Computing and Informatics, Vol. 39, 2020, 952–972, doi: 10.31577/cai_2020_5_952

A MAPREDUCE ALGORITHM FOR MINIMUM VERTEX COVER PROBLEMS AND ITS RANDOMIZATION

Morikazu Nakamura, Daiki Kinjo, Takeo Yoshida

Faculty of Engineering
University of the Ryukyus
Okinawa 903-0213, Japan
e-mail: morikazu@ie.u-ryukyu.ac.jp

Abstract. MapReduce is a programming paradigm for large-scale distributed information processing. This paper proposes a MapReduce algorithm for the minimum vertex cover problem, which is known to be NP-hard. The MapReduce algorithm can efficiently obtain a minimal vertex cover in a small number of rounds. We show the effectiveness of the algorithm through experimental evaluation and comparison with exact and approximate algorithms which demonstrates a high quality in a small number of MapReduce rounds. We also confirm from experimentation that the algorithm has good scalability allowing high-quality solutions under restricted computation times due to increased graph size. Moreover, we extend our algorithm to randomized one to obtain a good expected approximate ratio.

Keywords: MapReduce, minimum vertex cover, Hadoop, optimization, algorithm design, randomized algorithm

1 INTRODUCTION

MapReduce is a programming paradigm introduced by Google [1] as a promising software platform for large-scale distributed information processing. MapReduce uses functional programming and is composed of *mappers* for per-record computation and *reducers* for results aggregation [2]. MapReduce platforms can be implemented on a large number of commodity computers or computer clusters which provides scalability and fault tolerance – the most important characteristics required

for large-scale distributed processing [3, 4]. The number of processing nodes can be easily increased to handle growing large data.

Spark is another distributed computing platform for big-data analysis [5]. Compared to Hadoop, a well-known MapReduce platform, Spark's in-memory computation is high-speed [6, 7]. On the other hand, Hadoop was designed for efficiency in cost and time. This mechanism, based on an enormous volume of data on several storage nodes leads to outstanding scalability with lower costs, even though the real-time computation is sacrificed. Although Hadoop and Spark are often compared, their roles are differentiated, and they can coexist mutually [6, 7].

Large graphs are often used for modeling various real life objects, systems, and services, for example, road networks, relations among SNS (social network service) members, citation networks of research papers, the hyperlink structure of web pages, and various relations among pieces of digital content on the Internet. The number of vertices in such graphs may be several million, hundreds of millions, several billion, or even more. For such huge graphs, structured data mining requires graph algorithms such as breadth-first search (BFS), depth-first search (DFS), minimum spanning tree (MST), or shortest path problem (SPP). Traditional computing platforms and paradigms are not suited to this task because they are insufficiently scalable. Therefore, MapReduce algorithms for graph problems are an important research field [8, 9, 10, 11, 12, 13, 14, 15]. Many application areas of this topic are expected in engineering, biology, and the medical sciences [16, 17]. Several MapReduce programming platforms have been so far developed [13, 14, 15, 18, 19, 20] that provide APIs for graph operations and show how to implement some basic algorithms, such as page ranking, SPP, and MST, by using those APIs. There are also MapReduce algorithms for the maximum clique problem [21], the maximum cover problem [22], the maximum flow problem [23], and the shortest-path problem [24].

The minimum vertex cover problem (MVC) is a basic problem in graph theory, and it is well known to be computationally intractable, that is, NP-hard [25]. Approximate solutions with a two-approximation ratio can be easily constructed from maximal matching, and then the approximation factor has been slightly improved [26, 27].

We propose a MapReduce algorithm for MVC, a greedy algorithm that drastically improves the solution quality compared to maximal matching-based algorithms. In our previous works, we presented the first version of the algorithm without detailed experimental evaluation and deep discussion [28]. Moreover, in this paper we extend the original algorithm into randomized one to avoid the worst case solution quality happened in specific situations.

In [32], MapReduce algorithms for well-known problems are proposed, where they show the theoretical approximation ratio is two for the minimum vertex cover problem. On the other hand, our first algorithm may lead to solutions of worse quality for special cases than the MapReduce algorithm in the literature. However our randomized version can overcome such worst case situation and obtain solutions with expected approximation ratio 4/3. Moreover we show by experimental eval-

uations that obtained solutions are quite better than the expected approximation ratio.

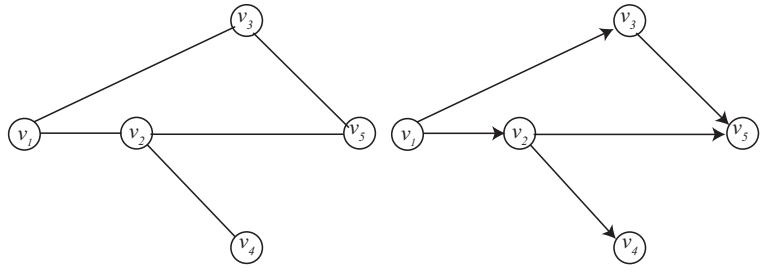
The remainder of the paper is organized as follows. In Section 2, we briefly give some basic background on graphs, MVC and MapReduce. In Section 3, we propose our MapReduce algorithm for MVC and show its correctness. In Section 4, we perform a computational experiment to evaluate our proposal. In Section 5, a randomized MapReduce algorithm is presented. Finally, in Section 6, we conclude the paper with some remarks.

2 PRELIMINARIES

This section explains some graph notations, the definition of a minimum vertex cover, and a basic background on MapReduce for the readability of the remaining paper.

2.1 Graphs

Graphs are denoted by a 2-tuple $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ with vertex set $\mathbf{V}$ and edge set $\mathbf{E}$, where the elements of $\mathbf{E}$ are 2-element subsets of $\mathbf{V}$. If each edge in $\mathbf{E}$ has an orientation (that is, a direction), the graph becomes a directed graph and is shown as $\vec{\mathbf{G}} = (\mathbf{V}, \vec{\mathbf{E}})$. Figure 1 a) shows the graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where $\mathbf{V}=\{v_1, v_2, v_3, v_4, v_5\}$ and $\mathbf{E} = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_5)\}$. Figure 1 b) shows a directed graph whose underlying graph is the graph in (a).



a) Undirected graph b) Directed graph

Figure 1. Graph

The degree $\mathsf{d}(v)$ of a vertex v is the number of edges connected to v. In the case of directed graphs, the in-degree $\mathsf{d}^{in}(v)$ and the out-degree $\mathsf{d}^{out}(v)$ of a vertex v are defined as the number of incoming and outgoing edges, respectively. Examples are $\mathsf{d}(v_1) = 2$, $\mathsf{d}(v_2) = 3$ in Figure 1 a) and $\mathsf{d}^{in}(v_2) = 1$ and $\mathsf{d}^{out}(v_2) = 2$ in Figure 1 b). A matching $\mathbf{M}$ of a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is a subset of $\mathbf{E}$ such that no pair of edges in $\mathbf{M}$ shares a vertex. A matching is maximal if adding any edge not in $\mathbf{M}$ results in no longer being a matching. A vertex cover $\mathbf{VC}$ of a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is a set of

vertices such that each edge in **E** is incident to at least one vertex in **VC**. For example, in Figure 1, $\{(v_1, v_3), (v_2, v_5)\}$ is a maximal matching and $\{v_2, v_3\}$ is a vertex cover. More detailed information on graphs can be referred to the literature [31, 30].

2.2 Minimum Vertex Cover Problem

The minimum vertex cover problem is a classical graph problem and is known to be NP-hard. The problem is formulated for an input graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ with the vertex set $\mathbf{V} = \{1, 2, \ldots, n\}$ and the edge set $\mathbf{E} = \{(v, u) | v \text{ and } u \in \mathbf{V}\}$ as an integer programming problem.

MVC (Integer Programming):

$$\min \sum_{v \in \mathbf{V}} x_v \tag{1}$$

s.t.

$$x_u + x_v \geq 1, \forall (u, v) \in \mathbf{E}, \tag{2}$$

$$x_v \in \{0, 1\}. \tag{3}$$

For the problem, a factor-2 approximate solution can be obtained from a maximal matching constructed by a simple greedy algorithm. Figure 2 gives an approximate solution. The dotted edges express a maximal matching. We can easily confirm that all endpoints of the dotted edges, the vertices with bold lines, comprise a vertex cover of the graph. The approximation factor is no more than two since the minimum vertex cover should include at least one endpoint of each edge in the matching.

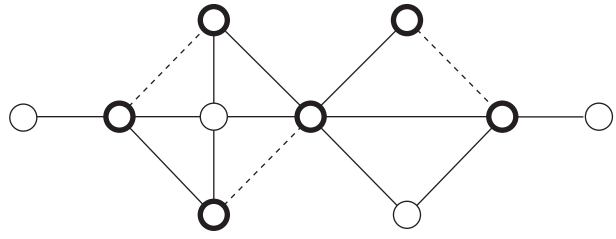


Figure 2. Approximation solution based on maximal matching

2.3 MapReduce Model

MapReduce is a framework for large-scale distributed computing based on the well-known master-slave parallel processing pattern [1]. MapReduce programs are composed of map operations and reduce operations. Map operations play a role in the

per-record computation and Reduce operations in the results aggregation. In Map operations, the master divides the input data into multiple data sets and assigns these sets to worker nodes. Note that the input data can be stored beforehand in distributed data storage near worker nodes to reduce the overhead incurred by data division and assignment. Each worker processes the assigned data according to the map operation and returns the output to its master node. In Reduce operations, the master lets workers collect the outputs of the map operations and combine them to form the answer to the problem. Workers perform these operations in parallel, taking key–value pairs as processing primitives. The MapReduce processing can be formally explained as follows.

For sets of the input key–value pairs $U_0^i, i = 0, 1, \ldots, m-1$, the MapReduce program performs the following steps.

For $t = 1, 2, 3, \ldots, T$ do

1. **Execute Map**: Input each pair (k, v) in U_{t-1}^i to mapper M_i, $i = 0, 1, \ldots, m-1$ and the mapper performs the Map operation. Each mapper generates a sequence of pairs, (k_1, v_1), (k_2, v_2), $\ldots$, as the result. Let us denote the multiset of key–value pairs generated by M_i at t^{th} round by $\hat{U}_t^i$, used in **Shuffle** step.
2. **Shuffle**: For each k, let $V_{k,i}$ be the multiset of values v_j such that $(k, v_j) \in \hat{U}_t^i$. The MapReduce system constructs the multiset $V_{k,i}$ from $\hat{U}_t^i$.
3. **Execute Reduce**: For each k, input k and some arbitrary permutation of $V_{k,i}$ to reducer $R_i, i = 0, 1, \ldots, r-1$ and perform the Reduce operation. The reducer generates a sequence of key–value pairs (k, v_1'), (k, v_2'), $\ldots$, as the result. Let U_t^i, $i = 0, 1, \ldots, m-1$ be the multiset of key–value pairs generated by R_j, $j = 0, 1, \ldots, r-1$.

MapReduce is a promising platform for distributed large-scale computation. However, algorithms for MapReduce are quite different from ordinary algorithms we have used. Therefore, we need to design suitable algorithms for the platform.

3 MAPREDUCE ALGORITHM FOR MVC

In this section we propose a MapReduce algorithm for the minimum vertex cover problem. Algorithm 1 shows a pseudocode for the algorithm, MapReduceMVC.

Before going to the explanation, we introduce some notations. Functions $\mathsf{N} : \mathbf{V} \to 2^{|\mathbf{V}|}$ and $\mathsf{d} : \mathbf{V} \to \{0, \ldots, |\mathbf{V}| - 1\}$ return the set of the vertices neighboring v and degree of $v, v \in \mathbf{V}$, respectively. Function $\mathsf{index} : \mathbf{V} \to \{1, \ldots, |\mathbf{V}|\}$ returns the index of v.

A relation $\prec$ on $\mathbf{V}$ and directed graphs induced by $\prec$ is defined as follows:

Definition 1. Let $\mathbf{G}$ be an undirected graph with vertex set $\mathbf{V}$ and edge set $\mathbf{E}$. Let us denote by $v \prec u$ the relation such that $(\mathsf{d}(v), \mathsf{index}(v))$ is smaller than $(\mathsf{d}(u), \mathsf{index}(u))$ in the lexicographical order on $\{0, \ldots, |V| - 1\} \times \{1, \ldots, |V|\}$.

Algorithm 1 Pseudocode for MapReduce algorithm

procedure MapReduceMVC:
−−STEP1:
$VC[v] \leftarrow$ **true**, for all v;
$Adj[v] \leftarrow$ **true**, for all v;
−−STEP2:
if v is source **then**
  $VC[v] \leftarrow$ **false**;
end if
repeat
  −−STEP3:
  if $VC[u] =$ **true**, for all $u \in \mathsf{N}(v)$ **then**
    $Adj[v] \leftarrow$ **true**;
  else
    $Adj[v] \leftarrow$ **false**;
  end if
  −−STEP4:
  if $VC[v] =$ **true and** $Adj[v] =$ **true and** $\exists u \in \mathsf{N}(v),\ Adj[u] =$ **false then**
    $VC[v] \leftarrow$ **false**;
  end if
  if $VC[v] =$ **false and** $Adj[v] =$ **false then**
    Let N^f be $\{u | u \in \mathsf{N}(v), VC(u) =$ **false**$\}$;
    if $u \prec v, \exists u \in \mathsf{N}^f$ **then**
      $VC[v] \leftarrow$ **true**;
    end if
  end if
until No modification is taken place in STEP4
−−STEP5:
$x_v \leftarrow 1$ if $VC[v] =$ **true**, otherwise $x_v \leftarrow 0$, for all $v \in \mathbf{V}$;

Definition 2. For an undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, its directed graph induced by $\prec$, $\vec{\mathbf{G}} = (\mathbf{V}, \vec{\mathbf{E}})$, is constructed by orienting each edge $(v, u) \in \mathbf{E}$ from v to u when $v \prec u$.

The following lemma is straightforwardly obtained from the definition since the relation $\prec$ is transitive.

Lemma 1. Let $\vec{\mathbf{G}} = (\mathbf{V}, \vec{\mathbf{E}})$ be the directed graph induced by $\prec$ for a given undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. Then $\vec{\mathbf{G}}$ is an acyclic graph.

Definition 3. For an acyclic directed graph $\vec{\mathbf{G}} = (\mathbf{V}, \vec{\mathbf{E}})$, $v \in \mathbf{V}$ is called a *source* if $\mathsf{d}^{in}(v) = 0$.

Figure 3 shows an example of the edge orientation, where the number in each vertex v represents $\mathsf{index}(v)$. Figure 3 b) shows the directed graph constructed from

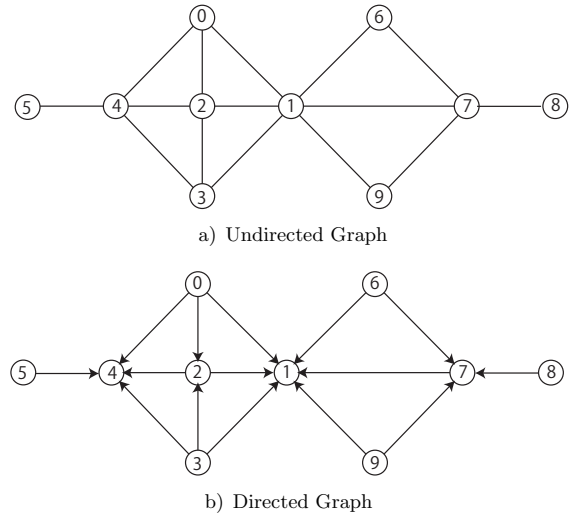


Figure 3. Example of edge orientation

the undirected graph in Figure 3 a). The vertices 0, 3, 5, 6, 8, and 9 are sources of the acyclic graph. In MapReduceMVC, each worker node can compute the relation $\prec$ between the assigned vertex and its neighbors since workers know the index and the degree of all the neighbors.

Arrays of *boolean* variables $VC[v]$ and $Adj[v]$ indicate whether or not v is in the vertex cover and whether or not all the neighbor vertices of v are in the cover, respectively. At the end of the algorithm (Line 28), a vertex cover is constructed by collecting all vertices v such that $VC[v] = \textbf{true}$.

The algorithm is composed with five STEPs and each STEP corresponds to one MapReduce operation. No worker can proceed to the next STEP (MapReduce operation) before all the workers complete the current operation.

At STEP1 (Lines 3 and 4), $VC[v]$ and $Adj[v]$ are initialized to **true** for all the vertices. Note that our algorithm initially adds all the vertices into the vertex cover, then gradually excludes unnecessary vertices in STEP3 and 4.

STEP2 (Lines 6 to 8) sets $VC[v]$ to **false** for each v if v is a source. Each source vertex becomes a trigger to reduce the size of the vertex cover from the initial vertex cover constructed at STEP1.

STEP3 (Lines 11 to 15) and STEP4 (Lines 17 to 25) are iteratively executed in the repeat-until statement. STEP3 updates $Adj[v]$ for all vertices. $Adj[v]$ needs to be updated whenever there is a neighbor $u \in \mathsf{N}(v)$ such that $VC[u]$ was changed in the previous iteration.

STEP4 is composed of two parts. The first part (Lines 17 to 19) is for improving the vertex cover by attempting to decrease its size. The second part (Lines 20 to 25) checks and maintains feasibility of the solution. No solution is feasible when there

exists a pair v and u such that $(v, u) \in \mathbf{E}$, yet $VC[v] = VC[u] =$ **false**. In such case, $VC[v]$ is set to **true** if $u \prec v$, otherwise $VC[u]$ is set to **true** to repair its infeasibility.

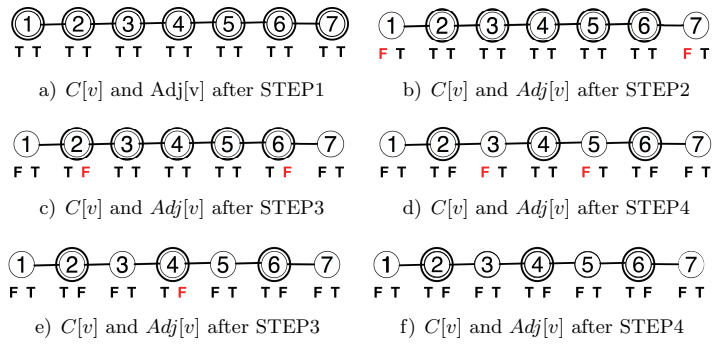


Figure 4. Demonstration of MapReduceMVC for a line graph

Figure 4 shows a demonstration example where we depict the steps of the algorithm when applied to a line graph with seven vertices. In the figure, a pair of letters such as T T, T F, or F T (where T means **true** and F means **false**) below a vertex corresponds to a pair of values of $VC[v]$ and $Adj[v]$. Time proceeds from top to bottom in the figure. The pair of *boolean* values at each vertex is initialized to T T by STEP1 shown in Figure 4 a). Both vertices 1 and 7 are sources in this example, and so become F T at STEP2. Figure 4 c) shows that $Adj[v]$ becomes **false** for each of vertices 2 and 6 since 1 and 7 are removed from the vertex cover at Figure 4 b). In Figure 4 d), each of vertices 3 and 5 is removed from the vertex cover since $Adj[v]$ for 2 and 6 was changed to **false**. In Figure 4 e), $Adj[v]$ for vertex 4 is updated. Finally in (f), we confirmed no modification is performed where each vertex should be either T F or F T. All vertices with T F are included in the vertex cover obtained by the algorithm.

We show now the validity of our algorithm by the following lemma and theorem.

Lemma 2. MapReduceMVC completes its execution after the k^{th} STEP4 if and only if $VC[v] =$ **true** and $Adj[v] =$ **false** or $VC[v] =$ **false** and $Adj[v] =$ **true**, for all $v \in \mathbf{V}$ at the end of the k^{th} STEP3, where k is a natural number.

Proof.

Sufficiency: We assume the following condition holds: For all $v \in \mathbf{V}$, $VC[v] =$ **true** and $Adj[v] =$ **false** or $VC[v] =$ **false** and $Adj[v] =$ **true** after the k^{th} STEP3. Considering synchronous execution of MapReduce operations, $Adj[v]$, for all $v \in \mathbf{V}$, has a correct value after STEP3, that is, $Adj[v] =$ **true** when $VC[u] =$ **true**, for all $u \in \mathsf{N}[v]$, otherwise $Adj[v] =$ **false**. At the kth STEP4, $VC[v]$ can be modified when $VC[v] =$ **true** and $Adj[v] =$ **true** or $VC[v] =$ **false**

and $Adj[v] =$ **false** but not otherwise. Therefore, no modification occurs at the k^{th} STEP4, so the algorithm can break the repeat condition and complete its execution.

Necessity: We assume the algorithm stops its execution and there exists a node v such that $VC[v] =$ **true** and $Adj[v] =$ **true** or $VC[v] =$ **false** and $Adj[v] =$ **false**.

(CASE1: Suppose that $VC[v] = Adj[v] =$ **true**, $\exists v \in \mathbf{V}$.) for all $u \in \mathsf{N}[v]$, $VC[u] =$ **true** since $Adj[v] =$ **true** and also for all $u \in \mathsf{N}[v], Adj[u] =$ **true** because the condition at STEP4 (Line 17) does not hold at the end of the algorithm. By repeating the same consideration, we finally find that for all $v \in \mathbf{V}, VC[v] = Adj[v] =$ **true**. However, according to Lemma 1, at STEP2 there exists at least one source vertex v that lets $VC[v] =$ **false**. Only Line 20 in STEP4 can change $VC[v]$ from **false** to **true**, and it is performed only when there exists a neighbor u such that $VC[v] = VC[u] =$ **false**. Moreover, only one side of both vertices v and u can be changed from **false** to **true** according to the order relation $\prec$. Therefore there should exist at least one vertex v such that $VC[v] =$ **false** even if such situations occur successively on adjacent vertices. This situation contradicts the first assumption.

(CASE2: Suppose that $VC[v] = Adj[v] =$ **false**, $\exists v \in \mathbf{V}$.) There exists at least one vertex $u \in \mathsf{N}(v)$ such that $VC[u] =$ **false** since $Adj[v] =$ **false**. Moreover, $Adj[u] =$ **false** because $VC[v] =$ **false**. This situation holds the condition of the second *if* statement in STEP4. Therefore, the algorithm cannot break the *repeat-until loop* statement. This situation contradicts the first assumption. □

Theorem 1. MapReduceMVC outputs a minimal vertex cover for a given graph.

Proof. From Lemma 2, for all $v \in \mathbf{V}$, $VC[v] =$ **true** and $Adj[v] =$ **false** or $VC[v] =$ **false** and $Adj[v] =$ **true** after the algorithm stops its execution. Therefore, the output of the algorithm should generate a vertex cover since $VC[u] =$ **true** for all $u \in \mathsf{N}(v)$ when $VC[v] =$ **false**. It is also obvious that no proper subset of the vertex cover obtained by the algorithm can be a vertex cover, that is, it shows minimality. □

4 EXPERIMENTAL EVALUATION

We implemented our MapReduce algorithm under Hadoop [3] and evaluated its solution quality and the number of MapReduce rounds. For comparison purposes, we developed a Hadoop program of the maximal matching-based vertex cover algorithm described above.

Test graph data were randomly generated based on two topological characteristics: random graphs and scale-free graphs. A graph is scale-free if its degree distribution follows a power law. Scale-free graphs are known as a model often observed in actual networks [29]. Random graphs with average degree d were generated such that vertex degrees follow the Guassian distribution with average d and variance 1.

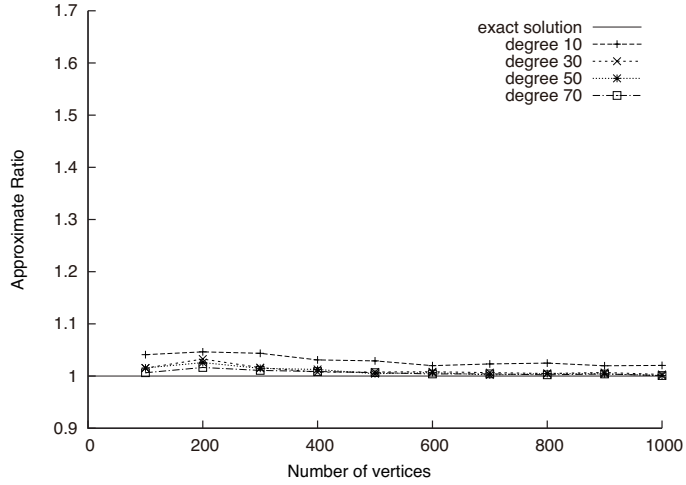


Figure 5. Solution quality vs. graph size for random graphs (our proposal)

4.1 Solution Quality

We first evaluate the solution quality of our MapReduce algorithm by comparing it with its maximal matching-based algorithm. The Gurobi optimizer, a commercial solver, was used to obtain exact solutions but it could not exactly solve problem instances of huge size random graphs within a reasonable time. We therefore eval-

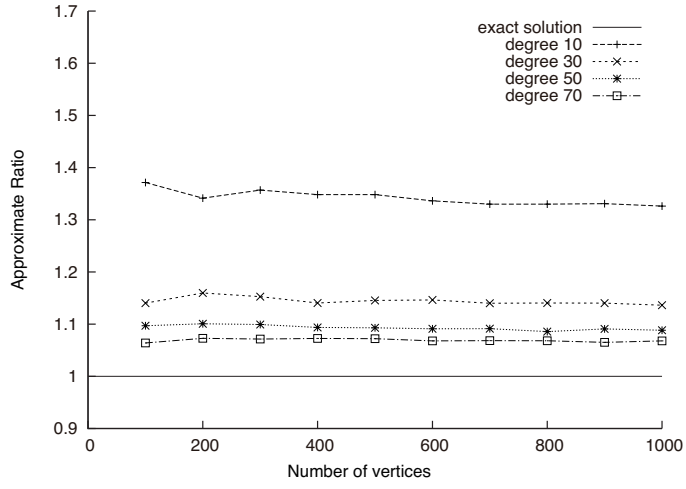


Figure 6. Solution quality vs. graph size for random graphs (maximal matching-based algorithm)

uated, for random graphs of size 100 to 1 000, the quality of approximate solutions by comparing with that of exact solutions, while for random graphs with 1 000 to 10 000 vertices, we just compared both approximate algorithms, our proposal and the maximal matching-based algorithm.

When it comes to scale-free graphs, the Gurobi optimizer was able to solve problem instances of 1 000 000 vertices. This is because the exact algorithm can drastically prune branches of the search tree for the power-law distribution of edge degree in the scale-free graphs. Therefore, for scale-free graphs, we varied the number of vertices up to 1 000 000 to compare with exact solutions.

Figures 5 and 6, respectively, depict the solution quality of our algorithm and that of the matching-based algorithm.

In the figures, the horizontal axis shows the number of vertices and the vertical axis shows the approximate ratio compared to the exact solution. Here we take the quality of the exact solution as 1. For the experiments, we varied the average degree of a vertex from 10 to 70 and prepared ten different instances of each. Therefore, the values in the figures show the average of ten executions. Figures 7 and 8 show results for larger random graphs, where the curves express the size of the obtained minimal vertex covers for each degree. These figures indicate that both algorithms obtained lower quality solutions for smaller degrees of random graphs. That is, loosely-coupled random graphs are harder to solve approximately than are tightly-coupled ones. The results indicate that our algorithm can obtain quite good solutions of less than 5 % approximate ratio, even for degree 10, while the approximate ratio of the matching-based algorithm was more than 30 % in case of the degree 10.

For scale-free graphs, we compare both approximate algorithms with the exact algorithm and depict the quality versus the graph size in Figure 9. The results confirm that our algorithm performs very well also for scale-free graphs compared to the matching-based algorithm.

Through experimental evaluation, we confirmed the effectiveness of our proposed algorithm. From a quality point of view, our algorithm outperformed the maximal matching-based algorithm. The maximal matching-based algorithm progressively constructs a maximal matching, then generates a minimal vertex cover from the maximal matching. The algorithm is based on a greedy policy for the maximality of matching, but not for the minimality of the vertex cover. In contrast, our algorithm focuses on minimizing the size of the vertex cover. Our algorithm generates an initial vertex cover that includes only $|V| - |\text{Sources}|$ vertices, and then removes unnecessary vertices step-by-step in a greedy manner. This direct greedy operation greatly improves the solution quality as compared with the maximal matching-based algorithm.

4.2 Number of Rounds

Instead of measuring real computation time of MapReduce operations, we usually evaluate the number of rounds of MapReduce operations. Actual computation time is much more understandable for performance evaluations, but it strongly depends

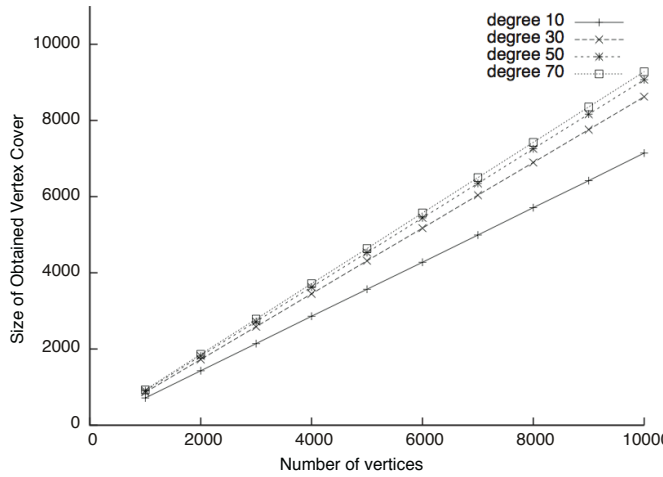


Figure 7. Results for large random graphs (our proposal)

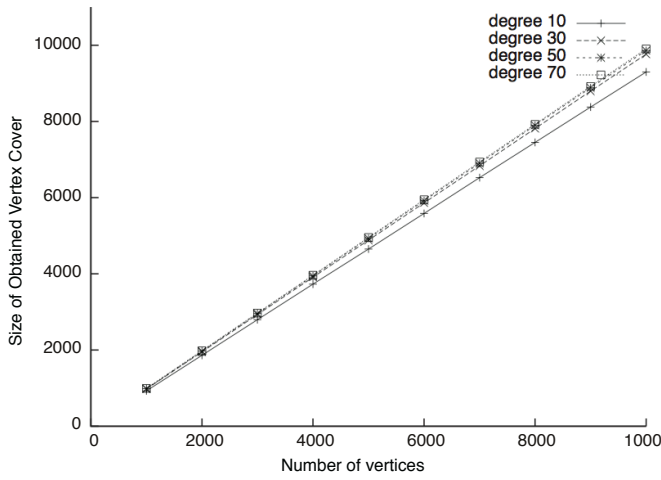


Figure 8. Results for large random graphs (maximal matching-based algorithm)

on conditions of the computing platform, such as the number of processing nodes, cpu spec, memory size, network architecture, and traffic situation. Therefore, the number of rounds becomes the most reliable factor by which to evaluate the computation time of MapReduce programs. In our experiment, we measured the number of rounds of two approximate algorithms. Figures 10 and 11 respectively depict the number of rounds used in our algorithm and the maximal matching-based algorithm for random graphs. Figure 12 compares the number of rounds for scale-free graphs.

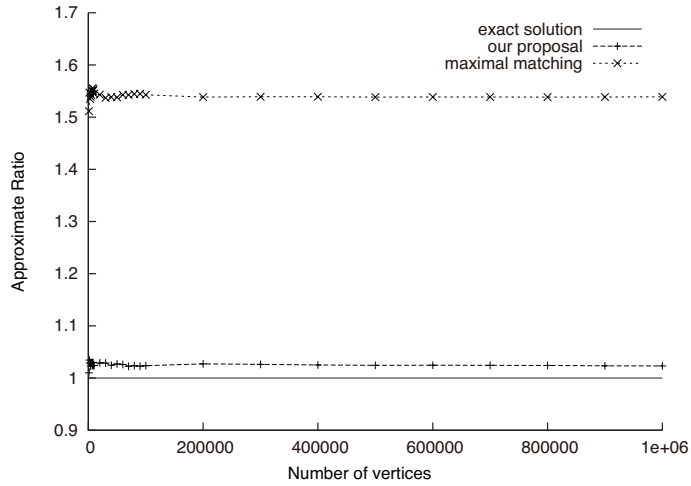


Figure 9. Solution quality vs. graph size for scale-free graphs (our proposal)

The results indicate that our algorithm requires relatively few rounds, compared to the maximal matching-based algorithm. The real computation time is proportional to the number of rounds in the experiment and the curves of the number of rounds are almost constant with respect to the number of vertices. Therefore, our algorithm has good scalability regarding graph size.

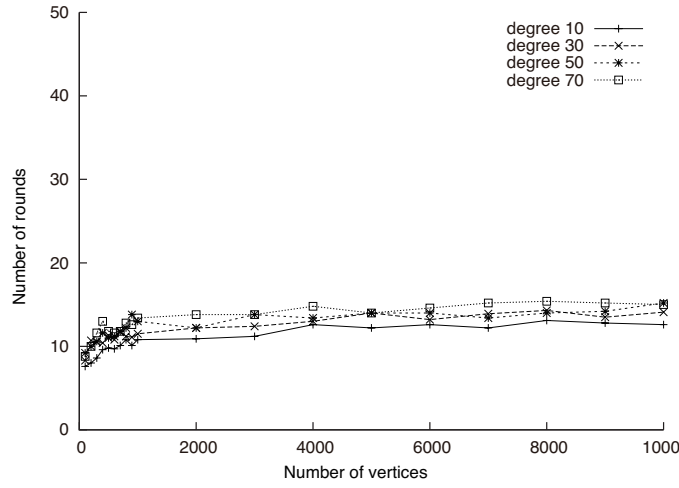


Figure 10. Number of rounds vs. graph size for random graphs (our proposal)

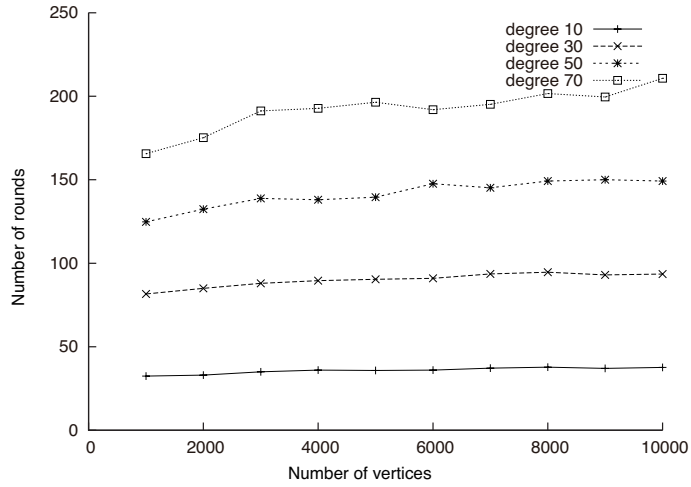


Figure 11. Number of rounds vs. graph size for random graphs (maximal matching-based algorithm)

5 RANDOMIZED MAPREDUCEMVC

MapReduceMVC is not always highly efficient. We consider here a special class of graphs, called *k-flower graphs* for which our original MapReduceMVC may not output good quality of solutions.

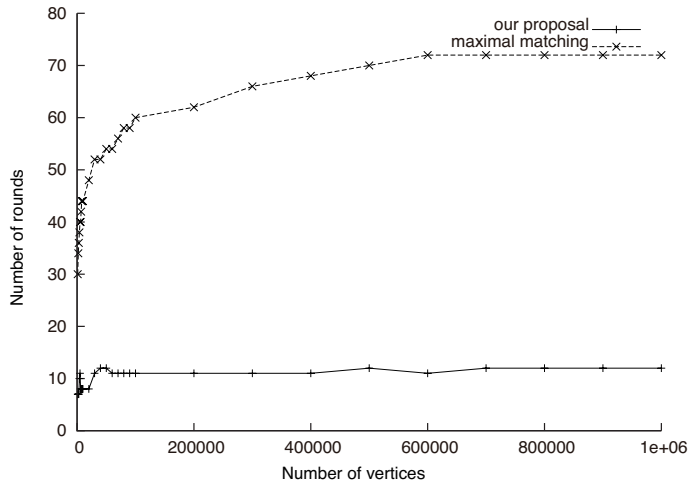


Figure 12. Number of rounds vs. graph size for scale-free graphs

Definition 4. A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is called a k-flower graph when $\mathbf{V} = \{v_0, v_1, v_2, \ldots, v_{3\cdot(k-1)+3}\}$ and $\mathbf{E} = \cup_{i=1}^{k}\{(v_0, v_{3\cdot(i-1)+1}), (v_{3\cdot(i-1)+1}, v_{3\cdot(i-1)+2}), (v_{3\cdot(i-1)+2}, v_{3\cdot(i-1)+3}), (v_{3\cdot(i-1)+3}, v_0)\}$.

Definition 5. For k-flower graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, the vertex v_0 is called *center vertex* and subgraph $\mathbf{G}(i) = (\mathbf{V}(i), \mathbf{E}(i)), i = 1, 2, \ldots, k$, is called a petal of k-flower graph $\mathbf{G}$ where $\mathbf{V}(i) = \{v_0, v_{3\cdot(i-1)+1}, v_{3\cdot(i-1)+2}, v_{3\cdot(i-1)+3}\}$, $\mathbf{E}(i) = \{(v_0, v_{3\cdot(i-1)+1}), (v_{3\cdot(i-1)+1}, v_{3\cdot(i-1)+2}), (v_{3\cdot(i-1)+2}, v_{3\cdot(i-1)+3}), (v_{3\cdot(i-1)+3}, v_0)\}$.

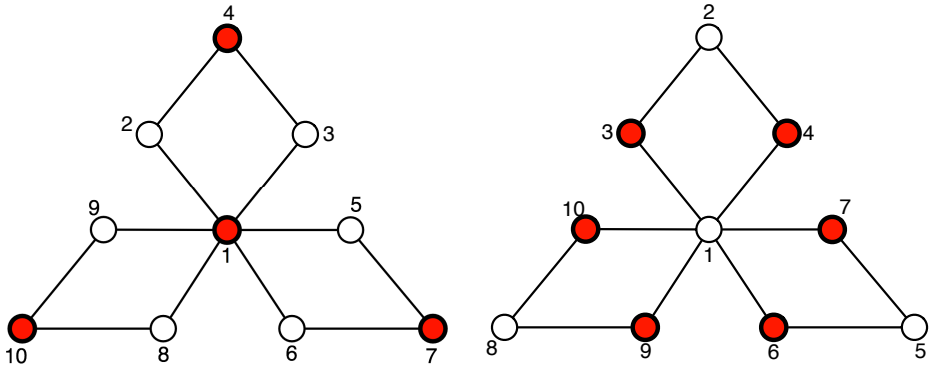
Figure 13 depicts 3-flower graph and its exact and the worst case vertex covers, where the vertices colored in red denote the vertices in the obtained vertex cover and the number beside a vertex shows its index. Figures 13 a) and 13 b) represent the exact solution and the worst case solution obtained by our MapReduce algorithm, respectively. From the definition, we can easily prove that the size of the optimal solution and the worst case solution for k-flower graphs are

$$|\mathbf{VC}^{\mathsf{best}}| = k + 1, \tag{4}$$

$$|\mathbf{VC}^{\mathsf{worst}}| = 2 \cdot k + 1, \tag{5}$$

respectively. Therefore, the approximation ratio for k-flower graphs is

$$\frac{|\mathbf{VC}^{\mathsf{worst}}|}{|\mathbf{VC}^{\mathsf{best}}|} = \frac{2 \cdot k + 1}{k + 1} \approx 2. \tag{6}$$



a) Optimal solution b) Worst case solution

Figure 13. Solution example for 3-flower graphs

Randomized algorithms allow us to avoid the worst case solution, where we utilize only randomized index function in our MapReduceMVC. Our algorithm requires the construction of a directed graph for the input graph by the edge orientation

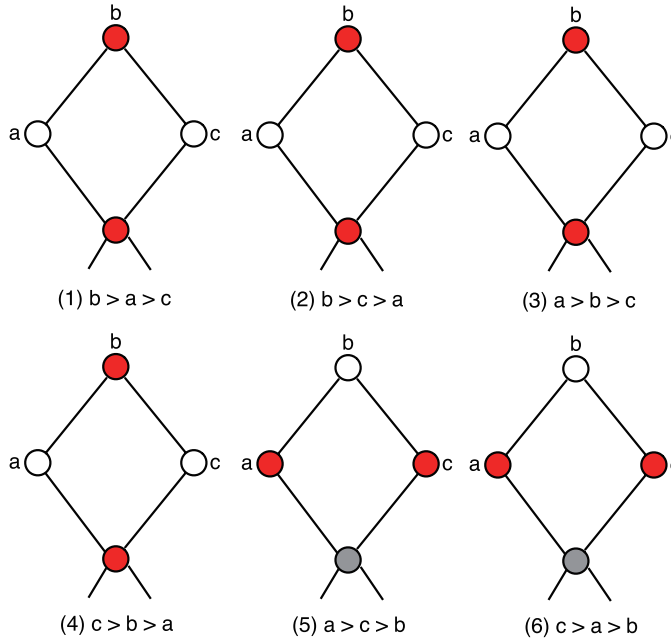


Figure 14. All patterns of randomized index

based on the lexicographical order on $(\prec, \mathsf{index})$. In k-flower graphs, the edge orientation depends on the index values of the terminal vertices since all the vertices except for the center vertex have degree 2. This is the reason why it is hard for our original MapReduceMVC to solve k-flower graphs.

Let us calculate the expected size of minimal vertex covers to be obtained by the randomized MapReduceMVC for k-flower graphs. Figure 14 shows all the patterns of indexing for one petal of a k-flower graph, where a, b, and c express the index value for the corresponding vertex. The vertex with the smallest index number out of a, b, and c should be a source which can not be in the final vertex cover. The vertices colored in red are in the vertex cover. From the figures, the expected value of the number of vertices in $\mathbf{VC}$ per petal except for the vertex v_0, $E(|\mathbf{VC}(petal)|$ is

$$E(|\mathbf{VC}(petal)|) = 1 \cdot \frac{2}{3} + 2 \cdot \frac{1}{3} = \frac{4}{3}. \tag{7}$$

Since v_0 is in $\mathbf{VC}$ only when the indexing is either $a > c > b$ or $c > a > b$ for all the petals, the expected value for the center vertex in $\mathbf{VC}$, $E(|\mathbf{VC}(v_0)|$ is

$$E(|\mathbf{VC}(v_0)|) = 1 - \left(\frac{1}{3}\right)^k. \tag{8}$$

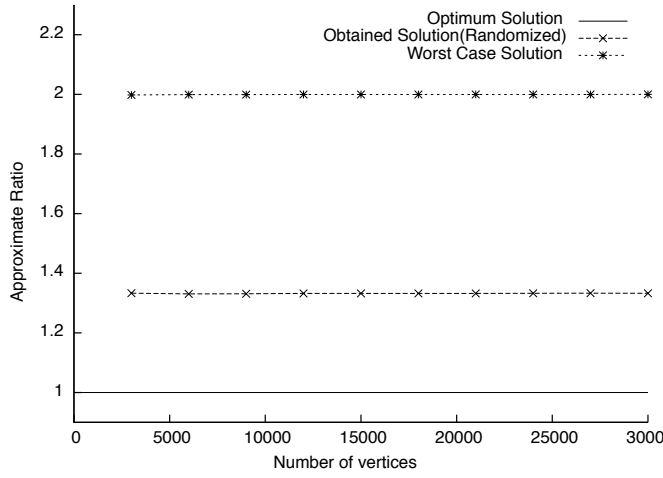


Figure 15. Solution quality of randomized MapReduceMVC for k-flower graphs

Therefore, the expected value of $|\mathbf{VC}|$ is

$$E(|\mathbf{VC}|) = k \cdot E(|VC(petal)|) \tag{9}$$

$$= \frac{4}{3} \cdot k + 1 - \left(\frac{1}{3}\right)^k. \tag{10}$$

The expected approximation ratio is as follows:

$$E\left(\frac{|\mathbf{VC}|}{|\mathbf{VC}^{\mathsf{best}}|}\right) = \frac{\frac{4}{3} \cdot k + 1 - \left(\frac{1}{3}\right)^k}{k+1} \approx \frac{4}{3}. \tag{11}$$

To confirm the validity of the expected approximate ratio, we performed an experiment. Figure 15 shows the average curves of the approximate ratio in which we solved MVC for k-flower graphs 20 times by the Randomized MapReduceMVC with varying the number of vertices. From the figures, our randomized MapReduceMVC could achieve the same approximate ratio as the one we calculated in (11). Our randomized algorithm can be easily implemented since we just need to introduce a randomized index function to the original one.

Theorem 2. RandomizedMapReduceMVC outputs a minimal vertex cover with the expected approximate ratio 4/3 for k-flower graphs.

6 CONCLUSION

We proposed a MapReduce algorithm for the minimum vertex cover problem and proved its validity. We performed an experimental evaluation of our proposal and

measured the quality of solutions and the number of rounds of MapReduce operations. We observed that our algorithm could generate a reasonable quality of approximate solutions compared to the exact algorithm for random graphs with 100 to 1 000 vertices and scale-free graphs with 1 000 to 1 000 000 vertices. We also compared our algorithm with the well-known maximal matching-based minimum vertex cover algorithm, and our algorithm outperformed it not only in terms of solution quality but also in terms of computation time.

Finally we introduced a class of graphs, k-flower graphs, which it is hard for our algorithm to solve, and we have proposed a randomized version of MapReduceMVC. Just by using the randomized index function, our algorithm can obtain the expected approximate ratio 4/3 for k-flower graphs.

REFERENCES

[1] Dean, J.—Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04), 2004, pp. 137–150.

[2] Karloff, H.—Suri, S.—Vassilvitskii, S.: A Model of Computation for MapReduce. Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, 2010, pp. 938–948, doi: 10.1137/1.9781611973075.76.

[3] Holmes, A.: Hadoop in Practice. Manning Publications Co., Greenwich, 2012.

[4] Guhther, N. J.—Puglia, P.—Tomasette, K.: Hadoop Superlinear Scalability. Communications of the ACM, Vol. 58, 2015, No. 4, pp. 46–55, doi: 10.1145/2719919.

[5] Apache Spark. `https://spark.apache.org/`.

[6] Ketu, S.—Mishra, P. K.—Agarwal, S.: Performance Analysis of Distributed Computing Frameworks for Big Data Analytics: Hadoop vs. Spark. Computación y Sistemas, Vol. 24, 2020, No. 2, pp. 669–689, doi: 10.13053/cys-24-2-3401.

[7] Mostafaeipour, A.—Jahangard Rafsanjani, A.—Ahmadi, M.—Arockia Dhanraj, J.: Investigating the Performance of Hadoop and Spark Platforms on Machine Learning Algorithms. The Journal of Supercomputing, Vol. 77, 2021, pp. 1273–1300, doi: 10.1007/s11227-020-03328-5.

[8] Lin, J.—Schatz, M.: Design Patterns for Efficient Graph Algorithms in MapReduce. Proceedings of the Eighth Workshop on Mining and Learning with Graphs. 2010, pp. 78–85, doi: 10.1145/1830252.1830263.

[9] Warashina, T.—Aoyama, K.—Sawada, H.—Hattori, T.: Efficient K-Nearest Neighbor Graph Construction Using MapReduce for Large-Scale Data Sets. IEICE Transactions on Information and Systems, Vol. E97.D, 2014, No. 12, pp. 3142–3154, doi: 10.1587/transinf.2014edp7108.

[10] Devi, N. S.—Mane, A. C.—Mishra, S.: Computational Complexity of Minimum P4 Vertex Cover Problem for Regular and K1, 4-Free Graphs. Discrete Applied Mathematics, Vol. 184, 2015, pp. 114–121, doi: 10.1016/j.dam.2014.10.033.

[11] Guruswami, V.—Sachdeva, S.—Saket, R.: Inapproximability of Minimum Vertex Cover on k-Uniform k-Partite Hypergraphs. SIAM Journal on Discrete Mathematics, Vol. 29, 2013, No. 1, pp. 36–58, doi: 10.1137/130919416.

[12] Lattanzi, S.—Moseley, B.—Suri, S.—Vassilvitskii, S.: Filtering: A Method for Solving Graph Problems in MapReduce. Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11), 2011, pp. 85–94, doi: 10.1145/1989493.1989505.

[13] Plimpton, S. J.—Devine, K. D.: MapReduce in MPI for Large-Scale Graph Algorithms. Parallel Computing, Vol. 37, 2011, No. 9, pp. 610–632, doi: 10.1016/j.parco.2011.02.004.

[14] 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. Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10), 2010, pp. 135–146, doi: 10.1145/1807167.1807184.

[15] Kambatla, K.—Rapolu, N.—Jagannathan, S.—Grama, A.: Asynchronous Algorithms in MapReduce. Proceedings of 2010 IEEE International Conference of Cluster Computing, Heraklion, Greece, 2010, pp. 245–254, doi: 10.1109/cluster.2010.30.

[16] Bellettini, C.—Camilli, M.—Capra, L.—Monga, M.: State Space Exploration of RT Systems in the Cloud. Cornell University Library, arXiv:1203.6806 [cs.SE], 2012, 6 pp.

[17] Taylor, R. C.: An Overview of the Hadoop/MapReduce/HBase Framework and Its Current Applications in Bioinformatics. BMC Bioinformatics, Vol. 11, 2010, No. S1, doi: 10.1186/1471-2105-11-s12-s1.

[18] Liang, F.—Lu, X.: Accelerating Iterative Big Data Computing Through MPI. Journal of Computer Science and Technology, Vol. 30, 2015, No. 2, pp. 283–294, doi: 10.1007/s11390-015-1522-5.

[19] Yu, W. K.—Wang, Y. D.—Que, X. Y.—Xu, C.: Virtual Shuffling for Efficient Data Movement in MapReduce. IEEE Transactions on Computers, Vol. 64, 2015, No. 2, pp. 556–568, doi: 10.1109/tc.2013.216.

[20] Kang, U.—Tsourakakis, C. E.—Faloutsos, C.: PEGASUS: Mining Peta-Scale Graphs. Knowledge and Information Systems, Vol. 27, 2011, No. 2, pp. 303–325, doi: 10.1007/s10115-010-0305-0.

[21] Wu, B.—Yang, S.—Zhao, H.—Wang, B.: A Distributed Algorithm to Enumerate All Maximal Cliques in MapReduce. Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology, 2009, pp. 45–51, doi: 10.1109/fcst.2009.30.

[22] Chierichetti, F.—Kumar, R.—Tomkins, A.: Max-Cover in Map-Reduce. Proceedings of the 19th International Conference on World Wide Web (WWW '10), 2010, pp. 231–240, doi: 10.1145/1772690.1772715.

[23] Halim, F.—Yap, R. H. C.—Wu, Y.: A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. Proceedings of the 2011 31st International Conference on Distributed Computing Systems, 2011, pp. 192–202, doi: 10.1109/icdcs.2011.62.

[24] Kumar, P.—Singh, A. K.: MapReduce Algorithm for Single Source Shortest Path Problem. International Journal of Computer Network and Information Security (IJCNIS), Vol. 12, 2020, No. 3, pp. 11–21, doi: 10.5815/ijcnis.2020.03.02.

[25] Karp, R. M.: Reducibility Among Combinatorial Problems. In: Miller, R. E., Thatcher, J. W., Bohlinger, J. D. (Eds.): Complexity of Computer Computations. Springer, Boston, MA, The IBM Research Symposia Series, 1972, pp. 85–103, doi: 10.1007/978-1-4684-2001-2_9.

[26] Karakostas, G.: A Better Approximation Ratio for the Vertex Cover Problem. In: Caires, L., Italiano, G. F., Monteiro, L., Palamidessi, C., Yung, M. (Eds.): Automata, Languages and Programming (ICALP 2005). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3580, 2005, pp. 1043–1050, doi: 10.1007/11523468_84.

[27] Bar-Yehuda, R.—Even, S.: A Local-Ratio Theorem for Approximating the Weighted Vertex Cover Problem. In: Ausiello, G., Lucertini, M. (Eds.): Annals of Discrete Mathematics 25. North-Holland Mathematics Studies, Vol. 109, 1985, pp. 27–45, doi: 10.1016/s0304-0208(08)73101-3.

[28] Kinjo, D.—Nakamura, M.: A MapReduce Algorithm for Minimum Vertex Cover Problem. Proceedings of International Technical Conference on Circuits and Systems, Computers and Communications, 2013, pp. 505–508.

[29] Li, L.—Doyle, J. C.—Willinger, W.—Alderson, D.: Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications. Internet Mathematics, Vol. 2, 2005, No. 4, pp. 431–523, doi: 10.1080/15427951.2005.10129111.

[30] Bondy, J. A.—Murty, U. S. R.: Graph Theory with Applications. Elsevier Science Publishing, 1976.

[31] Diestel, R.: Graph Theory. Second Edition. Springer-Verlag, New York, 2000.

[32] Harvey, N. J. A.—Liaw, C.—Liu, P.: Greedy and Local Ratio Algorithms in the MapReduce Model. Proceedings of the 30$^{\text{th}}$ ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '18), 2018, pp. 43–52, doi: 10.1145/3210377.3210386.

Morikazu Nakamura received his B.E. and M.E. degrees from the University of the Ryukyus in 1989 and 1991, respectively, and D.E. degree from Osaka University in 1996. He is currently Professor in the area of computer science and intelligent systems, Faculty of Engineering, University of the Ryukyus, Japan. His research interests include theory and applications on mathematical systems. He is a member of IEICE and IEEE.

Daiki Kinjo received his B.E. and M.E. degrees in information engineering from the University of the Ryujyus in 2012 and 2014, respectively. He is currently Engineer in KLab Inc. His research interests include distributed computing, network computing, and data analysis.

Takeo Yoshida received his B.E. and M.E. degrees in electrical engineering from the Nagaoka University of Technology and the D.E. degree in electrical engineering from the Tokyo Metropolitan University in 1991, 1993 and 1997, respectively. He is currently Assistant Professor in the Department of Engineering, University of the Ryukyus, Japan. His research interests include dependable computing, VLSI design, and graph theory. He is a member of IEEE and IPSJ.