


Article

A Multi-Threading Algorithm to Detect and Remove Cycles in Vertex- and Arc-Weighted Digraph

Huanqing Cui ^{1,2,3,*} , Jian Niu ^{1,2}, Chuanai Zhou ⁴ and Minglei Shu ²

¹ College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266510, China; JeneoNew@163.com

² Shandong Provincial Key Laboratory of Computer Networks, Shandong Computer Science Center (National Supercomputer Center in Jinan), Jinan 250101, China; shuml@sdas.org

³ Shandong Province Key Laboratory of Wisdom Mine Information Technology, Shandong University of Science and Technology, Qingdao 266510, China

⁴ Business School, Qingdao Binhai College, Qingdao 266555, China; iamxai@126.com

* Correspondence: smart0193@163.com; Tel.: +86-532-8605-7852

Received: 28 August 2017; Accepted: 9 October 2017; Published: 10 October 2017

Abstract: A graph is a very important structure to describe many applications in the real world. In many applications, such as dependency graphs and debt graphs, it is an important problem to find and remove cycles to make these graphs be cycle-free. The common algorithm often leads to an out-of-memory exception in commodity personal computer, and it cannot leverage the advantage of multicore computers. This paper introduces a new problem, cycle detection and removal with vertex priority. It proposes a multithreading iterative algorithm to solve this problem for large-scale graphs on personal computers. The algorithm includes three main steps: simplification to decrease the scale of graph, calculation of strongly connected components, and cycle detection and removal according to a pre-defined priority in parallel. This algorithm avoids the out-of-memory exception by simplification and iteration, and it leverages the advantage of multicore computers by multithreading parallelism. Five different versions of the proposed algorithm are compared by experiments, and the results show that the parallel iterative algorithm outperforms the others, and simplification can effectively improve the algorithm's performance.

Keywords: digraph; cycle detection and removal; vertex- and arc-weighted; multi-threading; iteration

1. Introduction

Graphs can describe many applications in the real world, such as social networks, communication networks, dependency among software packages, and debt networks, and detecting cycles in a graph is a fundamental algorithmic problem. However, the existing solutions cannot handle large-scale digraphs using single commodity personal computers (PC), because they often lead to out-of-memory exceptions. This paper presents a multi-threading parallel and iterative algorithm of detecting and removing cycles, and this algorithm can take full advantage of multicore PCs. The main contributions of this paper include:

- It defines a new problem of detecting and removing cycles in a vertex- and arc-weighted digraph according to vertices' priority.
- It presents a multi-threading parallel and iterative algorithm to solve the problem. The algorithm avoids the out-of-memory exception by simplification and iteration, and it leverages the advantage of multicore computers by multithreading parallelism.
- It performs thorough experiments to show the performance of the proposed algorithm.

The organization of the rest of paper is as follows. Section 2 presents the related work. Section 3 introduces the problem. Section 4 presents the detail of proposed algorithm and time complexity analysis. Section 5 illustrates the performance comparison among five different versions of the proposed algorithms, using randomly generated graphs and real world digraphs. The final section contains conclusions and discussions.

2. Related Work

Many literatures have focused on the cycle detection of a directed/undirected graph. Hamiltonian cycle is a special cycle that covers each vertex exactly once. The decision problem of whether a graph contains Hamiltonian cycle is NP-complete, so [1] presents some conjectures. A non-recursive algorithm to detect Hamiltonian cycle is presented in [2], and this algorithm is applied to flow-shop scheduling problem.

Another two important cycles are the shortest and longest cycles of a graph, because they decide the girth and diameter/circumference of a graph, respectively. An approximate algorithm is given in [3] to find the shortest cycle in an undirected unweighted graph, whose expected time complexity is between $O(n \log n)$ and $o(n^2)$, where n is the number of vertices of a graph. The algorithm presented in [4] aims to find the shortest cycle for each vertex in a graph. In a non-Hamiltonian graph, the longest cycles through some special vertices are discussed in [5–7], where [5] focuses on the longest cycles through large degree vertices, while [6,7] pay attention to the longest cycles passing all vertices of degree at least a given threshold.

Besides characterizing or detecting special cycles of a graph, enumerating and counting cycles is also an important problem. Some explicit formulae for the number of 7-cycles in a simple graph are obtained in [8]. Recently, distributed algorithms to count or enumerate cycles are presented in [9], in order to solve this problem more efficiently.

Sometimes a graph is dynamic, it grows by arc insertions. For this kind of graphs, the cycle detection problem is referred to as incremental cycle detection. Two online algorithms are presented in [10] to solve this problem, which handle m arc additions for n vertices in $O(m^{\frac{2}{3}})$ and $O(n^{\frac{5}{2}})$ time, respectively. The algorithms presented in [11] are designed for sparse and dense graphs, which take $O(\min\{m^{\frac{1}{2}}, n^{\frac{2}{3}}\})$ and $O(n^2 \log n)$ time, respectively.

Most of the above algorithms are based on DFS (Depth-First Search), which is inherently sequential [12], and they often cause out-of-memory exception when they are run on commodity PCs to solve large-scale graphs.

Recently, many parallel and distributed graph-processing algorithms have been proposed. A message-passing algorithm is presented in [13] to count short cycles in a graph. The distributed cycle detection algorithm proposed in [14] is based on the bulk synchronous message passing abstraction, which is suitable for implementation in distributed graph processing systems. The cloud computing systems breed parallel graph processing platforms, such as Pregel [15], GraphX [16], and GPS [17]. These platforms are based on the BSP (Bulk Synchronous Parallel) model, which depends on costly distributed computing systems. Therefore, PCs cannot play to these algorithms' strengths.

The purpose of this paper is to propose an algorithm of detecting and removing cycles of large-scale digraphs using a single commodity PC.

3. Problem Statements

A digraph or directed graph G consists of a finite set of vertices $V = \{v_1, v_2, \dots\}$, and a set of directed arcs $E = \{\langle v_i, v_j \rangle \mid v_i \in V, v_j \in V, i \neq j\}$ that each connects an ordered pair of vertices. Each arc $\langle v_i, v_j \rangle \in E$ is associated with a numerical weight w_{ij} . Each vertex v_i is also associated with an integer priority p_i , and for any $i \neq j$, $p_i \neq p_j$. We denote the set of arc weights and vertex priorities by W and P , respectively. Given $\langle v_i, v_j \rangle \in E$, v_i is called the tail, and v_j is called the head of the arc. For a vertex v_i , the number of head ends adjacent to v_i is called its in-degree, and the number of tail ends adjacent to v_i is its out-degree.

In a digraph, a directed path is a sequence of vertices in which there is an arc pointing from each vertex in the sequence to its successor in the sequence. A *simple directed cycle* is a directed path where the first and last vertices are the same, and there are no repeated arcs or vertices (except the requisite repetition of the first and last vertices). Let $C = \{v_1, v_2, \dots, v_k, v_1\}$ be a simple directed cycle, we define its *cycle weight* w_C as

$$w_C = \min\{w_{ij} | \langle v_i, v_j \rangle \in C\}. \quad (1)$$

The problem of detecting cycles in such a vertex- and arc-weighted digraph is defined as follows. Given a digraph $G = (V, E, W, P)$, the vertex

$$v_{beg} = \operatorname{argmin}_{v_i \in V} \{p_i | p_i \in P\}, \quad (2)$$

is chosen as the beginning vertex to detect cycle. “argmin” gets the vertex with the minimal weight. Suppose v_{cur} is the current visiting vertex, the next vertex to visit should be

$$v_{next} = \operatorname{argmin}_{v_{cur}, v_i \in E} \{p_i | p_i \in P\}. \quad (3)$$

Once v_{beg} is visited again, a cycle C_{beg} is detected whose first and last vertices are both v_{beg} . Next, this cycle's weight $w_{C_{beg}}$ is calculated by Equation (1). Finally, C_{beg} is removed by subtracting $w_{C_{beg}}$ from all arcs of this cycle, so that the arcs of weight $w_{C_{beg}}$ are deleted from G . Repeat this procedure until there are no cycles in G . The characteristics of this problem are:

- The cycles are detected according to the vertex priority. The first and last vertices of the cycle should be the vertexes of minimal priority, and the next vertex to visit from current one should be the one with minimal priority in all of the direct successors.
- The cycles are removed according to the arc weight. The arcs with cycle weight are deleted to destroy the cycle.

This problem stems from removing the dependency chain in weighted dependency graph [18] and debt chain in financial field. Taking debt chain as an example, the debt relationship among enterprises can be formulated as a weighted digraph, as defined above. In a graph G , (1) each vertex $v \in V$ stands for an enterprise; (2) each arc $\langle v_i, v_j \rangle \in E$ stands for a debt relationship between corresponding enterprises and means v_i owes v_j ; (3) w_{ij} associated with $\langle v_i, v_j \rangle$ is the amount of debt of corresponding enterprises; and, (4) p_i of vertex v_i represents the importance of the corresponding enterprises, and the smaller p_i has the higher priority. This kind of digraph is called a debt graph for convenience. Figure 1 illustrates an example including 12 enterprises. Debt cycle is a simple directed cycle in a given debt graph. It is very useful to solve the debt problem, because the enterprises belonging to a debt cycle can solve their debts without transferring capital.

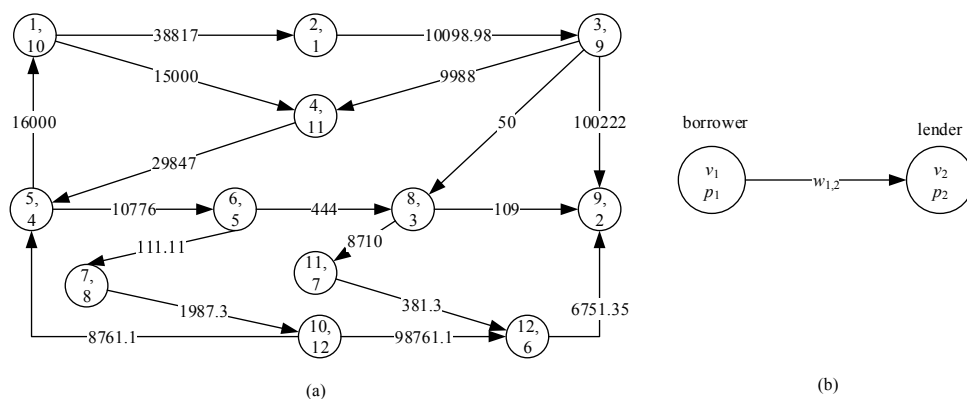


Figure 1. An example of debt graph. (a) A debt graph; (b) legend.

4. Cycle Detection and Removal with Vertex Priority

Algorithm 1 shows the sketch of the proposed algorithm, and its details are given next.

Algorithm 1. Detect and Remove Cycles of a Weighted Digraph

Step1: Simplify G .

Step2: Divide G into strongly connected components.

Step3: Detect, output and remove cycles of each strongly connected component.

4.1. Graph Simplification

In Algorithm 1, line 1 simplifies the given graph G by deleting the vertices and arcs that do not belong to any cycle. This step aims to decrease the scale of the graph, especially when G is sparse. In fact, many graphs derived from real world are sparse [19]. Table 1 shows seven digraphs taken from SNAP (Stanford Network Analysis Platform) [20]. We can see that more than 50% of vertices having zero in- or out-degree. The debt graph is sparser than these graphs, as shown in Section 5.3.

Table 1. The sparse characteristic of Stanford network analysis platform (SNAP) graphs.

Graph Name	p2p-Gnutella04	p2p-Gnutella25	p2p-Gnutella30	p2p-Gnutella31	Email-EuAll	Web-NotreDame	Wiki-Talk
$ V $	10,876	22,687	36,682	62,586	265,214	325,729	2,394,385
$ E $	39,994	54,705	88,328	147,892	420,045	1,497,134	5,021,410
ratio ¹	54.80%	74.06%	74.12%	74.29%	95.79%	57.65%	94.88%

¹ “ratio” is the ratio of vertices of out- or in-degree 0 to $|V|$.

Proposition 1. Given a digraph $G = \{V, E, W, P\}$, a vertex $v \in V$ does not belong to any cycle if its out-degree or in-degree is 0.

According the above proposition, Algorithm 2 shows the simplification process. It first gets the in- and out-degrees of vertices (lines 2–5). The *while* loop (lines 7–18) repeats deleting vertices of $d_i^{in} = 0$ or $d_i^{out} = 0$, and their corresponding arcs. Lines 8–14 find the first vertex v_k satisfying Proposition 1. If such a vertex exists (line 15), it calls Algorithm 3 to delete this vertex. Algorithm 3 uses two *for* loops to delete the arcs associated with u (lines 2–5 and 6–9) and u itself (line 10).

Algorithm 2. Simplify a Weighted Digraph

```

1.  Function SIMPLIFY( $G$ )
2.    for all  $v_i \in V$  do
3.       $d_i^{in} \leftarrow$  in degree of  $v_i$ 
4.       $d_i^{out} \leftarrow$  out degree of  $v_i$ 
5.    end for
6.     $k \leftarrow 1$ 
7.    while  $k > 0$  do
8.       $k \leftarrow 0$ 
9.      for  $i \leftarrow 1$  to  $|V|$  do
10.     if  $v_i \in V \wedge (d_i^{in} = 0 \vee d_i^{out} = 0)$  then
11.        $k \leftarrow i$ 
12.       break
13.     end if
14.   end for
15.   if  $k > 0$  then
16.     DELETE( $G, v_k$ )
17.   end if
18. end while
19. return  $G$ 
20. end function

```

Algorithm 3. Delete a Vertex from a Weighted Digraph

```

1. Function DELETE( $G, u$ )
2.   for all  $v_i, u \in E$  do
3.      $E \leftarrow E - \{v_i, u\}$ 
4.      $d_i^{out} \leftarrow d_i^{out} - 1$ 
5.   end for
6.   for all  $\langle u, v_j \rangle \in E$  do
7.      $E \leftarrow E - \{\langle u, v_j \rangle\}$ 
8.      $d_j^{in} \leftarrow d_j^{in} - 1$ 
9.   end for
10.   $V \leftarrow V - \{u\}$ 
11. end function

```

4.2. Cycle Detection

Proposition 2. Given a digraph $G = \{V, E, W, P\}$, if $\{v_1, v_2, \dots, v_k, v_1\}$ is a directed cycle, then these vertices belong to the same strongly connected components of G .

This is the basis of step 2 of Algorithm 1. There are several efficient algorithms to calculate strongly connected components (SCCs) of a digraph, so we do not discuss the detail of step 2 of Algorithm 1. After SCCs are obtained, the cycles can be detected in each SCC. The cycle calculation is the core of Algorithm 1.

Because the problem needs visit the vertices according to their priorities, cycle should be detected using DFS. For large-scale graphs, the recursive DFS will cause out-of-memory exception, so we need an iterative version. The following data structures are adopted.

Stack S is used to store the visited sequence of vertices. Its interfaces include:

- PUSH(S, v) to push v into S .
- POP(S) to pop an element of S .
- PEEK(S) to return the top element of S without popping out it.
- SEARCH(S, v) to return the index of v in S , and it returns -1 if v is not in S .
- CLEAR(S) to clear the S to empty.
- SUBLIST(S, m, n) to return the sub-sequence of S from index m to n ($m \leq n$).
- ISEMPY(S) to return **false** if S is not empty, and return **true** if S is empty.

Key-value pair table T is applied to record each neighbor vertex of a visited vertex is visited or not. As shown in Figure 2, each key $(v_i, i = 1, 2, \dots, n)$ is a vertex that is unique in the whole table. Each v_i has a list of values, and each value is either 0 or 1 representing the corresponding neighboring vertex of v_i is unvisited or not. It has the following interfaces:

- CLEAR(T, v) to reset all values of key v to 0.
- SUM(T, v) to return the sum of values of key v . The sum represents the number of neighboring vertices have been visited until now.
- GETFIRST(T, v) to return the first unvisited neighboring vertex of v .
- SET(T, v, u) to set the value of neighboring vertex u of v to 1.

Minimum heap H is applied to store vertices based on their priorities, due to the high-efficiency of sorting and deleting members in heap. It only has one function, GETFIRST(H), to get the vertex of the highest priority.

Algorithm 4 gives the cycle detection algorithm. It is a multi-threading parallel algorithm. Line 2 forks threads, and all of the threads run in parallel (line 4). Each thread gets an SCC (line 5), and calls FUN to detect and remove cycles (line 6).

The set of vertices V^{SCC} is stored in a minimum heap H . The outer *while* loop (lines 12~40) detects cycles starting from each vertex of V^{SCC} . Lines 13~17 initialize variables. For a given vertex, u , the inner *while* loop (lines 18~36) detects all of the cycles including u . Let v be the top element of S . If all its neighbor vertices are visited (line 20), pop it out and clear the status of all its neighbor vertices. Otherwise, get the first unvisited neighbor vertex r of v and visit it. In further, if r is not in S (line 27), r is pushed into S . If r is in S and $r = u$ (line 29), then the sub-list of S from r to the top element is a cycle. If u cannot find any cycle, delete it from graph (line 38).

Algorithm 4. Detect Cycles of a Weighted Directed Graph

```

1.  Function DETECTCYCLE( $G$ )
2.    Fork  $K$  threads
3.     $SC \leftarrow \Phi$ 
4.    for each thread do in parallel
5.       $G^{SCC} \leftarrow$  the next SCC to detect and remove cycle
6.       $SC \leftarrow SC \cup FUN(G^{SCC})$ 
7.    end for
8.    return  $SC$ 
9.  end function
10. Function FUN( $G^{SCC}$ ) //  $G^{SCC} = \{V^{SCC}, E^{SCC}, W^{SCC}, P^{SCC}\}$  is a SCC of  $G$ ,  $V^{SCC}$  is in heap  $H$ 
11.   $SSC \leftarrow \Phi$ 
12.  while  $V^{SCC} \neq \Phi$  do
13.     $u \leftarrow GETFIRST(H)$ 
14.    CLEAR( $S$ )
15.    CLEAR( $T, u$ )
16.    PUSH( $S, u$ )
17.     $flag \leftarrow \text{false}$ 
18.    while ( $flag = \text{false}$  and  $ISEMPTY(S) = \text{false}$ )
19.       $v \leftarrow PEEK(S)$ 
20.      if  $|SUM(T, v)| = d_v^{out}$  then
21.        POP( $S$ )
22.        CLEAR( $T, v$ )
23.      else
24.         $r \leftarrow GETFIRST(T, v)$ 
25.        SET( $T, v, r$ )
26.         $k = SEARCH(S, r)$ 
27.        if  $k = -1$  then
28.          PUSH( $S, r$ )
29.        else if  $r = u$  then
30.           $C \leftarrow SUBLIST(S, k, |S|)$  //  $C$  is a cycle
31.           $SSC \leftarrow SSC \cup \{C\}$ 
32.          REMOVECYCLE( $G^{SCC}, C$ )
33.           $flag \leftarrow \text{true}$ 
34.        end if
35.      end if
36.    end while
37.    if  $flag = \text{false}$  then
38.      DELETE( $G^{SCC}, u$ )
39.    end if
40.  end while
41.  return  $SSC$ 
42. end function

```

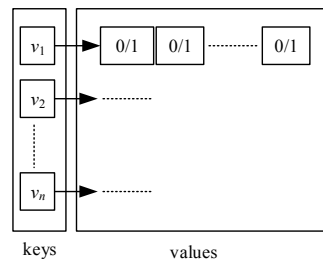


Figure 2. An example of key-value pair table.

When a cycle is detected, it should be removed at once using REMOVECYCLE (line 32) shown in Algorithm 5. The first *for* loop (lines 3–7) finds the cycle weight. The second *for* loop (lines 8–20) decreases all of the arc weights of C by w_C . If the weight of an arc becomes 0, delete this arc (line 11). In further, if the in-degree or out-degree of a vertex becomes 0, delete this vertex (lines 13 and 16).

Algorithm 5. Remove a Cycle

```

1. Function REMOVECYCLE( $G^{SCC}, C$ ) //  $G^{SCC} = \{V^{SCC}, E^{SCC}, W^{SCC}, P^{SCC}\}, C = \{v_1, v_2, \dots, v_k\}$ 
2.    $w_C \leftarrow w_{k,1}$ 
3.   for  $i \leftarrow 1$  to  $k - 1$  do
4.     if  $w_{i,i+1} < w_C$  then
5.        $w_C \leftarrow w_{i,i+1}$ 
6.     end if
7.   end for
8.   for  $i \leftarrow 1$  to  $k$  do
9.      $j \leftarrow 1 + (i \bmod k)$ 
10.     $w_{i,j} \leftarrow w_{i,j} - w_C$ 
11.    if  $w_{i,j} = 0$  then
12.       $E^{SCC} \leftarrow E^{SCC} - \{v_i, v_j\}$ 
13.      if  $d_i^{out} = 0$  then
14.        DELETE( $G^{SCC}, v_i$ )
15.      end if
16.      if  $d_j^{in} = 0$  then
17.        DELETE( $G^{SCC}, v_j$ )
18.      end if
19.    end if
20.  end for
21. end function

```

Proposition 3. Given a strongly connected component $G^{SCC} = \{V^{SCC}, E^{SCC}, W^{SCC}, P^{SCC}\}$, Algorithm 4 can terminate with $V^{SCC} = \Phi$.

Proof. Let u be the vertex with the highest priority in SCC G^{SCC} , i.e., p_u is the minimal. From u , DFS is used to visit this SCC. Let v be the current visiting vertex.

Case 1. v has no outgoing arc or all its neighbor vertices are visited in previous steps. In this case, $|\text{Sum}(T, v)| = d_v^{out}$, and v should be popped out from S .

Case 1-1. If $v = u$, S becomes empty, so the inner *while* (line 18) exits and *flag*=**false**. This result means no cycle can be detected from u , so u is deleted (line 38).

Case 1-2. If $v \neq u$, the inner *while* (line 18) goes on checking the top element of S .

Case 2. v has at least one unvisited outgoing arc. In this case, we select one unvisited neighbor vertex r to continue searching.

Case 2-1. If $r = u$, a cycle C is detected (line 29). This cycle is removed by Algorithm 5. Because w_C equals to at least one arc's weight, Algorithm 5 deletes at least one arc. Finally, $\text{flag}=\text{true}$, and the inner *while* loop exits.

Case 2-2. If r is not visited, r is pushed into S and *while* loop continues.

Case 2-3. If r is in stack but $r \neq u$, it continues finding the next vertex to visit.

Note that Case 1-2, Case 2-2 and Case 2-3 will fall into Case 1-1 or Case 2-1 finally, so each iteration of outer *while* loop deletes an arc or vertex at least. Therefore, $V^{\text{SCC}} = \Phi$ when Algorithm 4 finishes.

Remark 1. Algorithm 4 cannot detect all cycles of a given SCC, because a cycle may be broken when another one is removed.

4.3. Time Complexity Analysis

Let \bar{d}_i, \bar{d}_o be, respectively, the average in-degree and out-degree of vertices.

In Algorithm 2, the *for* consumes $O(|V| + |E|)$ to compute the in- and out-degree of all vertices. When no vertex can be deleted, the *while* only needs $O(|V|)$ to scan all vertices. When only one vertex can be deleted in each iteration, it consumes $O(|V|(\bar{d}_i + \bar{d}_o))$.

Algorithm 3 is very simple, whose time complexity is $O(\bar{d}_i + \bar{d}_o)$.

Let l be the length of cycle as the input of Algorithm 5. The first *for* consumes $O(l)$. If only one arc can be deleted, the second *for* consumes $O(l)$. In the worst case, $l - 1$ arcs of this cycle are all deleted, then the second *for* consumes $O(l(\bar{d}_i + \bar{d}_o))$.

For function $\text{FUN}(G^{\text{SCC}})$ of Algorithm 4, in the worst case, each iteration of the inner *while* detects a cycle, and each cycle only deletes one arc using Algorithm 5, so this function calls $(|E^{\text{SCC}}| - 1)$ times Algorithm 5. Obviously, the longest cycle is of length $|E^{\text{SCC}}|$, and the shortest one is of length 2, so the average length of cycles is

$$\frac{1}{|E^{\text{SCC}}| - 1} \sum_{i=2}^{|E^{\text{SCC}}|} i = \frac{|E^{\text{SCC}}| + 2}{2}. \quad (4)$$

If G has N SCCs, each thread deals with $\frac{N}{K}$ SCCs on average. Therefore, the worst time complexity is $O(\frac{N}{K} \times \frac{|E^{\text{SCC}}| + 2}{2} \times (\bar{d}_i + \bar{d}_o))$.

The time complexity of Algorithm 1 is the sum of Algorithms 2–4. In the worst case, the digraph has only one SCC, so its time complexity is $O(|E|(\bar{d}_i + \bar{d}_o))$.

4.4. An Example

Taking Figure 1 as an example, vertex 9 is deleted in first for its out-degree is 0, and its adjacent arcs are deleted too. Vertices 12, 11, and 8 are deleted in succession.

After simplification, the remaining vertices are all in one SCC. Vertex 2 becomes the first one to find cycle due its highest priority. Starting from vertex 2, vertices 3, 4, 5 are visited successively. Between two adjacent vertices of 5, 6 is chosen as the next vertex due to its higher priority. After visiting 7, 10 and 5, a cycle (5, 6, 7, 10) is detected. Since it does not contain 2, so the algorithm backtracks and visits 1, and then visits 2 to detect a cycle {1, 2, 3, 4, 5}. This cycle's weight $w_C = 9988$. After calling Algorithm 3, the arc (3, 4) is deleted. Repeat the above process, cycles (5, 6, 7, 10) and (1, 4, 5) are detected and removed, and then the graph is cycle-free. In one word, Algorithm 4 can find 3 cycles in Figure 1.

5. Experiments and Analysis

The experiments are performed on a PC with Inter Core i7-5500U @ 2.40GHz CPU (dual-core), 6GB memory, Windows 7 operating system. The programs are coded with Java in Eclipse, and we set the upper-bound stack size to 4GB. The following versions of algorithm are compared:

- Recursion without simplification (RWOS). It detects cycle recursively without simplification. It is the popular algorithm used by many literatures.
- Recursion with simplification (RWS). It detects cycle recursively after simplification.
- Iteration without simplification (IWOS). It detects cycle iteratively without simplification.
- Iteration with simplification (IWS). It detects cycle iteratively after simplification.
- Multi-thread parallel IWS (MPIWS). It detects cycle in parallel with IWS using multi-threads.

In order to compare their performance, we define $R = \frac{T_A}{T_B}$ as the ratio of execution time between algorithm A (T_A) and B (T_B). Obviously, $R < 1$ means that algorithm A is better than B . Considering the algorithms to be compared, we define

$$R_1 = \frac{T_{IWOS}}{T_{RWOS}}; R_2 = \frac{T_{IWS}}{T_{RWS}}; R_3 = \frac{T_{RWS}}{T_{RWOS}}; R_4 = \frac{T_{IWS}}{T_{IWOS}}; R_5 = \frac{T_{MPIWS}}{T_{IWS}}. \quad (5)$$

5.1. Experiments with Randomly Generated Digraphs

First, we generate 100 digraphs randomly to analyze the performance of the above algorithms. Each kind of digraph has different number of vertices and arcs. These digraphs have $1000 \times k$ ($k = 1, 2, \dots, 10$) vertices, and the average out degree of vertices is 1 to 10 in step of 1. For each kind of digraph, we generate 10 different digraphs, and each algorithm runs 10 times for each digraph. The results are average over these 10 runs.

As shown in Figure 3, the iterative algorithm is significantly better than recursive algorithm because $R_1 < 1$ and $R_2 < 1$, and R_1 and R_2 decrease while the number of vertices increases. R_1 and R_2 are almost the same when $|V|$ is the same. When $|V| = 1000$, $R_1 = 0.657$, and $R_2 = 0.656$. When $|V| = 10,000$, $R_1 = R_2 = 0.29$. Figure 3 also shows that simplification cannot reduce execution time effectively. For a different number of vertices, $R_3 \in [0.975, 0.997]$, and $R_4 \in [0.986, 0.996]$. MPIWS outperforms IWS in further due to $R_5 \in [0.378, 0.558]$.

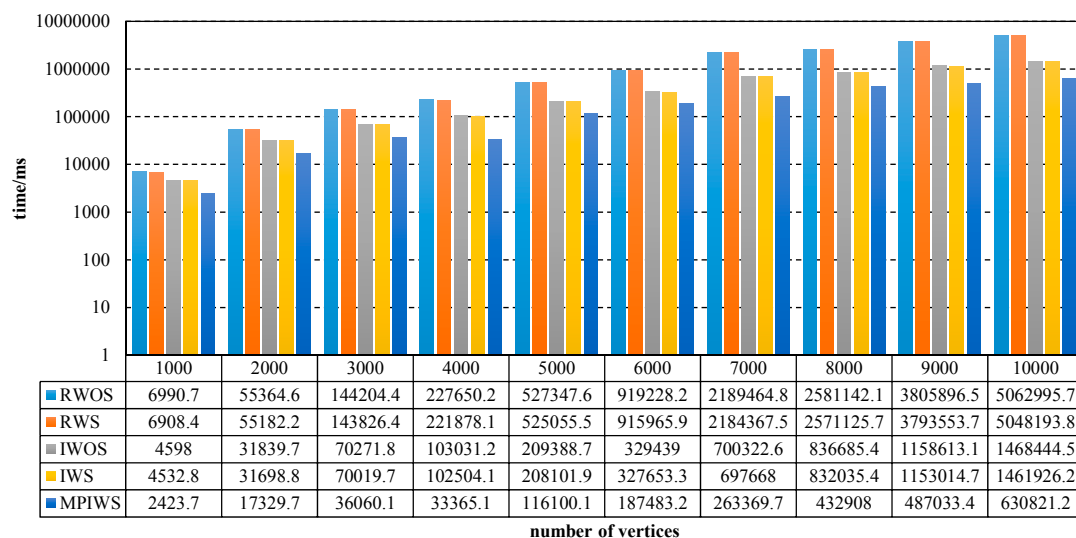


Figure 3. Impacts of number of vertices on execution time of each algorithm.

Figure 4 shows that the iterative algorithm outperforms the recursive version when the average degree of vertices is the same, and R_1 and R_2 decrease while the average out-degree of vertices increases. When out-degree is 1, $R_1 = 0.881$, $R_2 = 0.981$. When out-degree is 10, $R_1 = 0.307$, $R_2 = 0.285$. MPIWS also outperforms IWS in this figure, and $R_5 \in [0.323, 0.505]$.

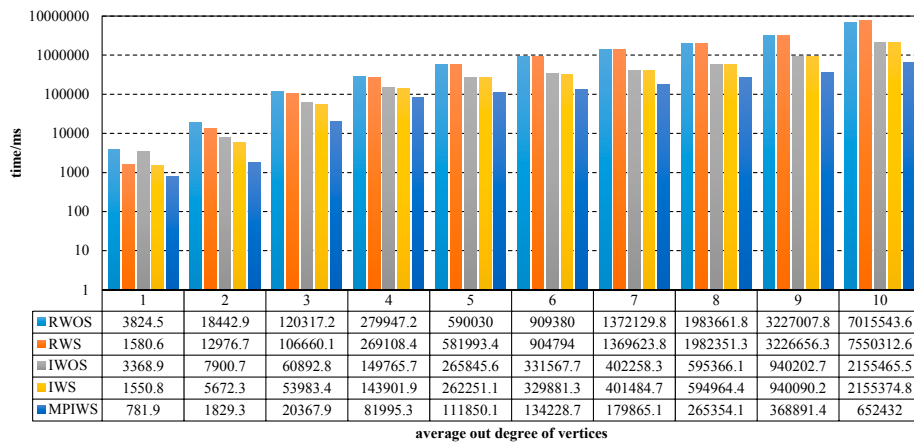


Figure 4. Impacts of average degree of vertices on execution time of each algorithm.

Figure 4 shows that simplification has much effect on the performance of these algorithms. Figure 5 shows the numeric results of R_3 and R_4 , R_3 and R_4 increase rapidly when out-degree increases from 1 to 3, but they change a little when out-degree is larger than 3. Specially, R_3 and R_4 equal to or are greater than 1 when the average degree is 10, which means that the simplification has no effect on the algorithm's efficiency.

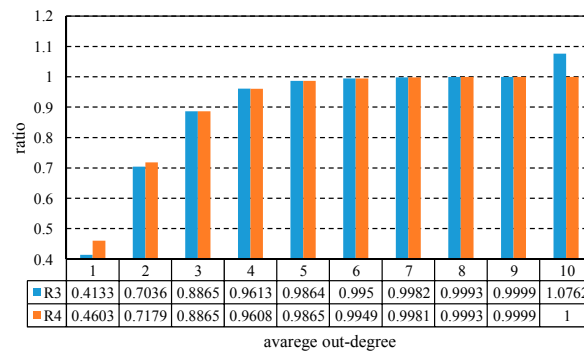


Figure 5. R_3 and R_4 under different out-degree.

In fact, when the average degree is relatively small, a large portion of vertices and arcs will be deleted by Algorithm 2, as shown in Figure 6. When the average degree is 1, the residual vertices and arcs are only about 0.7% of original graph. When the degree is 2, this ratio increases to 63% rapidly. Only less than 10% of vertices and arcs can be deleted when the average degree is larger than 3.

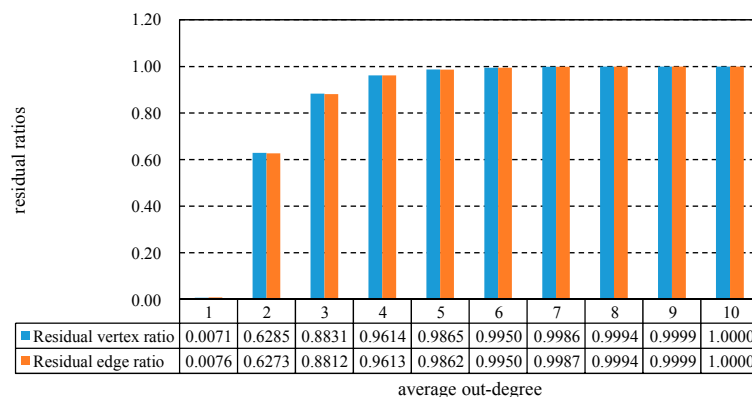


Figure 6. Ratio of number of residual vertices and arcs after simplification.

5.2. Experiments with SNAP Datasets

RWS, IWS, and MPIWS are used to detect and remove cycles of seven digraphs from SNAP datasets, and Table 2 shows the results. RWS cannot complete correctly for the last five graphs. The stack of RWS reaches 4GB before RWS can solve the problem, so it leads to out-of-memory exception in Java. IWS and MPIWS can solve the problem correctly, but MPIWS outperforms IWS because $R_5 < 1$.

Table 2. Performance Analysis of Algorithms (time unit: ms. ‘-’ means it has no value).

Graph Name	p2p-Gnutella04	p2p-Gnutella25	p2p-Gnutella30	p2p-Gnutella31	email-EuAll	Web-NotreDame	Wiki-Talk
T_{RWS}	355.5179	615.3466	-	-	-	-	-
T_{IWS}	315.4235	592.1235	1804.2942	2181.8719	4788.6087	6075.5061	7560.0935
T_{MPIWS}	147.0574	342.7287	891.0342	1238.7748	2798.9455	3680.5876	4907.7557
R_5	0.4662	0.5788	0.4938	0.5678	0.5845	0.6058	0.6492

5.3. Experiments with Real Dept Graphs

In order to analyze the performance in real application, we utilize them for a real debt graph. The debt data comes from Qingdao YouRong Development Co., Ltd., in Shandong Province, China. This graph has 7692 vertices and 8737 arcs, so the average out-degree of each vertex is about 1.14. After simplification, the graph only has 184 vertices and 345 arcs, namely only 2.39% vertices and 3.95% arcs are saved after simplification. The simplified graph is shown in Figure 7. This figure is drawn by Pajek.

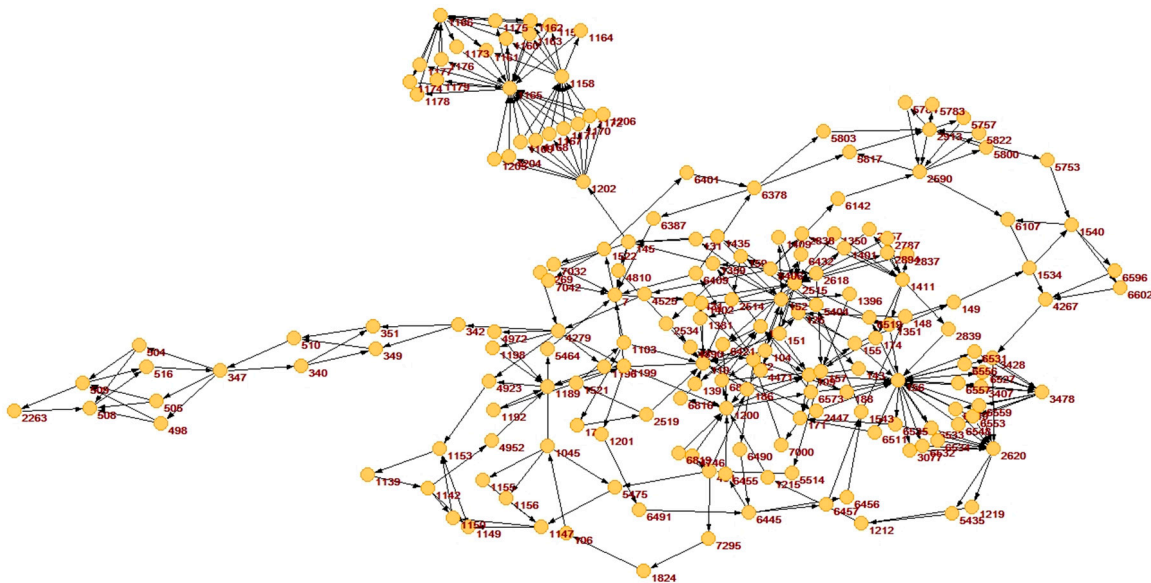


Figure 7. A real debt graph after simplification.

Figure 8 presents the execution times of RWOS, RWS, IWOS, IWS, and MPIWS to solve the real debt graph. We can see that $R_1 = 0.467$, $R_2 = 0.456$, $R_3 = 0.663$, $R_4 = 0.647$, $R_5 = 0.452$, which shows simplification, iteration, and parallelism improves the performance when compared with traditional recursive algorithms without simplification.

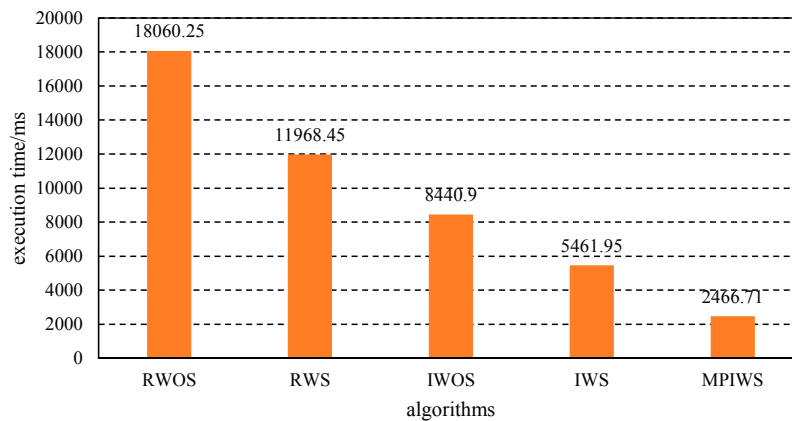


Figure 8. Execution time of four algorithms.

5.4. Performance Comparisons on Different Computers

We compare the speedups of IWS and MPIWS on the follow CPUs: Intel Core i7-7700 (quad-cores), Intel Core i7-5820K (six-cores), and the results are shown in Figure 9. The main memory is still 6GB. Since RWS still cannot solve all graphs, we only compare IWS and MPIWS. In Figure 9, IWS-2, IWS-4, and IWS-6 are the execution time of IWS on i7-5500U, i7-7700, and i7-5820K, respectively. MPIWS-2, MPIWS-4, and MPIWS-6 are the execution time of MPIWS on i7-5500U, i7-7700, and i7-5820K, respectively. The “debt-graph” is the graph used in Section 5.3.

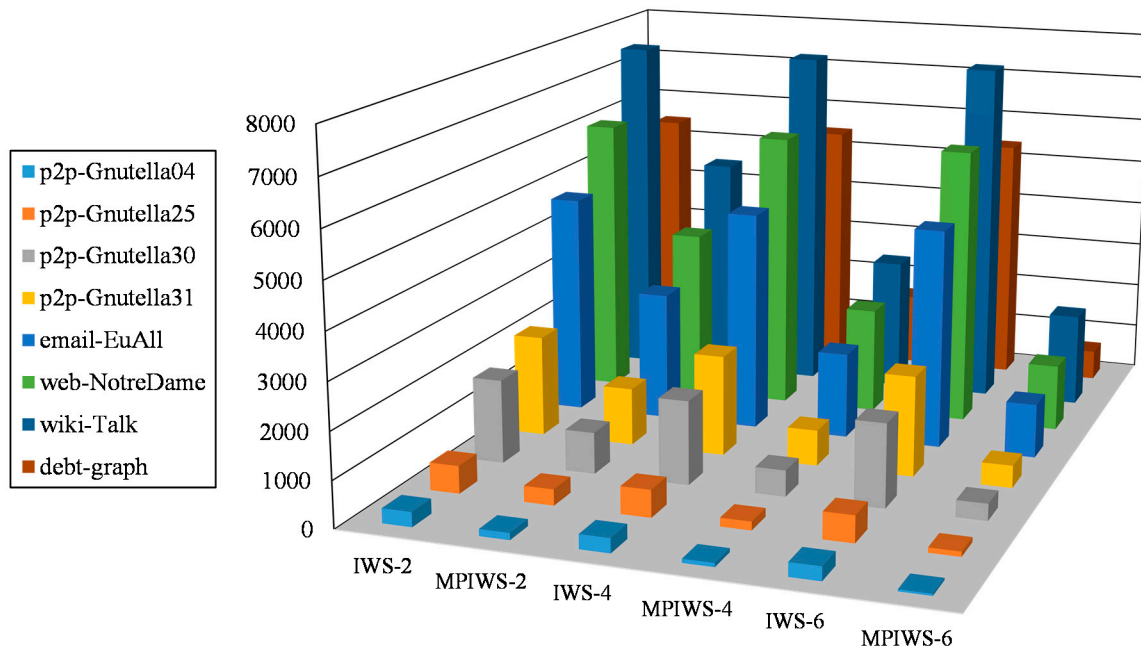


Figure 9. Execution time of different algorithms on different CPU.

Figure 9 shows that the increasement of cores has little influence on IWS. However, MPIWS changes a lot using different CPU. T_{MPIWS} using i7-7700 is about 59.1~66.3% of T_{MPIWS} using i7-5500U, and T_{MPIWS} using i7-5820K is about 41.1~67.4% of T_{MPIWS} using i7-7700. This figure also shows that R_5 decreases with an increase of cores. Using i7-5500U, i7-7700, and i7-5820K, the values of R_5 are, respectively, $R_5 \in [0.4516, 0.6492]$, $R_5 \in [0.2705, 0.3885]$, and $R_5 \in [0.1208, 0.2685]$.

6. Conclusion and Discussion

Many real-world scenarios and applications can be modeled as a graph, and many problems need to detect cycles. Derived from cycle detection and removal problem in debt graph, this paper defines a new problem that detects cycle according to the vertex priority, and removes cycles according to arc weight. A multi-threading parallel and iterative algorithm is proposed to solve this problem. Iteration is applied to replace recursion and to avoid out-of-memory exception, and multi-thread is applied to improve the efficiency. The experiments exhibit that the iterative algorithm can be run on a single commodity PC to solve the large-scale graph, and parallel algorithm outperforms sequential algorithm obviously. In further, the proposed algorithm uses simplification to reduce the scale of graph. The experiments show that the execution efficiency is improved significantly by simplification when there are many vertices having 0 in- or out-degree.

Generally, iteration outperforms recursion while solving a same problem, but simplification is only applicable to sparse graph. If the graph is not sparse, line 2 of Algorithm 1 can be deleted. Moreover, commodity PCs cannot solve the problem, even use of MPIWS in case of the graph is too large, so parallel algorithms depending on computer cluster must be applied, such as the algorithms proposed in [14,21,22].

Acknowledgments: This work is supported by the National Key Research and Development Program of China Grant No. 2017YFC0804406 and 2017YFB0202001, and the Open Project of Shandong Provincial Key Laboratory of Computer Networks, Grant No. SDKLCN-2015-03.

Author Contributions: H. Cui conceived and designed the experiments; J. Niu performed the experiments; C. Zhou and M. Shu analyzed the data; H. Cui wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kühn, D.; Osthus, D. A survey on hamilton cycles in directed graphs. *Eur. J. Combin.* **2012**, *33*, 750–766. [CrossRef]
2. Silva, J.L.C.; Rocha, L.; Silva, B.C.H. A new algorithm for finding all tours and hamiltonian circuits in graphs. *IEEE Lat. Am. Trans.* **2016**, *14*, 831–836. [CrossRef]
3. Lv, X.; Zhu, D. An approximation algorithm for the shortest cycle in an undirected unweighted graph. In Proceedings of the International Conference on Computer, Mechatronics, Control and Electronic Engineering, Changchun, China, 24–26 August 2010; IEEE: New York, NY, USA, 2010; pp. 297–300.
4. Yuster, R. A shortest cycle for each vertex of a graph. *Inform. Process. Lett.* **2011**, *111*, 1057–1061. [CrossRef]
5. Paulusma, D.; Yoshimoto, K. Cycles through specified vertices in triangle-free graphs. *Discuss. Math. Gr. Theory* **2007**, *27*, 179–191. [CrossRef]
6. Li, B.; Zhang, S. Heavy subgraph conditions for longest cycles to be heavy in graphs. *Discuss. Math. Gr. Theory* **2016**, *36*, 383–392. [CrossRef]
7. Li, B.; Xiong, L.; Yin, J. Large degree vertices in longest cycles of graphs, I. *Discuss. Math. Gr. Theory* **2016**, *36*, 363–382. [CrossRef]
8. Gerbner, D.; Keszegh, B.; Palmer, C.; Patkos, B. On the Number of Cycles in a Graph with Restricted Cycle Lengths. Available online: <https://arxiv.org/abs/1610.03476> (accessed on 9 October 2017).
9. Sankar, K.A.; Sarad, V. A time and memory efficient way to enumerate cycles in a graph. In Proceedings of the International Conference on Intelligent and Advanced Systems, Kuala Lumpur, Malaysia, 25–28 November 2007; IEEE: New York, NY, USA, 2007; pp. 498–500.
10. Haeupler, B.; Kavitha, T.; Mathew, R.; Sen, S.; Tarjan, R.E. Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. Available online: <https://arxiv.org/abs/1105.2397> (accessed on 9 October 2017).
11. Bender, M.A.; Fineman, J.T.; Gilbert, S.; Tarjan, R.E. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms* **2016**, *12*. [CrossRef]
12. Reif, J.H. Depth-first search is inherently sequential. *Inform. Process. Lett.* **1985**, *20*, 229–234. [CrossRef]

13. Karimi, M.; Banihashemi, A.H. Message-passing algorithms for counting short cycles in a graph. *IEEE Trans. Commun.* **2013**, *61*, 485–495. [[CrossRef](#)]
14. Rocha, R.C.; Thatte, B.D. Distributed cycle detection in large-scale sparse graphs. In Proceedings of the Simpósio Brasileiro de Pesquisa Operacional, Pernambuco, Brazil, 25–28 August 2015; XLVII SBPO: Pernambuco, Brazil, 2015; pp. 1–11.
15. Malewicz, G.; Austern, M.H.; Bik, A.J.C.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 6–11 June 2010; ACM: New York, NY, USA, 2010; pp. 135–146.
16. Gonzalez, J.E.; Xin, R.S.; Dave, A.; Crankshaw, D.; Franklin, M.J.; Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, Broomfield, CO, USA, 6–8 October 2014; USENIX: Berkeley, CA, USA, 2014; pp. 599–613.
17. Salihoglu, S.; Widom, J. GPS: A graph processing system. In Proceedings of the 25th International Conference on Scientific and Statistical Database Management, Baltimore, MD, USA, 29–31 July 2013; ACM: New York, NY, USA, 2013; pp. 1–31.
18. Féray, V. Weighted dependency graphs. Available online: <https://arxiv.org/abs/1605.03836> (accessed on 9 October 2017).
19. McLendon, W.; Hendrickson, B.; Plimpton, S.J.; Rauchwerger, L. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.* **2005**, *65*, 901–910. [[CrossRef](#)]
20. Leskovec, J.; Soscic, R. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.* **2016**, *8*. [[CrossRef](#)] [[PubMed](#)]
21. Seidl, T.; Boden, B.; Fries, S. CC-MR—Finding connected components in huge graphs with MapReduce. In Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Bristol, UK, 24–28 September 2012; Springer: Berlin/Heidelberg, Germany, 2014; pp. 458–473.
22. Slota, G.M.; Rajamanickam, S.; Madduri, K. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 19–23 May 2014; IEEE: New York, NY, USA, 2014; pp. 550–559.



© 2017 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).