Distributed cycle detection in large-scale sparse graphs

Rodrigo Caetano Rocha
Department of Computer Science, Universidade Federal de Minas Gerais
Belo Horizonte, MG – Brazil
rcor@dcc.ufmg.br

Bhalchandra D. Thatte
Department of Mathematics, Universidade Federal de Minas Gerais
Belo Horizonte, MG – Brazil
bhalchandra@mat.ufmg.br

ABSTRACT

In this paper we present a distributed algorithm for detecting cycles in large-scale directed graphs, along with its correctness proof and analysis. The algorithm is then extended to find strong components in directed graphs. We indicate an application to detecting cycles in number theoretic functions such as the proper divisor function. Our prototype implementation of the cycle detection algorithm, when applied to the proper divisor function, detects all sociable groups of numbers (cycles in the proper divisor function) up to 10^7.

KEYWORDS. Graph theory, cycle detection, distributed algorithms.

Main Area: TAG - Theory and Algorithms in Graphs.

RESUMO

Nesse artigo nós apresentamos um algoritmo distribuído para detectar ciclos em grafos massivos direcionados, juntamente com a sua análise e prova de corretude. O algoritmo é extendido para detectar componentes fortemente conectadas em grafos direcionados. Indicamos uma aplicação para a detecção de ciclos em funções de teoria dos números tal como a função dos divisores próprios. Nosso protótipo de implementação do algoritmo de detecção de ciclos, quando aplicado à função dos divisores próprios, detecta todos os grupos de números sociáveis (ciclos na função dos divisores próprios) até 10^7.

PALAVRAS CHAVE. Teoria dos grafos, detecção de ciclos, algoritmos distbuídos.

Área Principal: TAG - Teoria e Algoritmos em Grafos.

1. Introduction

The growing scale and importance of graph data has driven the development of numerous new parallel and distributed graph processing systems. For these massive data applications, the resulting graphs can have billions of connections, and are usually highly sparse with complex, irregular, and often a power-law structure [28].

There are great many theoretical and practical problems that deal with sparse graphs. For example, the Internet is a sparse network with average degree less than 4, see [8] and [10]. The networks of citations of scientific papers are also sparse, with the average number of references in a paper of the order of 10 [19]. Similarly, data sets of biological interactions often constitute sparse networks [7, 21], e.g., protein-protein interaction networks, gene regulation networks, etc. Biological neural networks are also represented by sparse networks [25]. Problems involving number theoretic functions also lead to sparse graphs, as described in Section 3.1.4.

Most parallel graph processing systems, such as Pregel [16], GraphX [29, 28], GPS [22], and Giraph [3], are mainly based on the *bulk synchronous parallel model* [26]. PowerGraph [12] supports both bulk synchronous and asynchronous implementations.

In the bulk synchronous parallel model [26] for graph processing, the large input graph is partitioned to the worker machines. Each worker machine is responsible for executing the vertices that are assigned to it. Then, a superstep concept is used for coordinating the parallel execution of computations on worker machines. A superstep consists of three parts: (i) concurrent computation, (ii) communication, (iii) synchronisation barrier. Every worker machine concurrently executes computations for the vertices it is responsible for. Each worker machine sends messages on behalf of the vertices it is responsible for to its neighbours. The neighbouring vertices may or may not be in the same worker machine. When a worker machine reaches the barrier, it waits until all other worker machines have finished their communication actions, before the system as a whole can move to the next superstep. A computation involves many supersteps executed one after the other in this manner. So, in a superstep, the worker uses values communicated via messages from the previous superstep, instead of most recent values.

In this paper, we propose a distributed algorithm for detecting cycles on large-scale directed graphs based on the bulk synchronous message passing abstraction. The proposed algorithm for detecting cycles by message passing is suitable to be implemented in distributed graph processing systems, and it is also suitable for implementations in systems for disk-based computations, such as the GraphChi [15], where the computation is mainly based on secondary memory. Disk-based computations are necessary when we have a single computer for processing large-scale graphs, and the computation exceeds the primary memory capacity.

The rest of this paper is structured as follows. Graph theoretic notation and preliminaries are fixed in Section 2. A cycle detection algorithm (Algorithm 1) based on message passing is presented in Section 3.1. The correctness of Algorithm 1 is proved in Section 3.1.1. In Sections 3.1.3 and 3.1.4, the total number of iterations and the number of messages exchanged at each iteration of Algorithm 1 are analysed. In Section 4, an algorithm for finding the strongly connected components of a graph is presented (Algorithm 2); the algorithm makes use of the cycle detection algorithm (Algorithm 1).

2. Graph theoretic notation

Throughout this paper we assume that $G \coloneqq (V, E)$ is a finite directed graph, where V is the set of vertices, $E \subseteq V \times V$ is the set of arcs, $\nu(G) \coloneqq |V|$, and $\epsilon(G) \coloneqq |E|$. Let $v \in V$. We denote the set of out-neighbours of v by $N^+(v)$, the set of in-neighbours of v by $N^-(v)$, the out-degree of v by $d^+(v)$, and the in-degree of v by $d^-(v)$. We denote by $\Delta^+(G)$ the maximum out-degree of vertices of G. A *walk* in G is a sequence $(v_0, e_1, v_1, \ldots, e_k, v_k)$, where each v_i is a vertex, and each e_i is an arc from v_{i-1} to v_i. The length of a walk is the number of arcs in the walk. A closed walk is a walk $(v_0, e_1, v_1, \ldots, e_k, v_k)$ in which the *origin* v_0 and the *terminus* v_k are equal. A *cycle* is a closed walk in which all vertices except the origin and the terminus are distinct.

A cycle of length 0 has a single vertex and no arcs. In this paper, we consider only non-trivial cycles, i.e., cycles containing at least one arc. A cycle of length 1 has one arc, and corresponds to a loop in the graph. Since our definition of a directed graph permits at most one arc between any pair of vertices, there is no ambiguity when we write a cycle of length k by a sequence of $k+1$ vertices. For example, (v) is a cycle of length 0, and (v, v) is a cycle of length 1, and so on. We refer to [27] for standard graph theoretic notions not defined above.

3. Searching for cycles

An efficient method for detecting cycles in a directed graph is to use the depth-first search (DFS) algorithm, considering the fact that a directed graph has a cycle if and only if DFS finds a *back arc*. The running time of DFS on a directed graph G is $\Theta(\nu(G) + \epsilon(G))$, and it is asymptotically optimal [6].

Although DFS is asymptotically optimal, Reif [20] suggests that it cannot be effectively parallelised, by proving the polynomial time completeness (P-completeness) of DFS. The remainder of this section describes and analyses a distributed cycle detection algorithm with intrinsic potential for parallelism.

3.1. Detecting cycles by message passing

In the context of *big data*, where the graph structure can be large enough to saturate the processing power or memory capacity of a single machine, it is difficult to effectively parallelise the DFS algorithm. Hence we need an algorithm that divides the problem into subproblems among computational nodes, so that the nodes can search for cycles in a parallel manner with the certainty that all cycles are found.

We propose a general algorithm for detecting cycles in a directed graph G by message passing among its vertices, based on the bulk synchronous message passing abstraction. This is a vertex-centric approach in which the vertices of the graph work together for detecting cycles. The bulk synchronous parallel model consists of a sequence of iterations, in each of which a vertex can receive messages sent by other vertices in the previous iteration, and send messages to other vertices.

In each pass, each active vertex of G sends a set of sequences of vertices to its out-neighbours as described next. In the first pass, each vertex v sends the message (v) to all its out-neighbours. In subsequent iterations, each active vertex v appends v to each sequence it received in the previous iteration. It then sends all the updated sequences to its out-neighbours. If v has not received any message in the previous iteration, then v deactivates itself. The algorithm terminates when all the vertices have been deactivated.

For a sequence $(v_1, v_2, \ldots, v_k)$ received by vertex v, the appended sequence is not forwarded in two cases: (i) if $v = v_1$, then v has detected a cycle, which is reported (see line 9 of Algorithm 1); (ii) if $v = v_i$ for some $i \in \{2, 3, \ldots, k\}$, then v has detected a sequence that contains the cycle $(v = v_i, v_{i+1}, \ldots, v_k, v_{k+1} = v)$; in this case, the sequence is discarded, since the cycle must have been detected in an earlier iteration (see line 11 of Algorithm 1); to be precise, this cycle must have been detected in iteration $k - i + 1$. Every cycle $(v_1, v_2, \ldots, v_k, v_{k+1} = v_1)$ is detected by all $v_i, i = 1$ to k in the same iteration; it is reported by the vertex $\min\{v_1, \ldots, v_k\}$ (see line 9 of Algorithm 1).

The total number of iterations of the algorithm is the number of vertices in the longest path in the graph, plus a few more steps for deactivating the final vertices. During the analysis of the total number of iterations, we ignore the few extra iterations needed for deactivating the final vertices and detecting the end of the computation, since it is $O(1)$. In practice, the actual number of these final few iterations depends on the framework being used to implement the algorithm.

We count iterations as $i = 0, 1, \ldots$. Let $M_i^{(v)}$ be the set of messages (sequences of vertices) received by v at iteration i. Since messages sent in iteration $i = 0$ are received in iteration $i = 1$, $M_0^{(v)} = \emptyset$.

Algorithm 1 Pseudocode for the compute function of the distributed cycle detection algorithm. The algorithm takes G as input, and for each superstep i the function COMPUTE($M_i^{(v)}$) is executed for each active vertex v.

```
1:  function COMPUTE(M_i^(v))
2:      if i = 0 then
3:          for each w ∈ N+(v) do
4:              send (v) to w
5:      else if M_i^(v) = ∅ then
6:          deactivate v and halt
7:      else
8:          for each (v_1, v_2, ..., v_k) ∈ M_i^(v) do
9:              if v_1 = v and min{v_1, v_2, ..., v_k} = v then
10:                 report (v_1 = v, v_2, ..., v_k, v_{k+1} = v)
11:             else if v ∉ {v_2, ..., v_k} then
12:                 for each w ∈ N+(v) do
13:                     send (v_1, v_2, ..., v_k, v) to w
```

Figure 1 presents an example of the execution of the algorithm. In iteration $i = 3$, all the three vertices detect the cycle [2, 3, 4]. We ensure that the cycle is reported only once by emitting the detected cycle only from the vertex with the least identifier value in the ordered sequence, which is the vertex 2 in the example.

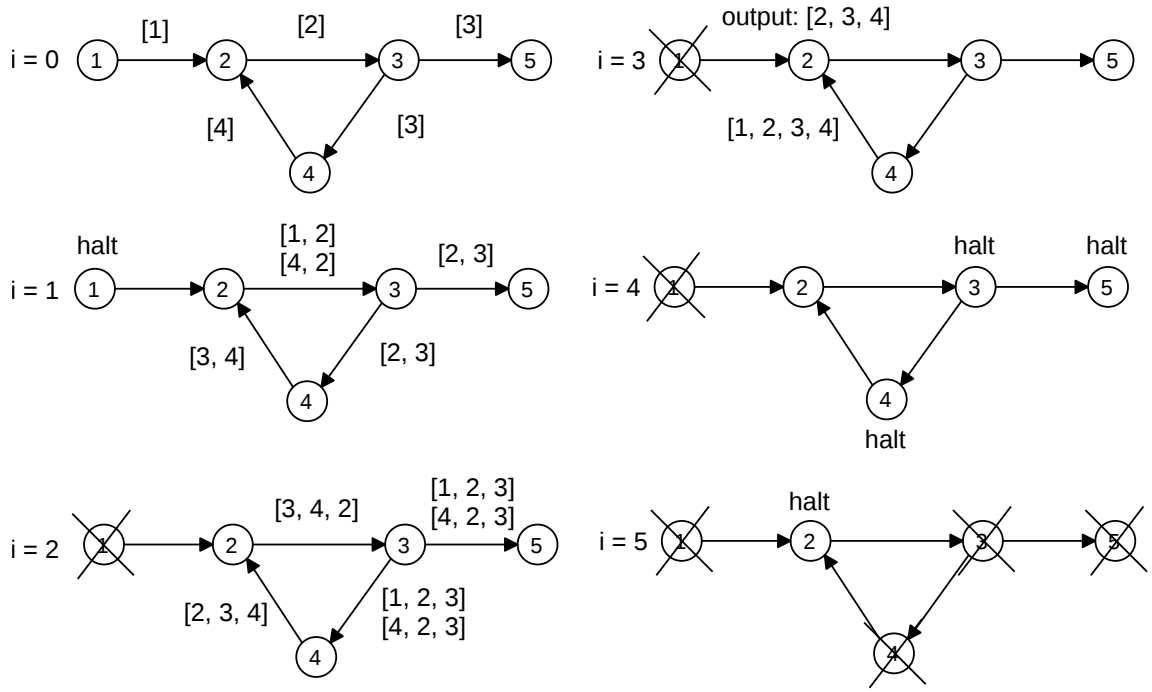

Figure 1: Example of the algorithm for detecting cycles by message passing.

3.1.1. A correctness proof using loop-invariant

Let $\mathcal{C}$ be the set of all non-trivial cycles in G (i.e., cycles with at least one arc). Let $S := (v_1, v_2, \ldots, v_i)$ be a message and u a vertex; we define $S.u := (v_1, v_2, \ldots, v_i, u)$ as the concatenation of the message S with the vertex u.

Loop-invariant. For $i \geq 1$, the set $\mathcal{C}_i$ contains all non-trivial cycles of length i, which are detected and reported in iteration i, and the set $M_i^{(v)}$ contains all messages of length i received by v in iteration i.
Base case. In iteration $i = 1$, each vertex v receives a message (w) from each of its in-neighbours w. Hence, $M_1^{(v)} = \{(w)|w \in N^-(v)\}$. If there is a loop at v, then v receives a message (v) in iteration $i = 1$; in this case, v detects and reports the cycle (v, v) of length 1. Hence, at the end of iteration $i = 1$, the set $\mathcal{C}_1$ contains all cycles of length 1.
Induction hypothesis. Suppose the loop invariant holds at the end of iteration i for some $i \geq 1$, i.e., $\mathcal{C}_i$ contains all cycles of length i detected in iteration i, and $M_i^{(v)}$ contains all messages of length i received by v in iteration i.
Inductive step. We prove that

$$M_{i+1}^{(v)} = \bigcup_{w \in N^-(v)} \{S.w | S \coloneqq (v_1, v_2, \ldots, v_i) \in M_i^{(w)} \text{ and } w \neq v_k, \forall k \in [1, i]\}$$

and

$$\mathcal{C}_{i+1} = \bigcup_{v \in V(G)} \{S.v | S \coloneqq (v_1, v_2, \ldots, v_{i+1}) \in M_{i+1}^{(v)} \text{ and } v_1 = v\}.$$

By induction hypothesis, the set $M_i^{(w)}$ contains all messages of length i that reach w, for all $w \in N^-(v)$. If $S \coloneqq (v_1, v_2, \ldots, v_i) \in M_i^{(w)}$ and $w \notin S$, then w sends the message $S.w \coloneqq (v_1, v_2, \ldots, v_i, v_{i+1} = w)$ (of length $i + 1$) to all its out-neighbours (one of them being v). Hence the message $S.w$ is received by v in iteration $i+1$. Thus $M_{i+1}^{(v)}$ is the set of messages of length $i+1$ that reach v in iteration $i+1$. To prove that C_{i+1} contains all cycles of length $i+1$, observe that the set $M_{i+1}^{(v)}$ contains $(v_1 = v, v_2, \ldots, v_{i+1})$ iff there exists a cycle $(v_1 = v, v_2, \ldots, v_i, v_{i+1}, v_{i+2} = v)$ of length $i+1$ that starts and finishes at vertex v. Therefore, the loop-invariant still holds at iteration $i + 1$, and the algorithm constructs $\mathcal{C} = \bigcup_{k=1}^{\nu(G)} \mathcal{C}_k$.

3.1.2. Graph partitioning and communication by message passing

When considering distributed graph processing frameworks, the input graph partitioning is crucial for achieving an efficient distributed execution. Considering that the graph structure describes data movement, minimisation of storage and communication overhead, and balanced computation depend on the graph partitioning performed by the framework.

Most of the frameworks for processing graphs try to optimise the partitioning strategy, maximising the number of messages exchanged directly via shared memory communication. The two most common partitioning strategies are based on edge-cut and vertex-cut.

In the *edge-cut* partitioning scheme [12, 29], the vertex set of a graph is partitioned into blocks, and each block of the partition is processed on a distinct worker machine. Messages between vertices in the same block are exchanged directly via main memory, reducing communication overhead and data movement via network. Since constructing an optimal edge-cut for large-scale graphs can be prohibitively expensive, many graph processing frameworks use a random edge-cut (i.e., randomly distribute vertices across the cluster).

Vertex-cuts evenly assign edges to machines, and allow vertices to span multiple worker machines. For power-law graphs, the vertex-cut strategy can reduce communication overhead and ensure balanced computation by evenly assigning edges to machines in a way that minimises the number of machines spanned by each vertex [12, 29].

3.1.3. The worst case and the average case analysis of the number of messages sent in iteration t

Let G be a graph on n vertices. The worst case scenario occurs when G is a complete directed graph with loops. In this case, we have n iterations. Let $n^{\underline{t}}$ denote the falling factorial

$n(n-1)\cdots(n-t+1)$. Let Y_t be the total number of messages sent during iteration t, for $t \geq 0$. Let $\mathcal{D}_{n,p}$ denote the model of random directed graphs in which each possible arc (i,j) (with possibly $i = j$) has probability p. This model is similar to the Erdös-Rényi random graph model $\mathcal{G}_{n,p}$ [4, 13]. In the case of a random graph, Y_t is the random variable representing the total number of messages sent during iteration t.

Proposition 3.1.

1. *When G is a complete graph, we have $Y_t = n\, n^{\underline{t+1}}$.*
2. *When G is a random graph from $\mathcal{D}_{n,p}$, we have*

$$\mathbb{E}[Y_t] = n\, n^{\underline{t+1}}\, p^{t+1}.$$

Proof. First we prove the formula for Y_t for a complete graph. Every message sent in iteration t is of the type $(v_0, ..., v_t)$, where v_i are distinct vertices. There are $n^{\underline{t+1}}$ such sequences. Moreover, each such sequence is a message. The multiplicity of a fixed message $(v_0, ..., v_t)$ is n; it is sent by v_t to all its out-neighbours (including itself). Hence

$$Y_t = n\, n^{\underline{t+1}}.$$

The argument is similar when G is a random graph from $\mathcal{D}_{n,p}$. Let $S := (v_0, ..., v_t)$, where v_i are distinct vertices. Let u be an arbitrary vertex. Let a binary random variable $X_{S,u}$ be defined as follows.

$$X_{S,u} = \begin{cases} 1 & \text{if message } S \text{ is sent by } v_t \text{ to } u; \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$Y_t = \sum_{S,u} X_{S,u}.$$

The message S is sent by vertex v_t to vertex u in iteration t if each of the arcs $(v_0, v_1), \ldots, (v_{t-1}, v_t), (v_t, u)$ is present in G. Thus $\mathbb{E}[X_{S,u}] = \mathtt{Pr}\{X_{S,u} = 1\} = p^{t+1}$. There are $n\, n^{\underline{t+1}}$ possible pairs (S, u). Hence, by linearity of expectation,

$$\mathbb{E}[Y_t] = n\, n^{\underline{t+1}}\, p^{t+1}. \qquad \square$$

Let Z_G be the number of iterations before Algorithm 1 terminates. In the case of a random directed graph, Z_G is the random variable representing the number of iterations before Algorithm 1 terminates.

Proposition 3.2. *The number of iterations Z_G of Algorithm 1 on a graph G is equal to the length of the longest path in G. When G is a random graph from $\mathcal{D}_{n,p}$, for every ϵ in $(0,1)$, we have*

$$\mathbb{E}[Z_G] = \begin{cases} \Theta(\log n) & \text{if } p = \frac{1-\epsilon}{n} \\ \Theta(\sqrt{n \log n}) & \text{if } p = \frac{1}{n} \\ \Theta(n) & \text{if } p = \frac{1+\epsilon}{n} \end{cases}$$

Proof. The first part is obvious, since a message longer than the longest path has at least one vertex repeated, and hence is discarded. The second part follows directly from Ajtai et al. [2], where the length of the longest path in a random directed graph is analysed in the three cases stated above. $\square$

3.1.4. Graphs with out-degree at most 1

In this section, we assume that G has maximum out-degree $\Delta^+(G) = 1$.

Lemma 3.3. *The number of messages Y_t sent in iteration t is at most $\nu(G)$, for all $t \geq 0$.*

Proof. At $t = 0$, the total number of messages sent is $\sum_v d^+(v) = \epsilon(G)$; hence the number of messages received at $t = 1$ is $|M_1| = \epsilon(G)$, which is at most $v(G)$ since $d^+(v) \leq 1$ for all v.

In any iteration t, the same message S is never received by two distinct vertices u and v, since otherwise there would exist a vertex w that sent a message S to u and v, which is not possible since $d^+(w) \leq 1$ for all $w \in V(G)$.

Suppose that u receives message S at time t. Then there are 3 possibilities:

1. if $d^+(u) = 0$, then u does not send any message.
2. if $d^+(u) = 1$ and $S.u$ contains a cycle, then u does not send the message $S.u$.
3. if $d^+(u) = 1$ and $S.u$ does not contain a cycle, then u sends the message $S.u$ at time t to its (unique) out-neighbour.

Therefore, there is a one-to-one map ($S.u \to S$) from the set of messages sent at time t (i.e., the set M_{t+1} of messages received at time $t + 1$) to the the set of messages sent at time $t - 1$ (i.e., the set M_t of messages received at time t). Hence $|M_{t+1}| \leq |M_t|$ for all t, which implies that $|M_t| \leq |M_1| = \epsilon(G) \leq \nu(G)$. □

Since $\Delta^+(G) = 1$, each component G_i of G is either a tree or a unicyclic graph. If a component G_i is a tree, then it contains a unique vertex r of out-degree 0, and each arc (u, v) is directed towards r (i.e., vertex v is on the path from u to r). If G_i is a unicyclic graph, then G_i contains a unique directed cycle, say $(u_1, u_2, \ldots, u_k, u_1)$, and each arc (u, v) not on the cycle is directed towards a unique point on the cycle (i.e., the vertex v is on the path from u to a unique vertex on the cycle). Figure 2 shows a graph with two unicyclic components and directed trees rooted at points on the cycle.

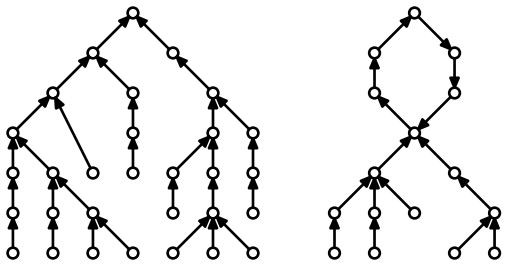

Figure 2: Example of a graph G with $\Delta^+(G) = 1$ and two components, one of which is a (directed) tree and the other is unicyclic. In this case, the total number of iterations is $\max_i\{h_i + k_i + 1\} = 3 + 6 + 1 = 10$

Lemma 3.4. *Let $G_i, i = 1, 2, \ldots$ be the components of G. Let k_i be the length of the cycle in G_i. Let h_i be the maximum height of a tree rooted at a vertex on the cycle in G_i. If G_i is a tree, then $k_i = 0$, and h_i is the maximum distance between a leaf of G_i and the unique sink vertex in G_i. Then the total number of iterations of Algorithm 1 is $\max_i\{h_i + k_i + 1\}$.*

Proof. Let $v \in V(G_i)$. If the path from v to a unique vertex on the cycle has length l (i.e., it has l arcs), then a message starting at v at $t = 0$ reaches a vertex on the cycle at $t = l$. The message takes k_i more iterations to go around the cycle. Hence it is discarded at $t = l + k_i + 1$ when the cycle is detected. The maximum value of l is h_i. Hence the last message is discarded at iteration $\max_i\{h_i + k_i + 1\}$. □

Algorithm 1 in the special case $\Delta^+(G) = 1$ may be used to detect cycles in number theoretic functions. Consider a function $f : \mathbb{N} \to \mathbb{N}$. Let $G_{\mathbb{N},f}$ be a directed graph with vertex set $\mathbb{N}$ and the set of arcs $E \coloneqq \{(a, f(a)) \mid a \in \mathbb{N}\}$. To detect cycles in $G_{\mathbb{N},f}$ on vertices in $[1, n]$, we define $U \coloneqq [1, n] \bigcup \{f(k) | k \in [1, n]\}$, and $G_{n,f}$ to be the restriction of $G_{\mathbb{N},f}$ to U. As in André Joyal's proof of Cayley's formula [14], each component of $G_{n,f}$ is either a tree or is unicyclic.

As an illustration, we have implemented Algorithm 1 to discover *sociable numbers*. For a positive integer n, let $\sigma_1(n)$ be the number of divisors of n. We denote by $s(n)$ the sum of proper positive divisors of n, i.e., $s(n) \coloneqq \sigma_1(n) - n$. The function $s(n)$ is called the *restricted divisor function*. We say that n is a *perfect number* if $n = s(n)$. Similarly, a pair (m, n) of positive integers is an *amicable pair* if $s(n) = m$ and $s(m) = n$ [30]. *Sociable numbers* are numbers that result in a *periodic aliquot sequence*, also known as *aliquot cycles*, where an aliquot sequence is the sequence $n, s(n), s^2(n), \ldots, s^k(n)$ obtained by repeatedly applying the restricted divisor function. If the period of the aliquot cycle is 1 (i.e., $s(n) = n$), then n is a perfect number. If the period is 2 (i.e., $s^2(n) = n$), then the two numbers n and $s(n)$ constitute an amicable pair. In general, if the period is t, then the sequence of numbers is said to be a *sociable group of order* t [5, 9, 30]. Our prototype implementation of Algorithm 1, running on the GraphChi disk-based processing system, detected all 111 known social groups up to 10^7, including perfect and amicable numbers, in a total of 180 iterations. By using this graph-based approach, we can search for sociable numbers in a more systematic manner than when compared to other exhaustive approaches [11, 17, 18].

4. Detecting strongly connected components

Define a relation $\sim$ on $V(G)$ as follows: for all $u, v \in V(G)$, define $u \sim v$ if and only if there are paths from u to v and from v to u. The relation $\sim$ is an equivalence relation. The subgraphs of G induced by the equivalence classes of $\sim$ are called strongly connected components of G. The definition of $\sim$ implies that each cycle in G is contained in a unique strongly connected component (SCC).

Let $\mathcal{C} \coloneqq C_i, i \in [1, n]$ be a sequence of cycles such that either $n = 1$ or for all $k \in [2, n]$, C_k intersects $\cup_{i=1}^{k-1} C_i$. Then $H \coloneqq \cup_{i=1}^{n} C_i$ must be a subgraph of a strongly connected component.

Lemma 4.1. *If $\mathcal{C}$ is maximal in the sense that each cycle C not in $\mathcal{C}$ is either vertex disjoint with H or is a subgraph of H, then H is a strong component.*

Proof. If H is not a strong component, then there is a unique a strong component H' that contains H as a proper subgraph. Let $x \in V(H')\backslash V(H)$ and $y \in V(H)$ such that $(x, y) \in E(G)$. Moreover, there must be a directed path, say P, from y to x in H'. Now $P \cup (x, y)$ is a directed cycle not in $\mathcal{C}$ that intersects $\mathcal{C}$, which implies that $\mathcal{C}$ is not maximal, which is a contradiction. □

Similar to the problem of cycle detection, most algorithms for partitioning a graph into SCCs, such as Tarjan's algorithm [24] and Kosaraju-Sharir algorithm [1, 23], are based on DFS. Thus, detecting SCCs has similar issues as cycle detection, when dealing with large-scale graphs.

Although Algorithm 2 is intended to be a cetralised system, it can be integrated with Algorithm 1. Whenever Algorithm 1 finds a new cycle, the cycle is reported to the centrilised system, where the merge function may be concurrently executed. Since both algorithms execute in parallel, the time required to merge the SCCs overlaps with the time to compute the cycles.

Algorithm 2 Pseudocode for partitioning a directed graph into strongly-connected components.

```
1: function COMPOSESCC(G)
2:     SCC ← V(G)
3:     for each C ∈ CYCLES(G) do
4:         SCC ← MERGE(SCC, C)
5:     return SCC
6: function MERGE(SCC, C)
7:     newSCC ← ∅
8:     for each S ∈ SCC do
9:         if C ∩ S = ∅ then
10:            newSCC ← newSCC ∪ {S}
11:        else
12:            C ← C ∪ S
13:    newSCC ← newSCC ∪ {C}
14:    return newSCC
```

5. Acknowledgments

The authors thank Daniel M. Martin (UFABC) for many useful comments on this work.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data structures and algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Co., Reading, Mass., 1983.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. The longest path in a random graph. *Combinatorica*, 1(1):1–12, 1981.

[3] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[4] Béla Bollobás. *Random graphs*. Springer, 1998.

[5] Henri Cohen. On amicable and sociable numbers. *Math. Comp.*, 24:423–429, 1970.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[7] Amira Djebbari and John Quackenbush. Seeded bayesian networks: constructing genetic networks from microarray data. *BMC systems biology*, 2(1):57, 2008.

[8] Sergey N Dorogovtsev and Jose FF Mendes. Evolution of networks. *Advances in physics*, 51(4):1079–1187, 2002.

[9] P. Erdős. On asymptotic properties of aliquot sequences. *Math. Comp.*, 30(135):641–645, 1976.

[10] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[11] Achim Flammenkamp. New sociable numbers. *mathematics of computation*, pages 871–873, 1991.

[12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[13] Svante Janson, Tomasz Łuczak, and Andrzej Ruciński. Random graphs. 2000. *Wiley–Intersci. Ser. Discrete Math. Optim*, 2000.

[14] André Joyal. Une théorie combinatoire des séries formelles. *Adv. in Math.*, 42(1):1–82, 1981.

[15] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[16] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[17] David Moews and Paul C Moews. A search for aliquot cycles below 10^{10}. *mathematics of computation*, 57(196):849–855, 1991.

[18] David Moews and Paul C Moews. A search for aliquot cycles and amicable pairs. *Mathematics of computation*, 61(204):935–938, 1993.

[19] Sidney Redner. How popular is your paper? an empirical study of the citation distribution. *The European Physical Journal B-Condensed Matter and Complex Systems*, 4(2):131–134, 1998.

[20] John H. Reif. Depth-first search is inherently sequential. *Inform. Process. Lett.*, 20(5):229–234, 1985.

[21] Jean-François Rual, Kavitha Venkatesan, Tong Hao, Tomoko Hirozane-Kishikawa, Amélie Dricot, Ning Li, Gabriel F Berriz, Francis D Gibbons, Matija Dreze, Nono Ayivi-Guedehoussou, et al. Towards a proteome-scale map of the human protein–protein interaction network. *Nature*, 437(7062):1173–1178, 2005.

[22] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[23] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Comput. Math. Appl.*, 7(1):67–72, 1981.

[24] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[25] Henry C Tuckwell and Jianfeng Feng. Estimation of spike train statistics in spontaneously active biological neural networks. In *Networks: From Biology to Theory*, pages 129–141. Springer, 2007.

[26] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[27] D.B. West. *Introduction to graph theory*. Prentice Hall, 2001.

[28] Reynold S. Xin, Daniel Crankshaw, Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *CoRR*, abs/1402.2394, 2014.

[29] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[30] Song Y. Yan. *Perfect, amicable and sociable numbers*. World Scientific Publishing Co., Inc., River Edge, NJ, 1996. A computational approach.