Set-Centric Subgraph Isomorphism

Bachelor Thesis

Author(s):
Kapp-Schwoerer, Lukas

Publication date:
2021-02-07

Permanent link:
https://doi.org/10.3929/ethz-b-000506421

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Set-Centric Subgraph Isomorphism

Bachelor Thesis

Lukas Kapp-Schwoerer

February 7, 2021

Advisors: Maciej Besta, Prof. Dr. Torsten Hoefler
Department of Computer Science, ETH Zürich

Abstract

We investigate techniques for accelerating subgraph isomorphism (SI). SI is the task of finding occurrences of a pattern graph in a target graph and relevant to many practical applications. Our key idea is to use set algebra based formulations of SI. This facilitates employing set data structures to test the feasibility of partial mappings from the pattern graph to the target graph. The methods we use involve a range of algorithm variants with different set-centric formulations of subgraph isomorphism, different set data structures, and different parallelization schemes. Our results show that our set-centric variants outperform the baseline on some, but not all, pattern and target graphs. We conclude that the relative performance of the algorithm variants is strongly impacted by the choice of the input and propose further research directions to develop a more powerful algorithm.

Contents

Contents **iii**

1 Introduction **1**

2 Notation **3**

3 Background **4**
3.1 Subgraph Isomorphism 4
3.1.1 Problem Definition 4
3.1.2 VF2 Algorithm 5
3.2 Data-Structures for Sets 10
3.2.1 Arrays, Dense Bitmaps 10
3.2.2 Roaring Bitmaps 10
3.2.3 Robin Hood Hashmap 11

4 Set-Centric Subgraph Isomorphism **12**
4.1 Parallel Baseline 12
4.2 Set-Centric Implementation 15

5 Experiments **21**
5.1 Experimental Setup 21
5.1.1 Hardware and software setup 21
5.1.2 Measurement and reporting procedure 21
5.1.3 Pattern and target graphs 22
5.2 Results 23
5.2.1 Performance distribution over all variants 23
5.2.2 Performance comparison of specific VF2 variants . . . 23

6 Other Research Directions **30**
6.1 Frequent Subgraph Mining 30
6.2 Improving Sparse Bitmaps 30

6.2.1 Relabeling Vertices 31
6.2.2 Multiple Levels of Indirection 31

7 Conclusion **32**

Bibliography **33**

Chapter 1

Introduction

Subgraph isomorphism (SI) is the NP-complete problem to find occurrences of a pattern graph in a target graph. It is an important problem in structural pattern recognition with applications in biology [2, 16, 32], circuit design [30] and social sciences [33, 34]. Solving subgraph isomorphism is also a subroutine for other problems such as frequent subgraph mining [14, 20, 22].

Since the inception of VF2 [12] there has been continued research on accelerating subgraph isomorphism. In recent years, this has included techniques on parallel subgraph isomorphism [15, 4, 25]. On the other hand, for other graph problems such as maximal clique listing and triangle counting set-centric algorithms have been studied [17].

Our goal is to explore if such a set-centric approach can be applied for subgraph isomorphism. Many algorithms for subgraph isomorphism such as VF [9, 10], VF2 [11], VF2+ [6], VF2++ [21], VF3 [5], VF3-Light [3], and RI [2, 1] share the same algorithmic approach. They explore the state space of partial mappings between pattern and target graph in a depth-first order, while using heuristics to narrow down and speed up the search. We therefore focus on developing set-centric variants of one such algorithm with relatively simple heuristics, namely set-centric variants of VF2 [11].

To this end, we 1) implement a set-centric version of the heuristics in VF2; 2) develop an alternative set-centric formulation of subgraph isomorphism and implement a corresponding algorithm variant; 3) test the algorithm variants with set data structures based on roaring bitmaps [8, 24, 23] and robin hood hashmaps [7, 13, 7]; 4) test the scaling behaviour of our algorithm variants using different parallelization approaches with OMP [31].

We find that our set-centric VF2 variants outperform the non-set-based parallel baseline on some inputs. However, on many inputs, the baseline runs faster than any of our set-centric variants. We conclude that the choice of pattern and target graph heavily impacts how the analyzed algorithms per-

form relative to each other and propose further research directions to develop a more powerful algorithm for subgraph isomorphism.

Chapter 2

Notation

For quick reference we present an overview of our notation in the table below.

Notation	Description
$G = (V, E, l)$	A labelled graph with label function l
G_{P}	A pattern graph $(V_{\mathrm{P}}, E_{\mathrm{P}}, l_{\mathrm{P}})$
G_{T}	A target graph $(V_{\mathrm{T}}, E_{\mathrm{T}}, l_{\mathrm{T}})$
$M(s) \subseteq V_{\mathrm{P}} \times V_{\mathrm{T}}$	The mapping in state s
$M_{\mathrm{P}}(s)$	The vertices of the Graph G_{P} covered by $M(s)$
$M_{\mathrm{T}}(s)$	The vertices of the Graph G_{T} covered by $M(s)$
$N(G, u)$	The neighborhood of vertex u in graph G
$\mathrm{Pred}(G, u)$	The predecessors of vertex u in graph G
$\mathrm{Succ}(G, u)$	The successors of vertex u in graph G

Chapter 3

Background

3.1 Subgraph Isomorphism

3.1.1 Problem Definition

The subgraph isomorphism (SI) problem is about finding occurrences of a pattern graph G_P in a target graph G_T. It is a generalization of the graph isomorphism problem, which seeks to find a bijection between two graphs. In both problems, the mappings between the graphs can be constrained by vertex and edge labels. For this purpose, we can define a labelled graph as follows.

Definition 3.1 *A **labeled graph** $G = (V, E, l)$ is a graph with vertices E, directed or undirected edges E and a label function l. The label function associates vertex $v \in V$ with label $l(v)$ and edge $e \in E$ with label $l(e)$.*

Formally graph isomorphism is defined by Shuming et al. [19] as

Definition 3.2 *A graph $G = (V, E, l)$ is **isomorphic** to $H = (V', E', l')$ if and only if there exists a bijective mapping $f : V \to V'$ such that 1. $\forall u, v \in V : (u, v) \in E \leftrightarrow (f(u), f(v)) \in E'$, 2. $\forall u \in V : l(u) = l'(f(u))$, 3. $\forall (u, v) \in E : l((u, v)) = l'((f(u), f(v)))$.*

With this definition, we can define the graph isomorphism problem as

Definition 3.3 *The **graph isomorphism problem** is to decide whether two given labeled graphs $G = (V, E, l)$ and $H = (V', E', l')$ are isomorphic.*

As a generalization of definition 3.3 we can now define the subgraph isomorphism problem as

Definition 3.4 *Given two labeled graphs $G_P = (V_P, E_P, l_P)$ (called **pattern**) and $G_T = (V_T, E_T, l_T)$ (called **target**) the **subgraph isomorphism problem** is to decide whether there exists a subgraph of G_T that is isomorphic to G_P.*

With these definitions we follow the convention of the authors of VF2, Cordella et al. [12, 11], to define graph isomorphism based on induced graphs, i.e. there may not be extra edges present in the target graph which are not present in the pattern graph.

While definitions 3.3 and 3.4 are a decision problems, in many practical applications a listing or counting of the valid mappings is required [2, 16, 30]. We will focus on algorithmic approaches that count or list valid mappings for subgraph isomorphism.

3.1.2 VF2 Algorithm

VF2 is an algorithm for listing subgraph isomorphisms given a pattern graph $G_\mathrm{P} = (V_\mathrm{P}, E_\mathrm{P}, l_\mathrm{P})$ and a target graph $G_\mathrm{T} = (V_\mathrm{T}, E_\mathrm{T}, l_\mathrm{T})$. It was introduced by Cordella et al. [12] and later improved by the same authors [11]. In this work, we will refer with VF2 to this improved version.

The fundamental approach of VF2 is to recursively extend partial mappings between the pattern graph and the target graph. To this end, a state s contains a partial mapping $M(s) \subseteq V_\mathrm{P} \times V_\mathrm{T}$ and auxiliary variables for faster computations. This algorithmic approach can be viewed as searching for valid mappings in a state space as defined by [29]. VF2 traverses this state space with a depth-first strategy starting with an empty mapping. Going from a state s to a state s' corresponds to adding a pair (u, v) to the partial mapping $M(s)$. To ensure listing only valid mappings, at each step, it is verified that the new state s' is *consistent* with the graph structures and labels. Formally verifying that a state s' is consistent means ensuring that the vertices covered by $M(s')$ are isomorphic according to definition 3.2.

For many inputs the fraction of consistent states among all possible states is small. VF2 exploits this by not only ensuring consistency when traversing from s to s', but also by using additional heuristics called *k-look-ahead-rules* that exclude some states for which after k steps no consistent successor exists. Both the checking of consistency for adding (u, v) to s and the k-look-ahead-rules are subsumed as *feasibility rules*. These can be split up as

$$F(s, u, v) = F_\mathrm{syn}(s, u, v) \wedge F_\mathrm{sem}(s, u, v), \tag{3.1}$$

where syntactic feasibility rules $F_\mathrm{syn}(s, u, v)$ ensure consistency with regards to the structure of the graph and semantic feasibility rules $F_\mathrm{sem}(s, u, v)$ ensure consistency with regards to the labels. These rules will be discussed in detail later.

A high-level overview of the VF2 algorithm is displayed in figure 3.1. For traversing recursively through the state space, VF2 starts with an empty partial mapping. On each level of the recursion it first generates a list of candidate pairs $P \subseteq V_\mathrm{P} \times V_\mathrm{T}$ that could be added to current state s. For each

Algorithm 1: VF2

Input: An intermediate state s.

Output: Mappings between the graphs.

```
if |V_P| = |M(s)| then
    output M(s);
else
    P = Candidates(s)
    foreach (u, v) ∈ P do
        if Feasible(s, u, v) then
            s' = NewSate(s, u, v) ;          // (u, v) added to M(s)
            VF2(s');
        end
    end
end
```

Figure 3.1: VF2 algorithm

such pair $p = (u, v) \in P$ it is checked if the feasibility rules $F(s, u, v)$ are fulfilled. If $F(s, u, v)$ evaluates to true for a candidate (u, v), the successor state s' is generated by adding (u, v) to the partial mapping in s and VF2 is recursively called on s'.

Auxiliary variables

[11] use several auxiliary variables that are derived from the partial mapping $M(s)$. These are used in the candidate generation as well as feasibility rules and are defined as follows. The vertices of the pattern graph, V_{P}, respectively of the target graph, V_{T}, which are covered by $M(s)$ are denoted by $M_{\mathrm{P}}(s)$ respectively $M_{\mathrm{T}}(s)$. Formally

$$M_{\mathrm{P}}(s) = \{u \mid \exists v : (u, v) \in M(s)\}, \tag{3.2}$$

$$M_{\mathrm{T}}(s) = \{v \mid \exists u : (u, v) \in M(s)\}. \tag{3.3}$$

The vertices that are neighboring a vertex in $M_{\mathrm{P}}(s)$ respectively $M_{\mathrm{T}}(s)$ but are not covered by $M(s)$ are denoted by $T_{\mathrm{P}}^{\mathrm{out}}(s)$ for outgoing edges from $M_{\mathrm{P}}(s)$ and $T_{\mathrm{P}}^{\mathrm{in}}(s)$ for ingoing edges to $M_{\mathrm{P}}(s)$, respectively $T_{\mathrm{T}}^{\mathrm{out}}(s)$ and $T_{\mathrm{T}}^{\mathrm{in}}(s)$. With $\mathrm{Pred}(G, u)$ denoting the predecessors of a vertex u in a graph G and

$\mathrm{Succ}(G, u)$ denoting its successors, this can be formally written as

$$T_{\mathrm{P}}^{\mathrm{out}}(s) = (\bigcup_{u \in M_{\mathrm{P}}(s)} \mathrm{Succ}(G_{\mathrm{P}}, u)) - M_{\mathrm{P}}(s), \tag{3.4}$$

$$T_{\mathrm{P}}^{\mathrm{in}}(s) = (\bigcup_{u \in M_{\mathrm{P}}(s)} \mathrm{Pred}(G_{\mathrm{P}}, u)) - M_{\mathrm{P}}(s), \tag{3.5}$$

$$T_{\mathrm{T}}^{\mathrm{out}}(s) = (\bigcup_{v \in M_{\mathrm{T}}(s)} \mathrm{Succ}(G_{\mathrm{T}}, v)) - M_{\mathrm{T}}(s), \tag{3.6}$$

$$T_{\mathrm{T}}^{\mathrm{in}}(s) = (\bigcup_{v \in M_{\mathrm{T}}(s)} \mathrm{Pred}(G_{\mathrm{T}}, v)) - M_{\mathrm{T}}(s). \tag{3.7}$$

These variables are combined to terms using ingoing and outgoing edges, namely

$$T_{\mathrm{P}}(s) = T_{\mathrm{P}}^{\mathrm{out}}(s) \cup T_{\mathrm{P}}^{\mathrm{in}}(s), \tag{3.8}$$

$$T_{\mathrm{T}}(s) = T_{\mathrm{T}}^{\mathrm{out}}(s) \cup T_{\mathrm{T}}^{\mathrm{in}}(s). \tag{3.9}$$

Lastly, the remaining vertices that are neither in the partial mapping nor adjacent to a vertex in the partial mapping are denoted by $\tilde{V}_{\mathrm{P}}(s)$ and $\tilde{V}_{\mathrm{T}}(s)$. Formally this can be expressed as

$$\tilde{V}_{\mathrm{P}}(s) = V_{\mathrm{P}} - M_{\mathrm{P}}(s) - T_{\mathrm{P}}(s), \tag{3.10}$$

$$\tilde{V}_{\mathrm{T}}(s) = V_{\mathrm{T}} - M_{\mathrm{T}}(s) - T_{\mathrm{T}}(s). \tag{3.11}$$

Candidate Generation

A state can be reached through several paths in the state space. To reach each consistent state exactly once VF2 defines an arbitrary total order relation $\prec$ on V_{T}. With this total order relation a pair $(u, v) \in V_{\mathrm{P}} \times V_{\mathrm{T}}$ is only added to the candidate pairs if the partial mapping does not already contain a pair (u', v') with $v' \prec v$.

Given this total order relation $\prec$ on V_{T} the candidates are generated according to algorithm 2 in figure 3.2. Firstly, in line 1 the possible mappings between vertices in $T_{\mathrm{P}}^{\mathrm{out}}(s)$ and $T_{\mathrm{T}}^{\mathrm{out}}(s)$ are considered. Recall from equations 3.4 and 3.6 that these are those vertices not yet in the partial mapping $M_{\mathrm{P}}(s)$ respectively $M_{\mathrm{T}}(s)$, which are connected to a vertex already in the partial mapping by an edge going out of $M_{\mathrm{P}}(s)$ respectively $M_{\mathrm{T}}(s)$. If there are no such candidates, the same cartesian product is considered for ingoing edges in line 3. Finally, if this still has not yielded any candidates, then the cartesian product of all vertices not yet mapped is considered in line 6. Note that this can not be empty as long as the mapping is partial, and that this case is always taken when the routine is first called with empty partial mappings. From this procedure to generate candidate it immediately follows that the the obtained mapping is injective. For later use, we formalize this in the following corollary.

Algorithm 2: Candidates

Input: An intermediate state s.

Output: Candidate pairs for extending the partial mapping of s.

1 $P = \{(u,v) \mid (u,v) \in T_{\mathrm{P}}^{\mathrm{out}}(s) \times T_{\mathrm{T}}^{\mathrm{out}}(s) \wedge v \prec \min(M_{\mathrm{T}}(s))\}$;

2 **if** $P = \emptyset$ **then**

3 $\quad P = \{(u,v) \mid (u,v) \in T_{\mathrm{P}}^{\mathrm{in}}(s) \times T_{\mathrm{T}}^{\mathrm{in}}(s) \wedge v \prec \min(M_{\mathrm{T}}(s))\}$;

4 **end**

5 **if** $P = \emptyset$ **then**

6 $\quad P = \{(u,v) \mid (u,v) \in (V_{\mathrm{P}}(s) - M_{\mathrm{P}}(s)) \times (V_{\mathrm{T}}(s) - M_{\mathrm{T}}(s)) \wedge v \prec \min(M_{\mathrm{T}}(s))\}$;

7 **end**

8 **return** P

Figure 3.2: VF2 candidate pair generation

Corollary 3.5 *At any point during the execution of VF2 it holds for the state s that $M(s)$ represents an injective mapping from $M_{\mathrm{P}}(s) \subseteq V_{\mathrm{P}}$ to $M_{\mathrm{T}}(s) \subseteq V_{\mathrm{T}}$, as well as an injective mapping from $M_{\mathrm{T}}(s) \subseteq V_{\mathrm{T}}$ to $M_{\mathrm{P}}(s) \subseteq V_{\mathrm{P}}$, i.e.*

$$M(s) \subseteq V_{\mathrm{P}} \times V_{\mathrm{T}}$$

and

$$\forall (u,v),(u',v') \in M(s) : u = u' \leftrightarrow v = v'.$$

Feasibility Rules

When adding a pair $(u,v) \in V_{\mathrm{P}} \times V_{\mathrm{T}}$ to the partial mapping $M(s)$, VF2 checks the feasibility rules $F(s,u,v)$. In the high-level overview of VF2 in figure 3.1 this is done on line 6. In equation 3.1 we have seen that the feasibility rules are the conjunction of syntactic and semantic feasibility. We will now look at these in more detail.

The rules for syntactic feasibility $F(s,u,v)$ ensure that the partial mapping $M(s) \cup \{(u,v)\}$ is consistent with regards to the structure of the graphs, i.e.

$$\forall (u',v'),(u'',v'') \in M(s) \cup \{(u,v)\} : (u',u'') \in E_{\mathrm{P}} \leftrightarrow (v',v'') \in E_{\mathrm{T}}. \quad (3.12)$$

If s is consistent, then to guarantee the consistency of s' obtained by adding (u,v) to $M(s)$ it is sufficient to check neighbors of u and v. This is what the

rules $R_{\text{pred}}(s,u,v)$ and $R_{\text{succ}}(s,u,v)$ check for predecessors and successors, formally

$$\begin{aligned} &R_{\text{Pred}}(s,u,v) \Leftrightarrow \\ &\quad (\forall u' \in M_{\text{P}}(s) \cap \text{Pred}(G_{\text{P}},u)\ \exists v' \in \text{Pred}(G_{\text{T}},v) : (u',v') \in M(s)) \wedge \\ &\quad (\forall v' \in M_{\text{T}}(s) \cap \text{Pred}(G_{\text{T}},v)\ \exists u' \in \text{Pred}(G_{\text{P}},u) : (u',v') \in M(s)) \end{aligned} \tag{3.13}$$

$$\begin{aligned} &R_{\text{Succ}}(s,u,v) \Leftrightarrow \\ &\quad (\forall u' \in M_{\text{P}}(s) \cap \text{Succ}(G_{\text{P}},u)\ \exists v' \in \text{Succ}(G_{\text{T}},v) : (u',v') \in M(s)) \wedge \\ &\quad (\forall v' \in M_{\text{T}}(s) \cap \text{Succ}(G_{\text{T}},v)\ \exists u' \in \text{Succ}(G_{\text{P}},u) : (u',v') \in M(s)) \end{aligned} \tag{3.14}$$

These two rules would be sufficient to ensure equation 3.12. However, Cordella et al. [11] add the rules R_{in} and R_{out} that are regarding syntactic feasibility for states s'' succeeding s' in the state space (1-look-ahead) and R_{new} for states s''' succeeding s'' (2-look-ahead). These additional rules are not sufficient but necessary for equation 3.12 to be true. Hence they are heuristics that can exclude some states for which it is guaranteed that they do not have any consistent successors when looking one respectively two levels ahead. Concretely, these rules are

$$\begin{aligned} R_{\text{in}}(s,u,v) \Leftrightarrow\ & |\text{Succ}(G_{\text{P}},u) \cap T_{\text{P}}^{\text{in}}(s)| \geq |\text{Succ}(G_{\text{T}},v) \cap T_{\text{T}}^{\text{in}}(s)| \wedge \\ & |\text{Pred}(G_{\text{P}},u) \cap T_{\text{P}}^{\text{in}}(s)| \geq |\text{Pred}(G_{\text{T}},v) \cap T_{\text{T}}^{\text{in}}(s)| \end{aligned} \tag{3.15}$$

$$\begin{aligned} R_{\text{out}}(s,u,v) \Leftrightarrow\ & |\text{Succ}(G_{\text{P}},u) \cap T_{\text{P}}^{\text{out}}(s)| \geq |\text{Succ}(G_{\text{T}},v) \cap T_{\text{T}}^{\text{out}}(s)| \wedge \\ & |\text{Pred}(G_{\text{P}},u) \cap T_{\text{P}}^{\text{out}}(s)| \geq |\text{Pred}(G_{\text{T}},v) \cap T_{\text{T}}^{\text{out}}(s)| \end{aligned} \tag{3.16}$$

$$\begin{aligned} R_{\text{new}}(s,u,v) \Leftrightarrow\ & |\tilde{V}_{\text{P}}(s) \cap \text{Pred}(G_{\text{P}},u)| \geq |\tilde{V}_{\text{T}}(s) \cap \text{Pred}(G_{\text{P}},v)| \wedge \\ & |\tilde{V}_{\text{P}}(s) \cap \text{Succ}(G_{\text{P}},u)| \geq |\tilde{V}_{\text{T}}(s) \cap \text{Succ}(G_{\text{P}},v)| \end{aligned} \tag{3.17}$$

With these rules we define the syntactic feasibility rule as

$$F_{\text{syn}}(s,u,v) = R_{\text{Pred}} \wedge R_{\text{Succ}} \wedge R_{\text{in}} \wedge R_{\text{out}} \wedge R_{\text{new}}. \tag{3.18}$$

The semantic feasibility ensures consistency between vertex / edge labels. Such semantic feasibility is case specific. For example, the labels can be symbolic and require exact equivalence, or they can be numeric and only require similarity within some tolerance. For VF2 [11] define a compatibility relation $\approx$ between node / vertex labels that needs to be fulfilled for a mapping to be consistent. With this compatibility relation we express the semantic feasibility rule $F_{\text{sem}}(s,u,v)$ for adding $(u,v) \in V_{\text{P}} \times V_{\text{T}}$ to $M(s)$ as the requirement of the vertex labels being compatible, i.e. $u \approx v$, as well as the edges incident to u being compatible to the edges incident to v. Note that the existence of the edges is ensured by the syntactic feasibility, so for

semantic feasibility it is sufficient to ensure compatibility of the labels. Formally this combination of the compatibility of u, v as well as of their incident edges can be expressed as

$$\begin{aligned} F_{\text{sem}}(s, u, v) \Leftrightarrow \quad & u \approx v \\ & \wedge \forall (u', v') \in M(s) : (u, u') \in E_{\text{P}} \rightarrow (u, u') \approx (v, v') \\ & \wedge \forall (u', v') \in M(s) : (u', u) \in E_{\text{P}} \rightarrow (u', u) \approx (v', v). \end{aligned} \tag{3.19}$$

The feasibility rules introduced by Cordella et al. for VF2 [12, 11] are in this form already fairly set-centric. However, Cordella et al. explicitly use vectors for their data structures.

3.2 Data-Structures for Sets

In this section, we will introduce data structures for sets. First we introduce fundamental data structures in 3.2.1. Building on these fundamental data structures we will present roaring bitmaps in 3.2.2 and robin hood hashmaps in 3.2.3. These will be used in chapter 4 for developing a set-centric implementation for the subgraph isomorphism.

3.2.1 Arrays, Dense Bitmaps

Arrays

An array is a collection of elements such that the ith element is stored at the ith position. For integers, this typically means that each integer is encoded in base 2 with a fixed length. Arrays can be sorted or unsorted. Sorted arrays allow fast look-up through binary search.

Dense Bitmaps

A dense bitmap represents a domain on size d with d bits. The ith bit is set to one if and only if the ith element of the domain is present in the data structure. For integers, this typically means that an interval $[a, b]$ is represented by setting the ith bit to 1 if and only if the a+ith integer is present.

3.2.2 Roaring Bitmaps

Roaring Bitmaps were introduced by [8] to provide a data structure for integers with both a relatively small memory footprint, as well as fast look-up, insertion and removal times. This is achieved by combining several more fundamental data structures.

Arrays are efficient when the elements are very sparse. In contrast, dense bitmaps are efficient when the elements are dense. Roaring bitmaps combine these properties by partitioning the domain of possible elements into chunks. Each chunk is represented by a container. A container can assume one of several fundamental data structures.

In the original version by Chambi et al. [8] containers can either be arrays or dense bitmaps. In the follow-up paper by Lemire et al. [24] containers can assume a third type: run containers. Run containers represent the elements by containing several tuples (a, l). Each tuple indicates that the integers in the interval $[a, a + l]$ are present in the container. We are using the latest version of roaring containers by Lemire et al. [23]. This version improved on the previous ones by using SIMD for vectorized instructions of up to 256-bit vectors.

3.2.3 Robin Hood Hashmap

Another library for a set data structure that we are using is based on robin hood hashing. Robin hood hashing was introduced by Celis et al. [7] and has attracted widespread analysis [13, 28]. The basic principle is to implement a hash function that, in case of collision, allows the element that has travelled the furthest away to stay in its position.

This simple hashing scheme has an exponentially faster search time than open addressing hashing with a first come first served collision strategy [13] and low variance, even when alternating insertions and deletions [28].

The library we were provided with uses robin hood hashing to store elements of a set data structure. When calculating the intersection of two sets, it iterates through the elements in the set with smaller cardinality and tests for their membership in the other set.

Chapter 4

Set-Centric Subgraph Isomorphism

In this chapter, we will develop a parallel baseline and set-centric implementation. Both are based on a reference implementation of VF2 as described in section 3.1 that was made available to us. The set-centric variant uses the same mechanics as the parallel baselines to achieve parallelism but draws on section 3.2 for set-centric data structures and operations. For simplicity, the implementation considers all edges as undirected and labels are not considered.

4.1 Parallel Baseline

For parallelizing VF2 the Open Multi-Processing (OMP) 4.5 framework in C++ developed by [31] is used. The implementation of the high-level VF2 function is displayed in figure 4.1. In this figure key components of the algorithm are abstracted by functions, such as the generation of candidate pairs in line 11, the checking of feasibility in line 18, and the adding of a pair to a state in line 21. The high-level VF2 function can be parallelized by using OMP without changing the lower-level functions.

OMP is a framework which allows adding preprocessing directives called OMP pragmas to C++ code. These preprocessing directives instruct the compiler to generate parallel cause by using the parallelizing abstractions provided by OMP. One such abstraction is OMP tasks. An OMP task is defined by [31] as "A specific instance of executable code and its data environment", which is "generated when a thread encounters a task, taskloop, [or one of several other constructs]". It is a collection of code and data that a thread can work on, while other threads work on different tasks.

In OMP, we have to specify that we want to use multiple threads before creating tasks that these threads can work on. This is done using the *omp parallel* pragma. As can be seen in figure 4.2 we do this by surrounding the

```
void VF2::solve(const CSRGraph &target, const CSRGraph &pattern,
                State &state, Result &result)
{
    // if M1 is already full, save result
    if (state.patternMappedNodes.size() == state.patternVertexCount)
    {
        state.saveToResult(result);
        return;
    }

    std::vector<std::pair<NodeId, NodeId>> P = genCandidates(state);

    for (auto p = P.begin(); p != P.end(); p++)
    {
        const NodeId u = p->first;
        const NodeId v = p->second;

        if (feasible(state, target, pattern, u, v))
        {
            {
                state.addNewPair(u, v);
                solve(target, pattern, state, result);
            }
        }
    }
}
```

Figure 4.1: High-level VF2 function

initial call to the recursive VF2 solve function with a *omp parallel* pragma. To execute this initial call to the VF2 solve function not once per thread but only one single time, we have to add a *omp single* pragma. With this combination of *omp parallel* and *omp single*, we can create tasks which are distributed among threads. Note that both *omp parallel* and *omp single* act as implicit synchronization barriers for the threads so that all tasks get completed before any thread is terminated. Furthermore we enable nested parallelism in line 1 of figure 4.2 in order for tasks to be able to create tasks.

```
1 omp_set_nested(true);
2 #pragma omp parallel
3 {
4     #pragma omp single
5     {
6         solve(Target, Pattern, state);
7     }
8 }
```

Figure 4.2: Initial call to VF2 with omp pragmas

Using multiple threads means that different threads could concurrently find a valid isomorphism and execute the code in line 7 of figure 4.1 with a race condition. We therefore replace the saving of results with the code in figure 4.3. Thereby the results are saved in a separate data structure per thread and combined at the end of the execution.

```
1  // if M1 is already full, save result
2  if (state.patternMappedNodes.size() == state.patternVertexCount)
3  {
4      state.saveToResult(results[omp_get_thread_num()]);
5      return;
6  }
```

Figure 4.3: Saving the number of isomorphisms with multiple threads

One possibility for parallelizing the high level VF2 function in figure 4.1 with the OMP abstraction of tasks is by creating one task for each feasible state. This is done in figure 4.4 by adding an *omp task* pragma around the creation of a new state and the recursive call for that new state which is in lines 21-22 of figure 4.1. As parameters of the new task we specify how the data environment of this task is created. First we specify *default (none)* in line 3 of figure 4.4 to make explicit that only data which is mentioned in the other parameters is added to the data environment. Next, we specify in line 4 which data is shared between the previous data environment and the created task's new data environment. We share *std::cout* to be able to write to stdout from all tasks. Furthermore we share *pattern* and *target* which are constant data structures representing G_P and G_T. Lastly, we specify *first-private(state, n, m)*. This instructs copies of *state*, *n*, *m* to be private variables of the task's data environment. With these shared and private variables we can finally add the candidate pair in line 7 of 4.4 and recursively call the VF2 solve function in line 8.

```
1  if (feasible(state, target, pattern, n, m))
2      {
3      #pragma omp task default(none)
4                       shared(std::cout, pattern, target)
5                       firstprivate(state, n, m)
6      {
7          state.addNewPair(n, m);
8          solve(target, pattern, state);
9      }
10 }
```

Figure 4.4: Parallelizing over feasible states

An alternative to parallelizing over feasible states as displayed in figure 4.4 is to parallelize over candidate pairs. To this end the *omp taskloop* pragma can be used to parallelize the for loop in line 13 of figure 4.1. This variant is displayed in figure 4.5. Similar to the previous variant in figure 4.4, in figure 4.5 lines 1-3 we specify the creation of the data environment. For the OMP taskloop there is the additional parameter *grainsize* in line 4 of figure 4.5, which specifies how many iterations of the for-loop should be combined into one task.

```
1 #pragma omp taskloop default(none)
2                      shared(P, std::cout, pattern, target)
3                      firstprivate(state)
4                      grainsize(taskloop_grainsize)
5 for (auto p = P.begin(); p != P.end(); p++)
```

Figure 4.5: Parallelizing over candidate pairs

The variants of parallelizing over feasible states as displayed in figure 4.4 or over candidate pairs as displayed in figure 4.5 can be chosen alternatively to each other, or both can be implemented in combination. Hence in chapter 5 all possibilities will be tested.

4.2 Set-Centric Implementation

As we have seen in section 3.1 the feasibility constraints of VF2 have a natural expression with set operations. Nevertheless, the data structures being described by [11] in the paper and those used in their implementation are not sets. Since most of the computational effort is spent on evaluating feasibility constraints, the focus of our set-centric implementation is to use set operations for the checking of feasibility constraints.

We will now discuss possible implementations of the syntactic feasibility constraints one by one.

Core rule: R_{Pred} and R_{Succ}

Since we're considering all edges to be undirected, R_{Pred} and R_{Succ} can be combined to

$$\begin{aligned} &R_{\text{Core}}(s,u,v) \Leftrightarrow \\ &\quad (\forall u' \in M_{\text{P}}(s) \cap N(G_{\text{P}},u)\ \exists v' \in N(G_{\text{T}},v) : (u',v') \in M(s)) \wedge \\ &\quad (\forall v' \in M_{\text{T}}(s) \cap N(G_{\text{T}},v)\ \exists u' \in N(G_{\text{P}},u) : (u',v') \in M(s)), \end{aligned} \tag{4.1}$$

where $N(G,u)$ is the neighborhood of u in G. Naming this rule R_{core} is taken from [6] to express that it is the main rule in the sense that it implies the state obtained by adding (u,v) is consistent.

A set-centric implementation of R_{Core} is displayed in figure 4.6. In order to evaluate the rule efficiently for a given input s, u, v, we first compute $M_{\text{P}}(s) \cap N(G_{\text{P}}, u)$ and $M_{\text{T}}(s) \cap N(G_{\text{T}}, v)$ in lines 4-7. We can then iterate over the vertices in these intersections and check for their membership in $N(G_{\text{P}}, u)$ respectively $N(G_{\text{T}}, u)$. As discussed in section 3.2, how efficiently the intersection and the membership are computed depends on which set representation is used.

```
1  bool checkCoreRule(const State &state, const Graph &G1,
2                     const Graph &G2, const NodeId n, const NodeId m)
3  {
4      auto targetMappedNeighbors = GT.out_neigh(n).intersect(
5                                     state.targetMappedNodes);
6      auto patternMappedNeighbors = G2.out_neigh(m).intersect(
7                                     state.patternMappedNodes);
8
9      for (auto targetMappedNeighbor : targetMappedNeighbors) {
10         auto patterMappedNode = state.mappingTargetToPattern
11                                     [targetMappedNeighbor];
12         if (! patternMappedNeighbors.contains(patterMappedNode)) {
13             return false;
14         }
15     }
16
17     for (auto patternMappedNeighbor: patternMappedNeighbors) {
18         auto targetMappedNode = state.mappingPatternToTarget
19                                     [patternMappedNeighbor];
20         if (! targetMappedNeighbors.contains(targetMappedNode)) {
21             return false;
22         }
23     }
24     return true;
25 }
```

Figure 4.6: Core rule variant 1

A variation of the R_{Core} rule implementation in figure 4.6 is displayed in figure 4.7. It is based on the following theorem.

Theorem 4.1 *Under the assumption of corollary 3.5, the $R_{\text{Core}}(s, u, v)$ rule*

$$\begin{aligned} &(\forall u' \in M_{\text{P}}(s) \cap N(G_{\text{P}}, u)\ \exists v' \in N(G_{\text{T}}, v) : (u', v') \in M(s)) \\ \wedge\ &(\forall v' \in M_{\text{T}}(s) \cap N(G_{\text{T}}, v)\ \exists u' \in N(G_{\text{P}}, u) : (u', v') \in M(s)) \end{aligned} \tag{4.2}$$

is equivalent to the expression

$$\begin{aligned} &|\{v' \mid \exists u' : u' \in M_{\text{P}}(s) \cap N(G_{\text{P}}, u) \wedge (u', v') \in M(s)\} \cap N(G_{\text{T}}, v)| \\ = \ &|M_{\text{P}}(s) \cap N(G_{\text{P}}, u)| \\ = \ &|M_{\text{T}}(s) \cap N(G_{\text{T}}, v)|. \end{aligned} \tag{4.3}$$

Proof We will first proof that equation 4.2 implies equation 4.3 ($\Rightarrow$), and then that equation 4.3 implies equation 4.2 ($\Leftarrow$).

$\Rightarrow$

Suppose equation 4.2 holds. From the first term in the conjunction of equation 4.2 we know that every element in $M_{\text{P}}(s) \cap N(G_{\text{P}}, u)$ is mapped to an element in $N(G_{\text{T}}, v)$. By corollary 3.5 this mapping is injective and hence every element in $M_{\text{P}}(s) \cap N(G_{\text{P}}, u)$ is mapped to a unique element in $N(G_{\text{T}}, v)$, i.e.

$$\begin{aligned} &|\{v' \mid \exists u' : u' \in M_{\text{P}}(s) \cap N(G_{\text{P}}, u) \wedge (u', v') \in M(s)\} \cap N(G_{\text{T}}, v)| \\ = \ &|M_{\text{P}}(s) \cap N(G_{\text{P}}, u)|. \end{aligned} \tag{4.4}$$

We have already established that every element in $M_{\text{P}}(s) \cap N(G_{\text{P}}, u)$ is injectively mapped to an element in $N(G_{\text{T}}, v)$. Since all mapped elements in $N(G_{\text{T}}, v)$ are in $M_{\text{T}}(s) \cap N(G_{\text{T}}, v)$, it follows that every element in $M_{\text{P}}(s) \cap N(G_{\text{P}}, u)$ is injectively mapped to an element in $M_{\text{T}}(s) \cap N(G_{\text{T}}, v)$, and hence

$$\begin{aligned} &|M_{\text{P}}(s) \cap N(G_{\text{P}}, u)| \\ \leq &|M_{\text{T}}(s) \cap N(G_{\text{T}}, v)|. \end{aligned} \tag{4.5}$$

By the second term in the conjunction of equation 4.2 every element in $M_{\text{T}}(s) \cap N(G_{\text{T}}, v)$ is mapped to an element in $N(G_{\text{P}}, u)$. From corollary 3.5 we know that this mapping in injective. Since furthermore every mapped element in $N(G_{\text{P}}, u)$ is in $M_{\text{P}}(s) \cap N(G_{\text{P}}, u)$, it follows that every element in $M_{\text{T}}(s) \cap N(G_{\text{T}}, v)$ in injectively mapped to an element in $M_{\text{T}}(s) \cap N(G_{\text{T}}, v)$, and hence

$$\begin{aligned} &|M_{\text{T}}(s) \cap N(G_{\text{T}}, v)| \\ \leq &|M_{\text{P}}(s) \cap N(G_{\text{P}}, u)|. \end{aligned} \tag{4.6}$$

Combining equations 4.5 and 4.6 gives

$$|M_{\text{P}}(s) \cap N(G_{\text{P}}, u)| = |M_{\text{T}}(s) \cap N(G_{\text{T}}, v)|. \tag{4.7}$$

Equation 4.7 combined with equation 4.4 gives equation 4.3.

$\Leftarrow$

Suppose equation 4.3 holds. Assume for contradiction that equation 4.2 is false.

Since equation 4.2 is false at least one of the terms in its conjunction is false, which implies that at least one of the following holds: 1) There exists a $u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)$ for which there is no $v' \in N(G_{\mathrm{T}}, v)$ with $(u', v') \in M(s)$, or 2) there exists a $v' \in M_{\mathrm{T}}(s) \cap N(G_{\mathrm{T}}, v)$ for which there is no $u' \in N(G_{\mathrm{P}}, u)$ with $(u', v') \in M(s)$.

Case 1: $\exists u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u) \nexists v' \in N(G_{\mathrm{T}}, v) : (u', v') \in M(s)$
By the assumption of the case there exists a $u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)$ for which there is no $v' \in N(G_{\mathrm{T}}, v)$ with $(u', v') \in M(s)$. Since this $u' \in M_{\mathrm{P}}(s)$, u' is mapped, i.e. there exists a v' with $(u', v') \in M(s)$. By the assumption of the case it holds for this v' that $v' \notin N(G_{\mathrm{T}}, v)$. This can be rewritten as

$$\exists v' \in \{v' \mid \exists u' : u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u) \wedge (u', v') \in M(s)\} : v' \notin N(G_{\mathrm{T}}, v) \tag{4.8}$$

which implies that

$$\begin{aligned} &|\{v' \mid \exists u' : u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u) \wedge (u', v') \in M(s)\} \cap N(G_{\mathrm{T}}, v)| \\ &< |\{v' \mid \exists u' : u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u) \wedge (u', v') \in M(s)\}|. \end{aligned} \tag{4.9}$$

On the other hand, since by corollary 3.5 $M(s)$ is an injective mapping, it holds for any set S that

$$|\{v' \mid \exists u' : u' \in S \wedge (u', v') \in M(s)\}| \leq |S|. \tag{4.10}$$

By instantiating $S = M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)$ it follows that

$$\begin{aligned} &|\{v' \mid \exists u' : u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u) \wedge (u', v') \in M(s)\}| \\ &\leq |M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)|. \end{aligned} \tag{4.11}$$

Combining equations 4.9 and 4.11 gives

$$\begin{aligned} &|\{v' \mid \exists u' : u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u) \wedge (u', v') \in M(s)\} \cap N(G_{\mathrm{T}}, v)| \\ &< |M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)|, \end{aligned} \tag{4.12}$$

contradicting equation 4.3.

Case 2: $\exists v' \in M_{\mathrm{T}}(s) \cap N(G_{\mathrm{T}}, v) \nexists u' \in N(G_{\mathrm{P}}, u) : (u', v') \in M(s)$
For every $u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)$ there exists $v' \in M_{\mathrm{T}}(s)$ with $(u', v') \in M$. Since by corollary 3.5 this mapping is injective, and by the case assumption there exists $v' \in M_{\mathrm{T}}(s) \cap N(G_{\mathrm{T}}, v) \nexists u' \in N(G_{\mathrm{P}}, u) : (u', v') \in M(s)$, and by equation 4.3 it holds that $|M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)| = |M_{\mathrm{T}}(s) \cap N(G_{\mathrm{T}}, v)|$, it follows that there exists $u' \in M_{\mathrm{P}}(s) \cap N(G_{\mathrm{P}}, u)$ for which there exists $v' \in M_{\mathrm{T}}(s) \setminus$

$N(G_T, v)$ with $(u', v') \in M(s)$. Since by corollary 3.5 $M(s)$ is injective this is equivalent to $\exists v' \in M_T(s) \cap N(G_T, v) \nexists u' \in N(G_P, u) : (u', v') \in M(s)$. We have already shown in case 1 that this implies equation 4.12, contradicting equation 4.3. □

We have proven that under the assumption of corollary 3.5 the core rule of the feasibility constraints from equation 4.2 can be rewritten to the equivalent equation 4.3. This core rule is both necessary and sufficient for ensuring that the obtained mapping is a consistent subgraph isomorphism. In other words, rewriting the core rule has given us a new set-centric formulation of subgraph isomorphism.

Using theorem 4.1 we obtain a variant of the code in figure 4.6 with the equivalent feasibility check from the theorem. The resulting implementation is displayed in figure 4.7.

```
bool checkCoreRule(const State &state, const Graph &G1,
                   const Graph &G2, const NodeId n, const NodeId m)
{
    auto targetMappedNeighbors = GT.neigh(n).intersect(
                                    state.targetMappedNodes);
    auto patternMappedNeighbors = G2.neigh(m).intersect(
                                    state.patternMappedNodes);

    Set vDash;
    for (auto targetMappedNeighbor : targetMappedNeighbors) {
        vDash.add(state.mappingTargetToPattern
                        [targetMappedNeighbor]);
    }

    int intersect_count = vDash.intersect_count(
                                    patternMappedNeighbors);

    if (intersect_count != targetMappedNeighbors.cardinality()) {
        return false;
    }

    if (intersect_count != patternMappedNeighbors.cardinality()) {
        return false;
    }

    return true;
}
```


Figure 4.7: Core rule variant 2

1-look-ahead rule: R_{In} and R_{Out}

Using the simplification of treating all edges as undirected edges, the rules R_{in} and R_{out} can be implemented as displayed in figure 4.8.

```
bool checkTermRule(const State &state, const Graph &G1,
                   const Graph &G2, const NodeId n, const NodeId m)
{
    uint count1 = G1.neigh(n)
                    .intersect_count(state.targetOutFrontier);
    uint count2 = G2.neigh(m)
                    .intersect_count(state.patternOutFrontier);
    return count1 >= count2;
}
```

Figure 4.8: 1-look-ahead rule

2-look-ahead rule: R_{new}

Using the simplification of treating all edges as undirected edges, the rule R_{new} can be implemented as displayed in figure 4.9.

```
bool checkNewRule(const State &state, const Graph &G1,
                  const Graph &G2, const NodeId n, const NodeId m)
{
    uint count1 = G1.neigh(n).difference(
                        state.targetMappedNodes
                            .union_with(state.targetOutFrontier)
                        ).cardinality();
    uint count2 = G2.neigh(m).difference(
                       state.patternMappedNodes
                           .union_with(state.patternOutFrontier)
                       ).cardinality();
    return count1 >= count2;
}
```

Figure 4.9: 2-look-ahead rule

Chapter 5

Experiments

In this section we will present our experimental results. To this end, we will first describe our experimental setup in section 5.1 followed by our results in section 5.2.

5.1 Experimental Setup

Throughout the conduction and presentation of our results we strive to adhere to the guidelines recommended by Hoefler et al. for scientific benchmarking of parallel computing systems [18]. To this end we first present our hardware and software setups, followed by some remark on our measurement and reporting procedure. Later we present the pattern and target graphs used for our experiments.

5.1.1 Hardware and software setup

Our experiments were each run separately on a single node that was otherwise load-free. The node has two 64-core AMD EPYC 7742 processors with disabled hyper-threading and 504 GiB DDR4-3200 RAM. The storage and filesystem setup does not impact our results. All codes were compiled with gcc version 9.2.0 and OpenMP version 4.5.

5.1.2 Measurement and reporting procedure

As recommended by Hoefler et al. [18] we exclude the first measurement per run in order to allow for a warmup period. We also follow their advice to use the whole node for our result, i.e. we conduct all experiments with up to 128 threads (the number of cores available) independent of how well the code scales to this number of threads. Finally, we implement their guideline of reporting confidence intervals where appropriate, though it is still left open to test the implicit normal distribution assumption.

5.1.3 Pattern and target graphs

Graph	Nodes	Edges	Density	Max Deg	Min Deg	Mean Deg
bio-celegans	453	2′025	0.020	237	1	8
bio-celegansneural	297	2′346	0.053	134	1	15
bio-diseasome	516	1′188	0.009	50	1	4
bio-yeast	1′458	1′948	0.002	56	1	2
dimacs-brock200-2	200	9′876	0.496	114	78	98
dimacs-c-fat200-1	200	1′534	0.077	17	14	15
dimacs-c-fat200-2	200	3′235	0.163	34	32	32
dimacs-c-fat200-5	200	8′473	0.426	86	83	84
dimacs-c-fat500-1	500	4′459	0.036	20	17	17
dimacs-c-fat500-2	500	9′139	0.073	38	35	36
dimacs-c125-9	125	6′963	0.898	119	102	111
misc-erdos992	6′100	7′515	0.0004	61	0	2
delaunay-n10	1′024	3′056	0.006	12	3	5
dimacs-hamming6-2	64	1′824	0.905	57	57	57
dimacs-johnson16-2-4	120	5′460	0.765	91	91	91
dimacs-keller4	171	9′435	0.649	124	102	110
dimacs-mann-a9	45	918	0.927	41	40	40
san200-0-7-1	200	13′930	0.700	155	125	139

Figure 5.1: Target graphs

For testing our algorithm variants we use different pattern and target graphs. The generation of input graphs for subgraph isomorphism is a research area in its own [36, 27, 26]. Our target graphs cover a wide range of real-world and synthetic data. They are sourced from the Network Data Repository [35]. An overview of their characteristics is displayed in figure 5.1.

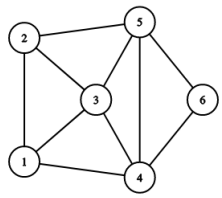


Figure 5.2: The graph we refer to as *Graph(n=6, m=10)*

To keep the required computation time for the experiments in a feasible range we chose to only use two different pattern graphs. One of them is a triangle and the other one drawn uniformly at random among all undirected graphs with 6 nodes and 10 vertices. This randomly drawn graph is displayed in figure 5.2 and will be referred to as *Graph(n=6, m=10)*.

5.2 Results

In this section, we will evaluate the performance of VF2 variants we have developed. Since the search space is so vast, we will analyze different aspects of the algorithm independently.

5.2.1 Performance distribution over all variants

Before we compare specific variations of VF2, we take a look at the performance distribution over all VF2 variants to see how large the range of performance results is.

In figure 5.3 we see the performance of all tested versions per combination of pattern and target graph when using 1 thread in 5.3a and 16 threads in 5.3b. For the 5.3b only parallel variants of VF2 have been used.

We note that with 1 thread the shortest and the longest execution time per pattern and graph combination are typically a factor of 5-10x apart. With 16 threads we observe that the longest and fastest execution time per pattern and target combination are typically a factor of circa 10x apart, with some being as far as a factor of 100x apart. From this high-level analysis, we conclude that the developed VF2 variants vary widely in performance. In the following sections we will take a closer look at the underlying aspects of this variability.

5.2.2 Performance comparison of specific VF2 variants

Sequential vs. 1-Thread Parallel

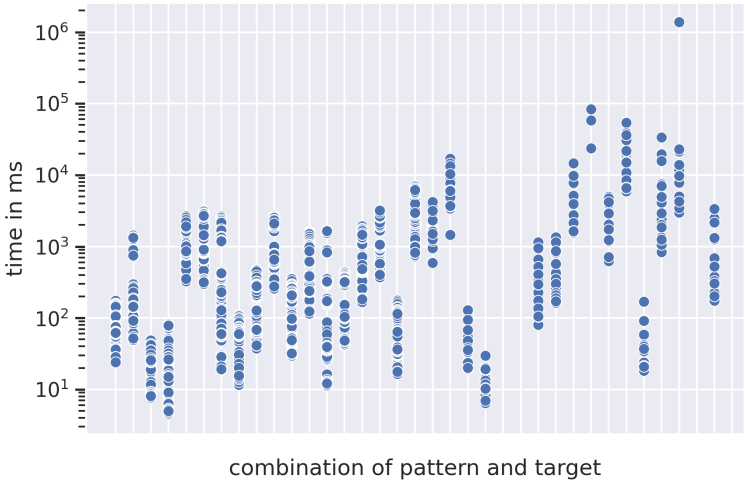
As described in section 4.1 we have used OMP to parallelize VF2. To test whether this parallelization includes thread management which may deteriorate performance even when using only 1 thread, we compare the parallel baseline against a sequential baseline. The sequential baseline is equivalent to the parallel baseline without OMP primitives. The result can be seen in figure 5.4.

We note that with 1 thread, the parallel baseline's performance is very similar compared to the performance of the sequential baseline, except for some pattern and target combinations for which the runtime is low. This holds for all variants of OMP parallelization introduced in section 4.1, namely using the OMP task primitive, taskloop primitive, or both simultaneously.

Parallelization options OMP task vs. OMP taskloop

In this part we will compare the parallelization variants introduced in section 4.1. In particular, we will analyze the performance of using the primitives OMP task, OMP taskloop, or both simultaneously for parallelization.

(a) 1 thread



(b) 16 threads

Figure 5.3: Distribution of average execution times per combination of pattern and target graphs across (a) variants of VF2 using 1 thread and (b) across parallel variants using 16 threads

We will also investigate the influence of different grainsizes (how many loop iterations to combine into one task) when parallelizing over a for-loop with OMP taskloop.

Since the search space would be too vast when combining the variability of OMP pragmas with other changes to VF2, we use only the parallel baseline in this comparison. That means we use the non-set-centric data structures

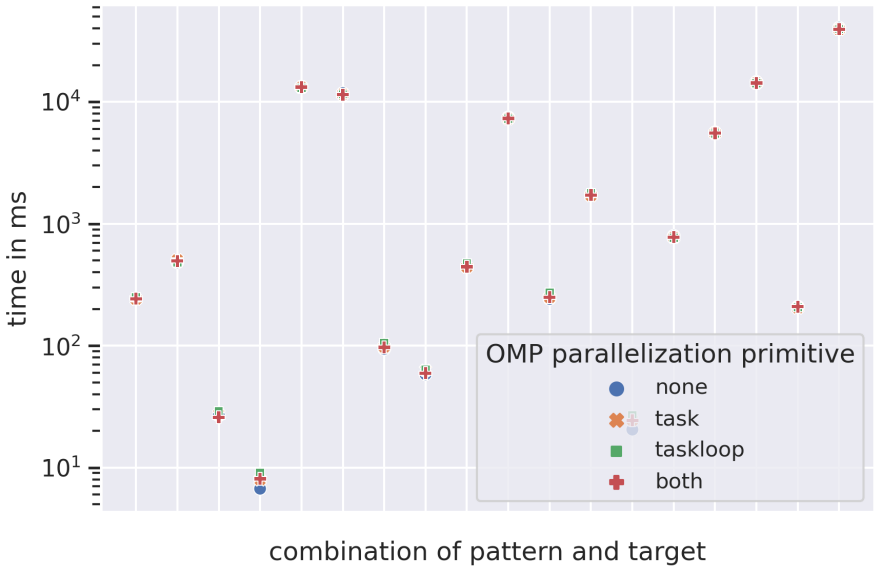


Figure 5.4: distribution of average execution times per combination of pattern and target graphs across non-set-centric variants of VF2 with 1 threads

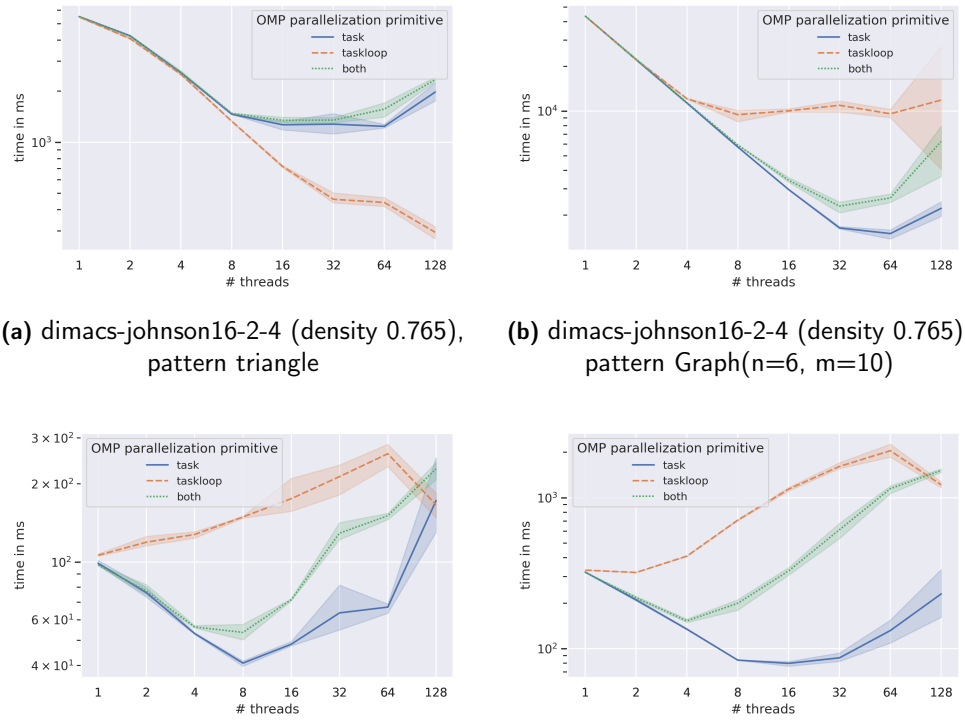
with all look-ahead rules as described by [11].

The results for some input and target graphs can be seen in figure 5.5. For the pattern Graph(n=6, m=10) in both figures 5.5b and 5.5d using the OMP task primitive performs best. In fact, with the pattern Graph(n=6, m=10) the OMP task primitive has outperformed OMP taskloop and combining both primitives with any target graph we have tested.

For the triangle target graph, the results are more mixed. In some runs such as 5.5a OMP taskloop performs best, in others such as 5.5c OMP task performs best. However, in no experiment using both task and taskloop simultaneously performed best.

We also note that the overall performance scaling varies widely across the experiments. For many algorithm variants and pattern/target graph combinations, speedups linear with the number of cores are observed when scaling wth 1-32 threads, such as in figure 5.5b for the OMP task parallelization. However, for target graphs where the sequential algorithm with the triangle pattern has an execution time of less than 200ms we see bad scaling behaviour, sometimes even when increasing runtime when scaling from 1 to 2 threads. Figure 5.5d is an example of this scaling behavior. Therefore, we exclude target graphs with such short execution times from further experiments.

Recall from section 4.1 that the taskloop has a *grainsize* parameter that con-



(a) dimacs-johnson16-2-4 (density 0.765), pattern triangle

(b) dimacs-johnson16-2-4 (density 0.765), pattern Graph(n=6, m=10)

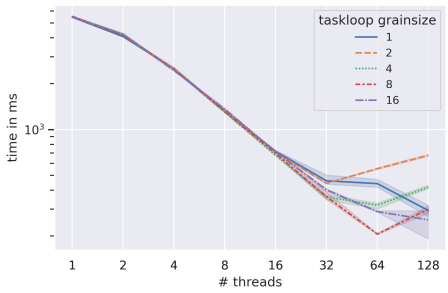
(c) misc-erdos992 (density 0.0004), pattern triangle

(d) misc-erdos992 (density 0.0004), pattern Graph(n=6, m=10)

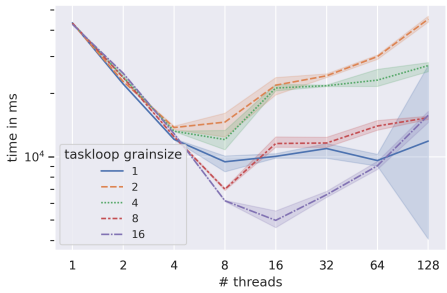
Figure 5.5: Execution time with 95% confidence interval across OMP parallelization options for some pattern and target graph combinations

trols how many loop-iterations are combined to one task, with the default being a grainsize of 1. To get more insight into the parallelization behaviour we have conducted a series of experiments with the parallel baseline using the OMP taskloop primitive and varying grainsize. The results for some combinations of pattern and target graphs can be seen in figure 5.6.

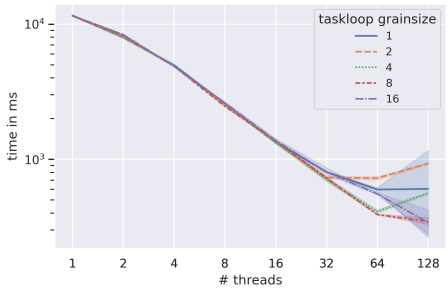
We tested grainsizes 1, 2, 4, 8, and 16. We observe that among these the grainsize 8 or 16 is optimal for most tested combinations of pattern and target graph (such as 5.6a, 5.6b and 5.6c). The only exception among all tested combinations is the combination of pattern Graph(n=6, m=10) and target graph bio-celegans displayed in figure 5.6d. Here grainsize 1 follows best, followed by 16, 2, 8, 4. Note that these results are averaged over 3 runs with low variance, the 95% confidence interval is displayed in the figures. This order persisted also after re-running the experiment.



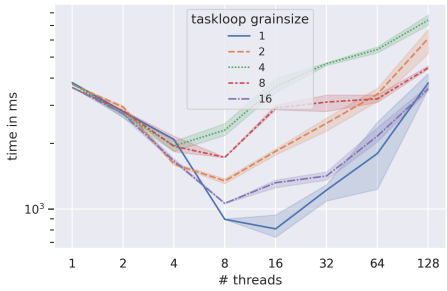
(a) dimacs-johnson16-2-4 (density 0.765), pattern triangle



(b) dimacs-johnson16-2-4 (density 0.765), pattern Graph(n=6, m=10)



(c) dimacs-c125-9 (density 0.898), pattern triangle



(d) bio-celegans (density 0.020), pattern Graph(n=6, m=10)

Figure 5.6: Execution time with 95% confidence interval across the grainsize for OMP taskloops for some pattern and target graph combinations

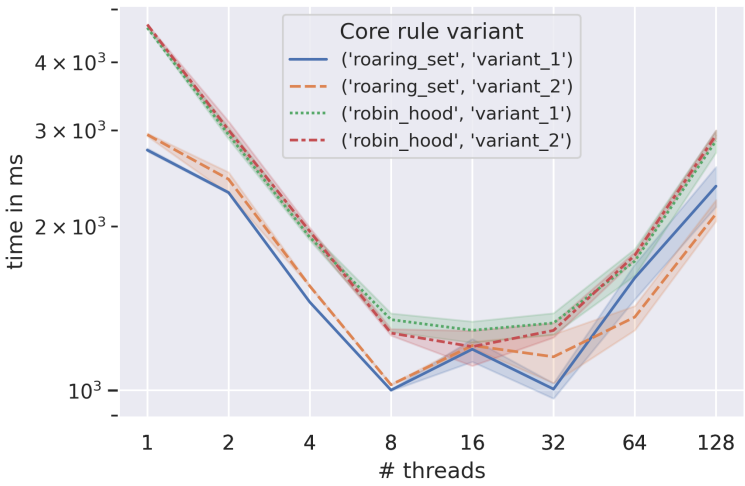
Set-Centric variants vs. parallel baseline

We will now study how the non-set-based parallel baseline from 4.1 compares to the set-centric variants from 4.2. For the set-centric variants, we have different parallelization options, variants of feasibility rules, and set-types. We have already studied the parallelization options for the non-set-based parallel baseline in the previous section. To confine the search space, we limit the parallelization option for the set-centric variants to the OMP task primitive and focus on varying the feasibility rules and the set-type.

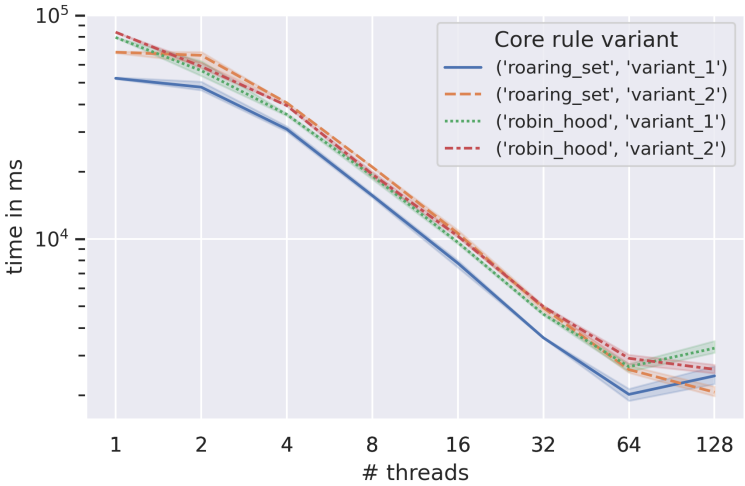
For the choice of the feasibility rules we have the following options: With theorem 4.1 we have proven that the core rule can be rewritten and implemented as either variant 1 displayed in figure 4.6 or variant 2 displayed in figure 4.7.

We start by assessing variant 1 of the core rule vs. variant 2 of the core rule since we expect this comparison to be independent of other variability. The result can be seen in figure 5.7.

From the experiments in figure 5.7 we can see that with both roaring bitmaps



(a) dimacs-johnson16-2-4 (density 0.765), pattern triangle

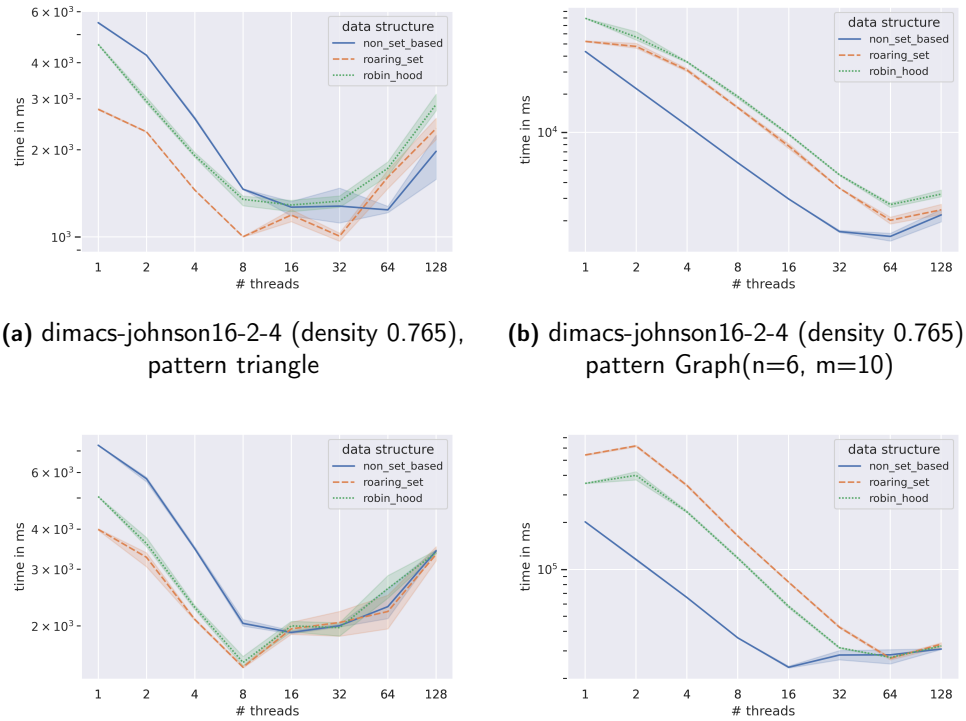


(b) dimacs-johnson16-2-4 (density 0.765), pattern Graph(n=6, m=10)

Figure 5.7: Execution time with 95% confidence interval across variant 1 and variant 2 of the core rule for some pattern and target graph combinations

and robin hood hashmaps the variant 1 of the core rule either outperforms the variant 2 (5.7b) or performs not significantly different than variant 2 (5.7a). This holds beyond the displayed combinations of pattern and target graph for all experiments we have conducted. We are hence using variant 1 of the core rule in the following experiments.

Finally, we compare the set-based variants against the parallel baselines. Since our experiments have shown that variant 2 of the set-centric core rule outperforms variant 1, we will only use variant 1 for these experiments. With the experiments on OMP parallelization options having been inconclusive but the choice of parallelization option presumably not influencing this comparison, we decide to use parallelization with the OMP task primitive. The results can be seen in figure 5.8.



(a) dimacs-johnson16-2-4 (density 0.765), pattern triangle

(b) dimacs-johnson16-2-4 (density 0.765), pattern Graph(n=6, m=10)

(c) dimacs-c-fat200-5 (density 0.426), pattern triangle

(d) dimacs-c-fat200-5 (density 0.426), pattern Graph(n=6, m=10)

Figure 5.8: Execution time with 95% confidence interval across various data structures for some pattern and target graph combinations

We note that in figure 5.8a and 5.8c with pattern triangle the robin hood data structure performs best. In contrast, in 5.8b and 5.8d with the pattern Graph(n=6, m=10) the non-set-centric parallel baseline performs best. Indeed across all our experiments with the pattern Graph(n=6, m=10) the non-set-centric parallel baseline performs best. For many of the experiments with the triangle pattern robin hood performs best, in some cases the parallel baseline performs best, though never the roaring set version.

Chapter 6

Other Research Directions

Throughout this project, we worked on several other lines of research that are related to the idea of set-centric subgraph isomorphism. We present a brief overview of this work here.

6.1 Frequent Subgraph Mining

As stated in the introduction, subgraph isomorphism is a central subroutine to algorithms in frequent subgraph mining [14, 20]. For finding frequent subgraphs in a graph, pattern graphs in a set of candidate graphs are iteratively grown. At each level it is tested with subgraph isomorphism which candidates have as much support as a threshold parameter requires.

Since we have been working on improving subgraph isomorphism, questions arise on the impact of a more set-centric algorithm for subgraph isomorphism on frequent subgraph mining. In particular, we are interested in methods that go beyond regarding the subgraph isomorphism routine in frequent subgraph isomorphism as a black box. For example, exploiting intersections between the candidates in frequent subgraph mining for faster subgraph isomorphism detection seems worth pursuing.

6.2 Improving Sparse Bitmaps

There has been recent work on optimizing bitmaps for sparse data with sparse bitmaps. Han et al. [17] suggest a bitmap data structure where the domain is displayed into chunks. For each chunk the common bit-prefix of the elements in that chunk is used as an index. The indices of non-empty chunks are stored in one data structure, and the non-empty chunks are stored as dense bitmaps in a separate data structure. Each entry in the array of chunk indices also contains a pointer to the corresponding chunk.

This allows searching through the index data structure with binary search and only accessing the dense bitmaps of the corresponding chunk afterwards.

6.2.1 Relabeling Vertices

Consider such a data structure being used for graphs, e.g. to store a vertex neighborhood. We realized that both the access times and memory requirement of such a data structure depends strongly on the indices of the vertices: ideally many of the vertices present in the data structure are labelled with indices in the same chunk as other indices.

Therefore, sparse bitmaps can be optimized by relabeling the vertices with different indices. This optimization problem shares interesting parallel with the relabeling of vertices in subgraph isomorphism. Bonnici et al. [1] have explored heuristics for vertex relabeling on subgraph isomorphism algorithms. VF3 [5] also introduced a heuristic for determining a good total order relationship on the vertices, which is equivalent to the relabeling the vertices with a given total order relationship.

6.2.2 Multiple Levels of Indirection

Recall that Han et al. [17] use an array to store the indices of non-empty chunks, alongside with pointers to the corresponding chunks. When searching for an element, this allows for one level of indirection: Instead of searching through all chunks directly, they search if the chunk index corresponding to the searched element is present in the array of chunk indices. Only if this chunk index is found in the array of chunk indices, they search through the chunk that the index is associated with.

Storing indices of non-empty chunks in an array is what we refer to as *one level of indirection*. It is possible to have multiple levels of indirection: Instead of storing the chunk indices in array, these chunk indices themselves can be stored in a more complex data-structure.

One concrete way to achieve this that we have prototyped is to use a sparse bitmap to store the indices of a sparse bitmap. However, it became apparent that only special distributions of elements would profit from this, namely those that are sparse except for some chunks, where each such chunk itself is sparse except for dense subdomains. Unfortunately, we do not know of any real-world data having such properties.

Chapter 7

Conclusion

We have developed a multitude of VF2 variants for finding subgraph isomorphisms. These include different set-centric data structures, different set-centric feasibility rules, and variations of parallelism with OMP.

Our results show that our set-centric VF2 variants outperform the non-set-based parallel baseline on some inputs. In particular, we have developed a set-centric VF2 variant using robin hood hashmaps as primary data structure which outperforms both a set-centric VF2 variant using roaring bitmaps and the non-set-based parallel baseline, for some combinations of pattern and target graphs.

However, across all combinations of input graphs and algorithm variants which we have tested results are mixed. While a set-centric variant with robin hood hashmaps outperformed the non-set-based parallel baseline on some inputs, on many inputs the non-set-based parallel baseline performed better. We have also shown that the performance of parallelization primitives depends strongly on the specific input graphs.

We conclude that the choice of pattern and target graph heavily impacts how the analyzed algorithms perform relative to each other. Thus we propose as further research direction to develop a more powerful algorithm for subgraph isomorphism. This could include adaptive data structures similar to roaring bitmaps but with more variability and an adaptive parallelization scheme. The depth-first traversal of the state space employed by VF2 and other subgraph isomorphism algorithms would allow such an adaptive algorithm to sample trajectories through the state space and choose variants of parallelization schemes and data structures accordingly.

Bibliography

[1] Vincenzo Bonnici and Rosalba Giugno. On the variable ordering in subgraph isomorphism algorithms. *IEEE/ACM transactions on computational biology and bioinformatics*, 14(1):193–203, 2016.

[2] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14(7):1–13, 2013.

[3] Vincenzo Carletti, Pasquale Foggia, Antonio Greco, Mario Vento, and Vincenzo Vigilante. Vf3-light: A lightweight subgraph isomorphism algorithm and its experimental evaluation. *Pattern Recognition Letters*, 125:591–596, 2019.

[4] Vincenzo Carletti, Pasquale Foggia, Pierluigi Ritrovato, Mario Vento, and Vincenzo Vigilante. A parallel algorithm for subgraph isomorphism. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 141–151. Springer, 2019.

[5] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):804–818, 2017.

[6] Vincenzo Carletti, Pasquale Foggia, and Mario Vento. Vf2 plus: An improved version of vf2 for biological graphs. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 168–177. Springer, 2015.

[7] Pedro Celis, Per-Ake Larson, and J Ian Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288. IEEE, 1985.

[8] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better

bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[9] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings 10th International Conference on Image Analysis and Processing*, pages 1172–1177. IEEE, 1999.

[10] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Fast graph matching for detecting cad image components. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, volume 2, pages 1034–1037. IEEE, 2000.

[11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[12] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[13] Luc Devroye, Pat Morin, and Alfredo Viola. On worst-case robin hood hashing. *SIAM Journal on Computing*, 33(4):923–936, 2004.

[14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, 2014.

[15] Lukas Gianinazzi and Torsten Hoefler. Parallel planar subgraph isomorphism and vertex connectivity. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 269–280, 2020.

[16] Helen M Grindley, Peter J Artymiuk, David W Rice, and Peter Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *Journal of molecular biology*, 229(3):707–721, 1993.

[17] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602, 2018.

[18] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015.

[19] Shu-Ming Hsieh, Chiun-Chieh Hsu, and Li-Fu Hsu. Efficient method to perform isomorphism testing of labeled graphs. In Marina L. Gavrilova, Osvaldo Gervasi, Vipin Kumar, C. J. Kenneth Tan, David Taniar, Antonio Laganá, Youngsong Mun, and Hyunseung Choo, editors, *Computational Science and Its Applications - ICCSA 2006*, pages 422–431, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[20] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *Knowledge Engineering Review*, 28(1):75–105, 2013.

[21] Alpár Jüttner and Péter Madarasi. Vf2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018.

[22] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 313–320, 2001.

[23] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.

[24] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, 46(11):1547–1569, 2016.

[25] Ciaran McCreesh. *Solving hard subgraph problems in parallel*. PhD thesis, University of Glasgow, 2017.

[26] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *Journal of Artificial Intelligence Research*, 61:723–759, 2018.

[27] Ciaran McCreesh, Patrick Prosser, and James Trimble. Heuristics and really hard instances for subgraph isomorphism problems. In *IJCAI*, pages 631–638, 2016.

[28] Michael Mitzenmacher. A new approach to analyzing robin hood hashing. In *2016 Proceedings of the Thirteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 10–24. SIAM, 2016.

[29] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[30] Miles Ohlrich, Carl Ebeling, Eka Ginting, and Lisa Sather. Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *Proceedings of the 30th International Design Automation Conference*, pages 31–37, 1993.

[31] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, November 2015.

[32] N Pržulj, Derek G Corneil, and Igor Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, 2006.

[33] Ursula Redmond and Pádraig Cunningham. Temporal subgraph isomorphism. In *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*, pages 1451–1452. IEEE, 2013.

[34] Ursula Redmond and Pádraig Cunningham. Subgraph isomorphism in temporal networks. *arXiv preprint arXiv:1605.02174*, 2016.

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

[36] Christine Solnon. Experimental evaluation of subgraph isomorphism solvers. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 1–13. Springer, 2019.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Set-Centric Subgraph Isomorphism

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Kapp-Schwoerer	Lukas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 07.02.2021	L. Kapp-Schwoerer

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.