MapReduce-Based Pattern Finding Algorithm Applied in Motif Detection for Prescription Compatibility Network

Yang Liu, Xiaohong Jiang*, Huajun Chen, Jun Ma, and Xiangyu Zhang

College of Computer Science, Zhejiang University,
Zheda Road. 38, Hangzhou, China, 310027
{darkwarrior,jiangxh,huajunsir,majun,xiangyu}@zju.edu.cn

Abstract. Network motifs are basic building blocks in complex networks. Motif detection has recently attracted much attention as a topic to uncover structural design principles of complex networks. Pattern finding is the most computationally expensive step in the process of motif detection. In this paper, we design a pattern finding algorithm based on Google MapReduce to improve the efficiency. Performance evaluation shows our algorithm can facilitates the detection of larger motifs in large size networks and has good scalability. We apply it in the prescription network and find some commonly used prescription network motifs that provide the possibility to further discover the law of prescription compatibility.

Keywords: complex network, motif detection, pattern finding, MapReduce, prescription compatibility.

1 Introduction

Network motifs are specific pattern of local interconnections with potential functional properties and can be seen as the basic building blocks of complex network [1]. Pattern finding in a complex network is the first and most important step to analyze motifs. Some pattern finding methods are already used to analyze the network motifs in real world such as biochemistry network, ecology network, neurobiology network, and engineering network [1]. And these applications obtain many valuable research results. However, in the pattern finding area, there are many NP-complete problems, such as determining graph isomorphism and maximum independent set [2]. For this reason, the pattern finding algorithms always have high time-space complexity. Moreover, when the size of the pattern is big (usually bigger than 4), the number of the intermediate becomes very large (above millions of items), which makes pattern finding time consuming and memory exhausted.

Google's MapReduce framework is known as the framework of Clouding Computing. MapReduce built on top of the distributed Google File System provides a parallelization framework that has garnered considerable acclaim for its ease-of-use,

* Corresponding Author.

scalability, and fault-tolerance [3]. Therefore, we try to use the Google's Mapreduce framework to speed up pattern finding and avoid running-out-of memory in a PC-cluster environment. We design a MapReduce-based Pattern Finding algorithm (MRPF) that provides good efficiency and scalability. We also apply it in prescription network and successfully find some commonly used prescription structures that propose the possibility to discover law of prescription compatibility.

In our MRPF algorithm, we reorganize the traditional pattern finding process into four steps: distributed storage, neighbor vertices finding and pattern initialization, pattern extension, and frequency computing. Each step is implemented by a MapReduce pass. In each MapReduce pass, the task is divided into a number of sub-tasks of the same size and each sub-task is distributed to a node of the cluster. MRPF uses an extended mode to find the target size pattern. That is trying to add one more vertex to the matches of i-size patterns to create patterns of size i+1. The extension doesn't stop until patterns reach the target size.

To test the computational efficiency of MRPF, we apply it to the prescription compatibility structure detection. The knowledge discovery of prescription compatibility is an important part of Traditional Chinese Medicine (TCM) research. Prescription compatibility investigates the composite structure of herbal medicines. One prescription contains five or six herbal medicines. However prescriptions are commonly given based on experiences without theoretical instruction on prescription structures. So we construct the prescription compatibility network and use our algorithm to detect the prescription compatibility structure.

The rest of the paper is organized as follows. Section 2 introduces some related works on pattern finding methods, applications of MapReduce and some data mining method used in TCM. Section 3 describes our MRPF algorithm in detail. Section 4 gives the case study on prescription compatibility using our algorithm. Section 5 provides some concluding remarks and discussion for future work.

2 Relate Works

The main step in motif detection is pattern finding in the complex network. There are two distinct problem formulations for pattern finding in graph datasets. One is the graph-transaction setting that use a set of relatively small graphs as input data, the other is the single-graph setting using a single large graph instead[4]. Pattern finding in graph-transaction attracts more attention, so that a number of efficient algorithms [5-10] have been developed. However, few investigations have been made in pattern finding from the single-graph setting. Moreover, some algorithms, such as GBI [11] and SUBDUE [12], will lose a large number of patterns, and at the same time not scale well for large datasets due to computational complexity. In recent years, with the application of pattern finding increasingly used in many fields, researchers start to pay more attention in designing algorithms for single-graph setting. In 2005, Michihiro Kuramoch and George Karrypis developed an algorithm to find patterns in a large sparse graph [4]. Falk Schreiber and Henning Schwobbermeyer designed a FPF multiple-thread algorithm [13] to improve the performance of Michihiro's algorithm. Jin Chen et al. designed a NeMoFinder algorithm that can mine meso-scale network motifs in large protein-protein interaction networks [14]. In 2007, Chen Chen et al. invented a gApprox algorithm that does consider approximate matching in its search space [15]. However all these algorithms mentioned above ignored to consider the limitation of the main memory of one computer. So for further improving the performance of pattern finding and breaking through single computer resource constraints, we design a parallel pattern finding algorithm based on MapReduce Framework. It's a complete algorithm without losing any target-size pattern in the network.

MapReduce Framework, as a parallel model, is often used in data mining, such as machine learning [16], svm [17]. These experiments demonstrate that MapReduce Framework is effective for problems with high complexity and large dataset. It is also proved that MapReduce can be adapted to manipulating graphs. Implementation of pattern finding in the context of MapReduce Framework is able to address the issues of insufficient memory, computational complexity and fault tolerance. Many of data mining methods have been used in the Modernization of Traditional Chinese Medicine. Text mining method is used for finding functional community of TCM Knowledge [18]. Mining compatibility rules are used for TCM databases [19]. Clustering method is applied to analyzing Chinese Text Categorization [20]. Prescription compatibility is investigated in [21-23], but very little work has been done on motif detection in prescription network, which is very important for law discovery of prescription compatibility. We analyze the prescription compatibility using the complex network and find some commonly used prescription structures.

3 MRPF: MapReduce-Based Pattern Finding

MapReduce-based pattern finding (MRPF) framework aims to implement frequent pattern finding on complex graphs based on Hadoop. Although it also works well on undirected graphs, here we still focus on introducing its application on directed graphs. It's more interesting and representative to apply this framework on directed graphs. For clearly depicting MRPF, we show the serial pattern finding algorithm in Algorithm 1.

3.1 MRPF Framework

Here we define the size of a pattern by its vertices number. We use the generation of a canonical label described in [24] to check graphs for isomorphism. After loading a dataset of a network, MRPF uses one MapReduce pass to parse the dataset and form three information tables. Another MapReduce pass is used to extend *matches* that are subgraphs of the network from size i to i+1. The frequency of new patterns will be calculated after all matches of patterns of size i+1 have been obtained. Fig. 1 depicts the outline of MRPF.

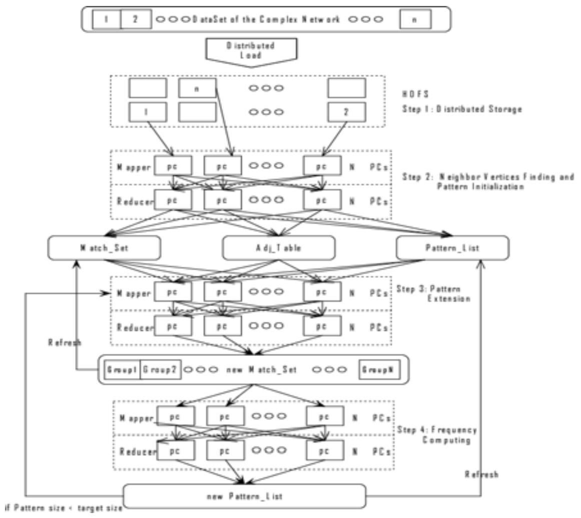

1
2
DataSet of the Complex Network
n
Distributed Load
n
1
2
HDFS
Step 1: Distributed Storage
Mapper
pc
N PCs
Reducer
Step 2: Neighbor Vertices Finding and Pattern Initialization
Match_Set
Adj_Table
Pattern_List
Step 3: Pattern Extension
Refresh
Group1
Group2
new Match_Set
GroupN
Step 4: Frequency Computing
Refresh
new Pattern_List
if Pattern size < target size

Fig. 1. The MRPF framework

Algorithm 1. Normal Pattern Finding

```
Data: Dataset of Graph G, target pattern size s, minimum support (f_min)
Result: Set P of pattern of target size
begin
  P ← {all pattern of size 2};
  size ← 2;  /* initial size */
  $MATCH_{p2}$ ← all matches of p2;
  TPS ← Φ;  /* TPS: target pattern size */
  while size < target size do
    foreach pattern p ∈ P do
      foreach match m ∈ $MATCH_p$ do
        foreach incident  vertex v of m do
          m′ ← m ∪{v };
          p′ ←  pattern of {m′};
          TPS ← TPS∪{p};
          $MATCH_{p'}$ ←  $MATCH_{p'}$∪{ m′};
        end
      end
    end
    P ← Φ;
    foreach  p ∈ TPS do
      frequency ← sizeof ( MATCH_p);
      if frequency > f_min then P  ← PU{p};
     end
     size++;
    end.
```

Step 1: Distributed storage. MRPF is based on Hadoop, a Google's GFS implementation, hosted as a project of the Apache Software Foundation [16]. In Step 1, the target network is stored as textual files in a specific format. Using Hadoop, the file can be easily divided into a set of blocks with the same size and distributed on nodes of the cluster to keep load balance in the cluster. Hadoop can process the blocks concurrently on nodes where the data is located.

Step 2: Neighbor vertices finding and pattern initialization. In this step we use a MapReduce pass to do two tasks, one is to find adjacent neighbor of each vertex to form an adjacent vertices table *(Adj_Table)*, the other is to find patterns of size two (one edge and two vertices) and their matches. Each mapper inputs one block of the dataset. The results are respectively stored in *Adj_Table, Match_Set* and *Pattern_List*. Please note that Adj_Table is distributed to every node in the cluster and it will be used in pattern extension (Step 3). It is used to detect the patters on the borders of blocks and to guarantee our algorithm to be complete (against losing patterns). Match_Set and Pattern_List are updated by Step 3 and Step 4 respectively. We will introduce the details in Section 3.2.

Step 3: Pattern extension. It is the key step of the MRPF. This step also takes one MapReduce pass. The map stage working with reduce stage extends patterns of size i to i+1. The details will be explained in Section3.3.

Mapper – extend the matches of size i to i+1, calculate their patterns and produce a group key with the patterns and matches. Each mapper outputs one or more key-value pairs, and the pairs with the same key will automatically be grouped into the same reducer.

Reducer - remove the duplicated matches. Since different matches may get the same subgraph of size i+1 when the matches i are extended. During the grouping process mentioned above, we compare the canonical label of each match and keep just one of the same matches. The outputs of reducers are grouped into different files according to the pattern label.

Step 4: Frequency computing. After pruning the identical subgraphs, a MapReduce pass is used to count the support value of all patterns that appear in the big simple graph. We prune the patterns lower than the minimum required frequency. Then we store new patterns in Pattern_List. Go back to Step 3 to process iteratively till the target pattern size is reached. The details are given in Section 3.4.

3.2 Neighbor Vertices Finding and Pattern Initialization

Just like a classical application of MapReduce, each mapper of the first MapReduce pass is fed with one block of dataset. The input key-value pairs would be like <key, value = edge (V_i,V_j) > (V_i and V_j are adjacent vertex to each other), where edges belong to dataset. Mappers produce two kinds of keys: the vertex key according to vertex label and the pattern key according to the pattern *canonical label*. Mappers travel through all edges of the graph, each Mapper outputs three key-value pairs <key_1 = V_i, $value_1$=V_j>, <key_2 =V_j, $value_2$=V_i> and <key_3 = pattern2, $value_3$= edge (V_i,V_j) >.

After all mapper instances have finished, the MapReduce infrastructure automatically collects and groups the key-value pairs according to the keys. The values with the same key are put into the same group, called G (key), and reducers receive the key value pairs <key, G (key)> where G (key) is adjacent vertices of a vertex or the match of a pattern whose size is two. Reducers compose the G (key) into an adjacent vertices list or match list and outputs <key, list> into Adj_Table or Match_Set according to the class of each key, where the list is vertices list or match list. Algorithm 2 presents the pseudo code of this step. Through this process, it registers each vertex's adjacent vertices. Meanwhile, it finds the smallest patterns (of size 2) and their matches.

Algorithm 2. Neighbor Vertices Finding and Pattern Initialization

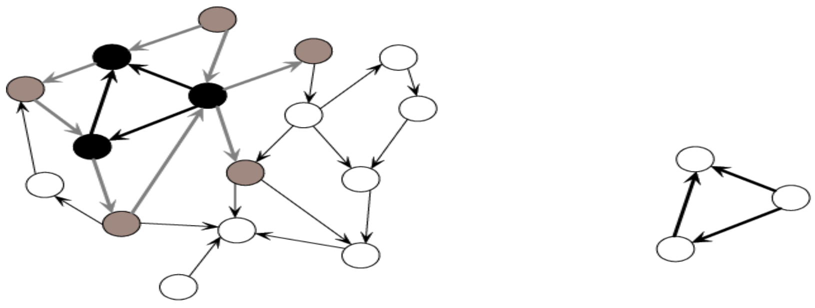
```
Procedure: Mapper(key, value = Edge($V_i$ :$V_j$))
  /*  p is the canonical label of Edge($V_i$:$V_j$)   */
    p getPattern(Edge($V_i$:$V_j$))
    EmitIntermediate (<key = $V_i$, value = $V_j$>)
    EmitIntermediate (<key = $V_j$, value = $V_i$>)
    EmitIntermediate (<key = p, value = Edge($V_i$:$V_j$)>)

Procedure: Reduce(key, value = G(key))
  /* Adj_List : adjacent vertices list;
    Match_List: matches of the same pattren */
  Adj_List
  Match_List
  if key is vertex label then
```

```
    foreach item $v_i$  in G(key) do
      Adj_List Adj_List {$v_i$}
    end
    Emit(<key, Adj_List >)
  else
    foreach item $match_i$ in G(key) do
      Match_List Match_List { $match_i$}
    end
    Emit(<key, Match_List>)
  end
```

3.3 Pattern Extension

This step is the key part of the MRPF algorithm. This step, together with the step 4 frequency computing, will be repeated until the target size pattern is obtained. In the pseudo code of the Algorithm 3 below we will see the procedure of how we use the MapReduce Framework clearly.



(a) Example graph G_T **(b)** example pattern of size 3

Fig. 2. (a) G_T is a graph. The subgraph (highlighted with bold lines) in G_T is a match of the pattern in (b). Gray vertices in G_T are *incident* vertices of the match and gray edges are *detected* edges. (b) A pattern of size 3.

First, we load the adjacent vertices table (Adj_Table) which can be stored in memory to find incident vertices of matches. Then load the Pattern_List where keeps all the patterns of size i. The initial state of the Pattern_List is defined as “Starting”. The input of mapper is from Match_Set. As shown in Fig. 2, if a match (highlighted in bold black line) is found in the graph, we call the vertices (in gray) adjacent to the matched vertices (in black) incident vertex of a match. And the edge between an incident vertex and the match is called detected edge.

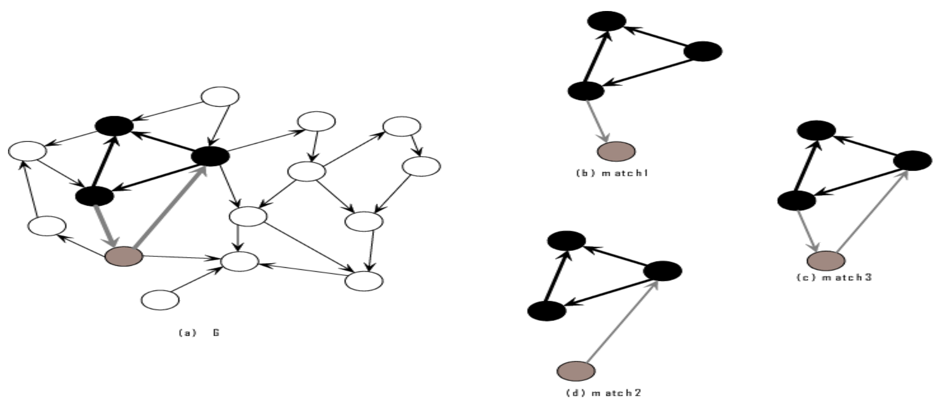


Fig. 3. (a) A graph with a randomly selected subgraph (highlighted with bold lines). The subgraph is a match (M_3) of some pattern. (b), (c), (d) are the extension matches from M_3.

In mappers, the input pair would be like <key, value=Pattern$_i$ & match$_i$> from the Match_Set, where Pattern$_i$ means the pattern has i vertices and match$_i$ means the match has i vertices. If the Pattern$_i$ is contained in Pattern_List or Pattern_list is in initial state, it extends match$_i$ to matchi+1 through adding each incident vertex into match$_i$ as shown in Fig.3. Then it adds various combinations of detected edges into match$_i$ and forms new matches.

We compute Pattern$_{i+1}$ of match$_{i+1}$ and the canonical label of the Pattern$_{i+1}$ as part of output value. Each mapper outputs one or more pairs like <key'= match$_{i+1}$, value'=pattern $_{i+1}$>.

Each reducer receives a <key'= match$_{i+1}$, S (key'))> where the S (key') has only one element------the pattern of the match$_{i+1}$ that is the key of key-value pair. In this way we can easily wipe off the identical matches. Each reducer outputs one <key'' = Pattern$_{i+1}$, key'>, the MapReduce infrastructure sorts and groups the key-value pairs according the key'' value, then produces the successive block based on grouping. Blocks are stored in N different computers, which is convenient to deal with the data in the next MapReduce process.

Algorithm 3. Pattern Extension

```
Procedure: Mapper(key, value = Pattern_i & match_i)
  Load  Adj_Table
  Load  Pattern_List
  if Pattern_i in Pattern_List
    or Pattern_list is in the Starting state then
    foreach incident vertex of match_i  do
      foreach combination C_i of detected edges
        between the incident vertex and match_i do
        $match_{i+1}$  $match_i$ { $C_i$ }
        $Pattern_{i+1}$  corresponding pattern of $match_{i+1}$
        EmitIntermediate(<key = $match_{i+1}$, $Pattern_{i+1}$>)
      end
    end

Procedure: Reduce(key= $match_{i+1}$, S (key))
    $Pattern_{i+1}$  one of S (key)
    Emit(<key = $Pattern_{i+1}$, $match_{i+1}$>)
```

3.4 Frequency Computing

Frequency computing is a simple counting process, a classical application of MapReduce. Its input is the output of the step3. Algorithm 4 presents the pseudo code of the two steps: grouping and parallel counting. The mapper's input is <key, value = T>, where T is composed of pattern canonical label and matches. It picks up the pattern canonical (P) from T. The mapper outputs a key-value pair <key' = P, value'=1>.

After completing all of the Mapper instances, for each key transmitted by Mapper, a value set (S (key')) is automatically formed. And each reduce is fed with <key', S (key')>. The reducer outputs <key''= key' value'' = sum(S (key'))>.

Then the pattern frequency is calculated based on the occurrence quantity of each pattern. In this paper, to show the full potential of the prescription network we use the frequency concept which counts every match of the pattern. It gives a complete overview of all possible occurrences of a pattern even if elements of the target graph are used several times. So it does not satisfy the Downward Closure Property [4]. And we do not prune the infrequent patterns that lower than the target pattern size. Note that the occurrence quantity of some patterns is too small to affect the pattern finding results. We call these patterns *dust patterns.* A minimum required frequency variable (f_min) is defined to prune the dust patterns. The value of f_min is given by experts according to their experience.

Algorithm 4. Frequency Computing

```
Procedure: Mapper(key, value = $Pattern_i$ & $match_i$)
  /* PL: pattern label*/
  PL  the canonical label of $Pattern_i$
  EmitIntermediate(<key = PL, '1'>)

Procedure: Reduce(key , S (key))
  Sum 0
  foreach item '1' in S(key) do
    Sum Sum + 1
```

```
end
/* total is the quantity of all patterns, f_min is a minimum
required frequency variable */
if  Sum total * f_min
   Emit(<key = Pattern$_{i+1}$,  Sum>)
end
```

4 Application to Prescription Compatibility Structure Detection

4.1 Motifs Detection Results

A key subject of prescription research is theoretical study on prescription compatibility regularity. The structure of Monarch, Minister, Assistant and Guide is the compatibility principle for prescriptions and the base for the overall efficacy of prescriptions. However, people as yet know nothing about the commonly used compatibility structure. In other words, people still have no acquaintance with most appropriate ratio of these four kinds (e.g. Monarchs, Ministers, Assistants and Guides) of Chinese herbal medicines respectively participating in the compatibility of a prescription, which is very important to exert the overall efficacy.

In the prescription compatibility network, node represents Chinese herbal medicine, while edge describes the compatibility relation that might exist between the two herb nodes, and edge direction indicates the relative position between the two connected herb nodes from the higher one to the lower one. According to the multi-types of the relative positions between any two herb nodes, the compatibility relations of them vary greatly. Table1 shows in detail all types of possible compatibility relations in terms of criteria for the classification of herbs.

Table 1. All types of possible compatibility relations

Herbal	compatibility relation	Herbal	compatibility relation
Monarch, Minister	Monarch → Minister	Monarch, Assistant	Monarch → Assistant
Monarch, Guide	Monarch → Guide	Minister, Assistant	Minister → Assistant
Minister, Guide	Minister → Guide	Assistant, Guide	Assistant → Guide

We select 201 prescriptions that are explicit in the compatibility structure from [25] to construct the prescription compatibility network. The network contains about 300 vertices and 2, 000 edges, as shown in Fig.4.

We apply our algorithm in the prescription network, and after comparing with random networks, we find a number of motifs of prescription network (see Fig.5) and their occurrence quantity shown in table 2.

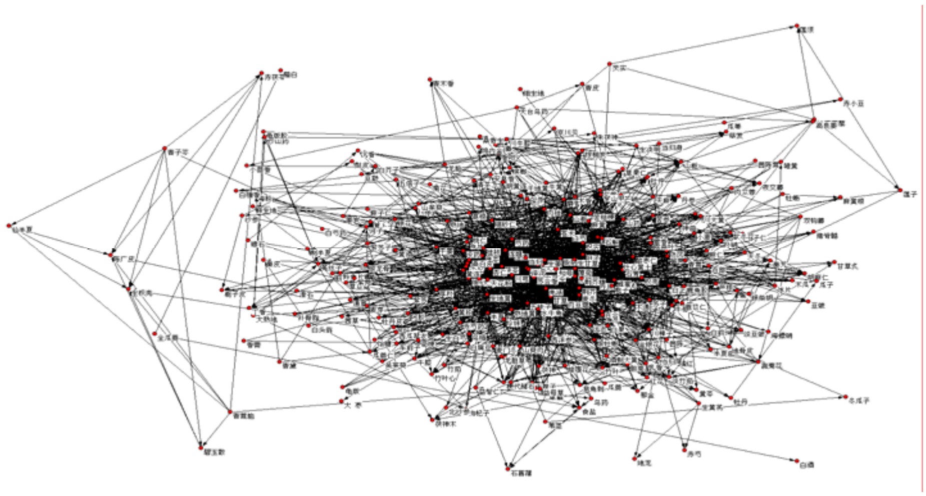


Fig. 4. The topological structure of the prescription compatibility network

These motifs are the basic structure of the prescription compatibility. For example, Motif1 consists of one Monarch, one Minister, one Assistant and one Guide; Motif3 contains one Monarch, one Minister, one Assistant and one Guide; Motif5 contains one Monarch, two Ministers and two Assistants. They are of great value to further discover the law of prescription compatibility.

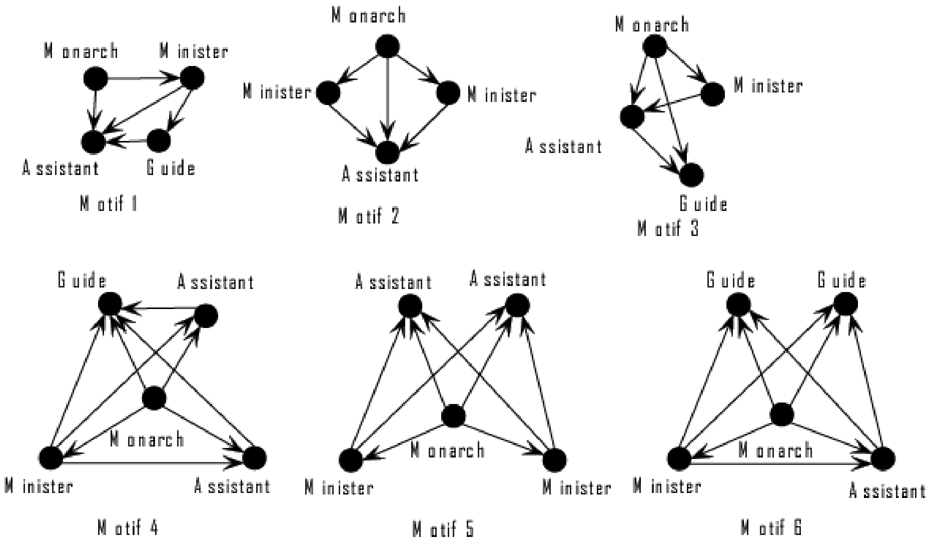


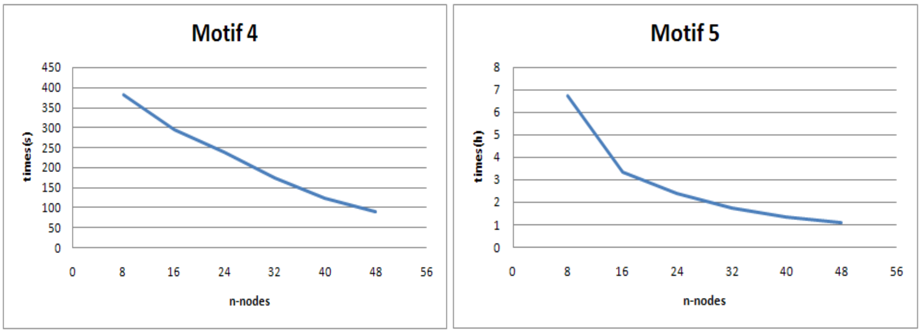
Fig. 5. Six motifs of size 4 and size 5 separately

Table 2. Frequency for each motif in Fig.4

Motif	Frequency
Motif1	0.4093750%
Motif2	0.3534964%
Motif3	0.3728833%
Motif4	0.0038188%
Motif5	0.0026400%
Motif6	0.0018862%

4.2 Performance Analysis

Our algorithm automatically divides the job and distributes them to each node. So we can dynamically add the quantity of nodes, which will enhance the performance of the algorithm. We run the program on a blade-cluster with 48 nodes. Each node is equipped with Intel (R) Xeon (TM) CPU 2.80 GHZ and 1 GB memory. In the experiment, we run the algorithm to do the same task on cluster of varying nodes. And the experiment results of finding size 4 and size 5 motifs are shown in Fig.6.



(a) Motif size = 4 (b) Motif size = 5

Fig. 6. Algorithm performance on the cluster

From the above figure, it's clear that execution time decreases quickly while the cluster nodes increase. It implies that our algorithm scales well with the computing nodes. However, the performance acceleration decreases when the cluster exceeds a number of nodes for a fixed size task. In this experimentation, we just prove the scalability of MRPF. Here we theoretically analyze reasons of the acceleration decreasing. We define the formula of the execution time of MRPF as followed:

$$\mathbf{T_{total}(N)} = \frac{\mathbf{C_{fix}}}{\mathbf{N}} + \mathbf{T_{overhead}} = \frac{\mathbf{C_{fix}}}{\mathbf{N}} + \mathbf{T_{map}}\,(\mathbf{N}) + \mathbf{T_{reduce}(N)} + \mathbf{T_{M/_S}(N)}\,. \quad (1)$$

In the formula (1), N is the number of data nodes; $\mathbf{C}_{\mathbf{fix}}$ is a constant that represents the computation complexity of the fixed size task and it is distributed to each node evenly; $\mathbf{T_{map}(N)}$, $\mathbf{T_{reduce}(N)}$ and $\mathbf{T_{M/_S}(N)}$ respectively denote Map Task initialization time, the time for each reducer receiving the intermediate, and the time for the communication between the master node and slave nodes. The Map Task initialization time $\mathbf{T_{map}(N)}$ includes assigning tasks, preparing data and task issuing. According to our experience, the total number of map tasks is better to be set about 3 or 4 times of the number of nodes, which can make full use of the resource of the cluster. So we divide the task dynamically according to N. While the number of the data blocks increasing with N, the number of the Map Task increases and the total time of map initialization and intermediate distribution increase too. It is obvious that$\mathbf{T_{map}(N)}$ and $\mathbf{T_{reduce}(N)}$ are increasing with N. And the cluster is organized in master/slaves mode, with only one master node responsible for data retrieval, task assignment and task snooping on the slave nodes. So while the node number N increase, the communication overhead between master and slaves will also increase. So the value of $\mathbf{T_{M/_S}(N)}$ is increasing with N. The speedup of MRPF can be calculated using the following formula:

$$\mathbf{Speedup} = \frac{\mathbf{T_{total}(N)}}{\mathbf{T_{total}(N+1)}} = \frac{\frac{\mathbf{C_{fix}}}{\mathbf{N}} + \mathbf{T_{map}}\,(\mathbf{N}) + \mathbf{T_{reduce}(N)} + \mathbf{T_{M/_S}(N)}}{\frac{\mathbf{C_{fix}}}{\mathbf{N+1}} + \mathbf{T_{map}}\,(\mathbf{N+1}) + \mathbf{T_{reduce}(N+1)} + \mathbf{T_{M/_S}(N+1)}}\,. \quad (2)$$

From the formula (2), it can be deduced that the speedup might degrade due to the increasing overhead even if C_{fix} is allocated by the cluster nodes. It can be clearly observed that there is an inflection point in exertion time curve in figure 6(b) when the number of the data nodes equals 16. It may be caused by the topology of the cluster or architecture of MapReduce Framework. We need to do further experiments to investigate into it.

5 Conclusion

In summary, the contributions of this paper are as follows:

1. We designed a MapReduce-based pattern finding algorithm (MRPF) for analyzing the complex network. We reorganized the pattern finding process and implemented each step using the MapReduce framework, which makes MRPF parallelizable and extensible. The experiment evaluation on the expending of nodes in Section 4.2 indicated that increasing the number of the nodes would enhance the performance of MRPF.

2. We applied the complex network analysis method to the prescription compatibility network and used MRPF to find the commonly used compatibility structure. And we found some prescription structures which reflect characters of the law of compatibility of medicines in prescriptions in some way.

More experiments need to be done to evaluate the algorithm performance considering the factors of data block size, node number, and network bandwidth, etc. In fact, developing MapReduce based pattern finding algorithm is actually the first step to our target, to develop a parallel data mining library based on MapReduce that can be applied in many fields. And we will also testify these parallel algorithms in data mining in TCM.

Acknowledgements

Supported by Program for Changjiang Scholars and innovative Research Team in University (IRT0652, PCSIRT), China 863 project under grant 2006AA01A123, the National 973 Basic Research Program of China under grant No.2007CB310900, National Science Fund for Distinguished Young Scholars under grant No.60525202, J20060103, J20050710, the Defense Advanced Research Foundation of the General Armaments Department of the PLA under Grant No.9140A06060307JW0403.

References

1. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network Motifs: Simple Building Block of Complex Networks. Science 5594, 824–827 (2002)
2. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York (1979)
3. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: ACM OSDI (2004)
4. Kuramochi, M., Karypis, G.: Finding Frequent Patterns in a Large Sparse Graph. In: Data Mining and Knowledge Discovery, vol. 5810, pp. 243–271. Springer, Heidelberg (2005)
5. Yan, X., Han, J.: gSpan: Graph-based substructure pattern mining. In: 2002 IEEE International Conference on Data Mining, 2002. ICDM 2002. Proceedings, pp. 721–724. IEEE Press, Maebashi City (2002)
6. Inokucbi, A., Wasbio, T., Motoda, H.: Complete mining of frequent patterns from graphs: Mining graph data. Machine Learning 50(3), 321–354 (2003)
7. Hong, M., Zhou, H., Wang, W., Shi, B.: An efficient algorithm of frequent connected subgraph extraction. In: Whang, K.-Y., Jeon, J., Shim, K., Srivastava, J. (eds.) PAKDD 2003. LNCS, vol. 2637, pp. 40–51. Springer, Heidelberg (2003)
8. Yan, X., Hart, J.: CloseGraph: Mining closed frequent patterns. In: The 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003), pp. 286–295. ACM, Washington (2003)
9. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraph in the presence of isomorphism. In: 2003 International Conference on Data Mining (ICDM), Melbourne, pp. 549–552. IEEE, Florida (2003)
10. Gudes, E., Shimony, S.E., Vanetik, N.: Discovering frequent graph patterns using disjoint paths. IEEE Transactions on Knowledge and Data Engineering 18(11), 1441–1456 (2006)
11. Yoshida, K., Motoda, H., Indurkhya, N.: Graph-based induction as a unified learning framework. Journal of Applied Intelligence 4, 297–328 (1994)

12. Cook, J., Holder, L.: Substructure discovery using minimum description length and background knowledge. J. Artificial Intelligence Research, 231–255 (1994)
13. Schreiber, F., Schwöbbermeyer, H.: Frequent Concepts and Pattern Detection for the Analysis of Motifs in Networks. In: Priami, C., Merelli, E., Gonzalez, P., Omicini, A. (eds.) Transactions on Computational Systems Biology III. LNCS (LNBI), vol. 3737, pp. 89–104. Springer, Heidelberg (2005)
14. Chen, J., Hsu, W., Lee, M.-L., Ng, S.-K.: Nemofinder: dissecting genome-wide protein-protein interactions with meso-scale network motifs. In: KDD, pp. 106–115 (2006)
15. Chen, C., Yan, X., Zhu, F., Han, J.: gApprox: Mining frequent approximate patterns from a massive network. In: Perner, P. (ed.) ICDM 2007. LNCS (LNAI), vol. 4597, pp. 445–450. Springer, Heidelberg (2007)
16. Chu, C., Kim, S.K., Lin, Y., Yu, Y.Y., Bradski, G.: Map-Reduce for Machine Learning on Multicore. NIPS (2006)
17. Chang, E., Zhu, K., Wang, H., Bai, H., Li, J., Qiu, Z., Cui, H.: PSVM: Parallelizing Support Vector Machines on Distributed Computers. NIPS (2007)
18. Wu, Z., Zhou, X., Liu, B., Chen, J.: Text Mining for Finding Functional Community of Related Genes using TCM Knowledge. In: Boulicaut, J.-F., Esposito, F., Giannotti, F., Pedreschi, D. (eds.) PKDD 2004. LNCS (LNAI), vol. 3202, pp. 459–470. Springer, Heidelberg (2004)
19. Ying, T., Guo-fu, Y., Gui-bing, L., Jian-ying, C.: Mining Compatibility Rules from Irregular Chinese Traditional Medicine Database by Apriori Agorithm. Journal of Southwest Jiaotong University (English Edition) 15, 288–292 (2007)
20. Xuezhong, Z., Zhaohui, W.: Distributional Character Clustering for Chinese Text Categorization. In: Zhang, C., Guesgen, H.W., Yeap, W.-K. (eds.) PRICAI 2004. LNCS (LNAI), vol. 3157, pp. 575–584. Springer, Heidelberg (2004)
21. Xiao, H., Liang, X., Lu, P., Chan, C.: New method for analysis of Chinese herbal complex prescription and its application. Chinese Science Bulletin 44, 1164–1172 (1999)
22. Feng, Y., Wu, Z., Zhou, X., Zhou, Z., Fan, W.: Knowledge discovery in traditional Chinese medicine: State of the art and perspectives. Artificial Intelligence in Medicine. 38(3), 219–236 (2006)
23. Chang, Y.-H., Lin, H.-J., Li, W.-C.: Clinical evaluation of the traditional Chinese prescription Chi-Ju-Di-Huang-Wan for Dry Eye. Phytotherapy Research 19(4), 349–354 (2005)
24. Kuramochi, M., Karypis, G.: An efficient algorithm for discovering frequent subgraphs. Technical Report 02-026, Department of Computer Science, University of Minnesota (2002)
25. Fujing, D.: Prescription: for the Specialty of Chinese Traditional Medicine. Shanghai Publishing House of Science and Technology Press, Shanghai (2006)