Optimization Letters
https://doi.org/10.1007/s11590-023-02001-z

ORIGINAL PAPER

On linear algebraic algorithms for the subgraph matching problem and its variants

Maxim D. Emelin[1] · Ilya A. Khlystov[2] · Dmitry S. Malyshev[3] · Olga O. Razvenskaya[4]

Received: 11 January 2023 / Accepted: 27 March 2023

Abstract
For a given simple data graph G and a simple query graph H, the subgraph matching problem is to find all the subgraphs of G, each isomorphic to H. There are many combinatorial algorithms for it and its counting version, which are predominantly based on backtracking with several pruning techniques. Much less is known about linear algebraic (LA, for short), i.e., adjacency matrix algebra, algorithms for this problem. Revisiting old ideas of J. Nešetřil and S. Poljak, which reduce the general case to the case of clique-queries, and updating them, we present the first LA algorithm for the subgraph matching/counting problem. For the k-clique matching/counting problem, we present static and dynamic LA algorithms, which may be of independent interest. For the k-clique counting problem, we also provide results of computational experiments of our solver with some large graphs and several k, which speed up results of several recent solvers for it.

Keywords Subgraph matching/counting problem · k-clique matching/counting problem · Linear algebraic algorithm

1 Introduction

All graphs, considered in this paper, are looples, without multiple edges, non-oriented or partially oriented graphs. Graphs of the first type are called *simple*, and second-type graphs are called *mixed*.

Let G and H be simple graphs. For the pair (G, H), the *subgraph matching problem*, briefly called the SM problem, is to find all the subgraphs of G, each of which is isomorphic to H. The *subgraph counting problem* for (G, H) is to determine the quantity of these subgraphs. Sometimes, G is called a *data graph* and H is called a *query graph*.

✉ Dmitry S. Malyshev
dsmalyshev@rambler.ru

Extended author information available on the last page of the article

Published online: 09 April 2023

The SM problem and its counting variant are fundamental with numerous applications in network science, including social network analysis and bioinformatics. In protein research, the physical contacts between proteins in the cell are represented as a network, and this protein-protein interaction network (PPIN) helps to develop new drugs. Large PPINs may contain millions of interactions, while they usually contain repeated local structures. Finding and counting these subgraphs is essential to compare different PPINs. In social network analysis, graph sizes could even reach trillion of edges, where a subgraph could be a group of users, sharing specific interests. Studying these groups improves the design of social networks and searching algorithms in them.

Triangles (or 3-*cliques*) and, more generally, *k-cliques*, i.e., sets of k pairwise adjacent vertices, are important types of subgraphs, arising in applications and being basic structures for matching/counting more complex fragments. For example, triangles counting is used in computing the local clustering coefficient, which is an important measure of the ability for nodes to form clusters [37].

All the algorithms for the SM problem can be classified by the execution model and the type of data graphs reading. *Sequential algorithms* run on a single processing machine only, but *parallel algorithms* can be executed on multiple simultaneously working processors. A *full computational* (or *static*) algorithm is an algorithm, where the data graph is explicitly given at once. In *incremental* algorithms, the data graph is accumulated by arriving its previously unknown simple parts, called *batches*. In *fully dynamic* algorithms, the vertex set is fixed and edges can be both added and deleted. In *batch-dynamic* algorithms, the set of vertices is not fixed, but batch updates can be both for insertions and deletions of edges.

There are several full computational sequential and parallel algorithms for matching/counting cliques [8, 12, 16, 19, 23, 24, 29–31, 33, 36]. For some other types of queries, like small-size subgraphs, paths, cycles and etc., efficient full computational sequential and parallel algorithms have also been developed, see [1, 3, 7, 9, 13, 22, 32, 35]. There are several efficient algorithms, see [4, 10, 11, 14, 15, 20, 27], for dynamically given data graphs and some types of queries.

In many applications of the SM problem, the sizes of data graphs are huge. Therefore, the use of parallel rather than sequential computations is the only way to obtain a result in reasonable time. Using the linear algebraic (briefly, LA) approach, i.e., the only matrix–vector instructions, is a natural choice to organize parallelism efficiently. Indeed, LA algorithms can be easily implemented, parallelized and have a small loss of performance under their scalability.

Only a few LA algorithms for the SM problem are known. For all the 4-vertex queries but the 4-clique, full computational LA algorithms have been developed in [18]. Exploiting the idea of color coding from [2], a full computational LA algorithm was proposed in the paper [6], when H is a tree. Apparently, according to ideas from [5], the approach from [6] can be extended to queries with tree-width at most 2.

The old paper [30] by J. Nešetřil and S. Poljak contains a reduction of the SM problem to its subproblem, where the only clique-queries are considered. This reduction is completely combinatorial. It has two drawbacks. The first of them is the absence of a bijection between solutions of the original and reduced SM problem.

The second one is memory overflow, because of the simultaneous use of the data graph and its complement graph. In this paper, we present the first LA version of the Nešetřil-Poljak's reduction, correcting these drawbacks.

Additionally, the paper [30] also presents a full computational algorithm for matching/counting k-cliques, which is almost completely combinatorial, except that it uses the fast matrix multiplication to match/count triangles in some graphs. In this article, we give a completely LA full computational version of their algorithm. Moreover, for matching/counting k-cliques, we present another full computational LA algorithm and an incremental LA algorithm, which have several handles for tuning their performance. For the k-clique counting problem, we provide results of computational experiments of our full computational solver for some large graphs and several k, which speed up results of several recent solvers for it.

2 Some definitions and notations

As usual, for sets A and B, by $A \cap B, A \cup B, A - B, A \times B$, we denote their intersection, union, difference, Cartesian product, respectively. For any set A, the cardinality of A is denoted by $|A|$. For any natural n, the notation $[n]$ means the set $\{1, 2, \dots, n\}$.

In this paper, we consider real-valued matrices only. For matrices $\mathbf{A}$ and $\mathbf{B}$ of the corresponding sizes, by

$$\mathbf{A} + \mathbf{B}, \mathbf{A} \cdot \mathbf{B}, \mathbf{A} \otimes \mathbf{B}, \mathbf{A} \circ \mathbf{B}$$

we denote their sum, the (usual) product, Kronecker product, Hadamard product, respectively. If $\mathbf{A}$ and $\mathbf{B}$ are binary matrices of the corresponding sizes, then $\mathbf{A} \vee \mathbf{B}, \mathbf{A}|\mathbf{B}, \mathbf{A} \bullet \mathbf{B}$ mean the sum, difference, and product over the logical semiring, i.e., we have

$$(\mathbf{A} \vee \mathbf{B})_{ij} = (\mathbf{A})_{ij} \vee (\mathbf{B})_{ij}, (\mathbf{A}|\mathbf{B})_{ij} = (\mathbf{A})_{ij}|(\mathbf{B})_{ij}, (\mathbf{A} \bullet \mathbf{B})_{ij} = \bigvee_k ((\mathbf{A})_{ik} \wedge (\mathbf{B})_{kj}).$$

Let $\mathbf{A}$ be a matrix, subsets I and J be some sets of its rows and columns. By $\mathbf{A}^T$, we denote the transposed matrix of $\mathbf{A}$. The notation $\mathbf{A}[I][J]$ means the submatrix of $\mathbf{A}$ with rows from I and columns from J. If I coincides with the set of all the rows or J coincides with the set of all the columns, then we write $I = *$ or $J = *$, respectively. By $\mathbf{I}_n, \mathbf{0}_n$, and $\mathbf{1}_n$, we denote the identity, all-zeroes matrices, and all-ones vector of the order n, respectively. By $SUM(\mathbf{A})$, we denote the sum of all the elements of $\mathbf{A}$. This sum for $n \times m$ matrices $\mathbf{A}$ can be computed as the product $(\mathbf{1}_n)^T \cdot \mathbf{A} \cdot \mathbf{1}_m$. For n-ary vector $\mathbf{v}$, by $nnz(\mathbf{v})$, we denote the set $\{i : \mathbf{v}[i] \neq 0\}$.

For a (simple or mixed) graph G, by $V(G)$ and $E(G)$, we denote its vertex set $\{v_1, \dots, v_n\}$ and edge set $\{e_1, \dots, e_m\}$, respectively. By $\mathbf{A}_G$, we denote the *adjacency matrix* of G, i.e., the Boolean $n \times n$ matrix, such that, for any $i, j \in [n]$, its ij-th element is equal to one iff $(v_i, v_j) \in E(G)$. Assuming that G is simple, by $\mathbf{I}_G$, we denote its *incidence matrix*, i.e., the $n \times m$ Boolean matrix, in which, for any $i \in [n], j \in [m]$, the ij-th element is one iff v_i is incident to e_j.

For $v, u \in V(G)$, by $d_G(v, u)$ we denote the distance between v and u. If G is connected, then $diam(G)$ denotes the *diameter of* G, i.e. $\max\limits_{v,u \in V(G)} d_G(v, u)$. For a vertex v of a oriented graph G, the *left* and *right neighborhoods* of v, denoted by $N_l(v)$ and $N_r(v)$, respectively, are defined as follows:

$$N_l(v) = \{u \in V(G) : (u, v) \in E(G)\}, N_r(v) = \{u \in V(G) : (v, u) \in E(G)\}.$$

For a vertex v of a simple graph G, the *open* and *closed neighborhoods* of v, denoted by $N(v)$ and $N[v]$, respectively, are defined as follows:

$$N(v) = \{u \in V(G) : \{v, u\} \in E(G)\}, N[v] = N(v) \cup \{v\}.$$

Suppose that G is simple. By $\overline{G}$, we denote the complement graph of G. For any $V' \subseteq V(G)$, by $G \backslash V'$ we denote the resultant graph after deletion of all the vertices from V' with their incident edges.

By $Aut(G)$ and $Orbits(G)$, we denote the automorphism group of G and the set of all its orbits, respectively. Recall that a permutation π on $V(G)$ is an *automorphism* of G iff

$$\forall \{v, u\} \in E(G) \longleftrightarrow \{\pi(v), \pi(u)\} \in E(G).$$

The *orbit, generated by a vertex* $v \in V(G)$, is defined as the set

$$\{u : \exists \pi \in Aut(G), \pi(v) = u\}.$$

The set orbits gives a disjunctive partition of $V(G)$. By $Sym(n)$ and $Sym(n_1) \times \ldots \times Sym(n_l)$, we denote the symmetric group over n elements and the direct product of symmetric groups over $n_1, \ldots, n_l$ elements, respectively.

3 A linear algebraic version of the Nešetřil-Poljak's reduction

For given a simple data graph G and a simple query graph H, it is proposed in [30] to deal with a special graph F, such that all the entries of H into G correspond to k-cliques of F, where $k = |V(H)|$ and $n = |V(G)|$. In other words, it reduces the general case to the case of clique-queries. More precisely, the graph F is defined as a graph on the vertex set $V(G) \times V(H)$, having the edge set

$$\{\{(g_1, h_1), (g_2, h_2)\} : \{h_1, h_2\} \in E(\overline{H}) \vee (\{g_1, g_2\} \in E(G) \wedge \{h_1, h_2\} \in E(H))\}.$$

The set $\{(g_1, h_1), \ldots, (g_k, h_k)\}$ is a k-clique in F iff G has an entry of H on the vertex set $\{g_1, \ldots, g_k\}$ with the set of edges, formed by the rule

$$\{g_i, g_j\} \text{ is an edge iff } \{h_i, h_j\} \in E(H).$$

Unfortunately, the algorithm from [30] could match entries of H into G with multiplicities, i.e., distinct k-cliques of F may correspond to the same containment, due to automorphisms of H. Moreover, it involves simultaneous working with G and $\overline{G}$, at least one of them is not sparse. This section is aimed to present the first LA version

of the Nešetřil-Poljak's reduction, which completely avoids the mentioned multiplicities and avoids in part a possible memory overflow.

It is known that, for any graphs G_1 and G_2, possibly mixed, $\mathbf{A}_{G_1} \otimes \mathbf{A}_{G_2}$ is the adjacency matrix of the graph

$$(V(G_1) \times V(G_2), \{((v_1, u_1), (v_2, u_2)) : (v_1, v_2) \in E(G_1), (u_1, u_2) \in E(G_2)\}).$$

Hence, to obtain F, one may use the formula

$$\mathbf{A}_F = \mathbf{A}_G \otimes \mathbf{A}_H + \mathbf{A}_G \otimes \mathbf{A}_{\overline{H}} + \mathbf{A}_{\overline{G}} \otimes \mathbf{A}_{\overline{H}}. \tag{1}$$

Indeed, $\mathbf{A}_G \otimes \mathbf{A}_H, \mathbf{A}_G \otimes \mathbf{A}_{\overline{H}}, \mathbf{A}_{\overline{G}} \otimes \mathbf{A}_{\overline{H}}$ correspond to the conformity of edges of H to edges of G, non-edges of H to edges of G, non-edges of H to non-edges of G, respectively.

To avoid repetitions, according to automorphisms of H, i.e., to produce a bijection between all the k-cliques of F and all the entries of H into G, we will use special orientations for $E(G)$ and for some parts of $E(H)$ and $E(\overline{H})$. Firstly, we arbitrarily acyclically orient all the edges of G and $\overline{G}$, for example, by numbering vertices of G with numbers in $[n]$ and orienting each edge from the smallest number to the biggest number. Further, we identify vertices and their numbers. The resultant graphs are denoted by $\overrightarrow{G}$ and $\overrightarrow{\overline{G}}$, respectively.

Next, we will work with the automorphism groups of H and some its induced subgraphs. There are several combinatorial algorithms for computing the automorphism group of a given graph, see, for example, the papers [26, 28, 34] and references therein. When H is small, one can simply split $V(H)$ into subsets $V_1, \ldots, V_l$ of vertices of the same degrees $d_1, \ldots, d_l$ and put $n_i = |V_i|$, for any $i \in [l]$. Then,

$$Aut(H) \subseteq Sym(n_1) \times \ldots \times Sym(n_l),$$

and we enumerate all the l-tuples $\pi = (\pi_1, \ldots, \pi_l)$, where π_i is a permutation of V_i, and check whether π is an automorphism of H by verifying whether

$$\forall \{x, y\} \in E(H) \longleftrightarrow \{\pi(x), \pi(y)\} \in E(H).$$

To find $Orbits(H)$, we enumerate all the vertices of H and find sets of vertices, to which vertices of H are transferred by permutations from $Aut(H)$. Finally, we apply the following combinatorial algorithm:

Algorithm 1: Orienting in part the edge sets of H and $\overline{H}$

```
Compute Aut(H) and Orbits(H);
S ← ∅; Aut ← Aut(H); Orbits ← Orbits(H);
while (|Aut| ≥ 2) do
    Choose any orbit Orb ∈ Orbits with |Orb| ≥ 2 and any vertex x ∈ Orb;
    Orient all the edges {x, y} ∈ E(H), y ∈ Orb as (x, y) and all the edges
      {x, z} ∈ E(H̄), z ∈ Orb as (x, z);
    Put S ← S ∪ {x} and compute Aut(H \ S) and Orbits(H \ S);
    Aut ← Aut(H \ S); Orbits ← Orbits(H \ S);
end
return H and H̄
```

The proposed algorithm can be explained as follows. It supports the invariant that *Aut* is the automorphism group of $H \setminus S$ and *Orbits* is the set of its orbits. Hence, vertices from any orbit of *Aut* have equal rights between each other. Therefore, in each entry of H into G, any orbit's element can be identified with the minimum vertex among G's vertices, corresponding to elements of the orbit. The orientation of edges from x arranges x to be a minimum vertex for elements from the x's orbit. Algorithm 1 is finished, when *Aut* is constituted by the trivial permutation only.

The resultant mixed graphs after orientation above are denoted by $\overrightarrow{H}$ and $\overrightarrow{\overrightarrow{H}}$, respectively. Hence, the formula (1) can be rewritten as follows

$$\mathbf{A}_{\overrightarrow{F}} = \mathbf{A}_{\overrightarrow{G}} \otimes \mathbf{A}_{\overrightarrow{H}} + \mathbf{A}_{\overrightarrow{G}} \otimes \mathbf{A}_{\overrightarrow{\overrightarrow{H}}} + \mathbf{A}_{\overrightarrow{\overrightarrow{G}}} \otimes \mathbf{A}_{\overrightarrow{H}}. \tag{2}$$

The adjacency matrices $\mathbf{A}_{\overrightarrow{G}}$ and $\mathbf{A}_{\overrightarrow{\overrightarrow{G}}}$ could be too dense to apply the multiplications with them. To overcome this phenomena, one may use some filtering technique for edges and non-edges of G and possible sequential splitting the resultant graphs into batches for incremental keeping $\overrightarrow{F}$ and using an incremental algorithm for the k-clique matching problem.

Suppose that G and H are both connected. Let us note that if v,u $\in E(\overline{G})$corresponds to $\{x, y\} \in E(\overline{H})$ of some copy of H in G, then $d_H(v, u) \leq d_G(x, y)$. Hence, we do not need those anti-edges $\{v, u\}$ of G that $d_G(v, u) > diam(H)$. This idea can be implemented and improved as follows. The distances in G and H can be found, using a LA form of Breadth First Search, see, for example, page 33 from [21]. For any $i \in [diam(H)]$, the v-th column of $\mathbf{D}^i_G$, where $v \in V(G)$, keeps the mask of $\{u \in V(G) : d_G(v, u) = i\}$, and the x-th column of $\mathbf{D}^i_H$, where $x \in V(H)$, keeps the mask of $\{y \in V(H) : d_H(x, y) = i\}$. By $\mathbf{SD}^i_G$, we denote the matrix $\bigvee_{j=1}^{i} \mathbf{D}^j_G$.

More precisely, we use the following algorithm:

Algorithm 2: LA computing distances in G and H

$\mathbf{D}_H^{-1} \leftarrow \mathbf{0}_k$; $\mathbf{D}_H^0 \leftarrow \mathbf{I}_k$; $i = 0$;
while $(\mathbf{D}_H^i \neq \mathbf{0}_k)$ **do**
  $\mathbf{D}_H^{i+1} \leftarrow (\mathbf{A}_H \bullet \mathbf{D}_H^i) | (\mathbf{D}_H^i \vee \mathbf{D}_H^{i-1})$; $i \leftarrow i + 1$;
end
$diam(H) = i - 1$; $\mathbf{D}_G^{-1} \leftarrow \mathbf{0}_n$; $\mathbf{D}_G^0 \leftarrow \mathbf{I}_n$; $\mathbf{SD}_G^0 \leftarrow \mathbf{0}_n$;
for $(i \in [diam(H)])$ **do**
  $\mathbf{D}_G^i \leftarrow (\mathbf{A}_G \bullet \mathbf{D}_G^{i-1}) | (\mathbf{D}_G^{i-1} \vee \mathbf{D}_G^{i-2})$; $\mathbf{SD}_G^i \leftarrow \mathbf{SD}_G^{i-1} \vee \mathbf{D}_G^i$;
end
return $(\mathbf{SD}_G^1, \ldots, \mathbf{SD}_G^{diam(H)})$ and $(\mathbf{D}_H^1, \ldots, \mathbf{D}_H^{diam(H)})$

For any $x \in V(H)$, by $V(x)$, we denote the set

$$\{v \in V(G) : \forall i \in [diam(H)] \, |\{y : d_H(x, y) = i\}| \leq |\{u : d_G(v, u) \leq i\}|\}$$
$$= \{v \in V(G) : \forall i \in [diam(H)] \, SUM(\mathbf{D}^i_H[*][x]) \leq SUM(\mathbf{SD}^i_G[*][v])\}.$$

Therefore, for any $x \in V(H)$, the only vertices from $V(x)$ can correspond to x in copies of H in G.

For any edge or anti-edge e of H, the matrix $\mathbf{A}_e$ denotes the adjacency matrix of the graph on $V(H)$ with the unique edge e. Then, the formula (2) can be rewritten as

$$\begin{aligned}\mathbf{A}_{\overrightarrow{F}} = &\bigvee_{e=(x,y)\in E(\overrightarrow{H})} (\mathbf{A}_{\overrightarrow{G}}[V_x \cup V_y][V_x \cup V_y] \otimes \mathbf{A}_e) \\ + &\bigvee_{e=(x,y)\in E(\overrightarrow{\overline{H}})} (\mathbf{SD}_{\overrightarrow{G}}^{d_H(x,y)}[V_x \cup V_y][V_x \cup V_y] \otimes \mathbf{A}_e),\end{aligned} \tag{3}$$

where $\mathbf{SD}^i_{\overrightarrow{G}}[v][u] = \mathbf{SD}^i_G[v][u]$, if $v < u$, otherwise, $\mathbf{SD}^i_{\overrightarrow{G}}[v][u] = 0$.

The graph $\overrightarrow{F}$ is an acyclically completely oriented graph, in which all the oriented k-cliques bijectively correspond to all the entries of H into G. To reduce the amount of used memory, edges and non-edges of G can be split into parts to obtain, according to (3), the graph $\overrightarrow{F}$ incrementally. Full computational and incremental LA algorithms for k-clique matching and counting will be presented in the next sections.

4 Full computational LA algorithms for the *k*-clique matching and counting problems

4.1 The Nešetřil-Poljak's algorithm for *k*-clique matching/counting and its LA version

The paper [30] also presents a combinatorial full computational algorithm for matching/counting k-cliques. Its idea is to use a recursion and matching/counting triangles in auxiliary graphs. For the simplicity, let us assume that $k = 3l$. For a given simple graph G, its auxiliary graph G' is defined as

$$V(G') = \{X \subseteq V(G) : \ X \text{ is a } l\text{-clique in } G\},$$

$$E(G') = \{\{X, Y\} : \ X \cup Y \text{ is a } 2l\text{-clique in } G\}.$$

Clearly that, to match/count $3l$-cliques in G, one only needs to match/count triangles in G'. This idea via the fast matrix multiplication was used for designing full computational [12] and fully dynamic [10] algorithms. Unfortunately, these algorithms do not explain how to construct and keep the graph G'. In this subsection, we overcome this difficulty and show how the algorithm from [30] can be completely implemented as a LA algorithm. It uses the following classical LA algorithm for matching/counting triangles:

Algorithm 3: LA matching/counting triangles in G

$Matches \leftarrow \emptyset$;
Arbitrarily acyclically orient G to obtain a graph $\overrightarrow{G}$;
$\mathbf{A}^* \leftarrow (\mathbf{A}_{\overrightarrow{G}})^2 \circ \mathbf{A}_{\overrightarrow{G}}$;
for $(\{v, v'' : \ \mathbf{A}^*[v][v''] \neq 0\})$ **do**
  $Matches \leftarrow Matches \cup \{(v, v', v'') : \ v' \in nnz(\mathbf{A}_{\overrightarrow{G}}[v][*] \circ (\mathbf{A}_{\overrightarrow{G}}[*][v''])^T)\}$;
end
return $Matches/SUM(\mathbf{A}^*)$

Indeed, ones of $(\mathbf{A}_{\overrightarrow{G}})^2$ correspond to end-vertices of oriented 2-paths, i.e., paths with 2 edges, in $\overrightarrow{G}$. Ones of $(\mathbf{A}_{\overrightarrow{G}})^2 \circ \mathbf{A}_{\overrightarrow{G}}$ mean that edges connect end-vertices of oriented 2-paths, meaning that triangles are formed. Hence, $SUM(\mathbf{A}^*)$ is the number of triangles. Algorithm 3 can also be used for matching triangles. To this end, for any $v, v'' \in V(G)$, such that $\mathbf{A}^*[v][v''] \neq 0$, we find

$$N_r(v) \cap N_l(v'') = nnz(\mathbf{A}_{\overrightarrow{G}}[v][*] \circ (\mathbf{A}_{\overrightarrow{G}}[*][v''])^T).$$

In the following algorithm, $\mathbf{Inc}_s$ is the incidence matrix, which rows correspond to vertices of G and columns are masks of all the ordered s-cliques of G.

Algorithm 4: LA algorithm for matching/counting k-cliques in G

```
if (k = 1) then
    Inc_1 = I_n;
    return Inc_1
end
if (k = 2) then
    Inc_2 = I_G;
    return Inc_2
end
if (k = 3) then
    Apply Algorithm 3 to obtain Inc_3;
    return Inc_3
end
if (k ≥ 4) then
    p ← ⌊k/3⌋;
    r ← k mod 3;
    Recursively apply Algorithm 4 to obtain Inc_p;
    Compute (Inc_p)^T · A_G · Inc_p, replace to zeroes all its < p^2 elements, replace to
      ones all its = p^2 elements, and write the resultant matrix into A_{p,p};
    Apply Algorithm 3 to A_{p,p} to obtain a matrix Inc_{3p};
    if r = 0 then
        return Inc_{3p}
    end
    if (r = 1) then
        Arbitrarily acyclically orient G to obtain a graph →G;
        A* ← A_{→G} · Inc_{3p}; Inc_{3p+1} ← Inc_{3p};
        for ({i, j : A*[i][j] = 3p + 1}) do
            Inc_{3p+1}[i][j] ← 1;
        end
        return Inc_{3p+1}
    end
    if (r = 2) then
        Arbitrarily acyclically orient G to obtain a graph →G;
        A* ← A_{→G} · Inc_{3p}; Inc_{3p+1} ← Inc_{3p};
        for ({i, j : A*[i][j] = 3p + 1}) do
            Inc_{3p+1}[i][j] ← 1;
        end
        A* ← A_{→G} · Inc_{3p+1}; Inc_{3p+2} ← Inc_{3p+1};
        for ({i, j : A*[i][j] = 3p + 2}) do
            Inc_{3p+2}[i][j] ← 1;
        end
        return Inc_{3p+2}
    end
end
```

The matrices $\mathbf{A}_{\overrightarrow{G}} \cdot \mathbf{Inc}_{3p}$ and $\mathbf{A}_{\overrightarrow{G}} \cdot \mathbf{Inc}_{3p+1}$ show the sizes of vertex right neighborhoods in all the ordered $3p$- and $3p+1$-cliques of G. Rows and columns of

$(\mathbf{Inc}_p)^T \cdot \mathbf{A}_G \cdot \mathbf{Inc}_p$ are all the p-cliques of G, and all its elements are the quantities of edges between pairs of p-cliques. These properties certify the correctness of Algorithm 4.

Even for counting k-cliques, Algorithm 4 may consume a large amount of memory, as it uses the incidence matrix of ordered $\lfloor \frac{k}{3} \rfloor$-cliques to vertices. It may be impractical even for relatively small k. A more efficient full computational LA algorithm will be described in the next subsection.

4.2 Another LA full computational *k*-clique matching/counting algorithm

Our algorithm can be briefly explained as follows: firstly, possibly, use a preprocessing step, secondly, somehow acyclically orient the remaining subgraph, finally, use descending to 3- or 4-cliques by right neighborhoods with subsequent matching/counting for k-cliques of the original graph. For deleting redundant vertices and edges, i.e., as a preprocessing step, we propose the following LA procedures:

Algorithm 5: LA deleting redundant vertices and edges from G (I)

$\mathbf{v} \leftarrow \mathbf{A}_G \cdot \mathbf{1}_n$;
By $\mathbf{v}$, find the mask $mask'$ of all its $\geq k-1$ elements;
return $\mathbf{A}_G[mask'][mask']$

The vector $\mathbf{v}$ keeps degrees of all the vertices of G. Any its vertex of degree at most $k-2$ cannot belong to any k-clique of G. This certifies the correctness of Algorithm 5.

Algorithm 6: LA deleting redundant vertices and edges from G (II)

$\mathbf{A}' \leftarrow (\mathbf{A}_G)^2 \circ \mathbf{A}_G$;
Replace all $< k-2$ elements of $\mathbf{A}'$ to zeroes;
Find a mask $mask'$ of those rows of $\mathbf{A}'$ that have at least $k-1$ non-zero elements;
Put $\mathbf{A}'' \leftarrow \mathbf{A}'[mask'][mask']$ and replace all non-zero elements in $\mathbf{A}''$ to ones;
return $\mathbf{A}''$

For any $x \in V(G)$, the x-th row of $\mathbf{A}'$ keeps $|N(x) \cap N(y)|$, for any $y \in N(x)$. Therefore, edges $\{x, y\}$ of G with $\mathbf{A}'[x][y] \leq k-3$ cannot belong to any its k-clique, and they can be removed. After that vertices of G of degree at most $k-2$ can also be deleted, as they cannot belong to any k-clique. This explains Algorithm 6.

Our full computational LA algorithm for matching/counting k-cliques with descending to 3-cliques is presented in the following pseudocode below:

Algorithm 7: LA matching/counting k-cliques in G

$Matches \leftarrow \emptyset; Count \leftarrow 0;$
if $(k = 3)$ **then**
  $Matches/Count \leftarrow$ Apply Algorithm 3 to G;
else
  $G \leftarrow$ Apply Algorithm 5 or 6 to G or avoid this step;
  Arbitrarily acyclically orient G to obtain $\mathbf{A}_{\overrightarrow{G}}$; $\mathbf{A}_1 \leftarrow \mathbf{A}_{\overrightarrow{G}}$;
  for (*any row* v_1 *of* $\mathbf{A}_1$) **do**
    $N_r \leftarrow \mathbf{A}_1[v_1][*]$; $\mathbf{A}_2 \leftarrow \mathbf{A}_1[N_r][N_r]$;
    for (*any row* v_2 *of* $\mathbf{A}_2$) **do**
      $N_r \leftarrow \mathbf{A}_2[v_2][*]$; $\mathbf{A}_3 \leftarrow \mathbf{A}_2[N_r][N_r]$;
      ...
      for (*any row* v_{k-3} *of* $\mathbf{A}_{k-3}$) **do**
        $N_r \leftarrow \mathbf{A}_{k-3}[v_{k-3}][*]$; $\mathbf{A}_{k-2} \leftarrow \mathbf{A}_{k-3}[N_r][N_r]$;
        $Matches'/Count' \leftarrow$ Apply Algorithm 3 to $\mathbf{A}_{k-2}$;
        $Matches \leftarrow \{(v_1, \ldots, v_{k-3}, v', v'', v''') : (v', v'', v''') \in Matches'\}$;
        $Count \leftarrow Count + Count'$;
      end
    end
  end
end
return $Matches/Count$

In the last loop of Algorithm 7, instead of matching/counting triangles, one may use the following LA full computational algorithms for matching/counting 4-cliques:

Algorithm 8: LA matching/counting 4-cliques in G (I)

$Matches \leftarrow \emptyset; Count \leftarrow 0;$
Arbitrarily acyclically orient G to obtain $\mathbf{A}_{\overrightarrow{G}}$;
for $(\{v, u : \mathbf{A}_{\overrightarrow{G}}[v][u] \neq 0\})$ **do**
  $mask_e \leftarrow \mathbf{A}_{\overrightarrow{G}}[v][*] \circ (\mathbf{A}_{\overrightarrow{G}}[*][u])^T$; $\mathbf{A}_e \leftarrow \mathbf{A}_{\overrightarrow{G}}[mask_e][mask_e]$;
  $Matches \leftarrow Matches \cup \{(v, v', u', u)\}$, for any $v' < u'$ with $\mathbf{A}_e[v'][u'] = 1$;
  $Count \leftarrow Count + SUM(\mathbf{A}_e)$;
end
return $Matches/Count$

Algorithm 9: LA matching/counting 4-cliques in G (II)

$Matches \leftarrow \emptyset; Count \leftarrow 0;$
Arbitrarily acyclically orient G to obtain a graph $\overrightarrow{G}$;
$\mathbf{A}^* \leftarrow (\mathbf{A}_{\overrightarrow{G}})^2 \circ \mathbf{A}_{\overrightarrow{G}}$;
for $(v \in V(G))$ **do**
  $N_r(v) \leftarrow \mathbf{A}_{\overrightarrow{G}}[v][*]$; $\overline{N_r(v)} \leftarrow V(G) \setminus N_r(v)$; $\mathbf{A}'_v \leftarrow \mathbf{A}_{\overrightarrow{G}}[N_r(v)][N_r(v)]$;
  $\mathbf{A}^*_v \leftarrow \mathbf{A}^*[N_r(v)][N_r(v)]$; $\mathbf{A}^{**}_v \leftarrow (\mathbf{A}_{\overrightarrow{G}}[N_r(v)][\overline{N_r(v)}] \cdot \mathbf{A}_{\overrightarrow{G}}[\overline{N_r(v)}][N_r(v)]) \circ \mathbf{A}'_v$;
  $Count \leftarrow Count + SUM(\mathbf{A}^*_v) - SUM(\mathbf{A}^{**}_v)$;
  $Matches \leftarrow Matches \cup \{(v, v', v'', v''') : v'' \in nnz(\mathbf{A}'_v[v'][*] \circ (\mathbf{A}'_v[*][v'''])^T)\}$,
    for any $\mathbf{A}'_v[v'][v'''] \neq 0$;
end
return $Matches/Count$

The correctness of Algorithm 8 is obvious. Let us certify the correctness of Algorithm 9. For any $v \in V(G)$, $SUM(\mathbf{A}^*_v)$ is the number of triangles with the

min- and max-vertices from $N_r(v)$. Hence, to compute the number of 4-cliques in G with the min-vertex v, it is sufficient to substract

$$\sum_{\{v',v''':\ \mathbf{A}_v^*[v'][v''']\neq 0\}} |N_r(v') \cap N_l(v''') - N_r(v)|$$

from $SUM(\mathbf{A}_v^*)$. Therefore, the number of 4-cliques equals

$$\sum_{v\in V(G)} (SUM(\mathbf{A}_v^*) - SUM(\mathbf{A}_v^{**})).$$

To match 4-cliques, we find

$$N_r(v) \cap N_r(v') \cap N_l(v''') = nnz(\mathbf{A}_v^{'}[v'][*] \circ (\mathbf{A}_v^{'}[*][v'''])^T),$$

for any $v, v', v''' \in V(G)$, such that $\mathbf{A}_v'[v'][v'''] \neq 0$.

5 An incremental LA algorithm for the *k*-clique matching/counting problem

We assume that edge-disjoint batches $G_1, \ldots, G_T$ of G follow one by one, such that $V(G) = \bigcup_{i=1}^{T} V(G_i)$ and $E(G) = \bigsqcup_{i=1}^{T} E(G_i)$. For any $i \in [T]$, accumulated $G^{(i-1)} = \bigcup_{j=1}^{i-1} G_j$, where $G_0 = (\emptyset, \emptyset)$, and given G_i, the incremental k-clique matching/counting problem asks to match/count all the k-cliques Q of $G^{(i)}$ with $Q \cap E(G_i) \neq \emptyset$.

Full computational and incremental algorithms for the k-clique matching/counting problem can be useful for each other. For example, after applying the removing procedure by Algorithm 5 or 6 to G, the resultant graph can be split into batches and an incremental algorithm can be used. In incremental algorithms, for working with G_1, when it is large enough, one may use a full computational algorithm.

The main idea of our algorithm is to split $E(G_i)$ into classes, such that any two edges from the same class cannot belong to a common k-clique of $G^{(i)}$, to consider the classes one by one, and to match/count $(k-2)$-cliques in parallel in the subgraphs, induced by common neighbors of x and y in $G^{(i)}$, for any $\{x, y\} \in E(G_i)$.

The next procedure, for a given simple graph $G' = (V', E'), V' = \{1, 2, \ldots, n'\}$, gives an edge-disjoint partition $E' = \bigsqcup_{i=1}^{s'} Cl'[i]$ into classes with a heuristic minimization of s'. It uses the set of arrays $\{Forb'[v'] : v' \in V'\}$, where, for any $v' \in V'$, we have

$$Forb'[v'] = \{i' : \exists \{a', b'\} \in Cl'[i'], N[v'] \cap \{a', b'\} \neq \emptyset\}.$$

Algorithm 10: LA splitting G' into classes

$s' \leftarrow 1; Cl'[1] \leftarrow \emptyset; \forall v' \in V'\ Forb'[v'] \leftarrow \emptyset;$
for $(e' = \{v', u'\} \in E')$ **do**
  $V_{e'} \leftarrow \{1, \ldots, s'\} \setminus (\bigcup_{w' \in N[v'] \cup N[u']} Forb'[w']);$
  if $(V_{e'} \neq \emptyset)$ **then**
    $i' \leftarrow \min(V_{e'}); Cl'[i'] \leftarrow Cl'[i'] \cup \{e'\};$
    for $(w' \in N[v'] \cap N[u'])$ **do**
      $Forb'[w'] \leftarrow Forb'[w'] \cup \{i'\};$
    end
  else
    $s' \leftarrow s' + 1; Cl'[s'] \leftarrow \{e'\};$
    for $(w' \in N[v'] \cap N[u'])$ **do**
      $Forb'[w'] \leftarrow Forb'[w'] \cup \{s'\};$
    end
  end
end
return $(Cl'[1], \ldots, Cl'[s'])$

At each step of Algorithm 10, the minimum number class is searched, such that e' can be put in it without forming triangles by edges from the same class, which is guaranteed by updating $Forb'[w']$ with $w' \in N[v'] \cap N[u']$. Using the intersection of $N[v']$ and $N[u']$ instead of their union decreases the quantity of used edge classes. This certifies the correctness of Algorithm 10.

Our incremental k-clique matching/counting LA algorithm is presented in the following pseudocode:

Algorithm 11: Incremental k-clique matching/counting LA algorithm for $G^{(i-1)}$ and G_i

By $\mathbf{A}_{G^{(i-1)}}$ and $\mathbf{A}_{G_i}$, compute G'_i as the subgraph of $G^{(i)}$, induced by $\bigcup_{v \in V(G_i)} N[v]$, and the subgraph G''_i of G'_i, induced by $V(G_i)$;
$(Cl'[1], \ldots, Cl'[s']) \leftarrow$ Apply Algorithm 10 to G''_i;
$Matches \leftarrow \emptyset; Count \leftarrow 0;$
Arbitrarily acyclically orient G'_i to obtain a graph $\overrightarrow{G'_i}$;
for $(i \in [s'])$ **do**
  $\overrightarrow{G'_i} \leftarrow \overrightarrow{G'_i} \setminus Cl'[i];$
  for $(\{x, y\} \in Cl'[i]$ *in parallel*$)$ **do**
    $Matches'/Count' \leftarrow$ Apply the matching/counting phase of Algorithm 7 for the subgraph of $\overrightarrow{G'_i}$, induced by $N_r(x) \cap N_r(y)$;
    $Matches \leftarrow \{(x, y, v_1, \ldots, v_{k-2}) : (v_1, \ldots, v_{k-2}) \in Matches'\};$
    $Count \leftarrow Count + Count';$
  end
end
return $Matches/Count$

The induced subgraphs, appearing in Algorithm 11, can be obtained by masking adjacency matrices of their supergraphs. If $i = 1$, then the only matching/counting phase of Algorithm 7 can be used without splitting the edge set into classes. Vertex and edge deletions by Algorithms 5 and 6 can also be used for G'_i after its definition and/or in the first for-cycle.

6 Computational experiments

In this section, we provide results of computational experiments for four known datasets, the graphs $com-dblp, com-orkut, com-friendster, com-lj$, see [25], whose quantities of vertices and edges are shown in Table 1 below:

We considered the only counting variant of the k-clique matching problem and full computational algorithms for it, based on Algorithm 7, with three handles for tuning. The first of them is a choice between the absence of any graph reduction and the use of Algorithm 5 or 6. The second one is a choice between downing to 3-cliques and using Algorithm 3 and downing to 4-cliques and using Algorithm 8 or 9. The third one is a choice between the following three orientations:

- from smaller end-vertices to bigger end-vertices, called the *id orientation*,
- from smaller-degree end-vertices to bigger-degree end-vertices, if these degrees are distinct, otherwise, the id orientation between these vertices is used, called the *degree orientation*,
- the ϵ-Goodrich-Pszona orientation, for $\epsilon \in \{0.1, 0.2, 0.5, 1, 2, 5, 10\}$.

The *ϵ-Goodrich-Pszona orientation* is a generalization of the degree orientation, see [17] or the pseudocode below:

Algorithm 12: ϵ-Goodrich-Pszona orientation algorithm for a graph $G = (V, E)$

```
H ← G; L ← ∅;
while (V(H) ≠ ∅) do
    S ← (ε·|V(H)|)/(2+ε) vertices of the lowest induced degree;
    Append S to L;
    H ← H \ S;
end
Orient all the vertices of G, according to L;
```

The ϵ-Goodrich-Pszona orientation guarantees that the right neighborhood of any vertex is relatively small, see, for example, [33], for more details.

For any of these 72 possibilities, generated by concrete choices in the handles, and considered pairs (*dataset*, k), we chose the best computation time. Our hardware was one machine CPU: Kunpeng 920 4826 2.6GHz*2 (128 physical cores, one thread per core), 950 GiB of main memory, the used API is GraphBLAS.

We compared our solution with the methods, called Arb-count [33], BinaryJoin [29], kClist [8], Pivoter [19], WCO [23]. For now, Arb-count seems to be a state-of-the-art algorithm for k-clique counting. Unfortunately, we did not find open code for BinaryJoin and were not able to execute Pivoter and WCO on our cluster in the parallel regime of computations. The method Arb-count has not been launched on our machine.

So, we could provide experiments on our machine only with kClist and choosing the best option from the edge and vertex parallelisms. For the remaining approaches, we took runtimes from Table 2 of [33], which have been obtained on a machine with 30 two-way hyper-threading cores with 3.8GHz Intel Xeon Scalable (Cascade Lake)

Table 1 Quantities of vertices and edges

Graph	q. of vertices	q. of edges
com-dblp	317,080	1,049,866
com-orkut	3,072,441	117,185,083
com-friendster	65,608,366	1,806,067,135
com-lj	3,997,962	34,681,189

processors and 240 GiB of main memory. To make the conditions of experiments more comparable, we divided these times by $\frac{128*2.6}{30*3.8} \approx 2.919$. The results are shown in Table 2 below:

Table 2 Best runtimes in seconds

Graph	k	Our solver	Arb-count	BinaryJoin	kClist	Pivoter	WCO
com-dblp	4	0.05*	**0.034**	*0.041*	1	0.987	0.065
	5	*0.05**	**0.044**	0.144	1	0.987	0.127
	6	**0.05***	*0.103*	0.713	1	0.987	1.316
	7	**0.06****(5)**	*0.70*	13.46	3	0.987	22.63
	8	**0.05****(5)**	8.24	214.96	20	*0.987*	385.98
	9	**0.06****(5)**	96.40	2494.96	230	*0.987*	3336.07
	10	**0.05****(5)**	1021.49	>1.712 h	2416	*0.987*	>1.712 h
	11	**0.04****(5)**	>1.712 h	>1.712 h	>5 h	*0.987*	>1.712 h
com-orkut	4	4.92*	**1.062**	4.36	27	100.15	*3.669*
	5	*6.71**	**1.692**	9.97	29	131.91	17.30
	6	*14.77**	**4.306**	31.88	30	158.29	91.63
	7	*43.77***(2)*	**14.42**	141.66	47	177.21	479.23
	8	146.10**(0.5)	**51.69**	663.95	*120*	191.76	2064.74
	9	474.59**(0.2)	**200.20**	3334.31	422	*205.17*	>1.712 h
	10	1431.75**(0.1)	*793.38*	>1.712 h	1648	**221.71**	>1.712 h
	11	10574.10**(0.1)	*3029.64*	>1.712 h	6515	**221.71**	>1.712 h
com-friendster	4	95.97*	**37.50**	*56.15*	655	>1.712 h	69.14
	5	92.51*	**38.28**	*72.81*	649	>1.712 h	130.04
	6	105.86*	**39.58**	*76.03*	665	>1.712 h	343.10
	7	*139.19**	**47.95**	216.65	651	>1.712 h	1448.85
	8	*259.40**	**102.99**	1552.79	1041	>1.712 h	>1.712 h
	9	**536.70****(0.5)**	*615.32*	>1.712 h	2489	>1.712 h	>1.712 h
	10	**1365.39****(0.5)**	*5767.87*	>1.712 h	>5 h	>1.712 h	>1.712 h
	11	5502.29**(0.5)	>1.712 h	>1.712 h	>5 h	>1.712 h	>1.712 h
com-lj	4	*1.02**	**0.606**	1.404	9	91.83	2.268
	5	*4.94***(1)*	**2.576**	14.49	15	505.65	27.67
	6	**87.71****(0.1)**	*88.54*	622.43	257	2677.67	1181.47
	7	**1825.18****(0.1)**	*3677.02*	>1.712 h	10166	>1.712 h	>1.712 h
	8	>5h**(0.1)	>1.712 h	>1.712 h	>5 h	>1.712 h	>1.712 h

Table 3 Runtimes for $(com - dblp, k)$ and $12 \leq k \leq 20$ in seconds

k	12	13	14	15	16	17	18	19	20
Time	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**

For kClist, we provided the computation times in integer number of seconds. The first- and second-best times are emphasized with the bold and italic environments, respectively. In the cases, where it is not possible to select the best and second-best times, we do not use any emphasizing. The footnote * means that we use the id orientation, Algorithm 5, descending to 4-cliques by Algorithms 7 and 8. The footnote $^{**(\epsilon)}$ refers to the use of the ϵ-Goodrich-Pszona orientation, Algorithm 5, descending to 4-cliques by Algorithms 7 and 8.

For many pairs (*dataset*, k), Arb-count showed the best results. For $com - dblp$ and $6 \leq k \leq 11$, our approach gave 2.06-−24.6-speedups over the second-best solution. For $com - friendster$ and $k \in \{10, 11\}$, our approach gave 1.15- and 4.22-speedups over the second-best solution. For $com - lj$ and $k \in \{6, 7\}$, our approach gave 1.01- and 2.01-speedups over the second-best solution.

To clarify the situation for $com - dblp$ and large values of k, we conducted an additional experiment. The experiment showed the following results, where we everywhere used the conditions of $^{**(5)}$, see Table 3:

Our solver showed good results for $12 \leq k \leq 20$.

7 Conclusions and future work

In this paper, we considered the subgraph matching problem, which is, for given simple graphs G and H, to find all the entries of H in G. Linear algebraic (LA, for short) algorithms are well suited for parallelisation of computational process. Prior to this paper, LA algorithms for the subgraph matching problem were known only for a few types of H. The first LA algorithm for this problem and general H was presented in this paper. It uses a LA reduction to the case, when H is a clique. Specifically for k-clique matching/counting, we presented a LA algorithm with several optimization techniques. Conducted computational experiments for its counting variant, several large graphs, and values of k showed the viability of our approaches. Developing new LA algorithms for the subgraph matching problem and improving the existing ones is a challenging research problem for possible future research.

Acknowledgements The work of Malyshev D.S. was conducted within the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE).

References

1. Ahmed, N., et al.: Graphlet decomposition: framework, algorithms, and applications. Knowl. Inf. Syst. **50**(3), 689–722 (2017)
2. Alon, N., et al.: Biomolecular network motif counting and discovery by color coding. Bioinformatics **24**(13), 241–249 (2008)

3. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica **17**, 209–223 (1997)
4. Bonne, M., Censor-Hillel, K.: Distributed detection of cliques in dynamic networks. In: Baier, C., Chatzigiannakis, I., Flocchini, P., Leonardi, S. (eds.) Proceedings of International Colloquium on Automata, Languages, and Programming, pp 132:1–132:15. Dagstuhl Publishing (2019)
5. Chakaravarthy, V., et al.: Subgraph counting: Color coding beyond trees. In: O'Conner, L. (ed.) International Symposium on Parallel and Distributed Processing Symposium Proceedings, pp 2–11. Piscataway, IEEE (2016)
6. Chen, L., et al.: A GraphBLAS approach for subgraph counting. ArXiv, https://doi.org/10.48550/arXiv.1903.04395.
7. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1), 210–223 (1985)
8. Danisch, M., Balalau, O., Sozio, M.: Listing *k*-cliques in sparse real-world graphs. In: Champin, P.-A. et al. (eds.) Proceedings of International Conference on World Wide Web, pp 589–598. International WWW Conference Committee (2018)
9. Dave, V., Ahmed, N., Hasan, M.: PE-CLoG: Counting edge-centric local graphlets. In: Nie, J-Y et al. (eds.) Proceedings of International Conference on Big Data, pp 586–595. IEEE, Piscataway (2017)
10. Dhulipala, L., Liu, Q., Shun, J., Yu, S.: Parallel batch-dynamic *k*-clique counting. In: Shapira, M. (ed.) Proceedings of Symposium on Algorithmic Principles of Computer Science, pp. 129–143. SIAM, Philadelphia (2021)
11. Dvorak, Z., Tuma, V.: A dynamic data structure for counting subgraphs in sparse graphs. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) Proceedings of Workshop on Algorithms and Data Structures, pp. 304–315. Springer-Verlag, Berlin (2013)
12. Eisenbrand, F., Grandoni, F.: On the complexity of fixed parameter clique and dominating set. Theoret. Comput. Sci. **326**(1–3), 57–67 (2004)
13. Eppstein, D.: Arboricity and bipartite subgraph listing algorithms. Inf. Process. Lett. **51**(4), 207–211 (1994)
14. Eppstein, D., Goodrich, M., Strash, D., Trott, L.: Extended dynamic subgraph statistics using h-index parameterized data structures. Theoret. Comput. Sci. **447**, 44–52 (2012)
15. Eppstein, D., Spiro, E.: The h-index of a graph and its application to dynamic subgraph statistics. In: Dehne, F., Gavrilova, M., Sack, J., Tóth, C. (eds.) Proceedings of Workshop on Algorithms and Data Structures, pp 278–289. Springer-Verlag, Berlin (2009)
16. Finocchi, F., Finocchi, M., Fusco, E.: Clique counting in MapReduce: algorithms and experiments. J. Experimen. Algorithmics **20**, 1.7:1-1.7:20 (2015)
17. Goodrich, M., Pszona, P.: External-memory network analysis algorithms for naturally sparse graphs. In: Demetrescu, C., Halldórsson, M (eds.) Proceedings of European Symposium on Algorithms, pp 664–676. 2011. Springer-Verlag, Berlin (2011)
18. Greyser, V., Soszynski, A., Kao, E.: Leveraging linear algebra to count and enumerate simple subgraphs. In: Proceedings of High Performance Extreme Computing Conference, pp. 1-8. IEEE, Piscataway (2020)
19. Jain, S., Seshadhri., C.: The power of pivoting for exact clique counting. In: Caveree, J., Hu, B., Lalmas, M., Wang, W. (eds.) Proceedings of the 13th International Conference on Web Search and Data Mining, pp. 268–277. Association for Computing Machinery, New York (2020)
20. Kara, A., et al.: Counting triangles under updates in worst-case optimal time. In: Barcelo, P., Calautti, M. (eds.) Proceedings of International Conference on Database Theory, pp 4:1–4:18. Dagstuhl Publishing (2019)
21. Kepner, J., Gilbert, J.: Graph algorithms in the language of linear algebra. SIAM, Philadelphia (2011)
22. Kloks, T., Kratsch, D., Müller, H.: Finding and counting small induced subgraphs efficiently. Inform. Process. Lett. **74**(3–4), 115–121 (2000)
23. Lai, L., et al.: Distributed subgraph matching on timely dataflow. Proc. VLDB Endow. **12**(10), 1099–1112 (2019)
24. Latapy, M.: Main-memory triangle computations for very large (sparse (power-law)) graphs. Theoret. Comput. Sci. **407**(1–3), 458–473 (2008)
25. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection.http://snap.stanford.edu/data (2022). Accessed 30 December 2022

26. López-Presa, J., Chiroque, L., Anta, A.: Novel techniques to speed up the computation of the automorphism group of a graph. J. Appl. Math. **2014**(934637), 15 (2014)
27. Makkar, D., Bader, D., Green, O.: Exact and parallel triangle counting in dynamic graphs. In: Smari, W. (ed.) Proceedings of International Conference on High Performance Computing and Simulation, pp 2–12. IEEE, Piscataway (2017)
28. McCay, B., Piperno, A.: Practical graph isomorphism, II. J. Symb. Comput. **60**, 94–112 (2014)
29. Mhedhbi, A., Salihoglu, S.: Optimizing subgraph queries by combining binary and worst-case optimal joins. Proc. VLDB Endow. **12**(11), 1692–1704 (2019)
30. Nešetřil, J., Poljak, S.: On the complexity of the subgraph problem. Comment. Math. Univ. Carol. **26**(2), 415–419 (1985)
31. Papadimitriou, C., Yannakakis, M.: The clique problem for planar graphs. Inf. Process. Lett. **13**, 131–133 (1981)
32. Pinar, A., Seshadhri,C., Vishal, V.: ESCAPE: Efficiently counting all 5-vertex subgraphs. In: Barrett, R., Cummings, R. (eds.) Proceedings of International Conference on World Wide Web, pp. 1431–1440. International WWW Conference Committee (2017)
33. Shi, J., Dhulipala, L., Shun, J.: Parallel clique counting and peeling algorithms. In: Bender, M., Gilbert, J., Hendrickson, B., Sullivan, B. (eds.) Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms, pp. SIAM, Philadelphia (2021)
34. Stoichev, S.: New exact and heuristic algorithms for graph automorphism group and graph isomorphism. ACM J. Experimen. Algorithmics **24**(1.15), 1–27 (2019)
35. Sun, Z., et al.: Efficient subgraph matching on billion node graphs. Proc. VLDB Endow. **5**(9), 788–799 (2012)
36. Vassilevska, V.: Efficient algorithms for clique problems. Inf. Process. Lett. **109**(4), 254–257 (2009)
37. Watts, D., Strogatz, S.: Collective dynamics of small-world networks. Nature **393**, 440–442 (1998)

Authors and Affiliations

Maxim D. Emelin[1] · Ilya A. Khlystov[2] · Dmitry S. Malyshev[3] · Olga O. Razvenskaya[4]

Maxim D. Emelin
makcum888e@mail.ru

Ilya A. Khlystov
ilya.khlyustov@huawei.com

Olga O. Razvenskaya
olga.razvenskaya@huawei.com

[1] Lobachevsky State University of Nizhny Novgorod, 23 Gagarina Av., Nizhny Novgorod, Russia 603950

[2] Nizhny Novgorod Huawei Research Center, 117 Maksima Gorkogo Str., Nizhny Novgorod, Russia 603006

[3] Laboratory of Algorithms and Technologies for Networks Analysis, National Research University Higher School of Economics, 136 Rodionova Str., Nizhny Novgorod, Russia 603093

[4] Nizhny Novgorod Huawei Research Center, 117 Maksima Gorkogo Str., Nizhny Novgorod, Russia 603006