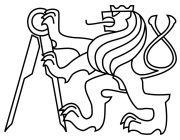
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Parallel searching for Network Motifs
Student:	Bc. Jiří Stránský
Supervisor:	RNDr. Tomáš Valla, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

The topic of the thesis is finding so-called networks motifs in a network, i.e., finding small vertex-induced subgraphs in a graph that occur significantly more often than in random graph. This problem is often utilized in analysis of biological protein interactions, searching for frequent friendship patterns in social networks, analysis of financial transactions, and others.

Our approach is to compare the frequencies of subgraphs in the input graph with their frequencies in random perturbations of the input graph. The main difficulty lies in the enormous size of the input graph, which we tackle by employing various parallelism methods.

The goals of the thesis are to:
1) survey existing algorithms and theoretical background for this problem,
2) design an own parallel algorithm, utilising both the CPU and GPU parallelism,
3) implement the designed algorithm in the C/C++ language,
4) run and evaluate performance benchmarks.

References

Will be provided by the supervisor.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 9, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Parallel searching for Network Motifs

Bc. Jiří Stránský

Supervisor: RNDr. Tomáš Valla, Ph.D.

10th May 2016

Acknowledgements

I would like to thank my supervisor RNDr. Tomáš Valla, Ph.D. for his great leadership and patience. I would also like to thank my family and colleagues for their support and the Faculty of Information Technology for the acquired knowledge, which were essential for creating this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 10th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Stránský, Jiří. *Parallel searching for Network Motifs.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Tato práce studuje takzvané vzorky v sítích. Vzorky v sítích jsou souvislé podgrafy v dané síti, které se vyskytují ve výrazně vyšších četnostech, než je očekávané v náhodných sítích. Jsou popsány různé přístupy pro hledání vzorků v sítích a na základě jejich analýzy je představen nový, vylepšený, paralelní algoritmus pro hledání vzorků v sítích, nazvaný Efise. Praktické výsledky ukazují, že Efise nabízí výrazně vyšší výkonnost než ostatní nástroje. Díky efektivní metodě paralelizace dosahuje Efise lineárního zrychlení a dokáže hledat vzorky v sítích řádově rychleji než doposud nejrychlejší nástroje.

Klíčová slova vzorky v síti, paralelní, efektivní, podgrafy, enumerace podgrafů

Abstract

This thesis studies so-called network motifs. Network motifs are connected subgraphs of a given network that occur in significantly higher frequencies than would be expected in random networks. Different approaches to finding network motifs are described and based on their analysis a new improved parallel algorithm for finding network motifs called Efise was introduced. Practical results show, that Efise offers significantly better performance than other tools and thanks to efficient parallelization method, it also achieves linear speedups, which makes it an order of magnitude faster than previous state-of-the-art tools for finding network motifs.

Keywords network motifs, parallel, efficient, subgraphs, subgraph enumeration

Contents

List of Figures

List of Tables

Introduction

In this thesis, we focus on finding so called *network motifs*. *Network motif* refers to a vertex induced subgraph that appears in a given network with a significantly higher frequency than in random networks with similar mathematical properties. *Network motifs* are a useful concept for studying structural design principles of complex networks. Originally, *network motifs* were used in various studies of biological networks, but the principle of *network motifs* can be used for analysis of any general network.

Motivation

Network motifs have been useful for a wide variety of studies. Previous applications include mostly biology studies, where networks motifs can be applied to many different networks such as metabolic networks, protein-protein interaction networks, or gene regulatory networks [8–10]. However, the principle of network motifs can be applied to practically any network to help uncover the structure of a given network. Communication networks, social networks, banking networks, web networks and many others. Practically any network can be analyzed using network motifs.

Unfortunately, the task of computing network motifs have been found to be very challenging [11]. The problem lies in the total number of subgraphs in a given network. The number of subgraphs in a given network grows exponentially with k, the worst case being a network forming a complete graph, where the number of k-vertex subgraphs is equal to the combination number $\binom{n}{k}$, where n is the number of vertices in the given network.

Due to the complexity of this task, the size of the searched subgraphs in the network has to be relatively small. Otherwise, the task is computationally unfeasible. However, programs that can compute larger networks and even faster, could help researchers uncover more information about various networks and also make searching for network motifs affordable for more researchers because of lower hardware requirements. Therefore, there is a de-

mand for efficient programs, which can find network motifs of greater size and faster.

There are already dozens of programs for finding network motifs such as Mfinder [22], FPF [25], Pajek [2], QuateXelero [13], Mavisto [26], NetMODE [17], g-triesScanner [24], FANMOD [31], Kavosh [11], FaSE [23], Subenum [28], Subdigger [29], MRSUB [27] and others. Because of this, new programs must focus more on the implementation to offer even better performance than previous programs. Therefore, we need to consider various heuristics and analyze implementations of other programs to create a new state-of-the-art program, that can compute network motifs faster than all other programs.

Goals

Since the usability of *network motifs* was discovered, dozens of programs have been developed for solving the task of finding network motifs, usually focused mainly on enumerating k-vertex subgraphs of a given network. The first programs, introduced more than 10 years ago, were already significantly improved to offer calculation speeds by orders of magnitude higher. However, recent results indicate, that further improvement is still possible. This was most recently demonstrated by programs FaSE [23] and Subdigger [29], which were introduced in the last couple of years and yet again achieved better performance. Utilizing multicore processors has also helped to achieve this goal.

We would like to examine current methods of finding network motifs, analyze the state-of-the-art programs, find a way to further improve their performance and offer an even faster tool for finding network motifs. Furthermore, we will look into possibilities of using parallelism to tackle the computational complexity of the task. We will consider various options of parallelization, from consumer level processors containing only units of cores, through specialized computational processors with dozens of cores, up to modern GPUs with hundreds to thousands of cores.

Achieved results

We present a new parallel algorithm for finding k-vertex network motifs called *Efise*. Our algorithm is based on a modification of previous algorithm and uses a significantly more efficient implementation to achieve better performance than other tools. We also present a new dynamically balanced method of parallelization. We evaluated our algorithm on various networks and shown, that our algorithm achieves linear speedups for parallel execution and is significantly superior to other tools in terms of computational time.

Notation

We use standard graph theory notation, such as $G = (V, E)$ for a graph G with the set of vertices V and the set of edges E. For simplicity, we sometimes denote the number of vertices as $n = |V|$. Vertices $v \in V$ are uniquely labeled by integers $1, \ldots, n$. For $u, v \in V$, by $u > v$ we denote that the label of vertex u is greater than the label of vertex v. Notation $G = (V, E)$ is used for both directed and undirected graphs. In case of a directed graph, we denote an edge from vertex u to vertex v as (u, v). In case of an undirected graph, we use the notation $\{u, v\}$ to denote an edge between vertices u and v.

In an undirected graph $G = (V, E)$, we call a vertex v *adjacent* to vertex u if and only if $\{u, v\} \in E$. In a directed graph $G = (V, E)$, we call vertex v adjacent to vertex u if at least one of the edges (u, v) and (v, u) is present in the graph G.

For a set of vertices $V' \subseteq V$, we define the set $N(V')$ such that for each vertex $v \in V$, $v \in N(V')$ if and only if v is adjacent to at least one vertex $u \in V'$. The set $N(V')$ is refered to as a set of *vertices adjacent to set* V' or shortly *neighbors*.

A connected vertex induced subgraph of G is called *candidate*. More precisely, for unoriented graph $G = (V, E)$ and set of vertices $V' \subseteq V$, we define a candidate g induced by set V' as $g[V'] = (V', \binom{V'}{2} \cap E)$, where $G(V')$ must be connected. For oriented graph $G = (V, E)$ and set of vertices $V' \subseteq V$, we define a candidate g induced by set V' as $g[V'] = (V', V'^2 \cap E)$, where $g[V']$ must be connected. For a candidate of size $k = |V'|$ we use the term *k-candidate*.

For a given graph G and an integer k, we denote the set of all k-candidates of G as $S_k(G)$.

We define a binary relation $\simeq$ between two candidates g_1 and g_2. We define the $g_1 \simeq g_2$ if and only if candidates g_1 and g_2 are isomorphic. Observe, that the relation $\simeq$ is equivalence, therefore it partitions the set $S_k(G)$ into equivalence classes $S_k^i(G)$, such that $S_k(G) = \bigcup_i S_k^i(G)$. Subsets $S_k^i(G)$ are called *candidate classes* of G.

We define a *frequency* $F_k^i(G)$ of a candidate class $S_k^i(G)$ as

$$F_k^i(G) = \frac{|S_k^i(G)|}{|S_k(G)|}.$$

We often use a data structure called *map*, which is a set of pairs (*key*, *value*), where each *key* may appear in the map at most once. The *key* usually represents a candidate in form of an adjacency matrix and *value* represents number of candidates with such matrix.

Problem description

Current overflow of finding network motifs can be formulated in the following way. Given a graph G and an integer $k \geq 2$, perform the following tasks.

1. Select a random graph model for generating random graphs. The most commonly used model is described in Section 1.1
2. Find frequencies $F_k^i(G)$ of all candidate classes $S_k^i(G)$.
3. Determine, which candidate classes $S_k^i(G)$ appear in graph G with a higher frequency $F_k^i(G)$, than in a random graph G' defined by the selected random graph model. Different approaches used for finding frequencies in random graphs are described in Chapter 1.

Chapter 1

Existing methods of finding network motifs

In this chapter, we will first describe the general approach to finding network motifs, that is used by recent state-of-the-art programs. Afterwards, we will describe few of the most used, most recent and most efficient programs. Many of them are strongly based on the previous ones and usually only a modification is introduced to improve performance, rather than a completely different approach. Because of this nature, we will order the programs chronologically, with respect to the date of publication.

1.1 General approach

The general approach of finding network motifs usually involves the following three steps.

1. Find all candidates in the network.
2. Determine, which candidates are isomorphic and partition them into candidate classes.
3. Determine, which candidate classes occurred with higher frequencies than in random networks under some specific random graph model.

The first step is usually reduced to enumerating only candidates of a certain size k. The enumeration time grows with the number of candidates being enumerated and as the total number of k-candidates grows exponentially with k, we are restricted only to candidates of a small size. The enumeration process can be very time consuming even for small, sparse networks.

Another approach is to sample a small subset of candidates. If the subset is chosen randomly without any bias, we should get approximately the same

frequencies as we get for a full enumeration. However, this approach can completely miss some candidate classes, especially if their frequency is low.

The second step involves a well known problem of graph isomorphism. There are many efficient programs for detecting graph isomorphism available. Most of the algorithms for finding network motifs use such programs for calculating so-called *canonical labeling* of a candidate.

Definition 1 (Canonical labeling)**.** *Canonical labeling is any transformation $f(M)$, which transforms the adjacency matrix M of a graph G into a matrix M' called* canonical label *of G, such that the transformation satisfies that $f(M_1) = f(M_2)$ if and only if $G_1 \simeq G_2$, where M_1, M_2 are an adjacency matrices of G_1 and G_2, respectively.*

Canonical labeling allows us to partition candidates into subgraph classes.

Because calculating canonical labeling for each found candidate has been a performance bottleneck for finding network motifs for a long time, recent programs began to use various heuristics to partition candidates in multiple phases. This works as a multilevel system, where on the input, we have each candidate as an individual group and on the output we have them partitioned into subgraph classes. Each level merges some of the groups together, with an idea that the calculation complexity increases and number of groups decreases, therefore the more complex calculations are not performed too often.

Current classification systems usually use the following levels.

1. Put candidates with the same adjacency matrix into the same group.
2. Use a heuristic to merge as many groups as possible, while maintaining the calculation fast.
3. Use canonical labeling to perform final merging of groups into subgraph classes.

For the third step, we need to specify the random graph model. The most common model for this task is, that a random graph G' is chosen uniformly from $SEQ(G)$, where $SEQ(G)$ is a set of all graphs with the same degree sequence as G. There have been studies concerning the problem of subgraph distribution within such graphs for directed sparse random graphs [18, 21].

With respect to the given random graph model, the third step of finding network motifs is usually performed by generating a large number of random graphs (hundreds to thousands), finding frequencies of subgraph classes in them and calculating an average frequency. Random graphs are generated with respect to a random graph model being used. This approach gives accurate results, but is very time consuming, because to find frequencies of all generated random graphs, steps one and two need to be repeated hundreds of times and therefore the time assigned to these steps has to be significantly reduced. Therefore, only candidates of even smaller size k can be enumerated.

An alternative approach is to use approximations to determine candidate frequencies in random graphs. This approach does not involve explicit random graph generation and enumeration and therefore can offer better performance in terms of required execution time. However, this comes for a cost of less accurate results, as we get only an approximation of the frequencies.

1.2 FANMOD

FANMOD [31] by Wernicke is one of the most notable programs. Even though it was published over 10 years ago, it is still being used for comparison in benchmarks. Wernicke introduced approaches, which are (with modifications) still applied in recent programs. In this section, we will describe some of the approaches introduced by Wernicke. We focused on those, which have influenced recent programs the most.

1.2.1 ESU

ESU (Enumerate SUbgraphs) is an algorithm introduced by Wernicke for enumerating all k-candidates of graph G. The ESU algorithm is given a graph $G = (V, E)$ and an integer k, $2 \leq k \leq |V|$ and finds all k-candidates of G. The algorithm is shown in Algorithm 1.1.

Algorithm 1.1 ESU(G,k)

Input: A graph $G = (V, E)$ and an integer $1 \leq k \leq |V|$.
Output: A map of all k-candidates in G.
Procedure:

1. $Map := \emptyset$
2. For each vertex $v \in V$
    1. $Ext := \{u \in N(\{v\}) \mid u > v\}$
    2. $Map :=$ ExtendSubgraph(G,k,v,$\{v\}$,Ext,Map)
3. return Map

In Algorithm 1.1 we use function GetCanonicalLabel($g[Sub]$), which returns a canonical label of candidate $g[Sub]$. This operation is performed by *NAUTY*.

NAUTY by McKay [19] is the current leading algorithm for detecting graph isomorphism. Therefore, it is used to perform the canonical labeling subtask by most algorithms for finding network motifs.

Definition 2 (ESU-tree)**.** *Calling the function* ESU*(G,k) creates a tree of recursive function calls of function* ExtendSubgraph*, called the* ESU-tree*, where function* ESU*(G,k) represents the root.*

Each node of the *ESU-tree*, except the root, is labeled by a pair of sets (Sub, Ext), which correspond to the parameters of function ExtendSub-

Algorithm 1.2 ExtendSubgraph(G,k,v,Sub,Ext,Map)

Input: A graph $G = (V, E)$, an integer $1 \leq k \leq |V|$, a vertex $v \in V$, a set Sub, a set Ext and a map Map.
Output: A map of k-candidates in G.
Procedure:

1. If $|Sub| = k$
   1. $label :=$ GetCanonicalLabel($g[Sub]$)
   2. If Map contains key $label$, then increment value of Map_{label}
   3. Else insert ($label$,1) into Map
2. Else
   1. While Ext is not empty
      1. Remove an arbitrary vertex w from Ext
      2. $Ext' := Ext \cup \{u \in N(\{w\}) \mid u \notin N(Sub), u > v\}$
      3. $Map :=$ ExtendSubgraph(G,k,v,$Sub \cup w$,Ext',Map)
3. Return Map

graph(G,k,v,Sub,Ext,Map), which is represented by the node. Nodes with label (Sub, Ext) are located at depth $|Sub|$ of the ESU-tree. Each call of function ExtendSubgraph(G,k,v,Sub,Ext,Map) is represented by an edge in the *ESU-tree*, leading from the node representing the caller function to the node representing the callee function.

Theorem 1 (Wernicke [31])**.** *Algorithm* ESU*(G,k) enumerates each candidate of a given size $k \geq 2$ in a given graph G exactly once.*

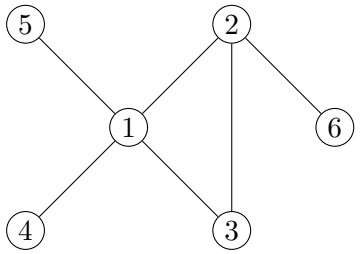


Figure 1.1: Sample graph G

In Figure 1.2 we can see an example of ESU-tree. This example shows the ESU-tree for candidates of size 3 in a graph G ilustrated in Figure 1.1. The ESU-tree has 8 leafs at depth $k = 3$, which correspond to 3-candidates in the graph G.

1.2.2 Uniform partial enumeration

The ESU algorithm provides an efficient way of enumerating all k-candidates. However, as the complexity of enumerating all k-candidates grows exponen-

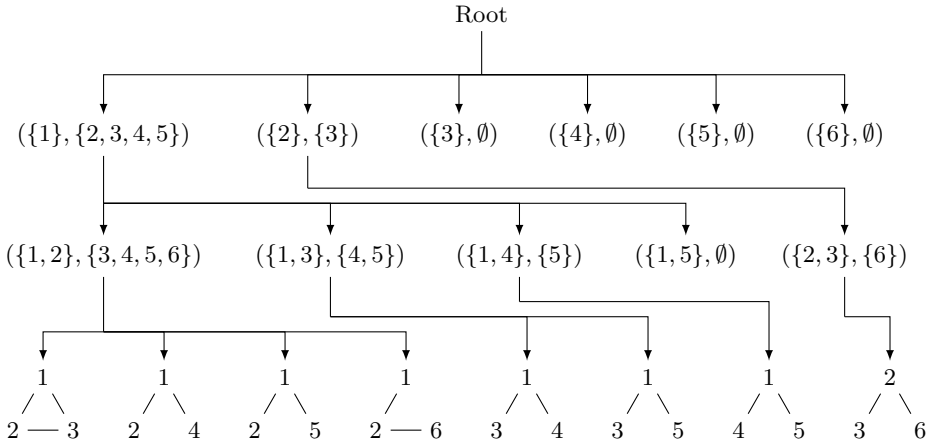


Figure 1.2: *ESU-tree* created by calling function ESU(G,3), where G corresponds to the sample graph shown in Figure 1.1.

tially with k, full enumeration may often be too slow.

To overcome this problem, Wernicke [31] proposed a different approach, which enumerates only a subset of candidates by traversing only limited parts of the ESU-tree. For unbiased results, we have to ensure, that each leaf is reached with equal probability. For this purpose, we introduce a slight modification of the ESU algorithm.

For each depth $0 \leq d < k$ in the ESU tree, we introduce a probability coefficient $0 \leq p_d \leq 1$. This coefficient determines, whether we will traverse each particular subtree in ESU-tree, or not. At step 1.2. of Algorithm 1.1, we execute the call of function ExtendSubgraph with a probability of p_0 and at step 2.1.3. of Algorithm 1.2, we execute the recursive call of ExtendSubgraph with a probability of $p_{|Sub|}$. This modified algorithm is called *RAND-ESU*.

Lemma 1 (Wernicke [31])**.** *RAND-ESU visits each leaf in ESU-tree with equal probability*

$$\prod_{d=0}^{k-1} p_d.$$

Proof. Let l be a leaf of the ESU-tree. Let us denote the nodes on the path from root to the leaf l in the ESU-tree as $n_0, n_1, \ldots, n_k$ and the probability of visiting the node n in ESU-tree as $P(n)$. We always visit the root, hence $P(n_0) = 1$. The probability of calling a recursive function ExtendSubgraph(G,k,v,*Sub*,*Ext*,*Map*) at depth d is p_d, therefore $P(n_i) = p_{d-1}P(n_{i-1})$. Now, for any leaf node t in the ESU-tree, the probability of visiting such node is by induction $P(t) = \prod_{d=0}^{k-1} p_d$. □

1.2.3 Direct calculation of candidate class frequency

In Section 1.1 we have pointed out, that the third step of finding network motifs can be very time-consuming, as it usually involves many iterations of the first and second step, i.e. enumeration and classification af a given graph. Wernicke [31] offers an alternative approach, called *Direct*, which does not rely on explicit random graph enumeration.

As a random graph model, we take the model described is Section 1.1. *Direct* uses a mathematical estimation of candidate class frequency based on the degree sequence of the graph. The main advantage of this approach is, that it does not require as much time, as explicit enumeration, but in addition this approach also offers the ability to focus on specific candidate classes.

We will derive the formula by following the steps performed in explicit enumeration. Ideally, we would calculate the candidate frequencies for each candidate class in each graph $G' \in SEQ(G)$. The average frequency of a particular candidate class would be the average frequency of such candidate class among all $G' \in SEQ(G)$. Formally,

$$\overline{F_k^i(G)} = \frac{1}{|SEQ(G)|} \sum_{G' \in SEQ(G)} \frac{|S_k^i(G')|}{|S_k(G')|},$$

where $\overline{F_k^i(G)}$ denotes the average frequency of a candidate class $S_k^i(G)$ among all graphs $G' \in SEQ(G)$.

Milo et al. [20] observed, that the total number of k-candidates among different graphs $G' \in SEQ(G)$ tends not to vary very much. Therefore, we will assume, that $\sum_j |S_k^j(G')|$ is the same for all $G' \in SEQ(G)$, which allows us to use a common denominator and get the following expression.

$$\overline{F_k^i(G)} = \frac{\sum\limits_{G' \in SEQ(G)} |S_k^i(G')|}{\sum\limits_{G' \in SEQ(G)} |S_k(G')|}.$$

In the nominator of the formula above, we are searching for the number of candidates belonging to specific candidate class among all graphs with prescribed degree sequence. However, we can flip the order and look for the number of graphs with a prescribed degree sequence, which contain a specific candidate class. (For the denominator, we are looking for any connected candidate, but we can still perform the same operation.) As graphs $G' \in SEQ(G)$ all have the same set V, we can reformulate the formula in the following way.

$$\overline{F_k^i(G)} = \frac{\sum\limits_{V_k \subseteq V} |\{G' \in SEQ(G) \mid g[V_k] \in S_k^i\}|}{\sum\limits_{V_k \subseteq V} |\{G' \in SEQ(G) \mid g[V_k] \text{ is connected}\}|}$$

where V_k is a subset of V of size k and $g[V_k]$ denotes a candidate induced by vertices V_k in graph G'.

Wernicke [31] shows, that using results of Bender [3] and Canfield [4], both the nominator and denominator can be estimated with a Monte Carlo approach by randomly sampling k-size subsets of V.

The *Direct* approach is much more efficient than explicit enumeration of candidates in random generated graphs and provides a reasonable estimation of candidate classes frequencies (see section Results in [31]).

1.2.4 Implementation

Algorithm 1.1 gives a general idea of how the ESU algorithm works, but to analyze the efficiency of FANMOD, we have to also look into its source code and consider the actual implementation and data structures in use. This can significantly vary from the algorithm description and can both positively or negatively influence the performance of the program.

First difference between the description of ESU and the implementation in FANMOD is that program does not utilize recursion. A simple loop with a depth variable is used to simulate recursive traversal of the ESU-tree.

The input graph G is represented as an ordered adjacency list, i.e. for each vertex v a sorted array of vertices $u \in N(\{v\})$ is stored. Additionally, a separate hash table is used to store all edges in the graph, which is used to efficiently answer the query for presence of a given edge in graph G. This is required when building an adjacency matrix of a candidate.

The set *Ext* varies at each depth. However, it is represented as a an array of size $k \cdot \max_{v \in V} |N(\{v\})|$, an additional array of size n and a few arrays of size k. To explain how the set *Ext* is represented with these arrays, we need to describe how the whole implementation of ESU works.

Let us introduce a few variables, which are shown in Table 1.1 along with a brief description.

Variable	Description
d	Current depth in the ESU-tree.
$sub[1 \dots k]$	Current vertices in *Sub*.
$vis[1 \dots n]$	Vertex $v \in N(Sub)$ if and only if $vis[v] \neq 0$.
$ext[0 \dots m]$	Sets *Ext*, $m = k \max_{v \in V} \|N(\{v\})\| - 1$.
$ext_{min}[1 \dots k]$	Indexes of the beginnings of the sets added into *Ext*.
$ext_{pos}[1 \dots k]$	Indexes of the beginnings of sets *Ext*.
$ext_{max}[1 \dots k]$	Indexes of the ends of sets *Ext*.

Table 1.1: Important variables used in the implementation of FANMOD.

A continuous subarray $ext[ext_{pos}[d] \dots ext_{max}[d] - 1]$ represents the set *Ext* at depth d. When removing an arbitrary vertex w from set *Ext* at depth d, we choose the vertex at position $ext[ext_{pos}[d]]$, which is done by simply incrementing $ext_{pos}[d]$.

A continuous subarray $ext[ext_{min}[d] \dots ext_{max}[d] - 1]$ represents a set of vertices added to Ext at level d to create Ext', as described in Algorithm 1.1.

At depth $d = 1$, we always have the initial set Ext of size 1 containing only the element $sub[1]$. A new set Ext' is created by setting $ext_{pos}[d+1] := ext_{pos}[d]$ and $ext_{max} := ext_{max}[d]$ and expanding set Ext' by additional vertices, each time incrementing $ext_{max}[d+1]$.

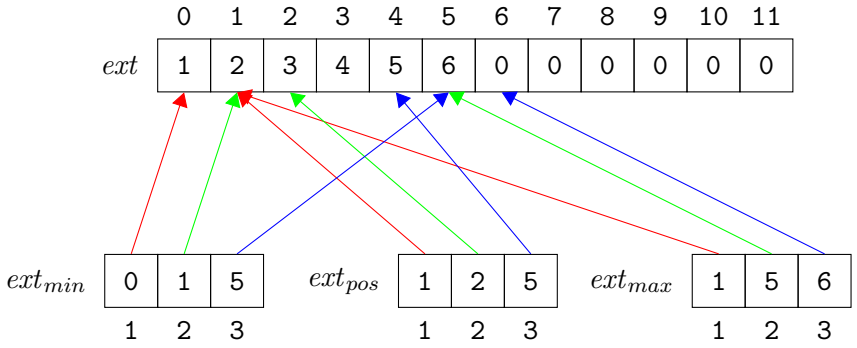


Figure 1.3: An example of $ext*$ arrays while traversing ESU-tree in 1.2 for graph G in 1.1.

Figure 1.3 shows an example of how the $ext*$ arrays can look while traversing the ESU tree shown in Figure 1.2. In Figure 1.1, we can observe, that $\max_{v \in V} |N(\{v\})| = 4$, hence $|ext| = k \max_{v \in V} |N(\{v\})| = 12$.

Let us describe the situation in Figure 1.3. We begin with $Ext = \{1\}$ at depth $d = 1$. We remove vertex 1, obtain $Ext = \emptyset$ and add neighbors of vertex 1 to create $Ext' = \{2, 3, 4, 5\}$. At depth 2, we remove vertex 2, obtain $Ext = \{3, 4, 5\}$ and add neighbors of 2 to create $Ext' = \{3, 4, 5, 6\}$. Finally, we remove vertices from Ext at depth 3 one by one and obtain $Ext = \{5, 6\}$. This situation is shown in the illustration, where for each depth $1 \leq d \leq k$, the set Ext is stored in $ext[ext_{pos}[d] \dots ext_{max}[d] - 1]$.

The Algorithm 1.3 gives a detailed description of how the arrays mentioned above are utilized.

We still need to address the problem of creating the adjacency matrices of candidates and their subsequent partitioning into subgraph classes. Building the adjacency matrix is handled by the procedure UpdateSubgraph(G,g,d,sub), used in Algorithm 1.3 and shown in 1.4. ESU is continuously building an adjacency matrix of candidates and stores it in a 64-bit variable. This restrains the implementation to $k \leq 8$ for directed graphs.

For each candidate its adjacency matrix is passed to NAUTY, which transforms it into the form of canonical labeling.

To maintain the number of candidates in each candidate class, a simple hash table is used. The hash table stores key-value pairs, where the 64-bit canonical label acts as a key and the number of candidates currently found with

Algorithm 1.3 IMPLEMENTATIONESU(G,k)

Input: A graph $G = (V, E)$ and an integer $1 \leq k \leq |V|$.
Output: All k-candidates in G.
Procedure:

1. For each vertex $v \in V$
    1. $h :=$ EMPTYHASHTABLE()
    2. $g :=$ EMPTYSUBGRAPH(k)
    3. $d := 1$
    4. $ext_{min}[d] := 0$
    5. $ext_{pos}[d] := 0$
    6. $ext_{max}[d] := 1$
    7. $ext[0] := v$
    8. $vis[v] := 1$
    9. While $d \neq 0$ do
        - If $ext_{pos}[d] = ext_{max}[d]$
            1. For each $u \in ext[ext_{min}[d] \ldots ext_{max}[d] - 1]$ set $vis[u] := 0$
            2. $d := d - 1$
        - Else if $d = k$
            1. For each $u \in ext[ext_{pos}[d] \ldots ext_{max}[d] - 1]$
                1. $sub[d] := u$
                2. call UPDATESUBGRAPH(G,g,d,sub)
                3. call INCREMENTSUBGRAPHCOUNT(h,g)
                4. $ext_{pos}[d] := ext_{pos}[d] + 1$
        - Else
            1. $u := ext_{pos}[d]$
            2. $sub[d] := u$
            3. $ext_{pos}[d] := ext_{pos}[d] + 1$
            4. call UPDATESUBGRAPH(G,g,d,sub)
            5. $ext_{min}[d + 1] := ext_{max}[d]$
            6. $ext_{max}[d + 1] := ext_{max}[d]$
            7. $ext_{pos}[d + 1] := ext_{pos}[d]$
            8. $d := d + 1$
            9. For each $w \in N(u) \mid w > v$
                - If $vis[w] = 0$
                    1. $ext[ext_{max}[d]] := w$
                    2. $ext_{max}[d] := ext_{max}[d] + 1$
                    3. $vis[w] := 1$
2. return h

Algorithm 1.4 UpdateSubgraph(G,g,d,sub)

Input: A graph $G = (V, E)$, a candidate matrix g as a 64-bit variable, an integer $1 \leq d \leq k$ and a set Sub of size d as an array sub.
Procedure:

1. $v := sub[d]$
2. For each $i \in [1 \dots d-1]$
    1. $u := sub[i]$
    2. Delete edges (i, d) and (d, i) from candidate g
    3. If $(u, v) \in E$, add edge (i, d) to candidate g
    4. If $(v, u) \in E$, add edge (d, i) to candidate g

Algorithm 1.5 IncrementSubgraphCount(h,g)

Input: A hash table h and a candidate matrix g as a 64-bit variable.
Procedure:

1. $key :=$ GetCanonicalLabel(g)

- If h contains key key, increment $h[key]$
- Otherwise insert $(key, 1)$ into h

such canonical labeling is stored as a value. Algorithm 1.5 shows, that for each candidate, first a canonical label is found and afterwards the corresponding value is the hash table is incremented.

We consider the implementation of FANMOD shown in Algorithm 1.3 to be very efficient for traversing the ESU-tree. To analyze it in more detail, let us denote the number of nodes in the ESU-tree as T and the number of candidates, i.e. the number of leaves at depth k, as R. The traversal of the ESU-tree itself is linear with T. However, if we consider the concurrent operations for classification of candidates, the complexity may change. The procedure UpdateSubgraph(G,g,d,sub) is called from each node of the ESU-tree and in Algorithm 1.4 we can observe that its time complexity is $\mathcal{O}(d)$ (lookup for edge presence in graph G is executed as a query to the hash table of edges in $\mathcal{O}(1)$ time). Now, we will consider three possible situations and analyze the time complexity of ImplementationESU(G,k) for each of them. (Note, that we do not count the complexity of NAUTY used in function GetCanonicalLabel(g))

1. $R = \Theta(T)$
   In this case we assume, that the number of leafs at depth k is asymptotically linear with the total number of nodes. This holds if the ESU-tree is for example a full binary tree. Now the complexity of the algorithm is $\Theta(Rk)$. This case is the most common for real-life input graphs (see Subsection 4.4.2).

2. $R = \Omega(\frac{T}{k})$
   In this case we assume, that the number of leafs is asymptotically at most

k times smaller then the total number of nodes. An example situation is when all node of the ESU-tree except the root and leafs at depth k have exactly one child node. Now the complexity is $\mathcal{O}(R\sum_{d=1}^{k} d) = \mathcal{O}(Rk^2)$.

3. $R = o(\frac{T}{k})$
   Last case covers situations, where many nodes have no child nodes. We cannot determine the upper bound with respect to R, therefore we claim only the complexity $\mathcal{O}(Tk)$, which stands for all cases.

The naive way of finding the adjacency matrices of candidates would be creating the whole matrix in leafs, which would have the time complexity of $\Theta(Rk^2)$. This would be an improvement over the third situation, but a vast majority of graphs have an ESU-tree with properties described in the first situation, hence the naive approach is considered to be worse.

The last issue is the procedure IncrementSubgraphCount(h,g). It calls function GetCanonicalLabel(g), which is handled by NAUTY. The problem of finding a canonical labeling is a hard problem and despite the fact, that NAUTY does a very good job when finding a canonical label in practice, it usually consumes about 1 μs of computation time, which turns out to be the main limit for the performance of FANMOD.

1.3 Kavosh

Kavosh [11] introduced a new method of enumerating k-candidates of a graph G. The general approach to finding network motifs is the same as the one used in FANMOD, but the enumeration part does not utilize the ESU-tree. Additionally, Kavosh also includes various statistical measures to determine motifs significance.

1.3.1 Enumeration

Similarly to FANMOD, Kavosh uses a recursive traversal of a tree, which is dynamically creating during the traversal. However, the tree itself is different from the ESU-tree. For each vertex $v \in V$ a separate tree is created, where v is the root. Tree rooted in v is denoted as T_v. After traversing the tree T_v, v is removed from the graph G. Because only vertices $\{u \in V \mid u > v\}$ are allowed to be present in T_v, we can select roots by increasing label and do not have to worry about removing vertices $v \in V$ from the graph G after traversing each tree T_v.

During the traversal of a tree T_v, we need the sets *Sub*, *Last* and *Vis*. Set *Sub* contains vertices currently selected for the candidate g, same as in FANMOD. Set $Last \subseteq Sub$ contains vertices added to *Sub* in the parent node of current node in tree T_v. Set *Vis* contains vertices in *Sub* and all their neighbours with label greater than v. Formally, $Vis = Sub \cup \{u \in N(Sub) \mid u > v\}$.

In each node, we create a set $Open = \{u \in V \mid u \notin Vis, u > v\}$, which is afterwards added into the set Vis.

Next, we need try all combinations of vertices $C \subseteq Open$, such that $1 \leq |C| \leq \min(|Open|, k - |Sub|)$ and for each such combination, we add it to the set Sub and follow a recursive call deeper in the tree T_v.

For $Sub = k$, we have reached a leaf of tree T_v and found a new candidate $g[Sub]$.

We can see a description of Kavosh in Algorithm 1.6 and 1.7. Figure 1.5 illustrates a tree T_v for a sample graph G shown in Figure 1.4, $v = 1$ and $k = 4$.

Algorithm 1.6 Kavosh(G,k)

Input: A graph $G = (V, E)$ and an integer $1 \leq k \leq |V|$.
Output: A map of all k-candidates in G.
Procedure:

1. $Map := \emptyset$
2. For each vertex $v \in V$ in increasing order
    1. $Map :=$ EnumerateVertex(G,k,v,$\{v\}$,$\{v\}$,$\{v\}$, Map)
3. Return Map

Algorithm 1.7 EnumerateVertex(G,k,v,Sub,$Last$,Vis,Map)

Input: A graph $G = (V, E)$, an integer $1 \leq k \leq |V|$, a vertex $v \in V$, a set $Sub \subseteq V$, a set $Last \subseteq V$, a set $Vis \subseteq V$ and a map Map.
Output: A map of all k-candidates in G found in the tree T_v.
Procedure:

1. If $|Sub| = k$, then insert candidate $g[Sub]$ into Map
2. Else
    1. $Open := \{u \in N(Last) \mid u \notin Vis, u > v\}$
    2. For each $C \subseteq Open$ such that $1 \leq |C| \leq k - |Sub|$
        1. $Map :=$ EnumerateVertex(G,k,v,$Sub \cup C$,C,$Vis \cup Open$, Map)
3. Return Map

In Figure 1.5, we can see 4 groups of nodes, where each represents a different depth in the tree T_v. We also see different selections of vertices highlighted in red. Nodes with complete k-candidates are highlighted in green. In contrast to the ESU-tree, we can observe, that k-candidates are located at multiple depths in the tree. This is a consequence of selecting multiple vertices in a single node, which is not allowed in the ESU-tree.

In Algorithm 1.7 we need to iterate over all nonempty subsets of set $Open$ of a specific maximum size. For this task, Kavosh uses the "revolving door ordering" [14], which is considered to be the fastest algorithm for this task [11].

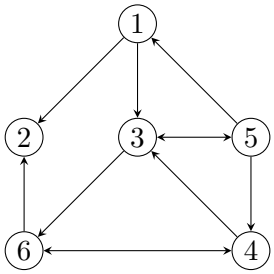


Figure 1.4: Sample graph G

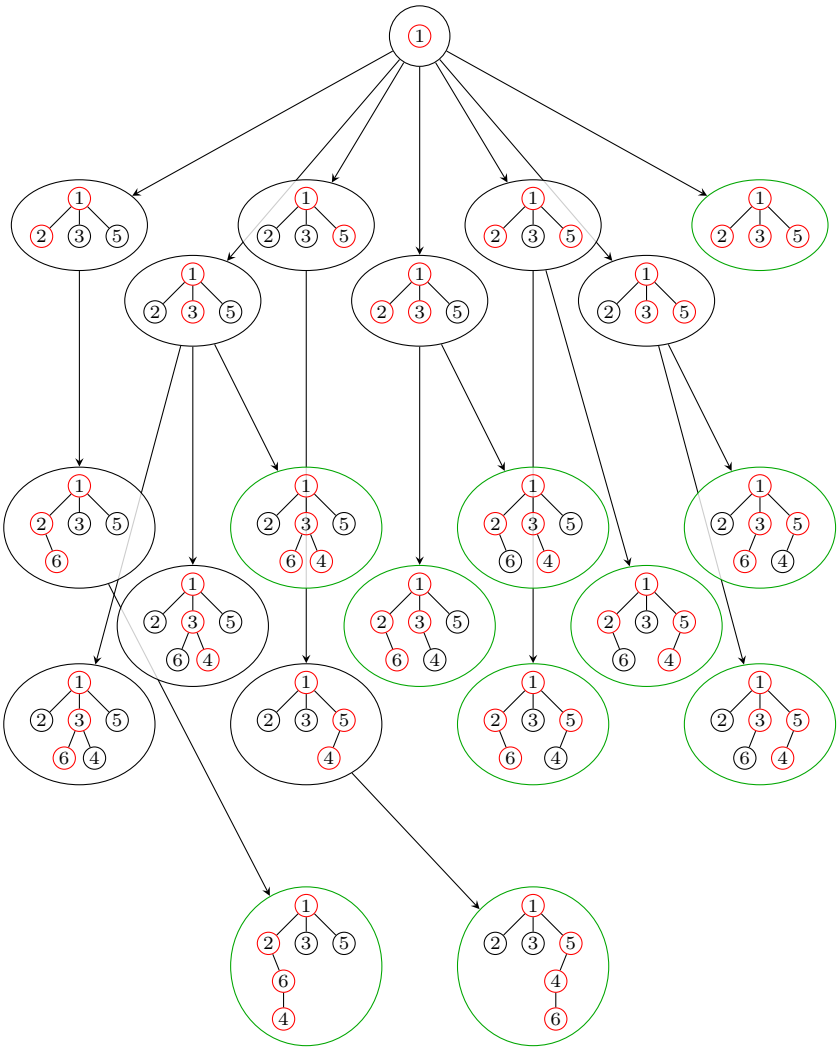


Figure 1.5: Recursive tree of the call of EnumerateVertex(G,4,1,{1},{1},2,$\emptyset$) for sample graph G in Figure 1.4

Theorem 2. Kavosh*(G,k) enumerates each k-candidate in a given graph G exactly once.*

Proof. Let us denote all candidates enumerated in the tree T_v as a multiset S_v.

We will prove theorem 2 in the following way.

1. Kavosh(G,k) enumerates each k-candidate at most once.
   - Each tree T_v, $v \in V$, contains each candidate $H \in S_v$ at most once. Therefore, S_v is a set.
   - For each pair of trees T_i, T_j, $i \neq j$, sets S_i, S_j are disjoint.
2. Kavosh(G,k) enumerates each k-candidate at least once.

First let us prove, that each tree T_v, $v \in V$, contains each candidate $H \in S_v$ at most once. We will assume the opposite statement and show a contradiction for it. Therefore, we assume there are two different nodes $n_1, n_2 \in T_v$, $n_1 \neq n_2$, where both nodes n_1 and n_2 represent the same candidate $H \in S_v$. To both nodes n_1, n_2, there exists a path from the root, denoted as p_1, p_2 respectively. As $n_1 \neq n_2$, there has to exist a node x, which is the last common node for paths p_1, p_2. We claim that for the candidate H, the only way to choose the set C in Algorithm 1.7 is such that $C = \mathit{Open} \cap H$. We are not allowed to select more vertices, as they would not belong to H and we are also not allowed to omit any vertex $u \in C$, because u would be added into the set *Vis* and we could not select vertex u later. Therefore, $x = n_1 = n_2$. As we have shown, that each $H \in S_v$ is contained in S_v at most once, we can state, that S_v is not only a multiset, but also a set.

Next let us show, that for each pair of trees T_i, T_j, $i \neq j$, the sets S_i, S_j are disjoint. Observe, that a candidate H has to be present in tree T_m, where $m = \min_{v \in H} v$. Now, without loss of generality, let us assume that $i < j$. We know from the observation, that $i \in H$ for each candidate $H \in S_i$. Candidates from S_j can contain only vertices $v \geq j$, hence none of the candidates from S_j can contain vertex i, because $v \geq v_j > v_i$. Therefore, the sets S_i and S_j are disjoint.

Finally, we will prove, that Kavosh(G,k) enumerates each k-candidate at least once. We have shown in the previous paragraph, that there exists only one unique way of choosing the set C. Now we need to show, that $C \neq \emptyset$ until $\mathit{Sub} = H$. We claim that $N(\mathit{Sub}) \in \mathit{Open}$. We know that each $u \in N(\mathit{Sub})$ must appear in the current *Open*, or in some *Open* set at lower depths of tree T_v. If u appeared in *Open* previously, it must have already been selected, as proved before, and hence $u \in \mathit{Sub}$ and $u \notin N(\mathit{Sub})$. Therefore, $N(\mathit{Sub}) \subseteq \mathit{Open}$. Now, if $C = \mathit{Open} \cap H = \emptyset$, then also $N(\mathit{Sub}) \cap H = \emptyset$, which means, that H is not connected and does not satisfy the definition of a candidate.

□

1.3.2 Classification

In Algorithm 1.7, a complete candidate is simply processed as "insert candidate $g[Sub]$ into *Map*". Behind this statement, the whole classification process is hidden. In Algorithm 1.8, we describe the classification process in more detail as the procedure Classify(G,Sub,k,T), where the binary search tree T represents the map of candidates.

Algorithm 1.8 Classify(G,Sub,k,T)

Input: A graph $G = (V, E)$, a set $Sub \subseteq V$, an integer $1 \leq k \leq |V|$ and a binary search tree T, representing a map of k-candidates in G.
Output: A map of k-candidates in G represented as a binary search tree.
Procedure:

1. $g :=$ EmptySubgraph(Sub)
2. For each $v \in Sub$
    1. For each $u \in Sub$ such that $u \neq v$
        1. If $(u, v) \in E$, add edge (u, v) to candidate g
        2. If $(v, u) \in E$, add edge (v, u) to candidate g
3. $key :=$ GetCanonicalLabel(g)
4. For each bit $b \in key$
    1. If $b = 1$, then follow left child in T
    2. Else follow right child in T
5. Increment counter in current node of T
6. Return T

Procedure Classify(G,Sub,k,T) builds an adjacency matrix for the candidate passed in Sub and then, similarly to FANMOD, utilizes NAUTY to transform it into the form of canonical labeling. However, in contrast to FANMOD, Kavosh does not use a hash table to store candidate classes, but uses a binary search tree instead. Each candidate class has a unique path of length $k(k-1)$ in this tree, defined by its canonical labeling. Each leaf of the tree has an additional counter, holding the number of candidates belonging to the respective candidate class.

1.3.3 Implementation

In contrast to FANMOD, Kavosh actually utilizes recursion.

The main difference to FANMOD (in terms of implementation) is in the way Kavosh stores the input graph G. In addition to an ordered adjacency list, which both FANMOD and Kavosh use, FANMOD uses an additional hash table of edges to answer queries to whether an edge (u, v) is present in the graph G or not. For this query, Kavosh uses a full adjacency matrix. The adjacency matrix requires n^2 bits of memory, which makes Kavosh unusable for large graphs and at least cache unfriendly for medium sized graphs.

The set *Sub* is represented by a 2D array *sub* of size k^2. At each depth d of the tree T_v, vertices $u \in C$, are stored in the d-th row of the array *sub*.

Set *Vis* is represented by a simple boolean array $vis[1 \dots n]$. If $vis[v] = 1$, then $v \in Vis$. Otherwise, $v \notin Vis$.

Set *Open* is represented by a 2D array, where similarly as for set *Sub*, vertices of the set *Open* at depth d are stored in the d-th row. Expectedly, the array has k rows. The number of columns is $k \max_{v \in V} |N(\{v\})|$. This size is sufficient, as *Open* contains only vertices $u \in N(\{Sub\})$, where $|Sub| <= k$ and each vertex $u \in Sub$ has $\max_{v \in V} |N(\{v\})|$ space reserved, which is obviously not less than $|N(\{u\})|$. The total amount of memory required for sets *Open* across all depths of the tree T_v is therefore $\mathcal{O}(k^2 \max_{v \in V} |N(\{v\})|)$.

The method of choosing $C \subseteq Open$ is well described in the original paper [14] and will not be further discussed here.

The implementation of Classify(G,*Sub*,k,T) in 1.8 suffers from similar problem as the classification part in FANMOD. That is mostly the usage of NAUTY.

The first step is creating a continuous array of *Sub* from a 2D array *sub*. This is actually done efficiently thanks to the counters of items in each $sub[i]$, $1 \leq i \leq k$. Counters help reach the time complexity of $\mathcal{O}(k + k) = \mathcal{O}(k)$.

Next we need to create the adjacency matrix of candidate *Sub*. Unfortunately, Kavosh does not create this matrix continuously while traversing the tree, as FANMOD does, but instead creates the whole matrix in the leaf. This yields the time complexity of $\mathcal{O}(k^2)$.

Finally, after receiving a canonical labeling from NAUTY, the binary tree of candidate classes is traversed in $\mathcal{O}(k^2)$ time to increment the counter.

Following the evaluation from the previous section, where R is the number of leafs in the tree, we can claim a time complexity of $\Omega(Rk^2)$, as we do not evaluate the complexity of the whole traversal, but only the cumulative complexity in leafs and yet again, without the complexity of canonical labeling.

Overall, the implementation of Kavosh is generally less optimized than FANMOD and is also very memory demanding, because of the adjacency matrix of G that requires $\mathcal{O}(n^2)$ memory. The main issue however is still the inefficient usage of NAUTY. Finding canonical labeling of every single candidate still creates a bottleneck for the execution time and the differences between other parts of Kavosh and FANMOD can not make much of a difference.

1.4 FaSE

FaSE [23], published by Pedro Paredes and Pedro Ribeiro in 2013, made a significant improvement in the performance of enumerating network candidates by solving the main problem of FANMOD and Kavosh, the extensive usage of NAUTY.

The previous standard procedure was calling NAUTY for every single candidate. However, before calling NAUTY, the initial adjacency matrix must first be built. The problem is, that NAUTY is usually called many times on the same initial matrix. Therefore, if we managed to somehow cache the results of NAUTY calls on each candidate, we could save a lot of redundant calls of NAUTY.

If we were to make it simple, FaSE pretty much works as such a cache. It can also explained as a parallel to what cache does for main memory. A processor cache is not big enough for a computer, but it is much faster than the main memory and can relieve from extensive calls to main memory. In a similar way, storing only initial adjacency matrices in our task is not enough, but it relives the extensive and time-expensive calls to NAUTY.

FaSE builds on ESU algorithm from FANMOD, but to offer better performance, it utilizes so-called *G-Tries* and *LS-labeling*.

1.4.1 G-Trie

G-Trie is a data structure based on standard tries, but it is designed to store graphs. If we traverse a path from the root in a trie containing symbols, we are forming a string. A g-trie contains incremental data so that when we traverse it, we are creating a graph.

The structure of g-tries is very suitable for the ESU algorithm. In ESU, we add a single vertex v into *Sub*. This can be easily done as traversing one node deeper in a g-trie, by following the appropriate edge.

Let us describe the structure of g-tries more precisely. The root node in a g-trie represents an empty graph g. By following an edge e deeper in the g-trie, we are adding a vertex v to the graph g along with edges between vertex v and other vertices in g. These edges are stored as labels on the g-trie edge e. This method incrementally builds a graph, such that in a node n at depth d of the g-trie, we can store an arbitrary graphs of size d, whose structure is stored on the path from the root to n.

In Figure 1.6 we can see an example of how a g-trie represents graphs. In each node, newly added vertex and its edges are highlighted in red. In leafs, we can also see, why g-trie is not sufficient. Two pairs of leafs (2,4 and 3,5) are isomorphic, thus belong to the same candidate class. This problem is caused by adding vertices in different order. However, candidates which are already identical, when the initial adjacency matrix is created, always end up in the same leaf node of the g-trie.

1.4.2 LS-Labeling

A g-trie can generally use many different types of labeling. We could actually store candidates already in the form of canonical labeling in each node of the g-trie. Then, the labels of edges would except of the added vertex and edges

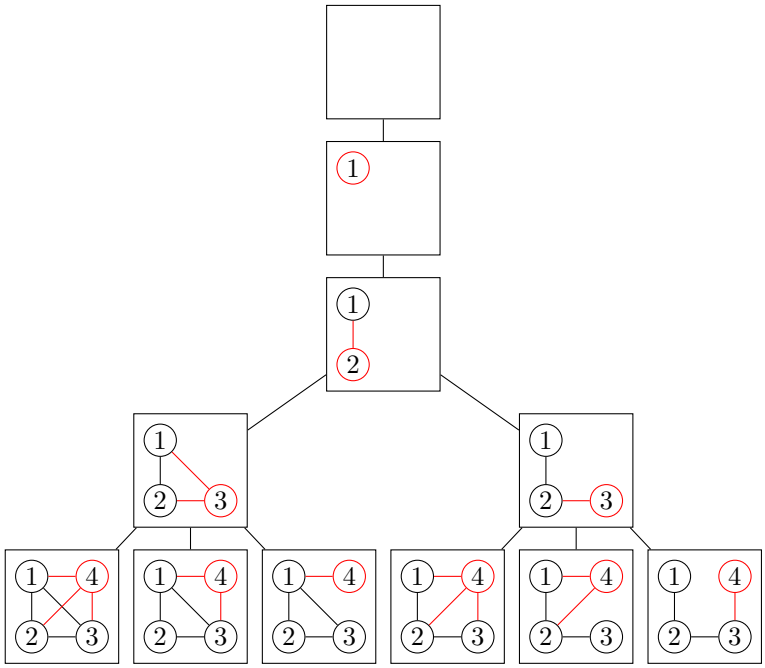


Figure 1.6: An example of a g-trie

also describe the permutation of vertices required to make the graph in destination node also canonical labeled. However, this would be computationally expensive and probably yield the g-trie inefficient due to the time spent on computing labels.

We also want the labeling to be incremental. G-tries allow labels of edges to describe the whole graph in the destination node, but that would be inefficient in both time and space usage. Therefore, we want the label to describe only the incremental information, i.e. the parts of the new graph, which were not previously present in the graph.

FaSE proposes two types of labelings. A label describing edges of the new vertex as an adjacency list and a label describing edges as a corresponding row/column in adjacency matrix. Let us have candidate Sub in the current g-trie node. When using an adjacency list LS-labeling, the g-trie edge adding a new vertex v will be labeled with a list of vertices $\{u \in N(\{v\}) \mid u \in Sub\}$. In case of adjacency matrix labeling, the g-trie edge will be labeled with b-bit identifier, where $b = |Sub|$.

1.4.3 FaSE

In Algorithm 1.9 and 1.10 we can observe how FaSE utilizes the g-trie in the original ESU algorithm. The ESU algorithm itself remains untouched, but calls to the recursive function, in this case Enumerate(G,k,T,v,Sub,Ext) are

enclosed between procedures DOWN(T,*label*) and UP(T). These procedures traverse the g-trie T to make sure the current node in T represents the current candidate in *Sub*.

Algorithm 1.9 FASE(G,k)

Input: A graph $G = (V, E)$ and an integer $1 \leq k \leq |V|$.
Output: A map of all k-candidates in G.
Procedure:

1. $T :=$ GTRIE()
2. For each vertex $v \in V$
    1. $Ext := \{u \in N(\{v\}) \mid u > v\}$
    2. $label :=$ LSLABEL($\emptyset$,v)
    3. DOWN(T,*label*)
    4. $T :=$ ENUMERATE(G,k,T,v,$\{v\}$,*Ext*)
    5. UP(T)
3. $Map := \emptyset$
4. For each candidate g with frequency f in leaves of T
    1. $label :=$ GETCANONICALLABEL(g)
    2. If *Map* contains key *label*, then increment value of Map_{label} by f
    3. Else insert (*label*,f) into *Map*
5. Return *Map*

Algorithm 1.10 ENUMERATE(G,k,T,v,*Sub*,*Ext*)

Input: A graph $G = (V, E)$, an integer $1 \leq k \leq |V|$, a g-trie T, a vertex $v \in V$, a set $Sub \subseteq V$ and a set $Ext \subseteq V$.
Output: A g-trie T.
Procedure:

1. If $|Sub| = k$, then increment counter in the current node of g-trie T
2. Else
    1. While $Ext \neq \emptyset$
        1. Remove an arbitrary vertex w from *Ext*
        2. $Ext' := Ext \cup \{u \in N(\{w\}) \mid u \notin N(Sub), u > v\}$
        3. $label :=$ LSLABEL(Sub,w)
        4. DOWN(T,*label*)
        5. $T :=$ ENUMERATE(G,k,T,v,$Sub \cup \{w\}$,Ext')
        6. UP(T)
3. Return T

Procedure LSLABEL(G,S,v) returns a label for an edge in g-trie. Parameter S contains the original candidate and v is a vertex to be added into S. The procedure identifies edges between vertex v and vertices in S from which it creates a label in a form of adjacency list/matrix, as described earlier. Procedure DOWN(T,*label*) then follows the label deeper in the g-trie. If the label

does not exist yet, it is created. Procedure Up(T) moves one node up in g-trie T, to revert the operation done by Down(T,*label*).

1.4.4 Implementation

FaSE offers a possibility to choose from different structures that can be used. The input graph G can be represented as both adjacency matrix or adjacency list. Similarly, LS-labeling can use either adjacency list or adjacency matrix type of label. Unfortunately, experiments on our dataset has shown, that multiple settings of FaSE result in incorrect results. The only combination calculating correct results for directed graphs in our dataset is an adjacency matrix representation of graph G and adjacency list type of labels for LS-labeling. For this reason, we will describe the implementation for this setting.

For the adjacency matrix of graph G FaSE uses a 2D array of size $n \times n$ and similarly as Kavosh, FaSE also stores an additional adjacency list representation.

For the set *Ext*, FaSE does not use an efficient implementation as FANMOD and instead allocates a 2D array of size $k \times n$ and when each time a new set Ext' is created, the previous set *Ext* is copied.

Another difference between FaSE and FANMOD is, that FaSE does not use a *vis* array to mark vertices adjacent to *Sub*. Therefore, when creating set Ext', FaSE has to check if each vertex $u \in N(Sub)$ in $\mathcal{O}(k)$ time.

The differences stated above show, that the implementation of the ESU algorithm itself for traversing the ESU-tree is less efficient in FaSE, than it was initially in FANMOD. However, as shown in Subsection 4.4.4, FaSE has significantly outperformed FANMOD on our dataset. This shows the efficiency of g-tries.

Nodes of a g-trie are implemented as objects. Edges of a g-trie are implemented as another trie contained within the source node. In other words, each node N_1 of a g-trie contains its own trie t. A path p from the root of trie t to node x represents a label l. If label l from the g-trie node N_1 leads to the g-trie node N_2, then the trie node x contains a pointer to N_2. As labels of the g-trie contain up to k vertices of a candidate labeled $1, \ldots, k$, nodes of a trie have also size $\mathcal{O}(k)$ to allow direct indexing of their children.

The path from the root of a g-trie to a leaf, including the lengths of tries in nodes of the g-trie, as length $\mathcal{O}(k^2)$. And as each trie node requires $\mathcal{O}(k)$ space, the g-trie with L leafs has a space complexity of $\mathcal{O}(Lk^3)$. However, if we assumed, that the number of g-trie leafs was linear to the number of g-trie leafs, similarly as we did with the ESU-tree, the space complexity would only be $\mathcal{O}(Lk^2)$.

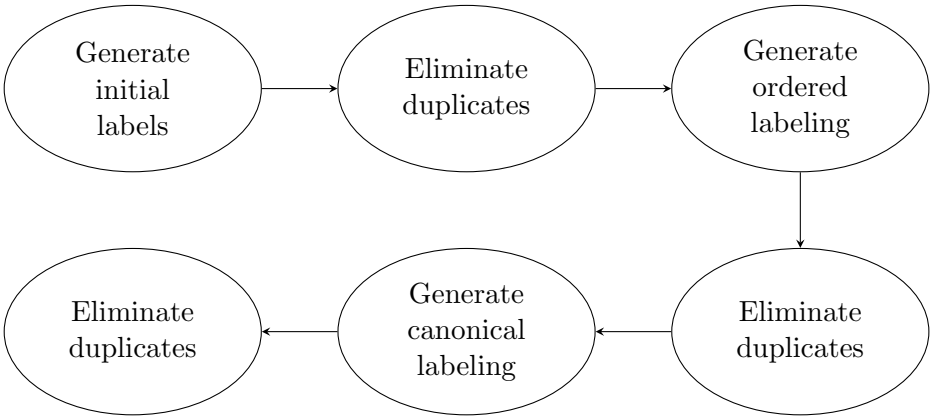


Figure 1.7: Classification process in Subdigger.

1.5 Subenum/Subdigger/MRSUB

Subenum [28], Subdigger [29] and MRSUB [27] are all programs created by Saeed Shahrivari and Saeed Jalili published in years 2014 and 2015. Sharivari and Jalili again used ESU as a base of their algorithm and introduced a powerful heuristic called "ordered labeling" to accelerate isomorphism detection. Furthermore, a parallel ESU algorithm was introduced to enable utilization of modern multicore CPUs.

The differences between Subenum and Subdigger are not significant, therefore we will only describe the more recent one, Subdigger. MRSUB is an implementation of Subdigger that supports Hadoop framework for massive parallelization.

1.5.1 Ordered labeling

Subdigger introduces a new approach to the problem of too time-consuming isomorphism detection. It utilizes a heuristic called "ordered labeling" to partition the original matrices before the canonical labeling is applied. The general idea is similar to FaSE. However in Subdigger, there is another level of reducing the number of unique matrices of candidates. Figure 1.7 illustrates, how the general process of classification in Subdigger works.

The goal of ordered labeling heuristic is to provide an oracle, which can guess, whether a relation $g_1 \simeq g_2$ holds. We have two main requirements for the oracle.

- The oracle may not produce false positives. In other words, if relation $g_1 \simeq g_2$ does not hold, the oracle may never indicate the opposite. On the other hand, the oracle may indicate, that a relation $g_1 \simeq g_2$ does not hold, even if it actually does.

- The oracle has to produce results very fast, by which we mean significantly faster, that canonical labeling.

Now, we need to define the following terms, which will help us explain how ordered labeling works.

Definition 3 (Ordered label)**.** *Let L be a list of vertices $v \in g$ for a given candidate g. The order of vertices in L is defined such that the vertices are first ordered by their in-degree and in case of a tie by their out-degree with respect to their degrees in candidate g. The adjacency matrix m of candidate g, where vertices in the matrix are sorted in the same way as in list L, is called* ordered label.

Definition 4. Candidate subclass $CS_k^i(G)$*, where i identifies a particular candidate subclass, is a set of candidates in graph G, such that $g_1 \simeq g_2$ for each two candidates $g_1, g_2 \in CS_k^i(G)$. A union of all candidate subclasses for an integer $k \geq 2$ forms the set $S_k(G)$. Formally, $\bigcup_i CS_k^i(G) = S_k(G)$.*

Now, we will explain, what the ordered labeling heuristic actually does. If we take an adjacency matrix m of a candidate g, ordered labeling will permute vertices of candidate g in such a way, that they will be ordered first by their in-degree and in case of a tie by their out-degree with respect to g. Then, the adjacency matrix m is recalculated according to the new ordering of vertices to produce the ordered labeling. If we obtain an ordered label of all candidates, we can partition them into candidate subclasses. We will assign two candidates g_1, g_2 to the same candidate subclass if and only if their ordered labels are the same. More detailed description of ordered labeling heuristic is given in Algorithm 1.11.

Algorithm 1.11 OrderedLabeling(G,C)

Input: A graph $G = (V, E)$ and a set of candidates C.
Output: A map of candidate subclasses.
Procedure:

1. $Map := \emptyset$
2. For each candidate $g \in C$
   1. $label :=$ GetOrderedLabel(G,g)
   2. If Map contains key $label$, then increment value of Map_{label}
   3. Else insert $(label, 1)$ into Map
3. Return Map

Now, let us describe the whole classification process in Subdigger, visualized in Figure 1.7.

1. Enumerate all k-candidates using ESU, while partitioning them into groups, where candidates having the same initial adjacency matrix generated by ESU, belong to the same group.

Algorithm 1.12 GetOrderedLabel(G,g)

Input: A graph $G = (V, E)$ and a candidate g.
Output: An ordered label of candidate g.
Procedure:

1. Let $M_{i,j}$ be the adjacency matrix of g and L be a list of vertices of g
2. Sort vertices in L first by their in-degree and in case of a tie by their out-degree with respect to their degrees in candidate g
3. Let the function $R(v)$ be the rank of vertex v in list L
4. Let $N_{i,j}$ be a new adjacency matrix of candidate g
5. For each pair of vertices $u, v \in L$
    1. If $M_{u,v} = 1$, set $R(v)$-th column in $R(u)$-th row of matrix N to 1
    2. Else set $R(v)$-th column in $R(u)$-th row of matrix N to 0
6. Return N

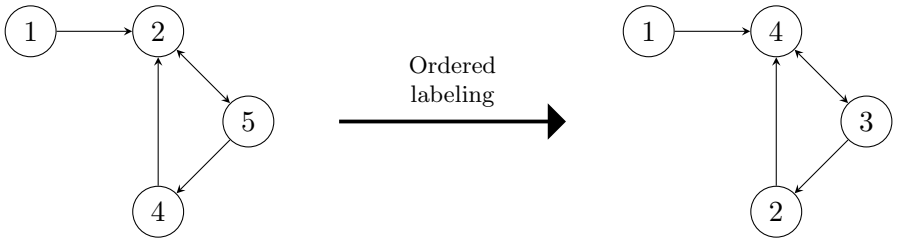


Figure 1.8: An example of ordered labeling.

2. Merge some of the candidate groups created in the previous step into candidate subclasses by applying ordered labeling.

3. Apply canonical labeling to candidate subclasses from the previous step to merge them into the final subgraph classes.

In contrast to FaSE, Subdigger does not utilize g-tries to perform step 1, but instead uses a simple hashing to perform the initial partitioning.

For step 2, we should note, that for a given candidate g, there may exist i, j, $i \neq j$, such that $g \in CS_k^i(G)$ and also $g \in CS_k^j(G)$. In other words, candidate subclasses $CS_k^i(G)$ and $CS_k^j(G)$ do not have to be disjoint, i.e. they may have a nonempty intersection.

Therefore, we can have multiple possible candidate subclasses, where candidate g could be assigned. We may select an arbitrary one of them, but we have to guarantee, that each candidate is assigned to exactly one candidate subclass. This is trivially achieved by generating an ordered label and assigning a candidate to the corresponding subclass only once.

From Definition 4, we can observe, that for each candidate subclass $CS_k^i(G)$, there exists a unique candidate class S_k^j such that $CS_k^i(G) \subseteq S_k^j(G)$. Therefore, we can simply assign candidate subclasses to candidate classes in step 3.

We can now validate the requirements made for the ordered labeling heuristic.

- Ordered labeling indicates, that two $g_1 \simeq g_2$ only if ordered labels of g_1 and g_2 are the same. If ordered labels, i.e. adjacency matrices, of g_1 and g_2 are the same, then they trivially must be isomorphic, therefore $g_1 \simeq g_2$ holds and therefore ordered labeling can not produce false positives.

- Function GetOrderedLabel(G,g) runs in $\mathcal{O}(k^2)$ time, because we can use an $\mathcal{O}(k^2)$ sorting algorithm and a simple precomputed lookup array, that performs function $R(v)$ in $\mathcal{O}(1)$ time. Compared to $\mathcal{O}(k!)$ complexity of canonical labeling, ordered labeling should be significantly more efficient.

1.5.2 Parallel subgraph enumeration and classification

Shahrivari and Jalili [28] offered a method of parallelization of both the ESU algorithm for enumeration phase and the classification phase.

The principle of proposed parallel enumeration algorithm is in encapsulating disjoint subtrees of the ESU-tree in tasks, which are then individually executed by multiple threads.

The naive way of partitioning the ESU-tree would be to encapsulate each subtree rooted at depth 1 as a separate task, which is called a vertex-based enumeration. However, Shahrivari and Jalili have shown in their experiments [27], that this can yield extremely unbalanced tasks, such as a single task containing around 40 % of all candidates, which can significantly decrease the efficiency of parallelization.

The solution to the problem of unbalanced task for vertex-based enumeration is an edge-based enumeration. We move one step deeper in the ESU-tree and use a pair of vertices as a root of each task. This is equivalent to using each edge in an undirected graph G as a root for each task, which can be well seen in Figures 1.1 and 1.2.

The edge-based enumeration does not produce as heavy tasks as the vertex-based enumeration and therefore is less likely to cause performance problems for parallel enumeration.

In algorithm 1.13 we can see detailed description of *Parallel Subgraph Enumeration (PSE)* algorithm. First, a set of all edges (in case of directed graph, each bidirectional edge is present only once in the set) is created and then p threads enumerate all k-candidates of the graph.

In Algorithm 1.14, we can also notice a different approach to traversing the ESU-tree. Instead of utilizing a recursive function, Subdigger uses a stack for pairs (Sub, Ext), which basically simulates the implicit stack of recursive function calls. The problem of this solution is the last line of the algorithm. We can notice, that it is called in a loop and in case that we have a vertex of

Algorithm 1.13 PSE(G,k,p)

Input: A graph $G = (V, E)$, an integer $1 \leq k \leq |V|$ and an integer $1 \leq p$.
Output: A map of all k-candidates in G.
Procedure:

1. $Map := \emptyset$
2. $Q := \{(u, v) \in E \mid u < v \text{ or } (v, u) \notin E\}$
3. Create p threads
4. For each thread in parallel
    1. While $Q \neq \emptyset$
        1. Remove an arbitrary edge (v, w) from Q
        2. call ENUMERATE(G,k,v,w,Map) over a shared map Map
5. Wait for all p threads to finish
6. Return Map

Algorithm 1.14 ENUMERATE(G,k,v,w,Map)

Input: A graph $G = (V, E)$, an integer $1 \leq k \leq |V|$, a vertex $v \in V$, a vertex $w \in V$ and a shared map Map.
Output: A shared map Map with k-candidates in G.
Procedure:

1. Let $Stack$ be a stack of pairs (Sub, Ext), representing nodes of ESU-tree
2. If $v > w$ then swap v and w
3. $Ext := \{u \in N(\{v\}) \mid u > w\} \cup \{u \in N(\{w\}) \mid u \notin N(\{v\}), u > v\} \setminus \{v, w\}$
4. Push pair $(\{v, w\}, \{Ext\})$ into $Stack$
5. While $Stack$ is not empty
    1. Pop pair (Sub, Ext) from $Stack$
    2. If $|Sub| = k$
        1. $label := g[Sub]$
        2. If Map contains key $label$, then increment value of Map_{label}
        3. Else insert $(label, 1)$ into Map
    3. Else
        1. While $Ext \neq \emptyset$
            1. Remove an arbitrary vertex x from Ext
            2. $Ext' := Ext \cup \{u \in N(\{x\}) \mid u \notin N(Sub), u > v\}$
            3. Push pair $(Sub \cup \{x\}, Ext')$ into $Stack$

a high degree in graph G, we can easily create a set *Ext* of size close to the upper bound of $\mathcal{O}(n)$. Therefore, the last line of the algorithm will require the stack to provide up to $\mathcal{O}(n^2)$ memory space.

The classification phase of Subdigger is parallelized simply by applying ordered and canonical labeling by multiple threads over a shared input set.

Chapter 2

Analysis and design

In this chapter, we will introduce a new algorithm called *Efise* (EFficient Implementation of Subgraph Enumeration) for enumerating all k-candidates of the graph G. Efise is based of the ESU algorithm, but it includes various modifications to strongly improve the performance of the final program.

We introduce modifications not only to the ESU algorithm itself, but we also offer an efficient implementation, which is essential to achieve better performance than the programs described in the previous chapter.

Efise is focused on finding network motifs in large, directed graphs and same as FANMOD, it is limited to k-candidates with $k \leq 8$.

2.1 Edge query problem

When analyzing algorithms described in the previous chapter, we observed a common operation required in both ESU and Kavosh approaches.

Definition 5 (Edge query)**.** *Given a graph $G = (V, E)$ and an edge $(u, v) \in V^2$, determine whether $(u, v) \in E$.*

The edge query is required for building an adjacency matrix of a candidate $g[Sub]$. We are given a set of vertices Sub an need to determine the edges present in $g[Sub]$. From the definition of a vertex induced subgraph, we have $g[Sub] = (Sub, Sub^2 \cap E)$, where we can observe, why we need the edge query to build the adjacency matrix of $g[Sub]$.

The edge query is a frequented operation. From the analysis of algorithms in the previous chapter, we can observe two usual ways of building adjacency matrices of candidates.

1. Building the adjacency matrix from a complete set of vertices Sub. This requires $\Theta(k^2)$ edge queries per candidate.

2. Building the adjacency matrix continuously during the enumeration. In this approach, we need to perform an edge query between u and current vertices $v \in Sub$ each time a new vertex u is being added into Sub. This requires between $\Omega(k)$ and $\mathcal{O}(k^2)$ edge queries per candidate, as analyzed previously in 1.2.4.

Both approaches described above indicate, that the edge query is truly a frequented operation during the enumeration task. Therefore, it is reasonable to suggest, that an efficient implementation of edge query should have a significant impact on the overall performance of the program.

Let us describe a few possible implementations of the edge query.

- Representing graph G as an adjacency matrix. The positive side of this approach is obvious, $\mathcal{O}(1)$ time complexity with a very small hidden constant. The negative side is, of course, memory. Firstly, we need $\mathcal{O}(n^2)$ memory, which we may not have for large graphs. Secondly, this approach is very cache unfriendly, which may cause performance issues. Examples of this approach are Kavosh, FaSE and Subdigger.

- Creating a hash table of edges. This approach does not have neither a positive or a negative side. While the time complexity is $\mathcal{O}(1)$, we need to calculate the hash function, so the performance will likely be worse then in the previous approach. Memory requirements are only $\mathcal{O}(E)$, but there is no data locality even when we query edges of a single vertex u, hence the cache hit rate will suffer for large graphs. Examples of this approach are FANMOD and Subdigger.

- Representing graph G as a sorted adjacency list. In this approach, we perform the query for an edge (u, v) as a binary search on a sorted list of vertices adjacent to vertex u. This yields a time complexity of $\mathcal{O}(\log |N(\{u\})|)$, which is $\mathcal{O}(\log \frac{|E|}{|V|})$ on average. The space complexity is $\mathcal{O}(|E|)$, but the representation can also serve as the main representation of graph G, therefore no additional representation is needed. The positive side of this approach is data locality, when we query edges of a single vertex u. On the other hand, the time complexity

We have described three possible ways of implementing the edge query. For each of them, we have pointed out the possible downsides, which can harm the performance of the program. This leads us to conclusion, that the best option would be not to require the edge query at all.

2.1.1 Getting rid of the edge query

We will introduce a new method of finding edges of a candidate $g[Sub]$. The key property of our method is that it does not require the edge query.

First, we will introduce additional notation. For simplicity, we will not understand sets *Sub*, *Ext* and $N(\{v\})$, where $v \in V$ as lists. This will allow us to specify the order of elements in the list and use notation A_i to denote the i-th element of a list A and a function $R(v, A)$, where $R(v, A) = i$ for minimum i, such that $A_i = v$. We will maintain using set operations for lists, but we define a more specific operation $B \cup C$, such that $A = B \cup C$ for lists means, that list A will contain elements of lists B and C, where the ordering is preserved and additionally, the first element of C follows the last element of B. We also define a set Adj_v for each vertex $v \in V$ and an adjacency matrix M, representing candidate $g[Sub]$, where $M(i, j)$ denotes an i-th row and j-th column of matrix M.

We introduce the algorithm Efise, shown in Algorithm 2.1 and Algorithm 2.2, based on ESU.

Algorithm 2.1 EFISE(G,k)

Input: A graph $G = (V, E)$ and an integer $1 \leq k \leq |V|$.
Output: A map of all k-candidates in G.
Procedure:

1. Let M be an adjacency matrix filled with zeroes
2. $Map := \emptyset$
3. For each vertex $v \in V$
    1. $Ext := \{u \in N(\{v\}) \mid u > v\}$
    2. For each vertex $u \in Ext$, add v into Adj_u
    3. $Map :=$ ENUMERATE(G,k,v,$\{v\}$,Ext,Map,Adj,M)
    4. For each vertex $u \in Ext$, remove v from Adj_u
4. return *Map*

Lemma 2. *Before each call of* ENUMERATE *in Algorithm 2.2,*

$$Adj_w = N(\{w\}) \cap Sub.$$

Proof. We will show the equality in two steps.

- We will show, that for each $u \in (N(\{w\}) \cap Sub)$, $u \in Adj_w$. Because $u \in Sub$, we know, that in a recursive call at depth $d = R(u, Sub)$, u was added into *Sub*. Let us denote We will denote variables w, N and *Sub* at depth d as $w' = u$, N', Sub'.

  Because u is adjacent to w, then w is also adjacent to u. Because $w \notin Sub$, then $w \notin Sub'$. Therefore, according to the step 2.1.2. of Algorithm 2.2, $w \in N'$ and therefore at step 2.1.4., $w' = u$ was added into Adj_w.

- Now we will show, that also for each $u \in Adj_w$, $u \in (N(\{w\}) \cap Sub)$, i.e. $u \in N(\{w\})$ and $u \in Sub$. From steps 2.1.2. and 2.1.4. we can

Algorithm 2.2 Enumerate(G,k,v,$\{v\}$,*Ext*,*Map*,*Adj*,M)

Input: A graph $G = (V, E)$, an integer $1 \leq k \leq |V|$, a vertex $v \in V$, a set *Sub*, a set *Ext*, a map *Map*, sets *Adj* and an adjacency matrix M.
Output: An map of k-candidates in G.
Procedure:

1. If $|Sub| = k - 1$
   1. $LastAdj := \emptyset$
   2. For each $w \in Ext$
      1. If $Adj_w = LastAdj$ then increment last visited element in *Map*
      2. Else
         1. For each vertex $u \in Adj_w$, $M(R(w, Sub), R(u, Sub)) := 1$
         2. Increment Map_M
         3. $LastAdj := Adj$
         4. For each vertex $u \in Adj_w$, $M(R(w, Sub), R(u, Sub)) := 0$
2. Else
   1. While *Ext* is not empty
      1. Remove the first vertex w from *Ext*
      2. $N := \{u \in N(\{w\}) \mid u \notin N(Sub), u > v\}$
      3. $Ext' := Ext \cup N$
      4. For each vertex $u \in N$, add w into Adj_u
      5. For each vertex $u \in Adj_w$, set $M(|Sub|, R(u, Sub)) := 1$
      6. $Map :=$ Enumerate(G,k,v,$Sub \cup \{w\}$,Ext',*Map*,*Adj*,M)
      7. For each vertex $u \in Adj_w$, set $M(|Sub|, R(u, Sub)) := 0$
      8. For each vertex $u \in N$, remove w from Adj_u
3. return *Map*

simply observe, that for each $u \in Adj_w$, $u \in N(\{w\})$. Each vertex, that has been added into any *Adj* set in step 2.1.4. is always added into *Sub* in step 2.1.6. (also steps 3.2. and 3.3. in Algorithm 2.1). Therefore, if $u \in Adj_w$, then $u \in (N(\{w\}) \cap Sub)$. As only vertices being added into *Sub* are added into sets *Adj*, the also $u \in Sub$.

□

Using Lemma 2, we know that at step 2.1.5. we correctly build the i-th line of adjacency matrix M for $i = |Sub|$.

Note, that Algorithm 2.1 currently works for undirected graphs, where we only build the bottom left triangle of matrix M.

Let us mention a few observations made for the Efise algorithm.

- In spite of the fact, that Algorithm 2.1 only works for undirected graphs, we can add a direction information into *Adj* sets and easily modify the lines modifying matrix M to build a full matrix M for directed graphs.

- Sets Adj can be used to test a condition, whether a vertex $u \in N(Sub)$ at step 2.1.2. of Alg. 2.2, because $u \notin N(Sub)$ if and only if $Adj_u = \emptyset$.

- The ESU-tree may normally have many different forms for the same graph G and candidate size k, because of the arbitrary order in which vertices w are removed from Ext. However, at step 2.1.1. of Alg. 2.2, we define a fixed order of choosing vertices. Also at step 2.1.3. we have a fixed ordering of the new Ext', as Ext and N are also lists. Therefore, as long as we have a fixed ordering of vertices in $N(\{v\})$ for each $v \in V$, the Efise algorithm will always create the same ESU-tree.

2.2 Accelerating repetitive occurrences

We suggest, that candidates g_1 and g_2, where candidate g_2 is enumerated by Efise immediately after the candidate g_1, can often have the same adjacency matrix M.

To support our suggestion, we considered the following ideas.

- Let us denote the paths of candidates g_1 and g_2 in the ESU-tree as p_1 and p_2 respectively and the matrices of g_1 and g_2 as M_1 and M_2 respectively. If paths p_1 and p_2 differ only in the last node, then matrices M_1, M_2 of candidates g_1 and g_2 can differ only in the last row. This situation becomes more likely, if the share of leaves at depth k of the ESU-tree among all nodes of the ESU-tree is high. In Subsection 4.4.2 we show, that on our dataset the share of such leaves among all nodes is high.

- In large sparse networks, the number of edges in a candidates is usually small. We suggest a situation, where $|Adj_v| = 1$ for a vertex v, where $R(v, Sub) = k$. This means, that in candidate $g[Sub]$, where is only one vertex adjacent to v. This situation should be more likely for sparse networks.

We will now suggest a heuristic, that can accelerate the algorithm is situations described above. Let us outline a specific situation. We are traversing the ESU-tree and are currently located at depth $d = k-1$, where $|Sub| = k-1$. For simplicity, we assume that $|Adj_w| = 1$, for each $w \in Ext$. Because the order of vertices in Ext is fixed, we can observe than in this situation, sets Adj_v for each $v \in Ext$ in the corresponding order would form $k-1$ consecutive groups, where each group is defined by the same set Adj. For clarification, see Figure 2.1, where the groups are indicated by red separators.

Because we have already built $k-1$ rows of matrix M, we have only the k-row to complete. From step 2.1.5. of Alg. 2.2 we can deduce, that vertices $v \in Ext$ belonging to the same group would all form the same final matrix M.

To solve this situation efficiently, we offer the approach shown in step 1. of Algorithm 2.2, where we memorize the last Adj_w and in case the next one

	0	1	2	3	4	5	6	7	8
Ext	7	8	12	11	14	9	13	17	20
Adj_{Ext}	{3}	{3}	{3}	{6}	{6}	{4}	{4}	{4}	{4}

Figure 2.1: An example of *ext*∗ arrays while traversing ESU-tree in 1.2 for graph G in 1.1.

is the same, we do not explicitly complete the matrix M, but instead only increment the number of occurrences of the last candidate.

Let us now consider the same situation for a directed graph G. Consider two vertices $v_1, v_2 \in Ext$ in the same group of list *Ext* as described above, i.e. $Adj_{v_1} = Adj_{v_2}$, $v \in Adj_{v_1}$. The edge from v_1 to v (or edge from v_2 to v) can be of 3 types. In-edge, out-edge or bi-edge based on its direction. If the types of edges from vertices v_1 and v_2 differ, then the matrices M_1 and M_2 would also be different.

For undirected graphs, we have filled the last row of matrix M only once for each group. However, now we need to fill the last row each time the type of an edge changes in the group. To address this problem, we would like to reorder vertices in *Ext* in such a way, that each group would be sorted by edge types. If we analyze, how the groups in *Ext* are created, we discover that each of $k-1$ groups is actually a subset of list $N(\{w\})$, for one of the vertices $w \in Sub$ (see step 2.1.2. of Alg. 2.2). Therefore, we only need to sort lists $N(\{u\})$, for each $u \in V$ in the desired order.

Let us associate each vertex $v \in N(\{u\})$ with a pair (t, v), where t is type of the edge between u and v. We define an ordering of edge types in the following way.

$$\text{bi-edge } (\leftrightarrow) < \text{out-edge } (\rightarrow) < \text{in-edge } (\leftarrow)$$

Now we define ordering of pairs (t_1, v_1) and (t_2, v_2).

$$(t_1, v_1) < (t_2, v_2) \Leftrightarrow (t_1 < t_2 \vee (t_1 = t_2 \wedge v_1 < v_2))$$

With the defined ordering, we sort list $N(\{u\})$ for each $u \in V$.

In Figure 2.2, we have illustrated the same situation as in Figure 2.1 but for a directed case. We can observe groups of *Ext* to be further partitioned into subgroups. Each group is partition into at most 3 subgroups according to 3 types of edges. All vertices in the same subgroup would form the same matrix M.

To make use of the describe property, we stop the recursive calls of function Enumerate when $|Sub| = k - 1$, instead of $|Sub| = k$.

	0	1	2	3	4	5	6	7	8
Ext	8	7	12	14	11	17	9	13	20
$Adj_{Ext[i]}$	{3}	{3}	{3}	{6}	{6}	{4}	{4}	{4}	{4}
Direction	→	←	←	→	←	↔	→	→	→

Figure 2.2: An example of $ext*$ arrays while traversing ESU-tree in 1.2 for graph G in 1.1.

2.3 Classification

For the classification process, we use the same approach as Subdigger. However, for directed graphs, we can achieve different results than Subdigger because of the heuristic described in Section 2.2. The vertices in lists $N(\{u\})$ have different ordering and therefore for the same candidate $g[Sub]$, the corresponding initial matrix M may not be the same for Efise and Subdigger.

We assume, that because of the fixed ordering of lists $N(\{u\})$, the initial matrices of candidates may tend to be more similar, then they do without using the ordering. Therefore, the size of the initial map *Map* should be smaller. Experimental results of this assumption can be seen in Subsection 4.4.2.

Another difference from Subdigger is that Efise does not use NAUTY to perform canonical labeling of candidates. We use the lexicographically smallest adjacency matrix M of candidate $g[Sub]$ as its canonical label. We find the lexicographically minimal matrix M by generating adjacency matrices for all permutations of list *Sub* and selecting the lexicographically smallest one. The number of all permutations of *Sub* is $k!$, but as Efise supports only $k \leq 8$, $k!$ is tolerable. For generating permutations of *Sub*, we use Heap's algorithm [6].

2.4 Implementation

For the description of our implementation, we will assume directed graph G, because undirected graphs can be easily emulated in a directed case by making all edges bidirectional.

Graph G is represented by two arrays P and A. List $N(\{u\})$, $u \in V$ is stored as a continuous subarray of A, such that particular indexes are specified in array P, which stores indexes to A. With respect to the ordering of

lists $N(\{u\})$ proposed in Section 2.2, we define array P as a non-decreasing sequence and store the edges in the following way.

- Vertices $v \in N(\{u\})$ such that edge (u, v) is a bi-edge are stored in a subarray $A[P[3u] \dots P[3u+1]]$.
- Vertices $v \in N(\{u\})$ such that edge (u, v) is a out-edge are stored in a subarray $A[P[3u+1] \dots P[3u+2]]$.
- Vertices $v \in N(\{u\})$ such that edge (u, v) is a in-edge are stored in a subarray $A[P[3u+2] \dots P[3u+3]]$.

Array P requires $\mathcal{O}(|V|)$ memory and array A requires $\mathcal{O}(|E|)$ memory.

An important part of Efise are the Adj sets. Set Adj_u, $u \in V$ is stored as a single variable. Therefore, for each $u \in V$, sets Adj_u can be stored as an array $adj[1 \dots n]$ of size $\mathcal{O}(|V|)$. The value of $adj[u]$ is formally defined as

$$adj[u] = \sum_{v \in Adj_u} 2^{R(v,Sub)-1}.$$

To explain the definition, we say, that for each vertex $v \in Adj_u$, we set the i-th bit of $adj[u]$ to 1, where i is the order of v in Sub. We can observe, that for variable in array adj we need only k bits, which is for our restriction $k \leq 8$ only a single byte. However, in Subsection 2.1.1 we have mentioned, that for oriented graphs, we would need to add a direction information in addition to Adj sets. This is solved by using $2k$ bits, where the lower k bits represent out-edges and upper k bits represent in-edges. Obviously, a bi-edge is denoted by setting the corresponding bit in both the lower and the upper k bits. This doubles the number of bits we require, but asymptotically, array adj requires only $\mathcal{O}(|V|)$ memory.

Our program strongly utilizes bit operations. One of such situations is the implementation of step 2.1.4. in Algorithm 2.2. In this step, we are adding a vertex w into sets Adj_u, for each $u \in N$. Together with this step, steps 2.1.2. and 2.1.3. are also performed. We split the iteration into 3 loops, with respect to different edge types. Before each loop, we prepare a bit mask m, where we set a bit b, bit $b+k$, or both bits b and $b+k$ according to the type of edge we will process in the loop. Then we iterate the corresponding subarray of A in decreasing order, such that we stop the loop when the condition $u > v$ is no longer true. In each iteration, we test, if $u \notin N(Sub)$, which we perform in a single operation by testing if $adj[u] \neq 0$ as mentioned in Subsection 2.1.1. If the condition is true, we use a bitwise or to update variable adj_u with a mask m and add vertex u into Ext.

Step 2.1.8. of Algorithm 2.2 is performed similarly by using a complementary mask and a bitwise and operation.

The adjacency matrix M is represented as a 64-bit variable. Because of this representation, our implementation has the restriction to $k \leq 8$. The

reason why we chose such representation is, that we also need to perform bit operations with matrix M. In step 2.1.5. of Algorithm 2.2, we need to fill the i-th row of matrix M (and because we consider a directed graph, then also the i-th column), where $i = |Sub|$. For this task, we can use a precomputed 2D table, where we specify parameters i and Adj_w and get a 64-bit mask, which can fill the i-th row and column with a bitwise or operation. With this setting, the table would need to have k rows and 2^{2k} columns filled with 64-bit values, which results in 2^{22} bytes. This would not be very cache friendly, therefore we actually use a separate tables for filling rows and columns of matrix M. This way, both tables need only k rows and 2^k columns, which results in only 2^{15} bytes in total. With this setup, we only need to perform two table lookups and two bitwise or operations to update matrix M.

Step 2.1.7. of Algorithm 2.2 is performed in a similar way, only this time we just need a single 1D array of size k with precomputed masks to clear row and column for each i, $1 \leq i \leq k$ with a bitwise and operation.

For the classification part, *Map* is represented by a custom hash table, crafted to offer high performance specifically for our task. It also offers a pointer to the last incremented value, which we exploit as described in Section 2.2.

For the canonical labeling transformation, we utilize Heap's algorithm for generating all permutations of vertices. The naive approach would be to generate permutations of vertices and for each permutation build the matrix M. However, this would have the time complexity of $\mathcal{O}(k^2k!)$. Therefore, instead of swapping vertices at positions specified by Heap's algorithm, we directly swap the corresponding rows and columns of matrix M. We again utilize bit operations together with shifts to perform each swap in $\mathcal{O}(1)$ time and therefore we keep the complexity of finding canonical label $\mathcal{O}(k!)$. The Heap's algorithm is naturally recursive, but we do not use recursive calls and simulate the recursion with a loop and also unroll the last 2 levels of recursion, to improve performance.

2.4.1 Corollary

In response to the complexity analysis in Subsection 1.2.4, we provide the same analysis for our implementation. If we considered an arbitrary parameter k, we would come to the same complexities as we did in Subsection 1.2.4. However, if we assume the restriction of our implementation, which is $k \leq 8$, the results of the analysis are different. (Note, that the restriction $k \leq 8$ also applies for FANMOD. Therefore, we do not favor Efise.)

1. $R = \Theta(T)$
   In this case the complexity of our solution is $\Theta(R)$, in contrast to $\Theta(Rk)$ of FANMOD. As this case is the most common for real-life input graphs (see Subsection 4.4.2), this makes a significant difference.

2. $R = \Omega(\frac{T}{k})$
   In this case we also achieve better complexity of $\mathcal{O}(Rk)$ in contrast to $\mathcal{O}(Rk^2)$ of FANMOD.

3. $R = o(\frac{T}{k})$
   For the last situation, we yet again achieve better complexity of $\mathcal{O}(T)$ in contrast to $\mathcal{O}(Tk)$ of FANMOD.

Moreover, we did not count the used heuristic, which processes each of the repetitive nodes with $O(1)$ complexity for all of the cases above. The actual number of instruction for this operation is also very low, making the heuristic useful even for the first case, where we reach $O(1)$ complexity even without the heuristic.

Chapter 3

Parallelization

In Section 1.5, we have described a parallelization method used in Subdigger, which is based on a shared job queue. The ESU tree is partitioned into subtrees and each job in the queue represents one of the subtrees. After the queue is prepared, individual threads dynamically fetch jobs from the queue for processing.

Each job in the queue corresponds to a single subtree of the ESU-tree, rooted at depth 2. Because the ESU-tree is not balanced, the jobs are not balanced either.

This approach is somewhere between static and dynamic scheduling. We can not say, that it is static, because threads dynamically fetch jobs from the queue. However, it is not fully dynamic, because the queue is predetermined and the jobs are unbalanced.

Because of the insufficient balancing, this approach does not provide enough dynamic load balancing and Subdigger can struggle with low speedups for parallel execution.

In this chapter, we will introduce better, more dynamic scheduling to improve load balancing, such that we do not have to rely on a reasonably balanced ESU-tree.

3.1 Scheduling

In contrast to Subdigger, our parallelization method does not utilize a shared job queue. Instead, we first statically assign partitioned ESU tree to individual threads and then balance the load by using direct communication between threads.

3.1.1 Job

We introduce the term *Job*, which is defined as a triple $J = (R, S, X)$. Let us describe the individual items in the Job triple.

- $R \subseteq V$ is a set of vertices called *roots*,
- $S \subseteq V$ is a list of vertices,
- $X \subseteq V$ is a list of vertices.

Let us remind, that nodes of the ESU-tree are labeled by the pair of sets (*Sub*,*Ext*). However, each node can be unambiguously identified by the set *Sub* alone (see Figure 1.2 for clarification).

The job J unambiguously identifies a set of subtrees D of the ESU-tree, which should be processed. The set D identified by job J is defined in the following way.

1. If $R \neq \emptyset$, then D is a set of subtrees rooted at such nodes m of the ESU-tree, where the set *Sub* in the label of m is equal to $\{v\}$, for each $v \in R$.
2. If $R = \emptyset$, then D is a set of subtrees rooted at such nodes m of the ESU-tree, where the set *Sub* in the label of m is equal to $S \cup \{v\}$, for each $v \in X$.

3.1.2 Parallel Efise

We modify Algorithm 2.1, such that it accepts an additional parameter R. Step 3. of the algorithm is modified such that we iterate over vertices $v \in R$ instead of $v \in V$.

Now, we introduce a parallel version of Efise algorithm, shown in Algorithm 3.1.

In step 4.2.3. of Algorithm 3.1, thread t_1 searches for a new job from other threads. When thread t_2 receives a job request from thread t_1, it has to split its job and provide a part of it to thread t_1. The procedure of splitting a job to provide a new job is described in Algorithm 3.2.

Theorem 3. *Algorithm 3.1 performs $\mathcal{O}(pk \log n)$ job exchanges between threads.*

Proof. From Algorithm 3.2, we can observe, that a job J is being reduced by requests from other threads in a way that can by simplified as shown in Algorithm 3.3.

If we analyze Algorithm 3.3 We can observe, that step 1. requires $\log |R|$ requests. Step 3.1. requires $\log |X|$ requests and is repeated $k-1$ times in step 3., therefore we require total of $\log |R| + (k-1) \log |X|$ requests to reduce job J into an empty job.

Let $C(J)$ be the number of requests required to reduce the job J. We refer to $C(J)$ as *difficulty of* J. The worst case giving the greatest difficulty is $|R| = n$ and $|X| = n$. This forms the difficulty $C(J) = k \log n$.

Algorithm 3.1 ParallelEfise(G,k,p)

Procedure:

1. Partition set V into p subsets denoted as V_i, $1 \leq i \leq p$
2. Create p empty maps denoted as Map_i, $1 \leq i \leq p$
3. Create p threads
4. For each thread t with an identifier i, run in parallel
    1. $J := (V_i, \emptyset, \emptyset)$
    2. While $J \neq \emptyset$
        1. If $R \neq \emptyset$
            1. Map_t := Efise(G,k,Map_t,J)
        2. Else
            1. Build sets Adj and an adjacency matrix M for candidate $g[S]$
            2. Map_t := Enumerate(G,k,v,S,X,Map_t,Adj,M)
        3. Ask for a job from other threads and store it in J
5. Wait for all threads
6. Run parallel reduction of all maps Map into one
7. Divide items of Map equally between p threads and perform ordered labeling on each key in Map
8. Run parallel reduction of Map to merge items with equal keys
9. Divide items of Map equally between p threads and perform canonical labeling on each key in Map
10. Run parallel reduction of Map to merge items with equal keys
11. Return Map

Algorithm 3.2 ProvideJob()

Output: A new job extracted from the current job.

Procedure:

1. Let J be a job
2. If R in EfiseParallel is empty
    1. Choose $R_1 \subseteq R$, such that $|R_1| = \lceil |R|/2 \rceil$
    2. $R := R \setminus R_1$
    3. Return job $(R_1, \emptyset, \emptyset)$
3. Else
    1. Find minimum d, $1 \leq d \leq k-1$, such that at depth d, $Ext \neq \emptyset$
    2. If no such d exists, then return $\emptyset$
    3. Let S_1 be the list Sub at depth d
    4. Let X_1 be the last $\lceil |Ext|/2 \rceil$ of list Ext at depth d
    5. Remove X_1 from Ext
    6. Return job $(\emptyset, S_1, X_1)$

Algorithm 3.3
Procedure:
1. While $R \neq \emptyset$ remove half of R
2. $|S| := 1$
3. While $|S| < k$
    1. While $X \neq \emptyset$, remove half of X
    2. Increment $|S|$

Let us assume the worst situation, where we have p threads with jobs of difficulty $C(J) = k \log n$. We will deduce the maximum number of job exchanges required to reduce all p jobs to difficulty $C(J) = 0$.

To perform a job exchange, we first need a thread t_1 to finish its job, such that it will request a new job from another thread. To achieve the worst case, we will assume, that except thread t_1 all other threads do not execute, so they do not reduce the difficulty of their job themselves.

When thread t_1 finishes its original job J_1, it requests a new job from thread t_2. Let J_2 and be the job of t_2. Then, after the job exchange, the job J_2 would be divided into two jobs J_1' and J_2', belonging to threads t_1 and t_2, respectively, such that $C(J_1') = C(J_2') = C(J_2) - 1$.

We can see, that the difficulty of job belonging to thread t_2 was decremented by 1. The difficulty of the original job J_1 belonging to thread t_1 in comparison to its new job was decreased by $C(J_1) - (C(J_2) - 1)$. We can see, that the difficulty may actually even increase. To minimize the decrease, we will choose t_1 such that the difficulty of its job is minimal among all threads and thread t_2 such that its difficulty is maximal among all threads.

Using this strategy, we can observe, that from the initial state, where all threads have the same difficulty C, the difficulty of all threads will decrease by 1 in $p-1$ job exchanges. This means, that all jobs will be reduced to difficulty 0 in $(p-1)k \log n = \mathcal{O}(pk \log n)$ job exchanges.

□

3.2 GPU

We have considered the possibility of utilizing GPUs for finding network motifs. However, when adapting the Efise algorithm to the GPU architecture, we ran into following problems.

- The Efise algorithm requires $\mathcal{O}(n)$ memory per thread for *Ext* set and *Adj* sets. Considering that current GPUs have thousands of cores, this could lead to significant memory problems, because the amount of memory available for each thread is for such large number of threads very limited. Therefore, for large graphs, we would probably not be even able to fit the necessary arrays into memory.

- Efise uses hash tables to store occurrences of candidates. Unfortunately, hash tables are not a suitable data structure for the GPU architecture, especially when dynamic resizing is required.

- The ESU-tree tends to be strongly unbalanced (see experiments in MR-SUB [27]). The proposed dynamic scheduling in parallel Efise algorithm can deal with this problem very well. However, the architecture of GPUs is not designed for dynamic scheduling based on communication between threads and this concept is not applicable. Because of this, we would struggle with performance issues and we would not be able to fully exploit the potential of GPUs.

We were able to design a memory efficient representation of the set Ext, where we do not store the set explicitly, but rather simulate its creation for each iteration over the set, which would require the amount of memory polynomial with k. The Adj sets can be omitted by using the original ESU algorithm. This would resolve the memory problems, but only for the cost of highly inefficient implementation, which would make the implementation useless in practice.

Hash tables can be replaced with sorted arrays, which are better suited for GPUs including the efficiency of related parallel algorithms on GPU.

Eventually, we found out that the problem of unbalanced ESU-tree and the subsequent need of dynamic scheduling is very hard to solve and we were not able to find an acceptable solution for this problem.

We concluded, that the problem of adapting existing algorithms for finding network motifs to GPU architecture is not trivial. A potential algorithm for finding network motifs with GPU would probably require a completely different approach, which may also be the reason why we were unable to find any research with a solution to this problem.

Chapter 4

Measuring and results

4.1 Testing environment

For our experiments, we used computational server *STAR* provided by CTU. During our experiments we have used 3 different setups both running OS Linux.

- For Comparison with other tools we used a setup with 4-core Intel i7 950 @ 3.07 GHz and 24 GB of RAM.
- For measuring scalability of our algorithm, we used a 12-core machine with 2× 6-core Xeon 2620 v2 @ 2.1 GHz processor and 32 GB of RAM.
- For measuring scalability on large number of cores, we used a specialized coprocessor Intex Xeon Phi 7120 [7] with 61 cores @ 1.24 GHz and 16 GB of RAM.

Unfortunately, we were unable to run FANMOD on our setup due to a missing graphical library. We refer to Subdigger [29], where FANMOD shows similar performance as Kavosh.

4.2 Measuring methodology

For measuring execution times, we have performed three executions for each measurement. Because all algorithms are deterministic, three executions are sufficient. The final execution time T was calculated as an average of all execution times. Formally, $T = \frac{1}{3}\sum_{i=0}^{3} t_i$, where t_i is the i-th execution time.

4.3 Dataset

As a benchmark dataset, we have selected mostly networks used in previous works about network motifs. Selected networks come from various fields like

social networks, biology, communication, web graphs, peer-to-peer networks and banking. In total, we have used seven different directed networks: Elegans (neuronal network of Caenorhabditis elegans [12]), Jazz (network of jazz musicians [5]), School (face to face contact patterns in a primary school [30]), Gnutella (structure of Gnutella p2p network from August 31, 2002 [16]), Slash (slashdot social network from February 2009 [15]), Notre (web graph of Notre Dame [1]) and Fraud (own network assembled from banking transactions).

Table 4.1: Properties of networks in dataset.

Parameter	Network						
	Elegans	Jazz	School	Gnutella	Slash	Notre	Fraud
Vertices	297	198	238	62,586	82,168	325,729	125,259
Edges	2,345	2,742	5,539	147,892	948,464	1,497,134	103,324
Average degree	15.79	27.70	46.55	4.73	23.09	9.19	1.65
Density	0.0267	0.0703	0.0982	3.78e−5	1.40e−4	1.41e−5	6.59e−6

Table 4.1, shows the main properties of used networks. We the first three networks are small with hundreds of vertices and thousands of edges. The next four networks are large with number of vertices between tens of thousands to hundreds of thousands and number of edges between hundreds of thousands to millions.

4.4 Results

4.4.1 Growth of the number of candidates

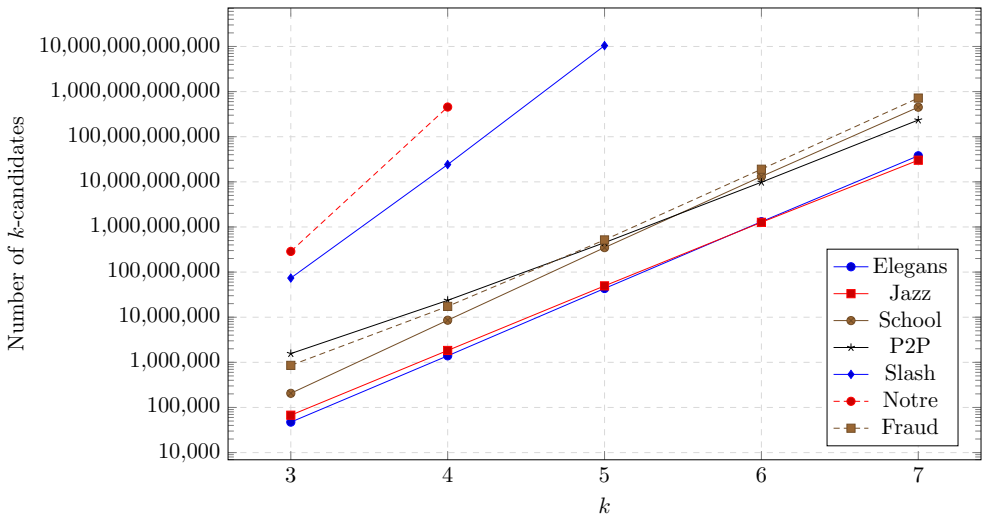


Figure 4.1: Number of k-candidates in graphs for different k.

We have previously proposed, that number of k-candidates in graph G grows exponentially with k. In Figure 4.1 we can observe, that this proposition is true for graphs in our dataset.

4.4.2 General properties

In Tables 4.2 and 4.3, we can observe multiple properties.

- In Subsection 1.2.4 we proposed, that the number of leafs in the ESU-tree at depth k (number of candidates) is usually linear with the total number of nodes in the ESU-tree. We can see that for our dataset the number of candidates is usually well over 90 % of total number of nodes.
- In Section 2.2 we proposed, that the number of adjacency matrices Efise has to calculate completely could be noticeably smaller, then the number of candidates. This has also shown to be true for networks in our dataset. Efise calculates on average between 1 out of 2.5 matrices and 1 out of 30 matrices, depending on the specific network.
- We also proposed, that the heuristic in Section 2.2 could reduce the number of different original matrices found by the enumeration phase. The difference can be seen between parameters "G-Trie nodes" and "Original labels (Efise)". The difference is not significant, but it is noticeable.
- The heuristic mentioned in the previous paragraph also influences the number of orderedl labeled created from the original labels, which seems to be slightly in favor of Efise compared to Subdigger.
- In Subsection 1.4.4 we proposed, that the similarly to the ESU-tree, the number of leafs in the G-Trie could be usually linear with the total number of nodes in the G-Trie. This seems to be true for our dataset.

Parameter	Network, Candidate size		
	Elegans, 6	Jazz, 6	School, 6
ESU-tree nodes	1,354,007,453	1,318,357,689	13,498,005,446
ESU-tree leaves (candidates)	1,309,307,357	1,266,953,062	13,140,615,595
Completed matrices (Efise)	390,018,407	507,390,387	4,912,880,740
G-Trie nodes	2,577,754	13,656	13,656
G-Trie leafs	2,499,645	13,144	13,144
Original labels (Efise)	1,453,569	13,144	13,144
Ordered labels (Efise)	394,987	7,099	7,099
Ordered labels (Subdigger)	416,083	9,820	9,820
Canonical labels	286,376	5,647	5,647

Table 4.2: Various parameters measured for small networks.

Parameter	Network, Candidate size		
	Gnutella, 6	Fraud, 6	Slash, 4
ESU-tree nodes	10,281,594,263	19,572,019,933	24,234,045,702
ESU-tree leaves (candidates)	9,806,726,769	19,037,020,269	24,159,680,898
Completed matrices (Efise)	1,832,970,543	772,183,732	798,833,196
G-Trie nodes	42,575	3,868	2,310
G-Trie leafs	39,216	3,004	2,261
Original labels (Efise)	24,953	1,485	1,073
Ordered labels (Efise)	4,621	513	251
Ordered labels (Subdigger)	5,137	659	1,144
Canonical labels	2,714	375	198

Table 4.3: Various parameters measured for large networks.

4.4.3 Scalability

To measure the scalability of Efise, we measured the execution times of Efise for different number of threads to calculate the achieved speedup. The experiments were performed on a 12-core machine and a specialized coprocessor with 61 cores.

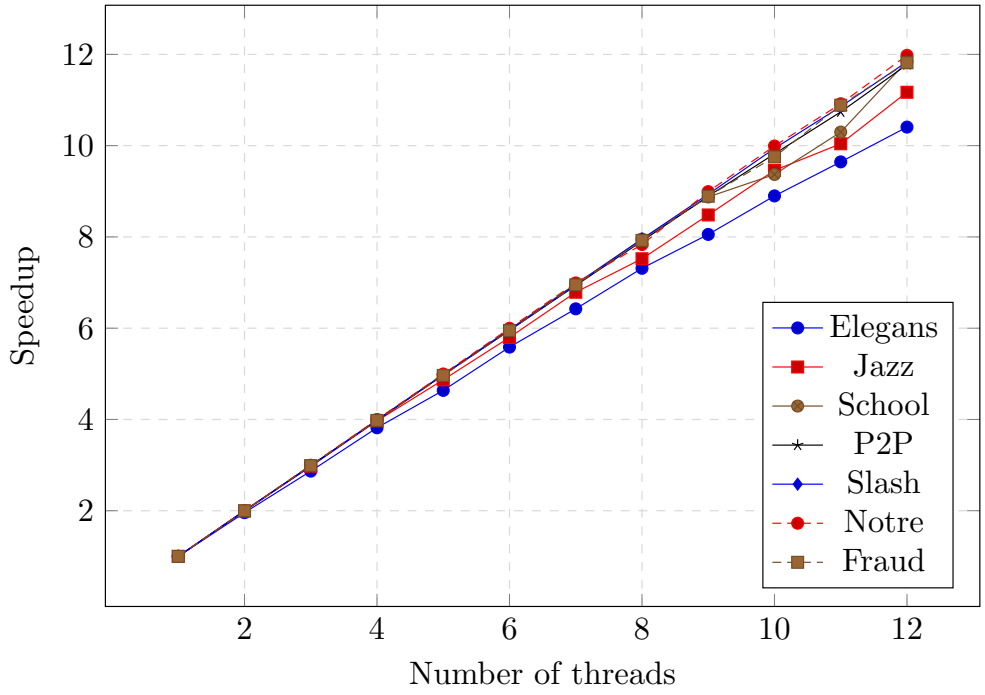


Figure 4.2: Scalability of Efise measured on a 12-core machine.

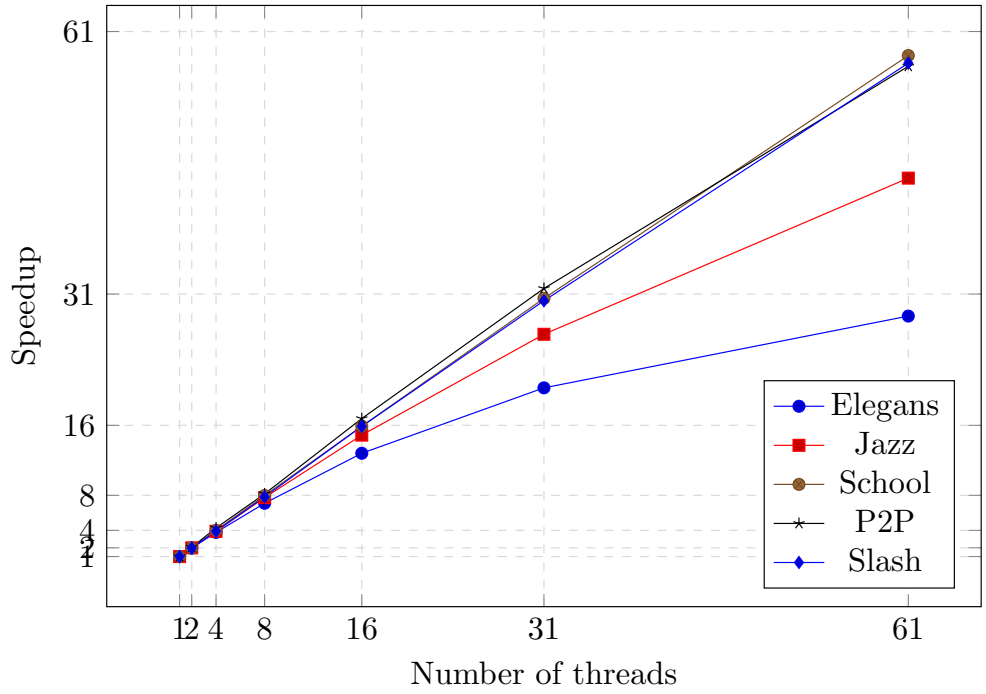


Figure 4.3: Scalability of Efise measured on 61-core Xeon Phi.

In Figures 4.2 and 4.3 we can see the achieved speedup. Results show that for most networks Efise achieves linear speedup, which proves a very good scalability of Efise.

4.4.4 Performance comparison with other tools

We have compared the performance of Kavosh, FaSE, Subdigger and Efise on our dataset. We measured execution times for different size k of candidates among all networks.

Each program was given 2 hours of execution time on the 4-core machine for each measurement. If the program did not finish within 2 hours, it was terminated and its result is denoted as TLE (time limit exceeded). In case the program ran out of memory before it finished execution or was terminated, its result is denoted as OOM (out of memory).

For Efise and Subdigger, we tested both serial and parallel performance. For parallel execution, we ran 8 threads, therefore the programs may reach speedups over 4 by using hyper-threading.

Results of the experiments in Tables 4.4 and 4.5 show, that Efise is significantly superior to all other programs. For parallel execution, Efise achieves 10 to 100 times better execution times than Subdigger. For serial execution,

Network	Tool	candidate size			
		4	5	6	7
Gnutella	Efise	0.21	1.67	21.9	485
	Subdigger	2.4	28.4	767	TLE
	Serial Efise	0.33	5.4	110	2493
	Serial Subdigger	10.5	127	3294	TLE
	Kavosh	18	554	TLE	TLE
	FaSE	3.0	51	1130	TLE
Slash	Efise	23.5	TLE	TLE	TLE
	Subdigger	1342	TLE	TLE	TLE
	Serial Efise	101	TLE	TLE	TLE
	Serial Subdigger	5824	TLE	TLE	TLE
	Kavosh	TLE	TLE	TLE	TLE
	FaSE	2028	TLE	TLE	TLE
Notre	Efise	237	TLE	TLE	TLE
	Subdigger	TLE	TLE	TLE	TLE
	Serial Efise	944	TLE	TLE	TLE
	Serial Subdigger	LTE	TLE	TLE	TLE
	Kavosh	OOM	OOM	OOM	OOM
	FaSE	OOM	OOM	OOM	OOM
Fraud	Efise	0.1	0.5	14.9	557
	Subdigger	6.5	39.7	2193	TLE
	Serial Efise	0.1	1.9	68	2731
	Serial Subdigger	21.6	103	3977	TLE
	Kavosh	16.6	812	TLE	TLE
	FaSE	0.86	24.1	1005	TLE

Table 4.4: Execution times is seconds for different tool on large networks.

FaSE seems to deliver better performance than Subdigger, but Efise offers even 5 to 10 times better performance than FaSE.

In Figure 4.4, we can see performance of FaSE, serial Efise and parallel Subdigger and Efise on some networks measured in number of processed candidates per second. We can clearly see, how even serial Efise considerable outperforms FaSE and even parallel Subdigger. Parallel Efise absolutely dominates with hundreds of millions of candidates per second and even about 1.3 billion candidates per second for the Fraud network. Considering the processor frequency of about 3 GHz, we can see that Efise processes roughly about a single candidate per 50 clocks, or even a candidate per 10 clocks for Fraud network. This is an outstanding performance, which probably can not be much more improved.

Network	Tool	candidate size 5	6	7
Elegans	Efise	0.23	4.3	210
	Subdigger	2.0	52	1583
	Serial Efise	0.58	20.1	940
	Serial Subdigger	4.2	134	4932
	Kavosh	57	2934	TLE
	FaSE	2.4	86	OOM
Jazz	Efise	0.21	4.2	107
	Subdigger	2.1	49	1267
	Serial Efise	0.89	22.3	567
	Serial Subdigger	5.3	150	4709
	Kavosh	46	1604	TLE
	FaSE	3.1	93	2783
School	Efise	1.1	41.1	1520
	Subdigger	12.5	496	TLE
	Serial Efise	5.6	223	TLE
	Serial Subdigger	34.5	1723	TLE
	Kavosh	309	TLE	TLE
	FaSE	21	970	TLE

Table 4.5: Execution times is seconds for different tool on small networks.

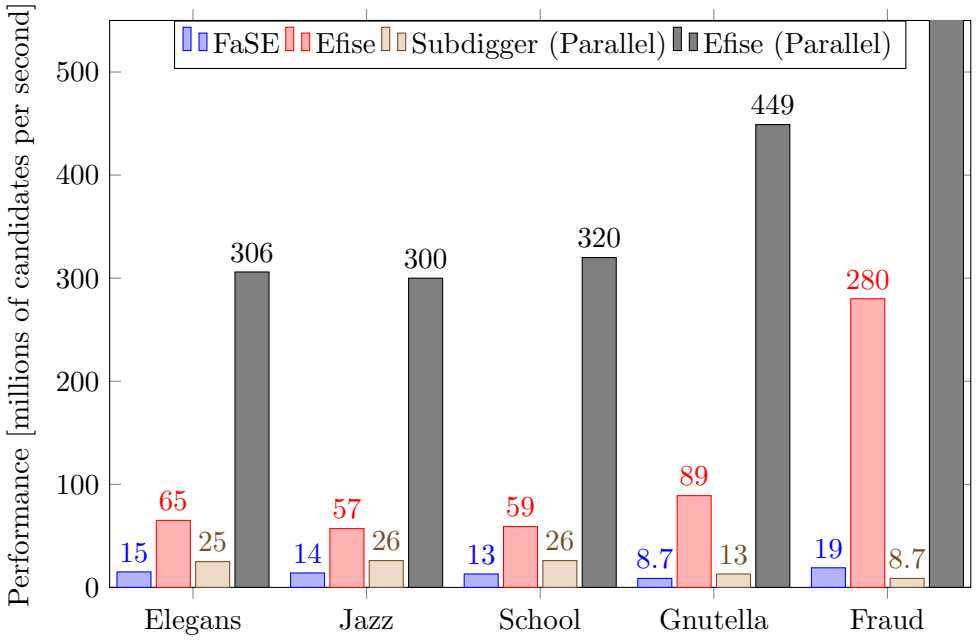


Figure 4.4: Performance comparison of Efise, FaSE and parallel Subdigger.

Conclusion

We reviewed the original algorithms for finding network motifs, where the basic principles of finding network motifs were described. Then, we surveyed recent state-of-the-art programs to understand the modifications leading to better performance. We have also analyzed their implementations to find performance bottlenecks.

We designed a new parallel algorithm called Efise, which was crafted from previous algorithms. We have applied modifications in 4 main categories, which are

- general algorithm design,
- data representation,
- heuristics for practical performance,
- dynamic scheduling for parallel execution.

We implemented Efise in the C++ language, which allowed us to include many optimization techniques and achieve better performance.

We ran performance benchmarks on our experimental dataset. Tests have shown, that Efise offers significant improvement over previous tools. Efise outperforms other tools (including parallel tools) already in sequential execution. On top of the outstanding sequential performance, Efise achieves linear speedups in parallel execution, which makes it an order of magnitude faster than current state-of-the-art programs.

Future work

We found that the adoption of current methods for finding networks motifs to GPU architecture is not trivial and a completely different approach would probably have to be designed to allow an efficient implementation.

Therefore, there is a possibility to further explore the problem with respect to the GPU architecture. This research could lead to even faster ways of finding network motifs by exploiting the processing power of GPUs.

Bibliography

[1] Albert, R., Jeong, H., and Barabási, A. The diameter of the world wide web. *CoRR cond-mat/9907038* (1999).

[2] Batagelj, V., and Mrvar, A. *Pajekanalysis and visualization of large networks*. Springer, 2004.

[3] Bender, E. A. The asymptotic number of non-negative integer matrices with given row and column sums. *Discrete Mathematics 10*, 2 (1974), 217–223.

[4] Bender, E. A., and Canfield, E. R. The asymptotic number of labeled graphs with given degree sequences. *J. Comb. Theory, Ser. A 24*, 3 (1978), 296–307.

[5] Gleiser, P. M., and Danon, L. Community structure in jazz. *Advances in Complex Systems 6*, 4 (2003), 565–574.

[6] Heap, B. R. Permutations by interchanges. *The Computer Journal 6*, 3 (1963), 293–298.

[7] Intel. Intel xeon phi product family. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[8] Jaimovich, A., Elidan, G., Margalit, H., and Friedman, N. Towards an integrated protein-protein interaction network: a relational markov network approach. *Journal of Computational Biology 13*, 2 (2006), 145–164.

[9] Jeong, H., Mason, S. P., Barabási, A.-L., and Oltvai, Z. N. Lethality and centrality in protein networks. *Nature 411*, 6833 (2001), 41–42.

[10] Jeong, H., Tombor, B., Albert, R., Oltvai, Z. N., and Barabási, A.-L. The large-scale organization of metabolic networks. *Nature 407*, 6804 (2000), 651–654.

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

[12] Kashtan, N., Itzkovitz, S., Milo, R., and Alon, U. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics 20*, 11 (2004), 1746–1758.

[13] Khakabimamaghani, S., Sharafuddin, I., Dichter, N., Koch, I., and Masoudi-Nejad, A. Quatexelero: an accelerated exact network motif detection algorithm. *PloS one 8*, 7 (2013), e68073.

[14] Kreher, D. L., and Stinson, D. R. Combinatorial algorithms: generation, enumeration, and search. *SIGACT News 30*, 1 (1999), 33–35.

[15] Leskovec, J., Huttenlocher, D. P., and Kleinberg, J. M. Predicting positive and negative links in online social networks. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010* (2010), pp. 641–650.

[16] Leskovec, J., Kleinberg, J. M., and Faloutsos, C. Graph evolution: Densification and shrinking diameters. *TKDD 1*, 1 (2007).

[17] Li, X., Stones, D. S., Wang, H., Deng, H., Liu, X., and Wang, G. Netmode: network motif detection without nauty. *PloS one 7*, 12 (2012), e50093.

[18] Maslov, S., and Sneppen, K. Specificity and stability in topology of protein networks. *Science 296*, 5569 (2002), 910–913.

[19] McKay, B. D., and Piperno, A. Practical graph isomorphism, II. *CoRR abs/1301.1493* (2013).

[20] Milo, R., Itzkovitz, S., Kashtan, N., Levitt, R., Shen-Orr, S., Ayzenshtat, I., Sheffer, M., and Alon, U. Superfamilies of evolved and designed networks. *Science 303*, 5663 (2004), 1538–1542.

[21] Milo, R., Kashtan, N., Itzkovitz, S., Newman, M. E., and Alon, U. On the uniform generation of random graphs with prescribed degree sequences. *arXiv preprint cond-mat/0312028* (2003).

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

[23] Paredes, P., and Ribeiro, P. Towards a faster network-centric subgraph census. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (New York, NY, USA, 2013), ASONAM '13, ACM, pp. 264–271.

[24] Ribeiro, P., and Silva, F. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery 28*, 2 (2014), 337–377.

[25] Schreiber, F., and Schwobbermeyer, H. Towards motif detection in networks: frequency concepts and flexible search. *Proc. Intl. Wsh. Network Tools and Applications in Biology (NETTAB04)* (2004), 91–102.

[26] Schreiber, F., and Schwöbbermeyer, H. Mavisto: a tool for the exploration of network motifs. *Bioinformatics 21*, 17 (2005), 3572–3574.

[27] Shahrivari, S., and Jalili, S. Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing 97*, 11 (2015), 1101–1120.

[28] Shahrivari, S., and Jalili, S. Fast parallel all-subgraph enumeration using multicore machines. *Scientific Programming 2015* (2015), 901321:1–901321:11.

[29] Shahrivari, S., and Jalili, S. High-performance parallel frequent subgraph discovery. *The Journal of Supercomputing 71*, 7 (2015), 2412–2432.

[30] Stehlé, J., Voirin, N., Barrat, A., Cattuto, C., Isella, L., Pinton, J., Quaggiotto, M., den Broeck, W. V., Régis, C., Lina, B., and Vanhems, P. High-resolution measurements of face-to-face contact patterns in a primary school. *CoRR abs/1109.1015* (2011).

[31] Wernicke, S. Efficient detection of network motifs. *IEEE/ACM Trans. Comput. Biol. Bioinformatics 3*, 4 (Oct. 2006), 347–359.

Appendix A

Acronyms

CTU Czech Technical University

GPU Graphics processing unit

OOM Out of memory

TLE Time limit exceeded

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `Efise` ..... implementation sources
  - `dataset` ..... benchmark dataset
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format