Ph.D. DISSERTATION

Fast Graph Isomorphism using Pairwise Color Refinement and Efficient Backtracking

쌍별 색 개선과 효율적인 백트래킹을 이용한 빠른 그래프 동형 알고리즘

AUGUST 2021

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Geonmo Gu

Fast Graph Isomorphism using Pairwise Color Refinement and Efficient Backtracking

쌍별 색 개선과 효율적인 백트래킹을 이용한 빠른 그래프 동형 알고리즘

지도교수 박 근 수

이 논문을 공학박사학위논문으로 제출함

2021 년 7 월

서울대학교 대학원

컴퓨터 공학부

구 건 모

구건모의 박사학위논문을 인준함

2021 년 7 월

Abstract

Fast Graph Isomorphism using Pairwise Color Refinement and Efficient Backtracking

Geonmo Gu
Department of Computer Science
and Engineering
College of Engineering
The Graduate School
Seoul National University

Graph isomorphism is a core problem in graph analysis of various domains including social networks, bioinformatics, chemistry, and so on. As real-world graphs are getting bigger and bigger, applications demand practically fast algorithms that can run on large-scale graphs. Existing approaches, however, show limited performances on large-scale real-world graphs either in time or space. Also, graph isomorphism query processing is often required in many applications, which is a natural generalization of graph isomorphism for multiple graphs. In this thesis we present fast algorithms for graph isomorphism and graph isomorphism query processing.

First, we present a new approach to graph isomorphism, which is the framework of pairwise color refinement and efficient backtracking. Within the framework, we introduce three efficient techniques, which together lead to a much faster and scalable algorithm for graph isomorphism. Experiments on real-world datasets show that our algorithm outperforms state-of-the-art solutions by up to several orders of magnitude in terms of running time.

Second, We develop an efficient algorithm for graph isomorphism query processing. We use a two-level index using degree sequences and color-label distributions. Experimental results on real datasets show that our algorithm is orders of magnitude faster than the state-of-the-art algorithms in terms of index construction time, and it runs faster than existing algorithms in terms of query processing time as the graph sizes increase.

Keywords: graph isomorphism; graph isomorphism query processing; pairwise color refinement; backtracking; partial failing set

Student Number: 2014-21758

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In 1979, Garey and Johnson [19] remarked 12 open problems that were not known to be in P or NP-complete. Graph isomorphism is included in the open problems, and while most of the open problems are resolved, graph isomorphism remains open till today. That is, there is no polynomial-time algorithm for graph isomorphism, which makes it very challenging to deal with the problems on large-scale graphs.

Graph isomorphism is a core problem in graph analysis of various domains including social networks [67, 81], bioinformatics [15], chemistry [58, 46], mechanics [76, 33], and so on [69]. As real-world graphs are getting bigger and bigger, applications demand practically fast algorithms that can run on large-scale graphs. Social network anonymization [80, 67] and circuit verification in VLSI design [69, 41] are examples of such applications.

In the area of social network anonymization, k-anonymity [61, 80] means that there exist at least k candidate users in the released data for any user in the original social network. To obtain k-anonymity, graph isomorphism tests are frequently conducted on the d-neighborhood graph for each vertex in the anonymized social network [67, 80], which is a subgraph induced by the vertex and its d-hop neighbors. The previous work [67, 80] handles only 1-neighborhoods (i.e., $d = 1$), and Zhou and Pei [80] mention that it is very challenging to test graph isomorphism when $d > 1$ because the neighborhood size increases exponentially as d increases.

In circuit design, an electronic design verification software determines whether an integrated circuit layout corresponds to the original schematic circuit design. For the verification, the integrated circuit layout and the original design are transformed into graphs, on which graph isomorphism testing is performed [69]. The software has to process large-scale graphs when verifying large circuits, for example, in VLSI design [41].

There are mainly two existing approaches to graph isomorphism. One is reducing graph isomorphism to the graph canonization problem [50, 51, 30, 72], and another is using subgraph isomorphism algorithms [78, 25, 8, 24, 57, 31, 68]. Existing approaches, however, show limited performances on large-scale real-world graphs either in time or space. We will summarize the two approaches in Section 2.3 and discuss the limitations in Chapter 3.

In this thesis we propose a new approach to graph isomorphism, which is the framework of pairwise color refinement and efficient backtracking. Within the framework, we use three efficient techniques, which together lead to a much faster and scalable algorithm for graph isomorphism. We conduct experiments comparing our algorithm and existing solutions on real-world datasets from various domains. Experimental results show that our algorithm outperforms state-of-the-art solutions by up to several orders of magnitude in terms of running time.

We also present an efficient algorithm for graph isomorphism query processing, which is a natural generalization of graph isomorphism for multiple graphs. We use a two-level index based on degree sequences and color-label distributions. Experimental results on real datasets show that our algorithm is orders of magnitude faster than the state-of-the-art algorithms in terms of index construction time, and it runs faster than existing algorithms in terms of query processing time as the graph sizes increase.

1.2 Organization

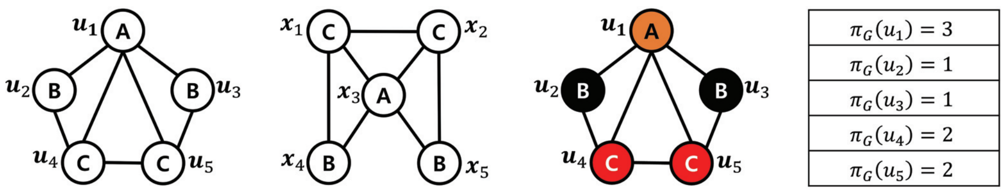
The remainder of the thesis is organized as follows. Chapter 2 provides problem definitions and related work. Chapter 3 describes a fast and scalable algorithm for graph isomorphism. Chapter 4 presents an efficient algorithm for graph isomorphism query processing. Finally, Chapter 5 concludes the thesis and discusses future directions.

Chapter 2

Preliminaries

2.1 Notation

We assume that graphs are simple, undirected, connected, and vertex-labeled, unless otherwise specified. However, all techniques in this thesis can be extended to handle more general cases (e.g., directed, disconnected, multiple edges, multiple labels on vertex/edge). Let $G = (V(G), E(G), L_G)$ be a graph consisting of a set $V(G)$ of vertices, a set $E(G)$ of edges, and a labeling function $L_G : V(G) \to \Sigma$, where Σ is a set of labels. The set of neighbors of u in G is denoted by $N_G(u) = \{v \in V(G) : (u, v) \in E(G)\}$. The *degree* of u is $|N_G(u)|$ and denoted by $\deg_G(u)$. The *degree sequence* of G is denoted by ds_G, which is $|V(G)|$ plus the nonincreasing sequence of the vertex degrees of G. For example, the degree sequence of G in Figure 2.1a is $ds_G = (5, 4, 3, 3, 2, 2)$. For a subset of vertices $S \subseteq V(G)$, the *vertex-induced subgraph* $G[S]$ of G is a subgraph of G whose vertex set is S and edge set consists of the edges in $E(G)$ that have both endpoints in S.



$\pi_G(u_1) = 3$
$\pi_G(u_2) = 1$
$\pi_G(u_3) = 1$
$\pi_G(u_4) = 2$
$\pi_G(u_5) = 2$

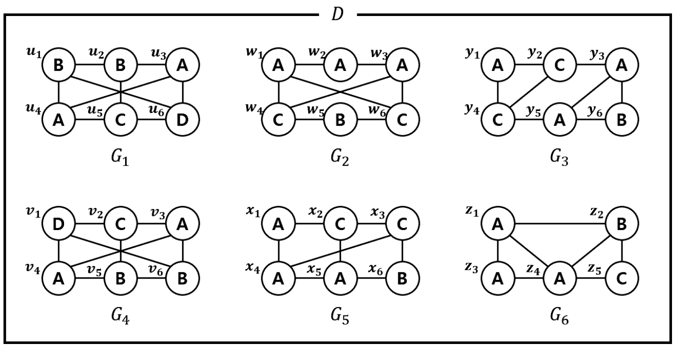
(a) G (b) H (c) (G, π_G)

Figure 2.1: (a–b) G and H are isomorphic. (c) Colored graph (G, π_G) has a stable coloring.

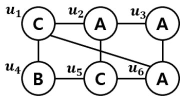
An *isomorphism* of graphs G and H is a bijective mapping $f : V(G) \to V(H)$ such that (1) for each vertex $u \in V(G)$, $L_G(u) = L_H(f(u))$ and (2) $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(H)$. We say that G and H are *isomorphic* if an isomorphism exists between them, and denote this fact by $G \cong H$. For example, G and H in Figure 2.1 are isomorphic.

An *embedding* of G in H is an injective mapping $M : V(G) \to V(H)$ such that (1) for each vertex $u \in V(G)$, $L_G(u) = L_H(M(u))$ and (2) $(M(u), M(v)) \in E(H)$ for every $(u, v) \in E(G)$. If an embedding of G in H exists with $|V(G)| = |V(H)|$ and $|E(G)| = |E(H)|$, then $G \cong H$. An embedding of a vertex-induced subgraph of G in H is called a *partial embedding* of G in H.

A *coloring* of graph G is a function $\pi_G : V(G) \to \{1, 2, \ldots, |V(G)|\}$, and a pair (G, π_G) is called a *colored graph*. For a vertex $u \in V(G)$, we say that $\pi_G(u)$ is the *color* of u. The number of used colors in π_G is denoted by $|\pi_G|$. If $|\pi_G| = 1$, then π_G is called a *unit* coloring. For a color c, the *cell* of π_G corresponding to c is the set of vertices $\pi_G^{-1}(c) = \{u \in V(G) : \pi_G(u) = c\}$. A coloring π_G of G is *stable* if, for any color c and two vertices $u, v \in V(G)$ such that $\pi_G(u) = \pi_G(v)$, $|N_G(u) \cap \pi_G^{-1}(c)| = |N_G(v) \cap \pi_G^{-1}(c)|$. For example, π_G in



(a) Data graphs



(b) Query graph

Figure 2.2: Example instance of graph isomorphism query processing.

Figure 2.1c is a stable coloring.

A directed acyclic graph (DAG) g is a directed graph that contains no cycle. We say that $u \in V(g)$ is a *parent* of $v \in V(g)$ (and v is a *child* of u) if a directed edge $u \to v$ is in $E(g)$. A vertex is a *root* of a DAG if it has no incoming edges.

2.2 Problem Definitions

In this thesis we tackle the following two problems.

Graph Isomorphism. Given two graphs G and H, the *graph isomorphism problem* is to determine whether G and H are isomorphic or not.

Graph Isomorphism Query Processing. Given a set of data graphs $\mathcal{D} = \{G_1, G_2, \ldots, G_k\}$ and a query graph q, the *graph isomorphism query processing (GIQP) problem* is to find all the data graphs in $\mathcal{D}$ that are isomorphic to q. That is, GIQP is to compute the answer set $A_q = \{G_i \in \mathcal{D} : G_i \cong q\}$. For example, the answer set for the data graphs and the query graph in Figure 2.2 is $A_q = \{G_2\}$.

Graph isomorphism is not known to be in P or NP-complete. That is, time complexities of existing algorithms are not bounded by polynomial time, and each algorithm can have an instance that takes exponential time in the worst case. GIQP is a natural generalization of graph isomorphism, and thus there is no polynomial time algorithm for GIQP as well.

Note that we consider vertex-labeled graphs, but all techniques in this paper can be easily applied to unlabeled graphs by setting every vertex to an identical label.

2.3 Related Work

Graph isomorphism can be reduced to other problems such as the *graph canonization* problem and the *subgraph isomorphism* problem. GIQP can be reduced to the *graph similarity search* problem. In this section we summarize related work on these problems. We also briefly summarize previous work of graph isomorphism on some graph classes.

Graph Canonization. *Graph canonization* [50, 30, 51] is the task of finding a canonical form of an input graph G. The canonical form of G is a unique representation of G such that for every H that is isomorphic to G, H has an identical canonical form as that of G. Graph isomorphism can be solved using graph canonization as two graphs are isomorphic if and only if their canonical forms are identical. A common approach to graph canonization is repeatedly performing *color refinement* and *individualization* [51]. Color refinement has been widely used in graph canonization, and it takes $O((E + V)\log V)$ time (while individualization takes constant time) [7], where E is the number of edges and V is the number of vertices in an input graph. One of the best-known graph canonization algorithms is `nauty` [50, 51], and the implementation of `nauty` has been developed for over 30 years since it was proposed. Recently, `Traces`

[51] was proposed with updated `nauty`, where `Traces` outperforms existing algorithms on many graph classes. In addition, the *minimum DFS code* [72] of a graph is a kind of graph canonization.

Subgraph Isomorphism. Given a query graph and a data graph, the *subgraph isomorphism* problem is to determine whether the data graph contains a subgraph that is isomorphic to the query graph [78]. Since graph isomorphism is a special case of subgraph isomorphism, one may use subgraph isomorphism algorithms to cope with graph isomorphism. The problem is called subgraph matching (or subgraph enumeration) if it is required to find all such subgraphs [25, 8, 24, 57]. Extensive research has been done to develop practical solutions for subgraph matching. Most practical solutions (such as `VF2++` [31], $\texttt{Turbo}_{\texttt{ISO}}$ [25], `CFL-Match` [8], and `DAF` [24]) are based on the backtracking approach [68], which recursively maps a query vertex to a data vertex. While performing backtracking, one may use search space pruning techniques.

Graph Similarity Search. Given a set of data graphs, a query graph, and a threshold τ, the *graph similarity search* problem [44, 79, 13, 12] is to find all the data graphs which have *graph edit distance* [62] less than or equal to τ from the query graph. If the graph edit distance between two graphs is zero, then the graphs are isomorphic. Thus, GIQP can be viewed as an exact version of graph similarity search.

Graph Isomorphism on Some Graph Classes. Graph isomorphism is not known to be in P or NP-complete for general graphs. However, linear-time algorithms exist for planar graphs [29] and interval graphs [9]. Also, the problem is solvable in polynomial time for bounded-degree graphs [47] and trivalent graphs [18]. For general graphs, there is a quasi-polynomial time algorithm [3, 27]. We refer the reader to [38, 39] for a comprehensive survey of the problem.

Chapter 3

Graph Isomorphism

Given two graphs G and H, the graph isomorphism problem is to determine whether G and H are isomorphic or not. We assume that G and H are the same size, because otherwise it is trivial to determine that they are not isomorphic. Throughout this chapter we use V and E to denote the number of vertices and the number of edges of G, respectively.

There are challenges when we deal with graph isomorphism on large-scale graphs. First, existing approaches show limited performances on large-scale graphs. Regarding the graph canonization approach, the number of iterations of color refinement is not small on large-scale real-world graphs, and existing solutions struggle on those graphs. Regarding the subgraph isomorphism approach, since the query graph is usually smaller than the data graph, algorithms are often optimized for small query graphs and thus they cannot handle graph isomorphism on large-scale graphs. Second, designing a time and space efficient data structure is challenging. $\texttt{Turbo}_{\texttt{ISO}}$, `CFL-Match`, and `DAF` use auxiliary data structures based on the query graph and the data graph. Their data structures have at least quadratic time and space complexities, and it is infeasible to use them in graph isomorphism. Lastly, implementing pruning techniques for large-scale graphs is challenging. *Pruning by failing sets* [24] is a technique used to prune out redundant search space during backtracking. The technique has been shown to be very effective in subgraph matching as the query size increases [24, 65]. However, the technique requires $O(|V(q)|^2/w)$ space, where q is the query graph and w is the word size, and thus the technique is impracticable when we deal with large-scale graphs.

In this chapter, we propose a new approach to graph isomorphism, which combines pairwise color refinement and efficient backtracking. In our framework, a color refinement algorithm is used to compute the *stable coloring* of the disjoint union of two input graphs. After the stable coloring, we perform a backtracking procedure that makes good use of the stable coloring and an efficient pruning technique to find an isomorphism of the input graphs. The main features of our approach are as follows:

1. *Pairwise color refinement and binary cell mapping.* Throughout our approach, we make use of the stable coloring computed by the pairwise color refinement. First of all, we define a *binary cell* for the stable coloring, which is a cell of size two containing one vertex from each input graph. We show that stable colorings of real-world datasets contain large numbers of binary cells. We compute an initial partial embedding by mapping the two vertices in every binary cell, which is an effective way to start backtracking.

2. *A linear-space data structure that can serve as a complete search space for backtracking.* To conduct backtracking on large input graphs, an essential job is to transfer the result of color refinement to backtracking in an

efficient way. Accordingly, we develop a new data structure that exploits the stable coloring of the pairwise color refinement, called the *compressed CS* (candidate space). The compressed CS uses $O(V + E)$ space, and we propose an algorithm to build the compressed CS in $O(V + E)$ time. The compressed CS improves upon `DAF`'s CS [24] both in time and space, which has $O(V^2 + E^2)$ time and space complexities.

3. *An effective pruning technique feasible for large-scale graphs.* We introduce the *partial failing set*, which has the same pruning effect as `DAF`'s failing set in [24], while it is much smaller than `DAF`'s failing set. Through the partial failing set, the pruning technique can be efficiently applied to graph isomorphism even on large-scale graphs.

We conduct experiments comparing our algorithm and existing solutions on real-world datasets from various domains. The datasets consist of 23 large-scale graphs that contain thousands to millions of vertices, and PubChem [66] chemistry database that contains 499,963 graphs with less than a thousand vertices. For the 23 large-scale datasets, experimental results show that our algorithm outperforms state-of-the-art solutions by up to several orders of magnitude in terms of running time. For isomorphic datasets, our algorithm outperforms existing solutions on 19 datasets, and runs about 2,430 times faster than the fastest existing solutions on average. For the remaining 4 datasets, our algorithm is the runner-up and about 1.3 times slower than the fastest solution on average. For nonisomorphic datasets, our algorithm is always the best and runs about 3,160 times faster than the fastest existing solutions on average. For PubChem, we conduct a parameter sensitivity analysis to evaluate the effect of input parameters (i.e., the number of vertices and average degrees). Overall in the analysis, our algorithm is faster than existing solutions.

Algorithm 1: Algorithm for Graph Isomorphism

```
Input  : graphs $G$ and $H$
Output: "YES" if $G \cong H$, "NO" otherwise
1  $GH \leftarrow G \cup H$;
2  $\pi_{GH} \leftarrow$ color vertices by degrees and labels;
3  $\pi'_{GH} \leftarrow \mathit{Refine}(GH, \pi_{GH})$;
4  if $\mathit{CheckColoring}(\pi'_{GH}) = \mathit{false}$ then
5      return "NO";
6  $M \leftarrow \mathit{BinaryCellMapping}(GH, \pi'_{GH})$;
7  $CS \leftarrow \mathit{BuildCompressedCS}(G, H, \pi'_{GH})$;
8  $G_D \leftarrow \mathit{BuildDAG}(G, \pi'_{GH})$;
9  $\mathit{Backtrack}(G_D, CS, M)$;
10 if an embedding of $G$ in $H$ is found then
11     return "YES";
12 else
13     return "NO";
```

3.1 Algorithm Overview

Algorithm 1 shows the overview of our solution that outputs whether two input graphs G and H are isomorphic. We begin with computing a stable coloring of the disjoint union of two input graphs (Lines 1 to 3). Initially, we color vertices in the disjoint union GH of input graphs according to degrees and labels of vertices. That is, two vertices $u, v \in V(GH)$ have the same color if and only if $\deg_{GH}(u) = \deg_{GH}(v)$ and $L_{GH}(u) = L_{GH}(v)$. Then, a stable coloring of the initial coloring is obtained by a color refinement algorithm. In order for

G and H to be isomorphic, the stable coloring must consist of the cells such that each cell contains the same number of vertices from each graph. Thus, if there exists a cell that contains a different number of vertices from G and H, then the graphs are not isomorphic (Lines 4 to 5). Otherwise G and H may be isomorphic, and we search for an isomorphism by using a backtracking approach, which recursively extends a partial embedding of G in H (Lines 6 to 13). Our backtracking approach consists of the following steps:

1. An initial partial embedding M is computed by mapping the two vertices in every binary cell. We will prove that the initial partial embedding is a unique partial embedding with respect to the binary cells that could lead to an embedding of G (Section 3.2).

2. A data structure, called the compressed CS, is computed, which serves as a complete search space for all embeddings of G in H. The compressed CS uses $O(V + E)$ space and it is built in $O(V + E)$ time (Section 3.3).

3. A DAG G_D of G is used to compute a matching order of vertices. We adopt an adaptive matching order with DAG ordering [24] (Section 3.4).

4. Finally, *Backtrack* is invoked to find an embedding of G in H by extending M. During backtracking, we use the pruning technique based on the failing set [24]. We introduce the *partial failing set*, which enables us to apply the pruning technique to large-scale graphs (Section 3.4).

3.2 Pairwise Color Refinement and Binary Cell Mapping

In this section we describe the pairwise color refinement, which outputs a stable coloring of $G \cup H$. Our graph isomorphism algorithm begins backtracking with an initial partial embedding of G in H that is obtained by mapping every two vertices in the binary cell of the stable coloring. We will show that the initial partial embedding is a unique partial embedding that could lead to an embedding of G in H.

Pairwise Color Refinement. Given a colored graph (G, π_G), *color refinement* [50, 10, 56, 40, 7] is the problem of finding the coarsest stable coloring π'_G that is finer than π_G. Let π, π' be two colorings of a graph. We say that π' is *finer* than π (and π is *coarser* than π') if every cell of π' is a subset of some cell of π. A coloring π' is the *coarsest* stable coloring finer than π if there is no stable coloring π'' such that π'' is finer than π and strictly coarser than π'.

Given two graphs G and H as an input of graph isomorphism, the *pairwise color refinement* is the process that outputs a stable coloring of π'_{GH} as follows. First, make a disjoint union GH of G and H, and compute a coloring π_{GH} of GH such that, for any two vertices $u, v \in V(GH)$, $\pi_{GH}(u) = \pi_{GH}(v)$ if and only if $\deg_{GH}(u) = \deg_{GH}(v)$ and $L_{GH}(u) = L_{GH}(v)$. Then, compute the coarsest stable coloring of π'_{GH} that is finer than π_{GH} using a color refinement algorithm.

Berkholz, Bonsma, and Grohe [7] presented an $O((E + V) \log V)$ time color refinement algorithm. They proved that the time complexity of the algorithm is optimal if it starts with a unit coloring. In the pairwise color refinement, we adopt their color refinement algorithm (see **Color Refinement Algorithm** in Section 4.1 for a brief description of the algorithm).

Binary Cell Mapping. Suppose that we refined the initial colored graph (GH, π_{GH}) and obtained the coarsest stable coloring π'_{GH}. We say that a cell of π'_{GH} is *binary* if the cell consists of two vertices u, x such that $u \in V(G)$ and $x \in V(H)$.

We show that vertices in the binary cells can be mapped to each other to produce a partial embedding of G in H.

Lemma 3.2.1. *Let B be the set of binary cells of ${\pi'}_{GH}$. The mapping M that maps $u \in V(G)$ to $x \in V(H)$ for every $\{u, x\} \in B$ is a partial embedding of G in H.*

Proof. Since ${\pi'}_{GH}$ is finer than π_{GH}, each cell of ${\pi'}_{GH}$ is a subset of some cell in π_{GH}. Thus, any two vertices in the same cell in ${\pi'}_{GH}$ have the same labels.

For any two binary cells $\{u, x\}, \{v, y\} \in B$ such that $u, v \in V(G)$ and $x, y \in V(H)$, we have $|N_{GH}(u) \cap \{v, y\}| = |N_{GH}(x) \cap \{v, y\}|$ because ${\pi'}_{GH}$ is stable. Thus, x is adjacent to y if u is adjacent to v.

Therefore, M is a partial embedding of G in H. □

If G and H have the same numbers of vertices and edges, then M in Lemma 3.2.1 is a unique partial embedding with respect to binary cells that could lead to an embedding of G in H.

Lemma 3.2.2. *If $|V(G)| = |V(H)|$ and $|E(G)| = |E(H)|$, then there is no embedding of G in H that maps $u \in V(G)$ to $x \in V(H)$ such that u and x are in different cells of ${\pi'}_{GH}$.*

Proof. We prove by contradiction. Assume that there exists an embedding M of G in H that maps u to x.

Let ${\pi''}_{GH}$ be the coloring of GH obtained by merging cells of ${\pi'}_{GH}$ in the following way: initially, ${\pi''}_{GH}$ is equal to ${\pi'}_{GH}$; for each $v \in V(G)$, if v and $M(v)$ belong to different cells of ${\pi''}_{GH}$, then we merge these two cells. Since at least two cells (i.e., cells that u and x belong to) are merged, ${\pi''}_{GH}$ is strictly coarser than ${\pi'}_{GH}$.

By the assumption of the lemma, M is an isomorphism of G and H. Thus, v and $M(v)$ have the same labels and degrees for each $v \in V(G)$, and in each merged cell of ${\pi''}_{GH}$, any two vertices have the same labels and degrees. Thus, each merged cell is a subset of some cell of π_{GH}, and so ${\pi''}_{GH}$ is finer than π_{GH}.

Let a, b be two vertices in the same cell of π''_{GH}. We will denote $a \equiv_{\pi''_{GH}} b$ if $|N_{GH}(a) \cap \pi''^{-1}_{GH}(c)| = |N_{GH}(b) \cap \pi''^{-1}_{GH}(c)|$ for any color c. If a and b belong to the same cell in π'_{GH}, then we have $a \equiv_{\pi''_{GH}} b$ because π'_{GH} is a stable coloring. Otherwise, a and b are put into the same cell due to some vertices v and $M(v)$ such that (1) $\pi'_{GH}(v) \neq \pi'_{GH}(M(v))$ and (2) $\pi'_{GH}(a) = \pi'_{GH}(v)$ and $\pi'_{GH}(b) = \pi'_{GH}(M(v))$. For each neighbor w of v, we have that $\pi''_{GH}(w) = \pi''_{GH}(M(w))$, and that $M(w)$ is a neighbor of $M(v)$ since M is an isomorphism. Thus, we get $v \equiv_{\pi''_{GH}} M(v)$. Since we have $\pi'_{GH}(a) = \pi'_{GH}(v)$ and $\pi'_{GH}(b) = \pi'_{GH}(M(v))$, we get $a \equiv_{\pi''_{GH}} v$ and $b \equiv_{\pi''_{GH}} M(v)$. It follows that $a \equiv_{\pi''_{GH}} b$, i.e., $|N_{GH}(a) \cap \pi''^{-1}_{GH}(c)| = |N_{GH}(b) \cap \pi''^{-1}_{GH}(c)|$ for any color c.

Therefore, π''_{GH} is a stable coloring that is finer than π_{GH} and strictly coarser than π'_{GH}. This contradicts that π'_{GH} is the coarsest stable coloring finer than π_{GH}. □

Due to Lemmas 3.2.1 and 3.2.2, we first compute the partial embedding by mapping the two vertices in every binary cell, and then we search for an embedding of G in H by extending the partial embedding using a backtracking procedure. This strategy is effective because we do not need to check adjacencies among the vertices in the binary cells. Also, in many real-world graphs, stable colorings contain large numbers of binary cells as shown in our experiments (see **Effect of Binary Cell Mapping** in Section 3.5.2).

3.3 Compressed Candidate Space

Candidate space (CS) [24] is a data structure that stores mapping relationships between a query graph and a data graph. In this section we define a CS associated with the stable coloring, which is thus called *colored CS*. To store the colored CS compactly, we design a new data structure called *compressed*

CS, which has $O(V+E)$ space complexity, and we present an $O(V+E)$ time algorithm to compute it.

Throughout this section, we assume that a stable coloring π_{GH} of $G \cup H$ is computed by the pairwise color refinement, and each cell of π_{GH} contains the same numbers of vertices from G and H (otherwise G and H are not isomorphic).

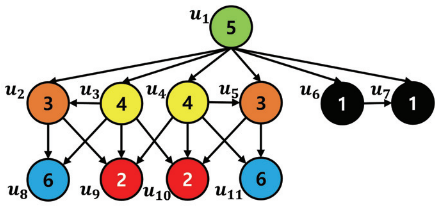
Colored CS. Given graphs G and H, and a stable coloring π_{GH} of $G \cup H$, a *colored CS* is defined as follows.

Definition 3.3.1. ***(Colored CS)*** *A colored CS on G and H for π_{GH} consists of the candidate set $C(u)$ for each $u \in V(G)$ and edges between the candidates as follows.*

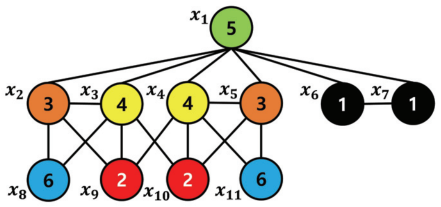
1. *For each $u \in V(G)$, there is a candidate set $C(u)$, where $C(u) = \{x \in V(H) : \pi_{GH}(x) = \pi_{GH}(u)\}$.*
2. *There is an edge between $x \in C(u)$ and $y \in C(v)$ if and only if $(u,v) \in E(G)$ and $(x,y) \in E(H)$.*

By Lemma 3.2.2, if there is an embedding M of G in H such that $M(u) = x$, then x must be in $C(u)$ of the colored CS. As in the CS [24], the colored CS serves as a complete search space for all embeddings of G in H. Figure 3.1 illustrates an example of a colored CS. Note that G_D is a DAG of G in the example.

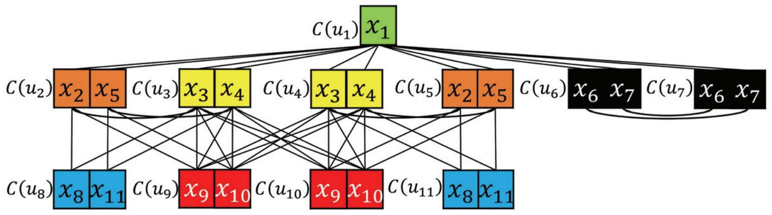
For a candidate $x \in C(u)$, let $N_v^u(x)$ denote the set of vertices $y \in C(v)$ that are adjacent to x in the colored CS. For example, in Figure 3.1c, $N_{u_3}^{u_1}(x_1) = \{x_3, x_4\}$ and $N_{u_5}^{u_4}(x_3) = \{x_2\}$. Recall that $C(v)$ is a set of vertices in $V(H)$ whose colors are the same as v's. Thus, $N_v^u(x)$ is the set of x's neighbors in H whose colors are the same as $\pi_{GH}(v)$.



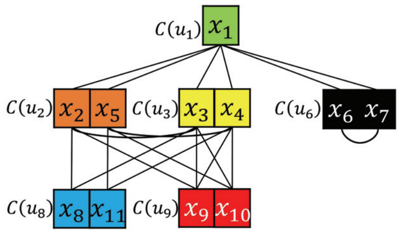
(a) G_D



(b) H



(c) Colored CS on G and H



(d) Compressed CS

Figure 3.1: (a–b) Colors appear within the vertices. (c) Colored CS. (d) Compressed form of the colored CS.

Suppose that we are given a colored CS on G and H for π_{GH}. By definition of the colored CS, we have the following properties.

Property 3.3.1. *For any $u, v \in V(G)$ such that $\pi_{GH}(u) = \pi_{GH}(v)$, the candidate set of u is the same as that of v, i.e., $C(u) = C(v)$.*

Property 3.3.2. *For any $(u, v), (u', v') \in E(G)$ and $x \in V(H)$ such that $\pi_{GH}(u) = \pi_{GH}(u')$, $\pi_{GH}(v) = \pi_{GH}(v')$, $x \in C(u)$, and $x \in C(u')$, we have $N_v^u(x) = N_{v'}^{u'}(x)$ (e.g., $N_{u_9}^{u_3}(x_3) = N_{u_{10}}^{u_4}(x_3) = \{x_9, x_{10}\}$ in Figure 3.1c).*

According to the two properties, we can see that some parts of the colored CS are duplications. By removing the duplications, we can store the colored CS in linear space.

Compressed CS. We present a new data structure, called *compressed CS*, which is a compressed form of the colored CS.

Definition 3.3.2. ***(Compressed CS)*** *A compressed form of the colored CS on G and H for π_{GH} consists of the following.*

- *For each color c of π_{GH}, there is a representative vertex rep(c) of c, which is a vertex in $V(G)$ that belongs to $\pi_{GH}^{-1}(c)$. The candidate set $C(rep(c))$ is stored for each color c.*

- *For any two (possibly non-distinct) representative vertices r and r', there is an edge between $x \in C(r)$ and $y \in C(r')$ in the compressed CS if and only if there exists an edge between $x \in C(u)$ and $y \in C(v)$ in the colored CS such that $rep(\pi_{GH}(u)) = r$ and $rep(\pi_{GH}(v)) = r'$.*

Figure 3.1d shows the compressed CS with respect to the colored CS in Figure 3.1c. In the example, u_1, u_2, u_3, u_6, u_8, and u_9 are representative vertices.

Lemma 3.3.1. *The compressed CS uses $O(V + E)$ space.*

Algorithm 2: BuildCompressedCS(G, H, π_{GH})

```
1 for each color $c \in \pi_{GH}$ do
2     $rep(c) \leftarrow$ select a vertex from $\pi_{GH}^{-1}(c)$ that is from $V(G)$;
3     $C(rep(c)) \leftarrow \{u \in \pi_{GH}^{-1}(c) : u \in V(H)\}$;
4 Sort $N_H(x)$ for every $x \in V(H)$ by color;
5 for each $x \in V(H)$ do
6     $idx \leftarrow 1$; $P_x[idx] \leftarrow 1$;
7     for $i \in \{2, \ldots, \deg_H(x)\}$ do
8         if $\pi_{GH}(N_H(x)[i]) \neq \pi_{GH}(N_H(x)[i-1])$ then
9             $idx \leftarrow idx + 1$; $P_x[idx] \leftarrow i$;
```

Proof. In the compressed CS, each vertex in $V(H)$ appears at most once as a candidate because for any two distinct colors c and c', $C(rep(c)) \cap C(rep(c')) = \emptyset$. Thus, an edge in $E(H)$ appears at most once in the compressed CS. The compressed CS does not have an edge that is not in $E(H)$ by Condition (2) in the definition of the colored CS. $\square$

We present an $O(V + E)$ time algorithm to build the compressed CS when graphs G and H, and a stable coloring π_{GH} of $G \cup H$ are given as input. Algorithm 2 shows a high-level description of the algorithm. For each color c in π_{GH}, we select the first vertex in $\pi_{GH}^{-1}(c)$ that is from $V(G)$ as the representative vertex, and compute the candidate set $C(rep(c))$ by checking each vertex in $\pi_{GH}^{-1}(c)$ whether the vertex is from $V(H)$. After the candidate sets are constructed, we compute edges between candidates as follows. For each $x \in C(r)$ of a representative vertex r, we first sort neighbors in $N_H(x)$ by their colors. Let $NC_x = (c_x^1, c_x^2, \ldots, c_x^{k_x})$ denote the ordered set of unique colors in the sorted

$N_H(x)$. The sorted $N_H(x)$ can be partitioned into k_x subarrays by the colors in NC_x. Hence, the i-th subarray of the sorted $N_H(x)$ stores $N^r_{rep(c^i_x)}(x)$. ($N^r_{r'}(x)$ in the compressed CS is defined just like $N^u_v(x)$ in the colored CS.) To access $N^r_{rep(c^i_x)}(x)$ directly, we maintain an array P_x such that $P_x[i]$ is the start position of color c^i_x in the sorted $N_H(x)$.

Example 3.3.1. *In Figure 3.1d, consider a candidate $x_1 \in C(u_1)$. The neighbors of x_1 are in ascending order of their colors, i.e., $N_H(x_1) = (x_6, x_7, x_2, x_5, x_3, x_4)$. There are three subarrays $(x_6, x_7), (x_2, x_5), (x_3, x_4)$ in $N_H(x_1)$, which correspond to $N^{u_1}_{u_6}(x_1), N^{u_1}_{u_2}(x_1), N^{u_1}_{u_3}(x_1)$, respectively.*

It takes $O(V)$ time to compute the candidate sets in the compressed CS. Sorting $N_H(x)$ for every $x \in V(H)$ takes $O(V + E)$ time as follows: for each edge $(x, y) \in E(H)$, make a pair $(x, \pi_{GH}(y))$; sort all edges in $E(H)$ by the lexicographic order of the pairs using the radix sort; get the sorted $N_H(x)$ for each $x \in V(H)$ from the sorted edges. Computing P_x for every $x \in V(H)$ takes $O(E)$ time because once $N_H(x)$ is sorted, we can compute P_x by scanning $N_H(x)$. Therefore, the time complexity of building the compressed CS is bounded by $O(V + E)$.

3.4 Backtracking and Partial Failing Sets

Backtracking is a recursive process that maps vertices in G to vertices in H so as to find an embedding of G in H. Extensive work has been done to develop a fast backtracking procedure, since it is the most time consuming part of the algorithms for subgraph isomorphism. Many practical solutions [25, 8, 31, 24] concern the two key issues in backtracking.

The first key issue is a matching order of vertices. We adopt the candidate-size order [24] as the matching order of our backtracking procedure.

Another key issue in backtracking is a search-space pruning technique. In `DAF` [24], a notion called *failing set* is introduced, which is used to prune out some parts of the search space. This pruning technique shows a good performance for subgraph isomorphism as the query size increases [24, 65]. However, since `DAF`'s failing set is usually a large part of the query graph, it is implemented using bit-arrays ($|V(q)|$ bits per vertex in query graph q), and the pruning technique requires $O(|V(q)|^2/w)$ space in total, where w is the word size. Thus, the technique may incur a big overhead especially in space when we deal with graph isomorphism on large-scale graphs. We introduce the *partial failing set*, which works in the same way as `DAF`'s failing set, while it is much smaller than `DAF`'s failing set. Therefore, the partial failing set enables us to apply the pruning technique to large-scale graphs without significant overheads.

Build DAG. A DAG G_D of G is used to compute a matching order in our backtracking procedure. We build the DAG by performing a breadth-first search (BFS). Since the vertices in the binary cells are mapped first, we execute a BFS from the vertices of G that belong to the binary cells of π_{GH}. In case there are no binary cells, we select a root r of G_D as

$$r \leftarrow \arg\min_{u \in V(G)} \frac{|\pi_{GH}^{-1}(\pi_{GH}(u))|}{\deg_G(u)},$$

and perform a BFS from the root. During BFS, we direct all edges from earlier to later visited vertices. Regarding the vertices in the same level, vertices are visited in descending order of vertex degrees.

Adaptive Matching Order with DAG Ordering. DAG ordering [24] is a matching order that follows a topological order of DAG G_D of G. Initially we have a partial embedding obtained by the binary cell mapping.

Definition 3.4.1. *An unmapped vertex u of G_D in a partial embedding M is* *called extendable with respect to M if all parents of u are mapped in M. A DAG ordering always selects an extendable vertex as the next matching vertex.*

For example, suppose that the current partial embedding of G in H is $M = \{(u_1, x_1), (u_4, x_4)\}$ in Figure 3.1. Extendable vertices are u_3, u_5, and u_6.

There can be multiple extendable vertices with respect to the current partial embedding. An adaptive matching order selects an extendable vertex that minimizes a certain value.

Definition 3.4.2. *Given a partial embedding M and an extendable vertex u, let $p_1, \ldots, p_k$ be the parents of u in G_D. The set of extendable candidates of u with respect to M is defined as $C_M(u) = \bigcap_{i=1}^{k} N_u^{p_i}(M(p_i))$.*

For example, suppose that the current partial embedding is $M = \{(u_1, x_1), (u_4, x_4)\}$ in Figure 3.1. The extendable candidates of u_5 is $C_M(u_5) = N_{u_5}^{u_1}(x_1) \cap N_{u_5}^{u_4}(x_4) = \{x_2, x_5\} \cap \{x_5\} = \{x_5\}$.

The *candidate-size order* [24, 36] selects an extendable vertex u such that $|C_M(u)|$ is the minimum. Since $C_M(u)$ is computed with respect to the partial embedding M, the next vertex selected in the candidate-size order may vary by different partial embeddings.

Search Tree. The process of a backtracking procedure can be illustrated as a search tree such that each internal node corresponds to a partial embedding, and each leaf node corresponds to an embedding of G in H or a mapping failure. There can be two types of mapping failures when trying to map the current matching vertex $u \in V(G)$: (1) a mapping conflict occurs when trying to map u to a candidate $x \in C(u)$ that is already mapped to another vertex in G; (2) there are no extendable candidates of u. According to the cases, the leaves are categorized into the following classes [24].

- A leaf belongs to the *conflict-class* if it corresponds to a conflict of mapping that tries to map the current matching vertex u to an already mapped candidate x. A conflict-class node is represented by $M = \{\dots, (v, x), \dots, (u, x)!\}$.

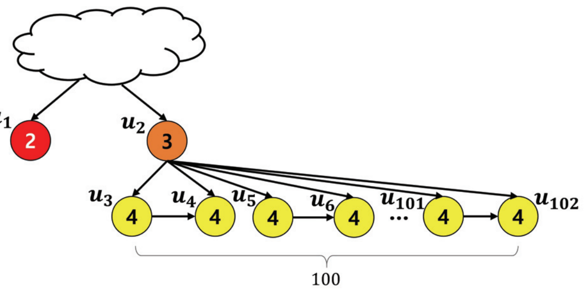
- A leaf belongs to the *emptyset-class* if it corresponds to the case that there are no extendable candidates of u. An emptyset-class node is represented by $M = \{\dots, (u, \emptyset)\}$.
- A leaf belongs to the *embedding-class* if it corresponds to an embedding of G in H.

In what follows, we use M to represent a node in the search tree as well as the corresponding partial embedding of G in H.

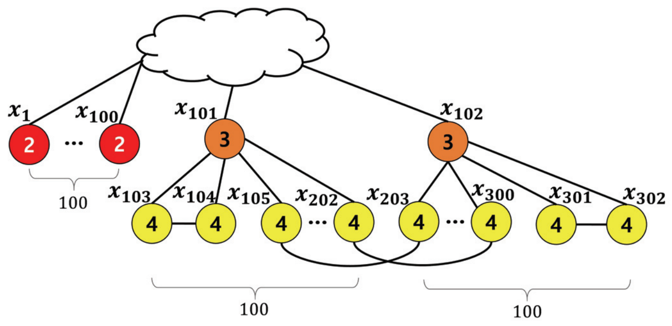
Example 3.4.1. *Figure 3.2c is a part of the search tree for DAG G_D in Figure 3.2a and graph H in Figure 3.2b. In Figure 3.2c, only the last mapping or the mapping failure is shown for each search tree node. Suppose that we just came back to node $M = \{\dots, (u_2, x_{101}), (u_3, x_{103}), (u_4, x_{104}), (u_1, x_1)\}$ after the exploration of the subtree rooted at M. During the exploration, we have tried all possible extensions to map u_5 and u_6, and all attempts to map u_6 have failed because there are no extendable candidates of u_6. Note that no matter how we change the mapping of u_1 in the current partial embedding M, it cannot lead to an embedding of G because all possible extensions will end up with failures in the same way.*

Failing Set. For a set of vertices $S \subseteq V(G)$ and a partial embedding M, let $M[S]$ denote the subset of M such that $(u, x) \in M[S]$ if and only if $(u, x) \in M$ and $u \in S$.

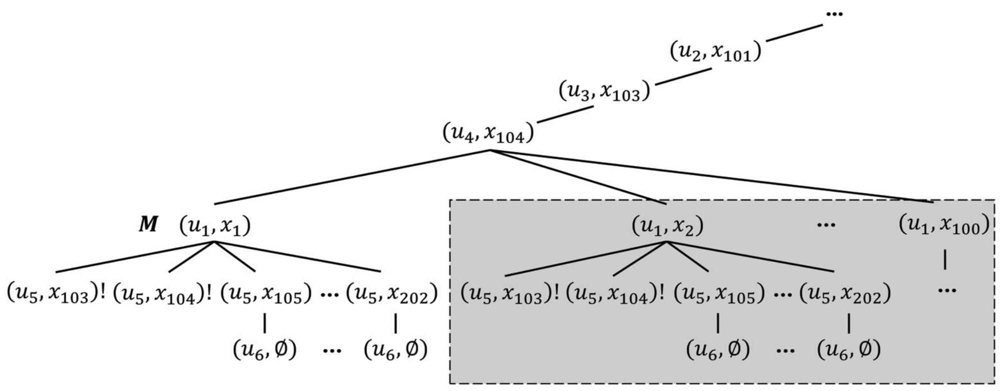
Definition 3.4.3. *A failing set of a node M in the search tree is a set of vertices $F_M \subseteq V(G)$ such that $M[F_M]$ cannot lead to an embedding of G by*



(a) G_D



(b) H



(c) Search tree

Figure 3.2: Running example of the partial failing set. (a–b) Colors appear within the vertices. (c) Shaded parts are redundant.

extending $M[F_M]$.

Lemma 3.4.1 ([24]). *Let M be a node in the search tree whose last mapping is (u, x) in the corresponding partial embedding, and let F_M be a non-empty failing set of node M. If $u \notin F_M$, then all siblings of node M (including itself) cannot lead to an embedding of G.*

Partial Failing Set. We define the *partial failing set F_M* for a leaf in the search tree as follows:

- For a conflict-class node $M = \{\dots, (v, x), \dots, (u, x)!\}$, the partial failing set is $F_M = \{u, v\}$.
- For an emptyset-class node $M = \{\dots, (u, \emptyset)\}$, the partial failing set is $F_M = \{u\}$.
- For an embedding-class node M, the partial failing set is the empty set.

The partial failing set F_M of an internal node is computed by the partial failing sets of its children. Suppose that an internal node M has k children $M_1, \dots, M_k$ that are all extensions of M to the next vertex u_c. Assume that we have computed the failing sets $F_{M_1}, \dots, F_{M_k}$ of $M_1, \dots, M_k$, respectively. We compute the partial failing set F_M of node M according to the following cases:

Case 1. If there exists a child node M_i such that $F_{M_i} = \emptyset$, we set $F_M = \emptyset$.

Case 2. Otherwise,

Case 2.1. If there exists a child node M_i such that $u_c \notin F_{M_i}$, we set $F_M = F_{M_i}$.

Case 2.2. Otherwise, we set $F_M = (\bigcup_{i=1}^{k} F_{M_i} - \{u_c\}) \cup \text{parent}(u_c)$, where $\text{parent}(u_c)$ denotes the set of parents of u_c in G_D.

The partial failing set of M is designed to contain only the vertices in DAF's failing set of M that are necessary to check the condition in Lemma 3.4.1.

Let M be a node in the search tree whose last mapping is (u, x) in the corresponding partial embedding, and let F_M be a non-empty partial failing set of node M. If $u \notin F_M$, we prune the siblings of node M.

Example 3.4.2. *In the search tree in Figure 3.2c of Example 3.4.1, the partial failing set of a leaf* $(u_5, x_{103})!$ *is* $\{u_3, u_5\}$*, that of* $(u_5, x_{104})!$ *is* $\{u_4, u_5\}$*, and that of* $(u_6, \emptyset)$ *is* $\{u_6\}$*. The partial failing set of each internal node* (u_5, x_i) *for* $105 \leq i \leq 202$ *is* $(\{u_6\} - \{u_6\}) \cup \{u_2, u_5\} = \{u_2, u_5\}$*. The partial failing set* F_M *of* M *is* $((\{u_3, u_5\} \cup \{u_4, u_5\} \cup \{u_2, u_5\}) - \{u_5\}) \cup \{u_2\} = \{u_2, u_3, u_4\}$*. Since* u_1 *is not in* F_M*, we do not extend all other siblings of node* M *that maps* u_1 *to another candidate (shaded in Figure 3.2c), and we set the partial failing set of the internal node* (u_4, x_{104}) *to* $\{u_2, u_3, u_4\}$ *by Case 2.1.*

Lemma 3.4.2. *The partial failing set has the same pruning effect as DAF's failing set (i.e., Lemma 3.4.1 holds for the partial failing set).*

Proof. Given a search tree node M, let F_M and F'_M denote the partial failing set of M and DAF's failing set of M, respectively. The differences between F_M and F'_M are as follows: (1) If $M = \{\ldots, (v, x), \ldots, (u, x)!\}$ belongs to the conflict-class, F'_M is set to $\text{anc}(u) \cup \text{anc}(v)$, where $\text{anc}(u)$ denotes the set of all ancestors of u in G_D including u itself; (2) if $M = \{\ldots, (u, \emptyset)\}$ belongs to the empty-set class, F'_M is set to $\text{anc}(u)$; (3) In Case 2.2 of the definition of the partial failing set, F'_M is set to $\bigcup_{i=1}^{k} F'_{M_i}$. In other cases, F'_M is set just like F_M.

Consider a search tree node M whose last mapping is (u, x), and suppose that F_M and F'_M are not empty. Since DAF's failing set was proved to be a failing set [24], siblings of M are pruned out if u is not in F'_M due to Lemma 3.4.1. To prove the lemma, therefore, we need to prove that $u \notin F_M$ if and only if $u \notin F'_M$.

For a set of vertices $S \subseteq V(G)$ and a search tree node M, let $S[M]$ denote the subset of S such that $v \in S[M]$ if and only if $v \in S$ and $(v, y) \in M$ for some y. Given a DAG g and a set of vertices $A \subseteq V(g)$, let $\mathit{Leaf}_g(A)$ denote the set of leaves in $g[A]$. For each search tree node M, the following invariant is satisfied: (1) $F_M \subseteq F'_M$, and (2) $\mathit{Leaf}_{G_D}(F'_M[M]) \subseteq F_M$.

We prove that $u \notin F_M$ if and only if $u \notin F'_M$ as follows.

- If $u \notin F'_M$, then $u \notin F_M$ because $F_M \subseteq F'_M$.
- If $u \in F'_M$, then $u \in \mathit{Leaf}_{G_D}(F'_M[M])$ because we use the DAG ordering and (u, x) is the last mapping of M. Since $\mathit{Leaf}_{G_D}(F'_M[M]) \subseteq F_M$, we get $u \in F_M$.

Therefore, the partial failing set has the same pruning effect as `DAF`'s failing set.

Let M denote a search tree node. We prove the invariant by an induction on M. If M is a leaf, there are three cases as follows: If $M = \{\ldots, (v, x), \ldots, (u, x)!\}$ is a conflict-class node, $\mathit{Leaf}_{G_D}(F'_M[M]) \subseteq \{u, v\}$; If $M = \{\ldots, (u, \emptyset)\}$ is an emptyset-class node, $\mathit{Leaf}_{G_D}(F'_M[M]) = \{u\}$; If M is an embedding-class node, $\mathit{Leaf}_{G_D}(F'_M[M]) = \emptyset$. Thus, the invariant holds when M is a leaf. Consider an internal node M in the search tree. Let's assume that the invariant holds for the search tree nodes that are proper descendants of M, and we show that it holds for M. We consider only Case 2.2 because F_M is set just like F'_M in other cases and thus the proof is trivial. In Case 2.2, note that $u_c \in F'_{M_i}$ for every $1 \le i \le k$, so we have $u_c \in F'_M$. We first show that $F_M \subseteq F'_M$ (the first item of the invariant). By induction hypothesis, we have $\bigcup_{i=1}^{k} F_{M_i} \subseteq F'_M$. Since $u_c \in F'_M$, we have $\mathrm{anc}(u_c) \in F'_M$, and thus $\mathrm{parent}(u_c) \subseteq F'_M$. Therefore, we get $F_M \subseteq F'_M$. For the second item of the invariant, we rewrite the left hand side by substituting F'_M:

$$\begin{aligned} \mathit{Leaf}_{G_D}(F'_M[M]) &= \mathit{Leaf}_{G_D}((\bigcup_{i=1}^{k} F'_{M_i})[M]) \\ &= \mathit{Leaf}_{G_D}(\bigcup_{i=1}^{k}(F'_{M_i}[M])). \end{aligned} \tag{3.1}$$

For a DAG g and two sets of vertices $A, B \subseteq V(g)$, we can derive that $\mathit{Leaf}_g(A \cup B) \subseteq \mathit{Leaf}_g(A) \cup \mathit{Leaf}_g(B)$ by definition of Leaf_g. Thus,

$$\mathit{Leaf}_{G_D}(\bigcup_{i=1}^{k}(F'_{M_i}[M])) \subseteq \bigcup_{i=1}^{k} \mathit{Leaf}_{G_D}(F'_{M_i}[M]). \tag{3.2}$$

Since $M = M_i - \{(u_c, x)\}$ for some x, we have that $u_c \notin F'_{M_i}[M]$ and that some vertices of parent(u_c) may be included in $\mathit{Leaf}_{G_D}(F'_{M_i}[M])$. Thus,

$$\mathrm{RHS}_{3.2} \subseteq (\bigcup_{i=1}^{k} \mathit{Leaf}_{G_D}(F'_{M_i}[M_i]) - \{u_c\}) \cup \mathrm{parent}(u_c), \tag{3.3}$$

where $\mathrm{RHS}_{3.2}$ is the right hand side of Equation (3.2). Lastly, by induction hypothesis, we get

$$\begin{aligned} \mathrm{RHS}_{3.3} &\subseteq (\bigcup_{i=1}^{k} F_{M_i} - \{u_c\}) \cup \mathrm{parent}(u_c) \\ &= F_M, \end{aligned} \tag{3.4}$$

where $\mathrm{RHS}_{3.3}$ is the right hand side of Equation (3.3).

Therefore, we have proved that the invariant holds for each search tree node M. □

Algorithm 3 shows the recursive backtracking procedure using adaptive matching order and the pruning technique by the partial failing sets. In our

Algorithm 3: Backtrack(G_D, CS, M)

```
if $|M| = |V(G_D)|$ then
    /* Found an embedding */
    return $\emptyset$;
else
    $u_c \leftarrow$ select an extendable vertex with min $|C_M(u_c)|$;
    if $C_M(u_c) = \emptyset$ then
        $F_M \leftarrow$ parent$(u_c)$;
    else
        $F_M \leftarrow \emptyset$;
        for each $x \in C_M(u_c)$ do
            if $x$ is unvisited then
                $M_c \leftarrow M \cup \{(u_c, x)\}$;
                Mark $x$ as visited;
                $F_{M_c} \leftarrow Backtrack(G_D, CS, M_c)$;
                Mark $x$ as unvisited;
                if $F_{M_c} = \emptyset$ then
                    return $\emptyset$;
                else
                    if $u_c \notin F_{M_c}$ then
                        $F_M \leftarrow F_{M_c}$;
                        break;
            else
                $F_{M_c} \leftarrow \{u_c, M^{-1}(x)\}$;
            $F_M \leftarrow F_M \cup F_{M_c}$;
        if $u_c \in F_M$ then
            $F_M \leftarrow (F_M - \{u_c\}) \cup$ parent$(u_c)$;
    return $F_M$;
```

implementation, the recursive procedure is realized by an iterative procedure that traverses the search tree in depth-first order. Thus, as soon as an embedding of G in H is found (Line 2), we can immediately terminate the procedure.

The partial failing set of a search tree node is implemented using a sorted array. For a partial embedding M, we need $|F_M|$ space to store the partial failing set, and membership operation can be done in $O(\log |F_M|)$ time. Union operation for two partial failing sets F_{M1} and F_{M2} can be done in $O(|F_{M1}| + |F_{M2}|)$ time.

3.5 Performance Evaluation

We compare our algorithm with state-of-the-art GI solutions on real-world datasets. The following algorithms are evaluated.

- `Traces` [51]: graph canonization algorithm.
- `nauty` [51]: graph canonization algorithm.
- `DAF` [24]: subgraph isomorphism algorithm.
- `CFL-Match` [8]: subgraph isomorphism algorithm.
- `VF2++` [31]: subgraph isomorphism algorithm.
- `CRaB`: our algorithm (Color Refinement and Backtracking).

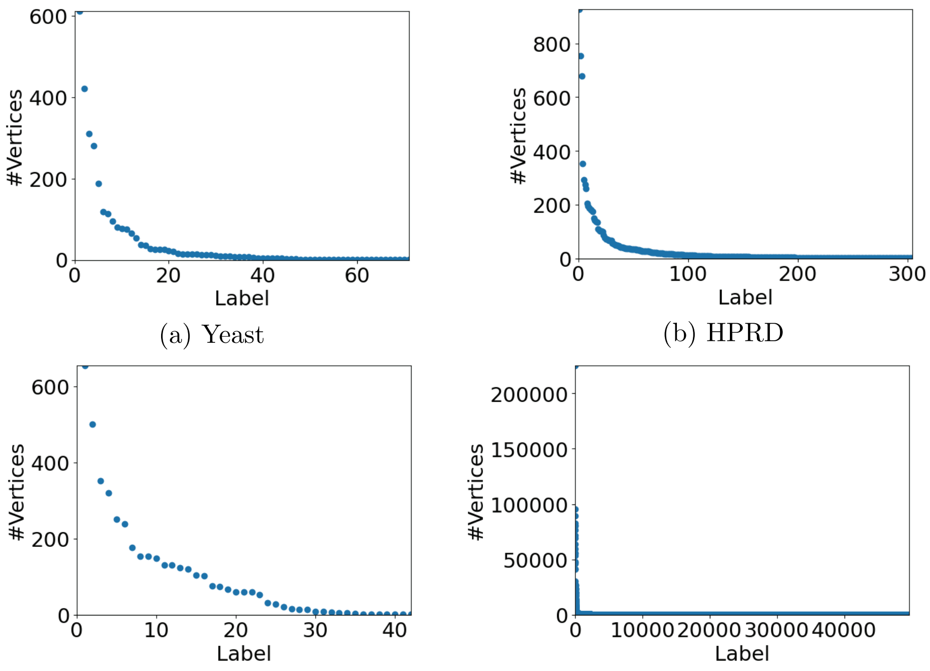
All the source codes were obtained from the authors of previous papers. `Traces` and `nauty` are implemented in C, while `DAF`, `CFL-Match`, `VF2++`, and `CRaB` are implemented in C++. We use version 2.6r12 of `Traces` and `nauty`. Experiments are conducted on a machine with two Intel Xeon E5-2680 v3 2.50GHz CPUs and 256GB memory running CentOS Linux.

Table 3.1: Characteristics of datasets.

Dataset (G)	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	$\lvert\Sigma\rvert$	avg-deg(G)
Yeast	2,974	12,442	71	8.37
HPRD	9,045	34,853	307	7.71
Human	4,271	84,890	44	39.75
YAGO	4,202,965	11,354,645	49,676	5.40
Hamster	1,788	12,476	-	13.96
GrQc	4,158	13,422	-	6.46
HepTh	8,638	24,806	-	5.74
Facebook	4,039	88,234	-	43.69
CondMat	21,363	91,286	-	8.55
HepPh	11,204	117,619	-	21.00
Email	33,696	180,811	-	10.73
AstroPh	17,903	196,972	-	22.00
Brightkite	56,739	212,945	-	7.51
Plus	283,872	428,384	-	3.02
Amazon	334,863	925,872	-	5.53
Gowalla	196,591	950,327	-	9.67
DBLP	317,080	1,049,866	-	6.62
Adaptec	870,532	1,874,507	-	4.31
RoadNet	1,957,027	2,760,388	-	2.82
Youtube	1,134,890	2,987,624	-	5.27
Bigblue	3,795,055	8,712,138	-	4.59
Skitter	1,694,616	11,094,209	-	13.09
LiveJournal	3,997,962	34,681,189	-	17.35

We note that the *minimum DFS code* [72] has been used to test graph isomorphism between subgraphs of a social network [80], and `gSpan` [72] is an available solution to compute the minimum DFS code. In our experiment, however, `gSpan` could run only on graphs with a few dozens of vertices since its memory usage was very high. Thus, we do not include `gSpan` in our comprehensive experiments.

Datasets. We use 23 real-world graphs in Table 3.1. If a graph is disconnected, we extract the largest connected component of the graph and record the char-



(a) Yeast (b) HPRD

(c) Human (d) YAGO

Figure 3.3: Label distributions of datasets.

acteristics of the component. Yeast, HPRD, and Human [8, 25, 24] are protein-protein interaction (PPI) networks, where vertices represent proteins and edges represent interactions between proteins. Vertices in the PPI networks are labeled by *Gene Ontology* information [26]. YAGO [63] is a resource description framework (RDF) dataset, which is transformed into a graph dataset by applying the *type-aware transformation* [37]. Figure 3.3 shows the label distributions of Yeast, HPRD, Human, and YAGO, where the labels on the x-axis are sorted by the number of vertices. We can see that some labels are assigned to many vertices, while most of the labels are assigned to a few vertices.

We also cover unlabeled graphs from various domains: Hamster, Facebook, Brightkite, Gowalla, Youtube, and LiveJournal [42, 43] are online social networks, where edges represent interactions between users. GrQc, HepTh, CondMat, HepPh, AstroPh, and DBLP [43] are collaboration networks, where vertices represent scientists and edges represent collaborations (co-authoring a paper). Plus and Skitter [43, 49] are router interconnection networks and internet topology graph, respectively. Email, Amazon, and RoadNet [43] are email communication network, product co-purchasing network, and road network, respectively. Adaptec and Bigblue [49] are graphs derived from circuits, where vertices represent electronic components and nodes in circuits, and edges represent connections between components and nodes.

Isomorphic Pairs. We generate five isomorphic graphs for each dataset by randomly permuting the vertices. An isomorphic graph pair consists of an original dataset and a generated isomorphic graph.

Nonisomorphic Pairs. We generate five nonisomorphic graphs for each dataset in the following way. For each isomorphic graph generated above, we randomly swap two edges until it becomes nonisomorphic, while maintaining the degree sequence so that the nonisomorphism is not revealed by a simple check. A nonisomorphic graph pair consists of an original dataset and a generated nonisomorphic graph.

Metrics. To evaluate an algorithm on isomorphic (or nonisomorphic) graph pairs of a dataset, we measure the average running time for processing each graph pair. Since `Traces` is a randomized algorithm, we run each of the five graph pairs three times and measure the average running time of 5×3 runs when we evaluate `Traces`. We set the time limit for processing a graph pair to 5 hours. If an algorithm on a pair does not finish within the time limit, we regard the running time of the run as 5 hours and include it to the average time.

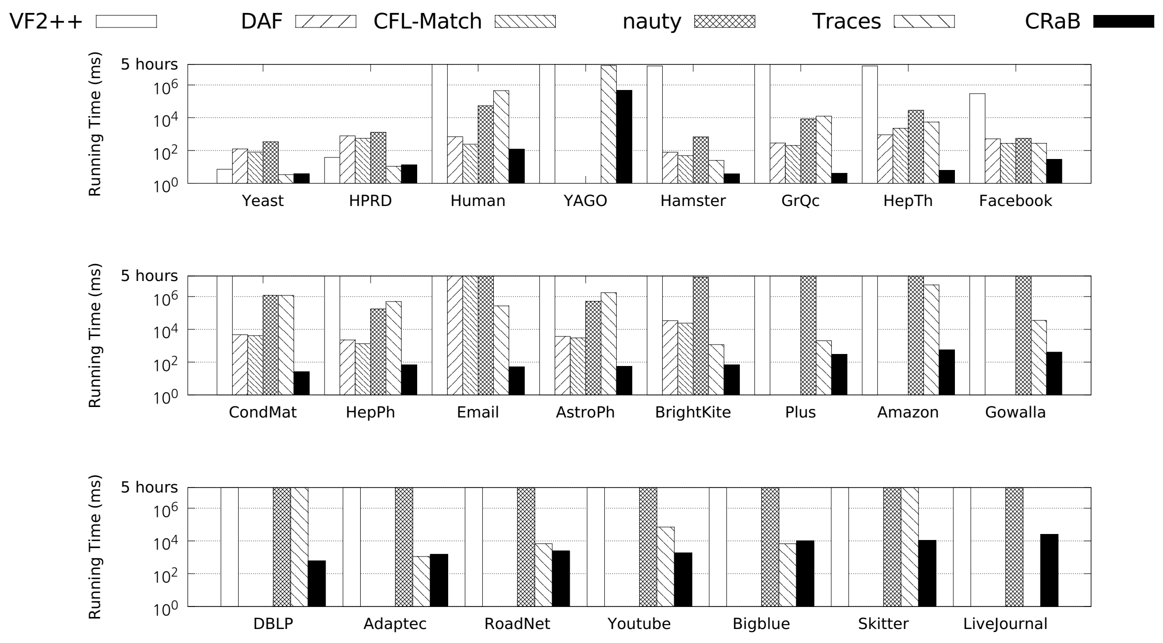


Figure 3.4: Running time of `VF2++`, `DAF`, `CFL-Match`, `nauty`, `Traces`, and `CRaB` for isomorphic graph pairs.

3.5.1 Comparing with Existing Solutions

In this section we compare our algorithm to existing solutions on real-world graphs. In Figures 3.4 and 3.5, missing bars indicate that an algorithm ran out of memory (256GB).

Isomorphic Pairs. Figure 3.4 shows the results on the isomorphic graph pairs. Our algorithm outperforms state-of-the-art solutions in almost all datasets by up to 4 orders of magnitude (see DBLP in Figure 3.4), and finishes all 23 datasets within the time limit. Our algorithm outperforms existing solutions on 19 datasets, and runs about 2,430 times faster than the fastest existing solutions on average. For the remaining 4 datasets (Yeast, HPRD, Adaptec, Bigblue), our algorithm runs marginally slower (about 1.3 times) than `Traces`, but much faster than other algorithms. In each of these 4 datasets, `Traces` computes the canonical form after one color refinement. In such a case, `CRaB` runs slightly slower than `Traces` because `CRaB` performs backtracking after the pairwise color refinement. In other datasets, however, the average number of iterations of color refinement in `Traces` is much higher (about 418,680).

Subgraph isomorphism algorithms can solve small datasets, but they are unable to handle large datasets as they run out of memory or exceed the time limit. `DAF` and `CFL-Match` fail to process large-scale datasets due to out-of-memory errors (e.g., YAGO, Plus, LiveJournal). `VF2++` does not finish within the time limit for most of the datasets. The results show that the standard backtracking approach for subgraph isomorphism is unable to handle large real-world graphs.

Regarding the graph canonization approach, `Traces` solves 20 datasets within the time limit, while `nauty` solves 11 datasets. Sometimes `nauty` is faster than `Traces` (e.g., Human, GrQc, HepPh, AstroPh), but in many datasets `Traces` outperforms `nauty`.

To analyze the performance of the backtracking procedure of `CRaB`, we measure the average number of candidates of a vertex, the number of search tree nodes, and the time used for backtracking in Table 3.2. Since the vertices that belong to the binary cells are mapped before backtracking, they are excluded when computing the average number of candidates.

In general, the backtracking takes more time when the number of candidates is large (see YAGO, Skitter, Plus, and Youtube) because the number of search tree nodes can be large. When two datasets have similar numbers of search tree nodes, the backtracking takes more time on the dataset with higher average degree because more edges should be checked for mapping a vertex. For

Table 3.2: Analysis of backtracking procedure of `CRaB`.

Dataset (G)	avg. $\|C(u)\|$	#search tree nodes	running time (ms)
Yeast	3.49	282	0.08
HPRD	2.00	167	0.05
Human	62.27	84,480	172.44
YAGO	787.69	513,024,341	34777.32
Hamster	3.18	1,100	0.43
GrQc	5.13	2,398	3.68
HepTh	3.45	3,107	2.38
Facebook	2.99	381	0.28
CondMat	2.97	13,582	8.43
HepPh	8.44	6,414	123.78
Email	3.51	18,026	9.05
AstroPh	4.91	9,302	28.75
Brightkite	4.58	5,337	1.86
Plus	99.59	1,011,479	101.39
Amazon	2.77	128,207	47.24
Gowalla	3.69	20,626	5.13
DBLP	3.22	227,180	126.64
Adaptec	2.57	84,263	5.98
RoadNet	2.01	44,110	11.21
Youtube	29.17	592,730	51.20
Bigblue	3.59	81,940	22.59
Skitter	18.37	269,773,419	18304.72
LiveJournal	4.62	349,633	454.62

example, Human and Adaptec have similar numbers of search tree nodes, but the backtracking on Human takes more time than that on Adaptec, where the average degree of Human (39.75) is much higher than that of Adaptec (4.31).

Nonisomorphic Pairs. Figure 3.5 shows the results on the nonisomorphic graph pairs, which are generally analogous to the results on the isomorphic pairs. After color refinement, our algorithm immediately determined that two graphs are not isomorphic. In fact, it is known that color refinement can well distinguish nonisomorphic graphs [4, 7, 1]. Our algorithm is always the best

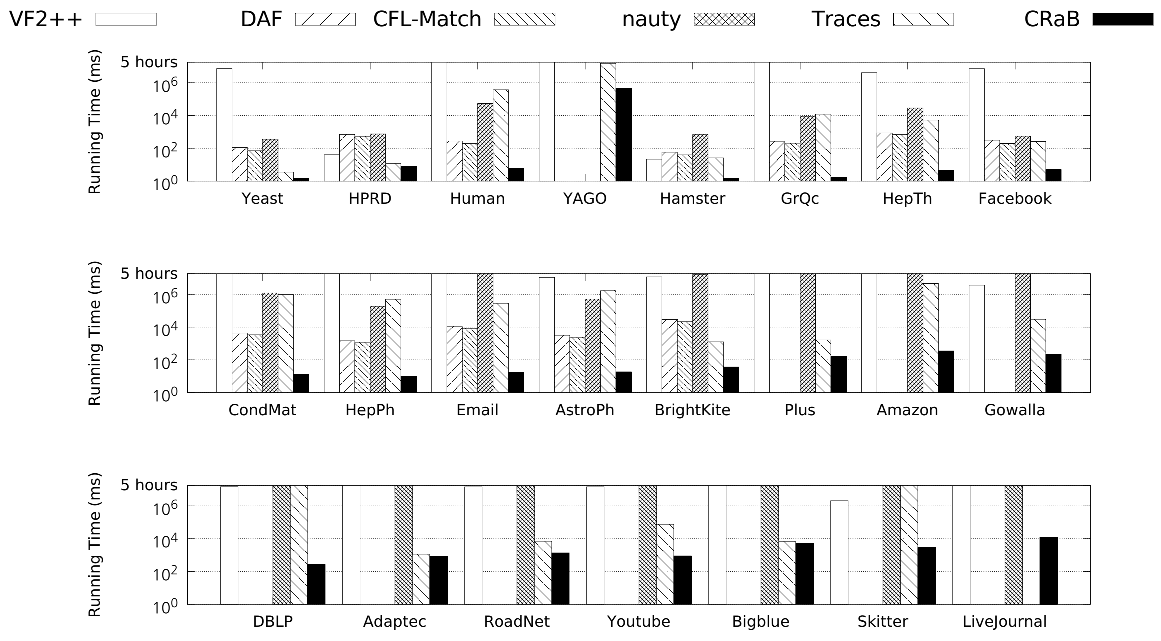


Figure 3.5: Running time of `VF2++`, `DAF`, `CFL-Match`, `nauty`, `Traces`, and `CRaB` for nonisomorphic graph pairs.

for nonisomorphic graph pairs, and outperforms the fastest existing solutions by up to 4 orders of magnitudes (see Amazon and DBLP in Figure 3.5). Our algorithm runs about 3,160 times faster than the fastest existing solutions on average.

Since `nauty` and `Traces` are graph canonization algorithms, they can determine that two graphs are not isomorphic after computing canonical forms of the graphs. Thus, the running times of these algorithms on nonisomorphic pairs are similar to those on isomorphic pairs.

3.5.2 Effectiveness of Individual Techniques

In this section we evaluate the practical impact of each of the new techniques introduced in this paper. Since the techniques are relevant to the backtracking, only the isomorphic graph pairs are used in these experiments.

Effect of Compressed CS. To evaluate the effect of the compressed CS, we compare the size of the compressed CS against the size of the CS computed by `DAF` [24]. We also compare the compressed CS with the colored CS in terms of the size. The size of a CS is the number of vertices and edges in the CS.

Table 3.3 shows the compression ratios, where CR1 is the size of `DAF`'s CS over the size of the compressed CS, and CR2 is the size of the colored CS over the size of the compressed CS. The highest compression ratio is 476.47 on YAGO.

The compression ratio is related to the number of colors in the stable coloring. The fewer colors the coloring has, the higher compression ratio is achieved. When there is a small number of colors, many vertices are colored by a color and thus there are many duplications in the colored CS. In the compressed CS, however, there are no duplications because it stores a candidate set only once for each color. CR1 is always larger than or equal to CR2 because the colored CS cannot be further refined by the CS refinement technique in `DAF`. In Table 3.3, we use $\rho = |\pi_{GH}|/|V(G)|$ to measure the number of colors over the number of vertices. The compression ratios on Human, YAGO, Email, Plus, and Youtube are high (CR1 ranges from 14.80 to 476.47) as the numbers of colors in the datasets are small (ρ ranges from 0.46 to 0.60). On HPRD, Facebook, and RoadNet, compression ratios are lower than 1.07 as the numbers of colors are large (ρ ranges from 0.96 to 0.99).

Effect of Binary Cell Mapping. In many real-world graphs, stable colorings

Table 3.3: Effect of the compressed CS (columns 2–4) and the binary cell mapping (columns 5–6). CR1 = (`DAF`'s CS size)/(compressed CS size), CR2 = (colored CS size)/(compressed CS size), $\rho = |\pi_{GH}|/|V(G)|$, and β is the proportion of vertices that belong to binary cells.

Dataset (G)	CR1	CR2	ρ	β	speedup
Yeast	1.07	1.05	0.95	92%	1.31
HPRD	1.01	1.01	0.99	98%	1.74
Human	54.48	11.19	0.46	32%	1.19
YAGO	476.47	340.81	0.57	53%	1.02
Hamster	1.61	1.29	0.81	71%	1.75
GrQc	49.43	3.98	0.78	64%	2.00
HepTh	12.91	1.85	0.86	76%	2.05
Facebook	1.06	1.03	0.96	94%	5.13
CondMat	2.63	1.52	0.79	66%	2.18
HepPh	68.26	6.45	0.78	68%	2.64
Email	19.27	14.57	0.60	50%	2.16
AstroPh	10.06	2.08	0.82	72%	2.15
Brightkite	1.63	1.47	0.83	75%	1.97
Plus	14.80	10.70	0.57	43%	1.08
Amazon	1.44	1.27	0.90	83%	1.46
Gowalla	8.52	6.64	0.89	83%	1.92
DBLP	7.31	1.98	0.74	58%	1.42
Adaptec	1.12	1.09	0.94	90%	1.57
RoadNet	1.04	1.03	0.98	96%	1.36
Youtube	59.24	44.63	0.60	52%	1.92
Bigblue	1.18	1.13	0.95	91%	1.62
Skitter	4.01	3.02	0.80	74%	2.27
LiveJournal	4.58	1.25	0.93	88%	3.13

Table 3.4: Average number of vertices and space usages of `DAF`'s failing set and the partial failing set.

Dataset	average #vertices		space usage (in KB)	
	`DAF`	`CRaB`	`DAF`	`CRaB`
Yeast	17.64	1.89	1,080	22
HPRD	2.62	1.13	9,987	40
Human	220.04	2.91	2,227	49
Hamster	50.43	1.85	390	13
GrQc	27.15	0.19	2,110	3
HepTh	21.83	0.33	9,108	11
Facebook	7.86	0.45	1,991	7
CondMat	154.61	0.12	55,710	10
HepPh	312.11	0.17	15,323	7
AstroPh	746.23	0.08	39,126	6
Brightkite	58.94	1.89	392,983	419

contain large numbers of binary cells. In Table 3.3, we use β to denote the proportion of vertices that belong to the binary cells.

To see the effect of binary cell mapping, we compare our algorithm and another version of our algorithm that does not use binary cell mapping. The results are shown in Table 3.3 (see speedup). The technique improves the performance of our algorithm by up to 5.13 times (see Facebook in Table 3.3) and 1.96 times on average. The improvement is related to the number of binary cells of a dataset. For instance, on Facebook, HepPh, and LiveJournal, the improvement is high because their proportions of vertices that belong to the binary cells are high. The proportions on Facebook, HepPh, and LiveJournal are 94%, 68%, and 88%, respectively. On Human and Plus, the improvement is small because the proportion is small. The proportions on Human and Plus are 32% and 43%, respectively.

Effect of Partial Failing Set. To see the effect of the partial failing set, we compare it with `DAF`'s failing set in terms of (1) the average number of vertices in the (partial) failing set of a search tree node and (2) the space usage. The space usage of `DAF`'s failing set is V^2/w bits, where $w = 32$ is the word size, and that of the partial failing set is $V \cdot |F_M| \cdot 4$ bytes, where $|F_M|$ is the average number of vertices in the partial failing set. Table 3.4 shows the results for the datasets that are solved by both `DAF` and `CRaB`. The average number of vertices and the space usage of the partial failing set are much smaller than those of `DAF`'s failing set.

3.5.3 Analysis with Varying Degrees of Similarity

In this section we analyze the performances of the algorithms with varying degrees of similarity between two input graphs. For each dataset in Table 3.1, we generate a nonisomorphic graph by randomly swapping two edges k times. We set $k = 20, 40, 60, 80, 100$. A generated graph is less similar to the original graph if more edges are swapped. We do not maintain the degree sequence of the original graph when generating nonisomorphic graphs, and each algorithm runs without checking the degree sequence. We set a time limit of one hour for processing a graph pair.

Figures 3.6, 3.7, and 3.8 show the running time of the comparing algorithms with different value of k. Missing lines indicate that an algorithm ran out of memory (256GB). Similar to the results of main experiments in Section 3.5.1, `CRaB` outperforms state-of-the-art algorithms for all the datasets. Overall, the performances of the algorithms do not significantly affected by k in all the datasets except for `VF2++` on three datasets.

On GrQc, CondMat, and RoadNet, the running time of `VF2++` visibly decreases as k increases. `VF2++` computes the candidate set of a vertex during

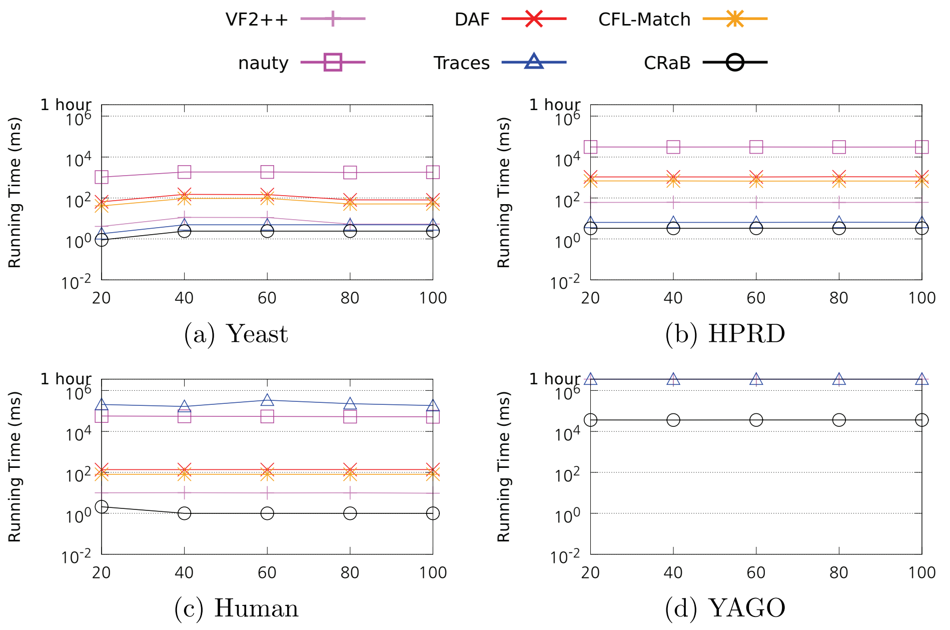


Figure 3.6: Analysis with varying degrees of similarity I.

backtracking. When k increases, the number of candidates decreases, and thus the backtracking time of `VF2++` significantly decreases in these datasets. In contrast, `CRaB`, `DAF`, and `CFL-Match` compute the candidate set of every vertex before backtracking. After computing the candidate sets, all of these three algorithms determined that two graphs are not isomorphic since there was a vertex with an empty candidate set. Therefore, the running time of `CRaB`, `DAF`, and `CFL-Match` does not change significantly with different value of k. Since the nonisomorphic graphs share common parts with the original graph, computing the canonical form of a nonisomorphic graph and computing the canonical form of the original graph take more or less the same time.

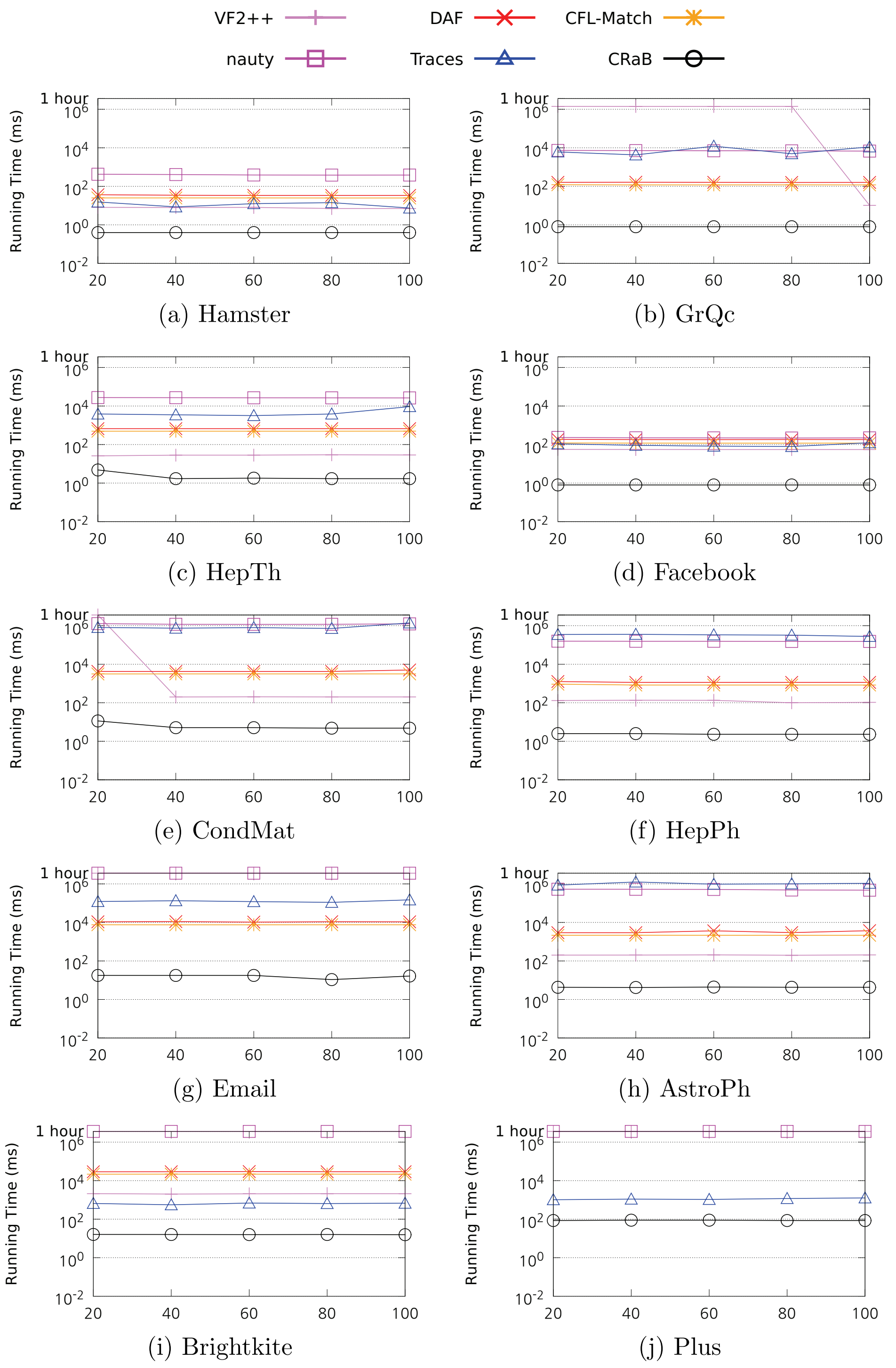


Figure 3.7: Analysis with varying degrees of similarity II.

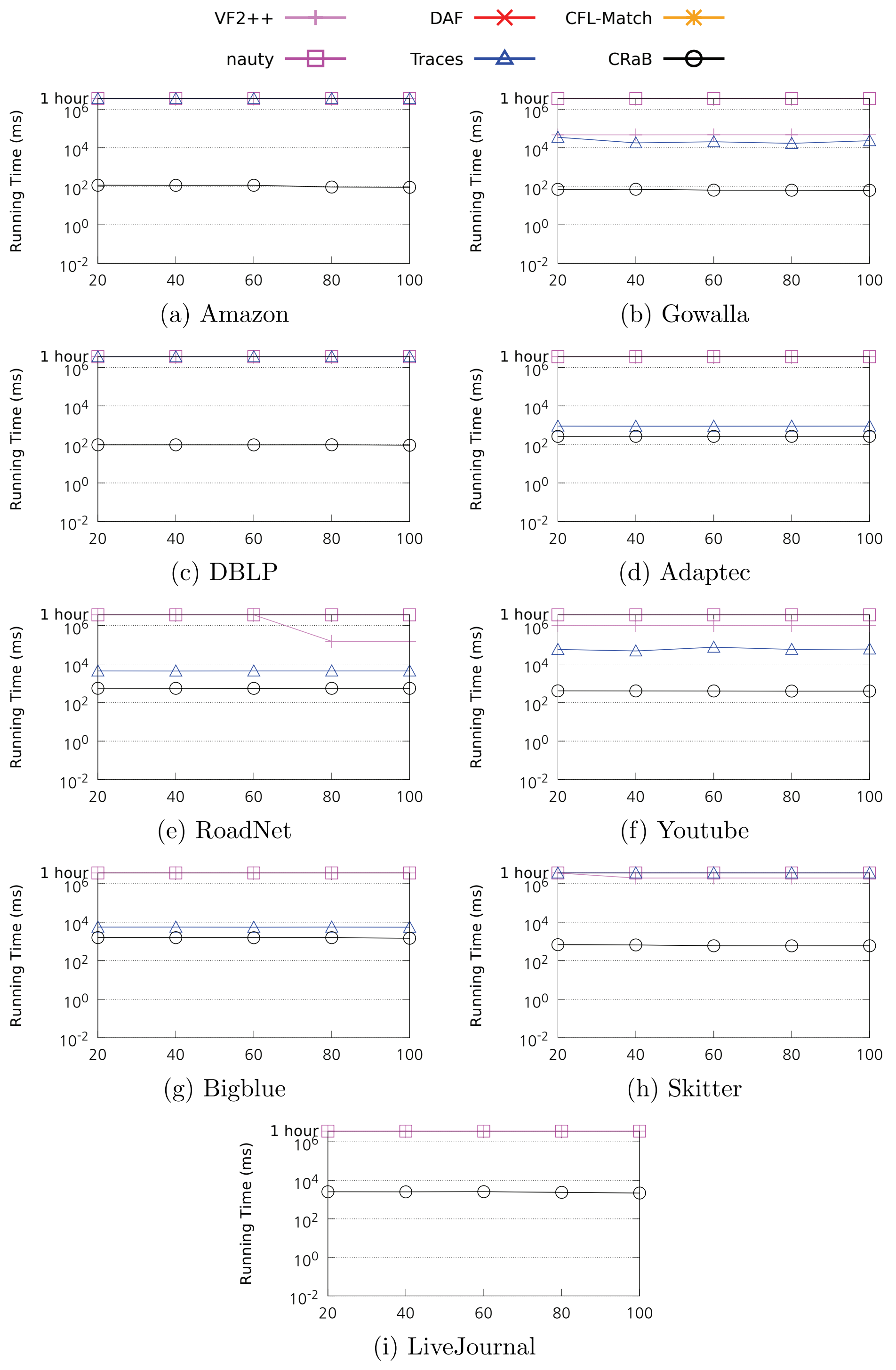


Figure 3.8: Analysis with varying degrees of similarity III.

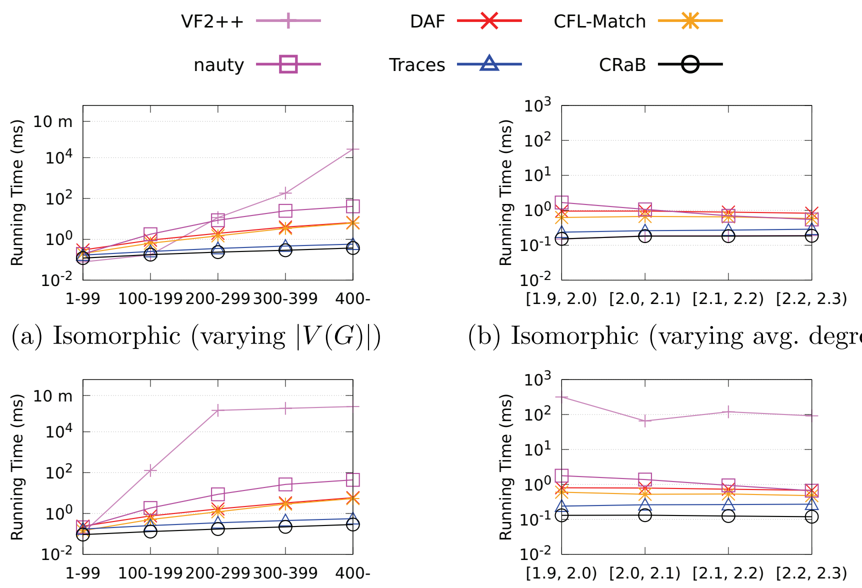
3.5.4 Sensitivity Analysis

In this section we conduct a sensitivity analysis to see how the changes in input parameters affect the running time of the algorithms using real-world graphs. We obtain 499,963 chemical compound structures from PubChem [66], the open chemistry database at the National Institutes of Health. A dataset has up to 801 vertices and 19 labels. We vary two parameters:

- number of vertices in the graph: 1–99, 100–199, 200–299, 300–399, 400 and more,
- average degree of the graph: $[1.9, 2.0)$, $[2.0, 2.1)$, $[2.1, 2.2)$, $[2.2, 2.3)$.

The default values for $|V(G)|$ and the average degree are 100–199 and $[2.0, 2.1)$, respectively. For each interval, we generate 500 isomorphic/nonisomorphic graph pairs as follows: We randomly sample a graph 100 times and generate five isomorphic graphs and five nonisomorphic graphs for each sampled graph as before. We set the time limit for processing a graph pair to 10 minutes in this experiment.

Figures 3.9a and 3.9b show the results on the isomorphic graph pairs. In Figure 3.9a, a general trend is that the running time grows as $|V(G)|$ increases. When there are less than 200 vertices, `VF2++` outperforms all the other algorithms since it is a much simpler algorithm than the others. However, when there are 200 vertices or more, its running time grows exponentially. This is because `VF2++` lacks the ability to reduce the candidate space, compared to other backtracking-based solutions. Unlike `VF2++`, the running time of our algorithm grows gradually, being the runner-up for the first two intervals, and the winner for the rest. Figure 3.9b shows that different average degrees have little effect on the running time. In all intervals of degrees, `VF2++` and `CRaB` outperform all



(a) Isomorphic (varying $|V(G)|$) (b) Isomorphic (varying avg. degrees)

(c) Nonisomorphic (varying $|V(G)|$) (d) Nonisomorphic (varying avg. degrees)

Figure 3.9: Sensitivity analysis.

the other algorithms, as in the interval 100–199 of the experiment on varying $|V(G)|$.

Figures 3.9c and 3.9d show the results on the nonisomorphic graph pairs, where `CRaB` outperforms all the other algorithms in every interval. When input graphs are not isomorphic, a backtracking-based algorithm must traverse its whole search space to solve the problem. Nevertheless, `CRaB`, `DAF`, and `CFL-Match` still performs well, since they use techniques to reduce the search space. In contrast, the average running time of `VF2++` is significantly increased due to its lack of such techniques.

Chapter 4

Graph Isomorphism Query Processing

Given a set of data graphs $\mathcal{D} = \{G_1, G_2, \ldots, G_k\}$ and a query graph q, the graph isomorphism query processing (GIQP) problem is to find all the data graphs in $\mathcal{D}$ that are isomorphic to q. That is, graph isomorphism query processing is to compute the answer set $A_q = \{G_i \in \mathcal{D} : G_i \cong q\}$.

GIQP can be applied by an adversary of social network anonymization to find a target user in the anonymized social network. The adversary identifies the target user by testing graph isomorphism between the d-neighborhood graph of the target user and the d-neighborhood graph of each vertex in the anonymized social network. In this situation, the number of data graphs increases as the size of the social network increases, and the size of a data graph increases as d grows. Since there is no polynomial time algorithm for graph isomorphism, there is no polynomial time algorithm for GIQP as well. Therefore, GIQP is very challenging when there are large numbers of data graphs and/or large-scale data graphs.

The canonical form [59] of a graph G is a unique representation of G such that any graph isomorphic to G has an identical canonical form as that of G. Canonical forms lead to a natural solution for GIQP: build an index that contains the canonical forms of the data graphs. Then, searching the index for the canonical form of the query graph gives the answer to the problem. This approach is suitable when there is a large number of data graphs, as the search can be quickly done once the index and the canonical form of the query graph are computed. However, computing the canonical form of a graph is computationally expensive, and sometimes it is impossible to build an index of large-scale graphs within a reasonable time.

Another approach is conducting a graph isomorphism algorithm multiple times (as many as the number of data graphs). On one hand, this approach is simple and easy to extend existing graph isomorphism algorithms [22, 11, 24] to handle GIQP. On the other hand, a lot of graph isomorphism tests may be conducted if there is a large number of data graphs.

Lastly, one may use subgraph/supergraph search [14, 20, 48, 64, 36, 35] algorithms to deal with GIQP. Given a set of data graphs and a query graph, subgraph search (resp. supergraph search) is the problem of finding all the data graphs that contain (resp. are contained in) the query graph as a subgraph. These problems are more general than GIQP, and thus subgraph/supergraph approach is slower than aforementioned approaches in general.

In this chapter we propose a fast graph isomorphism query processing algorithm, which consists of index construction stage and query processing stage. Our algorithm make use of the canonical coloring as follows.

1. We present a coloring method that outputs the *canonical coloring* of a vertex-labeled graph. If there exists an isomorphism between any two graphs, then every two vertices mapped in the isomorphism get the same

color in their canonical colorings. We define the *color-label distribution* to represent the canonical coloring, and we show that any two graphs are not isomorphic if their color-label distributions are different.

2. In the index construction stage, we construct a two-level index based on the degree sequences and the canonical colorings of the data graphs. The index helps us quickly filter out data graphs that are not isomorphic to the query graph during the query processing stage. The index can be built in $O(\sum_{G \in \mathcal{D}}(|V(G)| + |E(G)|) \log |V(G)| + |V(G^*)||\mathcal{D}| \log |\mathcal{D}|)$ time, where $\mathcal{D}$ is the set of data graphs and G^* is a data graph with the maximum number of vertices.

3. In the query processing stage, we retrieve data graphs from the index according to the degree sequence and the canonical coloring of the query graph. For each retrieved data graph, we verify whether it is isomorphic to the query graph. Specifically, we show that we can obtain a coarsest stable coloring of the disjoint union of two graphs from their canonical colorings, and thus we can skip the pairwise color refinement once the canonical colorings of a data graph and the query graph are computed. Utilizing the coarsest stable coloring, we perform backtracking to verify whether an isomorphism of the two graphs exists. We also present a method to determine whether a coloring of a graph is stable or not in linear time.

We conduct experiments comparing our algorithm with state-of-the-art algorithms on real-world datasets. We vary two types of parameters, i.e., the number of data graphs and the size of data graphs. Experimental results show that our algorithm consistently outperforms existing algorithms in terms of index construction time, and it runs faster than existing algorithms in terms of query processing time as graph sizes increase.

4.1 Canonical Coloring

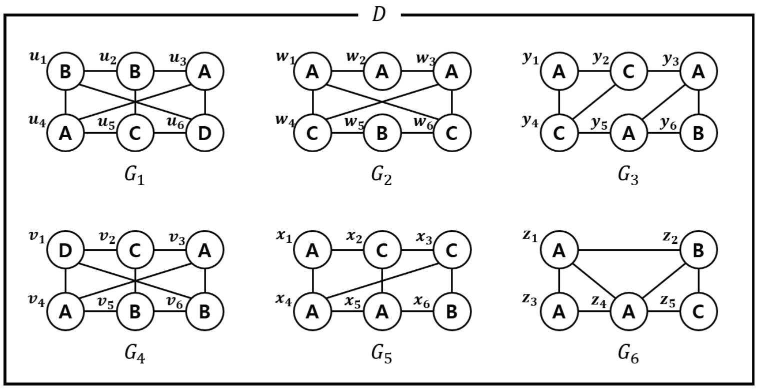
In this section we present an initial coloring method, and briefly describe the color refinement algorithm proposed by Berkholz, Bonsma, and Grohe [7]. Let π'_G be the coloring of a graph G obtained by applying the above color refinement algorithm on the initial coloring of G. We call π'_G the *canonical coloring* of G, and we will show that the canonical coloring is isomorphism-invariant. Then, we define the *color-label distribution* of a canonical coloring, and show that any two graphs are not isomorphic if the color-label distributions of their canonical colorings are different.

In what follows, a coloring algorithm or method outputs a coloring of an input (colored) graph.

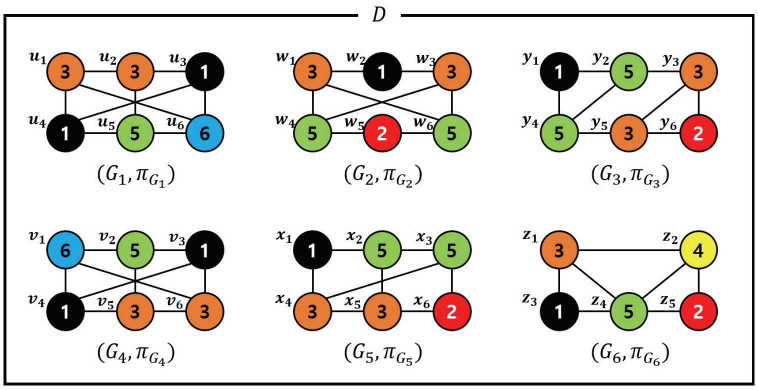
Definition 4.1.1. *We say that a coloring algorithm (or method) is isomorphism-invariant if, for any two graphs G and H with their output colorings π_G and π_H of the algorithm, respectively, the following statement holds: If there is an isomorphism $f : V(G) \to V(H)$, then $\pi_G(u) = \pi_H(f(u))$ for every $u \in V(G)$.*

For example, a coloring method that sets $\pi_G(u)$ to u's degree for each $u \in V(G)$ is isomorphism-invariant, because any two vertices mapped in an isomorphism have the same degree.

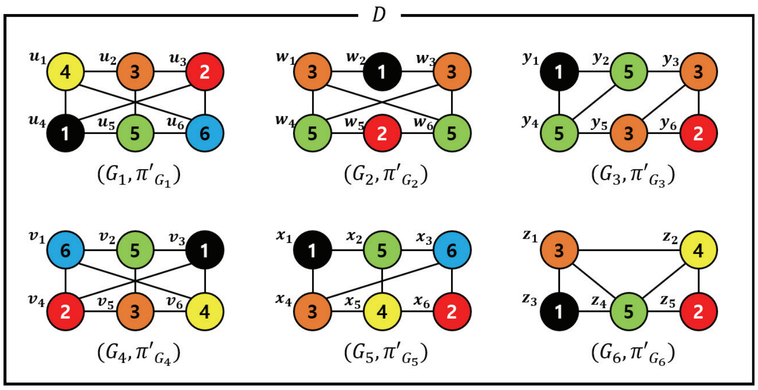
Initial Coloring Method. Given a graph G, our initial coloring method outputs a coloring π_G of G as follows. First compute an array S of vertices in $V(G)$ such that the vertices are sorted by the lexicographic order of (degree, label). Then, set the color of $S[i]$ from $i = 1$ to $|V(G)|$ as follows. Initially, $\pi_G(S[1])$ is set to 1. For each $S[i]$ with $i > 1$, if $S[i]$ has the same degree and label as those of $S[i-1]$, then $\pi_G(S[i])$ is set to $\pi_G(S[i-1])$. Otherwise, $\pi_G(S[i])$ is set to i. That is, $\pi_G(u)$ for each $u \in V(G)$ is set to the position of the leftmost vertex in S which has the same degree and label as those of u.



(a) Set of data graphs.



(b) Initial colorings



(c) Canonical colorings

Figure 4.1: Running example for graph isomorphism query processing.

Example 4.1.1. *Figure 4.1b shows the initial colorings of the data graphs in Figure 4.1a. Consider the initial coloring of* G_2 *as an example.* $S = (w_2, w_5, w_1, w_3, w_4, w_6)$ *according to the degrees and labels. We have* $\pi_{G_2}(w_6) = 5$ *because* $S[5] = w_4$ *is the leftmost vertex in* S *which has the same degree and label as those of* w_6*.*

Lemma 4.1.1. *The initial coloring method is isomorphism-invariant.*

Proof. Given two graphs G and H with their initial colorings π_G and π_H, respectively, suppose that there exists an isomorphism $f : V(G) \to V(H)$. Since G and H are isomorphic, for any pair of degree d and label l, the number of vertices in $V(G)$ whose degree is d and label is l is equal to the number of vertices in $V(H)$ whose degree is d and label is l. Since u and $f(u)$ for every $u \in V(G)$ have the same degree and label, $\pi_G(u)$ (the position of the leftmost vertex in S which has the same degree and label as those of u) is equal to $\pi_H(f(u))$. □

Color Refinement Algorithm. Given a colored graph (G, π_G) as an input, *color refinement* [50, 10, 56, 40, 7] is the problem of finding a coarsest stable coloring π'_G that is finer than π_G. A coloring π_G of G is *stable* if, for any color c and two vertices $u, v \in V(G)$ such that $\pi_G(u) = \pi_G(v)$, $|N_G(u) \cap \pi_G^{-1}(c)| = |N_G(v) \cap \pi_G^{-1}(c)|$. Let π_G, π'_G be two colorings of a graph G. We say that π'_G is *finer* than π_G (and π_G is *coarser* than π'_G) if every cell of π'_G is a subset of some cell of π_G. A coloring π'_G is a *coarsest* stable coloring finer than π_G if there is no stable coloring π''_G such that π''_G is finer than π_G and strictly coarser than π'_G. For example, in Figure 4.1c, each coloring π'_{G_i} for $1 \leq i \leq 6$ is a coarsest stable coloring finer than π_{G_i} in Figure 4.1b.

Berkholz, Bonsma, and Grohe [7] presented an $O((|V(G)| + |E(G)|) \log |V(G)|)$ time color refinement algorithm (BBG color refinement algorithm for short), which is briefly described as follows. The algorithm uses a stack containing colors. Initially, the stack contains all colors used in the input coloring in increasing order. For each color c in the stack, check if there exist $x, y \in V(G)$ such that $\pi_G(x) = \pi_G(y)$ and $|N_G(x) \cap \pi_G^{-1}(c)| \neq |N_G(y) \cap \pi_G^{-1}(c)|$. In the affirmative case, π_G is not stable, and thus recolor the vertices in $\pi_G^{-1}(\pi_G(x))$. When there are two or more such cells, the cell corresponding to the smallest color is recolored first. Suppose that the vertices in $\pi_G^{-1}(\pi_G(x))$ are divided into n groups according to the number of neighbors in $\pi_G^{-1}(c)$. The group with the smallest number of neighbors in $\pi_G^{-1}(c)$ maintains the original color. Each of the rest $n - 1$ groups introduces a new color, where the new color is the smallest color among the unused colors. The group with the smaller number of neighbors in $\pi_G^{-1}(c)$ gets the smaller new color. After recoloring the vertices in $\pi_G^{-1}(\pi_G(x))$, the colors involved in the recoloring (except one color) are pushed into the stack in increasing order. The whole process is repeated until the stack is empty. We refer the reader to [7] for the details of the algorithm.

Lemma 4.1.2 ([7])**.** *Let π'_G and π'_H be the outputs of the BBG color refinement algorithm on (G, π_G) and (H, π_H), respectively, where $G \cong H$. For any isomorphism $f : V(G) \to V(H)$, if $\pi_G(u) = \pi_H(f(u))$ for every $u \in V(G)$, then $\pi'_G(u) = \pi'_H(f(u))$ for every $u \in V(G)$.*

Canonical Coloring and Color-Label Distribution. Given a graph G, let π'_G be the output of the BBG color refinement algorithm on (G, π_G), where π_G is the initial coloring of G computed by our initial coloring method. We call π'_G the *canonical coloring* of G.

Lemma 4.1.3. *The coloring method that applies the initial coloring method and then the BBG color refinement algorithm is isomorphism-invariant.*

Proof. Given two graphs G and H, let π_G and π_H be the initial colorings of G and H, respectively, and let π'_G and π'_H be the outputs of the BBG color refinement algorithm on (G, π_G) and (H, π_H), respectively. If there exists an isomorphism $f : V(G) \to V(H)$, then $\pi_G(u) = \pi_H(f(u))$ for every $u \in V(G)$ (because the initial coloring method is isomorphism-invariant), and thus $\pi'_G(u) = \pi'_H(f(u))$ for every $u \in V(G)$ by Lemma 4.1.2. Therefore, the coloring method of the lemma is isomorphism-invariant. $\square$

Example 4.1.2. *Figure 4.1c shows the canonical colorings of the data graphs in Figure 4.1a. For an isomorphism* $f = \{(u_1, v_6), (u_2, v_5), (u_3, v_4), (u_4, v_3), (u_5, v_2), (u_6, v_1)\}$ *of* G_1 *and* G_4*, we can see that* $\pi'_{G_1}(u) = \pi'_{G_4}(f(u))$ *for every* $u \in V(G_1)$*.*

For a graph G and its canonical coloring π'_G, the *color-label distribution* of (G, π'_G) is denoted by $CLD(G, \pi'_G)$, which is the sequence of pairs $(\pi'_G(u), L_G(u))$ for $u \in V(G)$ such that the pairs are lexicographically ordered. For example, the color-label distribution of (G_2, π'_{G_2}) in Figure 4.1c is $CLD(G_2, \pi'_{G_2}) = ((1, A), (2, B), (3, A), (3, A), (5, C), (5, C))$.

Lemma 4.1.4. *Suppose that we are given two graphs* G *and* H *with their canonical colorings* π'_G *and* π'_H*, respectively. If there exists an isomorphism* $f : V(G) \to V(H)$*, then* $CLD(G, \pi'_G) = CLD(H, \pi'_H)$*.*

Proof. We have $\pi'_G(u) = \pi'_H(f(u))$ for every $u \in V(G)$ because the canonical coloring method is isomorphism-invariant. Since f is an isomorphism, we have $L_G(u) = L_H(f(u))$ for every $u \in V(G)$. Thus, we get $(\pi'_G(u), L_G(u)) = (\pi'_H(f(u)), L_H(f(u)))$ for every $u \in V(G)$. It follows that the sorted sequence of pairs $(\pi'_G(u), L_G(u))$ for $u \in V(G)$ is equal to the sorted sequence of pairs $(\pi'_H(v), L_H(v))$ for $v \in V(H)$. Therefore, $CLD(G, \pi'_G) = CLD(H, \pi'_H)$. $\square$

degree sequence	pointer
(5, 4, 3, 3, 2, 2)	1
(6, 3, 3, 3, 3, 2, 2)	2
(6, 3, 3, 3, 3, 3, 3)	5
∅	7

first level

	color-label distribution	G_i	coloring
1	((1,A), (2,C), (3,A), (4,B), (5,A))	6	π'_{G_6}
2	((1,A), (2,B), (3,A), (3,A), (5,C), (5,C))	2	π'_{G_2}
3	((1,A), (2,B), (3,A), (3,A), (5,C), (5,C))	3	π'_{G_3}
4	((1,A), (2,B), (3,A), (4,A), (5,C), (6,C))	5	π'_{G_5}
5	((1,A), (2,A), (3,B), (4,B), (5,C), (6,D))	1	π'_{G_1}
6	((1,A), (2,A), (3,B), (4,B), (5,C), (6,D))	4	π'_{G_4}

second level

Figure 4.2: Index $\mathcal{I}$ of the data graphs.

Corollary 4.1.1. *Suppose that we are given two graphs G and H with their canonical colorings π'_G and π'_H, respectively. If $CLD(G, \pi'_G) \neq CLD(H, \pi'_H)$, then G and H are not isomorphic.*

4.2 Index Construction

In this section we define an index for the data graphs and present an algorithm to compute it.

Index. We define the index $\mathcal{I}$ for a set $\mathcal{D} = \{G_1, G_2, \ldots, G_k\}$ of data graphs, which consists of two levels as follows.

- The second level is an array of triples $(CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i})$ for $1 \leq i \leq k$, where π'_{G_i} is the canonical coloring of G_i. The triples are sorted in the order as follows. Let $t_i = (CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i})$ and $t_j = (CLD(G_j, \pi'_{G_j}), j, \pi'_{G_j})$ be two triples. t_i precedes t_j if (1) ds_{G_i} is lexicographically smaller than ds_{G_j}; or (2) $ds_{G_i} = ds_{G_j}$ and $CLD(G_i, \pi'_{G_i})$ is lexicographically smaller than $CLD(G_j, \pi'_{G_j})$.

- The first level is an array of pairs (ds, p), where ds is a unique degree sequence within $\mathcal{D}$, p is an integer that indicates the position of the first triple $(CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i})$ in the second level such that $ds_{G_i} = ds$, and the pairs are sorted by the lexicographic order of the degree sequence. All distinct degree sequences of the data graphs in $\mathcal{D}$ are stored with associated positions, and a sentinel $(\emptyset, |\mathcal{D}| + 1)$ is stored at the end of the first level.

Since the set of data graphs is static, we implement each level of the index by a sorted array. In case that the set is dynamic, each level should be implemented by a balanced search tree such as the red-black tree.

Example 4.2.1. *Figure 4.2 illustrates the index of the data graphs in Figure 4.1a. See Figure 4.1c for the canonical colorings of the data graphs. In the first level,* $((6,3,3,3,3,3,3),5)$ *means that* G_1 *(which is stored in the 5th position of the second level) is the first graph stored in the second level whose degree sequence is* $(6,3,3,3,3,3,3)$.

Index Construction Algorithm. Algorithm 4 describes our index construction algorithm, which constructs an index $\mathcal{I}$ for $\mathcal{D}$. We compute the second level and then the first level as follows. For each data graph $G_i \in \mathcal{D}$, compute the degree sequence ds_{G_i} and the canonical coloring π'_{G_i} of G_i, and make a 4-tuple $(ds_{G_i}, CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i})$. Sort the 4-tuples for all the data graphs by the lexicographic order of $(ds_{G_i}, CLD(G_i, \pi'_{G_i}))$. Scan the sorted 4-tuples, and store the triples $(CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i})$ into the second level in the sorted order. Find all distinct degree sequences with their associated positions by scanning the sorted 4-tuples, and add them to the first level. Finally, add a sentinel $(\emptyset, k+1)$ to the end of the first level to denote the last position of the second level.

Lemma 4.2.1. *Given a set* $\mathcal{D}$ *of data graphs, the time complexity of the index construction of* $\mathcal{D}$ *is* $O(\sum_{G \in \mathcal{D}}(|V(G)| + |E(G)|) \log |V(G)| + |V(G^*)||\mathcal{D}| \log |\mathcal{D}|)$, *where* G^* *is a data graph with the maximum number of vertices.*

Algorithm 4: BuildIndex

```
Input  : a set of data graphs $\mathcal{D} = \{G_1, G_2, \ldots, G_k\}$
Output: an index $\mathcal{I}$
1  $\mathcal{I}.\mathit{first\text{-}level} \leftarrow$ an empty array;
2  $\mathcal{I}.\mathit{second\text{-}level} \leftarrow$ an empty array;
3  $Q \leftarrow$ an empty array of 4-tuples $(\mathit{ds}, \mathit{CLD}, i, \pi')$;
4  for $i$ from 1 to $k$ do
5      $\mathit{ds}_{G_i} \leftarrow$ degree sequence of $G_i$;
6      $\pi'_{G_i} \leftarrow \mathit{CanonicalColoring}(G_i)$;
7      $Q.\mathit{Append}(\mathit{ds}_{G_i}, \mathit{CLD}(G_i, \pi'_{G_i}), i, \pi'_{G_i})$;
8  Sort $Q$ by $(\mathit{ds}_{G_i}, \mathit{CLD}(G_i, \pi'_{G_i}))$;
9  $\mathcal{I}.\mathit{second\text{-}level.Append}(Q[1].\mathit{CLD}, Q[1].i, Q[1].\pi')$;
10 $\mathcal{I}.\mathit{first\text{-}level.Append}(Q[1].\mathit{ds}, 1)$;
11 $\mathit{ds} \leftarrow Q[1].\mathit{ds}$;
12 for $p$ from 2 to $k$ do
13     $\mathcal{I}.\mathit{second\text{-}level.Append}(Q[p].\mathit{CLD}, Q[p].i, Q[p].\pi')$;
14     if $Q[p].\mathit{ds} \neq \mathit{ds}$ then
15         $\mathcal{I}.\mathit{first\text{-}level.Append}(Q[p].\mathit{ds}, p)$;
16         $\mathit{ds} \leftarrow Q[p].\mathit{ds}$;
17 $\mathcal{I}.\mathit{first\text{-}level.Append}(\emptyset, k+1)$;
18 return $\mathcal{I}$;
```

Proof. The running time of the index construction algorithm is dominated by the time to compute the canonical colorings and the time to sort the 4-tuples for the data graphs. It takes $O(\sum_{G\in\mathcal{D}}(|V(G)| + |E(G)|) \log |V(G)|)$ time to compute the canonical colorings of the data graphs by the BBG color refinement algorithm. Sorting the 4-tuples takes $O(|V(G^*)||\mathcal{D}| \log |\mathcal{D}|)$ time because there are $|\mathcal{D}|$ 4-tuples and the comparison of degree sequences and color-label distributions in two 4-tuples can be done in $O(|V(G^*)|)$ time. □

4.3 Query Processing

In this section we present the query processing algorithm and give a method to determine whether a coloring of a graph is stable or not in linear time.

Query Processing Algorithm. Algorithm 5 describes our query processing algorithm, which takes a query graph q and an index $\mathcal{I}$ of $\mathcal{D}$, and finds an answer set A_q for q. We first compute the degree sequence ds_q of the query graph, and check whether it is in the first level of the index. If not, we can easily conclude that $A_q = \emptyset$. Otherwise, we compute the canonical coloring π'_q of q, and retrieve a list C of triples $(CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i})$ such that $CLD(G_i, \pi'_{G_i}) = CLD(q, \pi'_q)$. Other data graphs are not isomorphic to q by Corollary 4.1.1. The retrieval is done as follows:

1. Find the pair (ds_q, p) from the first level of $\mathcal{I}$ by the binary search. Let (ds, p') be the pair right after (ds_q, p).
2. Retrieve the triples from the second level by the binary search between p and p'.

The rest of the query processing is done based on the following lemmas. Suppose that we are given two colored graphs (G, π'_G) and (H, π'_H) such that

Algorithm 5: ProcessQuery

```
Input  : a query graph $q$ and an index $\mathcal{I}$
Output: $A_q$
1  $A_q \leftarrow \emptyset$;
2  $ds_q \leftarrow$ degree sequence of $q$;
3  if $ds_q$ is not in $\mathcal{I}$.first-level then
4      return $A_q$;
5  $\pi'_q \leftarrow CanonicalColoring(q)$;
6  $C \leftarrow \mathcal{I}.Retrieve(ds_q, CLD(q, \pi'_q))$;
7  for each triple $(CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i}) \in C$ do
8      if $CheckStableColoring(\pi'_q \cup \pi'_{G_i}, q \cup G_i)$ then
9          if $Isomorphic(q, G_i, \pi'_q \cup \pi'_{G_i})$ = "YES" then
10             Insert $G_i$ to $A_q$;
11 return $A_q$;
```

(1) π'_G and π'_H are the canonical colorings of G and H, respectively, and (2) $CLD(G, \pi'_G) = CLD(H, \pi'_H)$.

Lemma 4.3.1. *If $\pi'_G \cup \pi'_H$ is not stable on $G \cup H$, then G and H are not isomorphic.*

Proof. We prove by contradiction. Assume that there is an isomorphism $f : V(G) \to V(H)$. Since $\pi'_G \cup \pi'_H$ is not stable, there are two vertices $u \in V(G)$, $v \in V(H)$ and a color c such that $\pi'_G(u) = \pi'_H(v)$ and $|N_G(u) \cap \pi'^{-1}_G(c)| \neq |N_H(v) \cap \pi'^{-1}_H(c)|$. Since π'_G and π'_H are canonical colorings, $\pi'_G(w) = \pi'_H(f(w))$ for every $w \in V(G)$. Therefore, we have $|N_G(u) \cap \pi'^{-1}_G(c)| = |N_H(f(u)) \cap \pi'^{-1}_H(c)|$. Since $\pi'_G(u) = \pi'_H(f(u)) = \pi'_H(v)$ and π'_H is stable on H, we get

$|N_H(f(u)) \cap \pi_H'^{-1}(c)| = |N_H(v) \cap \pi_H'^{-1}(c)|$, and thus we get $|N_G(u) \cap \pi_G'^{-1}(c)| = |N_H(v) \cap \pi_H'^{-1}(c)|$ which is a contradiction. $\square$

Lemma 4.3.2. *If $\pi'_G \cup \pi'_H$ is stable on $G \cup H$, then $\pi'_G \cup \pi'_H$ is a coarsest stable coloring of $G \cup H$.*

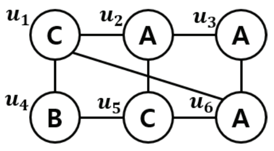
Proof. We prove by contradiction. Assume that there exists a stable coloring π'' of $G \cup H$ that is strictly coarser than $\pi'_G \cup \pi'_H$. Given a coloring π_Q of a graph Q and a vertex set $S \subseteq V(Q)$, let $\pi_Q|_S$ denote the restriction of π_Q to S such that $\pi_Q|_S(u) = \pi_Q(u)$ for each $u \in S$. Given $\pi''_G = \pi''|_{V(G)}$ and $\pi''_H = \pi''|_{V(H)}$, we will show that π''_G and π''_H are stable and strictly coarser than π'_G and π'_H, respectively, which contradicts that π'_G and π'_H are coarsest stable colorings.

Since π'' is stable on $G \cup H$, and G and H are disconnected in $G \cup H$, π''_G and π''_H are stable colorings of G and H, respectively.

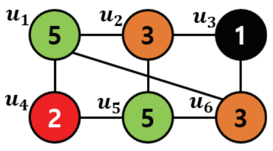
Now we show that π''_G is strictly coarser than π'_G (the proof for π'_H is symmetric). Since π'' is strictly coarser than $\pi'_G \cup \pi'_H$, each cell of π'_G is a subset of some cell of π''_G. Also, there exist two distinct colors c_1 and c_2 in $\pi'_G \cup \pi'_H$ such that $(\pi'_G \cup \pi'_H)^{-1}(c_1)$ and $(\pi'_G \cup \pi'_H)^{-1}(c_2)$ are subsets of a cell of π''. It follows that there exists a cell of π''_G that contains $\pi_G'^{-1}(c_1)$ and $\pi_G'^{-1}(c_2)$, where $\pi_G'^{-1}(c_1) \neq \emptyset$ and $\pi_G'^{-1}(c_2) \neq \emptyset$ because $CLD(G, \pi'_G) = CLD(H, \pi'_H)$.

Therefore, π''_G and π''_H are stable and strictly coarser than π'_G and π'_H, respectively, which is a contradiction. $\square$

For each triple $(CLD(G_i, \pi'_{G_i}), i, \pi'_{G_i}) \in C$, we check if $\pi'_q \cup \pi'_{G_i}$ is stable on $q \cup G_i$. If the coloring is not stable, then G_i is not isomorphic to q by Lemma 4.3.1. If the coloring is stable, we verify whether q and G_i are isomorphic based on the backtracking approach. In this case, $\pi'_q \cup \pi'_{G_i}$ is a coarsest stable coloring of $q \cup G_i$ by Lemma 4.3.2, which is equivalent to the output of the pairwise color refinement. Thus, utilizing the coarsest stable coloring, we perform backtracking



(a) q



(b) π'_q

Figure 4.3: Query graph q and its canonical coloring π'_q.

to find an isomorphism of q and G_i just like `CRaB` does in Chapter 3. (Note that we do not explicitly conduct the pairwise color refinement.) If an isomorphism of q and G_i is found during backtracking, then insert G_i to A_q.

Example 4.3.1. *Figure 4.3b shows the canonical coloring π'_q of the query graph q in Figure 4.3a. The degree sequence of q is $(6, 3, 3, 3, 3, 2, 2)$ and the color-label distribution of (q, π'_q) is $\{(1, A), (2, B), (3, A), (3, A), (5, C), (5, C)\}$. Consider the index in Figure 4.2. (G_2, π'_{G_2}) and (G_3, π'_{G_3}) have identical degree sequences and color-label distributions as those of (q, π'_q). We check whether $\pi'_q \cup \pi'_{G_2}$ and $\pi'_q \cup \pi'_{G_3}$ are stable on $q \cup G_2$ and $q \cup G_3$, respectively. Only $\pi'_q \cup \pi'_{G_2}$ is stable, and thus we verify whether q and G_2 is isomorphic by performing backtracking. G_2 is indeed isomorphic to q, and the answer set is $A_q = \{G_2\}$.*

Check Stable Coloring. Given a coloring π_G of G, we present a method to check whether π_G is stable on G. For each color c of π_G, check whether there exist two vertices u, v such that $\pi_G(u) = \pi_G(v)$ and $|N_G(u) \cap \pi_G^{-1}(c)| \neq |N_G(v) \cap \pi_G^{-1}(c)|$. The coloring is stable if and only if there is no such vertices. We use one array CNT of length $|V(G)|$ to check the above condition as follows. Initially $CNT[u] = 0$ for every $u \in V(G)$. Given a color c, we set $CNT[u] = CNT[u] + |N_G(u) \cap \pi_G^{-1}(c)|$ by scanning the neighbors of the vertices in $\pi_G^{-1}(c)$. Then, we check if there exist two vertices u, v among the visited neighbors such that $\pi_G(u) = \pi_G(v)$ and $CNT[u] \neq CNT[v]$. If such vertices exist, the coloring is not stable, and we immediately finish the algorithm. Otherwise, we move on to the next color in π_G. Note that by adding $|N_G(u) \cap \pi_G^{-1}(c)|$ to $CNT[u]$, we can avoid resetting $CNT[u] = 0$ for each time a color c is considered. This algorithm runs in $O(|V(G)| + |E(G)|)$ time because it scans adjacency list of G at most once to compute CNT.

4.4 Performance Evaluation

We conduct experiments to evaluate the performance of the proposed algorithm against the state-of-the-art algorithms from the existing approaches mentioned in Introduction. The following algorithms are evaluated.

- `Traces` [51]: graph canonization algorithm.
- `CRaB` [22]: graph isomorphism algorithm.
- `IDAR` [36]: supergraph search algorithm.
- `Ours`: our algorithm.

The index of `Ours` consists of two levels, where the first level filters data graphs using the degree sequence and the second level filters data graphs using the color-label distribution. Since the first level is simple to implement, we make a variant of `Traces` by adding the first level of our index to the index of `Traces`, and include it (which will be called `DS-Traces`) in our evaluation. We couldn't apply the first level to `CRaB` and `IDAR` because `CRaB` does not construct an index and `IDAR` constructs an index called the *IDAG*, which is an integrated DAG of the DAGs of the data graphs.

All the source codes were obtained from the authors of previous papers, where we use version 2.6r12 of `Traces`. All algorithms are implemented in

Table 4.1: Characteristics of datasets.

Dataset (G)	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	$\lvert\Sigma\rvert$	avg-deg(G)
Human	4,271	84,890	44	39.75
HPRD	9,045	34,853	307	7.71
HepTh	8,638	24,806	-	5.74
CondMat	21,363	91,286	-	8.55
HepPh	11,204	117,619	-	21.00
Plus	283,872	428,384	-	3.02
Amazon	334,863	925,872	-	5.53
DBLP	317,080	1,049,866	-	6.62
Bigblue	3,795,055	8,712,138	-	4.59
LiveJournal	3,997,962	34,681,189	-	17.35

C++. Experiments are conducted on a machine with two Intel Xeon E5-2680 v3 2.50GHz CPUs and 256GB memory running CentOS Linux.

Datasets. We use 10 real-world graphs in Table 4.1, which are benchmark datasets used in [51, 22, 35]. Human and HPRD are protein-protein interaction (PPI) networks, where vertices represent proteins and edges represent interactions between proteins. Vertices in the PPI networks are labeled by *Gene Ontology* information [26]. The rest of the graphs are unlabeled graphs. HepTh, CondMat, HepPh, and DBLP are collaboration networks, where vertices represent scientists and edges represent collaborations (co-authoring a paper). Plus and Amazon are router interconnection network and product co-purchasing network, respectively. Bigblue is a graph derived from a circuit, where vertices represent electronic components and nodes in the circuit, and edges represent connections between components and nodes. LiveJournal is an online social network, where edges represent interactions between users.

We generate a set $\mathcal{D} = \{G_1, G_2, \ldots, G_k\}$ of data graphs for a dataset in the following way: randomly choose k distinct vertices in the dataset and extract

Table 4.2: Average number of vertices and edges of a data graph for each value of d, where d is the number of hops.

	$d = 1$		$d = 2$		$d = 3$	
Dataset (G)	avg. $\lvert V\rvert$	avg. $\lvert E\rvert$	avg. $\lvert V\rvert$	avg. $\lvert E\rvert$	avg. $\lvert V\rvert$	avg. $\lvert E\rvert$
Human	40.9	1,555.3	367.0	11,139.8	1,097.6	29,268.0
HPRD	8.9	14.8	189.8	735.9	1,798.2	9,346.2
HepTh	6.8	15.5	47.3	128.6	297.2	971.5
CondMat	9.3	31.6	111.6	526.4	1,008.9	5,616.5
HepPh	21.9	893.1	282.2	10,790.4	1,733.1	42,332.4
Plus	4.0	6.5	60.1	165.5	394.6	1,715.9
Amazon	6.4	11.3	42.4	95.6	156.8	360.2
DBLP	7.6	32.8	87.1	360.2	1,139.3	5,311.3
Bigblue	5.6	4.6	115.7	124.8	890.5	1,071.6
LiveJournal	17.3	123.6	1,214.9	17,265.8	40,278.8	632,495.0

the d-neighborhood graph of each chosen vertex. To see how the size and the number of data graphs affects the performance of GIQP, we varied two types of parameters:

- number of data graphs: $|\mathcal{D}| = 1000, 2000, 4000, 8000$,
- number of hops: $d = 1, 2, 3$,

where the neighborhood size increases exponentially as d grows [80]. The default parameter settings are $|\mathcal{D}| = 4000$ and $d = 2$. Table 4.2 shows the average number of vertices and edges of a data graph for different d values. For each value of d, we generate a query set that contains 200 query graphs, where a query graph is generated in the same way as the data graph.

Metrics. Since there is no polynomial time algorithm for GIQP, an algorithm may not construct an index or process a query set within a reasonable time. Thus, we set a time limit of 1 hour for constructing an index or processing a query set. If an algorithm does not construct an index or process a query set within the time limit, we regard the construction time or processing time as 1 hour. To evaluate an algorithm with respect to a set of data graphs and a query set, we measure the following metrics:

- Index construction time: we measure the total running time in milliseconds to build an index of the data graphs.
- False positive ratio $FP_q = \frac{|C_q|-|A_q|}{|C_q|}$ for query graph q: we measure the average false positive ratio for a query graph, where C_q is the set of data graphs retrieved by an index that are required to be verified whether they are isomorphic to q, and A_q is the set of answer graphs for q (i.e., data graphs that are isomorphic to q). We evaluate three versions of our index: (1) one that uses only the first level (degree sequence); (2) one that uses only the second level (color-label distribution); and (3) one that uses both of the two levels (combination).
- Number of items: we measure the number of items in our index.
- Query processing time: we measure the average running time in milliseconds to process a query graph.
- Breakdown of query processing time: we break up the query processing time into smaller parts, and measure the proportion of each part. The query processing time of `Ours` breaks up into the time used in the first level, the second level, and the verification. The query processing time of `DS-Traces` breaks up into the time used in the first level, graph canonization, and the search for the canonical form, and that of `Traces` breaks up into the time used in graph canonization and the search.

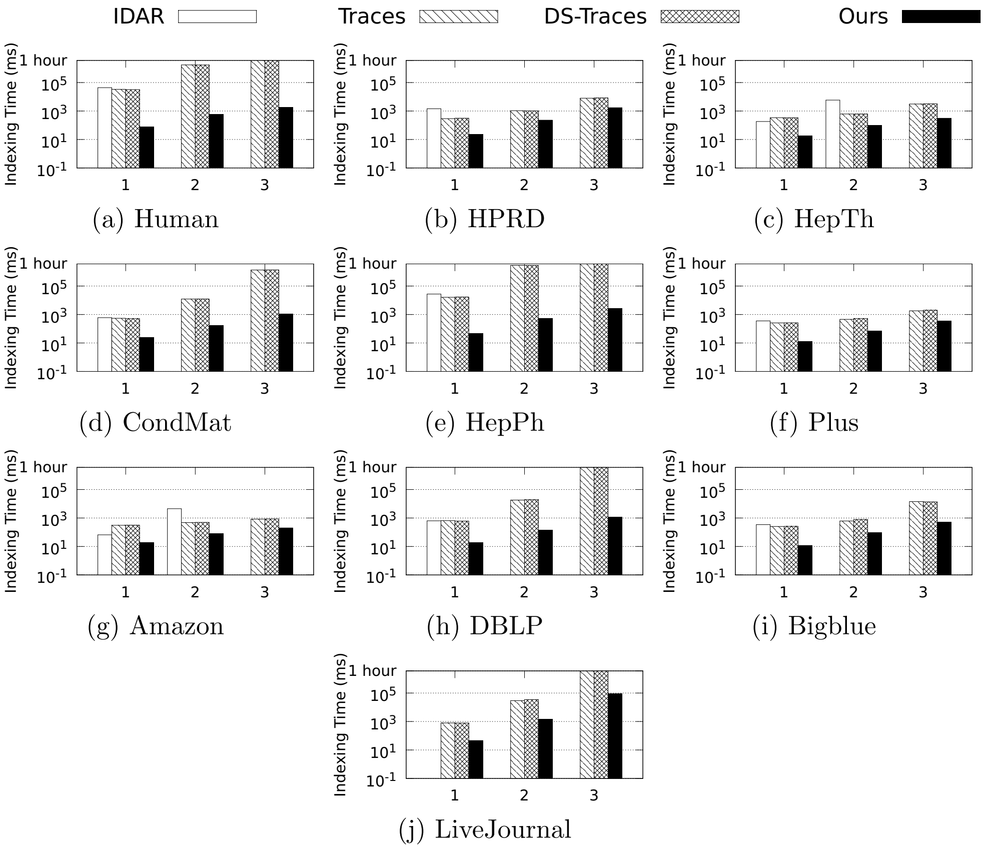


Figure 4.4: Index construction time for varying number of hops.

4.4.1 Varying Number of Hops.

First, we vary the number of hops, i.e., we set $d = 1, 2, 3$ with $|\mathcal{D}| = 4000$.

Index Construction Time. Figure 4.4 shows the index construction time of the algorithms. Since `CRaB` does not build an index, the result of `CRaB` is omitted. On some datasets, we were not able to run `IDAR` due to memory errors. Missing bars of `IDAR` indicates such failures.

In Figure 4.4, `Ours` is the fastest algorithm for all the datasets. The index construction time of `Ours` is bounded by polynomial time, whereas that

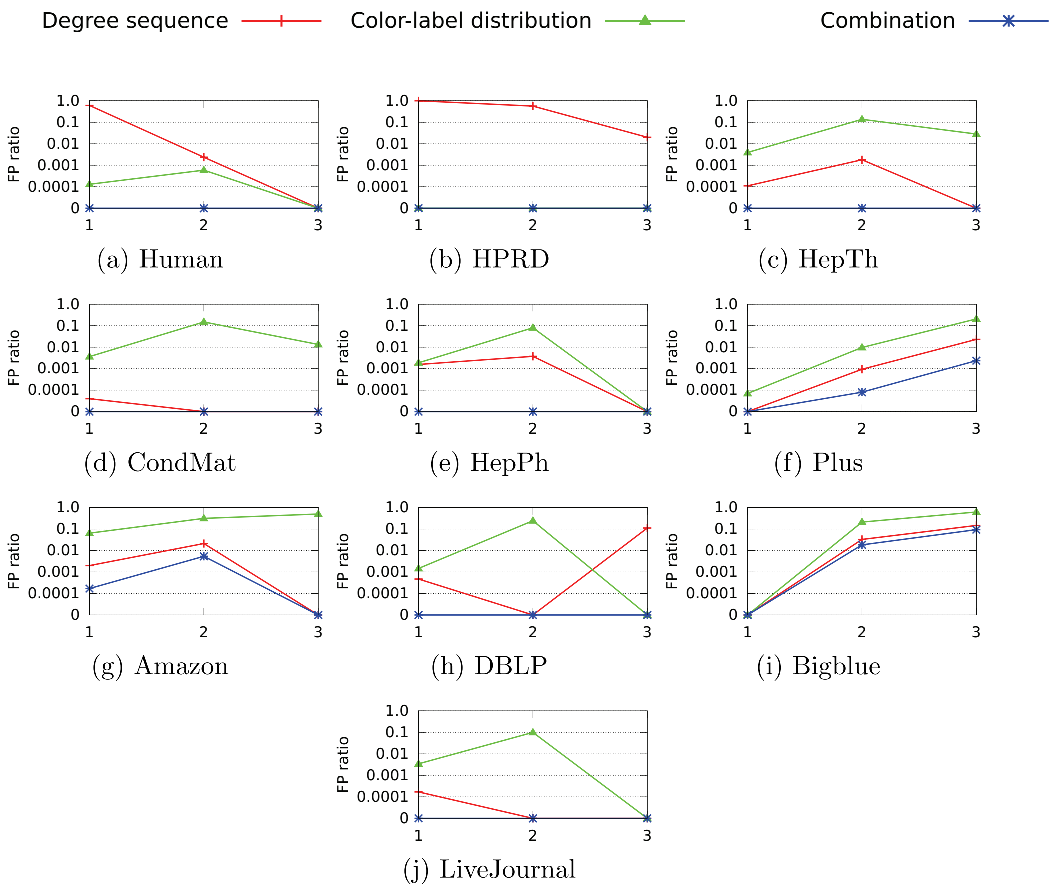


Figure 4.5: False positive ratio for varying number of hops.

of Traces is not. Thus, Ours runs orders of magnitude faster than Traces. Specifically, Ours runs up to 4 orders of magnitude faster than Traces (HepPh with $d = 2$ in Figure 4.4e), and overall 580 times faster on average. The index construction time of Traces exceeds the time limit on 4 datasets when $d = 3$. Although DS-Traces constructs an additional part of the index based on the degree sequence, DS-Traces is only marginally slower than Traces because degree sequences can be computed much faster than canonical forms. IDAR fails to build an index for most of the datasets, and its index construction time is

Table 4.3: Number of items in the first level of our index ($|\mathcal{D}| = 4000$).

Dataset	$d = 1$	$d = 2$	$d = 3$
Human	1130	1325	1235
HPRD	923	3403	3680
HepTh	1004	3149	3412
CondMat	1223	3502	3650
HepPh	1364	3194	3275
Plus	245	1909	3373
Amazon	875	3807	3963
DBLP	877	3631	3937
Bigblue	56	2555	3895
LiveJournal	1863	3870	3992

comparable to that of `Traces`.

False Positive Ratio. Figure 4.5 shows the false positive ratio of the filtering techniques used in our index. On Human and HPRD, which are labeled graphs, the false positive ratio of color-label distributions is smaller than that of degree sequences due to vertex labels. In unlabeled graphs, however, the false positive ratio of degree sequences is smaller. When we apply both degree sequences and color-label distributions in the filtering (i.e., combination), false positive ratios in most cases are zero or close to zero, which shows the effectiveness of our index. Average false positive ratios for degree sequences, color-label distributions, and the combination are 0.0846, 0.0895, and 0.0040, respectively.

Number of Items. In our two-level index, the number of items in the first level is equal to the number of distinct degree sequences of the data graphs, and the number of items in the second level is equal to the number of data graphs. Note that the number of data graphs is 4000 in this subsection. Table 4.3 shows the number of items in the first level of our index.

On Bigblue and Plus with $d = 1$, the numbers of items in the first level are relatively small. On these datasets, data graphs have a small number of vertices as shown in Table 4.1, which leads to a small number of distinct degree sequences. In general, if a dataset has a low average degree, small-scale data graphs are generated because data graphs are d-neighborhood graphs. On LiveJournal and HepPh with $d = 1$, the numbers of items in the first level is relatively large, because data graphs have a large number of vertices due to high average degrees.

When d grows, the average number of vertices of a data graph increases as shown in Table 4.1, and thus the number of distinct degree sequences increases in all datasets except Human. Human has a high average degree and a small number of vertices, so two vertices are likely to have an identical d-neighborhood. Therefore, many data graphs are isomorphic to each other on Human, and the number of distinct degree sequences is steady even if graph sizes increase.

Query Processing Time. Figure 4.6 shows the query processing time of the algorithms. For `IDAR`, `Traces`, and `DS-Traces`, if an index is not constructed, the query processing is not possible. Missing bars indicate such cases.

In Figure 4.6, the performances of the algorithms vary depending on d. When $d = 3$, `Ours` is the best for 7 datasets and `CRaB` is the best for 3 datasets. When $d = 2$, `Ours` is the best for 7 datasets and `DS-Traces` is the best for 3 datasets. When $d = 1$, `DS-Traces` is the best for 9 datasets and `Ours` is the best for 1 dataset. This results show that `Ours` is more scalable than other algorithms in terms of d.

The varying performances can be explained by the average size of a data graph and the average size of an answer set. When $d = 2$ or $d = 3$, the average size of a data graph is large as shown in Table 4.2, and the average size of an answer set for a query graph over the 10 datasets is small (11.4 for $d = 2$

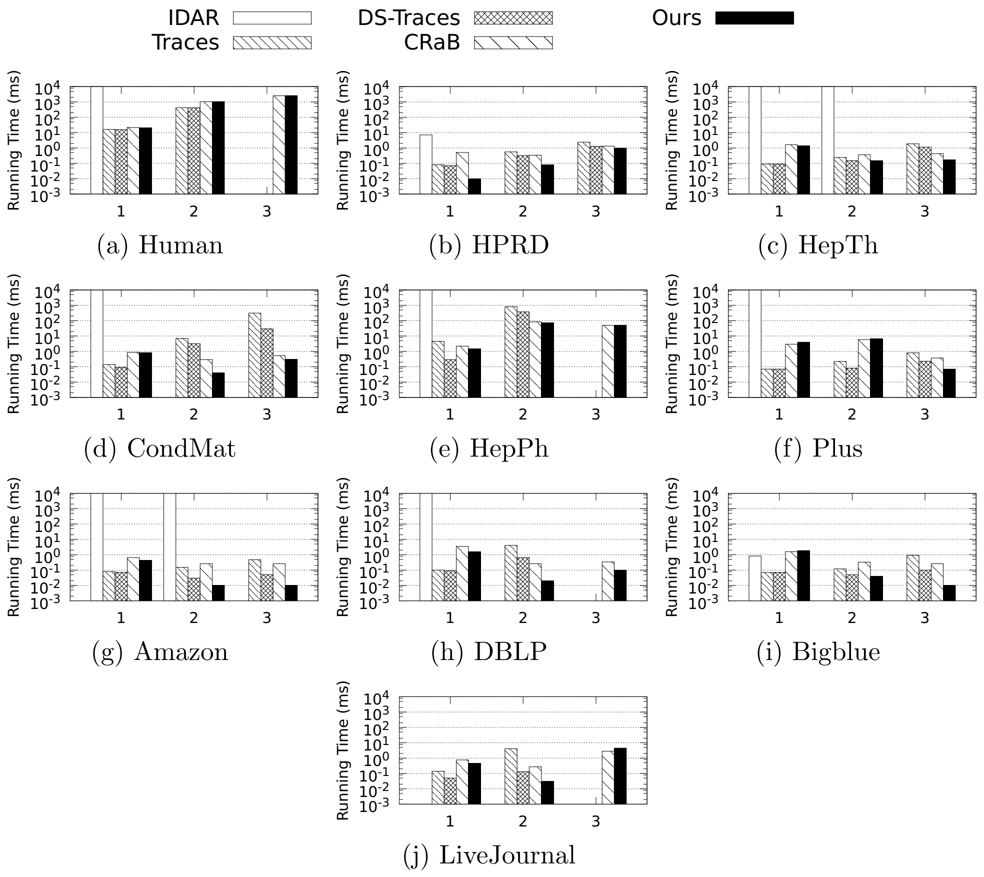


Figure 4.6: Query processing time for varying number of hops.

and 7.2 for $d = 3$). In this case, `Traces` takes a lot of time to compute the canonical form of a (big) query graph, whereas `Ours` verifies only a few data graphs since the false positive ratio of `Ours` is close to 0 as shown in Figure 4.5. Therefore `Ours` runs faster than `Traces` when $d = 2$ and $d = 3$. When $d = 1$, the average size of a data graph is very small, and the average size of an answer set for a query graph is 307.9. In this case, `Traces` (and `DS-Traces`) has an advantage because (1) computing the canonical form of a small query graph takes reasonably small time and (2) once the canonical form of a query graph is computed, the answer set can be found by the binary search. In addition, `Ours` runs the best even when $d = 1$ on HPRD because the average size of an answer set is very small (i.e., 0.9) due to the plenty of labels.

Among the 2×10 cases when $d = 2$ and $d = 3$, `Ours` outperforms `CRaB` for 15 cases, and `Ours` runs 10 times faster than `CRaB` on average. For the remaining 5 cases, `CRaB` runs marginally faster (about 1.1 times) than `Ours`. The performance gain of `Ours` comes from (1) the low false positive ratio, which leads to fewer executions of backtracking, and (2) the way we compute the output of the pairwise color refinement. Suppose that we have k data graphs to verify whether they are isomorphic to the query graph. In `CRaB`, k explicit pairwise color refinements are required, while `Ours` performs one canonical color refinement for the query graph and k checks of stable coloring. Among the 16 cases excluding the 4 cases that `DS-Traces` and `Traces` could not build indexes, `Ours` outperforms both `DS-Traces` and `Traces` for 14 cases, and it runs 18 times and 130 times faster than `DS-Traces` and `Traces`, respectively.

Among the 10 cases when $d = 1$, `DS-Traces` and `Traces` outperforms `Ours` for 9 cases and 8 cases, respectively, due to the small-scale graphs and a large number of answer graphs. `DS-Traces` always runs faster than or similar to `Traces` in these 10 cases, and it runs about 2.7 times faster than `Traces` on average.

Breakdown of Query Processing Time. Table 4.4 shows the proportion of each part of the query processing time of `Ours`, `DS-Traces`, and `Traces`. The verification time of `Ours` and the graph canonization time of `DS-Traces` and `Traces` take exponential time in the worst case, while other parts take polynomial time. When $d = 1$, `Ours` uses most of the time in verification (except HPRD) because there are many answer graphs. In general, the proportion of the verification time decreases as d grows, since the number of answer graphs

Table 4.4: Average proportions of the parts of the query processing time (%). `Ours` = first level : second level : verification, `DS-Traces` = first level : graph canonization : search, `Traces` = graph canonization : search.

Dataset	d	`Ours`	`DS-Traces`	`Traces`
Human	1	0.01 : 0.07 : 99.92	0.01 : 99.96 : 0.03	99.97 : 0.03
	2	0.00 : 0.01 : 99.99	0.00 : 99.99 : 0.01	99.99 : 0.01
	3	0.00 : 0.02 : 99.98	N/A	N/A
HPRD	1	8.12 : 37.99 : 53.90	1.93 : 95.77 : 2.30	96.43 : 3.57
	2	6.93 : 21.92 : 71.15	5.33 : 93.98 : 0.69	99.12 : 0.88
	3	4.57 : 16.48 : 78.94	5.09 : 94.19 : 0.72	99.50 : 0.50
HepTh	1	0.11 : 0.39 : 99.50	1.66 : 97.24 : 1.11	94.86 : 5.14
	2	2.01 : 5.88 : 92.11	3.60 : 95.93 : 0.47	98.56 : 1.44
	3	4.56 : 18.36 : 77.09	2.04 : 97.67 : 0.29	99.46 : 0.54
CondMat	1	0.22 : 0.57 : 99.22	2.04 : 97.03 : 0.93	97.96 : 2.04
	2	8.35 : 16.35 : 75.29	0.39 : 99.58 : 0.04	99.94 : 0.06
	3	10.30 : 15.44 : 74.27	0.14 : 99.85 : 0.01	100.00 : 0.00
HepPh	1	0.06 : 0.18 : 99.75	0.91 : 98.46 : 0.62	99.90 : 0.10
	2	0.01 : 0.09 : 99.90	0.00 : 99.99 : 0.00	100.00 : 0.00
	3	0.10 : 0.62 : 99.29	N/A	N/A
Plus	1	0.02 : 0.11 : 99.87	1.56 : 97.04 : 1.40	97.80 : 2.20
	2	0.03 : 0.06 : 99.90	6.27 : 92.57 : 1.16	98.46 : 1.54
	3	9.70 : 12.98 : 77.32	11.27 : 87.93 : 0.80	99.10 : 0.90
Amazon	1	0.37 : 1.16 : 98.47	2.38 : 96.22 : 1.40	96.62 : 3.38
	2	30.92 : 11.18 : 57.89	24.29 : 75.08 : 0.63	98.02 : 1.98
	3	44.94 : 13.36 : 41.70	41.68 : 57.78 : 0.54	98.97 : 1.03
DBLP	1	0.09 : 0.30 : 99.62	1.71 : 97.26 : 1.04	97.72 : 2.28
	2	20.46 : 8.91 : 70.63	0.31 : 99.68 : 0.01	99.95 : 0.05
	3	23.01 : 3.16 : 73.83	N/A	N/A
Bigblue	1	0.05 : 0.32 : 99.63	1.28 : 97.22 : 1.50	97.29 : 2.71
	2	4.28 : 5.42 : 90.30	7.24 : 91.50 : 1.25	97.30 : 2.70
	3	91.22 : 1.35 : 7.43	82.57 : 16.78 : 0.66	99.11 : 0.89
LiveJournal	1	0.33 : 0.41 : 99.26	5.52 : 93.33 : 1.15	98.06 : 1.94
	2	97.50 : 0.47 : 2.03	94.81 : 5.07 : 0.12	99.84 : 0.16
	3	37.05 : 2.51 : 60.43	N/A	N/A

decreases and the false positive ratio of `Ours` is low. Regarding `DS-Traces`, if no data graph has an identical degree sequence as that of a query graph, it can determine that the answer set is an empty set without computing the canonical form of the query graph. When there are many such cases in a query set, the proportion of graph canonization time of `DS-Traces` significantly decreases (e.g., Amazon with $d = 3$, Bigblue with $d = 3$, Livejournal with $d = 2$). Regarding `Traces`, we can see that most of the time is used in graph canonization, and the search time is minor.

4.4.2 Varying Number of Data Graphs.

Next, we vary the number of data graphs, i.e., we set $|\mathcal{D}| = 1000, 2000, 4000, 8000$ with $d = 2$. We omit the results of false positive ratios and the breakdown of query processing time as the general trends are similar to those in Section 4.4.1.

Index Construction Time. Figure 4.7 shows the index construction time of the algorithms. As in Figure 4.4, the result of `CRaB` is omitted and missing bars of `IDAR` indicates that it fails to construct an index due to the memory error. `Ours` is the fastest for all the datasets, and it runs up to 3 orders of magnitude faster than `Traces` (HepPh in Figure 4.7e). On average `Ours` runs 759 times faster than `Traces`. Overall, the index construction time increases linearly regarding the number of data graphs.

Number of Items. Table 4.5 shows the number of items in the first level of our index. In general, the number of items increases linearly regarding the number of data graphs. The number of items in the first level is related to the average number of vertices of a data graph as we discussed in Section 4.4.1. Note that we set $d = 2$ in this section, and thus the average number of vertices merely changes with different number of data graphs.

Query Processing Time. Figure 4.8 shows the query processing time of the

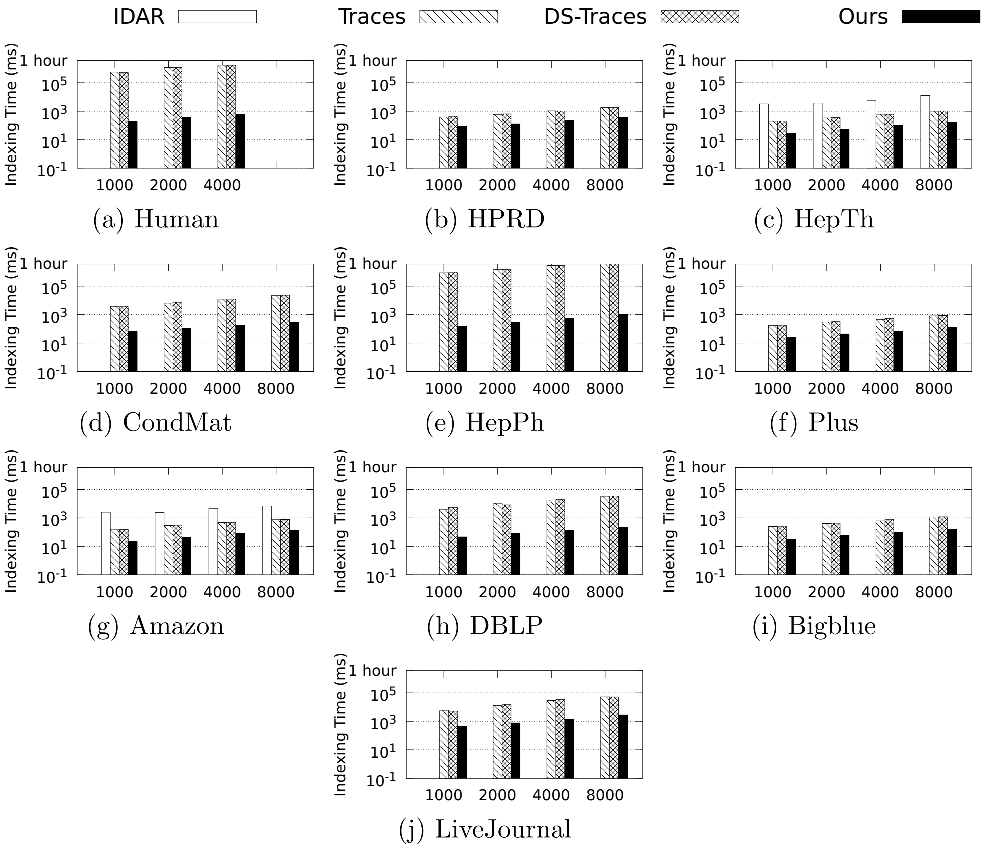


Figure 4.7: Index construction time for varying number of data graphs.

algorithms. Missing bars indicate that an index is not constructed for `IDAR`, `DS-Traces`, and `Traces`. We exclude the case of $|\mathcal{D}| = 8000$ for Human because the number of vertices in Human is less than 8000. Among the total 39 cases, `Ours` is the fastest for 31 cases, and `DS-Traces` is the fastest for 6 cases. For the 31 cases, `Ours` runs up to 74 times faster than `DS-Traces` (CondMat with $|\mathcal{D}| = 4000$ in Figure 4.8d) and runs 10 times faster than `DS-Traces` on average. `Ours` outperforms `Traces` for 32 cases, where it runs up to 889 times faster (DBLP with $|\mathcal{D}| = 1000$ in Figure 4.8h), and runs 112 times faster than `Traces`

Table 4.5: Number of items in the first level of our index ($d = 2$).

Dataset	$\lvert\mathcal{D}\rvert = 1000$	$\lvert\mathcal{D}\rvert = 2000$	$\lvert\mathcal{D}\rvert = 4000$	$\lvert\mathcal{D}\rvert = 8000$
Human	322	575	1325	-
HPRD	918	1776	3403	6529
HepTh	900	1684	3149	5789
CondMat	952	1837	3502	6459
HepPh	922	1725	3194	5856
Plus	565	1066	1909	3499
Amazon	976	1935	3807	7455
DBLP	941	1860	3631	7092
Bigblue	724	1377	2555	4734
LiveJournal	983	1952	3870	7675

on average. `DS-Traces` runs 51 times faster than `Ours` on average for 6 cases. Overall `DS-Traces` is faster than `Traces`, and it runs 22 times faster than `Traces` on average.

In general, the query processing time of the algorithms increases as $|\mathcal{D}|$ grows. The query processing time of `Traces` is less sensitive to the number of data graphs, because the graph canonization time dominates the query processing time of `Traces` as shown in Table 4.4. The query processing time of `DS-Traces` sometimes visibly increases as $|\mathcal{D}|$ grows (e.g., DBLP in Figure 4.8h and CondMat in Figure 4.8d). As the number of data graphs increases, $\mathcal{D}$ is more likely to contain a data graph that has an identical degree sequence as that of a query graph, and thus less queries are determined to have empty answer-sets before graph canonization. That is why the query processing time of `DS-Traces` visibly increased in these datasets.

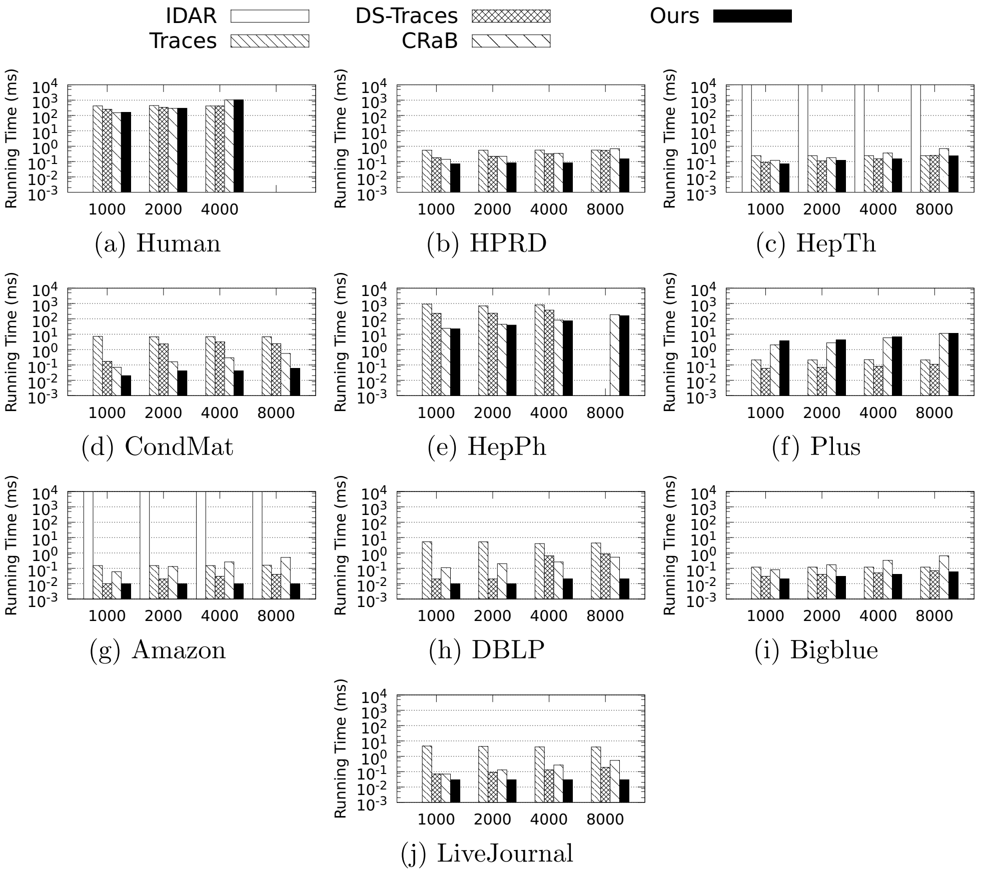


Figure 4.8: Query processing time for varying number of data graphs.

Chapter 5

Conclusion

5.1 Summary

We have proposed efficient algorithms for graph isomorphism and graph isomorphism query processing. The performances of the proposed algorithms were investigated using real-world datasets from various aspects.

In Chapter 2, we define the two main problems, i.e., graph isomorphism and graph isomorphism query processing. We also survey previous works on related problems including graph canonization, subgraph isomorphism, graph similarity search, and graph isomorphism on some graph classes.

In Chapter 3, we present a new approach to graph isomorphism, which combines the pairwise color refinement and efficient backtracking. Three techniques are introduced in our approach, which are (1) pairwise color refinement and binary cell mapping, (2) compressed candidate space, and (3) partial failing set, which together lead to a much faster and scalable algorithm for graph isomorphism. Extensive experiments are conducted to compare the performance of our algorithm against state-of-the-art algorithms on real-world datasets. The number of vertices in the datasets varies from tens to millions, and we classify the input instances into the case when two input graphs are isomorphic and the case when they are not isomorphic. Overall, our algorithm runs up to orders of magnitude faster than existing algorithms including nauty/Traces, which have been the best algorithms in the last decades. We also evaluate the effectiveness of each of the three new techniques by experiments.

In Chapter 4, we present an efficient algorithm for graph isomorphism query processing, which utilizes degree sequences and color-label distributions. We introduce (1) a two-level index based on the degree sequences and the canonical colorings of the data graphs and (2) an efficient query processing algorithm using the index. We evaluate our algorithm against state-of-the-art algorithms by experiments in terms of index construction time and query processing time. Experimental results on real datasets show that the index construction time of our algorithm is much smaller than that of existing algorithms, and our algorithm outperforms existing algorithms in terms of query processing time as the graph sizes increase. We also analyze the experimental results using false positive ratio and breakdown of query processing time.

5.2 Future Directions

Graph isomorphism and graph isomorphism query processing can be applied to various application domains including social networks, bioinformatics, chemistry, and so on. It is an interesting future work to develop an application-specific solution based on the proposed algorithms in this thesis.

Furthermore, applying our techniques to other problems is an interesting future work. In this section we discuss two problems related to graph isomorphism and graph isomorphism query processing. We also leave an open problem regarding the probability that two graphs are isomorphic.

Graph Homomorphism. A *homomorphism* [23, 17] from a graph G to a graph H is a function $f : V(G) \to V(H)$ such that u and $f(u)$ have the same label for every $u \in V(G)$, and $(f(u), f(v)) \in E(H)$ for every $(u, v) \in E(G)$. Note that homomorphisms are not one-to-one functions, and thus two vertices $u, v \in V(G)$ can be mapped to one vertex in $V(H)$. Given two graphs G and H, the graph homomorphism problem is to find a homomorphism from G to H. The decision version of the problem that asking whether there exists any homomorphism from G to H is NP-complete [28].

Graph homomorphism can be applied to RDF query processing [53, 16, 37, 71], where RDF (Resource Description Framework) is a standard for representing knowledge on the web. Extensive research has been done to develop efficient RDF query processing algorithms [54, 2, 82, 75, 77, 37]. A recent approach to RDF query processing is utilizing subgraph isomorphism algorithms based on backtracking by removing the injective condition in the matching conditions of subgraph isomorphism [82, 37].

Since the matching condition of graph isomorphism is different from that of graph homomorphism, modifications are needed to apply the techniques introduced in this paper to handle graph homomorphism. First of all, we need to modify the pairwise color refinement in such a way that two vertices are colored by a color in the resulting coloring if there is an homomorphism that maps the two vertices. Then, in order to use the binary cell mapping with respect to the above coloring, a new proof is required. We cannot use the compressed candidate space for the above coloring because Property 3.3.2 in Chapter 3 no longer holds, since $u \in V(G)$ can be mapped to $v \in V(H)$ in a homomorphism even if the degree of u is not equal to that of v. We also need to modify the backtracking procedure in order that a vertex in $V(H)$ can be mapped to multiple vertices in $V(G)$. While partial failing sets can still be used in backtracking for graph homomorphism, conflict-class will not be exist in the search tree because multiple vertices in $V(G)$ can be mapped to a vertex in $V(G)$, i.e., there are no conflicts of mappings.

Network Motif Discovery. Given a graph G, a *network motif* in G is a subgraph g of G such that g appears much more frequently in G than in random graphs whose degree distributions are similar to that of G [52, 73]. The Z-score of g is Z-score$(g) = \frac{N_g - N_g^r}{\sigma_g^r}$, where N_g is the number of occurrences of g in G, and N_g^r and σ_g^r are the mean number of occurrences of g and the standard deviation in random networks, respectively. Network motifs can be applied to graph analysis in various applications [74].

Given a real graph G, a set of random graphs $\mathcal{G} = \{G_1, G_2, \ldots, G_r\}$, a natural number $k > 2$, and a threshold $\alpha > 0$, network motif discovery is the problem of finding all network motifs in G which have k vertices and Z-score higher than α. In real applications a user hardly has random graphs for the real graph. Thus, most network motif discovery algorithms [70, 55, 32, 60, 34, 45] are designed to generate random graphs and compute Z-score with respect to the real graph and the generated random graphs. A common approach to network motif discovery is enumerating all subgraphs of G having k vertices (enumeration) and counting the number of occurrences for each distinct subgraph (classification), where the classification is known to be the most time-consuming part [21, 34].

Our techniques for graph isomorphism query processing can be applied to the classification task of network motif discovery, where we need to check whether an enumerated subgraph is isomorphic to any of the previously enumerated subgraphs. If no previously enumerated subgraphs are isomorphic to the newly enumerated subgraph, then the new subgraph should be stored for the next check. This requires the set of data graphs in graph isomorphism query processing to be dynamic. Therefore, we need to implement each level of our index for graph isomorphism query processing by a balanced search tree to handle the classification task.

Probability that Two Graphs are Isomorphic. Given two graphs with an identical degree sequence *ds*, what is the probability that the two graphs are isomorphic? The probability can be expressed as follows:

$$P_{ds} = \frac{\Sigma_{S \in \mathbb{I}_{ds}} |S|^2}{|\mathbb{G}_{ds}|^2},$$

where $\mathbb{G}_{ds}$ is the set of graphs having degree sequence *ds*, and $\mathbb{I}_{ds}$ is the set of subsets of $\mathbb{G}_{ds}$ in which graphs are pairwise isomorphic. For example, $\mathbb{G}_{ds}$ with $ds = (2, 2, 2, 1, 1)$ is shown in Figure 5.1, where $\mathbb{I}_{ds}$ consists of two subsets marked by A and B. The number of graphs in $\mathbb{G}_{ds}$ is 40, and the number of graphs in A and B are 30 and 10, respectively. Thus, we have $P_{ds} = \frac{30^2+10^2}{40^2} = 0.625$.

An asymptotic estimation of the number of graphs with a given degree sequence (i.e., $|\mathbb{G}_{ds}|$) has been studied [5, 6], but no research carried out to count (or estimate) the number of isomorphic graphs in $\mathbb{G}_{ds}$. If we find a formula of P_{ds} in terms of the given degree sequence, then we can answer interesting questions arise in graph analysis, e.g., in which case P_{ds} is higher? (when degrees are evenly distributed like in regular graphs or when degrees are distributed like in Hub-and-Spoke networks?) Therefore, it is an interesting open problem to find a formula for P_{ds} in terms of the degree sequence.

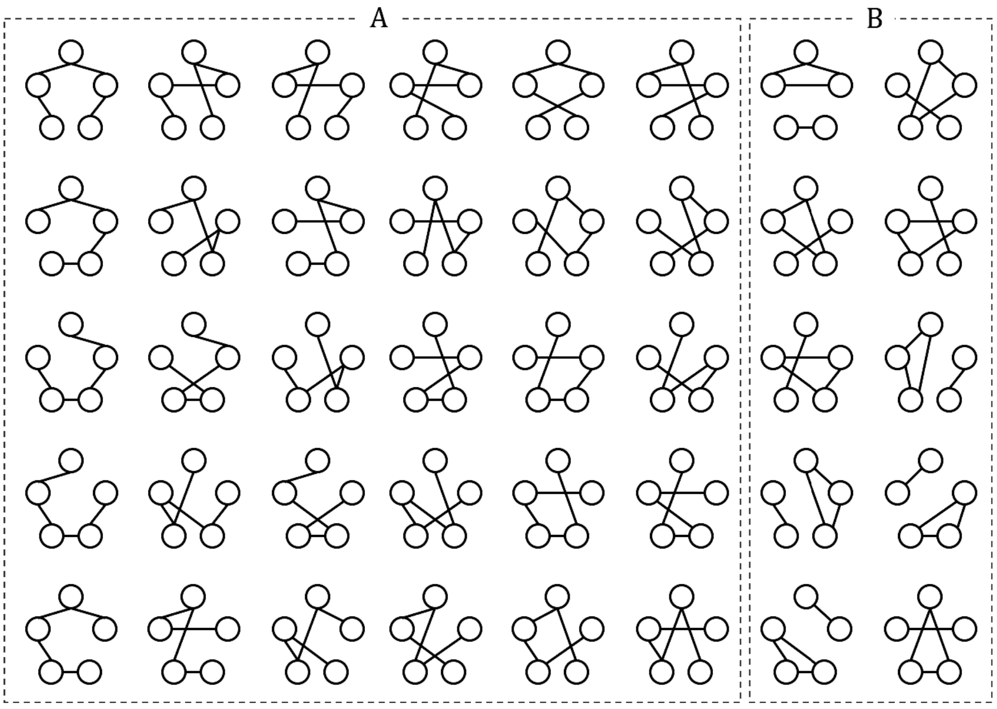


Figure 5.1: Graphs with degree sequence $(2, 2, 2, 1, 1)$.

Bibliography

[1] V. Arvind, J. Köbler, G. Rattan, and O. Verbitsky. On the power of color refinement. In *Proceedings of the International Symposium on Fundamentals of Computation Theory*, pages 339–350, 2015.

[2] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix “Bit” loaded: A scalable lightweight join query processor for RDF data. In *Proceedings of the International Conference on World Wide Web*, pages 41–50, 2010.

[3] L. Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 684–697, 2016.

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

[5] A. Barvinok. On the number of matrices and a random matrix with prescribed row and column sums and 0–1 entries. *Advances in Mathematics*, 224(1):316–339, 2010.

[6] A. Barvinok and J. A. Hartigan. The number of graphs and a random graph with a given degree sequence. *Random Structures & Algorithms*, 42(3):301–348, 2013.

[7] C. Berkholz, P. Bonsma, and M. Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory of Computing Systems*, 60(4):581–614, 2017.

[8] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing Cartesian products. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1199–1214, 2016.

[9] K. S. Booth and G. S. Lueker. Linear algorithms to recognize interval graphs and test for the consecutive ones property. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 255–265, 1975.

[10] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19(1):85–98, 1982.

[11] V. Carletti, P. Foggia, A. Saggese, and M. Vento. Introducing VF3: A new algorithm for subgraph isomorphism. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 128–139, 2017.

[12] L. Chang, X. Feng, X. Lin, L. Qin, W. Zhang, and D. Ouyang. Speeding up GED verification for graph similarity search. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 793–804, 2020.

[13] X. Chen, H. Huo, J. Huan, and J. S. Vitter. An efficient algorithm for graph edit distance computation. *Knowledge-Based Systems*, 163:762–775, 2019.

[14] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *The VLDB Journal*, 20(4):521–539, 2011.

[15] G. Ciriello and C. Guerra. A review on models and algorithms for motif discovery in protein–protein interaction networks. *Briefings in Functional Genomics and Proteomics*, 7(2):147–156, 2008.

[16] O. Corby and C. Faron-Zucker. Implementation of SPARQL query language based on graph homomorphism. In *Proceedings of the International Conference on Conceptual Structures*, pages 472–475, 2007.

[17] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *Proceedings of the International Conference on Very Large Data Bases*, 3(1-2):1161–1172, 2010.

[18] Z. Galil, C. M. Hoffmann, E. M. Luks, C. P. Schnorr, and A. Weber. An $O(n^3 \log n)$ deterministic and an $O(n^3)$ Las Vegas isomorphism test for trivalent graphs. *Journal of the ACM*, 34(3):513–531, 1987.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[20] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. GRAPES: A software for parallel searching on biological graphs targeting multi-core architectures. *PLOS One*, 8(10):e76911, 2013.

[21] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proceedings of the International Conference on Research in Computational Molecular Biology*, pages 92–106. Springer, 2007.

[22] G. Gu, Y. Nam, K. Park, Z. Galil, G. F. Italiano, and W.-S. Han. Scalable graph isomorphism: Combining pairwise color refinement and backtracking

via compressed candidate space. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1368–1379, 2021.

[23] G. Hahn and C. Tardif. Graph homomorphisms: structure and symmetry. In *Graph Symmetry*, pages 107–166. 1997.

[24] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1429–1446, 2019.

[25] W.-S. Han, J. Lee, and J.-H. Lee. Turbo$_{\text{ISO}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2013.

[26] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 405–418, 2008.

[27] H. A. Helfgott, J. Bajpai, and D. Dona. Graph isomorphisms in quasi-polynomial time. *arXiv preprint, arXiv:1710.04574*, 2017.

[28] P. Hell and J. Nešetřil. On the complexity of H-coloring. *Journal of Combinatorial Theory, Series B*, 48(1):92–110, 1990.

[29] J. E. Hopcroft and J.-K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[30] T. Junttila and P. Kaski. Conflict propagation and component recursion for canonical labeling. In *Proceedings of the International Conference on*

Theory and Practice of Algorithms in (Computer) Systems, pages 151–162, 2011.

[31] A. Jüttner and P. Madarasi. VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018.

[32] Z. R. M. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad. Kavosh: A new algorithm for finding network motifs. *Bioinformatics*, 10(1):1–12, 2009.

[33] K. Kazerounian, K. Latif, K. Rodriguez, and C. Alvarado. Nano-kinematics for analysis of protein molecules. *Journal of Mechanical Design*, 127(4):699–711, 2005.

[34] S. Khakabimamaghani, I. Sharafuddin, N. Dichter, I. Koch, and A. Masoudi-Nejad. QuateXelero: An accelerated exact network motif detection algorithm. *PLOS One*, 8(7):e68073, 2013.

[35] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2021 (to appear).

[36] H. Kim, S. Min, K. Park, X. Lin, S.-H. Hong, and W.-S. Han. IDAR: Fast supergraph search using DAG integration. *Proceedings of the International Conference on Very Large Data Bases*, 13(9):1456–1468, 2020.

[37] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi. Taming subgraph isomorphism for RDF query processing. *Proceedings of the International Conference on Very Large Data Bases*, 8(11), 2015.

[38] J. Köbler. On graph isomorphism for restricted graph classes. In *Proceedings of the Conference on Computability in Europe*, pages 241–256, 2006.

[39] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity.* Birkhäuser Boston, 1993.

[40] W. Kocay. On writing isomorphism programs. In *Computational and Constructive Design Theory*, pages 135–175. 1996.

[41] Y. Kumar and P. Gupta. External memory layout vs. schematic. *ACM Transactions on Design Automation of Electronic Systems*, 14(2):1–20, 2009.

[42] J. Kunegis. KONECT: The Koblenz network collection. In *Proceedings of the International Conference on World Wide Web*, pages 1343–1350, 2013.

[43] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[44] Y. Liang and P. Zhao. Similarity search in graph databases: A multi-layered indexing approach. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 783–794, 2017.

[45] W. Lin, X. Xiao, X. Xie, and X.-L. Li. Network motif discovery: A GPU approach. *IEEE Transactions on Knowledge and Data Engineering*, 29(3):513–528, 2016.

[46] X. Liu and D. Klein. The graph isomorphism problem. *Journal of Computational Chemistry*, 12(10):1243–1251, 1991.

[47] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982.

[48] B. Lyu, L. Qin, X. Lin, L. Chang, and J. X. Yu. Scalable supergraph search in large graph databases. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 157–168, 2016.

[49] B. McKay and A. Piperno. Benchmark graphs in the nauty Traces page. `http://pallini.di.uniroma1.it/Graphs.html`.

[50] B. D. McKay. Practical graph isomorphism. *Congressus Numeranitum*, 30:45–87, 1981.

[51] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

[52] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[53] M.-L. Mugnier. Knowledge representation and reasonings based on graph homomorphism. In *Proceedings of the International Conference on Conceptual Structures*, pages 172–192, 2000.

[54] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *Proceedings of the International Conference on Very Large Data Bases*, 3(1-2):256–263, 2010.

[55] S. Omidi, F. Schreiber, and A. Masoudi-Nejad. MODA: An efficient algorithm for network motif discovery in biological networks. *Genes & Genetic Systems*, 84(5):385–395, 2009.

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

[57] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: On compression and computation. *Proceedings of the International Conference on Very Large Data Bases*, 11(2):176–188, 2017.

[58] M. Randić. On canonical numbering of atoms in a molecule and graph isomorphism. *Journal of Chemical Information and Computer Sciences*, 17(3):171–180, 1977.

[59] R. C. Read and D. G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[60] P. Ribeiro and F. Silva. G-tries: An efficient data structure for discovering network motifs. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1559–1566, 2010.

[61] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, SRI-CSL-98-04, Computer Science Laboratory, SRI International, 1998.

[62] A. Sanfeliu and K.-S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, (3):353–362, 1983.

[63] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A large ontology from wikipedia and wordnet. *Journal of Web Semantics*, 6(3):203–217, 2008.

[64] S. Sun and Q. Luo. Scaling up subgraph query processing with efficient subgraph matching. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 220–231, 2019.

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

[66] The National Institutes of Health (NIH). PubChem. `https://pubchem.ncbi.nlm.nih.gov`.

[67] B. K. Tripathy and G. K. Panda. A new approach to manage security against neighborhood attacks in social networks. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining*, pages 264–269, 2010.

[68] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[69] T. Watanabe, M. Endo, and N. Miyahara. A new automatic logic interconnection verification system for VLSI design. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 2(2):70–82, 1983.

[70] S. Wernicke and F. Rasche. FANMOD: A tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[71] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr. RDF data storage and query processing schemes: A survey. *ACM Computing Surveys*, 51(4):1–36, 2018.

[72] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the IEEE International Conference on Data Mining*, pages 721–724, 2002.

[73] S. Yu, Y. Feng, D. Zhang, H. D. Bedru, B. Xu, and F. Xia. Motif discovery in networks: A survey. *Computer Science Review*, 37:100267, 2020.

[74] S. Yu, J. Xu, C. Zhang, F. Xia, Z. Almakhadmeh, and A. Tolba. Motifs in big networks: Methods and applications. *IEEE Access*, 7:183322–183338, 2019.

[75] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: a fast and compact system for large scale RDF data. *Proceedings of the International Conference on Very Large Data Bases*, 6(7):517–528, 2013.

[76] K. Zeng, X. Fan, M. Dong, and P. Yang. A fast algorithm for kinematic chain isomorphism identification based on dividing and matching vertices. *Mechanism and Machine Theory*, 72:25–38, 2014.

[77] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *Proceedings of the International Conference on Very Large Data Bases*, 6(4):265–276, 2013.

[78] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. GSI: GPU-friendly subgraph isomorphism. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1249–1260, 2020.

[79] X. Zhao, C. Xiao, X. Lin, W. Zhang, and Y. Wang. Efficient structure similarity searches: A partition-based approach. *The VLDB Journal*, 27(1):53–78, 2018.

[80] B. Zhou and J. Pei. Preserving privacy in social networks against neighborhood attacks. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 506–515, 2008.

[81] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu. Mining top-k large structural patterns in a massive network. *Proceedings of the International Conference on Very Large Data Bases*, 4(11):807–818, 2011.

[82] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the International Conference on Very Large Data Bases*, 4(8):482–493, 2011.

요약

그래프 동형 문제는 소셜 네트워크 서비스, 생물정보학, 화학정보학 등등 다양한 응용 분야에서 그래프 분석을 위해 다루고 있는 핵심 문제이다. 실생활에서 다루는 그래프 데이터의 크기가 커져 감에 따라, 대용량의 그래프를 처리할 수 있는 그래프 동형 알고리즘의 필요성이 높아지고 있다. 그러나 현재 존재하는 그래프 동형 알고리즘들은 대용량의 그래프에 대해서 시간 혹은 공간 측면에서 한계를 보여준다. 응용 분야 중에서는 여러 개의 그래프들 중에서 하나의 쿼리 그래프와 동형인 그래프를 모두 찾는 문제, 즉 그래프 동형 쿼리 프로세싱을 종종 요구하기도 한다. 본 논문에서는 대용량의 실제 그래프 데이터에 대해서 그래프 동형 문제와 그래프 동형 쿼리 프로세싱 문제를 빠르게 푸는 알고리즘들을 제안한다.

첫 번째로, 본 논문에서는 그래프 동형 문제를 위한 빠르고 확장성 있는 알고리즘을 제안한다. 이를 위해 쌍별 색 개선(pairwise color refinement)과 효율적인 백트래킹으로 구성된 프레임워크를 소개한다. 이 프레임워크 내에서 세 가지 효율적인 테크닉을 사용한다. 실제 그래프 데이터에 대한 실험을 통해 본 알고리즘이 현존하는 가장 빠른 알고리즘들보다 평균 수천 배 빠름을 보였다.

두 번째로, 본 논문에서는 그래프 동형 쿼리 프로세싱을 위한 효율적인 알고리즘을 개발한다. 본 알고리즘은 차수열과 색-레이블 분포를 이용한 인덱스를 이용한다. 실제 그래프 데이터에 대한 실험을 통해 본 알고리즘이 현존하는 알고리즘들보다 인덱싱 시간에서는 항상 평균 수천 배 빠르고, 쿼리 처리 시간에서는 중·대용량의 그래프들에 대해서 평균 수십 배 빠르게 동작하는 것을 보였다.

주요어: 그래프 동형; 그래프 동형 쿼리 프로세싱; 쌍별 색 개선; 백트래킹; 부분 실패 집합

학번: 2014-21758