An efficient exact algorithm for triangle listing in large graphs

Sofiane Lagraa, Hamida Seba

An Efficient Exact Algorithm for Triangle Listing in Large Graphs

Sofiane Lagraa(*) Hamida Seba(**)

(*)Université Grenoble Alpes, CNRS, TIMA, LIG, F-38000 Grenoble, France
(**)Université de Lyon, CNRS, Université Lyon 1, LIRIS, UMR5205, F-69622, France

Abstract

This paper presents a new efficient exact algorithm for listing triangles in a large graph. While the problem of listing triangles in a graph has been considered before, dealing with large graphs continues to be a challenge. Although previous research has attempted to tackle the challenge, this is the first contribution that addresses this problem on a compressed copy of the input graph. In fact, the proposed solution lists the triangles without decompressing the graph. This yields interesting improvements in both storage requirement of the graphs and their time processing.

1 Background and motivation

Graphs are an effective modeling tool. They are used to represent objects and to model problems in various domains and applications. A graph $G = (V, E)$ is composed of a set of vertices V (called also nodes) and a set of edges E with the cardinalities $|V(G)| = n$ and $|E(G)| = m$ where n is called the order of the graph and m its size. The set of edges E is a subset of $V \times V$ such that $(u, v) \in E$ means that vertices u and v are connected. Vertices generally represent objects and edges relations between these objects. Vertices and edges may also have labels that are associated to the properties of the objects and to their relationships. The notations related to graphs are summarized in Table 1.

Finding triangles in a graph is a major problem that is encountered in several applications related to the analysis of complex networks. A triangle, in an undirected[1] graph, is a set of three vertices such that each pair of them is connected by an edge, i.e., a clique of 3 vertices. According to the application, we may be interested in counting triangles or in listing all or some of them.

[1] i.e., we make no difference between (u, v) and (v, u) in $V \times V$. We also assume that G is simple ($(v, v) \notin E$ for all v, and that there is no multiple edge).

Table 1: Notation

Symbol	Description
$G(V, E)$	undirected graph where V is its vertex set and E its edge set
$V(G)$	vertex set of the graph G
$E(G)$	edge set of the graph G
$\overline{G}$	the complement of the graph G
$N(v)$	the set of neighbors of vertex v
$d(v)$	degree of vertex v
$d_{max}(G)$	the greatest vertex degree in graph G. d_{max} if there is no ambiguity.
$G[X]$	the subgraph of G induced by the set of vertices X
$\mathfrak{C}(G)$	compression of G

Existing algorithms fit into two categories: in-memory algorithms and external-memory ones. In-memory solutions are efficient when the graph can be entirely loaded in memory while external-memory solutions focus on graphs that are too large to be loaded in memory. These solutions try to ensure a low I/O cost [2, 3, 9, 12, 18, 20, 33] and are generally extensions of in-memory solutions. Most external-memory solutions are approximate solutions that do not find all the triangles.

The main in-memory solutions are surveyed in [22] where the authors studied space complexity of the existing algorithms and highlighted its importance in complex network study. In [30], the authors present an intensive experimental evaluation of various in-memory algorithms for counting and listing all triangles in a graph. In [28], existing in-memory solutions are described within a unified framework that simplifies understanding of the difference between existing solutions. The authors of [28] classify existing solutions into two approaches according to the triangle finding method: the neighborhood intersection approach and the adjacency testing approach. In the following, we review existing solutions according to this taxonomy and using the same names of algorithms as in [29], [22] and [28].

1.1 Neighborhood Intersection Approach

Algorithms in this category find triangles by parsing edges and computing intersections between the sets of neighbors of adjacent vertices. A representative algorithm is *Edge-Iterator* (see Algorithm 1) which can achieve $\mathcal{O}(md_{max}) \subseteq$

$\mathcal{O}(mn) \subseteq \mathcal{O}(n^3)$ time and $\mathcal{O}(m+n)$ space [22].

Algorithm 1: Edge-Iterator

```
Data: graph G = (V, E).
Result: all triangles of G.
begin
    for {u, v} ∈ E do
        foreach w ∈ N(u) ∩ N(v) do
            output triangle {u, v, w}
        end
    end
end
```

Algorithm *K3* [8] is a variant of *Edge-Iterator* that processes the vertices in decreasing degree order. Algorithm *Forward* [30] is a time enhancement of *Edge-Iterator* that computes intersection of just a subset of neighborhoods by using an orientation of the graph. Algorithm *Forward-Hashed* [30] is a refinement of *Forward* using hash sets for neighborhoods to compute intersection in constant time. Algorithm *Compact-Forward* [22] is an extension of *Forward* that reduces its space complexity.

1.2 Adjacency Testing Approach

In this approach, algorithms parse vertices and test the adjacency between each pair of neighbors of the current vertex. The representative algorithm is called *Node-Iterator* (see Algorithm 2). It can be implemented to achieve $\mathcal{O}(md_{max}) \subseteq \mathcal{O}(mn) \subseteq \mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space [22]. Enhancements of this algorithm can be obtained by ordering the vertices. Algorithm *Node-Iterator-Core* [1, 30] processes first the vertex with the lowest degree, and then removes it. This order can be obtained, in linear-time, by computing the core numbers of the graph[2]. *Node-Iterator-Core* can achieve $\mathcal{O}(m^{\frac{3}{2}})$. Hash sets are also used to enhance adjacency testing such as in algorithm *Node-Iterator-Hashed* [30].

A related technique called *Tree-Lister* [19] can also achieve $\mathcal{O}(m^{\frac{3}{2}})$. It computes a covering tree, checks for each edge (u, v) that does not belong to the

[2]The k-core of a graph is the largest node induced subgraph with a minimum degree of at least k.

tree if $\{u, v, father(v)\}$ is a triangle, removes the tree, and iterates.

Algorithm 2: Node-iterator

```
Data: graph G = (V, E).
Result: all triangles of G.
begin
    for v ∈ V do
        forall pairs of neighbors {u, w} of v do
            if {u, w} ∈ V then
                output triangle {u, v, w}
            end
        end
    end
end
```

Our approach is completely different from the related work as we use compressed graphs. In fact, we propose an exact algorithm that lists all triangles on the compressed graph without decompressing it. We conduct extensive experimental studies to evaluate the scalability and effectiveness of our algorithm on real large dense/sparse graphs. We also compare our approach with the most efficient algorithms of the literature to attest its efficiency.

The rest of the paper is organized as follows: Section 2 presents the compression algorithm used to summarize input graphs. Section 3 describes our approach: the exact triangle listing algorithm that operates on the compressed graphs. Section 4 presents experimental results of the proposed framework on dense and sparse graph datasets. The final section draws some conclusions and presents future work.

2 Preliminaries

As mentioned before, we propose an exact algorithm that lists triangles on a compressed graph without decompressing it.

With the era of big graphs, graph summarizing/compression is beginning to get the attention it deserves. However, there is a need to explore this topic further. The web graph is the first large graph subject to compression with the work of Boldi and Vigna [4]. Graph compression methods differ by the amount of graph properties preserved by compression, and are application dependent. Graph compression may be used to reduce the storage requirement of the graph, however, its main challenge is to obtain a compressed representation of the graph that can be used, instead of the original graph, by the intended application (analysis, mining, comparison, etc.). In [7], the authors propose an algorithm that finds all frequent subgraphs in a database of large graphs where the graphs are summarized by grouping the vertices that have the same label into supervertices. In [13], the authors observe that users typically adopt a type

Q of queries when querying a data graph G. They propose a graph compression that ensures that each query in Q returns the same result when applied to G and when applied to the compression of G. In [21], the authors approximate the similarity between two large graphs by a similarity measure between compressed versions of these graphs. They use modular decomposition [14, 25] to compress the graphs. A modular decomposition of a graph consists in finding within the graph all the sets of vertices that share the same neighborhood. These sets of vertices are called *modules*.

Definition 1 *A module of a graph $G = (V, E)$ is a set $M \subseteq V$ of vertices where all vertices in M have the same neighbors in $V \setminus M$.*

The empty set, the singletons, and the vertex set $V(G)$ satisfy the definition of a module: they are called *trivial modules*. A graph that has only trivial modules is called a *prime graph*.
By replacing each module by a vertex (a supervertex), we can compress a graph step by step until no further compaction is possible. However, modules may overlap leading to a non unique compression process. To avoid this and ensure the uniqueness of the compressed graph, compression is achieved only with modules that do not overlap. These modules are called *strong modules* [25].

Definition 2 *A module is strong if it does not overlap any other module.*

Two strong modules M_1 and M_2 are either adjacent or non adjacent [25]. M_1 and M_2 are adjacent if every vertex of M_1 is adjacent to every vertex of M_2 and nonadjacent otherwise [25].

A strong module M of G can take one of the following types [25]:

- **Prime:** the induced subgraph $G[M]$ is connected and its complement $\overline{G}[M]$ is also connected.
- **Series:** the induced subgraph $G[M]$ is a clique, i.e., a complete graph.
- **Parallel:** All strong modules contained in M are non-adjacent to each other in the induced subgraph $G[M]$, i.e., $G[M]$ is not connected and $\overline{G[M]}$ is a complete graph.

We note that the type of module gives sufficient information about the structure of the vertices within the module except for prime modules for which we need to store adjacency information. Thus, we keep the edges between the vertices of a prime module. Consequently, the resulting compressed graph retains all the structural information of the original graph while requiring smaller storage space.

Figure 1 illustrates the compression process. In this figure, (a) is the original graph. Each subsequent figure shows the processing of a module. The last graph in the figure, i.e., (f) is the final compressed graph. It is a prime module and cannot be compressed further because we must keep its edges. However, graphs

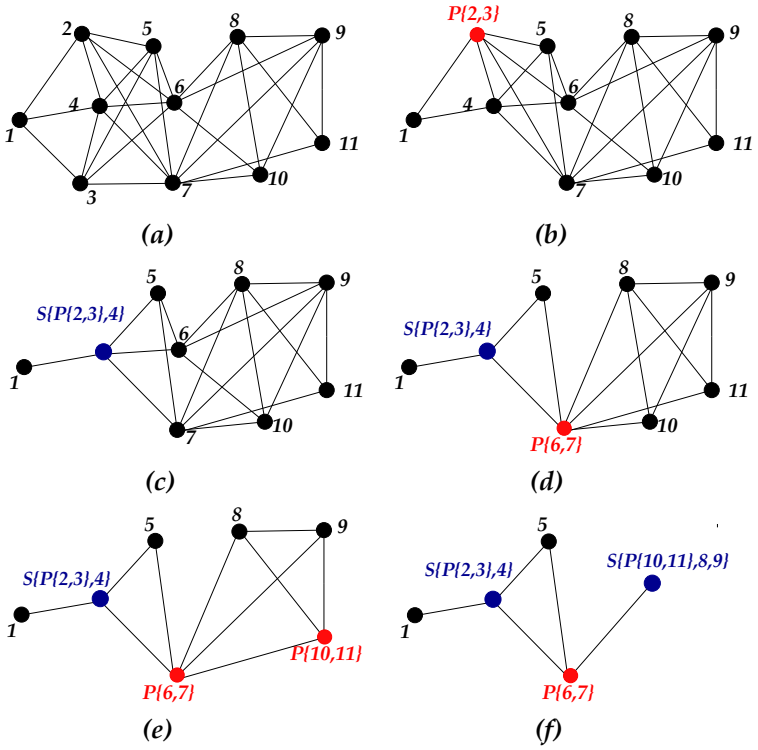


Figure 1: Compressed graphs obtained by compacting the strong modules of G.
Each module is a set of vertices (that may be modules). Each module is prefixed by its type: S for a series module, P for a parallel module and Pr for a prime module.

can be compressed to a single vertex especially when the last compression step concerns a series or a parallel module.

Modular decomposition of graphs has been the subject of several investigations mainly for its use in studying the properties of graphs and their classes [14, 26]. The first algorithm proposed for decomposing a graph into modules dates back to 1972 and runs in $\mathcal{O}(n^4)$ [10]. This result was followed by several improvements on both runtime complexity and simplicity of implementation of algorithms. The latest results provide quasi-linear algorithms that run in $\mathcal{O}(n+m\log n)$ [11, 17, 24] and more recently a linear-time algorithm that runs in $\mathcal{O}(n+m)$ [15, 32]. A comprehensive review of these algorithms can be found in [16]. Existing algorithms construct a modular decomposition tree of a graph. this is a tree where the leaves are the vertices of the graph and the internal nodes are the modules. The tree shows clearly the content of each module. However, it does not give the edges between the children of prime modules. We modified the algorithm of [6, 15] to compute the compressed graph by replacing

modules by vertices and inserting edges between the vertices of prime modules. This algorithm needs $\mathcal{O}(n+m)$ computation steps where n is the number of vertices and m the number of edges in the graph. It is based on the notion of factorizing permutations [6] which is a particular permutation of the set of vertices in which strong modules appear consecutively. This concept and the detailed algorithm are described in [6, 15]. In this paper, we do not introduce a new method for computing the modules of a graph. However, we use this concept with existing algorithms to compress a graph as depicted in Figure 1.

3 Proposed Framework

This section presents a new algorithm for exact triangle listing in large graphs. Throughout the rest of the paper, we will refer to it by CGT for Compressed Graph Triangulation. The key idea of CGT is to process the compressed version of the graph instead of the original graph for space saving. Compression is achieved off-line with modular decomposition as described in the previous section. So, in this section, we focus on how the framework uses the compressed graph to list the triangles of the original graph. The framework relies on two main steps: (1) parsing of the compressed graph, and (2) listing its triangles. We describe these steps in detail in the following sections.

3.1 Parsing the Compressed Graph

One of the key ideas of our algorithm is representing neighbors within a hash table where each edge is represented by one vertex. We achieve this by partitioning the compressed graph into a set of edge disjoint stars defined as follows:

Definition 3 *(Ordered-Star Structure (OST)) An Ordered-Star Structure $s = (r, L)$ is an attributed, single-level, rooted tree where r is the root vertex and L is the set of leaves. Edges exist between r and any vertex in L and no edge exists among vertices in L and $\forall l \in L, r < l$.*

To construct OSTs, the n vertices of the graph G must be numbered from 1 to n, independently of their labels that may not be integers and also not unique. The order used in Definition 3 is the numerical order of these numbers.

For any vertex v in a graph G, we can extract the corresponding OST s_v as follows: $s_v = (v, L_v)$ where $L_v = \{u | (u, v) \in E \text{ and } u < v\}$. Thus, we can derive at most $n-1$ $OSTs$ from a graph containing n vertices. We call the set of OSTs of a graph G the ordered star representation of G, denoted by $SS(G)$. The partitioning process is detailed in Algorithm 3. The algorithm takes as input a graph as a list of edges, and returns the set $SS(G)$ of OSTs of the graph in the form of a hash table.

We note that OSTs are edge disjoint stars. So, there is no edge overlapping between OSTs. The sum of all edges in the set of OSTs is equal to the number of

edges in the original graph. $OSTs$ allows us to represent neighborhood efficiently as each edge is represented by only one vertex.

Algorithm 3: Ordered Star Structure Construction(SS)

Data: A graph G as a set of edges $E(G) = \langle e_1, e_2, \ldots, e_m \rangle$.
Result: Hash table of the OSTs of G: $SS(G) = \{s_1, s_2, \ldots, s_l\}$.
begin
  $SS \leftarrow \emptyset$;
  foreach $e = (u, v) \in E(G)$ **do**
    if $u < v$ **then**
      insertIntoHash_SS $(u, L = L \cup v)$
    end
    else
      insertIntoHash_SS $(v, L = L \cup u)$
    end
  end
  return SS
end

Figure 2 shows an example of a graph and its partitioning into OSTs. In this figure, the black vertices represent root vertices and the white vertices represent their leaves.

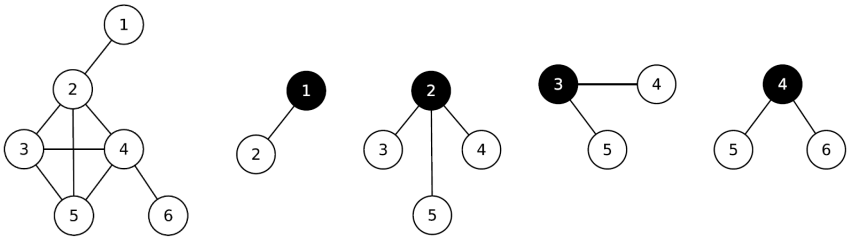


Figure 2: Graph partitioning into OSTs

3.2 Triangle Listing in the Compressed Graph

The aim of our algorithm is to compute the list of all triangles of a graph G by handling only $\mathfrak{C}(G)$. In our framework, we first partition $\mathfrak{C}(G)$ into OSTs using Algorithm 3. We obtain the hash table $SS(\mathfrak{C}(G))$. Every triangle in $\mathfrak{C}(G)$ can be classified into one of the following types:

- Type I: the three vertices of the triangle belong to the same module. We will denote the set of this type of triangles $\triangle_I$.
- Type II: the triangle involves two modules. This type of triangles concerns prime modules. We will denote the set of this type of triangles $\triangle_{II}$.

- Type III: the triangle involves three modules. It is a triangle of modules. This type of triangles also concerns prime modules. We will denote the set of this type of triangles $\triangle_{III}$.

We first describe how we obtain $\triangle_I$, $\triangle_{II}$, and $\triangle_{III}$ and then how we derive the triangles of G from them. We note that each of $\triangle_I$, $\triangle_{II}$, and $\triangle_{III}$ is a set of sets of vertices. Each of these subsets is either a singleton or a pair of connected vertices, i.e., a potential partial triangle. The key idea of the triangle listing algorithm is to rely on set operations applied when parsing the OSTs of the compressed graph. So, we first give the main function of our framework that implements a commutative product of sets of sets and allows us to construct triplets and pairs of connected vertices, i.e., triangles or partial triangles from adjacent modules. This product is denoted $\prod_{\triangle}$ and is detailed in Algorithm 4 that takes, as parameters, sets of potential partial triangles, i.e., pairs of connected vertices or singletons, and outputs the set of corresponding triangles and potential partial triangles.

Algorithm 4: Finding Connected Vertices ($\prod_{\triangle}$)

Data: sets of sets of connected vertices A_1, A_2, $\cdots$, A_t where $t \geq 2$.
Result: set of triangles and potential partial triangles.
begin
  Let $A_i = \{A_{i1}, A_{i2}, \cdots, A_{i|A_i|}\}$;
  output $\{\{a_1, a_2, a_3\} | a_1 \in A_{ij}, a_2 \in A_{i'j'}, a_3 \in A_{i''j''}$ where $i \neq i'$ or $i \neq i''$ or $i' \neq i''\}$;
  return $A_1 \cup A_2 \cup \cdots \cup A_t \cup \{\{a_1, a_2\} | a_1 \in A_{ij}, a_2 \in A_{i'j'}$ *where* $i \neq i'\}$
end

Example:
Let us consider $A = \{\{1\}, \{3, 4\}\}$ and $B = \{\{6, 7\}, \{8\}\}$ entries of Algorithm 4. A and B are, for example, the sets of potential partial triangles from two adjacent modules in the compressed graph. $\prod_{\triangle}(A, B)$, will output triangles $\{3, 4, 6\}$, $\{3, 4, 7\}$, $\{3, 4, 8\}$, $\{1, 6, 7\}$, $\{3, 6, 7\}$, $\{4, 6, 7\}$ and return the potential partial triangles: $\{1\}$, $\{3, 4\}$, $\{6, 7\}$, $\{8\}$, $\{1, 6\}$, $\{1, 7\}$, $\{1, 8\}$, $\{3, 6\}$, $\{3, 7\}$, $\{3, 8\}$, $\{4, 6\}$, $\{4, 7\}$ and $\{4, 8\}$.

3.2.1 Triangles of type I

A module, i.e., a vertex in $\mathfrak{C}(G)$, may contain potential partial triangles of the original graph G. As detailed in Algorithm 5 that takes as input a module M and returns all the triangles and potential partial triangles inside it, $\mathfrak{C}(G)$ has

3 types of modules: serial, parallel and prime. We consider four cases:

Algorithm 5: Listing Triangles in a Module ($List_I$)

Data: a module M.
Result: set of triangles of type I in M augmented with potential partial triangles.

begin
  $\triangle_I \leftarrow \emptyset$;
  switch *type of* M **do**
    case *a simple vertex* **do**
      $\triangle_I \leftarrow \{M\}$;
    case *a series module* **do**
      Let $M = \{A_1, A_2, \cdots, A_t\}$;
      $\triangle_I \leftarrow \triangle_I \cup \prod_{\triangle}(List_I(A_1), List_I(A_2), \cdots, List_I(A_t))$;
    case *a parallel module* **do**
      foreach $m \in M$ **do**
        $\triangle_I \leftarrow \triangle_I \cup List_I(m)$;
      end
    case *a prime module* **do**
      $\triangle_I \leftarrow \triangle_I \cup List_{III}(M)$;
      $\triangle_I \leftarrow \triangle_I \cup List_{II}(M)$;
  end
  return $\triangle_I$
end

1. **Case of a simple vertex v:** If the considered module is just a simple vertex, i.e., v, then we return the set $\{v\}$ (a potential partial triangle).

2. **Case of a parallel module:** The components of a parallel module are not connected. So, a triangle is not possible between them. However each of them is a potential element of a triangle of type II or type III. So, the algorithm is launched recursively on each component and we return the union of obtained results (see details in Algorithm 5).

   For example, in Figure 1(f), the parallel module $P\{10, 11\}$ consists of two elements. So, the algorithm will return the union of its result on each element, i.e., $\{\{10\}, \{11\}\}$. Each of $\{10\}$ and $\{11\}$ are potential partial triangles for the graph.

3. **Case of a series module:** In case of a series module, we return all the combinations of triangles from different elements of the module returned by calling recursively the algorithm on each of them. Furthermore, as each element may also be involved in a triangle of type II or type III, we also return the potential partial triangles using $\prod_{\triangle}$ detailed in Algorithm 4.

As an example, consider again Figure 1(f). The series module $S\{P\{10,11\},8,9\}$ is composed of three vertices: two simple vertices and a parallel module $P\{10,11\}$. We see that vertices 8 and 9 are in series with the parallel vertex and therefore with its elements (10 and 11). The series vertex returns the triangles $\{8,9,10\},\{8,9,11\}$ and the set of potential partial triangles consisting of one or two elements: $\{8\}$, $\{9\}$, $\{10\}$, $\{11\}$, $\{8,9\}$, $\{8,10\}$, $\{8,11\}$, $\{9,10\}$, and $\{9,11\}$.

4. **Case of a prime module**: In this case, we use Algorithm 7 to compute the set of triangles of modules $\triangle_{III}$, i.e., triangles of type III, and Algorithm 6 to compute $\triangle_{II}$ the set of triangles of type II as illustrated in Sections 3.2.2 and 3.2.3 respectively.

3.2.2 Triangles of type II

Adjacent modules may embed triangles of the original graph. This situation occurs only with a prime module. To find the set of triangles from two adjacent modules, the prime module is parsed edge by edge as illustrated in Algorithm 6 which returns the set of connected vertices by this edge using the commutative product of sets, $\prod_{\triangle}$, detailed in Algorithm 4.

For example, for the adjacent modules 1 and $S\{P\{2,3\},4\}$ in Figure 1(f), Algorithm 6 returns triangles $\{1,2,4\}$ and $\{1,3,4\}$ as well as the set of potential partial triangles $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, $\{2,4\}$, $\{3,4\}$, $\{1,4\}$, $\{1,2\}$, and $\{1,3\}$.

Algorithm 6: Listing Triangles of type II ($List_{II}$)

Data: A prime module M.
Result: List of triangles of type II.
begin
  $\triangle_{II} \leftarrow \emptyset$;
  foreach *edge* $(m_i, m_j) \in M$ **do**
    $\triangle_{II} \leftarrow \triangle_{II} \cup \prod_{\triangle}(List_I(m_1), List_I(m_2))$
  end
  return $\triangle_{II}$
end

3.2.3 Triangles of type III

From a triangle of modules, i.e., 3 adjacent modules, we list triangles involving vertices from each module participating in the triangle as well as potential partial triangles. To do so, we first compute the triangles of modules by parsing the OSTs of the prime module and intersecting their neighbors as illustrated by the first loop of Algorithm 7. Then, the second loop of this algorithm uses Algorithm 4 to list the triangles of the original graph that are embedded in the triangles of modules.

As an illustration, consider again Figure 1(f) that contains a triangle of modules involving vertices 5, $P\{6,7\}$, and $S\{P\{2,3\},4\}$. Algorithm 7 returns

triangles $\{2,5,6\}$, $\{2,5,7\}$, $\{3,5,6\}$, $\{3,5,7\}$, $\{4,5,6\}$ and $\{4,5,7\}$.

Algorithm 7: Listing Triangles of type III ($List_{III}$)

```
Data: A prime module M.
Result: List of triangles of type III.
begin
    △_M ← ∅;
    foreach s_i ∈ SS(M) do
        foreach v_i ∈ s_i.L do
            δ ← s_i.L ∩ s_{v_i}.L;
            foreach u ∈ δ do
                △_M ← △_M ∪ {u, s_i.r, v_i};
            end
        end
    end
    △_III ← ∅;
    foreach δ_i = {m_1, m_2, m_3} ∈ △_M do
        △_III ← △_III ∪ ∏_△(List_I(m_1), List_I(m_2), List_I(m_3));
    end
    return △_III
end
```

The compressed graph $\mathfrak{C}(G)$ is a module. So, to have its list of triangles, we just need to compute the list of triangles of type I, i.e., $\triangle_I$ by using Algorithm 5.

3.3 Complexity

Theorem 1 *CGT lists all triangles in $G = (V, E)$ in time $\mathcal{O}(t + m + n + n'd'_{max} + m')$ and space $\mathcal{O}(n + m)$ where t is the number of triangles in G, n and m are, respectively, the number of vertices and the number of edges of G and n', m' and d'_{max} are respectively the number of vertices, the number of edges and the maximum degree of $\mathfrak{C}(G)$.*

Proof:

Let G be a graph of n vertices and m edges. We assume that the graphs are available in their compressed form and we do not include the cost of compression in this analysis. We recall that we can obtain $\mathfrak{C}(G)$ in $\mathcal{O}(n+m)$ time complexity and $\mathcal{O}(n + m)$ space cost [27]. Let n' be the number of vertices in $\mathfrak{C}(G)$ and m' its number of edges. Let d'_{max} denotes the maximum vertex degree in $\mathfrak{C}(G)$ and let t be the number of triangles in G. The main part of the complexity of CGT is related to the execution of Algorithm 4, i.e., the complexity of function $\prod_{\triangle}$. $\prod_{\triangle}$ finds pairs and triads of connected vertices from the sets of vertices it takes as parameters using a cartesian product of sets. Whatever the size of the sets it takes as parameters and the number of times it is called, the final result includes the triangles of G, the set of edges of G and its set of vertices. So the

time complexity of this part of CGT is $\mathcal{O}(t+m+n)$. To process series and parallel modules, the algorithm executes set unions. As only pairs of vertices and singletons are stored, the total cost of these unions is equal to the size of the resulting set, i.e., $\mathcal{O}(m+n)$. Processing a prime module consists in using a *Node-Iterator* like algorithm to parse its vertices and computing neighborhood intersection in $\mathcal{O}(n'd'_{max})$ to find triangles of type III, and parse the edges in $\mathcal{O}(m')$ to find triangles of type II. Consequently, the total time cost is $\mathcal{O}(t+m+n+n'd'_{max}+m') \in \mathcal{O}(n^3)$ as t is $\mathcal{O}(n^3)$ and $n'd'_{max} \leq nd_{max} \leq nm$.

Space complexity is obtained by a content-array representation of the compressed graph. It is similar to the adjacency array representation with the difference that the array contains the elements of a vertex (i.e., the constituents of a module) and not its neighbors. Each entry of this array has two components: the label of the vertex and the index of its first element in the array. The neighbors are contained in hash maps using OSTs. Only prime modules need to have a hash map for neighboring information. Use of OSTs allows us to store each edge once. Consequently, storage of the compressed graph needs $2(n+\#M)+\#P*\#V_P+m'$ where $\#M$, which is $\mathcal{O}(n)$, is the number of modules in the compressed graph, $\#P$ is the number of prime modules and $\#V_P$ is the number of vertices in a prime module. The worst case is that all the modules are prime, i.e., $\#M=\#P$. We note that the product $\#P*\#V_P$ cannot be greater than $2n$ as the number of elements cannot exceed the number of vertices augmented with the number of modules. This gives a space cost of $6n+m'$. The algorithm also need an extra space to store $\triangle_I$, $\triangle_{II}$ and $\triangle_{III}$ which contain in the worst case all the pairs of connected vertices and all the vertices, i.e., $2m+n$ space cost.

Total space cost is then $7n+2m+m' \leq 7n+3m \subset \mathcal{O}(n+m)$ It is important to note that even if compression reduces storage space of the graphs as we will show in the next section, our memory representation does not reduce central memory space of the graph. However, it remains comparable to the other solutions. Furthermore, this graph representation allows us to store the labels of the vertices.

4 Experiments

In this section, we evaluate our algorithm on various real-word datasets consisting of sparse and dense graphs. We also compare our algorithm with the main solutions proposed in the literature. The aim of this comparison is to show that beside the storage efficiency of our approach due to graph compression, it has an execution time comparable to the fastest algorithms. We first describe the experimental environment, then the dataset, and finally we present our results and discuss them.

4.1 Experimental Setup

In this section, we present the computational experiments conducted by running our algorithm. We evaluate execution time performance over different types of graphs and scalability of the approach in terms of graph size. To compress the input graph, we use an extension of the algorithm proposed in [6, 15] that computes the modular decomposition of a graph in linear time. So, we compress the input graph in $O(n+m)$ time, where n is the number of vertices and m the number of edges of the graph.

The experiments are performed on a 2.80 GHz $Intel(R)$ $Core(TM)$ $i7-2640M$ 64 bits laptop with 8 GB of RAM running Linux. The algorithm is implemented in C++ using the Standard Template Library (STL) and the GNU compiler *gcc* version 4.8.1 set to the highest optimization level ($-O3$ optimization option).

We compared our CGT algorithm with the existing efficient solutions of triangle listing, namely Edge-Iterator, Edge-Iterator-Hashed [30], Forward and Forward-Hashed [30], Compact-Forward [22], K3 [8], Node-Iterator and Node-Iterator-Core [1, 30]. These algorithms were reviewed in Section 1. All these algorithms are implemented in the same context as ours in [28], i.e., C++, using the gcc compiler with the optimizer option $O3$. The algorithms were implemented for counting triangles, and we modified them to list triangles.

We conduct experiments on the sparse and dense datasets described below by considering two metrics: runtime and reduction rate.

4.2 Datasets

We evaluate the performance of our algorithm over large dense/sparse real datasets from the PPI-RI database of biochemical data[3] [5] and graphs from the SNAP project at Stanford [23]. We considered both dense and sparse graphs with dimensions varying from less than ten thousand vertices to more than a million vertices, and from less than one hundred thousand edges to more than one million edges. The characteristics of these datasets are shown in Table 2. Their description is as follows:

- *The PPI database (RI database)* [5, 31]: This dataset contains graphs describing the known and predicted protein interactions. The graphs describe the following organisms: Mus musculus, Saccaromyces cerevisiae, Caenorhabditis elegans, Drosophila melanogaster, Takifugu rubipres, Danio rerio, Xenopus tropicalis, Bos taurus, Rattus norvegicus, and Homo sapiens. They are large dense graphs with up to 332,458 edges. Table 3 describes them by giving the number of vertices $|V|$ and number of edges $|E|$.

  For most PPI graphs, computation of the number of triangles is provided by our algorithm and verified by another existing algorithm. We also report the size on disk (in megabytes) of these datasets in the format in

[3] http://ferrolab.dmi.unict.it/ri/ri.html#description

which we obtained them (a list of edges in plain text) and their compressed graph version computed by our algorithm.

Dataset	Type	$avg\|V\|$	$avg\|E\|$	$max\|V\|$	$max\|E\|$
PPI	dense	7,828.9	107,134.8	12,578	332,458
SNAP	sparse	1,706,900	2,812,893.75	2,394,385	5,021,410

Table 2: Graph Dataset Characteristics. $avg|V|$: average number of vertices. $avg|E|$: average number of edges. $max|V|$: maximum number of vertices. $max|E|$: maximum number of edges.

Graph Name	n	m	Δ	m/n	$Size(G)$	$Size(\mathfrak{C}(G))$	Compression time (sec)
Danio_rerio	5,723	51,464	93,784	8.99	0.49M	0.29M	0.112
Takifugu_rubipres	5,872	54,154	85,873	9.22	0.52M	0.30M	0.139
Saccaromyces_cerevisiae	6,139	332,458	870,978	54.15	3.2M	1.7M	0.309
Xenopus_tropicalis	6,153	59,538	87,187	9.67	0.57M	0.33M	0.113
Caenorhabditis_elegans	6,175	52,368	78,597	8.48	0.50M	0.30M	0.142
Bos_taurus	8,474	84,468	150,146	9.96	0.82M	0.47M	0.203
Drosophila_melanogaster	8,625	78,932	98,244	9.15	0.76M	0.45M	0.125
Rattus_norvegicus	8,766	79,864	84,757	9.11	0.77M	0.47M	0.204
MusMusculus	9,784	104,322	186,306	10.66	1M	0.58M	0.285
Homo_sapiens	12,578	173,780	227,982	13.81	1.8M	1M	0.240

Table 3: A summary of the dense graph dataset used in our experiments, showing for every graph, the number of vertices (n), the number of edges (m), the number of triangles in the graph (Δ), the ratio m/n, the size of the original graph on disk (stored as a list of edges in plain text)($Size(G)$), the size of the compressed graph on disk (stored as a list of strong modules augmented with the edges of prime modules in plain text)($Size(\mathfrak{C}(G))$), and the compression time.

Figure 3 gives the topography of the dense dataset in terms of modules.

- *SNAP dataset*: This is a substantial collection of graphs describing very large networks, including social networks, communication networks, and transportation networks. In our experiments, we use *California road network* graphs and *Wikipedia* talk (communication) networks.
  - California road networks describe the road networks of California. Intersections and endpoints are represented by vertices and the roads connecting these intersections or road endpoints are represented by undirected edges.
  - Wikipedia talk graph contains all the users and discussion from the inception of Wikipedia till January 2008. Vertices in the network represent Wikipedia users, and a directed edge from vertex i to vertex j means that user i edited a talk page of user j at least once.

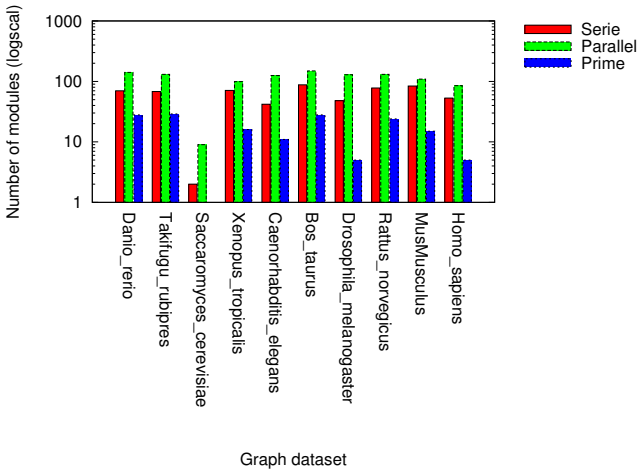


Figure 3: Module-based description of the dense dataset.

This dataset contains four large sparse graphs of up to 2,394,385 vertices and 5,021,410 edges. For most SNAP graphs, the exact triangle count is provided by the source (which we have verified); in other cases, we compute the exact count using our algorithm. We also report the size on disk of these graphs and their corresponding compressed graphs. Table 4 summarizes the characteristics of these graphs.

Dataset	n	m	Δ	m/n	$Size(G)$	$Size(\mathfrak{C}(G))$	Compression time (sec)
roadNet-CA	1,965,206	2,766,607	120,676	1.40	84M	58M	1080.405
roadNet-PA	1,088,092	1,541,898	67,150	1.41	46M	30M	340
roadNet-TX	1,379,917	1,921,660	82,869	1.39	59M	39M	449
wiki-Talk	2,394,385	5,021,410	9,203,519	2.09	66M	64M	73221.801

Table 4: A summary of the sparse graph dataset used in our experiments, showing for every graph, the number of vertices (n), the number of edges (m), the number of triangles in the graph (Δ), the ratio m/n, the size of the original graph on disk (stored as a list of edges in plain text)($Size(G)$), the size of the compressed graph on disk (stored as a list of supervertices augmented with the edges of prime modules, in plain text)($Size(\mathfrak{C}(G))$), and the compression time.

Figure 4 gives the topography of the sparse dataset in terms of modules.

As shown in Figures 3 and 4, the sparse graphs are characterized by a large number of parallel modules when compared to the number of series and prime modules. The numbers of the different kinds of modules are more balanced in the dense dataset.

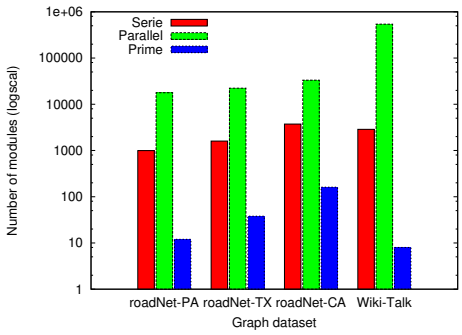


Figure 4: Module-based description of the sparse dataset.

4.3 Results

We first compared the runtime of the different algorithms. Figure 5 plots the results on both datasets. We note here that the graphs are compressed offline and that the time necessary to compress the graphs is not considered in the comparison.

Figure 5 shows clearly that our approach performs better when compared to the existing algorithms. For dense graphs, our algorithm is the most efficient in almost all datasets. For the sparse datatset, it achieves better than the other solutions except for the *roadNet-CA* graph for which it achieves almost as well as the fastest algorithm *Forward-Compact*. We can see in Figure 4 that the *roadNet-CA* graph has the greatest number of prime modules that are the most time-consuming in *CGT*.

We also studied the impact of compression on the runtime of *CGT*. Given a graph $G(V, E)$ and its compressed graph $\mathfrak{C}(G)$, the reduction rate RR of G is given by $RR = \frac{RR_{edge}}{RR_{vertex}} * 100\%$, where RR_{vertex} is the reduction rate of vertices $RR_{vertex} = ((|V(G)| - |V(\mathfrak{C}(G)|)/|V(G)|)$ and RR_{edge} is the reduction rate of edges $RR_{edge} = ((|E(G)| - |E(\mathfrak{C}(G)|)/|E(G)|)$. The average reduction rate of sparse graphs is 1.43% and the average reduction rate of dense graphs is 51.76%.

Figure 6 shows the impact of the reduction rate on runtime in sparse and dense graphs. We see that when reduction rate grows, runtime decreases in both sparse and dense graphs.

5 Conclusion and future work

Triangle listing is an important graph problem that can be encountered in several real word applications. Actually, this problem receives an increasing attention especially for large graphs. In this paper, we show that tackling this challenge goes also through the representation of the graphs. So, we propose an algorithm that lists triangles in a compressed graph without decompressing it. Our experimental evaluation shows clearly that our approach is space efficient and has a time complexity comparable to the fastest algorithms. Furthermore, our solution facilitates parallelism because we partition the compressed graphs into OSTs that can be handled by different machines with independent processing/discovering of triangles inside and outside each module. This can be envisaged as a future work of this paper. Also, the actual version of the solution works in-memory as we assume that the compressed file fits in memory. So, an interesting extension is to design an external memory solution by allying partitioning and compression and adequate solutions to decrease I/O costs. This last prospect also directs us towards working on external-memory solutions for graph compression. In fact, we relied on an in-memory algorithm to compress the graphs. Even if compression is achieved offline, it cannot be used on graphs that do not fit in memory. Consequently, we have to work on new methods to compress large graphs or adapt existing ones. It is also important to work on the representation of the compressed graph in memory as it plays an important role in space efficiency of the proposed framework.

Acknowledgements: The authors would like to thank the anonymous reviewers for their valuable comments on earlier drafts of this paper.

References

[1] Batagelj, V., Zaveršnik, M.z.: Fast algorithms for determining (generalized) core groups in social networks. Advances in Data Analysis and Classification **5**(2), 129–145 (2011)

[2] Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient algorithms for large-scale local triangle counting. ACM Trans. Knowl. Discov. Data **4**(3), 13:1–13:28 (2010). DOI 10.1145/1839490.1839494. URL http://doi.acm.org/10.1145/1839490.1839494

[3] Björklund, A., Pagh, R., Williams, V., Zwick, U.: Listing triangles. In: J. Esparza, P. Fraigniaud, T. Husfeldt, E. Koutsoupias (eds.) Automata, Languages, and Programming, *Lecture Notes in Computer Science*, vol. 8572, pp. 223–234. Springer Berlin Heidelberg (2014). DOI 10.1007/978-3-662-43948-7_19. URL http://dx.doi.org/10.1007/978-3-662-43948-7_19

[4] Boldi, P., Vigna, S.: The webgraph framework i: Compression techniques. In: Proceedings of the 13th International Conference on World Wide Web,

WWW '04, pp. 595–602. ACM, New York, NY, USA (2004). DOI 10.1145/988672.988752. URL http://doi.acm.org/10.1145/988672.988752

[5] Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinformatics **14** (2013)

[6] Capelle, C., Habib, M., de Montgolfier, F.: Graph decompositions and factorizing permutations. Discrete Mathematics & Theoretical Computer Science **5**(1), 55–70 (2002)

[7] Chen, C., Lin, C.X., Fredrikson, M., Christodorescu, M., Yan, X., Han, J.: Mining graph patterns efficiently via randomized summaries. Proc. VLDB Endow. **2**(1), 742–753 (2009). DOI 10.14778/1687627.1687711. URL http://dx.doi.org/10.14778/1687627.1687711

[8] Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1), 210–223 (1985). DOI 10.1137/0214017. URL http://dx.doi.org/10.1137/0214017

[9] Chu, S., Cheng, J.: Triangle listing in massive networks. ACM Trans. Knowl. Discov. Data **6**(4), 17:1–17:32 (2012). DOI 10.1145/2382577.2382581. URL http://doi.acm.org/10.1145/2382577.2382581

[10] Cowan, D., James, I., Stanton, R.: Graph decomposition for undirected graphs. 3rd S-E Conf. on Combinatorics, Graph Theory and Computing pp. 281–290 (1972)

[11] Dahlhaus, E., Gustedt, J., McConnell, R.M.: Efficient and practical algorithms for sequential modular decomposition. Journal of Algorithms **41**(2), 360 – 387 (2001). DOI http://dx.doi.org/10.1006/jagm.2001.1185. URL http://www.sciencedirect.com/science/article/pii/S019667740191185X

[12] Dementiev, R.: Algorithm engineering for large data sets hardware, software, algorithms. Ph.D. thesis, Saarland University (2006)

[13] Fan, W., Li, J., Wang, X., Wu, Y.: Query preserving graph compression. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pp. 157–168. ACM, New York, NY, USA (2012). DOI 10.1145/2213836.2213855. URL http://doi.acm.org/10.1145/2213836.2213855

[14] Gallai, T.: Transitiv orientierbare graphen. Acta Mathematica Hungarica **18**, 25–66 (1967)

[15] Habib, M., de Montgolfier, F., Paul, C.: A simple linear-time modular decomposition algorithm for graphs, using order extension. In: Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings, pp. 187–198 (2004)

[16] Habib, M., Paul, C.: A survey of the algorithmic aspects of modular decomposition. Computer Science Review **4**(1), 41 – 59 (2010)

[17] Habib, M., Paul, C., Viennot, L.: Partition refinement techniques: An interesting algorithmic tool kit. International Journal of Foundations of Computer Science **10**(2), 147–170 (1999)

[18] Hu, X., Tao, Y., Chung, C.W.: Massive graph triangulation. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pp. 325–336. ACM, New York, NY, USA (2013). DOI 10.1145/2463676.2463704. URL http://doi.acm.org/10.1145/2463676.2463704

[19] Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM Journal on Computing **7**(4), 413–423 (1978)

[20] Kolountzakis, M.N., Miller, G., Peng, R., Tsourakakis, C.E.: Efficient triangle counting in large graphs via degree-based vertex partitioning. In: R. Kumar, D. Sivakumar (eds.) Algorithms and Models for the Web-Graph, *Lecture Notes in Computer Science*, vol. 6516, pp. 15–24. Springer Berlin Heidelberg (2010). DOI 10.1007/978-3-642-18009-5_3. URL http://dx.doi.org/10.1007/978-3-642-18009-5_3

[21] Lagraa, S., Seba, H., Khennoufa, R., M'Baya, A., Kheddouci, H.: A distance measure for large graphs based on prime graphs. Pattern Recognition **47**(9), 2993 – 3005 (2014)

[22] Latapy, M.: Main-memory triangle computations for very large (sparse (power-law)) graphs. Theoretical Computer Science **407**(1-3), 458 – 473 (2008)

[23] Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (2014)

[24] Mcconnell, R.M., Spinrad, J.P.: Ordered Vertex Partitioning. Discrete Mathematics & Theoretical Computer Science **Vol. 4 no. 1** (2000). URL http://dmtcs.episciences.org/274

[25] Möhring, R.: Algorithmic aspect of the substitution decomposition in optimization over relation, set system and boolean function. Ann. Operations Research **4**, 195–225 (1985)

[26] Möhring, R., Radermacher., F.: Substitution decomposition and connection with combinatorial optimization. Ann. Discrete Mathematics **19**, 257–356 (1984)

[27] de MONTGOLFIER, F.: Modular decomposition of graphs: theory, extensions and algorithms. Ph.D. thesis, Université des Sciences et Techniques du Languedoc (2003)

[28] Ortmann, M., Brandes, U.: Triangle listing algorithms: Back from the diversion. In: 2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014, pp. 1–8 (2014)

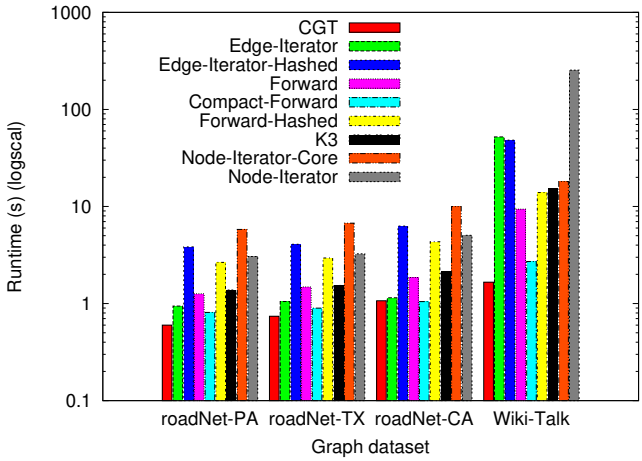
[29] Schank, T.: Algorithmic Aspects of Triangle-Based Network Analysis. Ph.D. thesis, Universität Karlsruhe (2007)

[30] Schank, T., Wagner, D.: Finding, counting and listing all triangles in large graphs, an experimental study. In: Proceedings of the 4th International Conference on Experimental and Efficient Algorithms, WEA'05, pp. 606–609. Springer-Verlag, Berlin, Heidelberg (2005). DOI 10.1007/11427186_54. URL http://dx.doi.org/10.1007/11427186_54

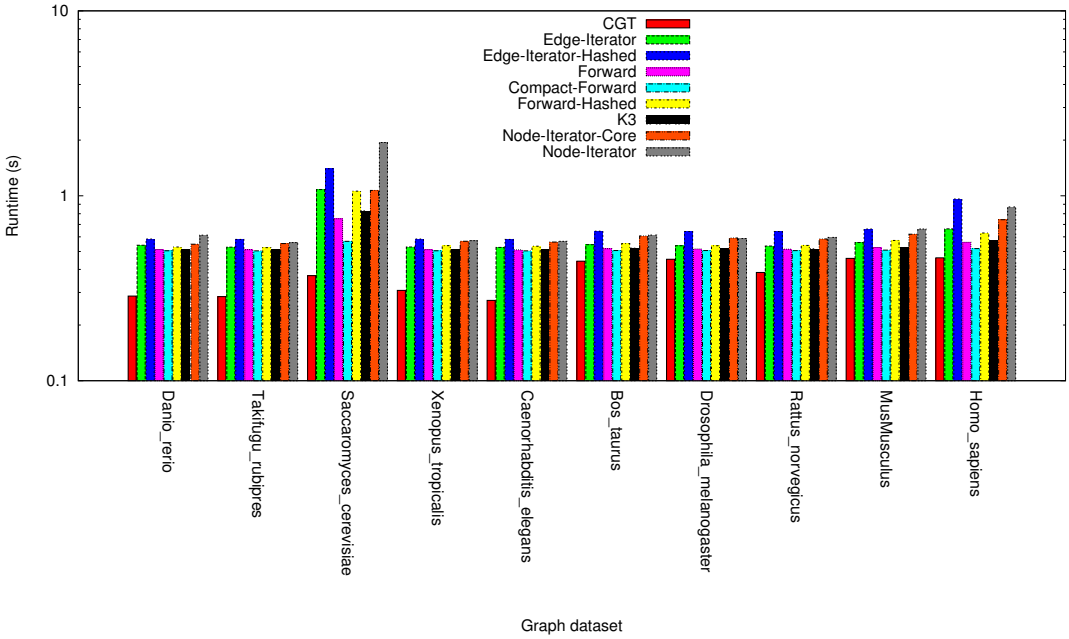
[31] Szklarczyk, D., Franceschini, A., Kuhn, M., Simonovic, M., Roth, A., Minguez, P., Doerks, T., Stark, M., Muller, J., Bork, P., Jensen, L., Mering, C.V.: The string database in 2011: functional interaction networks of proteins, globally integrated and scored. Nucleic Acids Res **39**, D561–D568 (2011)

[32] Tedder, M., Corneil, D., Habib, M., Paul, C.: Simpler linear-time modular decomposition via recursive factorizing permutations. In: L. Aceto, I. Damgrd, L. Goldberg, M. Halldrsson, A. Inglfsdttir, I. Walukiewicz (eds.) Automata, Languages and Programming, *Lecture Notes in Computer Science*, vol. 5125, pp. 634–645. Springer Berlin Heidelberg (2008). DOI 10.1007/978-3-540-70575-8_52. URL http://dx.doi.org/10.1007/978-3-540-70575-8_52

[33] Tsourakakis, C.E., Kang, U., Miller, G.L., Faloutsos, C.: Doulion: Counting triangles in massive graphs with a coin. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09, pp. 837–846. ACM, New York, NY, USA (2009). DOI 10.1145/1557019.1557111. URL http://doi.acm.org/10.1145/1557019.1557111

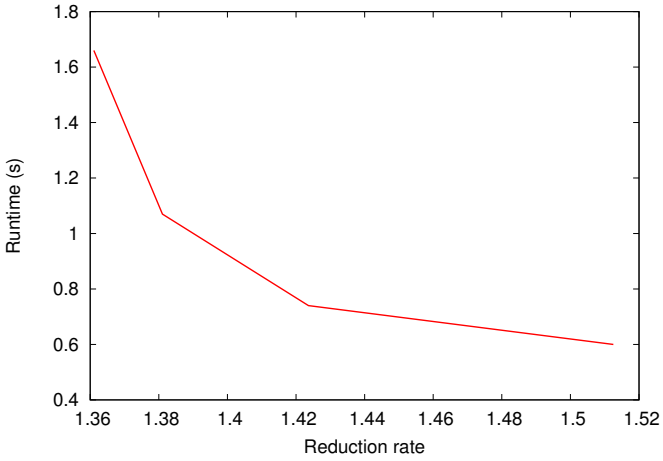


(a) On the sparse dataset

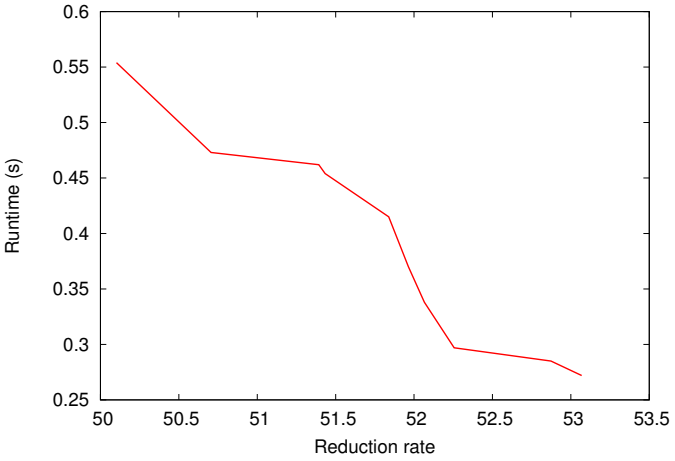


(b) On the dense dataset

Figure 5: Comparison with existing algorithms.



(a) Runtime on sparse large graphs



(b) Runtime on dense large graphs

Figure 6: Impact of reduction rate on runtime