Fast parallel algorithms for finding elementary circuits of a directed graph: a GPU-based approach

Amira Benachour, Saïd Yahiaoui, Didier El Baz, Nadia Nouali-Taboudjemat, Hamamache Kheddouci

Fast parallel algorithms for finding elementary circuits of a directed graph: A GPU-based approach

Amira Benachour[1*], Saïd Yahiaoui[2], Didier El Baz[3], Nadia Nouali-Taboudjemat[2] and Hamamache Kheddouci[4]

[1*]Department of Computer Science, USTHB, Bab Ezzouar, 16111, Algiers, Algeria.
[2]CERIST Research Center on Scientific and Technical Information, Ben Aknoun, 16028, Algiers, Algeria.
[3]LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France.
[4]Université de Lyon, CNRS, Université Lyon 1, LIRIS, UMR5205, F-69622, France.

*Corresponding author(s). E-mail(s): abenachour@usthb.dz;
Contributing authors: syahiaoui@cerist.dz; elbaz@laas.fr; nnouali@cerist.dz; hamamache.kheddouci@univ-lyon1.fr;

Abstract

Circuits in a graph are interesting structures and identifying them is of an important relevance for many applications. However, enumerating circuits is known to be a difficult problem, since their number can grow exponentially. In this paper, we propose fast parallel approaches for enumerating elementary circuits of a directed graphs based on Graphics Processing Unit (GPU). Our algorithms are based on a massive exploration of the graph in a Breadth First Search (BFS) strategy. Algorithm V-FEC explores the graph starting from different vertices simultaneously. To further reduce the search space, we present T-FEC, another algorithm that uses triplets as an initial set to start exploring. To the best of our knowledge, those are the first parallel GPU-based algorithms for finding all circuits of a given graph. The evaluation results show that the proposed approaches achieves up to 190x speed-up over Johnson's algorithm, one of the most efficient sequential algorithms for finding circuits.

Keywords: Finding circuits, Cycles, Directed graph, Parallel processing, GPU.

1 Introduction

The problems of enumerating structures such as circuits, paths, trees and cliques in graphs and networks are of fundamental importance. This kind of tasks is usually hard to deal with, due to its challenging time/space complexity. Indeed, even a small graph may contain a huge number of such structures. In the enumeration process, it is important to point that counting and finding are two different processes: finding is the construction of elements, counting is determining their number. Generally, knowing the count is rarely useful in finding the objects and even if we are able to count the elements in polynomial time, we might not be able to find all of them due to an exponential number of elements and huge memory requirements. In the following, we will use enumeration and finding interchangeably.

Our interest is for finding the elementary circuits of a directed graph. Not only it is a fundamental problem of graph theory, a circuit is a loop through which information propagates, and is widely used in real world applications: in the financial field, circuits can reveal money laundering [1]. In food webs, they represent fragile dependencies in ecosystems [2]. Finding circuits is also interesting in fraud detection [3]. For social networks, circuits represent a way to analyse and model networks [4] and are used to evaluate networks balance [5]. Finding circuits is also important in biology [6], [7], [8], and Internet systems [9].

Several algorithms for finding elementary circuits of a directed graph have been formulated but they either require large amounts of memory or are time exponential [10],[11]. Hence, they become impractical as the graph size grows. One of the most important algorithms is the algorithm by Johnson [12]. With the lowest time complexity, it is still the best state of the art algorithm. Yet, it may fail to handle some large graphs. To deal with larger graphs, Lu *et al.* [13] introduce a parallel algorithm based on Johnson's algorithm aiming at reducing the execution time. Their algorithm is based on firstly splitting the job into small ones and then applying Johnson's algorithm on each one in parallel. Giscard *et al.* [14] proposed an algorithm that counts the circuits of a graph without enumerating them. It gives the number of circuits for a length l, but it is tested for a limit of $l = 10$. In [15] the authors present an algorithm that deals with large graphs but only finds circuits of length less than or equal to k. The limit in their tests is $k = 6$.

In this paper, we present parallel approaches for finding all circuits of a directed graph. We first detect whether the graph contains circuits by searching for the strongly connected components, then we start exploring the graph to find all the circuits. Our algorithms proceed by a massive exploration of paths. We present two approaches based on the initial set used for exploration: the first uses vertices of the graph to which we assign a search area while the second uses triplets to reduce the search space of exploration. Our algorithms permit to find circuits of a given length l and circuits with a specific vertex or edge. We develop a parallel implementation of the proposed approaches in a Graphics Processing Unit (GPU) environment that is better suited for massive exploration. Our algorithms can find millions of circuits in a reduced time. We provide, to the best of our knowledge, the first GPU-based algorithms for finding elementary circuits of directed graphs.

The remaining of this paper is organized as follows. In Section 2 we present the background and related work. In Section 3 we introduce some preliminary definitions. The parallel algorithms are explained in Section 4. Details of the GPU implementation and examples are given in Section 5. Section 6 describes the experimental tests and results. Conclusion and future work are discussed in Section 7.

2 Related Work

Several algorithms have been proposed to find elementary circuits in a directed graph. Tiernan [10] was the first to propose the idea of blocking visited vertices. The algorithm is based on a backtracking procedure. It generates all elementary paths $p = (v_1, v_2, \ldots, v_k)$ with $v_1 < v_i$ for all $i \in \{2, \ldots, k\}$. Starting from some vertex v_1, it chooses an edge to traverse to some vertex v_2 such that $v_2 > v_1$ and continue this way. Whenever no vertex can be reached, the procedure backs up one vertex and chooses a different edge to traverse. If v_1 is adjacent to v_k the algorithm lists an elementary circuit $(v_1, v_2, \ldots, v_k, v_1)$. This approach examines more simple paths than necessary, making the worst-case time bound exponential in the number of elementary circuits and in the graph size.

Tarjan's algorithm [11] is based on Tiernan's depth-first method but it is faster. It lists all the cycles in $O((|V| + |E|)|C|)$ time. It uses backtracking procedure to avoid unnecessary work. Two stacks are needed, the *point stack* for storing the path currently being examined, and a boolean vector called *marked stack* to indicate whether a vertex is used in a path or not. Whenever a new circuit is found, all vertices in the current point stack will eventually be unmarked when popped from this stack. If no circuit is found involving a vertex, it will be deleted from the point stack, but continue to be marked. Some of the unnecessary work in Tiernan's algorithm is avoided by the condition that a vertex can be added to the *point stack* only if it is unmarked.

The algorithm by Weinblatt [16] begins processing the graph by eliminating vertices which cannot belong to any circuits. Next, it selects a starting vertex and chooses an edge to follow. Circuits are found when a path is cyclic or by combining parts of previously discovered circuits with a part of the path that is being processed. However, this requires more storage and the time saved by efficient search is lost in searching the constructed circuits and paths.

Of all the algorithms analyzed, the algorithm of Johnson [12] is the fastest algorithm having an upper time bound of $O((|V| + |E|)|C|)$. The success of this backtrack algorithm is due to an effective pruning technique which avoids much of the fruitless search present in earlier algorithms. For each start vertex s, a recursive backtracking procedure is invoked and its computation is similar to that of Tarjan's algorithm, except for the marking system, which was considerably enhanced. A vertex is blocked each time it enters the stack. It remains blocked until we backtrack from a vertex v that belongs to the last enumerated circuit. The unblock process recursively unblocks the vertices that have a path to v.

In 2006, an algorithm by *Liu and Wang* [17] present a way to enumerate circuits in directed and undirected graphs. It uses $k - 1$ paths to generate k circuits by exploring adjacent vertices of the tail (the last vertex in the path), this way it can enumerate circuits by length. Compared to Johnson's and Tiernan's algorithms, their algorithm is slower and not efficient.

In [18], the authors propose a parallelizable algorithm into $|V - 1|$ parts. They use a backtracking strategy using pointers allowing to backtrack more than one vertex at a time, jumping directly to the vertex from which further path extensions are possible.

The algorithm by *Lu et al* [13] is capable of handling large-scale graphs. The idea is to split the job into subroutines depending on the number of edges. Then apply Johnson's algorithm to find all elementary circuits including an edge in parallel for each subroutine. Every subroutine has a search area that consists of the graph after deleting all the edges in inputs for previous subroutines. For this, they grade the edges, by estimating the number of all elementary paths from a vertex v to a vertex u. The edge with higher *Id* will have smaller search area. This approach is implemented on Hadoop using the MapReduce technique. Compared to previous approaches, this algorithm reduces the running time for finding elementary circuits of a graph but it uses Depth First Search (DFS) sequential algorithm of Johnson. Theoretically, DFS has been proven by *Reif* [19] to be inherently sequential. It is considered to be a challenging problem for parallelization.

The algorithm we propose in this paper exploits the massive parallelism of the multi-cores GPU by running multiple searches simultaneously which makes it capable of finding millions of circuits in a short time. The exploration process is based on the Breadth First Search (BFS) algorithm that is better suited to parallel implementation. Details are given in the following.

3 Preliminaries

In this section, we present formal definitions that support our approaches to enumerate all elementary circuits of a graph. Let $G = (V, E)$ be a finite directed simple graph with vertex set V and edge set E. Let *Id(v)* denotes the number associated with the vertex v.

Definition 1 (Elementary path) A simple path p in a graph $G = (V, E)$ is a finite sequence of vertices $p = \langle v_1, v_2, \ldots, v_k \rangle$, such that $(v_i, v_{i+1}) \in E$ for $i \in \{1, 2, \ldots, k-1\}$. A path is *elementary* if no vertex appears twice.

Definition 2 (Elementary circuit) A *circuit* is a path in which the first and the last vertices are identical. An *elementary circuit* is a circuit where no vertex appears twice except the first and the last vertices.

Two elementary circuits are distinct if one is not a cyclic permutation of the other. Note that, in this paper we represent a circuit as a sequence of vertices $c = \langle v_1, v_2, \ldots, v_k \rangle$ without writing the redundant vertex twice.

Definition 3 (Strongly Connected Component) A strongly connected component (SCC) in a directed graph is defined as a subgraph in which every vertex is reachable from every other vertex. That is, for any two vertices u and v in a strongly connected component, there exists a path from u to v, and a path from v to u.

We define *In-edges*(v), the set of incoming edges of a vertex v such that:

$$In\text{-}edges(v) = \{(u, v) | (u, v) \in E\}$$

Similarly, we define *Out-edges*(v) the set of outgoing edges of v:

$$Out\text{-}edges(v) = \{(v, u) | (v, u) \in E\}$$

We name the destination vertex of the outgoing edge of v the *out-neighbor* of v. Likewise, the origin vertex of the incoming edge of v is called the *in-neighbor*.

Definition 4 (Triplets) We denote by $T(G)$ the set of triplets of $G = (V, E)$ defined as:

$$\begin{aligned} T(G) = \{\langle x, v, y \rangle | x, y, v \in V \qquad with \qquad & (x, v) \in In\text{-}edges(v), \\ & (v, y) \in Out\text{-}edges(v), \\ & Id(v) < Id(x), \\ & Id(v) < Id(y)\} \end{aligned}$$

A triplet is a sequence of vertices that initiate a path of length greater than two. We name v the *source-vertex* and x the *in-vertex* for a triplet $\langle x, v, y \rangle$. Using triplets allows to find every circuit only once and reduce the search space, as we shall see in the sequel.

In the following, we present our algorithms for finding all elementary circuits.

4 Parallel algorithms for Finding Elementary Circuits

We present here our parallel approaches for finding elementary circuits of directed graphs. Circuits are constructed starting from a *source-vertex* s. We build elementary paths by iteratively exploring adjacent vertices.

To avoid visiting vertices that do not belong to any circuit, we first find all the $SCCs$ of the graph as a pre-processing task. Definition 3 shows that an SCC consisting of more than one vertex must contain circuits and that all vertices in any circuit should be in the same SCC. Decomposing the graph into $SCCs$ reduce the number of initial source vertices and the number of neighbors to be visited. Only neighbors that are in the same SCC are considered, hence, reducing the number of paths to be explored. In this work, we use the algorithm by Tarjan [20] for finding $SCCs$ of the graph. The algorithm's running time is linear in the number of edges and vertices in G, it finds $SCCs$ in $O(|V|+|E|)$ time.

Once the graph has been partitioned into $SCCs$, we move to the exploration phase. A straightforward approach to obtain all circuits is to enumerate all combinations of distinct edges and check if the path may construct a circuit. On this basis, we propose two algorithms: Vertex-based algorithm for Finding Elementary Circuits (V-FEC) and Triplets-based algorithm for Finding Elementary Circuits (T-FEC). Our algorithms are based on a massive exploration of paths. For each SCC found, we run a parallel Breadth First Search (BFS) starting from a set of initial source vertices and build exploration trees. The detailed algorithm V-FEC, is presented in Sub-Section 4.1.

To reduce the search space and filter out some unnecessary combinations, we propose algorithm T-FEC, that uses triplets instead of vertices as initial set. As with the precedent algorithm, V-FEC, we first find $SCCs$ of the graph. Then, we generate the set of initial triplets and begin exploration. The same process of exploration is used. It is interesting to note that using triplets helps to balance exploration trees. The detailed algorithm is presented in Sub-Section 4.2.

For the following, C denotes the set of circuits found. Set P is used to contain the explored paths.

4.1 A vertex-based approach V-FEC

As explained above, the first algorithm is vertex-based. We begin by finding the $SCCs$ of the graph. Then, we initialize the set of paths P with vertices of the $SCCs$: $P = \{\langle s_1\rangle, \langle s_2\rangle, \ldots, \langle s_k\rangle\}$.

We define the Search Area (SA) of a vertex s by the subgraph induced by s and vertices with Id greater than the Id of s which are in the same SCC: $SA(s) = \{u \in V | SCC(u) = SCC(s) \text{ and } Id(u) > Id(s)\}$, $SCC(v)$ denotes the identifier of the SCC where vertex v belongs. In other words, for a vertex s we only consider vertices in the same SCC with Id greater than Id of s. Comparably, in Johnson's algorithm, we consider vertex s to be the smallest vertex in the SCC. When all circuits from s are found, we remove s, *In-edges(s)* and *Out-edges(s)*, then, we explore the rest of the SCC in a similar manner.

In the exploration phase, for every vertex s_i in P, we only explore vertices in its search area. We start by visiting the *out-neighbors* of s_i. Then, we iteratively explore the visited vertices until no more vertices can be visited. The above-mentioned process is outlined in Algorithm 1.

Algorithm 1 Vertex-based Finding Elementary Circuits V-FEC(G)

```
Require: Graph $G$
Ensure: Set $C$ of all elementary circuits of $G$
1: $C \Leftarrow \varnothing$
2: $SCC \Leftarrow$ findSCC($G$)
3: foreach $scc \in SCC$ do
4:     foreach $s \in scc$ in parallel do
5:         $C \Leftarrow$ exploreV-FEC($s$)                ▷ See Procedure 1
6: return $C$
```

4.1.1 Exploration

Now we detail the exploration phase.
Our algorithm is based on running multiple BFS graph exploration simultaneously from every initial paths $p_i \in P$ (P has been already initialised with vertices of the $SCCs$ found). We construct exploration trees for every *source-vertex* s_i of p_i by visiting the *out-neighbors* of the last vertex v of path p_i. For any neighbor u of v, one of the following situations occurs:

- Vertex u is the *source-vertex* s_i: p_i is an elementary circuit. Add p_i to C,
- u is not visited and belongs to the search area of s_i: p_i is extended. We add $\langle p_i, u\rangle$ to the set of paths P.
- u is already visited: the path $\langle p_i, u\rangle$ can not constitute an elementary circuit.

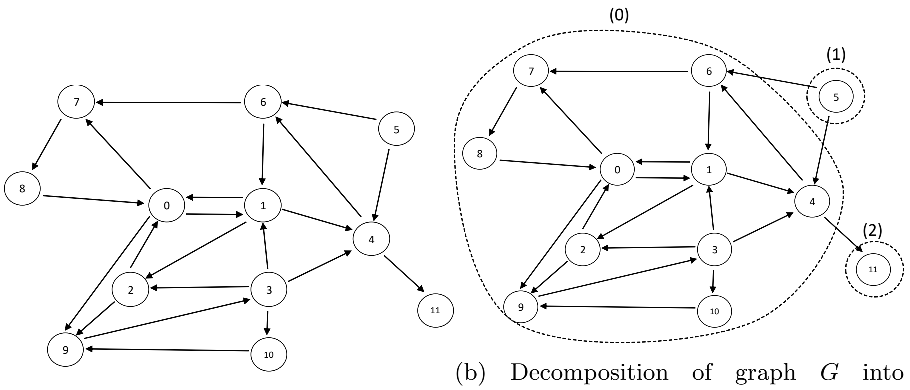
The algorithm proceeds iteratively by visiting the last visited vertex of each resulting path until no path can be found ($P = \varnothing$). At the end of the exploration phase, we would have listed all the circuits ordered by their length. A pseudo-code of the explained process is presented in Procedure 1.

Example 1 To better explain the proposed algorithm we run an example on graph $G = (V, E)$, $|V| = 12$ and $|E| = 22$ shown in Fig 1a. We begin with a search for the $SCCs$ of the graph to eliminate unnecessary vertices and edges. Graph G contains three strongly connected components, a large SCC: SCC_0 and two trivial ones: SCC_1 and SCC_2, illustrated in Fig 1b. We remove all the trivial $SCCs$ and the edges going to or from them and begin processing the remaining graph.

We initialize the set P of paths with the vertices of SCC_0 (l = 0 in Fig 2). Then we start constructing exploration trees for each *source-vertex* s. We visit *out-neighbors* of s that are in its search area. We recall that a vertex u is in the search

Procedure 1 exploreV-FEC

```
Require: Vertex $s$
Ensure: Set of circuits $C$
 1: $P \Leftarrow \{\langle s \rangle\}$
 2: $visited(s) \Leftarrow true$
 3: while $P \neq \varnothing$ do
 4:     foreach $p \in P$ do
 5:         $v \Leftarrow lastVertex(p)$                    ▷ returns the last vertex of p
 6:         foreach $u \in out\text{-}neighbors(v)$ do
 7:             if $Id(u) = Id(s)$ then
 8:                 $C \Leftarrow C \cup \{p\}$
 9:             else
10:                 if $\neg$ $visited(u)$ and $u \in SA(s)$ then
11:                     $P \Leftarrow P \cup \{\langle p, u \rangle\}$
12:                     $visited(u) \Leftarrow true$
13:         $P \Leftarrow P - \{p\}$
14: return $C$
```



(a) Directed graph G

(b) Decomposition of graph G into Strongly Connected Components (SCC)

Fig. 1: Graph decomposition into $SCCs$

area of vertex s if the two vertices are in the same SCC and $Id(u) > Id(s)$. Consider vertex s, $Id(s) = 1$. Vertex s has three *out-neighbors*: vertex 0, 2 and 4. Vertex 0 is not in $SA(s)$ since $0 < 1$. In this case, vertex 0 is not considered. If a vertex has no visited neighbors, the path is removed. e.g. paths: $\langle 8 \rangle$ and $\langle 9 \rangle$ have no visited neighbors, so they are removed. (l = 0 in Fig 2).

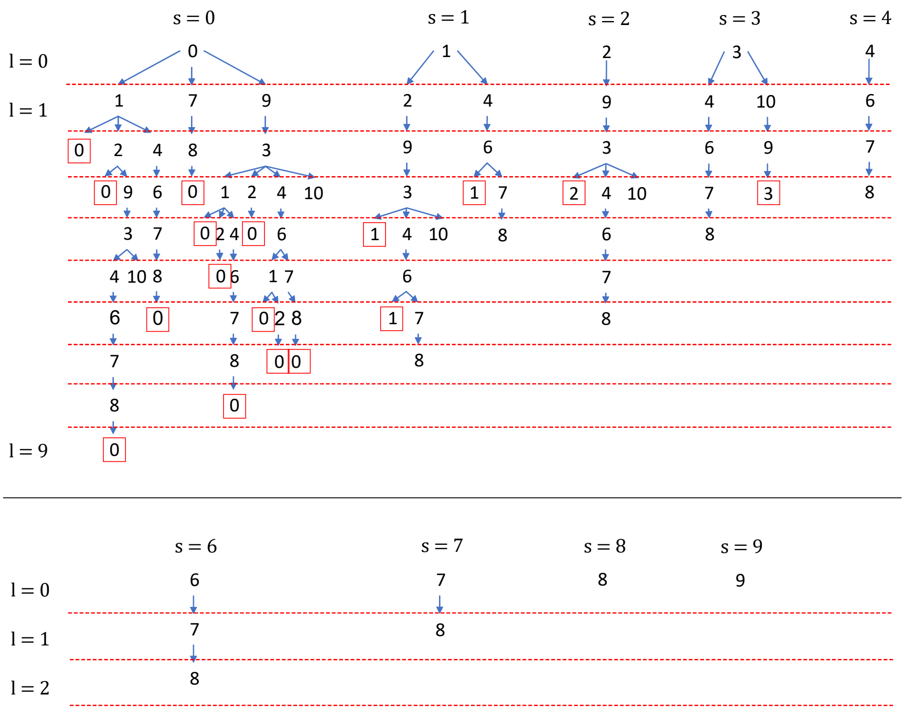


Fig. 2: Exploration trees from different source vertices: V-FEC approach

4.2 A triplets approach T-FEC

We aim to reduce the search space for the exploration phase by generating triplets using direct neighbors of each vertex. The idea is to construct what can constitute potential circuits by joining incoming and outgoing edges of a given vertex. We divide the process into two stages and define a parallel algorithm for each one of them. As a first step, (1) we generate the set of triplets $T(G)$, then, (2) we run a parallel breadth first search (BFS) for each resulting triplet in a likewise strategy as in the V-FEC approach. Algorithm 2 outlines the proposed approach.

4.2.1 Triplets generation

The idea of using triplets to begin exploration helps reducing the search space and avoids redundant circuits. Triplets also permit to have more balanced exploration trees, thus, a more balanced work charge. We build the set of triplets by making every possible combination $\langle x, v, y \rangle$ of the *in-neighbors* and *out-neighbors* of vertex v such as:

- x is the *in-neighbor* of v with $x \in SA(v)$, i.e. $Id(v) < Id(x)$ and v and x are in the same SCC.
- Similarly, y is the *out-neighbor* of v with $y \in SA(v)$.

Algorithm 2 Triplet-based Finding Elementary Circuits T-FEC(G)

```
Require: Graph $G$
Ensure: Set $C$ of all elementary circuits of $G$
1: $C \Leftarrow \varnothing$
2: $T \Leftarrow \varnothing$
3: $SCC \Leftarrow$ findSCC($G$)
4: foreach $scc \in SCC$ do
5:     foreach $v \in scc$ in parallel do
6:         $T \Leftarrow$ generateTriplets($v$)            ▷ See Procedure 2
7: foreach t $\in T$ in parallel do
8:     $C \Leftarrow$ exploreT-FEC($t$)                    ▷ See Procedure 3
9: return $C$
```

If $x = y$, $c = \langle v, y \rangle$ is a 2-length circuit. The generation process is presented in Procedure 2.

Procedure 2 generateTriplets

```
Require: Vertex v
Ensure: Set $T$ of initial triplets
1: foreach $x \in$ in-neighbors($v$) do
2:     foreach $y \in$ out-neighbors($v$) do
3:         if $x = y$ then
4:             $C \Leftarrow C \cup \{\langle x, v \rangle\}$
5:         else
6:             if $x \in SA(v)$ and $y \in SA(v)$ then
7:                 $T \Leftarrow T \cup \{\langle x, v, y \rangle\}$
8:                 $visited(x) \Leftarrow true$
9:                 $visited(v) \Leftarrow true$
10:                $visited(y) \Leftarrow true$
11: return $T$
```

4.2.2 Exploration

The exploration phase for Algorithm T-FEC is similar to the exploration in Algorithm V-FEC. Initially we visit the *out-neighbors* of y for every triplet $\langle x, v, y \rangle \in T$. Only neighbors in $SA(v)$ are considered. Then, we iteratively visit the *out-neighbors* of the last visited vertex v_k of every path $p_i \in P$. We recall that for a triplet $\langle x, v, y \rangle$, v denote the *source-vertex* and x is the *in-vertex*. For any neighbor u of vertex v_k, the exploration goes as follow:

- u is the *in-vertex* ($u = x$): p_i is an elementary circuit. Add p_i to C,
- u is not visited and $u \in SA(v)$: p_i is extended, its an elementary path. We add $\langle p_i, u \rangle$ to P.

- u is already visited: $\langle p_i, u\rangle$ is not an elementary path.

The exploration is over when no path can be expanded ($P = \varnothing$). The exploration process is described in Procedure 3.

Procedure 3 exploreT-FEC

```
Require: Triplet $t\langle x, v, y\rangle$
Ensure: Set of circuits $C$
1:  $P \Leftarrow \{t\}$
2:  while $P \neq \varnothing$ do
3:      foreach $p \in P$ do
4:          $v_k = lastVertex(p)$
5:          foreach $u \in out\text{-}neighbors(v_k)$ do
6:              if $u = x$ then
7:                  $C \Leftarrow C \cup \{p\}$
8:              else
9:                  if $\neg$ $visited(u)$ and $u \in SA(v)$ then
10:                     $P \Leftarrow P \cup \{\langle p, u\rangle\}$
11:                     $visited(u) \Leftarrow true$
12:         $P \Leftarrow P - \{p\}$
13: return $C$
```

4.3 Proof of correctness

In the following, we will prove that Algorithms V-FEC and T-FEC find every elementary circuit of a graph exactly once.

We start by showing that both algorithms find only elementary circuits of a graph.

Lemma 1 *Algorithms V-FEC and T-FEC find elementary circuits only.*

Proof In Algorithm V-FEC, every vertex in a path is marked as visited and a new vertex is added to a path if it is not already visited. If a vertex is visited we do not visit it again (line 10 of Procedure 1). Hence, a path can not contain redundant vertices. Let a path $p = \langle v_1, \ldots, v_i, \ldots, v_k\rangle$ to be expanded, and let vertices v_l and v_i be in *out-neighbors* of vertex v_k. Suppose that vertex v_l is not visited yet. If $v_l \in SA(v_1)$, it will be added to p and marked as visited. However, since vertex v_i is previously visited, it will not be explored again. This confirms that Algorithm V-FEC finds only elementary circuits.

Similarly, for Algorithm T-FEC, in Procedure 3 *exploreT-FEC*, in line 9, visited vertices are not explored again, thus, the resulting circuits are elementary. □

Theorem 1 *Algorithms V-FEC and T-FEC find all the elementary circuits of a graph exactly once.*

Proof The correctness of this lemma can be verified by the following: (1) Algorithms V-FEC and T-FEC find any elementary circuit exactly once, (2) all the elementary circuits of the graph are found.

To prove the first property, we suppose a circuit is found more than once. Let $\langle v_1, v_i, \dots, v_l \rangle$ be a circuit, such that $Id(v_i) = i$, $\forall v_i \in V$, and vertex v_1 has the smallest *Id*. Many permutations of the same circuit can be found by starting from different source vertices. A circuit can be expressed in a permuted form as many times as the number of vertices it has: $\langle v_1, v_i, \dots, v_l \rangle$, $\langle v_i, \dots, v_l, v_1 \rangle$, $\dots$, $\langle v_l, v_1, v_i, \dots \rangle$.

In Algorithm V-FEC, during the exploration phase every vertex of each SCC starts building its exploration tree by visiting its *out-neighbors*. A new vertex is added to a path if its *Id* is greater than the *Id* of the *source-vertex* (line 10 in Procedure 1: see the definition of $SA(s)$ in Section 4.1). As a consequence, the first vertex in a path is always the vertex with the smallest *Id*.

Consider a circuit $c = \langle v_1, v_i, \dots, v_l \rangle$ and two cyclic permutations of c: $c' = \langle v_i, \dots v_l, v_1 \rangle$ and $c'' = \langle v_l, v_1, v_i, \dots \rangle$. Following Procedure 1 *exploreV-FEC*, we try to construct circuit c and its permutations starting from different source vertices v_1, v_i, ..., and v_l. Each time we visit a neighbor we check if its *Id* is greater than the *Id* of the *source-vertex* of the path. Since $Id(v_1) < Id(v_i)$ and $Id(v_1) < Id(v_l)$, in both cases, $\langle v_i, \dots v_l, v_1 \rangle$ and $\langle v_l, v_1, v_i, \dots \rangle$, vertex v_1 is not explored, thus, circuits c' and c'' can not be constructed. Hence, no circuit can be found more than once.

In a similar way, when generating triplets for a vertex v in Algorithm T-FEC, we only consider neighbors that have *Ids* greater than $Id(v)$, and likewise for the exploration phase, this guarantees that circuits are not duplicated. We come to the conclusion that both algorithms find each circuit exactly once.

Now we prove the second property: Algorithms V-FEC and T-FEC find all the elementary circuits of a graph. We know from the definition of SCC that every SCC with more than one vertex must have circuits, and that all vertices in any circuit belong to the same SCC. Finding all elementary circuits of a graph is equivalent to finding all elementary circuits starting from a vertex, for every vertex of the graph.

In Algorithm V-FEC, Procedure *exploreV-FEC* finds for a vertex s all the elementary circuits starting from s such that $Id(s)$ is the smallest, by visiting every neighbor of every vertex within $SA(s)$ not visited yet in the same SCC, making every possible combination of possible paths (follows directly from Lemma 1 and the proof of property(1)). Thus, for every elementary circuit of the graph, with all vertices in $SA(s)$, its cyclic permutation starting by vertex s is returned at the end of the algorithm. Since Procedure *exploreV-FEC*, is called for every vertex of every SCC of the graph, every elementary circuit is found.

Similarly for T-FEC, in the triplets generation phase, we consider all neighbors of vertex v having *Id* greater than $Id(v)$. Procedure *exploreT-FEC* visits every neighbor of every vertex within $SA(v)$ not visited yet in the same SCC. It finds for a triplet $\langle x, v, y \rangle$ all the elementary circuits starting from v and including vertices x and y, such that $Id(v)$ is the smallest. The procedure is called for every generated triplet, thus, every elementary circuit is found. We conclude that the proposed algorithms find all the elementary circuits of a graph exactly once. This completes the proof.

$\square$

4.4 Circuits and paths given a source vertex

For some applications, we are interested in the enumeration of circuits that goes through a certain vertex v. The proposed approaches can be easily adapted to only find circuits that contains vertex v.

To achieve this, we make an adjustment in the initialization phase. In the vertex-based approach, we initialize the set of paths to v and then call Procedure 1 *exploreV-FEC* which only runs for vertex v. For the triplets approach, we generate triplets for vertex v by calling Procedure 2: *generateTriplets* and omit the condition about considering only vertices greater than v, which was mainly put to avoid enumerating circuits that are enumerated in the exploration process of other vertices. Since we just list circuits of v, circuits containing vertices of inferior *Id* are not found if we do not consider all vertices of the graph. That means, the set of triplets is generated using all of v's neighbors and there will be no redundant circuits. For both approaches, the exploration procedure remains the same, except for the condition about vertices *Ids* for the same reason explained above. In this case, the entire graph is considered.

In a similar manner, we can find circuits with a given edge (u, v) by initializing *Paths* to $\langle u, v\rangle$ in the vertex based approach. Or by generating triplets of form $\langle x_i, u, v\rangle$ with $x_i \in$ *in-neighbors* of u, for the triplet based approach. It is also possible to enumerate existing paths from a given source vertex s to a destination vertex d by searching circuits containing the edge (d, s) or Using triplets $\langle d, s, y_i\rangle$ where $y_i \in$ *out-neighbors* of s.

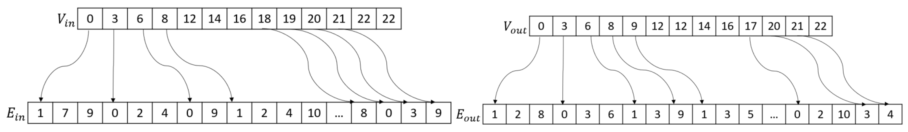
5 GPU implementation

In the following, we present a GPU implementation of the proposed approaches. We proceed in parallel, multiple BFS explorations. This model fits the multi-threaded GPU architecture. The GPU adopts a SIMD-based (Single Instruction Multiple Data) architecture, which gains high performance through massive parallelism.

The *SCC* decomposition algorithm was not parallelized due to the low impact in the processing time of the algorithm. A GPU-kernel is used to implement the exploration phase of the vertex-based version. For T-FEC two kernels are used: The first kernel generates the set of initial triplets, the second one explores the graph.

Developing efficient graph processing algorithms in GPU is a challenging task. A GPU has limitations such as a relatively small memory size. Thus, the choice of data structures and strategies employed to take advantage of the characteristics of the different types of GPU memories is important to overcome these limitations. In our implementation, we use data structures that are best suited to the GPU. We also take advantage of the GPU global, local and private memories.

In this section we discuss our choices for data structures and detail the implementation of our algorithms.


V_{in} 0 3 6 8 12 14 16 18 19 20 21 22 22
E_{in} 1 7 9 0 2 4 0 9 1 2 4 10 ... 8 0 3 9
V_{out} 0 3 6 8 9 12 12 14 16 17 20 21 22
E_{out} 1 2 8 0 3 6 1 3 9 1 3 5 ... 0 2 10 3 4

(a) CSR representation (b) CSC representation

Fig. 3: Data structure for graph representation

5.1 Data structures

5.1.1 Graph representation

Compressed adjacency list is the most common data structure for graph representation in GPU. It provides compact storage for large sparse graphs and regular memory access. It uses two arrays to store the graph $G = (V, E)$ and only requires $O(|V| + |E|)$ memory space. Adjacency lists of all vertices are packed into a single large array that we name the *edge array*. We store the starting position of the adjacency list in the *edge array* for each vertex [21]. When outgoing edges are used in the *edge array*, we name this adjacency list format, the Compressed Sparse Row (CSR). If incoming edges are used in the *edge array*, it is called Compressed Sparse Column (CSC)[22]. To generate triplets both CSR and CSC representations are used, since we need to access both incoming and outgoing edges of each vertex. Once the triplets are generated we no longer need the incoming edges, CSC arrays are deleted. An example of the CSR and CSC representations of the graph in Fig 1a is represented in Fig 3.

5.1.2 Intermediate data storage and results

When searching for circuits, a number of paths is generated in every iteration. We keep track of them by using a global integer array. Initially, it contains the initial vertices or generated triplets. Another array stores the circuits. Since the GPU do not allow dynamic memory allocation, every memory that becomes necessary must be previously allocated. We declare two static structures *Circuits* and *Paths* larger enough for the purpose they were intended to.

During exploration, previously visited vertices in a path must not be visited again. This guarantees that circuits found are elementary. A single bit indicating whether or not the vertex is in the path is sufficient. Accordingly, a bitmap (bit array) *Visited* is employed to mark the already visited vertices. This map is defined by a bi-dimensional matrix that contains a row for each path and $|V|$ columns of bits, one for each vertex of the graph. In terms of bytes, the number of columns is $\lceil\frac{|V|}{8}\rceil$. e.g. in a graph G with $|V| = 12$ a path storage occupy only 2 bytes. A Vertex v_j belongs to path i if and only if, bit j of row i is set to 1. Fig 4 shows an example of these arrays. The *Paths* matrix stores the vertices of an explored path in the order in which they are discovered. In the *Visited* matrix, each row contains a combination of bits that corresponds to the visited vertices of a path.

In addition to the small occupied space, adding a vertex or checking if it is already in the solution is done using a simple logical operation in the desired position. Because the bit-level operations are amongst the least computationally expensive, this method of storage contributes to improve the algorithm running time.

5.2 Parallel implementation

Now we detail the parallel implementations of V-FEC and T-FEC algorithms. We first describe the triplets generation process for T-FEC, then, we detail the exploration phase for both V-FEC and T-FEC.

5.2.1 Triplets generation

We begin by creating the set $T(G)$ of initial triplets $\langle x, v, y\rangle$. Such a set is created by selecting vertex $v \in V$ and analyzing all pairs of its adjacent vertices: x in the *in-neighbors* of v and y in *out-neighbors* of v. Procedure 2 *generateTriplets* describes the process of generating triplets. We launch a kernel *generateTriplets* and we assign a thread to every vertex v in the SCC. Each thread takes a vertex x from the *edge array* E_{in} and another vertex y from the *edge array* E_{out} and verify that they are in $SA(x)$. When this condition is satisfied the thread adds the combination $\langle x, v, y\rangle$ to the set of triplets. Another pair of neighbors is analyzed until all neighbors are checked. In the case where vertex x equals vertex y, $\langle x, v\rangle$ is a 2-length circuit and is added to the set C.

At the end of the triplets generation phase, the set of incoming edges is no longer needed. We free memory used by deleting V_{in} and E_{in} arrays. We only keep the set of outgoing edges: V_{out} and E_{out} arrays, for the graph representation.

5.2.2 Exploration

The exploration kernels, exploreV-FEC and exploreT-FEC, are quite similar. Kernel exploreV-FEC launches a number of threads equal to the number of initial vertices. Kernel exploreT-FEC launches a number of threads equal to the number of generated triplets. Each thread is assigned a vertex/triplet and starts exploring. During exploration, it is important to save:

- for V-FEC: the *source-vertex* and the last visited vertex.
- for T-FEC: the *source-vertex*, the *in-vertex* and the last visited vertex of a path.

These data are essential for exploration. The last visited vertex of a path permits to expand the set of paths with new paths. At each iteration, a thread visits the *out-neighbors* of the last visited vertex of the path it is assigned to. In V-FEC, the *source-vertex* determines when a circuit is found. If the new vertex is the *source-vertex*, it is a circuit. For T-FEC, it is the *in-vertex* that determines a circuit. We use an atomic operation to add the discovered circuits by copying the current path to *Circuits*. If the new vertex do not constitute a circuit, the path can be expanded when the two following conditions are met: we first check if the explored vertex is not visited using a simple bit-wise operation, then, we check if the vertex is in the search area of the *source-vertex*. Considering these data are used by a thread for as many *out-neighbors* there is to explore, it is more appropriate to copy data to shared memory because it is faster. In V-FEC, we use two vectors V_{source} and V_{last} to store the *source-vertex* and the last visited vertex of a path, respectively. Another vectors V_{in}, that stores *in-vertices*, is necessary for the triplet based approach. The number of copied paths equals the number of threads per block. Chunks of the *Visited* array corresponding to copied paths are also transferred to shared memory.

Each thread use a local array called *frontier* array to store the newly visited vertices to be copied back in global memory as new paths for the next iteration. We use a shared array *frontierOffsets*, set to 0, to compute the number of new paths. Each thread increments the value corresponding to its *Id* when it founds a new path. These values are used to copy the results of each thread in parallel to global memory. We first compute offsets for each thread on a block level by applying a parallel scan [21] on the *frontierOffsets* array. We use another array *blockOffsets* array to store the offsets of the blocks. Since there is no synchronization between blocks of threads, we use atomic operations to calculate the offsets of blocks. These two indices (*frontierOffsets* and *blockOffsets*) are used to compute the global memory address for each thread to copy its result. The schema represented in Fig 4 encapsulates the process.

Compared to the V-FEC algorithm, the exploration process in T-FEC requires less iterations considering that two vertices are already explored for each path, when generating triplets. Moreover, the triplets approach exposes more parallelism in early iterations, since there is more initial triplets than initial vertices which results in more thread occupancy.

Example 2 (Using triplets)

Consider the same graph $G = (V, E)$ in Fig 1a. In the T-FEC algorithm we start by initialising *triplets* and its corresponding *Visited* bitmap. A thread is mapped to a vertex. e.g. thread 2 generates triplets corresponding to vertex 2. We generate combinations of *in-neighbors* and *out-neighbors* of vertex 2. The *in-neighbors* of vertex 2 are vertex 1 and vertex 3. Vertex 1 is not considered since $1 < 2$. For vertex 3, $3 > 2$, so vertex 3 is considered. The *out-neighbors* of vertex 2 are vertex 0 and 9. $0 < 2$ vertex 0 is not considered as well. We are left with two valid vertices: vertex 3 as *in-neighbor* and vertex 9 as *out-neighbor*. As a result the triplet $\langle 3, 2, 9 \rangle$ is generated. Triplets from all *SCCs* are generated before starting exploration. The visited vertices of each triplet are marked as visited in the *Visited* array. Array *Visited* is an array of bytes (unsigned integers of length = 8 bits), we only use one bit to mark the visited vertices. Thus, $\lceil \frac{n}{8} \rceil$ columns are needed. In this example $n = 12$, as a consequence, *Visited* is a two columns array. During exploration, if a circuit is found, it is added to the *Circuits* array next to the circuits already found at earlier iterations. We take as an example triplet $\langle 2, 0, 1 \rangle$, vertex 2 is the *in-vertex*, vertex 0 is the

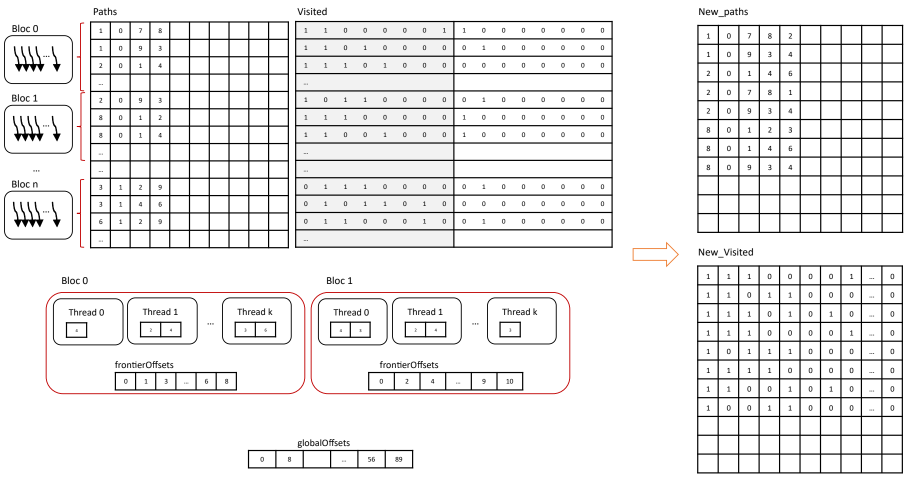


Fig. 4: A schema for paths exploration process

source-vertex and vertex 1 is the last vertex of the path. We explore *out-neighbors* of vertex 1 that are vertices 0, 2 and 4. Vertex 0 is already visited so it is not added to *Paths*. Vertex 2 is the *in-vertex*, path $\langle 2,0,1,2\rangle$ is a circuit, we add it to *Circuits*. $Circuits = Circuits \cup \{\langle 2,0,1\rangle\}$. Vertex 4 is not visited and is in $SA(0)$: the new path $\langle 2,0,1,4\rangle$ is added to *Paths*.

6 Experimental results

In this section, we present the results obtained by the CUDA implementation of our algorithms and compare them to the results of Johnson's algorithm. We use a Python implementation of Johnson's algorithm in the NetworkX package [1].

6.1 Experiment settings

Experiments were conducted on a GPU station with 32 GB RAM, with an Intel Xeon Silver 4216 2.10GHz CPU. The GPU card is an Nvidia GeForce RTX 2080 Ti with 11GB of GDDR6 VRAM and 4,352 CUDA cores, CUDA Version 11.0.

6.2 Results

We use two sets of data. The first set contains synthetic graphs generated using the R3MAT graph generator [23]. It generates graphs that resemble the properties observed in real-world graphs following a power law distribution. The description of these graphs, and the execution times of our implementation compared with the execution times of Johnson's algorithm are represented in Table 1.

[1] https://networkx.org/

Table 1: Experimental comparison of V-FEC, T-FEC and the algorithm by Johnson for synthetic graphs

				Execution time (ms)		
Graph	$\|V\|$	$\|E\|$	$\|C\|$	V-FEC	T-FEC	Johnson
graph-40-1-1-1	40	113	54121	24.66	**19.72**	1140.63
graph-40-1-1-0	40	70	1676	**9.83**	10.54	85.68
graph-50-1-1-1	50	149	120519	40.21	**28.71**	2779.66
graph-50-1-1-0	50	93	5767	**11.86**	18.31	176.45
graph-60-1-1-1	60	186	4942125	1547.52	**1161.07**	121862.10
graph-60-1-1-0	60	120	164783	**76.65**	386.73	8398.95
graph-70-1-1-1	70	225	281100	209.36	**129.30**	7143.13
graph-70-1-1-0	70	147	1820687	**764.34**	3712.17	96148.63
graph-80-1-1-1	80	264	329844	124.04	**93.69**	7043.80
graph-80-1-1-0	80	174	56067879	**18200.95**	-	3462429.93
graph-90-1-1-1	90	304	8257415	2730.90	**2105.35**	206860.77
graph-100-1-1-1	100	345	46865151	10839.7	**8891.98**	1186575.70

Table 2: Experimental comparison of V-FEC, T-FEC and the algorithm by Johnson for real world graphs

				Count circuits		Enumerate circuits		
Graph	$\|V\|$	$\|E\|$	$\|C\|$	V-FEC	T-FEC	V-FEC	T-FEC	Johnson
moreno_taro	22	78	21671	50.66	35.83	51.52	**36.24**	1618.73
moreno_sheep	28	250	19727891	27740.69	27450.56	**26723.50**	**26982.67**	703917.13
EPA	4772	8965	142	6.91	7.48	**7.14**	**7.38**	48.20
p2p-Gnutella04	10879	39994	l = 6: 1768	452.17	l = 5	**483.95**	l = 5	OOM
p2p-Gnutella08	6301	20777	l = 7: 15732	379.44	l = 6	**371.91**	l = 6	OOM
email-Eu-core	1005	24929	l = 3: 124765	154.11	213.42	**166.26**	214.28	OOM
Baywet	128	2106	l = 8: 7322229	l = 6	2673.55	l = 6	**l = 7**	OOM

Column $|C|$ represents the number of elementary circuits found by both Johnson's and our parallel algorithms. For each graph, we ran the algorithms 20 times and calculate the average time taken. The execution times are presented in milliseconds.

For the graph "graph-80-1-1-0", Algorithm T-FEC could not enumerate all circuits of the graph. This is due to the large number of intermediate results (paths) that could not fit in the GPU memory. As an alternative, we succeeded to list all the circuits of size equal or less than 16.

For some graphs, where the number of circuits is very important, it is not possible to enumerate all the circuits, due to memory limitations. The search for elementary circuits with Johnson's algorithm for these graphs result in an out of memory error (OOM). On the other hand, our algorithms proceed by levels. At each iteration, circuits of length l are found. This gives the possibility to enumerate circuits of a given length without having to enumerate all the circuits of the graph. For such graphs it is possible to proceed by enumerating the circuits until a maximum depth is reached. Table 3 shows these results.

Table 3: The results per level showing the maximum depth for each algorithm, the maximum number of circuits and the number of circuits for each level

				Count circuits		Enumerate circuits		
Graph	$\|V\|$	$\|E\|$	max circuits	V-FEC	T-FEC	V-FEC	T-FEC	Execution time / Circuits
graph-90-1-1-0	90	203	86533017	**l = 16**	l = 14	l = 13	l = 12	$T = 40191.56$ $C = \{$ 26 73 316 1203 4457 15307 49849 150785 424416 1106250 2667819 5945511 12234120 23230600 40702285 $\}$
graph-100-1-1-0	100	235	328908432	**l = 19**	l = 13	l = 14	l = 12	$T = 97980.37$ $C = \{$ 25 104 415 1564 5837 20236 64949 193905 536577 1365891 3195804 6859242 13478597 24201006 39647343 59191874 80492814 99652249 $\}$
graph-200-1-0-0	200	526	2746022	**l = 17**	l = 16	l = 15	l = 15	$T = 15829.14$ $C = \{$ 2 7 26 57 98 261 644 1482 3511 8600 20790 50882 120272 286170 672475 1580745 $\}$
graph-300-1-0-0	300	1256	501259	**l = 13**	**l = 13**	l = 12	l = 12	$T = 4937.91$ $C = \{$ 5 13 36 90 204 607 1671 4773 13969 40080 114002 325809 $\}$

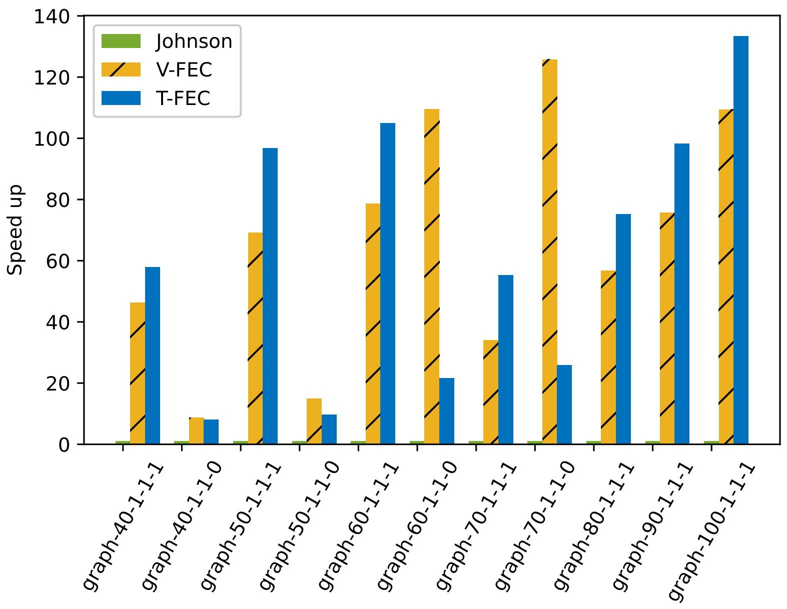


Fig. 5: Experimental comparison of Johnson's algorithm with V-FEC and T-FEC algorithms

In addition to the synthetic graphs generated by the R3MAT model [23], we use real-world datasets from four different sources: Konect [24] (moreno_taro, moreno_sheep), SNAP Library [25] (p2p-Gnutella04, p2p-Gnutella08 and email-Eu-core), networkrepository [26] (EPA) and Pajek [27] (Baywet). Characteristics of these graphs are given in Table 2.

We adapt our algorithms to count circuits without enumerating them. This significantly reduce the memory storage needed since we don't keep track of all the visited vertices. The *Paths* matrix is replaced by vectors that stores the *source-vertex* and the last vertex for each path (in addition to the *in-vertex* in the triplets approach). This permits to explore larger graphs and to explore more levels as shown in Table 3.

6.3 Analysis

The results presented in Tables 1, 2 and 3 show that our algorithms can significantly reduce the running time to find all elementary circuits compared to Johnson's algorithm. The speed-ups varies from 8.13, for graph-40-1-1-0, to 190.23 for graph-80-1-1-0, and increases with the number of circuits. Fig 5 shows the speed-ups achieved by V-FEC and T-FEC compared to the algorithm by Johnson.

Table 4: Experimental results for the impact of the SCC graph decomposition in V-FEC

			Time (ms) V-FEC + SCC			Time (ms) V-FEC - SCC	
Graph	$\|V\|$	$\|SCC\|$	SCC	exploration	total	exploration	total
p2p-Gnutella04	10879	4317	3.409	419.01	**448.18**	1609.35	1834.1
p2p-Gnutella08	6301	2068	1.819	351.33	**369.19**	804.23	859.62
email-Eu-core	1005	803	1.082	130.93	**163.44**	133.67	**150.84**
Baywet	128	103	0.100	494.69	**508.25**	1381.08	1487.08
EPA	4772	{2,11,3,10,4,2,5,2,2,6,3,2, 2,2,2,3,6,2,4,2,2,2,2,2,2,2}	0.990	5.08	**6.99**	48.54	52.35
moreno_taro	22	22	0.015	51.91	**53.31**	51.18	**52.48**
moreno_sheep	28	{2, 22}	0.024	26778.5	26927.6	-	-
graph-40-1-1-1	40	12	0.019	23.61	**24.72**	23.98	**24.90**
graph-40-1-1-0	40	17	0.019	9.14	**9.83**	10.26	11.06
graph-50-1-1-1	50	14	0.022	40.13	**41.79**	41.26	**42.71**
graph-50-1-1-0	50	23	0.020	10.51	**11.29**	11.37	**12.09**
graph-60-1-1-1	60	18	0.021	1474.35	**1502.46**	1522.29	**1551.51**
graph-60-1-1-0	60	30	0.022	74.70	**77.08**	92.35	94.63
graph-70-1-1-1	70	16	0.05	204.70	**207.88**	208.76	**211.74**
graph-70-1-1-0	70	30	0.025	722.09	**739.82**	1116.81	1135.71
graph-80-1-1-1	80	13	0.027	125.01	**127.63**	122.39	**124.85**
graph-80-1-1-0	80	36	0.058	16656.3	**17507.3**	29613	30959.6
G_02_03_02_1	64	51	0.025	3350.25	**3365.48**	4698.41	4723.84
G_02_03_02_3	64	52	0.024	3216.52	**3232.97**	4753.63	4779.21
G_06_01_015_1	64	32	0.025	1430.28	**1463.19**	2307.59	2348.33
G_06_01_015_2	64	33	0.026	984.37	**1003.3**	1507.08	1527.84
G_01_02_03_5	64	38	0.027	1593.95	**1601.35**	6859.72	6883.79

Table 5: Experimental results for the impact of the SCC graph decomposition in T-FEC

	Time (ms) T-FEC + SCC					Time (ms) T-FEC - SCC			
Graph	Triplets	SCC	T-time	exploration	total	Triplets	T-time	exploration	total
p2p-Gnutella04	29576	3.511	0.807	220.87	**230.42**	69411	0.95	466.27	476.12
p2p-Gnutella08	17656	1.963	0.932	336.11	346.54	41752	-	-	-
email-Eu-core	585882	1.133	37.954	156.36	**212.24**	609644	39.535	160.07	**215.75**
Baywet	1928	0.108	0.346	424.99	**439.27**	2547	0.397	1128.53	1142.53
EPA	126	1.014	0.027	4.27	**7.39**	6467	2.243	21.22	24.24
moreno_taro	54	0.019	0.028	35.43	**36.58**	54	0.029	36.34	**37.21**
moreno_sheep	323	0.026	0.114	26629.4	26758.8	497	-	-	-
graph-40-1-1-1	131	0.020	0.051	18.64	**19.87**	131	0.049	18.59	**19.62**
graph-40-1-1-0	85	0.019	0.045	9.69	**10.59**	128	0.053	13.77	14.51
graph-50-1-1-1	125	0.023	0.063	26.90	**28.52**	125	0.061	27.68	**29.04**
graph-50-1-1-0	152	0.022	0.079	16.83	**17.79**	223	0.089	25.39	26.26
graph-60-1-1-1	179	0.026	0.088	1137.43	**1163.57**	179	0.085	1155.23	**1180.97**
graph-60-1-1-0	220	0.025	0.070	374.19	**377.08**	272	0.073	465.84	468.48
graph-70-1-1-1	156	0.028	0.052	138.01	**141.21**	156	0.051	132.68	**134.97**
graph-70-1-1-0	297	0.027	0.091	3665.96	**3684.68**	388	0.132	6509.8	6528.48
graph-80-1-1-1	183	0.032	0.055	91.48	**93.93**	183	0.053	90.86	**93.07**
graph-80-1-1-0	427	0.032	0.195	16791.4	16939.6	598	-	-	-
G_02_03_02_1	107	0.028	0.034	3200.82	**3212.58**	127	0.029	4679.11	4691.74
G_02_03_02_3	110	0.027	0.025	2551.72	**2563.72**	122	0.027	4413.85	4427.28
G_06_01_015_1	280	0.032	0.079	4770.51	**4797.8**	351	0.086	7998.3	8029.54
G_06_01_015_2	252	0.032	0.075	3290.73	**3307.99**	305	0.102	5596.88	5624.02
G_01_02_03_6	135	0.027	0.031	24933.2	**25784.7**	139	0.038	33428.3	34310.3
G_01_02_03_5	102	0.033	0.031	723.22	**726.70**	135	0.038	3059.05	3063.8

Table 6: Results of the impact of the graph degree distribution on execution time

				Count circuits time (ms)		
Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert C\rvert$	V-FEC	Triplets	T-FEC
G_02_02_045_2	64	147	816347	**1827.96**	65	2102.32
G_01_02_03_4	64	179	29404	**342.46**	88	**331.87**
G_01_02_03_5	64	179	255064	1708.35	102	**748.35**
G_01_02_03_6	64	179	22280272	67482.4	135	**26221.7**
G_02_02_045_3	64	179	3558059	13296.96	102	**8240.42**
G_02_02_045_4	64	230	6903928 (l = 16)	31656.84	175	**14577.16**
G_02_03_02_1	64	147	770263	**3503.28**	107	**3298.44**
G_02_03_02_3	64	147	783545	3415.25	110	**2675.1**
G_035_015_035_2	64	147	938758	**4067.80**	91	**4123.50**
G_025_025_025_2	64	147	2873222	**8648.52**	99	**8105**
G_02_03_02_4	64	179	32980789 (l = 25)	**78766.44**	157	**79135.08**
G_02_03_02_5	64	179	11348731	**28826.70**	127	**25192.58**
G_02_03_02_6	64	179	37618516 (l = 28)	**68731.56**	159	87724.42
G_025_025_025_3	64	179	17920050 (l = 25)	81767.37	118	**43598.87**
G_035_015_035_3	64	179	22603921 (l = 26)	88475.50	134	**72638.87**
G_025_025_025_4	64	230	6669043 (l = 15)	**15742.1**	277	**15594.58**
G_035_015_035_4	64	230	4563399 (l = 16)	**12570.66**	215	14345.82
G_05_02_015_2	64	147	988967	**503.95**	241	2354.7
G_06_01_015_1	64	147	3180582	**1536.35**	280	5055.02
G_06_01_015_2	64	147	1766087	**1060.39**	252	3434.8
G_06_01_015_3	64	147	419801	806.74	172	**748.73**
G_05_02_015_3	64	179	42767911 (l = 21)	**17457.02**	289	84379.7
G_06_01_015_4	64	179	48576117 (l = 19)	**24837.16**	451	72551.22
G_06_01_015_5	64	179	35867587	**22945.82**	345	56180.6
G_06_01_015_6	64	179	16792958	**10489.28**	324	20738
G_05_02_015_4	64	230	13016638 (l = 13)	**4715.12**	506	17352.32

We analyse the effect of the *SCC* decomposition phase on the overall processing time by comparing the execution time of both approaches with and without decomposing the graph into *SCCs*. Detailed results for V-FEC and T-FEC algorithms are listed in Table 4 and Table 5. V-FEC + SCC and T-FEC + SCC present execution times of V-FEC and T-FEC using *SCC* decomposition, while V-FEC - SCC and T-FEC - SCC present the execution times of the algorithms without the *SCC* decomposition phase. The results show that the decomposition of the graph into *SCCs* can reduce processing time compared to when the graph is not decomposed into *SCCs*. In fact, finding the *SCCs* of the graph can reduce the number of vertices to be explored and avoid fruitless paths, thus, minimizing the time of the exploration phase, which justify the use of the pre-processing phase in our algorithms. Experiments demonstrate that time achieved by the decomposition phase is negligible compared to the overall running time.

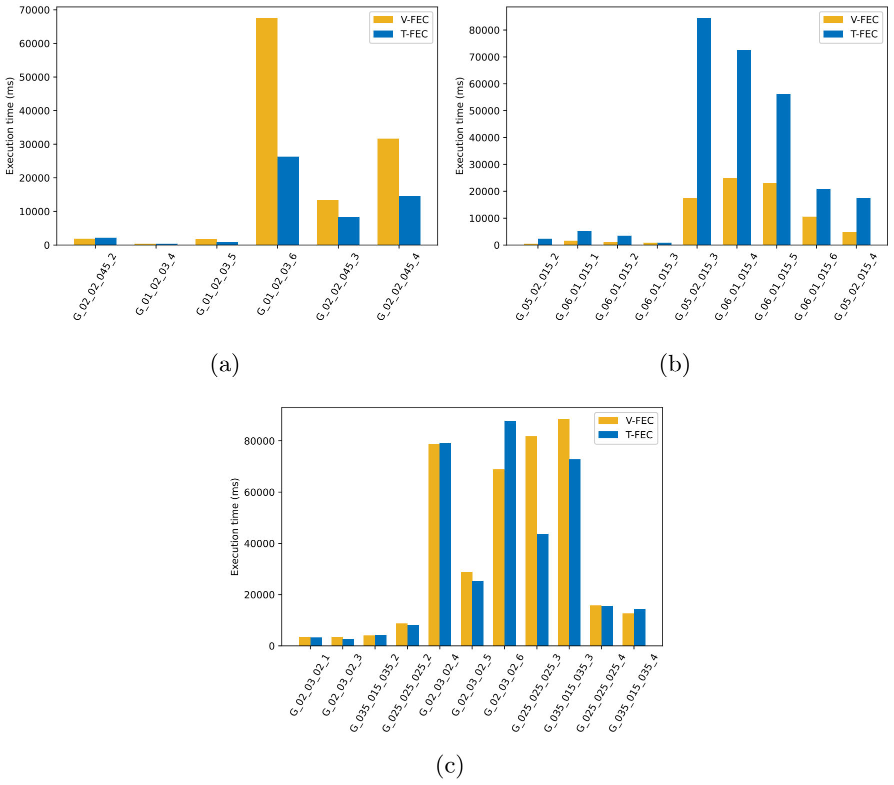


Fig. 6: Results of the impact of the graph degree distribution

We note that while V-FEC algorithm performs better in some graphs, T-FEC gives better results in others. In an attempt to identify what variables affect the results, we run a series of experiments on synthetic graphs, using the R-Mat graph generator model [28]. The generator takes as an input $|V|$, $|E|$ and the parameters a b c and d which represents the probabilities of an edge falling into partitions. We fix $|V|= 64$ and vary $|E|$ and the values of a, b, c and d and observe the behavior of both V-FEC and T-FEC approaches. Results are presented in Table 6. We observe that the degree distribution of the graph is a relevant parameter in the algorithms' performance. We divide Table 6 into three sets based on the degree distribution. We represent the execution times of both approaches on these graphs in Fig 6. The second set in the table represent graphs where the degrees of vertices are almost uniform. For these graphs, we remark that V-FEC and T-FEC algorithms are equivalent (Fig 6c). The first and the third set are graphs with few vertices having higher degree than other vertices with a difference in the ordering of these vertices. In the first set the high degree vertices are processed in the end, while in the last set they are processed in the beginning. We observe that V-FEC algorithm

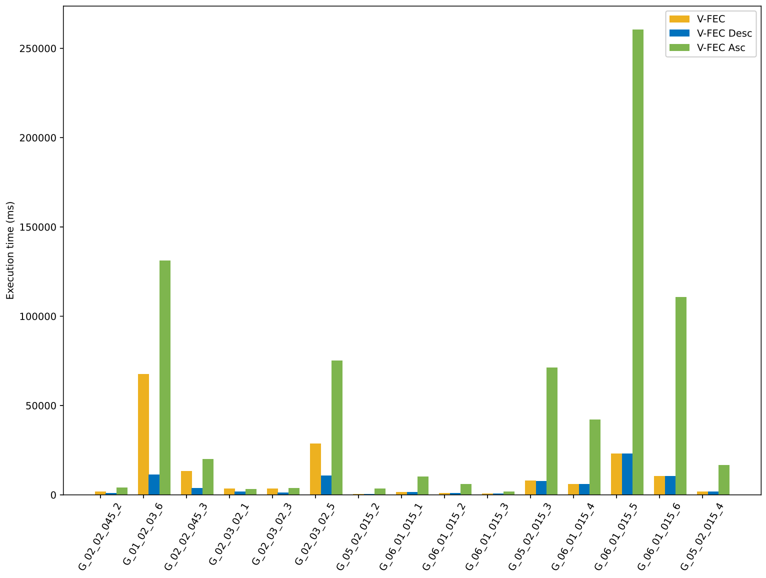


Fig. 7: Results of the impact of the degree distribution order

outperforms the T-FEC algorithm for the graphs in the last set of the table (Fig 6b). In this case, the number of triplets that are generated is important for those vertices with high degree. Indeed, a lot of combinations are generated, which means more paths to be explored and some of them are fruitless. On the contrary, the triplets approach outperforms the vertex-based one for the first set of graphs (Fig 6a).

To further analyze the impact of the distribution order of the vertices' degree, we use a graph labeling function based on the in and out degrees of the vertices of the graph. We know that the more *in-neighbors* and *out-neighbors* a vertex has the more probable it is to belong to circuits. Based on that, we order the vertices according to the product of their in and out degrees. We use two graph labeling functions: the first one l_{Asc}, organizes the vertices in an increasing order of *in-degree * out-degree*. The second function l_{Desc}, lists the vertices in a decreasing order of the vertices' *in-degree * out-degree*. We apply this labeling to some of the graphs in Table 6 and run Algorithm V-FEC. We compare the results with and without the labeling functions. Results are represented in Fig 7. We observe that Algorithm V-FEC gives better results when the l_{desc} label is applied. Indeed, by visiting vertices with more neighbors in early iterations we have higher probability to find circuits earlier and without exploring much of the fruitless paths. Experimentally observing, we can tell that the degree distribution order is a crucial parameter in the algorithm execution.

7 Conclusion

In this paper, we proposed parallel algorithms for enumerating all elementary circuits of a directed graph. Algorithms V-FEC and T-FEC first detect whether a circuit exists, by searching for Strongly Connected Components, then, explore possible paths in parallel to find elementary circuits. In addition, they enumerate circuits of a given length and circuits going through a given vertex. We have provided theoretical guarantees on the correctness of V-FEC and T-FEC. The presented algorithms were implemented and tested in a GPU environment. To the best of our knowledge, these are the first parallel GPU-based algorithms for finding all the elementary circuits of a graph. Conducted experiments showed a significant improvement in execution times of V-FEC and T-FEC compared with the algorithm by Johnson, due to the massive parallelism offered by the GPU. The speed-up over the sequential algorithm was from ≈ 8 to 190 times.

It is to mention that the size of the GPU memory is a restricting factor which do not allow to handle large graphs with millions of circuits. To further our research, we plan to work on a multi-GPU approach to scale to larger graphs.

Acknowledgments. This work was partly supported by the Franco-Algerian program PHC Tassili BiGreen n°18 MDU 111. The experiments were conducted using the GPU station YUVA II provided by the Research Center on Scientific and Technical Information CERIST (Algeria). Part of this study has been made possible via funding of Centre International de Mathématiques et d'Informatique de Toulouse, CIMI-ANR-11-LABX-0040 - LABX-2011.

Data availability. All data generated or analysed during this study have been deposited in the Open Science Framework repository (OSF) (https://osf.io/74dve/?view_only=a5291e6e898843c48686b06d49c03ed5).

References

[1] Fronzetti Colladon, A., Remondi, E.: Using social network analysis to prevent money laundering. Expert Systems with Applications **67**, 49–58 (2017). https://doi.org/10.1016/j.eswa.2016.09.029

[2] Dunne, J.A., Williams, R.J., Martinez, N.D.: Network structure and biodiversity loss in food webs: robustness increases with connectance. Ecology letters **5**(4), 558–567 (2002). https://doi.org/10.1046/j.1461-0248.2002.00354.x

[3] Bodaghi, A., Teimourpour, B.: Automobile insurance fraud detection using social network analysis. In: Applications of Data Management and Analysis : Case Studies in Social Networks and Beyond, pp. 11–16

(2018). https://doi.org/10.1007/978-3-319-95810-1_2. Springer International Publishing

[4] Safar, M., Mahdi, K., Kassem, A.: Universal cycles distribution function of social networks. In: 2009 First International Conference on Networked Digital Technologies, pp. 354–359 (2009). https://doi.org/10.1109/NDT.2009.5272805

[5] Giscard, P.-L., Rochet, P., Wilson, R.C.: Evaluating balance on social networks from their simple cycles. Journal of Complex Networks **5**(5), 750–775 (2017). https://doi.org/10.1093/comnet/cnx005

[6] Kwon, Y.-K., Cho, K.-H.: Analysis of feedback loops and robustness in network evolution based on boolean models. BMC bioinformatics **8**(1), 1–9 (2007). https://doi.org/10.1186/1471-2105-8-430

[7] Klamt, S., von Kamp, A.: Computing paths and cycles in biological interaction graphs. BMC bioinformatics **10**(1), 1–11 (2009). https://doi.org/10.1186/1471-2105-10-181

[8] Chitturi, B., Bein, D., Grishin, N.V.: Complete enumeration of compact structural motifs in proteins. In: Proceedings of the International Symposium on Biocomputing, pp. 1–8. Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1722024.1722047

[9] Parasar, M., Farrokhbakht, H., Enright Jerger, N., Gratz, P.V., Krishna, T., San Miguel, J.: Drain: Deadlock removal for arbitrary irregular networks. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 447–460 (2020). https://doi.org/10.1109/HPCA47549.2020.00044

[10] Tiernan, J.C.: An efficient search algorithm to find the elementary circuits of a graph. Commun. ACM **13**(12), 722–726 (1970). https://doi.org/10.1145/362814.362819

[11] Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM Journal on Computing **2**(3), 211–216 (1973). https://doi.org/10.1137/0202017

[12] Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing **4**(1), 77–84 (1975). https://doi.org/10.1137/0204007

[13] Lu, W., Zhao, Q., Zhou, C.: A parallel algorithm for finding all elementary circuits of a directed graph. In: 2018 37th Chinese Control Conference (CCC), pp. 3156–3161 (2018). https://doi.org/10.23919/ChiCC.2018.8482857. IEEE

[14] Giscard, P.-L., Kriege, N., Wilson, R.C.: A general purpose algorithm for counting simple cycles and simple paths of any length. Algorithmica **81**(7), 2716–2737 (2019). https://doi.org/10.1007/s00453-019-00552-1

[15] Gupta, A., Suzumura, T.: Finding all bounded-length simple cycles in a directed graph. CoRR (2021). https://doi.org/10.48550/ARXIV.2105.10094

[16] Weinblatt, H.: A new search algorithm for finding the simple cycles of a finite directed graph. J. ACM **19**(1), 43–56 (1972). https://doi.org/10.1145/321679.321684

[17] Liu, H., Wang, J.: A new way to enumerate cycles in graph. In: Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06), pp. 57–57 (2006). https://doi.org/10.1109/AICT-ICIW.2006.22. IEEE

[18] Sankar, K., Sarad, A.: A time and memory efficient way to enumerate cycles in a graph. In: 2007 International Conference on Intelligent and Advanced Systems, pp. 498–500 (2007). https://doi.org/10.1109/ICIAS.2007.4658438. IEEE

[19] Reif, J.H.: Depth-first search is inherently sequential. Information Processing Letters **20**(5), 229–234 (1985). https://doi.org/10.1016/0020-0190(85)90024-9

[20] Tarjan, R.: Depth-first search and linear graph algorithms. SIAM journal on computing **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010

[21] Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with cuda. GPU gems **3**(39), 851–876 (2007)

[22] Gui, C.-Y., Zheng, L., He, B., Liu, C., Chen, X.-Y., Liao, X.-F., Jin, H.: A survey on graph processing accelerators: Challenges and opportunities. Journal of Computer Science and Technology **34**(2), 339–371 (2019). https://doi.org/10.1007/s11390-019-1914-z

[23] Angles, R., Paredes, R., García, R.: R3MAT: A rapid and robust graph generator. IEEE Access **8**, 130048–130065 (2020). https://doi.org/10.1109/ACCESS.2020.3009577

[24] Kunegis, J.: KONECT – The Koblenz Network Collection. http://konect.cc/networks/ (2013)

[25] Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data (2014)

[26] Rossi, R.A., Ahmed, N.K.: The Network Data Repository with Interactive Graph Analytics and Visualization. http://networkrepository.com (2015)

[27] Batagelj, V., Mrvar, A.: Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/ (2006)

[28] Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining (SDM), pp. 442–446 (2004). https://doi.org/10.1137/1.9781611972740.43. SIAM