotides and edges represent interactions between them. RNA exhibits a highly modular structure with essential motifs that influence the three-dimensional configuration of the molecule and, consequently, define its role in the cell. Understanding these modules primarily requires identifying subgraphs with specific topologies, thus involving the use of subgraph isomorphism algorithms [6], [30], [36], [2]. In chemistry, subgraph isomorphism is used to identify similar chemical structures within large molecular databases, which can be useful for drug design and the prediction of chemical reactivity. Some tools [17], [40] used in bioinformatics or for code analysis incorporate subgraph isomorphism solvers, so the performance of these tools depends on the efficiency of the solver used. The subgraph isomorphism is also widely used in Pattern Recognition [25], [18], [39].

Subgraph isomorphism is an NP-complete problem [14]. Due to this complexity, even though subgraph isomorphism has numerous and varied applications, finding efficient solutions remains challenging. However, it may be possible to reduce the computation time in average cases. Thus, heuristics to increase efficiency rely on [5]: (1) The selection of an optimal node ordering for traversing the pattern graph during the search (Variable Ordering). However, to date, no universally effective ordering method has been proposed for all cases. (2) The use of pruning or inference heuristics to quickly demonstrate that a partial solution cannot lead to a valid solution. However, using these heuristics involves trade-offs: the more effective a heuristic, the more computationally costly it can be. Additionally, a heuristic may be beneficial for certain types of graphs but less so for others. Consequently, no algorithm in the literature consistently outperforms others across all types of graphs, underscoring the need for generic algorithms. Such algorithms should be able to adjust to different types of graphs to ensure high performance across various application contexts. Other factors, such as the size and/or density of a graph, can also influence the complexity of subgraph isomorphism. Some algorithms [12], [11] have been specifically designed for large (containing more than 1000 nodes) and dense ($density > 0.1$) graphs, but the growing datasets of large and/or dense graphs require continuous improvements to existing algorithms.

In this paper, we propose FASTiso, a subgraph isomorphism algorithm with a new strategy to compute the variable ordering, as well as two new heuristics that are low-cost in terms of runtime and highly effective on large and/or dense graphs. Thanks to better data structure utilization, FASTiso also provides an improvement in its memory use.

FASTiso was compared to VF3 and other state-of-the-art solvers (VF3l [9], RI [7], Glasgow [28]) on 7 well-known

W. Agbeto and V. Reinharz are with Université du Québec à Montréal
C. Coti is with École de Technologie Supérieure
This work is funded by an NSERC Discovery, and an FRQS Chercheur-Boursier Junior 1.

and widely used datasets: MIVIA LDG [12], proposed by the authors of VF3. Scalefree [33], a synthetic dataset of scale-free graphs. SI_dataset [33], consisting of bounded-valence graphs, random graphs, and 4D meshes, grouped into four distinct subsets. Images-PR15, Images-CVIU11, and Meshes [33], which contain graphs generated from segmented images and meshes modeling 3D objects. And finally, Phase [27], which consists of small graphs that are challenging to solve. The results show a significant reduction in execution time for FASTiso compared to VF3 and the other state-of-the-art algorithms, as well as FASTiso's ability to be highly generic.

The paper is structured as follows: Section II provides a review of previous works on subgraph isomorphism. In Section III, after formalizing the problem, we present the FASTiso algorithm in detail. Finally, Section IV provides a proof of correctness for FASTiso and presents the experimental results.

II. Related works

Existing approaches to solve the subgraph isomorphism problem can be grouped into three main categories: tree search, constraint programming, and graph indexing [12], [41].

Tree search methods define the subgraph isomorphism problem as a state space exploration problem, typically solved using a depth-first search technique with backtracking. Each state represents a partial solution, meaning a partial mapping between the nodes of the pattern graph and the target graph. The goal is to find valid solutions while minimizing the exploration of invalid states. These algorithms begin with an empty initial state and progressively add pairs of nodes that satisfy the subgraph isomorphism constraints until a feasible solution is found.

Pruning heuristics play a crucial role in the efficiency of a tree search-based algorithm. Some algorithms, such as VF3, incorporate lookahead functions after verifying that the current state satisfies the conditions of subgraph isomorphism. These functions assess whether the current state cannot lead to a valid solution, thereby reducing the number of explored states by avoiding partial solutions that are bound to fail.

These functions generally rely on the classification of nodes to detect inconsistencies. The analysis is usually performed locally, searching for inconsistencies among the neighbors of the node pairs being matched. However, the computational cost of a lookahead function is proportional to its ability to identify inconsistencies. When the graphs contain inconsistencies that the function cannot detect, the additional computational cost becomes unnecessary overhead. Therefore, a trade-off is necessary between the cost of the heuristic and its effectiveness in reducing the search space. For this reason, some algorithms, such as RI, do not use lookahead functions or use very lightweight ones and instead prioritize an optimal variable ordering.

It is important to have a variable ordering strategy to guide the search and eliminate unfruitful states in the early stages of exploration. This relies on consistency between the rules used to determine this order and those applied to prune the search space. The order in which the nodes of the target graph are selected to be matched to the nodes of the pattern graph, called the value ordering, can also be defined.

Ullmann's algorithm [37] was the first to adopt this approach, but it took more than 20 years before a truly efficient algorithm, namely VF2 [15], was proposed. Several improvements to VF2 have since been developed, such as VF2 Plus [13], VF2++ [22], and VF3 [12], [11]. Other notable algorithms include RI/RI-DS [7], VF3L [9], and L2G [3]. VF2 Plus and RI/RI-DS are particularly effective for biological data, but VF2 Plus is faster on large graphs. VF3 is better suited for large and dense graphs, and VF3L is an enhancement of VF3 that omits certain pruning heuristics that are less effective on large and dense graphs.

Subgraph isomorphism can also be solved using Constraint Satisfaction Programming (CSP), which generally involves assigning a value from the domain of pre-defined variables (the possible values that a variable can take) while respecting a set of constraints. The resolution of this problem often relies on inference techniques combined with search strategies using backtracking.

The constraints limit the valid combinations of values that subsets of variables can take. Thus, when a value is assigned to a variable, inference consists of propagating the constraints involving that variable to reduce the domains of other variables by removing values that are invalid or inconsistent with the propagated constraints.

In the case of subgraph isomorphism, the nodes of the pattern graph are treated as variables, while the nodes of the target graph form the domain. The constraints are those imposed by the conditions of subgraph isomorphism. Initially, before the search phase, a compatibility domain is calculated for all nodes of the pattern graph using filtering techniques based on the degree and label of the nodes. More advanced filtering techniques can also be used [24], [42], [26].

During the search, inference techniques are employed to propagate constraints and reduce the domains of the variables. These methods generally use more memory than tree search methods because they store compatibility domains and incompatibilities arising from constraint propagation to avoid re-evaluating them. Inference can be either global or local, but in both cases, unlike methods based on tree search, it enables the detection of inconsistencies involving nodes that are not neighbors of the pair of nodes currently being matched. However, this comes at the cost of increased computational effort. In the presence of graphs with symmetrical or specific structures, where constraints can significantly reduce the search space, constraint programming-based methods can be more efficient, whereas tree search-based methods may suffer from weaker prediction capabilities. However, there exist instances of the subgraph isomorphism problem that remain particularly challenging for both approaches [34], [27].

The first constraint programming-based algorithm to solve the subgraph isomorphism problem was proposed by McGregor [29]. Later algorithms such as nRF+ [23], ILF [42], LAD [32], FocusSearch [38], SND [4], and Glasgow [28] followed.

The graph indexing approach, on the other hand, is used to efficiently search large graph databases for those that contain

a given pattern. Generally, indexing methods build an index representing the structural and semantic characteristics of graphs. The index is used to filter target graphs by quickly eliminating those that cannot contain the searched pattern. Then, subgraph isomorphism constraints are verified on the remaining graphs.The efficiency of this method strongly depends on the quality of the index. A low-quality index can lead to a high false negative rate (rejecting graphs that contain the pattern) and a high false positive rate (accepting graphs that do not contain the pattern), the latter causing unnecessary comparisons during the subgraph isomorphism verification phase. However, a trade-off must be found between the quality of the index and its cost: the more precise and efficient an index is, the higher its construction cost.

Once the index is built, it can be reused for multiple searches, making these methods particularly effective for static databases or those that hardly change. Algorithms that use this method include GraphQL [21], QuickSI [31], SPath [44], GADDI [43], and TurboIso [20].

III. Method

A. Background

A graph G is defined by a set V of nodes and a set $E \subseteq V \times V$ of edges connecting these nodes. The order of a graph, denoted $|V|$, refers to the number of nodes, while its size, denoted $|E|$, corresponds to the number of edges.

A graph can be labeled if labels are assigned to its nodes or edges. The labeling functions $\lambda : V \rightarrow A_V$ and $\beta : E \rightarrow A_E$ respectively associate to each node and edge the corresponding labels.

The neighbors of a node u are the nodes to which it is connected by an edge. If multiple edges exist between the same pair of nodes, the graph is called a multigraph, and E is treated as a multiset. The set of neighbors of a node u is denoted $N(u)$, and its degree is given by $|N(u)|$.

A graph can be either directed or undirected. In an undirected graph, an edge $\{u, v\}$ represents a bidirectional relationship between nodes u and v. In a directed graph, an edge (u, v) represents a unidirectional relationship going from u to v. In this case, u is called the predecessor of v, and v is called the successor of u.

For the remainder of this article, the focus will be on simple directed graphs. However, the proposed approach remains applicable to simple undirected graphs as well as to both directed and undirected multigraphs.

Let $G = (V, E)$ be a pattern graph and $G' = (V', E')$ be a target graph. The subgraph isomorphism problem consists of finding an injective function $M : V \rightarrow V'$ that maps each node of G to a unique node of G' (condition 1-2 of the subgraph isomorphism). Moreover, for every pair of nodes u, v in G, and for every pair of corresponding nodes $u' = M(u), v' = M(v)$ in G', if an edge (u, v) exists in G, then there must be a corresponding edge (u', v') in G'; if an edge (u, v) does not exist in G, then there is no corresponding edge (u', v') in G' (condition 3-4 of the subgraph isomorphism). If the nodes or edges are labeled, then the labels must be preserved (condition 5-6 of the subgraph isomorphism). Thus, we have the following conditions:

1) $\forall u \in V, \exists u' \in V' : (u, u') \in M$
2) $\forall u, v \in V, u \neq v \Rightarrow u' \neq v'$
3) $\forall (u, v) \in E, \exists (u', v') \in E'$
4) $\forall u, v \in V, (u', v') \in E' \Rightarrow (u, v) \in E$
5) $\forall u \in V, \lambda_G(u) = \lambda_{G'}(u')$
6) $\forall (u, v) \in E, \beta_G(u, v) = \beta_{G'}(u', v')$

It is possible that edges in G' do not have a corresponding match in G; in this case, the subgraph isomorphism is referred to as a monomorphism [14] and can be defined by removing condition 4 above.

B. The foundation of FASTiso

FASTiso, like its predecessor VF3, is based on a tree search method that reformulates the subgraph isomorphism problem as an exploration of a state space. The set of possible states can be represented by a search state-space tree (SST) (Figure 1). Each node of this tree represents a possible match between a node u of G and a node $u' = M(u)$ of G'. The path from the root to a given node in this tree represents a partial mapping ($S_k = \big((u_0, M(u_0)), (u_1, M(u_1)), \ldots, (u_k, M(u_k))\big)$, with $k < |V|$) between G and G'. Using the terminology from VF3, a partial mapping (partial state) that satisfies the conditions of the subgraph isomorphism is called a consistent state. A consistent state that includes all the nodes of G ($k = |V|$) is called a goal state, which represents a solution to the problem. Similarly, a consistent state that can never lead to a goal state is a dead state.

Let S_c be a partial state, and define $V(S_c) = \{u_i \mid (u_i, M(u_i)) \in S_c\}$ as the set of nodes of G that are part of the partial state S_c. Also, define $V'(S_c) = \{M(u_i) \mid (u_i, M(u_i)) \in S_c\}$ as the set of nodes of G' that are part of the partial state S_c. The goal is to start from an initial state S_0 (the empty state $\emptyset$) and traverse the consistent states by progressively adding ordered pairs $(u_k, M(u_k))$ such that $u_k \notin V(S_c)$ and $M(u_k) \notin V'(S_c)$ until a solution or feasible solutions (goal states) are found.

One way to improve the algorithm's performance is to reduce the number of states visited by identifying dead states as early as possible. This can be achieved by selecting a good variable ordering and using pruning heuristics. A good pruning heuristic can significantly reduce the number of visited states. However, it is important to keep in mind that the execution time of the heuristic adds to the time spent on the visited states. Therefore, a trade-off is necessary between the heuristic's execution time and its effectiveness.

Algorithm 1 presents the foundation of FASTiso. The first step consists of computing an initial domain assignment for each node of the pattern graph, considering their label, degree, and the degree of their neighbors. We define a compatibility map C between the nodes u_i of the pattern graph and the nodes u'_j of the target graph, such that:

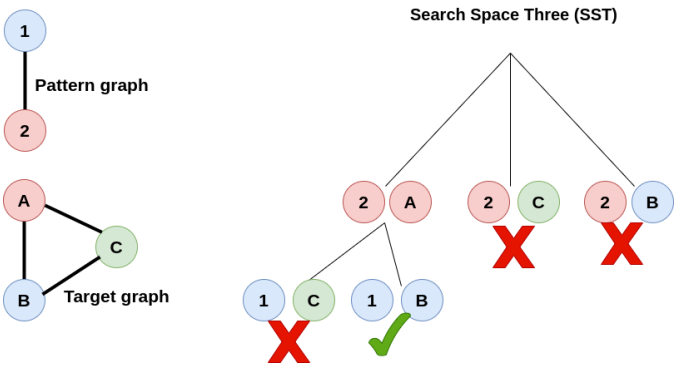


Fig. 1. **Search space tree**. The letters and numbers on the nodes represent their identifiers. The nodes of the pattern graph can only match with nodes of the target graph that have the same color.

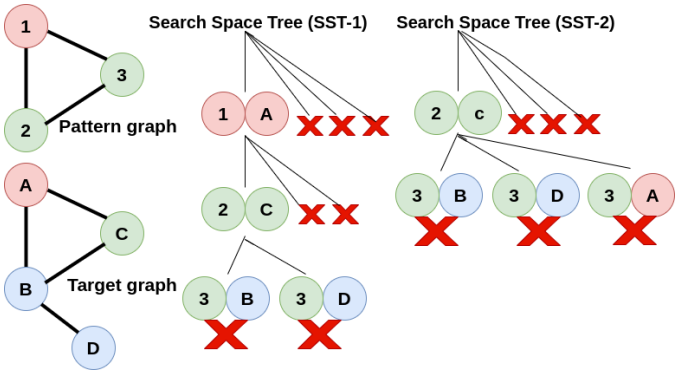


Fig. 2. The letters and numbers on the nodes represent their identifiers. The nodes of the pattern graph can only match with nodes of the target graph that have the same color.
SST-1 is the search space tree induced by the variable ordering 1-2-3, while SST-2 is the search space tree induced by the variable ordering 2-3-1.

$$C[u_i, u'_j] = \begin{cases} \text{true} & \text{if } \lambda(u_i) = \lambda(u'_j) \\ & \wedge\, |N(u_i)| \geq |N(u'_j)| \\ & \wedge \sum_{u_a \in N(u_i)} |N(u_a)| \geq \sum_{u_b \in N(u'_j)} |N(u_b)| \\ & \wedge \max_{u_a \in N(u_i)} |N(u_a)| \geq \max_{u_b \in N(u'_j)} |N(u_b)| \\ \text{false} & \text{otherwise.} \end{cases} \tag{1}$$

A node u'_j is compatible with a node u_i if the label of u_i is the same as that of u'_j, their degrees are identical, the sum of the degrees of u_i's neighbors is equal to that of u'_j, and the maximum degree among u_i's neighbors is the same as that of u'_j. We also define a list d_size that contains the domain size for each node u_i. Next, we compute the variable ordering (function computeOrdering, more details in Section III-C), and finally, we explore the SST using a depth-first search with backtracking to find the goal states (function match, more details in Section III-D).

Algorithm 1: FASTiso

Data: G, G'
1 C, d_size ← computeInitialDomains(G,G');
2 μ_G, parents, neighbs_dm ← computeOrdering(G, G', d_size); `// see Section III-C`
3 solutions ← {};
4 match(C, μ_G, S_0, parents, G, G', solutions); `// see Algorithm 2`
5 return solutions;

The improvement introduced by FASTiso compared to other tree search methods primarily lies in the variable ordering strategy and pruning heuristics. Regarding variable ordering, we added new rules to better distinguish nodes, enabling a more efficient selection of nodes in the pattern graph to prioritize during the search.

For the pruning heuristics, we introduced two new approaches to enhance the algorithm's efficiency.

(1) The verification of conditions 3 and 4 of subgraph isomorphism can be computationally expensive (see Section III-D). To address this, we incorporated a low-cost filtering heuristic, which avoids systematically evaluating these conditions for every node in the SST. (2) We proposed a lookahead function that enables faster detection of dead states.

C. Variable Ordering Strategy

One of the factors that significantly influence an algorithm's performance is the choice of a good variable ordering, which is the order in which the nodes of the pattern graph are traversed during the SST visit. For example, in Figure 1, we mapped node 2 before node 1, but we could have done the opposite. This section compares FASTiso's heuristic rules to the ones implemented in VF3 and RI. A summary is shown in Table I.

An overview of the different strategies for calculating the variable ordering can be found here [5]. Generally, there are two types of variable ordering: static variable ordering, where the order in which the pattern graph nodes are mapped is determined at the beginning and remains fixed during the SST visit, and dynamic variable ordering, which involves deciding, as the SST visit progresses, which node of the pattern graph to map.These two types of variable ordering can be computed based on information about the structure of the pattern graph and the target graph.

The variable ordering can be seen as a permutation of the nodes in the pattern graph. The goal is to find the permutation that allows to quickly determine the non-consistent or dead states. For example, it can be observed in Figure 2 that the variable ordering 2-3-1 is more efficient than the variable ordering 1-2-3, as it reaches the no solution state faster.

To achieve the aforementioned goal, the logic is to prioritize, i.e., place first in the variable ordering, the nodes of the pattern graph with the most verifiable constraints during the SST visit. This helps to quickly reduce the search space by eliminating invalid matches early on. For example, considering the condition 3-4 of subgraph isomorphism, nodes with the most already mapped neighbors can be prioritized.

Let μ_G be a variable ordering. Our strategy for computing μ_G is based on a combination of the approaches used by VF3 and RI, which we consider complementary while introducing

notable improvements. During our experiments with VF3, we observed that the selection of nodes to determine the variable ordering was often done randomly, due to the lack of additional rules to better discriminate the nodes. This led to poor performance on certain datasets. To address this drawback, we decided to combine the methods of VF3 and RI to achieve a more relevant and efficient variable ordering.

It is well-established that the selection of the first node to include in a variable ordering significantly impacts the quality of the resulting order [5]. Our strategy for determining the first node is based on two criteria: if the graph is unlabeled, we select the node with the highest degree; otherwise, we choose the node with the smallest compatibility domain.

The next nodes to be added to the partial variable ordering $\mu_G(k)$ are selected based on their neighbors and their compatibility domain. Let u_i be a candidate node to be added to $\mu_G(k)$, its neighbors $N(u_i)$ can be classified into three groups:

- $N_1(u_i) = \{u_j : u_j \in N(u_i) \text{ and } u_j \in \mu_G(k)\}$, the set of neighbors that are present in the partial ordering.
- $N_2(u_i) = \{u_j : u_j \in N(u_i) \text{ and } u_j \in N(\mu_G(k))\}$, the set of neighbors that are not in the partial ordering but have at least one neighbor that is.
- $N_3(u_i) = \{u_j : u_j \in N(u_i) \text{ and } u_j \notin N_1(u_i) \text{ and } u_j \notin N_2(u_i)\}$, the set of neighbors that do not belong to $N_1(u_i)$ or $N_2(u_i)$.

Let u_i, u_j be two candidate nodes to be added to $\mu_G(k)$ and $N_{\text{edges}}(u_i, S)$ the total number of incoming and outgoing edges between a node u_i and the node in the set S:

- u_i is selected if:
  Selection based on $N_1(u)$.
  - FASTiso: $|N_1(u_i)| > |N_1(u_j)|$.
  - VF3: not used.
  - RI: not used.
- In case of equality, u_i is selected if:
  Selection based on $N_1(u)$.
  - FASTiso: $N_{\text{edges}}(u_i, N_1(u_i)) > N_{\text{edges}}(u_j, N_1(u_j))$.
  - VF3: Same condition as in FASTiso.
  - RI: Same condition as in FASTiso.
- in case of equality, u_i is selected if:
  Selection based on $N_2(u)$.
  - FASTiso: $\sum_{u_a \in N_2(u_i)} |N_1(u_a)| > \sum_{u_b \in N_2(u_j)} |N_1(u_b)|$.
  - VF3: not used.
  - RI: $N_{\text{edges}}(u_i, N_2(u_i)) > N_{\text{edges}}(u_j, N_2(u_j))$.
- in case of equality, u_i is selected if:
  Selection based on $N_3(u)$.
  - FASTiso: $|N_3(u_i)| > |N_3(u_j)|$.
  - VF3: not used.
  - RI: $N_{\text{edges}}(u_i, N_3(u_i)) > N_{\text{edges}}(u_j, N_3(u_j))$.
- in case of equality, u_i is selected if:
  Selection based on target graph information.
  - FASTiso:
    * $d_{\text{size}}[u_i] > d_{\text{size}}[u_j]$, if G is unlabeled,
    * $d_{\text{size}}[u_i] < d_{\text{size}}[u_j]$, if G is labeled.
  - VF3: $P(u_i) < P(u_j)$, where $P(u)$ is the probability that the node u is compatible with nodes of G' based on its label, in-degree, and out-degree.
  - RI: not used.
- in case of equality, u_i is selected if:
  - FASTiso: $|N(u_i)| > |N(u_j)|$.
  - VF3: Same condition as in FASTiso.
  - RI: not used.
- in case of equality, a node is selected randomly.

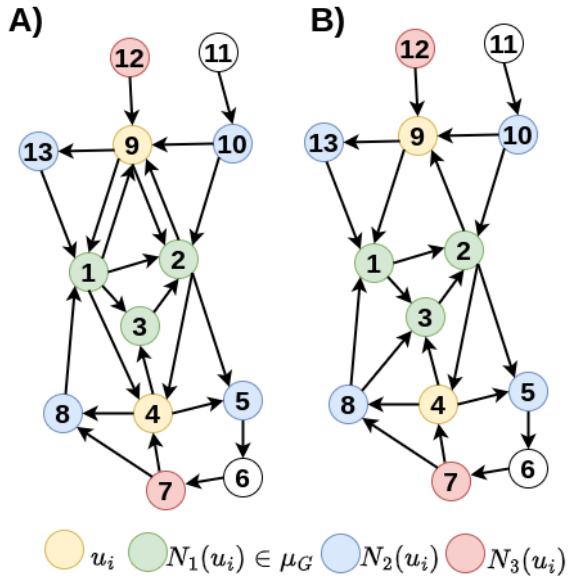


Fig. 3. Graphical representation of the 3 neighbor sets of 2 nodes in yellow (9 and 4) that are candidates to be added to $\mu_G(k)$.In Figure A), RI and VF3 will choose node 9, as it has more edges connecting it to mapped nodes, and FASTiso will choose node 4, as it has more neighbors that are mapped. On the other hand in Figure B), RI will opt for node 9 due to its higher number of edges connecting to the neighbors of already mapped nodes, and FASTiso will prefer node 4 because it has a greater number of neighbors whose neighbors are already mapped.

RI first prioritizes the node with the largest number of edges connecting it to mapped nodes or their neighbors, while VF3 first prioritizes the node with the largest number of edges connecting it to mapped nodes. In contrast, FASTiso first prioritizes the node with the most mapped neighbors or the node whose neighbors have the most mapped neighbors.

For example, in Figure 3-A), between node 9 and node 4, RI and VF3 will choose node 9 (as it has more edges connecting it to mapped nodes, a total of 4), whereas FASTiso will choose node 4 (as it has more neighbors that are mapped, a total of 3). Additionally, in the Figure 3-B), when using information based on $N_2(u_i)$ to decide between nodes 4 and 9, which cannot be distinguished using $N_1(u_i)$ information, RI will select node 9, as it has a greater number of edges pointing to the neighbors of the mapped nodes. Meanwhile, FASTiso will choose node 4, as it has more neighbors whose neighbors are mapped.

For each node u_i added to μ_G, we store its parent (the first neighbor that was added to μ_G) as well as $\max_{u_a \in N_2(u_i)} |N_1(u_a)|$. Let parents be the list of parents and

neighbs_dm be the list of $\max_{u_a \in N_2(u_i)} |N_1(u_a)|$ for each node u_i. We will see in the next section the usefulness of these lists.

D. The State-Space Tree (SST) Exploration

Starting from the initial state S_0 (the empty state $\emptyset$), each step involves selecting, at each level k of the SST, the candidate nodes u'_k in the target graph to match with the node u_k (the k-th node in the variable ordering μ_G).

The candidate nodes are chosen from among the neighbors of the node corresponding to the parent of u_k that are within the compatibility domain of u_k (i.e., $C[u_k, u'_k]$ = true, see Equation 1). As shown in Algorithm 2, Line 7-8, when u_k does not have a parent, the set of candidates is the set of all nodes in the target graph. This situation occurs either when u_k is the first node in the variable ordering or when the graphs have multiple connected components.

Algorithm 2: match: traverses the search state-space tree using a depth-first search with a backtracking mechanism. It starts from the empty state S_0 and adds pairs of nodes (u_k, u'_k) until a solution is reached.

Data: C,μ_G,parents,S_c,k,G,G',solutions
1 **if** $k = |V|$`// is goal state`
2 **then**
3 append(M(S_c),solutions);
4 **else**
5 $u_k \leftarrow \mu_G[k]$;
6 parent←parents[k];
7 **if** *parent is null* **then**
8 candidates← V'`// Candidates are all nodes in` G'
9 **else**
10 u'_p=M(parent);
11 candidates← N(u'_p) `// Candidates are the neighbors of` u'_p
12 **for** u'_k *in candidates* **do**
13 **if** $C[u_k, u'_k]$ *is True and isFeasible(*S_c,u_k,u'_k*)* **then**
14 $S_k \leftarrow$ addPair(S_c,u_k,u'_k);
15 match(C,μ_G,parents,S_k,k+1,G,G');
16 $S_c \leftarrow$ removePair(S_k,u_k,u'_k);

Before adding the pair (u_k, u'_k) to the state S_c (partial mapping), FASTiso uses feasibility rules (function isFeasible Algorithm 2, Line 13) to check whether this addition will result in a new state S_k that satisfies the constraints of subgraph isomorphism and predict whether it can not lead to a goal state.

The feasibility rules used by FASTiso rely on feasibility sets (defined below) as well as $V(S_c)$ and $V'(S_c)$, which respectively represent the set of nodes in the pattern graph and the target graph that are already mapped.

For a node u, the feasibility sets are the three sets of classified neighbors on which the rules used for computing the variable ordering are based (Section III-C, as we have said, there must be some consistency between the rules used for computing the variable ordering and those used for pruning the search space):

- $N_1(u)$: the set of neighbors of u that are already mapped,
- $N_2(u)$: the set of neighbors of u that are not yet mapped but have neighbors that are already mapped,
- $N_3(u)$: the set of neighbors of u that belong neither to $N_1(u)$ nor $N_2(u)$.

VF3 also uses these feasibility sets, but among the neighbors of u, it distinguishes between predecessors and successors.

The functionm isFeasible, which verifies feasibility rules, is formally defined as follows:

$$\text{isFeasible}(S_c, u_k, u'_k) = F_c(S_c, u_k, u'_k) \wedge F_d(S_c, u_k, u'_k) \quad (2)$$

The component F_c ensures that, before adding any pair (u_k, u'_k) to S_c, the resulting state satisfies the subgraph isomorphism conditions. The component F_d analyzes the neighbors of u_k and u'_k to detect potential inconsistencies and identify dead states. The remainder of this section is dedicated to the formalization of these components. Variants for graph isomorphism and monomorphism are also presented at the end of the section.

Component $\mathbf{F_c}$: Verify consistent states.

For every pair (u_k, u'_k) to be added to S_c, $F_c(S_c, u_k, u'_k)$ checks the following conditions:

1) $u'_k \notin V'(S_c)$, verify condition 2 of the subgraph isomorphism.
2) $N_1(u_k) = N_1(u'_k)$,
3) $N_{\text{in-edges}}(u_k, N_1(u_k)) = N_{\text{in-edges}}(u'_k, N_1(u'_k))$ and $N_{\text{out-edges}}(u_k, N_1(u(k))) = N_{\text{out-edges}}(u'_k, N_1(u'_k))$,
4) For every $\{u_k, u_a\} \in E$ where $u_a \in V(S_c)$, there exists $\{u'_k, M(u_a)\} \in E'$ and $\beta_G(u_k, u_a) = \beta_{G'}(u'_k, M(u_a))$, verify conditions 3, 4, and 6 of the subgraph isomorphism.

The conditions 1 and 4 of $F_c(S_c, u_k, u'_k)$ are sufficient to verify the subgraph isomorphism requirements. Condition 1 ensures that u'_k has not already been mapped to another node of the pattern graph. Condition 4 verifies structural consistency: if the edges between u_k and $V(S_c)$ have the same labels as the edges between u'_k and $V'(S_c)$. Additionally, if an edge exists between u_k and a node $u_a \in V(S_c)$, there must also be an edge between u'_k and the corresponding node $M(u_a) \in V'(S_c)$ (subgraph isomorphism condition 3). Conversely, if an edge exists between u'_k and a node $u_b \in V'(S_c)$, there must also be an edge between u_k and $M^{-1}(u_b) \in V(S_c)$ (subgraph isomorphism condition 4). The notation $\{u, v\}$ for undirected edges is used to denote both incoming and outgoing edges.

It is worth noting that condition 4 of subgraph isomorphism is not explicitly verified due to conditions 2 and 3 of $F_c(S_c, u_k, u'_k)$, which ensure that the number of neighbors, incoming edges, and outgoing edges of u_k in $V(S_c)$ are equal

Feature	FASTiso	RI	VF3
Type	Static	Static	Semi-static
Information used	Based on the structure of the pattern and target graph	Based on the structure of the pattern graph	Based on the structure of the pattern and target graph
Description	Choose by priority order: (1) The node with the most mapped neighbors or the node whose neighbors have the most mapped neighbors. (2) The node with the smallest domain if the graph is labeled, and if the graph is unlabeled, the node with the largest domain. (3) The node with the highest degree	Prioritizes the node with the largest number of edges or their neighbors	Choose by priority order: (1) The node with the largest number of edges. (2) The node with the lowest probability of finding a node in the target graph with the same label, and an equal or higher in-degree and out-degree. (3) The node with the highest degree

TABLE I
COMPARISON OF VARIABLE ORDERING STRATEGIES: FASTISO, RI, AND VF3.

to those of u'_k in $V'(S_c)$. VF3 does not incorporate condition 2, so it explicitly evaluates the conditions 3 and 4 of the subgraph isomorphism.

Checking the presence of an edge can be computationally expensive (especially for dense graphs or during frequent evaluations). For a node u, this check requires $\Theta(N(u))$ using an adjacency list. If the data is sorted, it can be done in $\Theta(\log(N(u)))$. Of course, if an adjacency matrix is used, the check can be performed in $\Theta(1)$. However, like VF3 and RI, FASTiso uses an adjacency list for memory efficiency, the verification of condition 4 of $F_c(S_c, u_k, u'_k)$ will therefore be done in $\Theta(N(u_k) \cdot \log(N(u'_k)))$.

Conditions 2 and 3 of $F_c(S_c, u_k, u'_k)$ serve as filtering tests to avoid evaluating condition 4 of $F_c(S_c, u_k, u'_k)$ at each node of the SST, which can lead to substantial computational costs.

Condition 5 of subgraph isomorphism is not verified in $F_c(S_c, u_k, u'_k)$ because it is already verified during the computation of compatibility domains (see Section III-B).

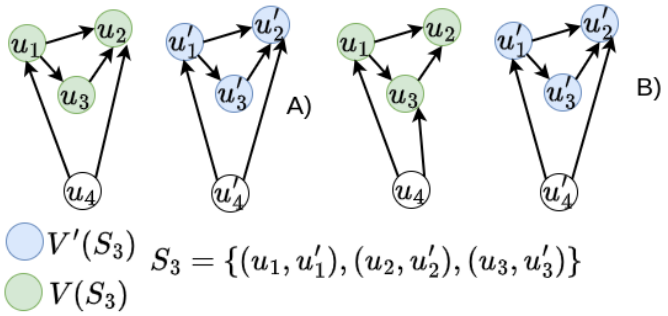


Fig. 4. Graphical representation of the sets $V(S_3)$ and $V'(S_3)$ when adding a pair of vertices (u_4, u'_4) to state S_3. In Figure A-), the addition results in a new coherent state, whereas in Figure B-), the addition results in an incoherent state.

In Figure 4-A), we can see that adding the pair (u_4, u'_4) will result in a new coherent state. The addition satisfies the four conditions of $F_c(S_c, u_k, u'_k)$; u_4 and u'_4 have the same number of neighbors, incoming edges, and outgoing edges in $V(S_3)$ and $V'(S_3)$, respectively. Furthermore, the edges (u_4, u_1) and (u_4, u_2) correspond to (u'_4, u'_1) and (u'_4, u'_2), respectively.

On the other hand, in Figure 4-B), adding the pair does not result in a new coherent state. While conditions 1, 2, and 3 of $F_c(S_c, u_k, u'_k)$ are satisfied, condition 4 is not. Specifically, the edge (u_4, u_3) does not correspond to the edge (u'_4, u'_2) because $M(u_3) = u'_3 \neq u'_2$. To strengthen the filtering tests (conditions 2 and 3), additional information about the identity of the nodes is introduced to detect such cases and avoid evaluating condition 4 as much as possible.

Suppose the nodes are also assigned an integer identifier corresponding to their index, e.g., $\mathrm{ID}(u_1) = 1$. We define a new condition:

$$\sum_{u_j \in N_1(u_k)} \mathrm{ID}(u_j) = \sum_{u'_j \in N_1(u'_k)} \mathrm{ID}(M^{-1}(u'_j)) \tag{3}$$

which verifies for every new pair (u_k, u'_k) whether the sum of the identifiers of u_k's neighbors already mapped equals the sum of the identifiers of the correspondents of the already mapped neighbors of u'_k.

In Figure 4-B), this condition (where $1 + 3 \neq 1 + 2$) makes it possible to avoid evaluating condition 4 of $F_c(S_c, u_k, u'_k)$. Using concatenation instead of a sum is more efficient (since two sums of different values can produce the same result), but we chose the sum because comparing integers is generally faster than comparing strings.

The use of filtering tests to avoid verifying condition 4 of $F_c(S_c, u_k, u'_k)$ as much as possible will only be effective if these tests have a complexity lower than that of verifying condition 4. We will present the implementation details that allow these filtering tests to be checked in constant time $\Theta(1)$.

To manage the sets $V(S_c)$ and $V'(S_c)$, two lists are used, structured as follows: $M[\mathrm{ID}(u_k)] = u'_k$ if u_k is mapped to u'_k, and null otherwise (see Algorithm 3, Line 1-2). These structures make it possible to find the correspondent of a node and verify the membership of this node in $V(S_c)$ and $V'(S_c)$ in $\Theta(1)$.

Filtering tests are also performed in $\Theta(1)$ using lists that store, for each node, the relevant information required for filtering: the number of mapped neighbors, the number of incoming edges with mapped neighbors, the number of outgoing edges with mapped neighbors, and the sum of the identifiers of the mapped neighbors.

Since FASTiso uses a static variable ordering, at each level k of the SST corresponds the same node of the pattern graph $u_k = \mu(k)$. These lists are therefore precomputed before the matching process for the nodes of the pattern graph. For

the nodes of the target graph, these lists are updated when a pair (u_k, u'_k) is added or removed, in $\Theta(|N(u'_k)|)$ (see Algorithm 3).

Algorithm 3: addPair: Updates the current state S_c by adding the pair (u_k, u'_k). The mapping structures and filtering information for neighboring nodes are updated accordingly. The function removePair(S_c,u_k,u'_k) performs the inverse operation of addPair. It decrements the corresponding values in S_c and sets $M[\text{ID}(u_k)] = M'[\text{ID}(u'_k)] = \texttt{null}$

Data: S_c,u_k,u'_k
$M[ID(u_k)] \leftarrow u'_k$;
$M'[ID(u'_k)] \leftarrow u_k$;
for u'_j *in* $N(u'_k)$ **do**
  S_c.num_mapped_neighbors[$ID(u'_j)$]+=1;
  S_c.num_in_mapp[$ID(u'_j)$]+=$N_{\text{in-edges}}(u'_k, \{u'_j\})$;
  S_c.num_out_mapp[$ID(u'_j)$]+=$N_{\text{in-edges}}(u'_k, \{u'_j\})$;
  S_c.sum_ids[$ID(u'_j)$]+=$ID(u_k)$
return S_c;

Component $\mathbf{F_d(S_c, u_k, u'_k)}$: Verify dead states. $F_d(S_c, u_k, u'_k)$ uses lookahead functions that are necessary but not sufficient conditions. When these conditions are false, they guarantee that adding the pair (u_k, u'_k) to S_c will not lead to a goal state.

For every pair (u_k, u'_k) to be added to S_c, $F_d(S_c, u_k, u'_k)$ checks the following conditions:

1) $|N_2(u_k)| \leq |N_2(u'_k)|$
2) $|N_3(u_k)| \leq |N_3(u'_k)|$

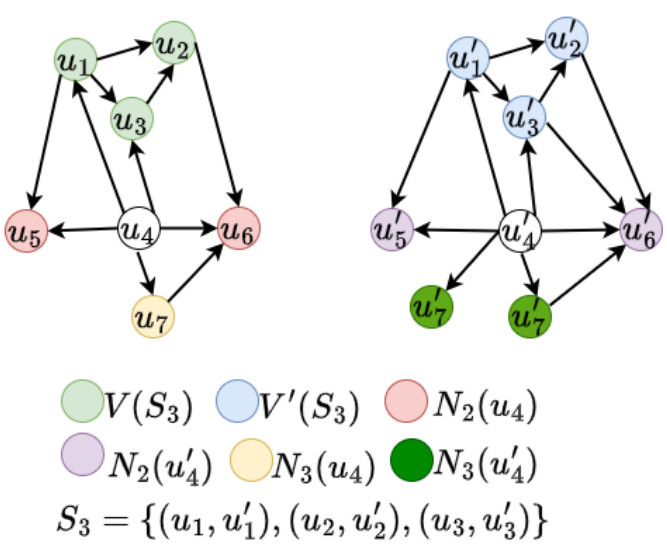


Fig. 5. Adding the pair (u_4, u'_4) to the state S_3 will result in a new valid state, but it is a dead state because all the nodes in $N_2(u_4)$ cannot be match with distinct nodes in $N_2(u'_4)$.

It is easy to verify from Figure 5 that adding the pair (u_4, u'_4) to S_3 will produce a new state that satisfies the conditions of $F_d(S_c, u_k, u'_k)$. However, we can observe that there is no solution between the two graphs. Indeed, all nodes in $N_2(u_4)$ cannot be match with distinct nodes in $N_2(u'_4)$, since the two nodes in $N_2(u_4)$ each have only one mapped neighbor, whereas $N_2(u'_4)$ contains only one node with only one mapped neighbor.

The function F_d can therefore be strengthened by partitioning the nodes to detect such inconsistencies. This methodology is already used by VF3, where nodes are partitioned based on their labels. As a result, this partitioning approach can only be applied to labeled graphs. In our case, we use a classification function that, for a node u, partitions $N_2(u)$ based on the number of their neighbors that are mapped.

For a node u, the sets of its classified neighbors are defined as:

$$N_2^i(u) = \{v \in N_2(u) : |N_1(v)| = i\}, i > 0.$$

For example, the set $N_2^1(u)$ represents all the neighbors of u that are not yet mapped but have exactly one mapped neighbor.

For a node u, there can be at most $\max_{v \in N_2(u)} |N_1(v)|$ classes. This value is recorded in the list neighbs_dm (see Section III-C) for the nodes of the pattern graph. For every pair (u_k, u'_k) to be added to S_c, we introduce a new condition to F_d based on classified neighbor sets. The condition is:

$$\forall 1 < i < \text{neighbs_dm}[k], \quad |N_2^i(u_k)| \leq |N_2^i(u'_k)|. \quad (4)$$

If u'_k has j classes and u_k has i classes with $j > i$, there is no need to verify classes $i+1, i+2, \ldots, j$, as the cardinality of these classes for u_k is zero.

Applying this new condition to the graphs in Figure 5 makes it possible to reject the addition of the pair (u_4, u'_4) to S_3, as $|N_2^1(u_4)| = 2 > |N_2^1(u'_4)| = 1$.

To manage the classified neighbor sets for a node u, a list L of size $t = \max_{v \in N_2(u)} |N_1(v)|$ is used, where the value at an index $i > 0$ corresponds to $|N_2^i(u)|$, and the value at index $i = 0$ corresponds to $|N_3(u)|$. This configuration allows for an easy calculation of $|N_2(u)| = |N(u)| - |N_1(u)| - |N_3(u)|$ and the verification of the last condition of $F_d(S_c, u_k, u'_k)$ in $\Theta(t)$. For nodes u_k in the pattern graph, L is precomputed before the matching process. However, for a node u'_k in the target graph, L is computed during the addition of the pair (u_k, u'_k) to S_c in the function $F_d(S_c, u_k, u'_k)$. This computation is performed in $\Theta(N(u'_k))$ by iterating over the neighbors u'_j of u'_k and incrementing $L[|N_1(u'_j)|]$ by 1.

We have also proposed versions for graph isomorphism and monomorphism (non-induced subgraph isomorphism) for isFeasible(S_c, u_k, u'_k).

For every pair (u_k, u'_k) to be added to S_c:

- For graph isomorphism, $F_c(S_c, u_k, u'_k)$ verifies the same conditions as for subgraph isomorphism. Additionally, $F_d(S_c, u_k, u'_k)$ verifies the following conditions:

1) $|N_2(u_k)| = |N_2(u'_k)|$,
2) $|N_3(u_k)| = |N_3(u'_k)|$,
3) For all $1 < i <$ neighbs_dm$[k]$, $|N_2^i(u_k)| = |N_2^i(u'_k)|$.

- For monomorphism, $F_c(S_c, u_k, u'_k)$ verifies the following conditions:

1) $u'_k \notin V'(S_c)$,
2) $N_1(u_k) \leq N_1(u'_k)$,
3) $N_{\text{in-edges}}(u_k, N_1(u_k)) \leq N_{\text{in-edges}}(u'_k, N_1(u'_k))$ and $N_{\text{out-edges}}(u_k, N_1(u_k)) \leq N_{\text{out-edges}}(u'_k, N_1(u'_k))$,
4) For every $\{u_k, u_a\} \in E$ where $u_a \in V(S_c)$, there exists $\{u'_k, M(u_a)\} \in E'$ and $\beta_G(u_k, u_a) = \beta_{G'}(u'_k, M(u_a))$.

Additionally, $F_d(S_c, u_k, u'_k)$ verifies the following conditions:

1) $|N_2(u_k)| \leq |N_2(u'_k)|$,
2) $\max_{u_j \in N_2(u_k)} |N_1(u_j)| \leq \max_{u'_j \in N_2(u'_k)} |N_1(u'_j)|$,
3) $\sum_{u_j \in N_2(u_k)} |N_1(u_j)| \leq \sum_{u'_j \in N_2(u'_k)} |N_1(u'_j)|$.

E. FASTisoM

To evaluate the contribution of the pruning strategy (Equations (3) and (4)) we also implemented FASTisoM, a lightweight version of FASTiso without these two heuristics. This version is particularly suited for cases where the additional pruning cost is not justified by efficiency gains. As discussed below, even in such cases, the difference between FASTisoM and FASTiso is rarely large due to the low cost of the new heuristics.

IV. Proofs of correctness and exhaustiveness

We will first show that any solution returned by FASTiso is an isomorphism, then that the lookahead performed by `isFeasible` to detect dead states should never reject a pair (u_k, u'_k) if its addition will result in a non-dead state, and finally that any valid isomorphism is returned by FASTiso.

Lemma 1: If FASTiso returns a solution, then it is valid.

Proof: By construction every time a matching (u_k, u'_k) is added to a partial solution, `isFeasible` checks whether the isomorphism properties are preserved (see Section III-D). Therefore any solution returned by FASTiso is necessarily valid.

Lemma 2: `isFeasible` never reject a pair (u_k, u'_k) if its addition will result in a non-dead state.

Proof: At a level k of the search tree, a node u_k can only be matched to a node u'_k if they have the same number of already mapped neighbors (this condition is necessary but not sufficient): $|N_1(u_k)| = |N_1(u'_k)|$.

We show that if $|N_1(u_k)| \neq |N_1(u'_k)|$ at a level l of the search tree, then u_k and u'_k can never be matched at a level $k > l$.

Suppose that u_k and u'_k are matched at level k, but at some level $l < k$ we had $|N_1(u_k)| \neq |N_1(u'_k)|$. For $|N_1(u_k)| = |N_1(u'_k)|$ to become true at level k, each new matching (u, u') added between levels l and $k-1$ must modify $N_1(u_k)$ and $N_1(u'_k)$ differently. This can only happen if a neighbor u of u_k is matched to a node v that is not a neighbor of u'_k, and vice versa, which contradicts the subgraph isomorphism constraints.

It follows that u_k and u'_k cannot be mapped together, contradicting our assumption. Thus, if $|N_1(u_k)| \neq |N_1(u'_k)|$ at level l, then they can never be matched at a level $k > l$.

The function isFeasible exploits this property to anticipate dead states. Specifically, for each pair (u_k, u'_k), it classifies the unmapped neighbors of u_k and u'_k based on the number of already mapped neighbors. Then, it checks whether a classified set of neighbors of u_k is larger than its equivalent for u'_k. If so, it is known that some neighbors of u_k will never find a correspondent at a level $l > k$ of the search tree.

Thus, the function `isFeasible` never reject a pair (u_k, u'_k) if its addition will result in a non-dead state.

Lemma 3: All valid solutions are returned by FASTiso

Proof: FASTiso uses exhaustive backtracking, ensuring that it explores all possible correspondences between G and G', subject to the conditions imposed by the function isFeasible (2).

Suppose that there exists a valid isomorphism M that is not found by FASTiso. This implies that at some level k of the search tree, the function isFeasible rejects a matching (u_k, u'_k) belonging to M. However, we know that the function `isFeasible` never reject a pair (u_k, u'_k) if its addition will result in a non-dead state (Lemma 2). This leads to a contradiction, proving that FASTiso necessarily returns all valid solutions.

V. Results

To evaluate the performance of FASTiso, we compared it with VF3, VF3L, RI, and Glasgow. We selected these solvers because they currently represent leading methods for solving the subgraph isomorphism problem [34], [8]. VF3, VF3L, and RI are also based on tree search, while Glasgow is based on constraint programming. The graph indexing approach is generally used to search large graph databases for those that contain a given pattern without returning their position. Therefore, we ignore their implementatations in our experiments. All of these solvers, like FASTiso, are implemented in C++.

A. Datasets

For a meaningful evaluation, we selected well-known datasets that have been used at least once to evaluate the chosen solvers, where they have demonstrated the best performance. Indeed, the complexity of subgraph isomorphism means that no solver is superior to the others in all cases.

- **MIVIA LDG [9]:** MIVIA Large and Dense Graphs consists of large, dense random Erdős–Rényi graphs. It includes both labeled (with uniform and non-uniform distributions) and unlabeled graphs, with densities of 0.2, 0.3, and 0.4. The target graphs contain between 300 and 10,000 nodes, while the motif graphs contain between 200 and 2,000 nodes (representing 20% of the corresponding target graphs). For most motif-target graph pairs, there is only one solution for the subgraph isomorphism.
- **Scalefree [33]:** A dataset representing scale-free networks, randomly generated using a power-law degree distribution. The target graphs contain between 200 and 1,000 nodes, and the motif graphs have 90% of the nodes of the corresponding target graph. This dataset includes

Feature	FASTiso	RI	VF3
Selection of candidates to map with a node u_k from the pattern graph	Based on the neighbors of the corresponding node of the first mapped neighbor of u_k	Based on the neighbors of the corresponding node of the first mapped neighbor of u_k	Based on a restriction of feasability sets
Feasability rules	Based on feasibility sets (without distinguishing between predecessors and successors of nodes) and a classification of nodes based on the number of mapped neighbors	Based on adjacency relations (only considering isomorphism conditions, without any lookahead heuristic)	Based on feasibility sets (distinguishing between predecessors and successors of the nodes) and a classification of nodes based on their labels

TABLE II
COMPARISON OF THE MAPPING PROCESS: FASTISO, RI, AND VF3.

certain motif-target graph pairs for which there is no solution.

- **Phase [27]:** This dataset consist of randomly generated graphs (200 graph pairs) selected to be near the satisfiable-unsatisfiable phase transition. The motif graphs have 30 nodes, while the target graphs have 150 nodes. All motif-target graph pairs have no solution. For our experiments, we selected the first 50 graph pairs.
- **SI_dataset [33]:** This dataset consists of graphs of different types: bounded-valence graphs, random graphs, and 4D meshes. The target graphs contain between 200 and 1,296 nodes, and the motif graphs contain 20 to 60% of the nodes of the corresponding target graph. These graphs are sourced from the MIVIA dataset [1].
- **Images-PR15 [33], [35]:** This dataset represents graphs generated from segmented images. The target graph consists of 4,838 nodes, while the 24 motif graphs range from 4 to 170 nodes.
- **Images-CVIU11 [33], [16]:** This dataset also represents graphs generated from segmented images. The motif graphs have between 15 and 151 nodes, while the target graphs contain between 1,072 and 5,972 nodes.
- **Meshes [33], [16]:** This dataset represents graphs generated from meshes modeling 3D objects. The motif graphs contain between 40 and 199 nodes, while the target graphs have between 201 and 5,873 nodes. All the motif-target graph pairs in this dataset also have no solution.

In our experiments, we measured both the matching time and memory usage to obtain all solutions for the subgraph isomorphism problem. An experiment was also conducted for graph isomorphism using the NetworkX [19] version of FASTiso on randomly generated graphs. (see supplementary file).

The experiments were conducted on a server equipped with an Intel(R) Xeon(R) Gold 6342 @ 2.80GHz processor with 54 cores, running in 64-bit mode, and featuring 1.7 MB of L1 cache, 216 MB of L2 cache, 864 MB of L3 cache, and 2 TB of RAM. The system was configured in NUMA mode with all 54 CPU cores assigned to a single NUMA node, ensuring optimized and balanced memory access for maximum performance.

Firstly, we compare FASTiso, VF3, and VF3L on the MIVIA LDG dataset. The goal is to demonstrate the efficiency of FASTiso compared to VF3/VF3L on the dataset originally used to evaluate VF3. Generally, FASTiso outperforms VF3 and VF3L in all cases, except for graphs labeled with a non-uniform distribution, where on the task of find all solutions VF3L performs better.

B. Benchmarks (MIVIA LDG): find all solutions

We tested unlabeled graphs, graphs labeled according to a uniform distribution, and graphs labeled with a non-uniform distribution for all densities d (0.2, 0.3, 0.4). The average matching times for finding all solutions and for finding one solution for $d = 0.3$ is hown in Figure 6, all additional results are provided in the supplementary material. For the results related to finding all solutions on unlabeled graphs, we can observe that as the number of nodes increases or the density increases, FASTiso performs better compared to VF3 and VF3L. The advantage of FASTiso becomes significant for $d = 0.2$ and $N = 3500$ (FASTiso is, on average, approximately 14.6 minutes faster than VF3L and 25.5 minutes faster than VF3), $d = 0.3$ and $N = 2000$ (FASTiso is, on average, approximately 19.5 minutes faster than VF3L and 32.95 minutes faster than VF3), $d = 0.4$ and $N = 1000$ (FASTiso is, on average, approximately 15.3 minutes faster than VF3L and 23.93 minutes faster than VF3), where N represents the number of nodes.

For $d = 0.2$ and $N = 5000$, the average time saved compared to VF3L and VF3 is 35.5 minutes (1.91 times faster) and 49.13 minutes (2.27 times faster), respectively. For $d = 0.3$ and $N = 3000$, the average time saved compared to VF3L and VF3 is 3 hours (2.87 times faster) and 4 hours (3.34 times faster), respectively. For $d = 0.4$ and $N = 1500$, the average time saved compared to VF3L and VF3 is 4 hours (3.03 times faster) and 5 hours (3.82 times faster), respectively.

For graphs labeled with a uniform distribution and $d = 0.2$, FASTiso and VF3L have very similar performance but are faster than VF3. VF3L is faster for $N \leq 6500$, while FASTiso becomes faster for $N \geq 7000$. For $d = 0.3$ and $d = 0.4$, FASTiso is the fastest, although its performance is close to VF3L for smaller values of N. The gain of FASTiso over VF3L becomes significant for $d = 0.4$ and $N = 6500$, at around 10 minutes on average (2.17 times faster), increasing to approximately 15 minutes for $d = 0.4$ and $N = 7000$ (2.16 times faster).

For graphs labeled with a non-uniform distribution, VF3L is the fastest, followed by FASTiso. The maximum average gain of VF3L over FASTiso is approximately 2 minutes (1.29 times faster), observed for $d = 0.4$ and $N = 6500$.

C. Benchmarks (MIVIA LDG): find one solution

Regarding the results for finding one solution, FASTiso is the fastest in all cases, followed by VF3L.

For unlabeled graphs, FASTiso is consistently at least twice as fast as VF3 and VF3L, regardless of size N or density d. The maximum acceleration is achieved for $N = 500$ and $d = 0.4$, where FASTiso is 19.17 times faster than VF3L and 25.53 times faster than VF3.

For $d = 0.2$, the highest gain is observed at $N = 4500$, with an average runtime of 4.6 minutes, making FASTiso 4.19 times faster than VF3L, and 7.63 minutes, or 6.30 times faster than VF3. For $d = 0.3$, the maximum gain occurs at $N = 3000$, with an average runtime of 24.28 minutes (4.88 times faster than VF3L) and 29.76 minutes (5.75 times faster than VF3). Finally, for $d = 0.4$, FASTiso achieves its maximum gain at $N = 1500$, reducing the average runtime to 24.08 minutes (4 times faster than VF3L) and 33.35 minutes (5.15 times faster than VF3).

For labeled graphs with either uniform or non-uniform distributions, FASTiso outperforms VF3L and VF3 by up to 2 orders of magnitude for certain sizes.

FASTiso also maintains a low memory consumption. With a 17% reduction in memory usage of FASTiso compared to VF3/VF3L. Detailed results on memory consumption are presented in the supplementary material.

D. Benchmarks: the other datasets

In the following experiments, we compare FASTiso, VF3, VF3L, RI, and Glasgow on a variety of instances. We also include FASTisoM in the tests, which will allow us to assess the effectiveness of the newly introduced pruning heuristics.

We set a time limit of 5 hours for the execution of each instance. If, for a given instance, more than half of the solvers manage to solve it within the allotted time, we increase the time limit to 10 days to allow the other solvers to solve it.

The Table III shows the results for the scalefree dataset, which consists of both sparse and dense graphs. Scalefree is known to be very challenging for VF3 compared to constraint programming-based algorithms ([8], [10]). As shown in the table, FASTiso achieves the best results on this dataset, followed by Glasgow which itself is faster by several orders of magnitude compared to FASTisoM, VF3, VF3L, and RI.

All solvers have similar execution times for most instances. FASTiso and Glasgow widen the gap only on a few instances, which are dense graphs.

These results are also confirmed for the RAND dataset (see Table IV), which consists of random graphs, both sparse and dense, derived from the SI_dataset. FASTiso is the fastest for all graph sizes; it is twice as fast as FASTisoM and Glasgow, and outperforms RI, VF3L, and VF3 by one to two orders of magnitude for graph sizes 200 and 600.

For the phase dataset (see Table V), which contains very dense graphs of small sizes, Glasgow is the fastest, followed by FASTiso. Glasgow outperforms FASTiso by one order of magnitude, while FASTiso is twice as fast as FASTisoM and four times faster than VF3L, VF3, and RI.

For the sake of space, the results for the other datasets, consisting of sparse graphs, are presented in the supplementary material. In general, Glasgow is the best on one dataset, while FASTisoM and FASTiso perform the best on the rest of the datasets.

E. Benchmarks global results

In summary, FASTiso is always better than VF3, with improvements ranging from 1 to 5 orders of magnitude across certain datasets. These results demonstrate the effectiveness of the variable ordering strategy and the new pruning heuristics introduced by FASTiso. This is confirmed by the performance of FASTiso compared to FASTisoM. Indeed, FASTiso is almost always at least twice as fast as FASTisoM, achieving up to 4 orders of magnitude on the scalefree dataset, which proves the effectiveness of the new pruning heuristics. Moreover, the fact that FASTisoM is always faster than VF3 confirms the effectiveness of the variable ordering strategy used by FASTiso/FASTisoM.

The superiority of FASTiso over FASTisoM is mainly observed on large and dense graphs. On the other hand, for smaller and sparse graphs, FASTisoM is slightly faster (see supplementary material). This can be explained by the fact that the new anticipation heuristic is particularly effective on large and dense graphs. Indeed, the larger or denser the graph, the more it contains classes, increasing the chances of detecting inconsistencies between nodes. In contrast, for smaller or sparser graphs, the number of classes is reduced, decreasing the opportunities for detecting inconsistencies between nodes.

Glasgow performs better than FASTiso on graphs with significant dissimilarities (such as the phase and meshes datasets, where instances have no solution), where its sophisticated filtering techniques and constraint propagation techniques give it a distinct advantage. However, in other datasets (RAND, BVG, M4D, M4DR, Images-PR15, Images-CVIU11, Scalefree), FASTiso is faster.

FASTiso is faster than VF3L on all datasets except the graphs labeled with a non-uniform distribution from the MIVIA LDG dataset, and faster than RI in all cases.

VI. Conclusion

In this paper, we proposed FASTiso, a subgraph isomorphism algorithm with new heuristics for efficient exploration and pruning of the search state tree. We show that FASTiso is adapted to a wide range of graphs and is particularly efficient for large or dense graphs. The performance of FASTiso was evaluated against VF3, FASTisoM (a version of FASTiso without the new pruning heuristics), VF3L, RI, and Glasgow on 7 standard and varied benchmark datasets. The results show that FASTiso outperforms VF3 on all instances, sometimes by several orders of magnitude, and competes on specialized datasets with other state-of-the-art solvers.

The experimental results suggest that FASTiso is particularly well-suited for large and dense graphs, while FASTisoM presents an interesting alternative for smaller or sparser graphs. However, in situations where an optimal choice cannot be made, FASTiso appears to be the most versatile solution, consistently ranking among the top solvers tested (either as the best or the second-best).

TABLE III
COMPARISON ON THE **SCALEFREE** DATASET. TIME IS IN SECOND, #MEM (MEAN MEMORY USAGE IN KB).

solver	Min	25%	50%	75%	Max	Mean	Sum	#Mem
FASTiso	**1E-06**	8.90E-05	**2.68E-04**	**5.32E-04**	**6.67E-03**	**4.01E-04**	**4.01E-02**	7274.70
Glasgow	1E-06	**1E-06**	2E-03	6.25E-03	2.10E-02	5.89E-03	5.89E-01	10790.96
FASTisoM	1E-06	2.03E-04	5.83E-04	1.46E-03	2.25E+03	2.64E+01	2.64E+03	7249.10
VF3	3.04E-04	1.25E-03	3.39E-03	5.30E-02	7.68E+03	1.61E+02	1.61E+04	7508.00
VF3L	1.88E-04	9.75E-04	4.40E-03	9.29E-02	3.83E+05	7.64E+03	7.64E+05	7456.60
RI	1E-04	4.39E-04	1.91E-03	1.29E-00	6.45E+05	1.27E+04	1.27E+06	**6761.16**

TABLE IV
COMPARISON ON THE **RAND** DATASET. TIME IS IN SECOND, #MEM (MEAN MEMORY USAGE IN KB).

Target size	Solver	Min	25%	50%	75%	Max	Mean	Sum	#Mem
200	**FASTiso**	7.10E-05	9.25E-04	**3.63E-03**	**1.34E-02**	**0.375**	**2.91E-02**	**1.746**	6689.66
	FASTisoM	7.70E-05	1.13E-03	5.02E-03	3.44E-02	0.867	6.70E-02	4.021	6669.33
	Glasgow	**1.00E-06**	**7.50E-04**	5.00E-03	9.52E-02	0.951	8.90E-02	5.345	8530.93
	RI	6.00E-05	2.49E-03	3.01E-02	0.342	2.810	0.358	21.509	**6370**.00
	VF3L	7.60E-05	6.68E-03	7.40E-02	0.421	2.737	0.415	24.951	6712.40
	VF3	1.32E-04	3.58E-03	4.53E-02	0.350	4.435	0.487	29.243	6720.40
400	**FASTiso**	5.33E-04	**9.47E-03**	**0.122**	**1.010**	358.435	**8.119**	**730.750**	7822.17
	FASTisoM	4.10E-04	1.12E-02	0.244	4.434	**307.252**	12.264	1103.760	7810.17
	Glasgow	**1.00E-06**	6.35E-02	0.331	6.561	687.019	14.300	1287.084	11709.06
	RI	5.60E-04	0.164	1.431	37.461	531.791	36.370	3273.385	**7052.00**
	VF3L	5.55E-04	0.226	1.967	33.866	515.917	40.612	3655.107	8147.11
	VF3	6.77E-04	0.216	2.056	51.057	1105.34	66.609	5994.840	8155.06
600	**FASTiso**	**3.75E-04**	**1.93E-02**	**0.237**	**4.025**	**578.340**	**16.226**	**1460.356**	9717.86
	FASTisoM	5.51E-04	3.00E-02	0.571	18.745	1273.170	39.075	3516.762	9700.00
	Glasgow	1.00E-03	1.97E-02	2.704	45.955	655.183	39.745	3577.115	18617.91
	RI	1.16E-03	0.174	5.809	170.789	6815.490	268.156	24134.087	**8181.37**
	VF3L	7.69E-04	1.090	8.127	226.381	740589.000	8528.661	767579.519	10565.51
	VF3	9.76E-04	0.851	9.625	314.234	800851.000	9326.362	839372.586	10573.22

TABLE V
COMPARISON ON THE **PHASE** DATASET. TIME IS IN SECOND, #MEM (MEAN MEMORY USAGE IN KB).

solver	Min	25%	50%	75%	Max	Mean	Sum	#Mem
Glasgow	**6.76**	**34.33**	**49.36**	**57.74**	**64.33**	**44.63**	**2231.59**	9197.20
FASTiso	12.72	210.84	707.50	1334.04	7598.06	1175.53	58776.58	7068.40
FASTisoM	24.33	400.50	1291.51	2932.09	16149.50	2417.50	120875.33	7056.40
VF3L	93.53	990.70	4352.05	7723.15	17182.40	5129.04	256452.06	7254.85
VF3	124.19	1084.07	4395.70	8576.57	15720.80	5364.50	268224.88	7266.00
RI	104.71	1398.17	5309.96	9017.92	14858.60	5593.71	290873.32	**6605.33**

Looking ahead, we plan to further improve FASTiso by exploring new pruning heuristics. We also intend to develop a parallel distributed version of the algorithm to handle very large graphs, containing millions or even billions of nodes.

REFERENCES

[1] MIVIA Lab: MIVIA dataset and MIVIA large dense graphs dataset. http://mivia.unisa.it/, 2017.

[2] W. Agbeto, C. Coti, and V. Reinharz. Parallel maximal common subgraphs with labels for molecular biology. *bioRxiv*, 2024.

[3] I. Almasri, X. Gao, and N. Fedoroff. Quick mining of isomorphic exact large patterns from large graphs. In *2014 IEEE International Conference on Data Mining Workshop*, pages 517–524, 2014.

[4] G. Audemard, C. Lecoutre, M. Samy-Modeliar, G. Goncalves, and D. Porumbel. Scoring-based neighborhood dominance for the subgraph isomorphism problem. In *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*, pages 125–141. Springer, 2014.

[5] V. Bonnici and R. Giugno. On the variable ordering in subgraph isomorphism algorithms. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 14(1):193–203, 2017.

[6] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14 Suppl 7:S13, 2013.

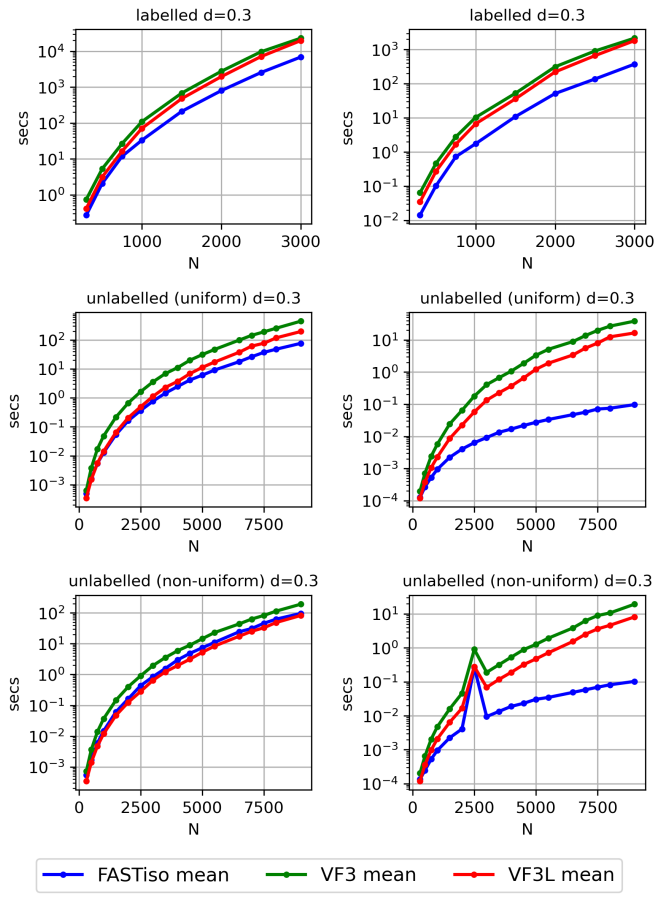
[7] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14:1–13, 04 2013.

[8] V. Carletti, P. Foggia, A. Greco, A. Saggese, and M. Vento. Comparing performance of graph matching algorithms on huge graphs. *Pattern Recognition Letters*, 134:58–67, 2020. Applications of Graph-based Techniques to Pattern Recognition.

[9] V. Carletti, P. Foggia, A. Greco, M. Vento, and V. Vigilante. VF3-Light: A lightweight subgraph isomorphism algorithm and its experimental evaluation. *Pattern Recognition Letters*, 125:591–596, 2019.

[10] V. Carletti, P. Foggia, G. Percannella, P. Ritrovato, and M. Vento. Two parallel versions of vf3: Performance analysis on a wide database of graphs. *Pattern Recognition Letters*, 146:150–157, 2021.

[11] V. Carletti, P. Foggia, A. Saggese, and M. Vento. Introducing VF3: A new algorithm for subgraph isomorphism. In Pasquale Foggia, Cheng-



On the left: The average time to find all solutions.

On the right: The time to find one solution.

Fig. 6. The average matching times for finding all solutions and one solution in the graphs from the NVIDIA LDG dataset with a density of $d = 0.3$. N is the number of nodes in the target graph.

Lin Liu, and Mario Vento, editors, *Graph-Based Representations in Pattern Recognition*, pages 128–139, Cham, 2017. Springer International Publishing.

[12] V. Carletti, P. Foggia, A. Saggese, and M. Vento. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):804–818, 2018.

[13] V. Carletti, P. Foggia, and M. Vento. VF2 Plus: an improved version of VF2 for biological graphs. In Cheng-Lin Liu, Bin Luo, Walter G. Kropatsch, and Jian Cheng, editors, *Graph-Based Representations in Pattern Recognition*, pages 168–177, Cham, 2015. Springer International Publishing.

[14] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03):265–298, 2004.

[15] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)Graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26:1367 – 1372, 11 2004.

[16] G. Damiand, C. Solnon, Colin de la Higuera, J.-C. Janodet, and É. Samuel. Polynomial algorithms for subisomorphism of nd open combinatorial maps. *Computer Vision and Image Understanding*, 115(7):996–1010, 2011. Special issue on Graph-Based Representations in Computer Vision.

[17] A. Ferro, R. Giugno, G. Pigola, A. Pulvirenti, D. Skripin, G. D. Bader, and D. Shasha. NetMatch: a cytoscape plugin for searching biological networks. *Bioinformatics*, 23(7):910–912, 02 2007.

[18] P. Foggia, G. Percannella, and M. Vento. Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01):1450001, 2014.

[19] A. Hagberg, P. Swart, and D. Chult. Exploring network structure, dynamics, and function using NetworkX. 06 2008.

[20] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2013.

[21] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418, 2008.

[22] A. Jüttner and P. Madarasi. VF2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018. Computational Advances in Combinatorial Optimization.

[23] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical structures in computer science*, 12(4):403–422, 2002.

[24] C. Lecoutre. Constraint Networks: techniques and algorithms 2009 hoboken ISTE. *Wiley*, 10:9780470611821.

[25] L. Livi and A. Rizzi. The graph matching problem. *Pattern Analysis and Applications*, 16(3):253–283, 2013.

[26] C. McCreesh and P. Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *International conference on principles and practice of constraint programming*, pages 295–312. Springer, 2015.

[27] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *Journal of Artificial Intelligence Research*, 61:723–759, 2018.

[28] C. McCreesh, P. Prosser, and J. Trimble. The Glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, volume 12150 of *Lecture Notes in Computer Science*, pages 316–324. Springer, 2020.

[29] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3):229–250, 1979.

[30] V. Reinharz, A. Soulé, E. Westhof, J. Waldispühl, and A. Denise. Mining for recurrent long-range interactions in RNA structures reveals embedded hierarchies in network families. *Nucleic AcidsResearch*, 46(8):3841–3851, 03 2018.

[31] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[32] C. Solnon. AllDifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12):850–864, 2010.

[33] C. Solnon. Solnon datasets. http://liris.cnrs.fr/csolnon/SIP.html, 2017.

[34] C. Solnon. Experimental evaluation of subgraph isomorphism solvers. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition*, pages 1–13, Cham, 2019. Springer International Publishing.

[35] C. Solnon, G. Damiand, Colin de la Higuera, and Janodet J.-C. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognition*, 48(2):302–316, 2015.

[36] A. Soulé, V. Reinharz, R. Sarrazin-Gendron, A. Denise, and J. Waldispühl. Finding recurrent RNA structural networks with fast maximal common subgraphs of edge-colored graphs. *PLOS Computational Biology*, 17(5):1–28, 05 2021.

[37] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[38] J. R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics (JEA)*, 15:1–1, 2011.

[39] M. Vento. A long trip in the charming world of graphs for Pattern Recognition. *Pattern Recognition*, 48(2):291–301, 2015.

[40] Y. Wang, Y. Lin, Z. Wan, and X. Yang. Task Context: a tool for predicting code context models for software development tasks. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 156–160, 2023.

[41] L. Wenle, D. Wenyong, and Y. Mengting. SLF: A passive parallelization of subgraph isomorphism. *Information Sciences*, 623:900–914, 2023.

[42] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15:327–353, 2010.

[43] S. Zhang, S. Li, and J. Yang. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th international conference on extending database technology: advances in database technology*, pages 192–203, 2009.

[44] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.