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 fundamental combinatorial problem that involves finding one or more occurrences of a pattern graph within a target graph. It arises in a wide range of application domains, including biology, chemistry, social network analysis, and pattern recognition. Although subgraph isomorphism is NP-complete in the general case, many exact algorithms allow it to be solved in practice on many instances. However, the increasing size and structural diversity of graph datasets continue to pose significant challenges in terms of robustness and scalability.**

In this article, we propose FASTiso, an exact subgraph isomorphism algorithm that emphasizes a strong consistency between the variable ordering strategy and the pruning rules used during search. This design enables a unified exploitation of structural information throughout the exploration process, leading to improved efficiency and stable performance across heterogeneous graph structures. An extensive experimental evaluation on widely used synthetic and real-world benchmarks shows that FASTiso consistently outperforms reference solvers such as VF3, VF3L, and RI, and achieves competitive performance compared to constraint programming–based approaches (Glasgow, PathLad+), while outperforming them on most datasets. The results further demonstrate that FASTiso remains highly efficient on small instances and scales well to large graphs, while maintaining a lower memory footprint than most evaluated solvers. The peak memory usage is 7.74 GB for FASTiso, 36.19 GB for PathLad+, over 500 GB for Glasgow, 9.62 GB for VF3/VF3L, and 4.31 GB for RI.

FASTiso code is available at https://gitlab.info.uqam.ca/cbe/fastiso as a C++ implementation, a Python module, and an integration within an extended version of NetworkX. The implementations support simple graphs and multigraphs, directed or undirected, with 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 pattern recognition. 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 [10], [46], [56], [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 [22], [63] 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 [39], [23], [61], semantic search [66], RDF query processing [31], [28], symbol recognition [40], and community detection [12].

Subgraph isomorphism is an NP-complete problem [19]. Due to this complexity, even though subgraph isomorphism has numerous and varied applications, finding efficient solutions remains challenging. Most exact algorithms rely on two key approaches to improve efficiency [9]. First, a variable ordering strategy is used to determine the order in which nodes of the pattern graph are matched during the search. This choice has a major impact on the size of the search tree, yet no universally effective ordering strategy has been proposed so far. Second, pruning or inference rules are employed to detect early that a partial mapping cannot be extended into a valid solution. While powerful pruning techniques can dramatically reduce the search space, they often come at a high computational cost and may be effective only for specific graph structures. As a result, no single algorithm consistently outperforms all others across heterogeneous datasets.

Additional factors, such as graph size and density, further influence the practical complexity of subgraph isomorphism. Several approaches [17], [16], [5] have been specifically designed for large graphs (with more than 1,000 nodes) or dense graphs (with density greater than 0.1). Nevertheless, the rapid growth of modern graph datasets, both in size and structural complexity, calls for continuous improvements to existing methods, particularly with respect to robustness and scalability.

Our contributions

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.

Approach	Community	Optimal	Algorithms
Tree Search	Artificial Intelligence/Bioinformatics	Yes	Ullman [59], VF2 [20], L2G [3], RI/RI-DS [11], VF2++ [29], VF3 [17], VF3L [14]
Constraint programming	Artificial Intelligence	Yes	Mc-Gregor [44], LAD [52], FocusSearch [60], PathLad [33], Glasgow [43], PathLad+ [62]
Graph indexing	Database	Yes	GraphQL [27], QuickSI [50], SPath [71] ,GADDI [70], CFL-Match [8], TurboIso [26], CECI [7], CFQL [57], DP-iso [24], BoostIso [47]
Learning-based (GNNs)	Artificial Intelligence	No	GNN [49], FGNN [34], Counting [38], Sub-GMN [35], ENDNet [51]

Fig. 1. **A summary of Main Approaches to Subgraph Isomorphism.**

Motivated by the need to improve the performance of exact subgraph isomorphism algorithms on heterogeneous datasets in terms of robustness and scalability, we propose FASTiso, a new exact algorithm built on the VF3 framework.

In this work, we argue that the robustness observed on diverse graph structures largely relies on a strong coherence between the heuristics used to compute the variable ordering and those employed to prune the search space. Based on this observation, FASTiso introduces a variable ordering strategy explicitly aligned with the pruning rules used to reduce the search space, ensuring that the same structural information guides both the exploration order and the elimination of unpromising search branches. Moreover, FASTiso introduces two new pruning rules that are both effective and computationally inexpensive. We integrate these contributions into an exact solver derived from VF3 while significantly improving robustness and scalability across diverse datasets.

FASTiso was compared to VF3 and other state-of-the-art solvers (VF3L [14], RI [11], Glasgow [43], and PathLad+ [62]) on widely used datasets, including both synthetic and real graphs. We also evaluated the scalability of FASTiso on very large graphs containing up to 23 million nodes. The results show that FASTiso significantly reduces matching time compared to VF3 and the other solvers while maintaining high generality and scalability. Further experimental analyses confirm that the proposed strategies strongly contribute to FASTiso's superior performance across diverse datasets.

The paper is structured as follows. Section II reviews previous work on subgraph isomorphism. In Section III, we first formalize the problem and then present the FASTiso algorithm. Finally, Section IV presents the experimental results.

II. Related works

Several exact algorithms have been developed over time to solve the subgraph isomorphism problem. Most of these approaches can be classified into three main categories: tree search, constraint programming, and graph indexing [17], [64].

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 rules play a crucial role in the efficiency of a tree search-based algorithm. Some algorithms, such as VF2, 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.

Generally, these functions 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 lookahead function 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 [59] was the first to adopt this approach, but it took more than 20 years before a truly efficient algorithm, namely VF2 [20], was proposed. Several improvements to VF2 have since been developed, such as VF2 Plus [18], VF2++ [29], and VF3 [17], [16]. Other notable algorithms include RI/RI-DS [11], RI+ [4], VF3L [14], 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 rules 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 [37], [69], [41].

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 [54], [42].

The first constraint programming-based algorithm to solve the subgraph isomorphism problem was proposed by McGregor [44]. Later algorithms such as nRF+ [36], ILF [69], LAD [52], PathLad [33], PathLad+ [62], FocusSearch [60], SND [6], and Glasgow [43] 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. Algorithms that use this method include GraphQL [27], QuickSI [50], SPath [71], GADDI [70], CFL-Match [8] TurboIso [26], CECI [7], DP-iso [24], and CFQL [57].

Other works have also addressed the non-induced subgraph isomorphism problem to reduce redundant exploration caused by symmetries in graphs. These approaches exploit symmetries to compress equivalence classes [25], [47], [45], [30], [68]. Portfolio-based approaches have also been proposed to dynamically select the best algorithm from a set of solvers for a given instance [33], [58]. Additionally, parallel computing strategies have been explored to accelerate the solution process [32], [15], [64].

Given the NP-completeness of the problem, approximate or inexact algorithms have also been proposed for scenarios where finding an optimal solution is not strictly required, and a sufficiently close solution is acceptable. These approaches include methods based on probabilistic relaxation [65], [67] as well as learning-based approaches using graph neural networks (GNNs) [49], [34], [38], [35], [51].

III. Method

A. Background

A graph G is defined by a set V of nodes and a set or multiset $E \subseteq V \times V$ 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 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 (arc) (u, v) represents a unidirectional relationship that goes from u to v. In this case, u is called the predecessor of v, and v is called the successor of u. A graph can also be labeled if labels are assigned to its nodes or edges. The labeling functions $\lambda : V \to A_V$ and $\beta : E \to A_E$, respectively, associate to each node and edge the corresponding labels.

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

The out-neighborhood, and in-neighborhood of a node u are defined as

$$N^+(u) = \{v \in V \mid \exists\, (u, v) \in E\},$$
$$N^-(u) = \{v \in V \mid \exists\, (v, u) \in E\}.$$

The (total) neighborhood of u is $N(u) = N^+(u) \cup N^-(u)$. Neighborhoods are defined as sets of distinct nodes and therefore ignore edge multiplicities. The out-degree of u, denoted $d^+(u)$, is defined as $d^+(u) = |\{(u, v) \in E \mid v \in V\}|$, that is, the number of outgoing edges incident to u, counting multiplicities. Similarly, the in-degree of u, denoted $\deg^-(u)$, is defined as $d^-(u) = |\{(v, u) \in E \mid v \in V\}|$. The total degree of u is then given by $d(u) = \deg^+(u) + \deg^-(u)$.

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 $M : V \to V'$ 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 (non-induced subgraph isomorphism) [19] 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) (see supplementary file, Figure F). 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 $(S_k^c = \big((u_0, M(u_0)), (u_1, M(u_1)), \ldots, (u_{k-1}, M(u_{k-1}))\big)$, with $k < |V|$) between G and G'. We define $V(S_k^c) = \{u_i \mid (u_i, M(u_i)) \in S_k^c\}$ and $V'(S_k^c) = \{M(u_i) \mid (u_i, M(u_i)) \in S_k^c\}$ as respectivly the set of nodes of G and G' that are part of the partial state S_k^c. The partial state S_k^c induces two subgraphs of G and G', defined respectively as

$$G[S_k^c] = (V(S_k^c), E(S_k^c)) \quad \text{and} \quad G'[S_k^c] = (V'(S_k^c), E'(S_k^c)),$$

where $E(S_k^c) = \{(u, v) \in E \mid u \in V(S_k^c) \wedge v \in V(S_k^c)\}$, and $E'(S_k^c) = \{(u', v') \in E' \mid u' \in V'(S_k^c) \wedge v' \in V'(S_k^c)\}$. Using the terminology of VF3, a partial mapping (partial state) S_k^c is said to be a consistent state if it satisfies all the conditions of the subgraph isomorphism problem, that is, if the induced subgraphs $G[S_k^c]$ and $G'[S_k^c]$ satisfy consditions 1-6 and are therefore isomorphic. A consistent state that includes all nodes of G (i.e., $k = |V|$) is called a goal state and represents a complete solution to the subgraph isomorphism problem. Conversely, a consistent state that cannot be extended to any goal state is called a dead state.

The Algorithm starts from an initial state S_0^0 (the empty state $\emptyset$) and traverses the consistent states by progressively adding ordered pairs $(u_k, M(u_k))$ such that $u_k \notin V(S_k^c)$ and $M(u_k) \notin V'(S_k^c)$ 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 rules.

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:

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

A node $u'_j \in V'$ is said to be compatible with a node $u_i \in V$ if u_i and u'_j have the same label, the degree of u_i is less than or equal to the degree of u'_j, the sum of the degrees of the neighbors of u_i is less than or equal to that of u'_j, and the maximum degree among the neighbors of u_i is less than or equal to 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_0^0, 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 rules. Regarding variable ordering, we introduced new rules that are strongly aligned with the pruning rules used to reduce the search space, allowing better node discrimination and a more effective prioritization of pattern vertices during the search.

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

(1) Verification of conditions 3 and 4 of the subgraph isomorphism can be computationally expensive (see Section III-D). To address this, we incorporated a low-cost filtering test, 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

An overview of the different strategies for calculating the variable ordering can be found here [9]. 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. 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.

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 [9]. 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:

(A) u_i is selected if: $|N_1(u_i)| > |N_1(u_j)|$.

(B) In case of equality, u_i is selected if: $N_{\text{edges}}(u_i, N_1(u_i)) > N_{\text{edges}}(u_j, N_1(u_j))$.

(C) in case of equality, u_i is selected if: $\sum_{u_a \in N_2(u_i)} |N_1(u_a)| > \sum_{u_b \in N_2(u_j)} |N_1(u_b)|$.

(D) in case of equality, u_i is selected if: $|N_3(u_i)| > |N_3(u_j)|$.

(E) in case of equality, u_i is selected if:

- $d_{\text{size}}[u_i] > d_{\text{size}}[u_j]$, if G is unlabeled,
- $d_{\text{size}}[u_i] < d_{\text{size}}[u_j]$, if G is labeled.

(F) in case of equality, u_i is selected if: $|N(u_i)| > |N(u_j)|$.

FASTiso 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. Figure A (see supplementary file) illustrates an example of the variable ordering computation for the graph on the left, using our strategy.

For each node u_i added to μ_G, we store its parent (its 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_0^0 (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.

Data: C,μ_G,parents,S_k^c,k,G,G',solutions

1 **if** $k = |V|$`// is goal state`
2 **then**
3 append($M(S_k^c)$,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_k^c,u_k,u'_k*)* **then**
14 $S_{k+1}^{c+1} \leftarrow$ addPair(S_k^c,u_k,u'_k);
15 match(C,μ_G,parents,S_{k+1}^{c+1},$k+1$,G,G');
16 $S_k^c \leftarrow$ removePair(S_{k+1}^{c+1},u_k,u'_k);

Before adding the pair (u_k, u'_k) to the state S_k^c (partial mapping), FASTiso uses feasibility rules (function isFeasible Algorithm 2, Line 13) to check whether this addition will result in a new state 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 $V(S_k^c)$ and $V'(S_k^c)$, 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, which are analogous to those used for computing the variable ordering (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):

- $N_1^M(u)$: the set of neighbors of u that are already mapped,

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

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:

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

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

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

1) $u'_k \notin V'(S_k^c)$, verify condition 2 of the subgraph isomorphism.
2) $N_1^M(u_k) = N_1^M(u'_k)$,
3) $N_{\text{in-edges}}(u_k, N_1^M(u_k)) = N_{\text{in-edges}}(u'_k, N_1^M(u'_k))$ and $N_{\text{out-edges}}(u_k, N_1^M(u(k))) = N_{\text{out-edges}}(u'_k, N_1^M(u'_k))$,
4) For every $\{u_k, u_a\} \in E$ where $u_a \in V(S_k^c)$, 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 $F_c(S_k^c, u_k, u'_k)$ are sufficient to verify the subgraph isomorphism requirements. It is worth noting that condition 4 of subgraph isomorphism is not explicitly verified due to conditions 2 and 3 of $F_c(S_k^c, u_k, u'_k)$, which ensure that the number of neighbors, incoming edges, and outgoing edges of u_k in $V(S_k^c)$ are equal to those of u'_k in $V'(S_k^c)$. 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 $F_c(S_k^c, u_k, u'_k)$ will therefore be done in $\Theta(N(u_k) \cdot \log(N(u'_k)))$.

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

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

To strengthen the filtering tests (conditions 2 and 3), additional information about the identity of the nodes is introduced to minimize the need to evaluate condition 4 as much as possible (see supplementary file, Figure B).

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

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

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.

The implementation details of the component F_c are described in the supplementary material (see Section C).

Component $\mathbf{F_d(S_k^c, u_k, u'_k)}$: Verify dead states. $F_d(S_k^c, u_k, u'_k)$ 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_k^c will not lead to a goal state.

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

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

The function F_d can be strengthened by partitioning the nodes to detect more inconsistencies (see supplementary file, Figure C). 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 $N_2^M(u)$ based on the number of their neighbors that are mapped.

For a node u, the sets of its classified neighbors are defined as: $N_2^{Mi}(u) = \{v \in N_2^M(u) : |N_1^M(v)| = i\}, i > 0$.

For example, the set $N_2^{M1}(u)$ 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 $\max_{v \in N_2^M(u)} |N_1^M(v)|$ 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_k^c, we introduce a new condition to F_d based on classified neighbor sets. The condition is:

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

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. This new condition is consistent with the condition (C) of our variable ordering.

The implementation details of the component F_d are described in the supplementary material (see Section D). We have also proposed versions for graph isomorphism and monomorphism (non-induced subgraph isomorphism) for isFeasible(S_k^c, u_k, u'_k) (see supplementary file, Section E).

IV. Results

To evaluate the performance of FASTiso, we compared it with VF3, VF3L, RI, Glasgow, and PathLad. We selected these solvers because they currently represent leading methods for solving the subgraph isomorphism problem [54], [13]. VF3, VF3L, and RI are also based on tree search, while Glasgow and PathLad are based on constraint programming. All of these solvers, like FASTiso, are implemented in C++. Most graph indexing approaches focus on the non-induced subgraph isomorphism problem and are therefore excluded from our experiments. Similarly, approaches based on graph neural networks (GNNs) are not considered, since they do not guarantee finding exact solutions.

A. Datasets

Subgraph isomorphism is an NP-complete problem whose difficulty depends less on the size of the graphs than on their structural properties. For this reason, we use in our experiments a diverse collection of datasets, including both real-world and synthetic data. Our datasets are divided into three benchmarks.

1) Classic benchmark [54]: The first benchmark is composed of relatively small graphs containing up to 10,000 nodes (see supplementary file, Figure E). These datasets include both sparse and dense graphs derived from real-world and synthetic sources.

- **MIVIA LDG [14]:** 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 [53]:** 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 certain motif-target graph pairs for which there is no solution.
- **Phase [42]:** 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 (RAND, BVG, M4D, M4DR) [53]:** 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 [53], [55]:** 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 [53], [21]:** 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 [53], [21]:** 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.

2) Graph-database benchmark [58]: These datasets contain eight real-world graphs (see Table I), which are used as target graphs. Although the graphs are labeled, we ignore these labels in our experiments. The authors of [58] generated the pattern graphs as follows. For each target graph G', pattern graphs are generated by randomly extracting connected subgraphs from G' using a random walk strategy. For each target graph, nine pattern sets are constructed, each containing 200 connected pattern graphs with the same number of nodes, resulting in a total of 1,800 pattern instances per target graph. The sizes of the pattern graphs range from 4 to 20 nodes (i.e., 4, 8, 12, 16, and 20) for the Human and WordNet graphs, and from 4 to 32 nodes (i.e., 4, 8, 16, 24, and 32) for the other graphs. Except for pattern graphs of size 4, the query sets are divided into dense and sparse categories, with four dense and four sparse query sets for each target graph.

TABLE I
ATTRIBUTES OF THE TARGET GRAPHS OF THE GRAPH-DATABASE BENCHMARK. $\bar{d}$ REPRESENTS THE AVERAGE DEGREE.

Category	Graph	#nodes	#edges	$\bar{d}$
Biology	Yeast	3,112	12,519	8.0
	Human	4,674	86,282	36.9
	HPRD	9,460	34,998	7.4
Lexical	WordNet	76,853	120,399	3.1
Citation	US-Patents	3,774,768	16,518,947	8.8
Social	Youtube	1,134,890	2,987,624	5.3
	DBLP	317,080	1,049,866	6.6
Web	Eu2005	862,664	16,138,468	37.4

3) Scalability benchmark [48]: This benchmark contains three target graphs (see Table II) and is used for scalability experiments. The graphs hugetrace-00020 and road-usa are originally directed. Since Glasgow and PathLad+ do not support directed graphs, we converted these graphs into undirected graphs by treating each directed edge between two nodes as a single undirected edge. For each target graph G', we generated pattern graphs by performing a random walk on G'. The generated pattern graphs are connected, and their sizes range from 9 to 499 nodes, covering 31 different sizes. For each size, we generated 50 pattern graph instances, resulting in a total of 1,550 instances per target graph.

TABLE II
ATTRIBUTES OF THE TARGET GRAPHS OF THE SCALABILITY BENCHMARK. $\bar{d}$ REPRESENTS THE AVERAGE DEGREE.

Category	Graph	#nodes	#edges	$\bar{d}$
Graph 500	scale21-ef16-adj	$\gtrsim$1M	31,731,650	51
DIMACS10	hugetrace-00020	$\gtrsim$16M	23,998,814	2
Infrastructure Networks	road-usa	$\gtrsim$23M	28,854,313	2

B. Experiment setup

In our experiments, we measured both the matching time and memory usage required to solve the subgraph isomorphism problem. For instances from the Classic benchmark, we search for all solutions. For the Graph Database and Scalability benchmarks, the algorithms stop as soon as a solution is found,

except for the hugetrace-00020 dataset, for which all solutions are searched. The default time limit is set to 5 hours for instances from the Classic benchmark (except for the MIVIA LDG dataset). If at least half of the solvers can solve a given instance within this limit, we attempt to solve the same instance with the remaining solvers using an extended timeout of 10 days. For the Graph-database benchmark, the timeout is set to 600 seconds, following previous works [58], [62]. For the Scalability benchmarks, the timeout is set to 1,200 seconds, i.e., doubled compared to the Graph-database benchmark, to account for the larger size of the graphs.

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. To accelerate the evaluation, we ran up to 50 instances in parallel for each algorithm, as all instances are independent and do not share data. This parallel execution strategy allows efficient utilization of the server resources without affecting the correctness or reproducibility of the results.

C. What is the magnitude of the improvement achieved by FASTiso over VF3/VF3L?

To measure the improvement of FASTiso over VF3 and VF3L, we compare all three algorithms on the MIVIA LDG dataset, originally used by the authors of VF3 to demonstrate its superiority. 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 shown in Figure 2, all additional results are provided in the supplementary material. As shown by the results, Fastiso significantly outperforms VF3 across all datasets, and is outperformed by VF3L only on graphs labeled with a non-uniform distribution when enumerating all solutions.

Finding all solutions: For unlabeled graphs, FASTiso's advantage increases with both size (N) and density (d). It outperforms VF3L and VF3 with substantial time gains on large instances, notably for $d = 0.3, N = 3000$ (3–4 hours saved, speed-up of 2.87×–3.34×) and $d = 0.4, N = 1500$ (4–5 hours saved, up to 3.82× faster). For uniformly labeled graphs, FASTiso and VF3L exhibit similar performance for $d = 0.2$ and $N \leq 6500$, both outperforming VF3. FASTiso becomes the fastest for $d = 0.3$, doubling VF3L's speed for $d = 0.4$. In contrast, VF3L maintains a slight advantage (1.29×) on non-uniform label distributions.

Finding a single solution: FASTiso is consistently the fastest algorithm. For unlabeled graphs, it is at least twice as fast as its competitors, with a peak speed-up of 25.53× ($N = 500, d = 0.4$). For labeled graphs (uniform or non-uniform), it outperforms VF3 and VF3L by up to two orders of magnitude in certain sizes.

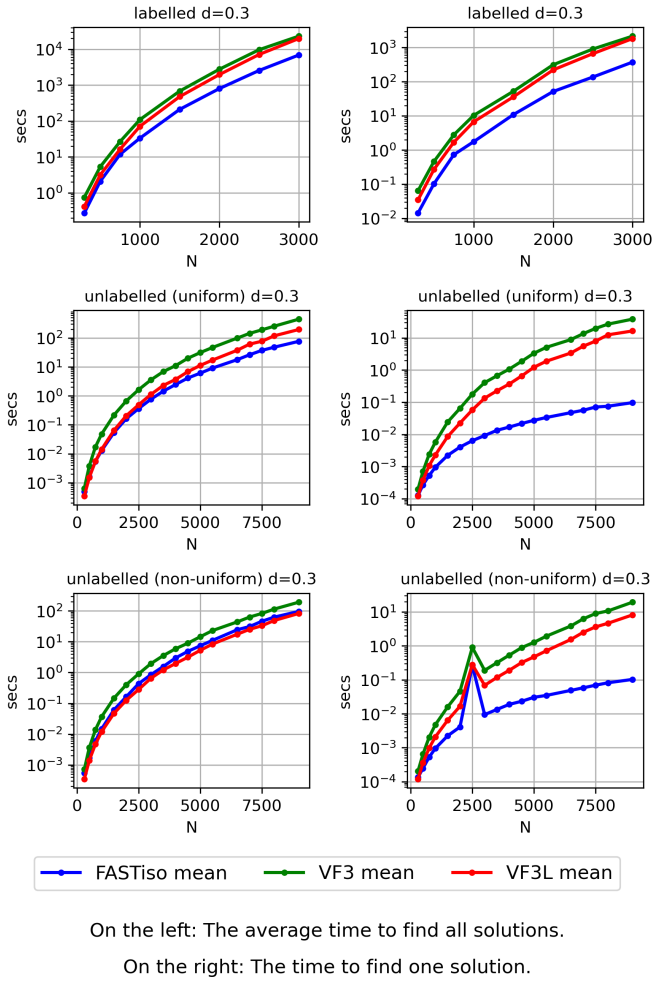

Fig. 2. The average matching times for finding all solutions and for finding a single solution on graphs from the NIVIA LDG dataset with density $d = 0.3$. Here, N denotes the number of nodes in the target graph.

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. How does FASTiso compare with state-of-the-art solutions?

We compare FASTiso with VF3, VF3L, and the other state-of-the-art solvers on all Classic benchmarks, except for the MIVIA LDG dataset. The Figure 3 reports the detailed experimental results. Overall, FASTiso achieves the best global performance and is the fastest solver on the majority of the datasets (6 out of 9).

Table 2 (see supplementary file) reports the results on the Scalefree dataset, which contains both sparse and dense graphs. This dataset is known to be particularly challenging for VF3 when compared to constraint programming-based approaches [13], [15]. As shown in the table, FASTiso achieves the best performance on this dataset, followed by Glasgow and PathLad+. Notably, PathLad+ is itself several orders of

magnitude faster than VF3, VF3L, and RI. For most instances, all solvers exhibit similar execution times. However, FASTiso, Glasgow, and PathLad+ significantly widen the performance gap on a subset of instances corresponding to dense graphs.

These observations are further confirmed on the RAND dataset (see supplementary file, Table 3), which consists of random graphs, both sparse and dense, derived from the SI dataset. FASTiso is consistently the fastest solver across all graph sizes. It is approximately twice as fast as Glasgow, and outperforms RI, VF3, VF3L, and PathLad+ by one to two orders of magnitude for graph sizes 200 and 600.

Overall, FASTiso performs particularly well on small dense graphs. An exception is observed on the Phase dataset (see supplementary file, Table 10), which consists of very dense graphs of small size. This dataset contains only unsatisfiable instances, where advanced filtering and constraint propagation techniques give Glasgow and PathLad+ a distinct advantage. On this dataset, Glasgow is the fastest solver, followed by PathLad+ and FASTiso. Glasgow outperforms FASTiso by approximately one order of magnitude, while FASTiso remains about four times faster than VF3, VF3L, and RI.

The six remaining datasets of the classic benchmark contain only small and sparse graphs. Even on these datasets, FASTiso exhibits the best overall performance, ranking as the top solver on four out of six datasets. It consistently outperforms VF3, VF3L, and RI, but is outperformed by Glasgow and PathLad+ on the Meshes dataset, which, like the Phase dataset, consists exclusively of unsatisfiable instances. However, the superiority of Glasgow and PathLad+ over FASTiso is observed on only a single instance that FASTiso fails to solve within the allotted time. Finally, FASTiso is also outperformed by PathLad+ on the M4D dataset.

Table III and Figure 3 report the results of our experiments on the Graph-database benchmark datasets. Glasgow fails to solve any instance on several datasets due to the high computational complexity of its filtering techniques, and consumes more than 500 GB of memory on each instance. These datasets are reported as NA in Table III. FASTiso solves the largest number of instances on five out of the eight datasets. Overall, FASTiso solves 12,837 instances, followed by PathLAD+ with 12,670 instances, VF3 with 12,477 instances, VF3L with 12,463 instances, and Glasgow with 6,336 instances. The most challenging dataset for all solvers is Eu2005, which contains the largest number of edges. However, it can be observed that constraint programming–based approaches are more effective than tree-search–based approaches on the Human dataset.

E. Scalability Test

Figures 5 and 6 report the scalability results. The Glasgow solver is not included in these figures, as it requires more than 500 GB of memory, and the time complexity of its filtering and constraint propagation techniques prevents it from solving the large instances of this benchmark.

As observed for the Eu2005 dataset in the Graph-database benchmark, the dataset on which the solvers resolve the fewest instances is scale21-ef16-adj, which also has the largest number of edges. This indicates a clear sensitivity of the algorithms to the number of edges. On the scale21-ef16-adj dataset, PathLad+ solves the largest number of instances (586), followed by FASTiso (458), VF3 (416), VF3L (396), and RI (331). Although FASTiso is faster than PathLad+ for pattern graphs of size approximately smaller than 40, the more sophisticated filtering and constraint propagation techniques employed by PathLad+ allow it to outperform FASTiso for pattern graphs of size greater than or equal to 40.

However, this advantage does not hold on the hugetrace-00020 dataset, whose target graph contains more than 16 million nodes. On this dataset, PathLad+ does not solve any instance within the time limit, whereas FASTiso and the other solvers successfully solve all instances. This suggests that PathLad+ is more sensitive to the number of nodes than FASTiso. To further confirm this observation, we compare FASTiso and PathLad+ on the road-usa dataset, whose target graph contains more than 23 million nodes. The results, reported in the supplementary material (see Figure D), are consistent with those obtained on hugetrace-00020: PathLad+ solves 275 instances, while FASTiso solves 1358 instances.

Although FASTiso is not the best-performing algorithm on the scale21-ef16-adj dataset, its new pruning rules, which are more effective than those of other tree search methods and less costly than those used by PathLad+ and Glasgow, allow it to outperform the other tree search methods on the scale21-ef16-adj and hugetrace-00020 datasets.

V. Conclusion

In this article, we propose FASTiso, an exact subgraph isomorphism algorithm that emphasizes a strong consistency between the variable ordering strategy and the pruning rules used during the mapping process. Extensive experiments conducted on a wide range of reference datasets demonstrate the effectiveness and robustness of FASTiso. The results show that FASTiso adapts well to diverse graph structures, performing efficiently on both small and large graphs. Overall, it achieves the best global performance among the evaluated algorithms and clearly outperforms its reference solver VF3, as well as the other tree-search-based methods such as RI and VF3L, on the vast majority of datasets. The only exception is observed on the MIVIA LDG dataset, where VF3L slightly outperforms FASTiso on graphs labeled with a non-uniform distribution when enumerating all solutions.

Compared to constraint programming-based approaches, FASTiso achieves competitive or superior performance on most datasets, while being outperformed on a few instances where advanced filtering and constraint propagation techniques provide a significant advantage. However, due to the higher computational overhead of these techniques, constraint-based solvers exhibit a stronger sensitivity to the number of nodes. In contrast, FASTiso demonstrates better scalability behavior, making it more suitable for large-scale graphs.

Looking ahead, we plan to further improve FASTiso by exploring additional pruning rules and incorporating symmetry handling techniques to reduce redundant exploration. We also intend to develop a parallel and distributed version of the algorithm to handle very large graphs containing billions of nodes.

TABLE III
COMPARISON ON THE **GRAPH-DATABASE** BENCHMARK. EACH DATASET CONTAINS 1,800 INSTANCES. FOR EACH SOLVER, WE REPORT THE NUMBER OF INSTANCES SOLVED.

solver	Yeast	Human	HPRD	WordNet	DBLP	Eu2005	Youtube	US-Patents
FASTiso	1744	1193	1794	**1800**	**1730**	**1161**	**1732**	**1683**
VF3	1720	1084	1786	1795	1712	1096	1637	1647
VF3L	1724	1043	1782	1794	1696	1131	1636	1657
RI	1727	950	1713	1796	1481	1011	1637	1646
PathLad	**1798**	1757	1798	1771	1630	1047	1688	1181
Glasgow	1797	**1763**	**1800**	976	NA	NA	NA	NA

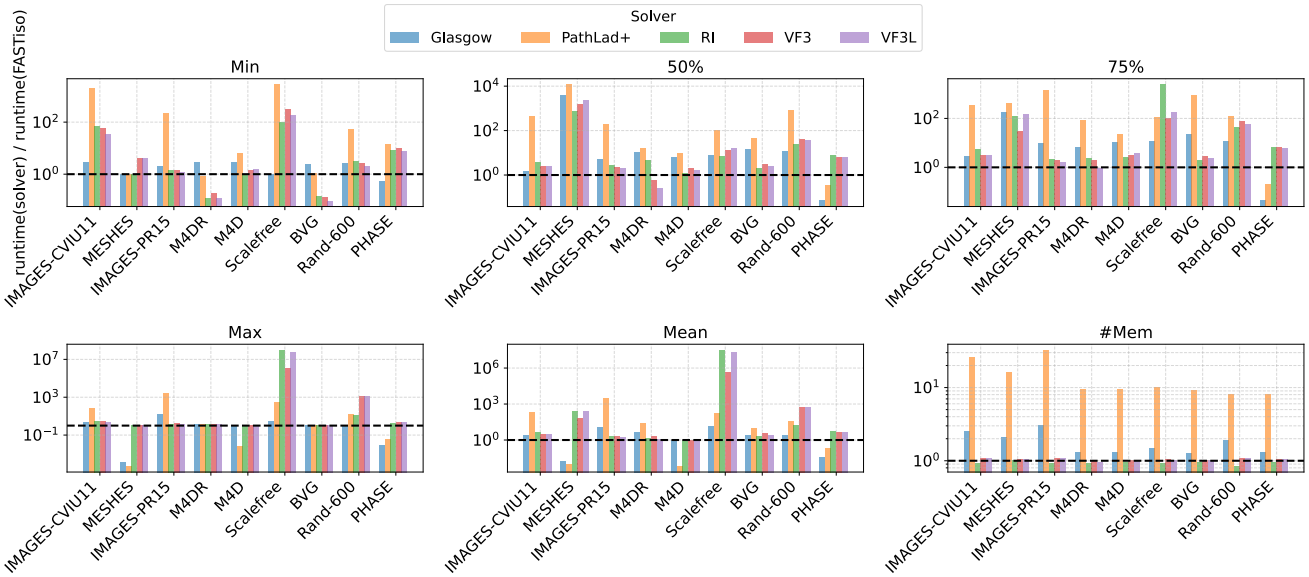

Fig. 3. Detailed results on the Classic benchmark. We report the ratios of average runtime, maximum runtime, total runtime, and average memory usage of different solvers, normalized by the FASTiso solver (lower is better, FASTiso = 1). FASTiso systematically outperforms other tree search approaches and surpasses constraint programming-based methods on most datasets. However, FASTiso is mainly outperformed by constraint programming approaches on unsatisfiable instances, such as Phase and Meshes.

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] Z. A. Ansari, Md. A. Parwez, I. R. Thoker, and Jahiruddin. Enhanced subgraph matching for large graphs using candidate region-based decomposition and ordering. *Journal of King Saud University - Computer and Information Sciences*, 35(8):101694, 2023.

[5] Z. Ali Ansari, J., and M. Abulaish. An efficient subgraph isomorphism solver for large graphs. *IEEE Access*, 9:61697–61709, 2021.

[6] 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.

[7] B. Bhattarai, H. Liu, and H. H. Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1447–1462, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[9] 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.

[10] 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.

[11] 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.

[12] T. Cai, J. Li, A. Mian, R. Li, T. Sellis, and J. X. Yu. Target-aware holistic influence maximization in spatial social networks. *IEEE Transactions on Knowledge and Data Engineering*, 34(4):1993–2007, 2022.

[13] 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.

[14] 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.

[15] 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.

[16] V. Carletti, P. Foggia, A. Saggese, and M. Vento. Introducing VF3: A new algorithm for subgraph isomorphism. In Pasquale Foggia, Cheng-Lin Liu, and Mario Vento, editors, *Graph-Based Representations in Pattern Recognition*, pages 128–139, Cham, 2017. Springer International Publishing.

[17] 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*

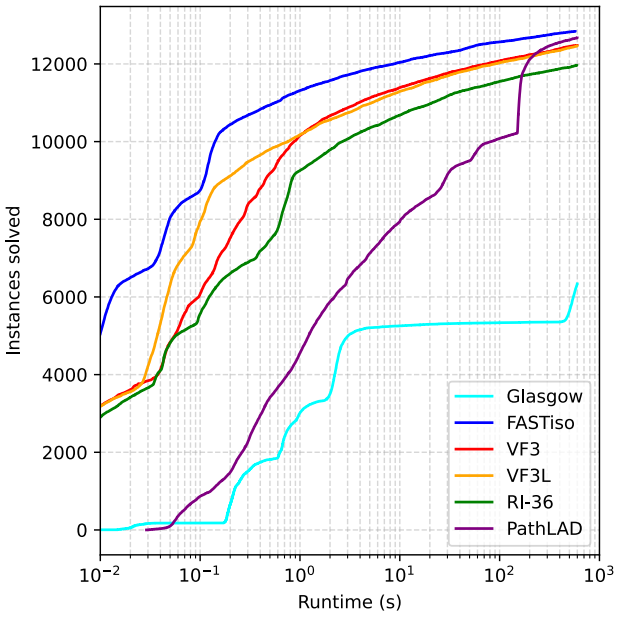

Fig. 4. Detailed Results on Graph-database benchmark. We report the cumulative number of instances solved as a function of runtime.

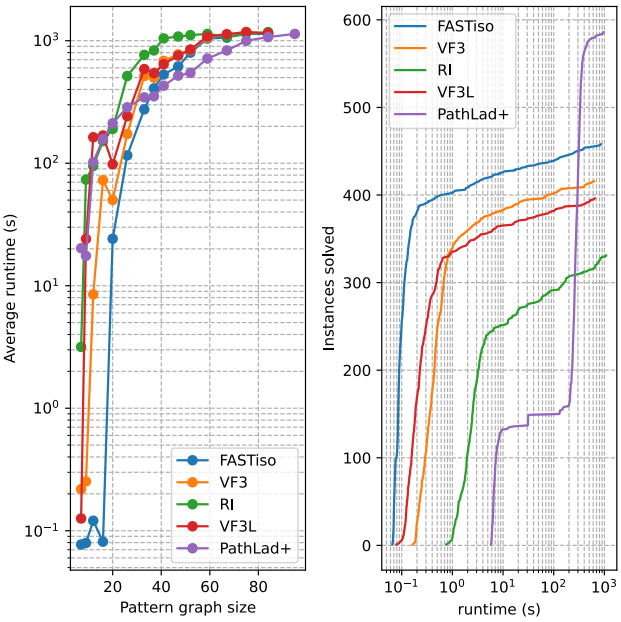

Fig. 5. Detailed Results on scale21-ef16-ad dataset (Scalability benchmark). On the left, we report the mean runtime with respect to the pattern graph size in terms of number of nodes, and on the right, the cumulative number of instances solved as a function of runtime. The peak memory usage of FASTiso is 5.15 GB, whereas PathLAD+ reaches 4.17GB.

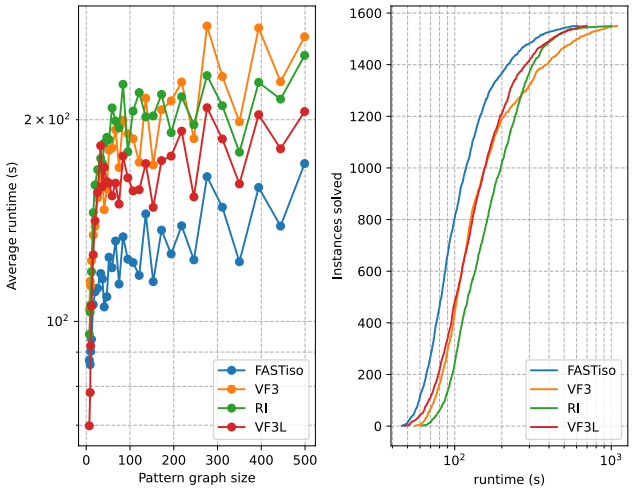

Fig. 6. Detailed Results on hugetrace-00020 dataset (Scalability benchmark). On the left, we report the mean runtime with respect to the pattern graph size in terms of number of nodes, and on the right, the cumulative number of instances solved as a function of runtime. PathLAD+ does not solve any instance in this dataset. The peak memory usage of FASTiso is 7.74 GB, whereas PathLAD+ reaches 36.19 GB.

Intelligence, 40(4):804–818, 2018.

[18] 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.

[19] 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.

[20] L. 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*, 26:1367 – 1372, 11 2004.

[21] 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.

[22] 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.

[23] 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.

[24] M. Han, H. Kim, G. Gu, K. Park, and W. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1429–1446, New York, NY, USA, 2019. Association for Computing Machinery.

[25] W. Han, J. Lee, and J. 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*, SIGMOD '13, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery.

[26] 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.

[27] 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.

[28] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao. Answering natural language questions by subgraph matching over knowledge graphs. *IEEE Transactions on Knowledge and Data Engineering*, 30(5):824–837, 2018.

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

[30] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, and W. Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 925–937, New York, NY, USA, 2021. Association for Computing Machinery.

[31] J. Kim, H. Shin, W. Han, S. Hong, and H. Chafi. Taming sub-

graph isomorphism for rdf query processing. *Proc. VLDB Endow.*, 8(11):1238–1249, July 2015.

[32] R. Kimmig, H. Meyerhenke, and D. Strash. Shared memory parallel subgraph enumeration. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 519–529, 2017.

[33] L. Kotthoff, C. McCreesh, and C. Solnon. Portfolios of subgraph isomorphism algorithms. In Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren, editors, *Learning and Intelligent Optimization*, pages 107–122, Cham, 2016. Springer International Publishing.

[34] D. Krleža and K. Fertalj. Graph matching using hierarchical fuzzy graph neural networks. *IEEE Transactions on Fuzzy Systems*, 25(4):892–904, 2017.

[35] Z. Lan, L. Yu, L. Yuan, Z. Wu, Q. Niu, and F. Ma. Sub-gmn: The neural subgraph matching network model. In *2023 16th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, pages 1–7, 2023.

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

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

[38] X. Liu, H. Pan, M. He, Y. Song, X. Jiang, and L. Shang. Neural subgraph isomorphism counting. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 1959–1969, New York, NY, USA, 2020. Association for Computing Machinery.

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

[40] J. Llados, E. Marti, and J.J. Villanueva. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1137–1143, 2001.

[41] 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.

[42] 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.

[43] Ciaran McCreesh, Patrick Prosser, and James 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.

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

[45] T. Nguyen, D. Yang, Y. Ge, H. Li, and A. L. Bertozzi. Applications of structural equivalence to subgraph isomorphism on multichannel multigraphs. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4913–4920, 2019.

[46] 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.

[47] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc. VLDB Endow.*, 8(5):617–628, January 2015.

[48] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[49] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[50] 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.

[51] M. Shirotani, M. Amagasaki, and M. Kiyama. Endnet: Extra-node decision network for subgraph matching. *IEEE Access*, 13:32424–32433, 2025.

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

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

[54] 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.

[55] 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.

[56] 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.

[57] S. Sun and Q. Luo. Scaling up subgraph query processing with efficient subgraph matching. In *Proceedings - 2019 IEEE 35th International Conference on Data Engineering, ICDE 2019*, Proceedings - International Conference on Data Engineering, pages 220–231. IEEE Computer Society, April 2019. Publisher Copyright: © 2019 IEEE.; 35th IEEE International Conference on Data Engineering, ICDE 2019 ; Conference date: 08-04-2019 Through 11-04-2019.

[58] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1083–1098, 2020.

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

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

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

[62] Y. Wang, C. Jin, and S. Cai. Pathlad+: Towards effective exact methods for subgraph isomorphism problem. *Artificial Intelligence*, 337:104219, 2024.

[63] 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.

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

[65] R. C. Wilson and Edwin R. H. A bayesian compatibility model for graph matching. *Pattern Recognition Letters*, 17(3):263–276, 1996.

[66] Q. Xu, X. Wang, J. Li, Y. Gan, L. Chai, and J. Wang. Starmr: An efficient star-decomposition based query processor for sparql basic graph patterns using mapreduce. In Yi Cai, Yoshiharu Ishikawa, and Jianliang Xu, editors, *Web and Big Data*, pages 415–430, Cham, 2018. Springer International Publishing.

[67] J. Yan, X. Yin, W. Lin, C. Deng, H. Zha, and X. Yang. A short survey of recent advances in graph matching. ICMR '16, page 167–174, New York, NY, USA, 2016. Association for Computing Machinery.

[68] D. Yang, Y. Ge, T. Nguyen, D. Molitor, J. D. Moorman, and A. L. Bertozzi. Structural equivalence in subgraph matching. *IEEE Transactions on Network Science and Engineering*, 10(4):1846–1862, 2023.

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

[70] 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.

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

Wilfried Agbeto Wilfried Agbeto received his M.Sc. degree in Computer Science with a concentration in Bioinformatics from the Université du Québec à Montréal (UQAM), Canada, in 2024. Since late 2024, he has been pursuing a Ph.D. in Computer Science at the same university. His research focuses on combinatorial optimization, exact algorithms, and parallel computing, with applications in bioinformatics and graph theory.

Camille Coti Camille Coti is a professor at ETS Montréal. After a PhD from university Paris Sud-XI in 2009, she spent a decade at university Sorbonne Paris Nord before moving to Canada in 2022. Among her interests are high performance computing, distributed computing, and playing the trumpet.

Vladimir Reinharz Vladimir Reinharz is a professor at UQAM Montréal since 2020. He completed his PhD at McGill University before doing a first postdoc at Ben-Gurion University of the Negev and a second at the Institute for Basic Science in South Korea. He is interested in developing new algorithms ranging from combinatorial sampling to numerical solvers to answer biological questions.