Computing (2015) 97:1101–1120
DOI 10.1007/s00607-015-0446-9

Distributed discovery of frequent subgraphs of a network using MapReduce

Saeed Shahrivari · Saeed Jalili

Received: 27 July 2014 / Accepted: 17 February 2015 / Published online: 26 February 2015

Abstract Discovery of frequent subgraphs of a network is a challenging and time-consuming process. Several heuristics and improvements have been proposed before. However, when the size of subgraphs or the size of network is big, the process cannot be done in feasible time on a single machine. One of the promising solutions is using the processing power of available parallel and distributed systems. In this paper, we present a distributed solution for discovery of frequent subgraphs of a network using the MapReduce framework. The solution is named *MRSUB* and is developed to run over the Hadoop framework. MRSUB uses a novel and load-balanced parallel subgraph enumeration algorithm and fits it into the MapReduce framework. Also, a fast subgraph isomorphism detection heuristic is used which accelerates the whole process further. We executed MRSUB on a private cloud infrastructure with 40 machines and performed several experiments with different networks. Experimental results show that MRSUB scales well and offers an effective solution for discovery of frequent subgraphs of networks which are not possible on a single machine in feasible time.

Keywords Frequent subgraph discovery · Distributed graph algorithms · MapReduce

Mathematics Subject Classification 68W15 · 05C85 · 68R10 · 05C60

S. Shahrivari · S. Jalili (✉)
Computer Engineering Department, Tarbiat Modares University (TMU), Tehran, Iran
e-mail: sjalili@modares.ac.ir

S. Shahrivari
e-mail: s.shahrivari@modares.ac.ir

1 Introduction

Discovering frequent subgraphs of a given network, has various applications in the domain of complex networks analysis. The problem usually involves finding frequent small subgraphs of a given input network, i.e. subgraphs with 3–10 nodes. For example, in biological networks, finding frequent subgraphs leads to discovery of network motifs that may reflect functional and nonfunctional properties [1]. Another important application is in the domain of social networks. Finding frequent subgraphs of a social network can help detection of stable and unstable communities, link prediction, and analysis of frequent patterns in the network [2]. There are similar applications in other complex networks like communication, citation, metabolic, epidemiology, and chemical networks.

However, discovering the frequent subgraphs of a network is a hard problem. The only available exact solution to this problem is to enumerate all subgraphs and count the number of occurrences of each isomorphism class [3]. This approach is very time consuming because the number of subgraphs of a network grows exponentially with size of the subgraphs or size of the input network. Two of the best known tools that can perform exact discovery are Kavosh [4] and FANMOD [3]. However, when the size of subgraph is big or when the input network is large, these tools can easily take days and even months to finish their execution. Available tools are also limited to small input networks, and they crash due to memory overflow when finding bigger subgraphs. Other inexact solutions like sampling-based approaches are also available, too. For example, FANMOD provides a fast sampling based solution in addition to exact solution [3]. Inexact solutions tend to be much faster but, exact solutions have proven to have significant superiority considering the quality and precision of the results.

A practical solution for overcoming the time consuming nature of frequent subgraph discovery, is to use the potential processing power of high performance computing (HPC) systems like cluster and cloud computing platforms. A public cloud computing platform like Amazon EC2 can be easily used to do the work that is done in days and weeks on a commodity personal computer, in hours using hundreds of virtual machine instances running on the cloud platform. We present a new solution to discovery of frequent subgraphs of a network based on the MapReduce framework, named *MRSUB*. MRSUB runs on the Hadoop framework and can efficiently use the power of tens and hundreds of machines for fast discovery of frequent subgraph. Besides using the power of the MapReduce framework and cloud computing, MRSUB uses an efficient heuristic for subgraph isomorphism detection which accelerates execution speed further.

We evaluated MRSUB on a private cloud platform consisting of 40 virtual machine instances. During the experiments, we used several different real-world complex network datasets with various sizes. The experimental results show that MRSUB provides a scalable and high performance distributed solution to finding frequent subgraphs of the input networks. The solution also scales nearly linear by using more machines. Using our test-bed, MRSUB can find frequent subgraphs of sample networks in several hours while serial tools need several days or even weeks to do the job.

The rest of this paper is organized as follows. In Sect. 2, we discuss the available related work and their advantages and shortcomings. In the Sect. 3, we give some definitions and materials that are necessary for explaining our solution. In Sect. 4, we discuss our solution, MRSUB, in details. In Sect. 5, we report the results of our experiments dealing with performance and scalability of MRSUB, and also we compare MRSUB to available solutions. And finally, we give a conclusion and discuss the opportunities for further work.

2 Related work

Considering algorithmic properties, related work can be categorized into three major classes:

(i) All-subgraph enumeration problem[5]: in which all subgraphs of size k of a given graph must be enumerated. Usually the goal is to enumerate non-isomorphic subgraphs of a network and then frequencies of each isomorphism class are extracted [3,4,6–9].
(ii) Large graph mining [10]: algorithms that try to find frequent subgraphs in a large input graph.
(iii) Graph dataset mining [11]: the goal is to discover frequent subgraphs that occur frequently across a set of input graphs. The most successful solutions in this section are based on the well-known Apriori method for finding frequent associative rules. AGM [11], FSG [12], and gSpan [13] are some good solutions for graph dataset mining.

MRSUB is a MapReduce-based solution that finds frequent subgraphs of a given network (graph) via all-subgraph enumeration. Hence, the third category is less related. We skip discussing it further and discuss the first and second classes in more detail.

The most notable efforts for all-subgraph enumeration problem are done in network motif finding problem. The best known exact approach for finding network motifs is via all-subgraph enumeration and counting [4]. The most notable work in this sector are Kavosh [4], ESU [2,3], FPF [14], and mfinder [1]. Considering performance and speed, Kavosh and ESU are superior. However, experimental results show no significant superiority between Kavosh and ESU [4]. Ribeiro et al. [5] have also proposed a parallel MPI-based solution for motif finding that can be run on cluster computing systems. RANGI is also a high performance motif finding algorithm that is able to use multicore CPUs but it is designed to find colored motifs [7].

There exist some inexact approaches for finding frequent subgraphs. GREW [15] and SUBDUE [16] are two greedy heuristic approaches that sacrifice soundness and completeness of results in order to reach a better speed. Wernicke has also proposed a randomized enumeration algorithm [2]. The main idea is to skip over some subgraphs during enumeration [2]. However, experiments show that skipping more than 80 percent of subgraphs can lead to biased sampling in most cases and hence, the output may be inaccurate [2].

To the best of our knowledge, no generic MapReduce-based solution is available for all-subgraph enumeration and frequent subgraph discovery. Afrati et al. [17]

have proposed some methods for enumerating subgraph instances using MapReduce. They have shown how every instance of simple specific subgraphs like triangles and rectangles can be enumerated using MapReduce. In an independent work, Cohen [18] has also proposed algorithms for enumerating triangles and rectangles using MapReduce. There are other solutions for enumerating triangles using MapReduce [19,20]. Pagh and Tsourakakis [21] have also proposed a method for colorful triangle counting using MapReduce. SAHAD [22] is a recent framework based on Hadoop that can be used for querying and enumerating subgraph instances using MapReduce. PEGASUS is another MapReduce-based graph mining framework that can be used for tasks like mining radii of large graphs but it is not capable of all-subgraph enumeration [10,23]. Considering the discussed work, all the available MapReduce-based solutions are developed for enumerating a single or special kind of subgraphs like triangles and developing MapReduce-based algorithms for enumeration of general forms of subgraphs is an open problem [18,22].

3 Preliminaries

A network (graph) is a collection of points that are connected by some links. The points of a network are called vertices and the links are called edges. We use G to denote a network, $V(G)$ is used to present its vertices, and $E(G)$ is used to present the edges. Vertices and edges of a network can be assigned by labels, weights, or colors. However, we assume networks to be directed, simple and unweighted. In other words, we assume that just the vertices take labels, and the edges are directed, do not have weights, and also there is at most one edge between two vertices in each direction.

For a vertex set $V' \subseteq V$ its open neighborhood $N(V')$ is the set of all vertices from $V - V'$ which are adjacent to at least one vertex of V'. For a vertex $v \in V - V'$ its exclusive neighborhood with respect to V' denoted by $N_{excl}(v, V')$ is the set of all vertices neighboring v that do not belong to $V' \cup N(V')$.

The graph H is a subgraph of G, if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. An induced subgraph of G on the vertices set N that is denoted by $G[N]$, is a subgraph of G with N as the vertex set containing all edges between vertices of $N \subseteq V$ that are in $E(G)$. When we say we are enumerating subgraphs of size k of a graph like G we mean enumerating induced subgraphs of G. Two subgraphs G_1 and G_2 are isomorphic if and only if there is a one to one correspondence between their vertices, and there is an edge between two vertices of G_1 if and only if there is an edge between the corresponding vertices in G_2. Actually, there is no polynomial time algorithm for graph isomorphism problem yet [24].

3.1 MapReduce and Hadoop

MRSUB is a MapReduce-based solution that runs over the Hadoop framework. MapReduce is a framework that allows declaring programs that can be easily run over hundreds and even thousands of machines [25]. MapReduce delivers three key

features simultaneously: easy programming logic, fault tolerant execution, and near linear scalability [26]. Every MapReduce program compromises a *Map* step and a *Reduce* step. The map step takes a set of key/value pairs and converts it into another set of key/value pairs. The reduce step takes the output of the map step and combines all the values that are associated to an identical key and then delivers the combined values to a reduce function. Hadoop is the most widely used open-source implementation of the MapReduce programming model. Hadoop is written in Java and it is capable of running across thousands of machines [27]. Further information about MapReduce and Hadoop can be found in [25–27].

4 The proposed solution: MRSUB

The most common approach for finding frequent subgraphs of a network is a two phase approach. In the first phase called *enumeration*, all subgraphs of size k are enumerated. Then in the second phase called *classification*, isomorphic class of each subgraph is determined and the number of members of each class is counted. The structures of the isomorphism classes that have the highest frequencies denote the most frequent subgraphs of the input network. Our solution uses this approach, too. MRSUB is a MapReduce based solution, and hence it is defined as a *map* function and a *reduce* function. In MRSUB, enumeration and part of classification are done in the map function, and the reduce function just deals with classification.

However, available solutions that use this approach have some shortcomings. Available subgraph enumeration algorithms are sequential and hence they are not suitable for distributed execution. The classification methods also use a centralized memory-based lookup table for storing isomorphic classes and hence they are not scalable. We have eliminated the shortcomings and improve the overall performance.

All of the available algorithms for subgraph enumeration, like ESU and Kavosh, enumerate subgraphs of each vertex one by one. The enumeration of subgraphs of each vertex is independent and hence, the enumeration phase can be easily parallelized. However, the generated parallel tasks are unbalanced because there is a huge variance in the number of subgraphs rooted from each vertex. To overcome this problem, we have proposed an algorithm which can enumerate subgraphs of each edge. Enumerating subgraphs using edges results more fine-grained tasks. In addition, the tasks are more load-balanced, too. In the experiments section, we show that how coarse-grained tasks provided by vertex-based approach can break load balancing and why our approach is better in more details. Also, we have used the natural ability of MapReduce framework to count, for the classification phase. Therefore, we provide an effective scalable classification mechanism. Next, we discuss MRSUB as the map and reduce steps.

Algorithm 1: The Map function of MRSUB

let *e(v,w)* be the input edge, *k*: $1<k\leq|V(G)|$ be the size of subgraphs to be enumerated, and *G* be the input network

Function *map*(*e(v,w), k*)

1. **If** *G* also contains *e(w,v)* **and** *v*>*w* **then return.** //to enumerate bidirectional edges once
2. **let** *Stack* be a stack of tuples
3. **if** *v*>*w* **then** swap *v* and *w* //to guarantee that *v* is smaller than *w*
4. $V_{Extension} \leftarrow \{u \in N(\{v\}) : u > w\} \bigcup \{u \in N_{excl}(w, \{v\}) : u > v\} - \{v, w\}$
5. push new tuple($\{v,w\}, V_{Extension}$, *v*) into *Stack*
6. **while** *Stack* is not empty **do**:
7. *top* ← pop the tuple on top of the stack
8. **if** |*top*[0]|=*k* **then** //top[0] is the first item of the tuple
9. *H*=*G*[*top*[0]] //*G*[*top*[0]] is the found subgraph
10. *label* ← *Canonical_Label*(*H*)
11. **emit**(*label*,1) //emits the found label to reduce phase
12. **endif**
13. **while** *top*[1] $\neq\emptyset$ **do:** //top[1] is the extension set
14. remove a vertex *x* from *top*[1]
15. $V'_{Extension} \leftarrow top[1] \bigcup \{u \in N_{excl}(x, V_{Subgraph}) : u > top[2]\}$ //top[2] is the root
16. push new tuple($top[0] \bigcup \{x\}, V'_{Extension}, top[2]$) into *Stack*
17. **endwhile**
18. **endwhile**

4.1 The map step of MRSUB

We assume that the input network is given as a file containing list of the network edges. The map function is applied on every edge. During the map function, all of the subgraphs of size *k* that contain the given edge are enumerated and for each found subgraph a canonical label is generated and the label is emitted to the reduce phase. The fastest mechanism available for polymorphism detection is *canonical labeling.* A canonical label is like a signature that demonstrates isomorphic class of a subgraph. Canonical labels are usually generated by concatenation of rows of the adjacency matrix of the canonical form of the input subgraph. Canonical labels for two subgraphs that belong to an identical isomorphism class are guaranteed to be the same and vice versa. There are several algorithms for generating canonical labels like *nauty* [28] and *bliss* [29] but the fastest one is nauty. The details of the map function are given in Algorithm 1.

Our enumeration algorithm resembles to state space exploration. Each state is denoted by a tuple with three elements. The first element is a set representing the subgraph associated with the state. The second element is the *extension set* which is the set of vertices that can be added to the subgraph and increase its size by one, and the last element of tuple is called *root* which is the smallest vertex of the subgraph. The enumeration procedure begins in line 2. We make an initial state in line 5 that denotes a subgraph of size 2 (the given edge) and the candidate vertices that can be added to that edge to make a subgraph of size 3. Then, algorithm proceeds by expanding the initial state and finds all subgraphs of size *k* (the *while* loop in lines 6–18). In each

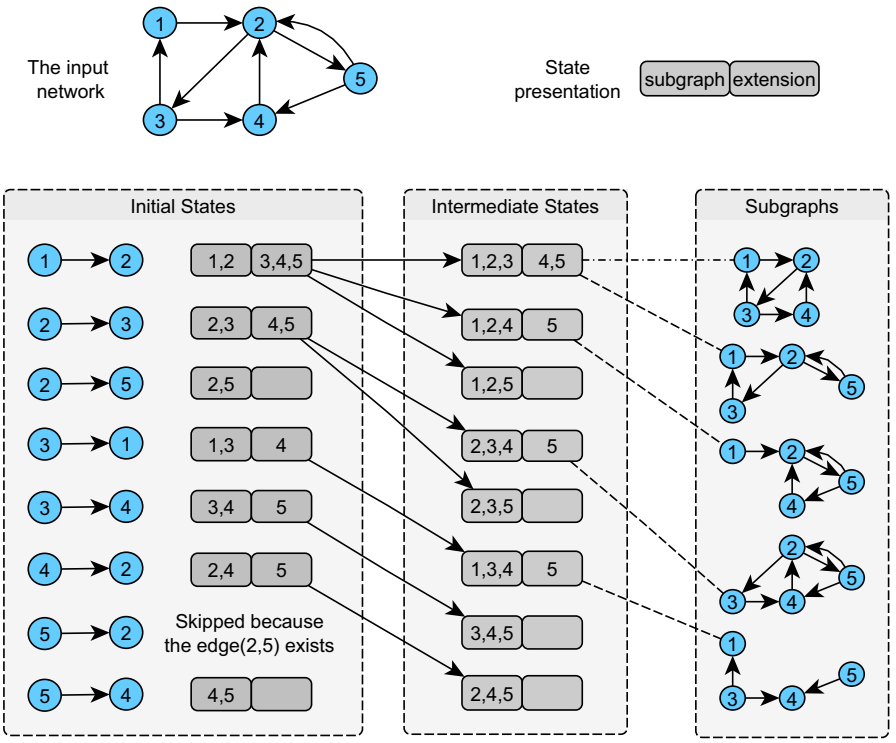

Fig. 1 The procedure of enumeration for a sample network

iteration of the *while* loop (lines 6–18), a state is picked from the stack. The stack initially contains just the initial state which was pushed to stack in line 5. When a state is picked from the stack there are two possibilities: (i) the size of the subgraph of the state is equal to k, (ii) the size of the subgraph of the state is smaller than k.

If a subgraph of size k is found (lines 8–12), its canonical label is computed by calling the *Canonical_Label* function, and the generated label is given to the reduce step by calling *emit*. Note that for canonical labeling we use the nauty algorithm which is discussed in details in [28]. This function gets a subgraph as the input and returns a string representing its canonical label. On the other hand, if the size of the subgraph of the picked state is smaller than k, the subgraph should be enlarged to reach a larger subgraph. This is done by expanding the picked state using its candidate vertices in extension set (lines 13–17), and the newly generated states are pushed to stack (line 16). This procedure is repeated until the stack becomes empty, i.e. all of the subgraphs of the input edge are enumerated. The procedure of enumeration of subgraphs of size 4 for a sample network is given in Fig. 1.

4.2 The reduce step of MRSUB

After the execution of map phase, a list of ordered labels is given to the reduce step. The reduce function is a simple summer that sums all the values (subgraph instances)

associated to each key (canonical labels) and outputs a label and its overall frequency as the output. The reduce functions is given in Algorithm 2. After the execution of reduce function the frequency of each canonical label, i.e. isomorphic class, is available and the most frequent subgraphs can be easily extracted by sorting canonical labels considering their frequencies.

4.3 Accelerating subgraph classification using ordered labeling

The biggest problem of canonical labeling is its exponential worst case time complexity. For every subgraph a canonical label must be generated. Using an exponential-time algorithm makes this phase very time consuming because the number of subgraphs grows exponentially. Instead, we have proposed a novel heuristic, called *ordered labeling*, in addition to the well-known canonical labeling algorithm, *nauty*. Ordered labeling is polynomial-time and is much faster than canonical labeling. Therefore, it can accelerate the classification phase by reducing calls to canonical labeling.

Algorithm 2: The reduce function of MRSUB

let *label* be the input key denoting a canonical label, and *values* be a list of integers

Function *reduce*(*label, values*)

1. *sum* ← 0
2. **for each** *val* **in** *values* **do:**
   sum ← *sum* + 1
3. **endfor**
4. **emit**(*label*,*sum*) //writes the canonical label and its total frequency to output

The basic idea behind ordered labeling is the well-known lemma in graph theory: "Two graphs G_1 and G_2 are isomorphic if and only if their adjacency matrices are the same" [30]. First, ordered labeling sorts vertices by their degrees. The degree of a vertex in a directed graph is the number of its neighbor vertices. Then, each vertex is labeled according to its rank in the sorted list. At last, a new adjacency matrix is built considering the new labels and the concatenation of adjacency matrix's rows is returned as the ordered label for the input subgraph. Algorithm 3 gives a more formal explanation for ordered labeling. Actually, ordered labeling algorithm just changes the labels of vertices and the subgraph structure is not changed. Hence, ordered labeling algorithm preserves graph isomorphism class and canonical labeling. Figure 2 shows an example of ordered labeling.

In the ordered labeling algorithm, we perform $O(k^2)$ lookups, assuming k as the size of the subgraph. Hence, if we use a data structure like *hash table* that provides

Fig. 2 An example of ordered labeling

O(1) expected lookups, then the time complexity of ordered labeling algorithm would be $O(k^2)$ which is far better than traditional canonical labeling algorithms that have $O(k!)$ time complexity. For using ordered labeling in MRSUB, we must replace the *Canonical_Label* function call in line 10 of Algorithm 1 with the *Ordered_Label* function given in Algorithm 3.

Algorithm 3: Ordered labeling function

let H be the input subgraph and M its adjacency matrix

Function *Oredered_Label*(H,M)

1. **let** L be a list of vertices, and initially $L=\emptyset$
2. **for each** vertex $v \in V(H)$ **do:**
3. insert v to L
4. **endfor**
5. sort L by degree (number of neighbors) of each vertex
6. let *Lookup* be a lookup table and *Lookup*[x] as the value associated to x.
7. **for each** vertex $v \in L$ **do:**
8. **set** *Lookup*[v] equal to rank of v in L
9. **endfor**
10. let N be a binary matrix of size M filled with zeros
11. **for each** $M_{i,j}$ in M **do**: //$M_{i,j}$ denotes the element of matrix M in row i and column j
12. **if** $M_{i,j}=1$ **then set** $N_{Lookup[i],Lookup[j]}$ to 1
13. **endfor**
14. **return** concatenation of rows of N

When we use ordered labeling, the output of MRSUB is a set of ordered labels and their frequencies. However, ordered labeling is just a heuristic for fast isomorphism detection and it may not detect all of the isomorphic subgraphs. For extracting the exact isomorphic classes we must apply nauty on the obtained ordered labels. Hence, the subgraph classification phase is broken into two steps: ordered labeling step and canonical labeling step. Generating canonical labels from ordered labels can be done in consecutive MapReduce job which is given in Algorithm 4. In the map function we compute the canonical label of each ordered label and in the reduce function we aggregate the frequency of each canonical label. Using ordered labeling instead of canonical labeling can significantly improve the performance of MRSUB because it significantly reduces the canonical labeling function call which is very time consuming.

Algorithm 4: MapReduce algorithm for extracting canonical labels from ordered labels

let *ordered_label* be the input key denoting an ordered label, and *count* be a its frequency

Function *map*(*ordered_label, count*)

1. **emit**(*Canonical_Label*(*ordered_label*), *count*) //delivers the canonical label to reduce

let *label* be the input key denoting a canonical label, and *values* be a list of integers

Function *reduce*(*label, values*)

2. *sum* ← 0
3. **for each** *val* **in** *values* **do:**
   sum ← *sum* + *val*
4. **endfor**
5. **emit**(*key*,*sum*)

4.4 Complexity analysis

In the worst case, for a complete input network of size n, the number of induced subgraphs of size k is $C(n,k)$. On average, for a general input network of size n, the number of subgraphs of size k should be exponential [3]. However, there is no formula for the number of subgraphs of size k of a given network. The PSE algorithm enumerates each subgraph just once. Hence, if we assume α as the number of subgraphs of size k, β as the number of ordered labels of subgraphs, and p as the number of processors, we can determine the time complexity of MRSUB as follows.

The map step of the enumeration phase, Algorithm 1, enumerates each subgraph just once and computes its ordered label. The ordered labeling algorithm performs $O(k^2)$ lookups for each subgraph. If we use a data structure like hash table which provides O(1) expected lookups, then the time complexity of ordered labeling algorithm would be $O(k^2)$. Therefore, the time complexity of the map step of enumeration phase is $O(\alpha \times k^2/p)$. The reduce step of the enumeration phase, Algorithm 2, receives ordered labels from the map step and counts the frequency of each ordered label. Since the MapReduce framework internally sorts the input of reduce function, the time complexity of the reduce function is $O(\beta \times \log \beta)$.

We use Algorithm 4 for extracting canonical labels from ordered labels which is a MapReduce job. In the map step, a canonical label is computed for each ordered label. The time complexity of computing a canonical label for a subgraph of size k is $O(k!)$. Hence, the time complexity of the map step is $O(\beta \times k!/p)$. In the reduce step of Algorithm 4, the frequencies of each canonical label is computed. Since the MapReduce framework internally sorts the input of reduce function, the time complexity of the reduce function is $O(\beta \times \log \beta)$.

Considering the time complexities discussed above, the overall time complexity of MRSUB, which involves Algorithms 1, 2, and 4, is $O((\alpha \times k^2 + \beta \times k!)/p + \beta \times \log \beta)$. The time complexity of other solution like Kavosh and FANMOD are $O(\alpha \cdot k!)$ [3,4]. In practice, β (the number of ordered labels) is very smaller than α (the number of subgraphs). Hence, we can state that even in the serial mode ($p = 1$) MRSUB is faster than other solutions because the coefficient of α is $k!$ for other solutions and k^2 for MRSUB (especially when $k \geq 4$).

4.5 Proof for the completeness of enumeration

Lemma 1 *The enumeration part of MRSUB, given in Algorithm* 1*, is complete. That is to say, MRSUB enumerates all of the subgraphs of size k of a given network.*

Proof If the enumeration is not complete, then there is a subgraph S with vertices $(v_1, v_2, \ldots, v_k)$ that is not found by MRSUB. We can show that such subgraph cannot exist. Without loss of generality, assume that v_1 is the smallest vertex of S, and v_2 is the smallest neighbor of v_1 in S. Algorithm 1 has certainly built a tuple like T_0 in line 6 for this pair. Again, without loss of generality if we assume that v_3 is the smallest vertex in S that is a member of $N(\{v_1, v_2\})$, v_3 must be a member of the extension set of T_0. If v_3 is not a member of the extension set of T_0, there are three possibilities:

(i) v_3 is smaller than v_2 which is a contradiction.
(ii) v_3 is a mutual neighbor of v_1 and v_2 that is smaller than $v_2$which is also a contradiction.
(iii) v_3 is smaller than v_1 which is also a contradiction.

Hence, v_3 is a member of the extension set of T_0. As a result a new tuple like T_1 is created in line 16 of Algorithm 1 so that the subgraph set of T_1 is equal to $\{v_1, v_2, v_3\}$. Continuing the above procedure inductively, we can reach to a tuple like T_k with subgraph set equal to $\{v_1, v_2, \ldots, v_k\}$. Therefore, MRSUB enumerates all subgraphs of size k and hence, the enumeration is complete. □

4.6 Implementation details of MRSUB

MRSUB is implemented on top of Hadoop. Therefore, we have tuned MRSUB according to Hadoop's architecture. Hadoop automatically sets the number of map tasks according to the number of blocks of the input file. But, we want to set the number of map task manually in order to control the granularity of tasks. To force Hadoop to create a given number of map tasks, we split the input file manually. Therefore, the number of both map and reduce tasks can be manually set to any number in MRSUB and this helps utilizing more processors and setting proper task granularity.

Another concern is about emitting too many tuples in the map tasks. During map task, a tuple is emitted when a subgraph is found. In Hadoop, emitting too many tuples can become a performance overhead. In order to decrease the number of emits, we use an in-memory cache in map tasks and instead of emitting a tuple each time a subgraph is found, we add tuples to the in-memory cache and when the cache becomes full or the task finishes we emit all of the contents of the cache altogether. This technique increases the speed of MRSUB. We have also defined a combiner method which does a local reduce step before the global reduce step begins. In addition, we have configured Hadoop to compress all output of map and reduce steps. Using compression in addition to combiner method, decreases network data transfer volume.

5 Experimental results

In order to evaluate MRSUB's performance, we have tested MRSUB with several real-world complex network datasets. The networks can be divided into two groups: small networks, and large networks. The small networks are: *Elegans* (neuronal network of *Caenorhabditis elegans*) [31], *PPI* (a protein interaction network) [32], *Jazz* (jazz musicians network) [33], and *School* (face to face contact patterns in a primary school) [34]. The large networks are: *Wiki* (Wikipedia who-votes-on-whom network) [35], *Slash* (Slashdot social network from February 2009) [35], *Tweet* (Social circles from Twitter) [36], *P2P* (Gnutella peer to peer network from August 31 2002) [37], and *Web* (Web graph of Notre Dame) [38]. For small networks, large subgraphs (e.g. larger than 6) can be enumerated in reasonable times, but for large networks only smaller subgraphs (less than 6) can be enumerated in reasonable times. More details about networks including number of nodes, number of edges, average of vertex degrees, and the standard deviation of vertex degrees are given in Table 1.

Table 1 Details of the networks

	Elegans	Jazz	PPI	School	P2P	Slash	Tweet	Web	Wiki
#Nodes	297	198	2,906	238	62,586	82,168	81,306	325,729	7,115
#Edges	2345	2742	13,161	5,539	147,892	948,464	1,768,149	1,497,134	103,689
Avg (deg)	15.56	28.02	10.22	48.77	7.58	48.77	38.21	10.29	30.08
σ (deg)	12.76	17.20	10.95	19.81	5.68	19.81	67.93	42.94	57.15

During experiments, we used 40 machine instances from our own private cloud. Each machine had two 2.4 GHz Xeon CPUs and 10 GB of RAM. Ubuntu 12.04 was installed on all machines and Hadoop 1.2 was also deployed on the infrastructure. The Hadoop cluster was configured to run 80 map tasks and 40 reduce tasks simultaneously.

We split this section into three subsections. First, we evaluate the effectiveness of ordered labeling heuristic. Next, we evaluate the scalability and parallelism performance of MRSUB. And finally, we compare MRSUB's performance to other similar sequential tools.

5.1 Effectiveness of ordered labeling algorithm

Ordered labeling is a heuristic that aims to reduce the expensive cost of canonical labeling. Ordered labeling algorithm preserves canonical labels. That is to say, no canonical labels are added or removed and also the frequencies of canonical labels are untouched. However, there are two concepts in the evaluation of ordered labeling: (i) "how many times does it reduce the number of calls to the expensive nauty algorithm for canonical labels", (ii) "what is the difference between the number of generated canonical labels and generated ordered labels". For this purpose we enumerated various subgraph sizes for each network. For each network and subgraph sizes, we reported the number of subgraphs, the number of generated ordered labels, and the number of generated canonical labels (i.e. number of isomorphic subgraphs). Beside the number of each type of label, we reported its count compared to enumerated subgraphs as a percentage number. The numbers are given in Table 2.

Comparing the numbers of generated ordered labels to generated canonical labels shows that our heuristic is usually close to the exact solution. The percentage of generated ordered labels to the number of subgraphs which are given in Table 2 show that our heuristic is effective in reducing the number of calls to the nauty algorithm. By using the ordered labeling algorithm instead of calling nauty for each found subgraph, nauty is called just for each ordered label. Since, ordered labeling takes $O(n^2)$ and nauty takes $O(n!)$ in the worst cases, this heuristic can accelerate isomorphism detection significantly. For example, for the subgraphs of size 7 of the Jazz network, instead of calling nauty about 6.64×10^{11} times (number of all subgraphs) which mean order of $6.64 \times 10^{11} \times 7!$, MRSUB calls ordered labeling about 6.64×10^{11} times which means order of $6.64 \times 10^{11} \times 7^2$, and then calls nauty about 3.8×10^7 (number of ordered labels) which means the cost is reduced from order of $6.64 \times 10^{11} \times 7!$ to order of $(6.64 \times 10^{11} \times 7^2) + (3.8 \times 10^7 \times 7!)$.

Table 2 The effectiveness of ordered labeling heuristic

	Subgraph size	#Subgraphs	#Generated ordered labels	Percentage to #subgraphs	#Generated canonical labels	Percentage to #subgraphs
Elegans	6	1,309,307,357	417,610	0.03190	286,376	0.02187
	7	37,818,052,163	13,974,346	0.03695	9,584,962	0.02534
	8	1,035,341,345,095	398,476,048	0.03849	267,762,831	0.02586
Jazz	6	1,266,953,062	9820	0.00078	5647	0.00045
	7	30,166,157,456	483,199	0.00160	237,008	0.00079
	8	664,444,626,570	38,375,868	0.00578	18,569,208	0.00279
PPI	5	238,434,368	16,858	0.00707	8900	0.00373
	6	9,444,066,996	776,884	0.00823	516,490	0.00547
	7	388,091,394,524	27,247,919	0.00702	19,262,345	0.00496
School	5	348,596,925	383	0.00011	267	0.00008
	6	13,140,615,595	9820	0.00007	5647	0.00004
	7	451,141,199,919	487,807	0.00011	237,319	0.00005
P2P	5	449,446,489	454	0.00010	291	0.00006
	6	9,806,726,769	5137	0.00005	2714	0.00003
	7	234,415,296,091	59,483	0.00003	25,230	0.00001
Slash	3	73,778,405	69	0.00009	4	0.00001
	4	24,159,680,898	1144	0.00000	36	0.00000
	5	10,446,369,545,391	38,123	0.00000	583	0.00000
Twe.	3	203,928,262	60	0.00003	13	0.00001
	4	79,482,295,703	1587	0.00000	200	0.00000
Web	3	287,061,164	94	0.00003	13	0.00000
	4	455,044,545,885	3661	0.00000	200	0.00000
Wiki	3	13,328,802	14	0.00011	13	0.00010
	4	2,513,413,248	276	0.00001	199	0.00001
	5	519,749,094,435	17,037	0.00000	9366	0.00000

5.2 Performance of MRSUB

For evaluating the performance of MRSUB, we extracted frequent subgraphs of various sizes for all input networks. We used the power of all of the 40 machines. For each network, we just processed subgraph sizes that take long enough to be efficient to use MapReduce; hence we skipped jobs that took a short time to finish. The details are given in Table 3. An important note that can be stated from Table 3 is the variable nature of the problem. When larger subgraph sizes are being enumerated (e.g. 8), the problem produces more data in the map and reduce stages. On the other hand when small subgraph sizes are being enumerated, the problem becomes totally CPU-intensive and less data is gathered in the map and reduce phase. Also, the amount of data produced for large subgraphs show that in this setting the problem cannot be handled on a single machine using memory-based solutions like Kavosh and FANMOD. The execution

Table 3 Details for execution of MRSUB using 40 machines in the cloud

	Subgraph size	Subgraph count	Elapsed time (min)	Number of map tasks	Map output data size	Reduce output data size
Elegans	7	37,818,052,163	3.5	500	2.9 GB	60 MB
	8	1,035,341,345,095	120	1000	97 GB	1.8 GB
Jazz	7	30,166,157,456	3.2	500	1.5 GB	2.7 MB
	8	664,444,626,570	81	1000	104 GB	192 MB
PPI	7	388,091,394,524	24	1000	8.4 GB	118 MB
	8	16,130,912,333,189	1280	1000	360 GB	3.6 GB
School	7	451,141,199,919	27	1000	6.5 GB	3.2 MB
	8	14,017,841,350,934	1356	1000	664 GB	285 MB
P2P	7	234,415,296,091	34	1000	163 MB	325 KB
	8	5,973,459,119,600	1132	1000	1.7 GB	4 MB
Slash	4	24,159,680,898	3.7	500	5.3 MB	6 KB
	5	10,446,369,545,391	1150	1000	153 MB	196 KB
Tweet	4	79,482,295,703	9.1	1000	6.8 MB	8 KB
Web	4	455,044,545,885	33	1000	19 MB	19 KB
Wiki	5	519,749,094,435	20	1000	151 MB	99 KB

times also indicate that for some networks and subgraph sizes, like subgraphs of size 8 for the School network which takes about a day, solving the problem on single machine is not feasible.

5.3 Scalability of MRSUB

Another important problem is the scalability of MRSUB. That is to say, how does MRSUB scale when additional machines are used and can MRSUB use their power? For evaluating the scalability of MRSUB, we executed MRSUB using 1, 2, 4, 8 … 36, and 40 machines with different networks. To perform this experiment in a feasible time, we just enumerated subgraphs that take below 5 min on 40 machines. Details for each network are given in Table 4. For giving a better overview, the average speedup values are plotted against ideal speedup values in Fig. 3. As Fig. 3 shows, MRSUB scales up well from 1 machine to 40 machines, and for 40 machines, MRSUB reaches around 86 % of ideal scalability which is promising for a MapReduce job.

Besides scalability, MRSUB provides a load balanced solution, too. When describing MRSUB, we mentioned that if we enumerate subgraphs of each vertex in parallel, the generated parallel tasks may be unbalanced because there is a huge variance in the number of subgraphs rooted from each vertex. In contrast, edge-based parallelism provides more fine-grained tasks and load balancing can be performed better.

The top 20 tasks that produce the highest load during enumeration of subgraphs of size 5 for the *Elegans* network are given in Fig. 4. We considered both vertex-based

Table 4 The speedup gained for each network using different number of machines

Network	Size	Number of machines											
		1	2	4	8	12	16	20	24	28	32	36	40
Elegans	7	1.00	1.86	3.70	7.65	10.66	14.18	18.14	21.91	25.61	28.86	31.93	34.75
Jazz	7	1.00	1.86	3.67	7.33	10.87	14.51	17.72	21.41	24.79	28.58	30.69	34.14
PPI	6	1.00	1.83	3.41	6.90	10.10	14.31	17.81	21.04	24.01	26.90	31.38	33.06
School	6	1.00	1.83	3.66	7.28	10.68	14.90	18.64	21.83	25.39	28.51	31.90	34.71
P2P	6	1.00	1.80	3.37	6.54	10.36	14.19	17.82	21.55	24.90	28.59	29.82	34.23
Slash	4	1.00	1.88	3.44	6.46	10.53	13.93	16.56	20.27	23.65	27.25	30.29	32.45
Tweet	4	1.00	1.88	3.58	7.03	10.24	14.05	17.81	21.08	24.43	27.61	31.43	33.40
Average		1.00	1.85	3.55	7.03	10.49	14.29	17.79	21.30	24.68	28.04	31.06	34.31
Deviation		0.00	0.03	0.13	0.40	0.25	0.30	0.58	0.52	0.66	0.72	0.75	0.83

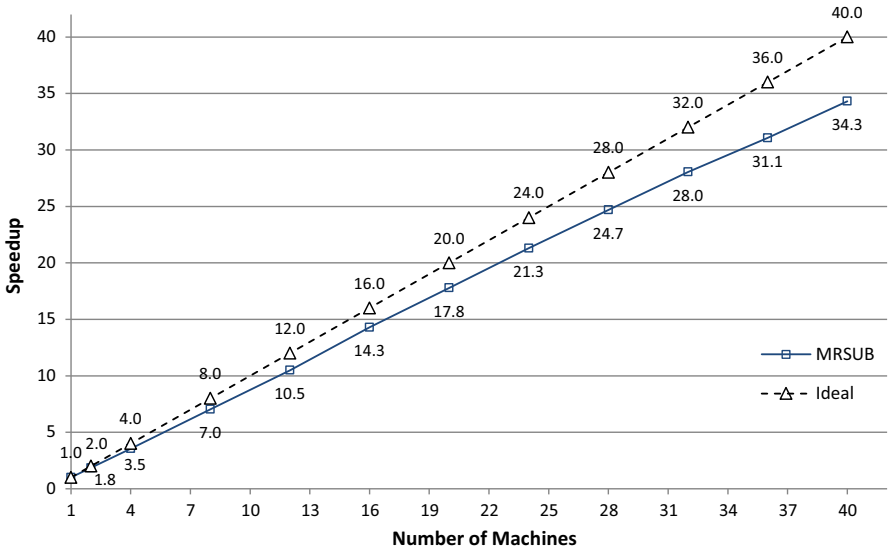

Fig. 3 Scalability of MRSUB from 1 machine to 40 machines

and edge-based approaches. In the vertex-based enumeration, we created a task per each vertex (297 tasks), and the top task produces more than 29 % of the overall load. On the other hand, in the edge-based enumeration, we created a task per each edge (2345 tasks) and the top task produces about 1 % of the overall load. Considering the loads, if we have a system with 100 processors, the tasks produced by vertex-based approach cannot be load balanced because a processor will get a coarse task that have about 29 % of the overall load. In contrast, the tasks of the edge-based approach can be distributed uniformly because they are more fine-grained. This shows that the edge-based enumeration results more fine-grained and load-balanced tasks.

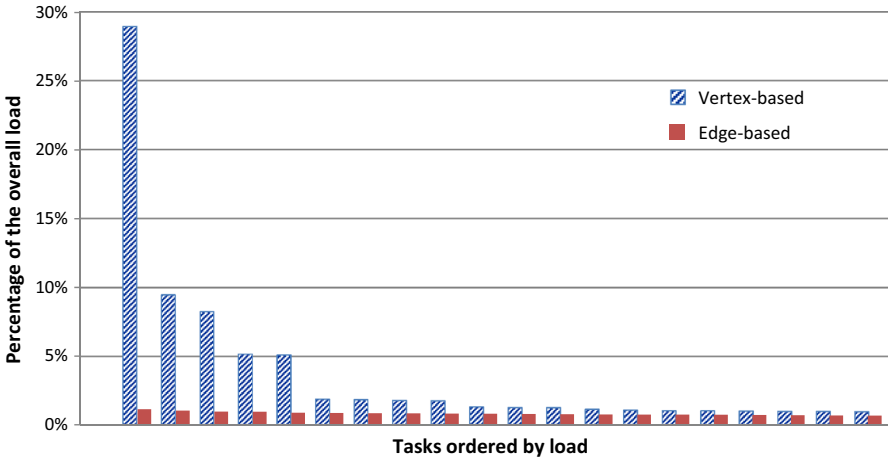

Fig. 4 Workload distribution for the *Elegans* network

5.4 Performance against other solutions

The ultimate goal of MRSUB is to provide the possibility of extracting frequent subgraphs that cannot be done on a single machine in a feasible time. As we stated in the related work section, to the best of our knowledge, there is no distributed solution available for frequent subgraph discovery of a single network. Therefore, the only available competing solutions are serial algorithms like Kavosh and FANMOD which run on a single machine. Since, comparing a parallel solution like MRSUB to other sequential solutions like FANMOD and Kavosh is not fair, we executed MRSUB using a single thread, and outside of Hadoop and compared its single-threaded performance against FANMOD and Kavosh. To finish the experiment in a reasonable time, we just executed the cases that finish in less than a day. The results are presented in Table 5.

In all cases, MRSUB outperforms Kavosh and FANMOD even when it uses a single thread of execution. For larger networks Kavosh was unable to load the network and terminated due to out of memory problems. For the case of *Tweet* network both Kavosh and FANMOD crashed. All of these cases are denoted by '*Crashed*' in Table 5. We also attempted to test subgraph size of 8 for *Elegans* network but both Kavosh and FANMOD crashed after three days of execution. The results tabulated in Table 5 show that besides the absolute superiority of MRSUB's performance, it also enables analysis of larger networks and subgraphs that are not possible using current tools and algorithms. An overall comparison of normalized execution times is given in Fig. 5. In this chart the execution times are normalized by dividing the execution times by the minimum one for each network. Since MRSUB is the fastest solution for all of the networks, its normalized execution time for all of the networks is 1. As Fig. 5 shows, FANMOD is a bit slower than Kavosh, but on the other hand, Kavosh is not able to handle large networks like Slash.

Table 5 Serial performance of MRSUB compared to FANMOD and Kavosh

	Subgraph size	FANMOD (s)	Kavosh (s)	MRSUB (s)	MRSUB's superiority
Elegans	5	93	72	7	10.7×
	6	3895	3650	224	16.3×
PPI	5	387	372	37	10.1×
	6	20,258	20,024	1714	11.7×
Jazz	5	72	58	8	7.1×
	6	2392	2075	248	8.4×
School	5	604	405	51	7.9×
	6	27,230	21,085	2547	8.3×
P2P	5	639	*Crashed*	182	3.5×
	6	17,625	*Crashed*	5048	3.5×
Slash	3	140	*Crashed*	25	5.6×
	4	67,074	*Crashed*	9378	7.3×
Tweet	3	*Crashed*	*Crashed*	62	–
	4	*Crashed*	*Crashed*	30,107	–
Wiki	4	5829	*Crashed*	315	18.5×
Web	3	364	*Crashed*	60	6.1×

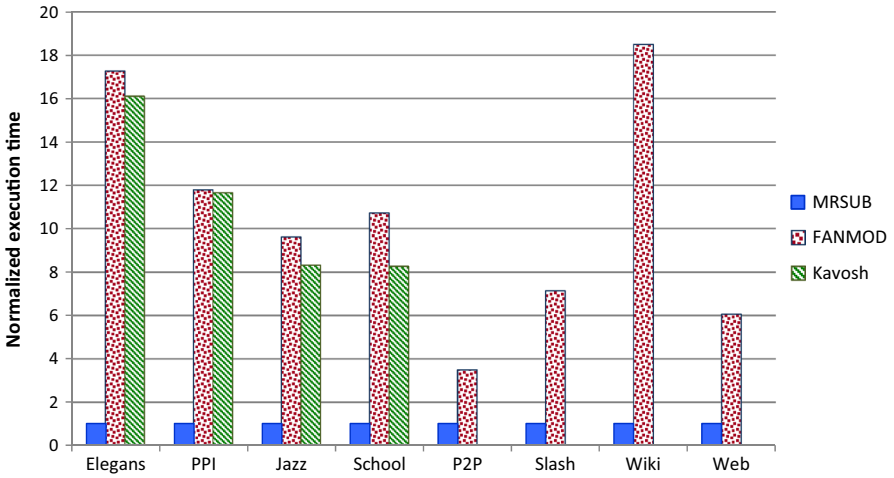

Fig. 5 Normalized sequential execution times considering different networks

5.5 Analysis and discussion

As the experimental results show, MRSUB is absolutely superior to other solutions even in single-threaded execution. There are two main reasons behind this superiority. First, MRSUB is programmed in such a way that it efficiently uses cache-friendly data structures. For example, we have used small sorted array instead of hash tables which

are less cache friendly. The second reason is MRSUB's novel and effective solution to subgraph isomorphism detection. As we discussed in evaluation of ordered labeling algorithm, this heuristic effectively reduces the costly calls to the nauty algorithm.

Note that Kavosh and FANMOD are sequential tools and making parallel versions of these tools is not straightforward. For example, both algorithms use a table for storing the canonical labels that are produced during execution, hence, either a distributed lookup table like Chord [39] must be used which increases the lookup cost from $O(1)$ to $O(logn)$ or, the algorithms must be modified. Therefore, making parallel versions of these algorithms for comparison with MRSUB needs a considerable time and is beyond the scope of this paper. On the other hand, naïve parallel versions of Kavosh and FANMOD will not reach to acceptable speedup because both of the algorithms use vertex-based enumeration which will cause unbalanced parallel loads and this matter will limit the scalability of their parallel versions. Also, there are communication and coordination overheads that are natural for distributed programs and these overheads decrease the parallel performance further.

6 Conclusion and further work

We presented a MapReduce-based solution, named MRSUB, for discovery of frequent subgraphs of a given network. MRSUB uses a parallel and load-balanced MapReduce-based algorithm for subgraph enumeration and a distributed subgraph isomorphism detection component that uses a fast heuristic for subgraph isomorphism detection. To show the effectiveness of MRSUB, we performed several experiments on a private cloud infrastructure consisting 40 machines with the Hadoop framework.

Our experiments showed that MRSUB is effective and scales well in our test-bed. Our experiments also showed that MRSUB can offer a solid solution for discovery of frequent subgraphs of networks which were not possible in feasible time using available tools. MRSUB also enabled processing larger networks and subgraph sizes which were not possible to be processed using available single machine solutions. We showed that MRSUB is superior to other competing solutions even when it is executed using a single machine and a single thread. We evaluated and approved the effectiveness of our heuristic for subgraph isomorphism detection, too.

For processing a network MRSUB must be able to load the network in the main memory of all machines. During our experiments, we processed large input networks with million-sized edges and this is enough for many complex networks. However, there are other networks available with billions of edges and nodes, like large-scale social networks and web graphs. For further work, we plan to extend MRSUB in a way that it can handle billion-scale networks. We have done a part of the work, and the preliminary results are promising.

Additionally, proposing an automatic and intelligent mechanism for setting the number of map and reduce tasks is a possible improvement. Another possibility is using other HPC solutions like GPU clusters for frequent subgraph discovery. Notwithstanding, fitting the frequent subgraph discovery problem into GPU computing may be very challenging. Proposing a better and faster isomorphism detection mechanism can

also improve performance significantly. Furthermore, there is a considerable potential for extending the current work to support multigraphs, hypergraphs, and weighted networks.

References

1. Milo R, Shen-Orr S, Itzkovitz S et al (2002) Network motifs: simple building blocks of complex networks. Science 298:824–827
2. Wernicke S (2006) Efficient detection of network motifs. IEEE/ACM Trans Comput Biol Bioinform 3:347–359
3. Wernicke S, Rasche F (2006) FANMOD: a tool for fast network motif detection. Bioinformatics 22:1152–1153
4. Kashani ZRM, Ahrabian H, Elahi E et al (2009) Kavosh: a new algorithm for finding network motifs. BMC Bioinform 10:318
5. Ribeiro P, Silva F, Lopes L (2012) Parallel discovery of network motifs. J Parallel Distrib Comput 72:144–154
6. Grochow J, Kellis M (2007) Network motif discovery using subgraph enumeration and symmetry-breaking. In: Speed T, Huang H (eds) Research in computational molecular biology. Springer, Berlin, pp 92–106
7. Rudi AG, Shahrivari S, Jalili S, Kashani ZRM (2013) RANGI: a fast list-colored graph motif finding algorithm. IEEE/ACM Trans Comput Biol Bioinform 10:504–513
8. Masoudi-Nejad A, Schreiber F, Kashani ZRM (2012) Building blocks of biological networks: a review on major network motif discovery algorithms. Syst Biol 6:164–174
9. Wong E, Baur B, Quader S, Huang C-H (2012) Biological network motif detection: principles and practice. Brief Bioinform 13:202–215
10. Kang U, Tsourakakis CE, Faloutsos C (2011) PEGASUS: mining peta-scale graphs. Knowl Inf Syst 27:303–325
11. Inokuchi A, Washio T, Motoda H (2003) Complete mining of frequent patterns from graphs: mining graph data. Mach Learn 50:321–354
12. Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: Proceedings of IEEE international conference on data mining, pp 313–320
13. Yan X, Han J (2002) gSpan: graph-based substructure pattern mining. In: Proceedings of IEEE international conference on data mining, pp 721–724
14. Schreiber F, Schwbbermeyer H (2004) Towards motif detection in networks: frequency concepts and flexible search. In: Proceedings of the international workshop on network tools and applications in biology. NETTAB, pp 91–102
15. Kuramochi M, Karypis G (2004) GREW—a scalable frequent subgraph discovery algorithm. In: Proceedings of fourth IEEE international conference on data mining ICDM '04, pp 439–442
16. Holder LB, Cook DJ, Djoko S (1994) Substructure discovery in the subdue system. In: Proceedings of the workshop on knowledge discovery in databases, pp 169–180
17. Afrati FN, Fotakis D, Ullman JD (2013) Enumerating subgraph instances using map-reduce. In: Proceedings of IEEE 29th international conference on data engineering (ICDE), pp 62–73
18. Cohen J (2009) Graph twiddling in a MapReduce world. Comput Sci Eng 11:29–41
19. Suri S, Vassilvitskii S (2011) Counting triangles and the curse of the last reducer. In: Proceedings of the 20th international conference on world wide web. ACM, NY, pp 607–614
20. Kolda TG, Pinar A, Plantenga T et al (2013) Counting triangles in massive graphs with MapReduce. arXiv:1301.5887
21. Pagh R, Tsourakakis CE (2012) Colorful triangle counting and a mapreduce implementation. Inf Process Lett 112:277–281
22. Zhao Z, Wang G, Butt AR et al (2012) Sahad: subgraph analysis in massive networks using hadoop. In: Proceedings of IEEE international parallel & distributed processing symposium (IPDPS). IEEE, pp 390–401
23. Kang U, Tsourakakis CE, Appel AP, Faloutsos C, Leskovec J (2011) Hadi: mining radii of large graphs. ACM Trans Knowl Discov Data 5(2):1–24
24. Johnson DS (2005) The NP-completeness column. ACM Trans Algorithms 1:160–176

25. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51:1–13
26. Dean J, Ghemawat S (2010) MapReduce: a flexible data processing tool. Commun ACM 53:72–77
27. White T (2012) Hadoop: the definitive guide. Yahoo Press, California
28. McKay BD (1981) Practical graph isomorphism. Department of Computer Science, Vanderbilt University, Tennessee, US
29. Junttila T, Kaski P (2007) Engineering an efficient canonical labeling tool for large and sparse graphs. In: Applegate D, Brodal GS, Panario D, Sedgewick R (eds) Proceedings of the ninth workshop on algorithm engineering and experiments and the fourth workshop on analytic algorithms and combinatorics. SIAM, pp 135–149
30. West DB (2001) Introduction to graph theory. Prentice Hall, Englewood Cliffs
31. Kashtan N, Itzkovitz S, Milo R, Alon U (2004) Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. Bioinformatics 20:1746–1758
32. Srihari S, Leong HW (2013) A survey of computational methods for protein complex prediction from protein interaction networks. J Bioinform Comput Biol 11:12–30
33. Pablo MG, Danon L (2003) Community structure in jazz. Adv Complex Syst 6:565–573
34. Stehlé J, Voirin N, Barrat A et al (2011) High-resolution measurements of face-to-face contact patterns in a primary school. PLoS ONE 6:e23176
35. Leskovec J, Huttenlocher D, Kleinberg J (2010) Predicting positive and negative links in online social networks. In: Proceedings of the 19th international conference on World wide web, pp 641–650
36. Leskovec J, Mcauley JJ (2012) Learning to discover social circles in ego networks. In: Proceedings of advances in neural information processing systems (NIPS), pp 539–547
37. Leskovec J, Kleinberg J, Faloutsos C (2007) Graph evolution: densification and shrinking diameters. ACM Trans Knowl Discov Data 1(1):1–41
38. Albert R, Jeong H, Barabási A-L (1999) Internet: diameter of the world-wide web. Nature 401:130–131
39. Stoica I, Morris R, Karger D et al (2001) Chord: a scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput Commun Rev 31:149–160