SAHAD: Subgraph Analysis in Massive Networks Using Hadoop

Zhao Zhao*, Guanying Wang†, Ali R. Butt†, Maleq Khan*, V. S. Anil Kumar* and Madhav V. Marathe*
* *Network Dynamics and Simulation Science Laboratory, Virginia Tech, Blacksburg, VA, 24060, U.S.*
Email: {zhaozhao,maleq,akumar,mmarathe}@vbi.vt.edu
† *Department of Computer Science, Virginia Tech, Blacksburg, VA, 24060, U.S.*
Email: {wanggy,butta}@cs.vt.edu

***Abstract*—Relational subgraph analysis, e.g. finding labeled subgraphs in a network, which are isomorphic to a template, is a key problem in many graph related applications. It is computationally challenging for large networks and complex templates. In this paper, we develop SAHAD, an algorithm for relational subgraph analysis using Hadoop, in which the subgraph is in the form of a tree. SAHAD is able to solve a variety of problems closely related with subgraph isomorphism, including counting labeled/unlabeled subgraphs, finding supervised motifs, and computing graphlet frequency distribution. We prove that the worst case work complexity for SAHAD is asymptotically very close to that of the best sequential algorithm. On a mid-size cluster with about 40 compute nodes, SAHAD scales to networks with up to 9 million nodes and a quarter billion edges, and templates with up to 12 nodes. To the best of our knowledge, SAHAD is the first such Hadoop based subgraph/subtree analysis algorithm, and performs significantly better than prior approaches for very large graphs and templates. Another unique aspect is that SAHAD is also amenable to running quite easily on Amazon EC2, without needs for any system level optimization.**

***Keywords*-subgraph isomorphism, frequent subgraph, motif, graphlet frequency distribution, MapReduce, Hadoop**

I. INTRODUCTION

Subgraph isomorphism is a canonical problem in several applications, such as social network analysis [11], data mining [18], [7], [31], [30], fraud detection [5], chemical informatics [6], web information management [23] and bioinformatics [13], [20], where people are interested in finding subsets of nodes with specific labels or attributes and mutual relationships that match a specific template. For example, in financial networks (based on [7], [5]), in which the nodes are banks and individuals, and edges represent financial transactions, an investigator might be interested in specific transaction patterns from an individual to banks, e.g., through suspicious intermediaries to deflect attention (see Figure 1 for an example). In many bioinformatics applications, frequent subgraphs (referred to as "motifs") in protein-protein interaction networks (PPI) have been used to characterize the network, distinguish it from random networks and identify functional groups (e.g., [20], [9]). Thus, as discussed in a survey by Getoor [11], general subgraph mining poses fundamental problems in a number of applications.

Many variants of subgraph isomorphism problems have been studied, such as: finding the most frequent subgraph (e.g., [17]), counting specific subgraphs (e.g., [1], [14], [32]), detecting labeled queries, computing functions on the space of embeddings such as graphlet frequency distribution (e.g., [22]). In general, these are computationally very challenging problems. Given an arbitrary template of size k and a graph with n nodes, the best known rigorous result for the subgraph isomorphism problem is obtained by Eisenbrand *et al.* [10] with a running time of roughly $O(n^{\omega k/3})$ (which improves on the naive $O(n^k)$ time), where ω denotes the exponent of the best possible matrix multiplication algorithm. If the template has an independent set of size s, Vassilevska *et al.* [28] give an algorithm with an improved running time of $O(2^s n^{k-s+3} k^{O(1)})$; this is improved slightly by Kowaluk *et al.* [16]. When the template is a tree or has a bounded treewidth, Alon *et al.* [1] develop the color coding technique which is a randomized approximation algorithm with running time $O(k|E|2^k e^k \log(1/\delta)\frac{1}{\varepsilon^2})$, where ε and δ are error and confidence parameters, respectively.

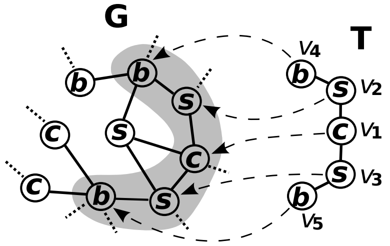

Figure 1. Here is an application of detecting subgraph isomorphism in financial networks (as in [7], [5]): G is an underlying graph whose nodes represent customers (c), suspicious (s) or banks (b). The template T is an abnormal transaction pattern, in which a customer v_1 uses suspicious intermediaries v_2 and v_3 to reach banks v_4 and v_5. The shaded subgraph shows a matching of the abnormal pattern.

A lot of practical heuristics have also been developed for various versions of these problems, especially for the frequent subgraph mining problem (as discussed later in Section II). An example is the "Apriori" method, which uses a level-wise exploration of the template [15], [17], for generating candidates for subgraphs at each level; these have been made to run faster by better pruning and exploration techniques, e.g., [17], [13], [30]. Other approaches in relational databases and data mining involve queries for specific labeled subgraphs, and have combined relational database techniques with careful depth-first exploration, e.g., [25], [24], [7]. Most of these approaches are sequential, and generally only scale to fairly small graphs and templates. It seems that parallelism is necessary to scale to much larger networks and templates.

In general, these approaches are notoriously hard to parallelize as it is very difficult to decompose the task into independent subtasks. It is not clear if candidate generation approaches [17], [13], [30] can be parallelized and scaled to large graphs and computing clusters. Two recent approaches for parallel algorithms, related to this work, are [7], [32]. The approach of Bröcheler *et al.* [7] requires a complex preprocessing and enumeration process, which has high end-to-end time, while the approach of [32] involves an MPI-based implementation with a very high communication overhead for larger templates. Two very recent papers [26], [21] develop MapReduce based algorithms for approximately counting the number of triangles with a work complexity bound of $O(|E|)$. It still remains an open problem to develop parallel algorithms for subgraph analysis with rigorous polynomial work complexity, which are implementable on heterogeneous computing resources. See Table I for a detailed comparison between different sequential and parallel approaches for subgraph isomorphism problems.

Our contributions. In this paper, we present SAHAD, a new algorithm for Subgraph Analysis using Hadoop, with rigorously provable polynomial work complexity for relational subgraph isomorphism problems in massive networks, in which the subgraph is in the form of a tree. It scales to very large graphs, and because of the Hadoop implementation, runs flexibly on a variety of computing resources, including Amazon EC2 cloud. Our specific contributions are discussed below.

1. SAHAD is the first MapReduce-based algorithm for solving a number of problems related with relational subgraphs on very large networks. It supports subgraphs in the form of any labeled tree. As discussed earlier, the only prior Hadoop based approaches have been on triangles [27], [21], [26] in very large networks, or more general subgraphs on relatively small networks [19]. Our main technical contribution is the development of a Hadoop version of the *color coding* algorithm of Alon *et al.* [1], [2], which is a (sequential) randomized approximation algorithm for subgraph counting. We prove that the work complexity of SAHAD is $O(k|E_G|2^{2k}e^k \log{(1/\delta)}\frac{1}{\varepsilon^2})$, which is more than the running time of the sequential algorithm of [1] by just a factor of 2^k.

2. We demonstrate our results on instances of the Erdös-Renyi random graph model and on synthetic social contact graphs for Miami city and Chicago city (with 52.7 million and 268.9 million edges, respectively), constructed using the methodology of [4]. We study the performance of counting unlabeled/labeled templates with up to 12 nodes. The total running times for templates with 12 nodes on Miami and Chicago networks are 15 and 35 minutes, respectively; note that these are the *total end-to-end* times, and do not require any additional pre-processing (unlike, e.g. [7]).

3. SAHAD is able to solve a variety of subgraph isomorphism problems, such as: (i) counting the number of embeddings of a given labeled/unlabeled template ; (ii) finding the most frequent subgraphs/motifs efficiently from a given set of candidate templates; and (iii) computing the graphlet frequency distribution. SAHAD is specifically suitable for computing on multiple templates, since they usually share common sub-templates such as edge, simple path or star, etc., which are only computed once.

4. SAHAD runs easily on heterogeneous computing resources, e.g., it scales well when we request up to 16 nodes on a medium size cluster with 32 cores per node. Our Hadoop based implementation is also amenable to running on public clouds, e.g., Amazon EC2 [3]. Except for a 10-node template which produces extremely large amount of data so as to incur the I/O bottle neck on the virtual disk of EC2, the performance of SAHAD on EC2 is almost the same as on the local cluster. This would enable researchers to perform useful queries even if they do not have access to large resources, such as those required to run previously proposed querying infrastructures. We believe this aspect is unique to SAHAD and lowers the barrier-to-entry for scientific researchers to utilize advanced computing resources.

Organization. We discuss the related work and some background in Sections II and III, respectively. The sequential color coding algorithm of [1] is discussed in Section IV. We discuss the details of SAHAD and its analysis in Section V, and extensions to other kinds of problems in Section VI. Our experimental results are discussed in Section VII, and we conclude the paper in Section VIII.

II. Related Work

A variety of different algorithms and heuristics have been developed for different domain specific versions of subgraph isomorphism problems. One version involves finding frequent subgraphs, and many approaches for this problem use the Apriori-method from frequent item set mining [15], [17], [11]. These approaches involve candidate generation during a breadth first search on the subset lattice and a determination of the support of item sets by subset test. A variety of optimizations have been developed, e.g., using a DFS order to avoid the cost of candidate generation [13], [30] or pruning techniques, e.g., [17]. A related problem is that of computing the "graphlet frequency distribution", which generalizes the degree distribution [22].

Another class of results for frequent subgraph finding is based on the powerful technique of "color coding" (which also forms the basis of our paper), e.g., [1], [14], [32], which

Table I
SOME OF THE REPRESENTATIVE RESEARCHES ON SUBGRAPH ISOMORPHISM PROBLEMS

Problem	**Reference**	**Network**	**Template**	**Running Time**	**Computing Environment**
Subgraph enumeration	[12]	PPIs with 10^3 nodes	7 nodes	1-2 hours	sequential
Graphlet frequency distribution	[22]	PPIs with 10^3 nodes	29 graphlets with 3 to 5 nodes each	up to 10 days	sequential
Motif counting	[1]	PPIs with 10^3 nodes	treelets with up to 10 nodes	12 hours	parallel on 8 cores
Labeled subgraph querying	[7]	social network with 778M edges	upto 6 nodes/23 edges	10.5 hours to partition, seconds to query	16 nodes with 8 core on each
subgraph isomorphism problems	SAHAD	synthetic networks with up to 269M edges	treelets with up to 12 nodes	less than 35 minutes	32 cores per node, up to 16 nodes

has been used for approximating the number of embeddings of templates that are trees or "tree-like".

In [1], Alon *et al.* use color coding to compute the distribution of treelets with sizes 8, 9 and 10, on the protein-protein interaction networks of Yeast. The color coding technique is further explored and improved in [14], in terms of worst case performance and practical considerations. E.g., by increasing the number of colors, they speed up the color coding algorithm with up to 2 orders of magnitude. They also reduce the memory usage for minimum weight paths finding, by carefully removing unsatisfied candidates, and reducing the color set storage.

Most of these approaches in bioinformatics applications involve small templates, and have only been scaled to relatively small graphs with at most 10^4 nodes (apart from [32], which shows scaling to much larger graphs by means of a parallel implementation). Other settings in relational databases and data mining have involved queries for specific labeled subgraphs. Some of the approaches for these problems have combined relational database techniques, based on careful indexing and translation of queries, with such depth-first exploration strategy that is distributed over different partitions of the graph e.g., [25], [24], [7], and scale to very large graphs. For instance, Bröcheler *et al.* [7] demonstrate queries with up to 7-node templates on graphs with over half a billion edges, by carefully partitioning the massive network using minimum edge cuts, and distributing the partitions on 15 computing nodes.

MapReduce/Hadoop has become a popular approach for parallel computing, and graph algorithms (e.g., [27], [21], [26], [19] for subgraph enumeration) are being developed using this approach. Among these, [27], [21], [26] develop algorithms for enumerating triangles and give worst case $O(|E|)$ work complexity bounds for this problem, using Hadoop. Liu *et al.* [19] develop heuristics based on MapReduce for subgraph isomorphism, but only scale to moderate size graphs. Developing Hadoop based algorithms for enumeration of subgraphs other than triangles with rigorous bounds is an open problem.

III. BACKGROUND

A. Labeled subgraph isomorphism problems and extensions

We consider labeled graphs $G = (V_G, E_G, L, \ell_G)$, where V_G and E_G are the sets of nodes and edges, L is a set of labels and $\ell_G : V \rightarrow L$ is a labeling on the nodes. A graph $H = (V_H, E_H, L, \ell_H)$ is a *non-induced subgraph* of G if we have $V_H \subseteq V_G$ and $E_H \subseteq E_G$. We say that a template graph $T = (V_T, E_T, L, \ell_T)$ is isomorphic to a non-induced subgraph $H = (V_H, E_H, L, \ell_H)$ of G if there exists a bijection $f : V_T \rightarrow V_H$ such that: (i) for each $(u, v) \in E_T$, we have $(f(u), f(v)) \in E_H$, and (ii) for each $v \in V_T$, we have $\ell_T(v) = \ell_H(f(v))$. We also call H a non-induced embedding of T, e.g., Figure 1 shows a non-induced embedding of template T. In this paper, we assume T is a tree.

We consider the following variations of subgraph isomorphism problems: (i) *Subgraph Counting:* this is the most common problem in subgraph mining. The problem is to count the number of embeddings of a given template T in a network G; (ii) *Supervised Motif Finding:* this problem involves counting the number of embeddings of multiple subgraphs and finding the ones with abnormal higher frequencies than those in random networks; (iii) *Graphlet Frequency Distribution:* this is an extension of degree distribution, and describes the number of nodes that have the same number of "graphlet" adjacent to them, where graphlet is an alias for template, or a treelet in this paper.

Let $emb(T, G)$ denote the number of all embeddings of template T in graph G, we say that an algorithm $\mathcal{A}$ produces an (ε, δ)-*approximation* to $emb(T, G)$, if the estimate Z produced by $\mathcal{A}$ satisfies: $\Pr[|Z - emb(T, G)| > \varepsilon \cdot emb(T, G)] \leq 2\delta$; in other words, $\mathcal{A}$ is required to produce an estimate that is close to $emb(T, G)$, with high probability.

B. MapReduce and Hadoop

MapReduce is an emerging computation model. It breaks a problem into distinct *map* tasks for distribution to multiple computing entities, and *reduce* tasks for merging the results of individual computing entities to produce the final result [8]. Users employ the model by defining application specific map and reduce functions, and the framework then takes care of managing and allocating appropriate resources to perform the tasks.

The MapReduce model works with data expressed as key-value pairs $< k, v >$. An application first takes $< k_1, v_1 >$ pairs as input to the map function, in which one or more $< k_2, v_2 >$ pairs are produced for each input pair. Then the MapReduce re-organizes all $< k_2, v_2 >$ pairs and puts together all v_2s that are associated with the same k_2, which are then processed by a reduce function.

The MapReduce model requires that input data be divided into small independent pieces that are processed independently in parallel without communication with other tasks. Provided a suitable large-scale problem, the level of parallelism is only limited by available resources, i.e., resources can be fully utilized. New MapReduce-friendly algorithms can be designed for many problems via innovative partitioning or hierarchical approaches, as is the case in this paper.

Hadoop [29] is an open-sourced implementation of MapReduce. Due to the reliability and scalability in handling vast amount of computation in parallel, Hadoop is becoming a *de facto* solution for large parallel computing tasks. In addition to supporting MapReduce, Hadoop features in the Hadoop Distributed File System(HDFS), which provides more data locality and reliability.

IV. The Sequential Algorithm: Color Coding

We first briefly discuss the main ideas of the (sequential) color coding technique of [2], to make the discussion of the parallel Hadoop version easier. As discussed in Section III, the goal is to count the number of embeddings of template T in graph G. The color coding technique consists of the following two key ideas: (i) we consider a coloring of G using k colors, where $k \geq |V_T|$, and compute the number of "colorful" embeddings of T in G (where a colorful embedding is one in which each node has a distinct color)—the main insight is that this problem can be solved by dynamic programming (in contrast to the original problem of counting all embeddings), and (ii) if the coloring of G is done randomly, the expected number of colorful embeddings equals the total number of embeddings times $\frac{m!\binom{k}{m}}{k^m}$.

Following the notation in [1], we pick a node ρ as the root of T, and let $T(\rho)$ to be the rooted tree. We define $C(v, T_i(\rho_i), S_i)$ to be the number of colorful embeddings of T_i with node $v \in V_G$ mapped to the root ρ_i, and using the color set S_i, where $|V_{T_i}| = |S_i|$. Figure 2 shows an example of the quantities $C(v, T_i(\rho_i), S_i)$, and a recurrence relation for computing them, using the counts corresponding to smaller trees. The complete algorithm (described below) involves (i) partitioning the template into sub-templates, and (ii) a dynamic program to compute the colorful counts.

1) Partitioning the template: The template is partitioned into sub-templates using Algorithm 1 and illustrated in the example in Figure 3. We denote the set of template T and all of its sub-templates T_i by $\mathcal{T}$. The dynamic program computes the counts for the sub-templates in a bottom-up manner.

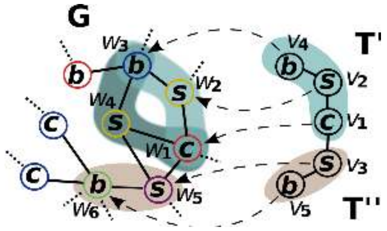

Figure 2. The example shows one step of the dynamic programming in color coding. T in Figure 1 is split into T' and T''. To count $C(w_1, T(v_1), S)$, or the number of embeddings of $T(v_1)$ rooted at w_1, using color set $S = \{\underline{r}ed, \underline{y}ellow, \underline{b}lue, \underline{p}urple, \underline{g}reen\}$, we first obtain $C(w_1, T'(v_1), \{r, y, b\}) = 2$ and $C(w_5, T''(v_3), \{p, g\}) = 1$. Then, $C(w_1, T(v_1), S) = C(w_1, T'(v_1), \{r, y, b\})C(w_5, T''(v_3), \{p, g\}) = 2$. The embeddings of T are subgraphs with nodes $\{w_3, w_4, w_1, w_5, w_6\}$ and $\{w_3, w_2, w_1, w_5, w_6\}$.

Algorithm 1 *Partition($T(\rho)$)*

```
if $T \notin \mathcal{T}$ then
    if $|V_T| = 1$ then
        $\mathcal{T} \leftarrow T$
    else
        Add $T$ to $\mathcal{T}$
        Pick $\tau \in N(\rho)$, the set of the neighbors of $\rho$, and partition $T$ into two sub-templates by cutting the edge $(\rho, \tau)$
        Let $T'$ be the sub-template containing $\rho$ (name as active child) and $T''$ the other (name as passive child)
        Partition($T'(\rho)$)
        Partition($T''(\tau)$)
```

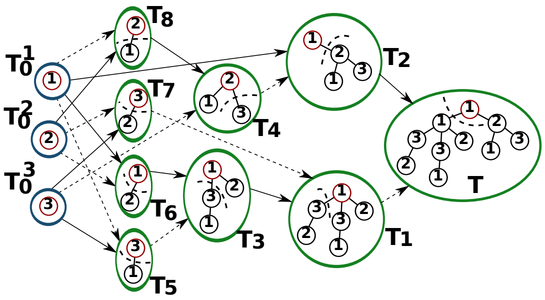

Figure 3. Here shows the template partitioning scheme discussed in Algorithm 1. The node are labeled 1, 2 or 3. $T_1, \ldots, T_8$ and T_0^1, T_0^2, T_0^3 are the sub-templates, in which T_0^1, T_0^2, T_0^3 are unit templates, which form the base case. T_i uses the counts of the *active child* T_i' and the *passive child* T_i'' to compute its counts, which have solid and dash arrows into it, respectively. The root of each sub-template is marked by a red circle.

2) The algorithm: The dynamic programming algorithm is described below (refer to [1] for more details).

I. Let $S = \{s_1, s_2, ..., s_k\}$ be a set of k colors, $k \geq m$, where $m = |V_T|$. Color each node $v \in V_G$ with $s_i \in S$, uniformly at random.

II. Use Algorithm 1 to construct set $\mathcal{T}$.

III. Perform the following dynamic program. For each node $v \in V_G$, consider sub-templates in increasing order of size.

a. If T_i consists of a single node ρ_i with color s_i, we let $C(v, T_i(\rho_i), \{s_i\}) = 1$ if $\ell(v) = \ell(v')$.

b. If T_i consists of children T_i', T_i'', for each subset $S_i \subseteq S$ of size $|S_i| = |V_{T_i}|$, we compute $C(v, T_i(\rho_i), S_i)$ in the following manner:

$$\begin{aligned} C(v, T_i(\rho_i), S_i) = \tfrac{1}{d} \sum_u \sum C(v, T_i'(\rho_i), S_i') \cdot \\ C(u, T_i''(\tau_i), S_i'') \end{aligned} \tag{1}$$

where the first summation is over all $u \in N(v)$ and the second summation is over all S_i', S_i'' such that $S_i' \cap S_i'' = \emptyset, S_i' \cup S_i'' = S_i$. Here d is the over-counting factor and is equal to one plus the number of siblings of τ_i which are roots of subtrees isomorphic to $T_i''(\tau_i)$.

c. Since the probability that a matched subgraph is colorful is $P = \frac{m!\binom{k}{m}}{k^m}$, we can approximate the counts of the embeddings of T rooted at v as:

$$U(v, T(\rho)) = \tfrac{1}{P} \textstyle\sum_{\forall S_m \subseteq S} C(v, T(\rho), S_m) \tag{2}$$

d. The number of embeddings of T is computed as:

$$Y = \tfrac{1}{q} \textstyle\sum_{\forall v \in V_G} U(v, T(\rho)) \tag{3}$$

Here q is the number of node $\rho' \in V_T$ such that T is isomorphic to itself when ρ is mapped to ρ'.

IV. We repeat the above steps $N = O(\frac{e^m \log(1/\delta)}{\varepsilon^2})$ times, and partition N samples $Y_1, ..., Y_N$ into $t = O(\log(1/\delta))$ sets. Let Z_j be the average of set j. Output the median of $Z_1, ..., Z_t$.

The analysis of the above algorithm, summarized below, follows from [1].

Theorem 4.1: For any $\varepsilon, \delta > 0$, the above algorithm gives an (ε, δ)-approximation to the number of embeddings of a tree T in time $O(k|E_G|2^k e^k \log(1/\delta) \frac{1}{\varepsilon^2})$.

V. Algorithm SAHad for labeled subgraph analysis

We now discuss the details of SAHad (Algorithm 2 below), and the various jobs— "*Colorer*", "*Counter*" and "*Analyzer*". As discussed in Section III, the input consists of a labeled template $T = (V_T, E_T, L, \ell_T)$ and a labeled graph $G = (V_G, E_G, L, \ell_G)$. SAHad is designed to solve a variety of labeled subgraph analysis problems.

A. *Overview of* SAHad *and its jobs*

In SAHad, we configure a distinct Hadoop job for each $T_i \in \mathcal{T}$, to compute $C(v, T_i(\rho_i), S_i)$ for $\forall v \in V_G$. There are three types of jobs: *Colorer*, *Counter* and *Analyzer*. *Colorer* jobs compute the above quantities for unit templates $T_0^1, \ldots, T_0^j$, and *Counter* jobs compute those for other templates. *Counter* for T_i is started as soon as the jobs for the two children T_i' and T_i'' are completed. For specific T_i, we use another job – *Analyzer* – to compute further results including total counts, graphlet distributions, etc., using the quantities $C(v, T_i(\rho_i), S_i)$ computed by its *Counter*. We discuss each type of the job in details in the following sections. Algorithm 2 is an overview of SAHad.

Algorithm 2 *Overview of* SAHad

```
Partition $T$ using Algorithm 1
Configure *Colorers* for $T_0^1, \ldots, T_0^j$, *Counters* for $T, T_1, \ldots, T_i$, and *Analyzers*.
Run *Colorers* for $T_0^1, \ldots, T_0^j$.
Run *Counter* for $T_i$ to compute $C(v, T_i(\rho_i), S_i)$, if the dependent jobs for $T_i'$ and $T_i''$ are completed.
Run *Analyzer* to compute additional functions.
```

B. *Colorer and color set representation*

We use a 4 byte integer to represent a color set which can store up to 32 colors. Each bit in the binary string represents a color. Bit "1" denotes that the correspondent color exists in the color set. With this data structure, the intersection and union of two color sets can be easily represented by bitwise "AND" and "OR" operations, respectively.

The *Colorer* job only consists of a map function. A *Colorer* for T_0^j reads G from HDFS and assigns each node $v \in V_G$ with a random color chosen from the color set S. It only outputs entries for the nodes whose labels are the same as the label of the single node ρ_0^j of T_0^j.

Algorithm 3 *Colorer.Mapper(line, $T_0^j(\rho_0^j)$)*

```
$(v, NL_v, l(v)) \leftarrow Parse(line)$
**if** $l(v) = l(\rho_0^j)$ **then**
    Pick $s_i \in \{s_1, \ldots, s_k\}$ uniformly at random
    $\triangleright$ Set $C(v, T_0^j, \{s_i\}) = 1$
    $CCP_v \leftarrow \{(\{s_i\}, 1)\}$
    Collect($\underline{key \leftarrow v}, \underline{value \leftarrow CCP_v, NL_v}$)
```

G is stored in HDFS in such a way, that each line of the file records a node $v \in V_G$, a list of its neighbor-label pairs NL_v, and $l(v)$. *Colorer.Mapper* takes each line as input and output a key-value pair for those node v whose label equals $l(\rho_0^j)$. In the output, the key is the node v, and the value is CCP_v and NL_v. We hereby introduce the notation of NL_v and CCP_v below:

- NL_v: a set of neighbor-label pairs, $\{(u, l(u)) | \forall u \in N(v)\}$
- CCP_v: a set of color count pairs, $\{(S_i, C(v, T_i(\rho_i), S_i)) | C(v, T_i(\rho_i), S_i) \neq 0\}$.

Note that in *Colorer*, CCP_v only contains one element $(\{s_i\}, 1)$, where s_i is the random color picked for v and 1 is the count of the unit template rooted at v. We don't record the label of the node in the output since only those nodes that have the same label as the unit template are in the output.

C. Counter

Counter for T_i takes (v, CCP_v, NL_v) for T_i' and T_i'' for $\forall v \in V_G$ as input, and outputs v, CCP_v, NL_v for T_i for $\forall v \in V_G$, which implements the dynamic programming in Equation 1. *Counter.Mapper* and *Counter.Reducer* are described in Algorithm 4 and 5, respectively.

Algorithm 4 *Counter.Mapper(line, $T_i(\rho_i)$)*

```
$(v, CCP_v, NL_v) \leftarrow Parse(line)$
if $(v, CCP_v, NL_v)$ is for $T_i'$ then
    $\triangleright$ output a single key-value pair, with key being $v$
    $\triangleright$ $flag'$ denotes active child
    $Collect(\underline{key \leftarrow v}, \underline{value \leftarrow CCP_v, flag', NL_v})$
else
    $\triangleright$ output multiple key-value pairs, each with key
    being a neighbor $u \in N(v)$
    for each $(u, l(u)) \in NL_v$ do
        if $l(u) = l(\rho_i')$ then
            $\triangleright$ $u$ is a potential root of $T_i'$
            $\triangleright$ $flag''$ denotes passive child
            $Collect(\underline{key \leftarrow u}, \underline{value \leftarrow CCP_v, flag''})$
```

In Algorithm 4, the map function differentiates whether the input colorful count is from the active child T_i' or the passive child T_i''. If it is from active child, CCP_v are mapped out with the key v. Otherwise, multiple lines are mapped out, each with a key from $\forall u \in N(v)$. Then, in *Counter.Reducer*, the counts of the colorful embeddings of T_i' rooted at v and those of T_i'' rooted at $u \in N(v)$ are reduced together, which are finally aggregated using Equation 1 and factorized by the over counting factor d_{T_i}. The output is written back to HDFS.

Algorithm 5 *Counter.Reducer(key, values)*

```
$CCP_v \leftarrow \emptyset$
$activeCCP_v \leftarrow \emptyset$
$passiveCCP_v \leftarrow \emptyset$
$v \leftarrow key$
for each $value \in values$ do
    $data \leftarrow Parse(value)$
    if $data[1] = flag'$ then
        $\triangleright$ $CCP_v$ from active child
        $activeCCP_v \leftarrow data[0]$
        $NL_v \leftarrow data[2]$
    else
        $\triangleright$ $CCP_v$ from passive child, $passiveCCP_v$ con-
        tains multiple $CCP_v$ from all $v$'s neighbors
        $passiveCCP_v.append(data[0])$
for each $(S_a, C_a) \in activeCCP_v$ do
    $\triangleright$ $C_a = C(v, T_i', S_a)$
    for each $array \in passiveCCP_v$ do
        for each $(S_b, C_b) \in array$ do
            $\triangleright$ $C_b = C(u, T_i'', S_b)$
            if $S_a \cap S_b = \emptyset$ then
                $S_c \leftarrow S_a \cup S_b$
                $\triangleright$ Add $C(v, T_i', S_a)C(u, T_i'', S_b)$ to $C(v, T_i, S_c)$
                if $S_c \notin CCP_v.colorset$ then
                    $CCP_v[S_c] \leftarrow C_a \cdot C_b$
                else
                    $CCP_v[S_c] \leftarrow CCP_v[S_c] + C_a \cdot C_b$
for $S_c \in CCP_v.colorset$ do
    $CCP_v[S_c] = CCP_v[S_c]/d_{T_i}$
if $CCP_v \neq \emptyset$ then
    $Collect(\underline{key \leftarrow v}, \underline{value \leftarrow CCP_v, NL_v})$
```

Algorithm 6 *Counts-Analyzer.Mapper(line)*

```
$(v, CCP_v, NL_v) \leftarrow Parse(line)$
$count \leftarrow 0$
for $\forall(S, C) \in CCP_v$ do
    $count = count + C$
$Collect(\underline{key \leftarrow \text{“}TotalCounts\text{”}}, \underline{value \leftarrow count})$
```

D. Analyzers

We use *Analyzers* to compute measurements that are related with subgraph isomorphism problems, including total counts, graphlet frequency distribution, etc. Algorithm 6 and 7 show an instance of computing the total counts of the template's embeddings.

Algorithm 6 computes the colorful counts of the template T rooted at v, i.e., $\sum_{\forall S_T \subseteq S} C(v, T(\rho), S)$ and output a key-value pair with the key as "TotalCounts". Then the counts are aggregated in Algorithm 7 and appropriately factored to obtain the total counts. Refer to Section IV-2 for the explanations on the factor q.

E. Performance analysis of SAHAD's jobs

We discuss various bounds on the complexity of each Hadoop jobs below.

Lemma 5.1: The sizes of the input and output of the *Colorer* are both $O(|E_G|)$.

Proof: Colorer only has the *Mapper*. The input consists of $(v, NL_v, l(v))$ for $\forall v \in V_G$, where the size of NL_v equals to the number of neighbors of v. Therefore, the size of the input is $O(|E_G|)$. Similarly, the size of the output is

Algorithm 7 *Counts-Analyzer.Reducer(key, values)*

```
$count \leftarrow 0$
$k \leftarrow$ size of the color set
$m \leftarrow$ template size
$P = \frac{m!\binom{k}{m}}{k^m}$
for $\forall C \in values$ do
    $count = count + C$
$count = \frac{count}{P \cdot q}$
$Collect(\underline{key \leftarrow \text{“}TotalCounts\text{”}}, \underline{value \leftarrow count})$
```

also $O(|E_G|)$, since it consists of $< v, (CCP_v, NL_v) >$ for $\forall v \in V_G$.

■

Lemma 5.2: For a template T_i, suppose the sizes of the two sub-templates T_i' and T_i'' are m' and m'', respectively, the sizes of the input, output, and work complexity corresponding to a node v are given below:

- The sizes of the input and output of *Counter.Mapper* are $O(\binom{k}{m'} + \binom{k}{m''} + d(v))$ and $O(\binom{k}{m''}d(v))$, respectively.
- The size of the input to *Counter.Reducer* is $O(\binom{k}{m''}d(v))$, and the work complexity is $O(\binom{k}{m'}\binom{k}{m''}d(v))$.

Proof: For a node v, the input to *Counter.Mapper* involves the corresponding CCP_vs for T_i' and T_i'', and NL_v, which together have size $O(\binom{k}{m'} + \binom{k}{m''} + d(v))$. If the input is from T_i'', *Counter.Mapper* generates multiple key-value pairs for a node v, in which each key-value pair is correspondent to a node $u \in N(v)$. Therefore, the output has size $O(\binom{k}{m''}d(v))$.

For a given v, the input to *Counter.Reducer* is the combination of the above, and is therefore, $O(\binom{k}{m''}d(v))$. For v and each neighbor $u \in N(v)$, *Counter.Reducer* aggregates every pair of (S_a, C_a) in the CCP_v corresponding to v, and (S_b, C_b) corresponding to u, which leads to a complexity of $O(\binom{k}{m'}\binom{k}{m''}d(v))$.

■

Lemma 5.3: The total work complexity of SAHAD is $O(k|E_G|2^{2k}e^k \log(1/\delta)\frac{1}{\varepsilon^2})$.

Proof: The overall complexity of the *Colorer* and *Analyzer* is $O(n)$ and $O(n \cdot \binom{k}{m})$, respectively. For each node v and template $T_i \in \mathcal{T}$, the work complexity of the *Counter* is $O(\binom{k}{m'}\binom{k}{m''}d(v))$. Since $|\mathcal{T}| \leq k$, the total work, over all nodes and templates is at most

$$O(\sum_{v,T_i}\binom{k}{m'}\binom{k}{m''}d(v)) = O(\sum_{v} k2^{2k}d(v)) = O(k|E_G|2^{2k}) \tag{4}$$

■

Since $O(e^k \log(1/\delta)\frac{1}{\varepsilon^2})$ iterations are performed in order to get the (ε, δ)-approximation, the total work complexity is as shown in the lemma.

VI. Extensions of SAHad: Variations of Subgraph Isomorphism Problems

So far we have discussed the basic framework of SAHAD. We have also discussed how to compute the total number of subgraph embeddings in an instance of *Analyzer* given in Section V-D. We now discuss a set of problems that are closely related with the subgraph isomorphism problem which can be computed by SAHAD, including finding supervised motif and computing graphlet frequency distribution.

Note that SAHAD is specifically suitable for computing on multiple templates if they have common sub-templates, since those common sub-templates only need to be computed once. This is the case in many problems, where common sub-templates such as single node, edge, or simple paths are shared.

A. Supervised Motif Finding

Motifs of a real-world network are specific templates whose embeddings occurring with much higher frequencies then in random networks and are referred as building blocks for networks. They have been found in many real-world networks[20]. SAHAD can reduce the computational cost for a group of templates since the common sub-templates are only computed once, therefore is amenable to be applied in supervised motif finding. Figure 4 shows the sub-templates' dependency network of a group of unlabeled subgraphs.

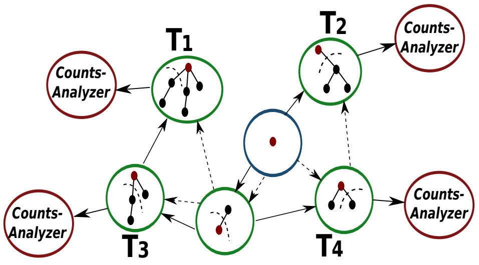

Figure 4. Here shows the dependency network for a group of templates from which we want to find the motif. We can obtain the counts of the embeddings of $T_1, \ldots, T_4$ in network G and a comparable random network. By comparing the counts of a template's embeddings in G to the random network, we are able to find the motif.

B. Graphlet Frequency Distribution

Graphlet frequency distribution has been proposed as a way of measuring the similarity of protein-protein networks [22], where common properties such as degree distribution, diameter, etc., may not suffice. Unlike "motifs", graphlet frequency distribution is computed on all selected small subgraphs regardless of whether they appear frequently or not.

Graphlet frequency distribution $D(i, T)$ measures the number of nodes from which i graphlets T are touched on. The number of graphlet touched on a single node v can be computed using a number of counts

$C(v,T(\rho_1),S), C(v,T(\rho_2),S),\ldots,C(v,T(\rho_i),S)$. E.g., in Figure 5, the graphlet frequency distribution of template 5-1 is computed by aggregating the counts of templates 5-1-1, 5-1-2, and 5-1-3.

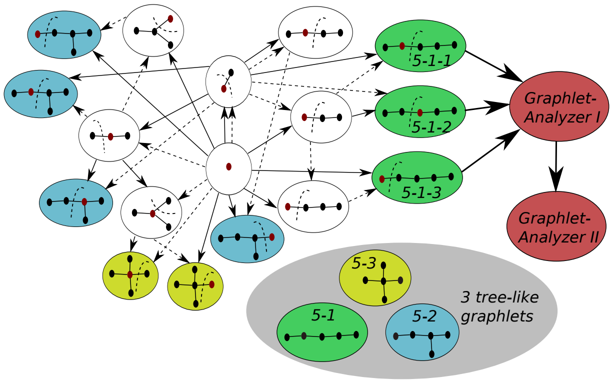

Figure 5. Here shows all the sub-templates needed for computing graphlet frequency distribution on template 5-1, 5-2 and 5-3. E.g., for template 5-1, one needs to obtain the counts $C(v, T(\rho), S_T)$ chosen multiple ρs, denoted as 5-1-1, 5-1-2 and 5-1-3. The root of a subgraph is marked red.

Algorithm 8 to 11 show the mappers and reducers for the two jobs corresponding to computing graphlet frequency distribution.

Algorithm 8 *Graphlet-AnalyzerI.Mapper(line)*

```
$T \leftarrow GetTemplateName(line)$
$P = \frac{m!\binom{k}{m}}{k^m}$
$(v, CCP_v, NL_v) \leftarrow Parse(line)$
$count \leftarrow 0$
for $\forall (s,c) \in CCP_v$ do
    $count = count + c$
$count = \frac{count}{P}$
$Collect(\underline{key \leftarrow v}, \underline{value \leftarrow count})$
```

Algorithm 9 *Graphlet-AnalyzerI.Reducer(key, values)*

```
$count \leftarrow 0$
for $\forall value \in values$ do
    $count = count + value$
$Collect(\underline{key \leftarrow v}, \underline{value \leftarrow count})$
```

Algorithm 10 *Graphlet-AnalyzerII.Mapper(line)*

```
$(v, count) \leftarrow Parse(line)$
$Collect(\underline{key \leftarrow count}, \underline{value \leftarrow 1})$
```

VII. Experiments

In this section, we evaluate various aspects of SAHAD's performance. Our main conclusions are summarized below.

Algorithm 11 *Graphlet-AnalyzerII.Reducer(key, values)*

```
$freq \leftarrow 0$
for $\forall value \in values$ do
    $freq = freq + 1$
$Collect(\underline{key \leftarrow key}, \underline{value \leftarrow freq})$
```

Table II summarizes the different experiments we perform, which are discussed in greater details later.

1. Approximation bounds: While the worst case bounds on the algorithm imply $O(e^k \log(1/\delta)\frac{1}{\varepsilon^2})$ rounds to get an (ε, δ)-approximation (see Theorem 4.1 and Lemma 5.3), in practice, we find that far fewer iterations are needed.

2. System performance: We run SAHAD on a diverse set of computing resources, including the publicly available Amazon EC2 cloud. We find SAHAD scales well with the number of nodes, and disk I/O is one of the main bottlenecks. *We posit that employing multiple disks per node (a rising trend in Hadoop) or using I/O caching will help mitigate this bottleneck and boost performance even further.*

3. Performance of SAHAD on various queries: We evaluate SAHAD on templates with sizes ranging from 5 to 12. We find labeled queries are significantly faster than unlabeled ones, and the overall running time is under 35 minutes for these queries on our computing cluster (described below). We also get comparable performance on EC2.

A. Datasets and Computing Environment

We use synthetic social contact networks for our experiments from [4] for Miami and Chicago cities. We consider demographic labels – {kid, youth, adult, senior} (based on the age) and gender for individuals. We also run experiments on the $G(n, p)$ model (denoted GNP100), with n nodes and each pair connected with probability p, and randomly assigned node labels. Table III summarizes the characteristics of the 3 networks.

Table III
Networks used in the experiments

Network	No. of Nodes(in million)	No. of Edges(in million)
Miami	2.1	52.7
Chicago	9.0	268.9
GNP100	0.1	1.0

The templates we use in the experiments are shown in Figure 6. The templates vary in size from 5 to 12 nodes, in which *U5-1,...U10-1* are the unlabeled templates and *L7-1 ,L10-1* and *L12-1* are the labeled templates. In the labels, *m*, *f*, *k*, *y*, *a* and *s* stand for *male*, *female*, *kid*, *youth*, *adult* and *senior*, respectively.

Table II

SUMMARY OF THE EXPERIMENT RESULTS (REFER TO SECTION VII-A FOR THE TERMINOLOGIES IN THIS TABLE)

Experiment	Computing resource	Template & Network	Key Observations
Approximation bounds	Athena	U7-1 & GNP100	error well below 0.5%
Impact of the number of data nodes	Athena	U10-1 & Miami, GNP100	scale from 4 hours to 30 minutes with data nodes from 3 to 13
Impact of the number of concurrent reducers	Athena & EC2	U10-1 & Miami	performance worsen on Athena
Impact of the number of concurrent mappers	Athena & EC2	U10-1 & Miami	no apparent performance change
Unlabeled/labeled templates counting	Athena & EC2	templates from Figure 6 and networks from Table III	all tasks complete in less than 35 minutes
Graphlet frequency distribution	Athena	U5-1 & Miami,Chicago	complete in less than 35 minutes

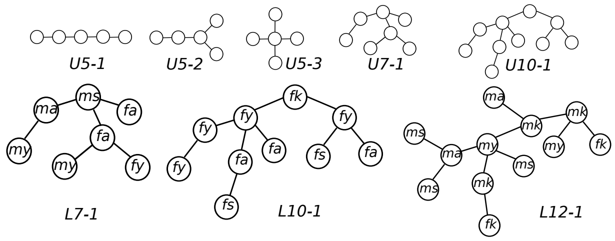

Figure 6. Templates used in the experiments.

We use a computing cluster, called **Athena**, with 42 computing nodes and a large RAM memory footprint. Each node has a quad-socket AMD 2.3GHz Magny Cour 8 Core Processor, i.e., 32 cores per node or 1344 cores in total, and 64 GB RAM(12.4 TFLOP peak). The local disk available on each node is $750GB$. Therefore, we can have maximum $31.5TB$ storage for the HDFS. In most of our experiments, we use up to 16 nodes, which give up to $12TB$ capacity for the computation. Although the number of cores and RAM capacity on each node can support a large number of mappers/reducers, the availability of a single disk on each node limits aggregate I/O bandwidth of all parallel processes on each node. To make it worse, aggregate I/O bandwidth of parallel processes doing sequential I/O could result in many extra disk seeks and hurt overall performance. Therefore, disk bandwidth is the bottleneck for more parallelism in each node. This limitation is further discussed in section VII-C.

We also use the public Amazon Elastic Computing Cloud (**EC2**) for some of our experiments. EC2 enables customers to instantly get cheap yet powerful computing resources, and start computing business with no upfront cost for hardware. We allocated 4 "High-CPU Extra-Large" instances from EC2. Each instance has 8 cores, 7 GB RAM, and two 250 GB virtual disks (Elastic Block Store Volume).

B. Approximation bounds

As discussed in Section III, the color coding algorithm averages the estimates over multiple iterations. In Figure 7, we show that the approximation error is below 0.5% for the template *U7-1* for the GNP100 graph, even for one iteration. The figure also plots the results based on using more than 7 colors, which can sometimes improve the running time, as discussed in [14]. In the rest of the experiments, we only use the estimate from one iteration, because of this result. The error for i iterations is computed using $\frac{|(\sum_i Z_i)/i - emb(T,G)|}{emb(T,G)}$.

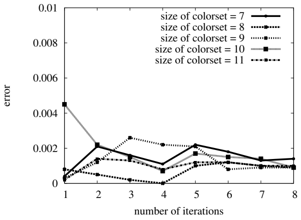

Figure 7. Approximation error in counting *U7-1* on GNP100.

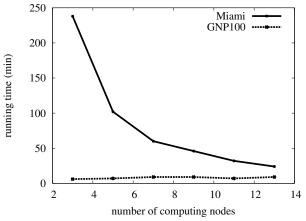

Figure 8. Running time for counting *U10-1* vs number of computing nodes.

C. Performance analysis of SAHAD

We now study how the running time is affected by the number of total computing nodes and number of reducers/mappers per node. We carry out 3 sets of experiments: (i) how the total running time scales with the number of computing nodes; (ii) how the running time is affected by varying assignment of mappers/reducers per node; and (iii) how the sizes of the output files vary in terms of sub-templates.

1) Varying number of computing nodes: Figure 8 shows that the running time for Miami reduces from over 200 minutes to less than 30 minutes when the number of computing nodes increases from 3 to 13. However, the curve for GNP100 does not show good scaling. The reason is that the actual computation for GNP100 only consumes a small portion of the running time, and there are overheads from managing the mappers/reducers. In other words, the curve for GNP100 shows a lower bound on the running time in SAHAD.

2) Varying number of mappers/reducers per node:

a. Varying number of reducers per node

Figure 9 and 10 show the running time on Athena when we vary the number of reducers per node. Here we fix the number of nodes to be 16 and the number of mappers per node to be 4. We find that running 3 reducers concurrently on each node minimizes the total running time. From Figure 10 we find that though increasing the number of reducers per node can reduce the time for the Reduce stage for a single job, the running time increases sharply in Map and Shuffle stage. As a result, the total running time increases with the number of reducers. It is because of the I/O bottleneck for concurrent accessing on Athena, since Athena has only 1 disk per node. This phenomenon does not show up on EC2, as seen from Figure 14, which indicates that EC2 is more optimized towards concurrent disk accessing for cloud usage.

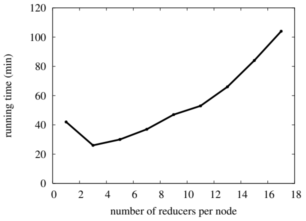

Figure 9. Total running time versus number of reducers per node.

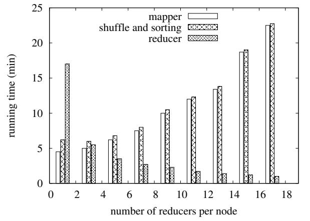

Figure 10. Running time of different job stages versus the number of reducers per node.

b. Varying number of mappers per node

Figure 11 and 12 show the running time on Athena when we vary the number of mappers per node while fixing the number of reducers as 7 per node. We find that varying the number of mappers per node does not affect the performance of SAHAD. This is also validated in EC2, as shown in Figure 13.

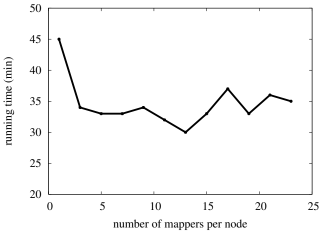

Figure 11. Total running time versus the number of mappers per node.

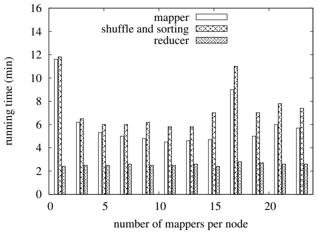

Figure 12. Running time of different job stages versus the number of mappers per node.

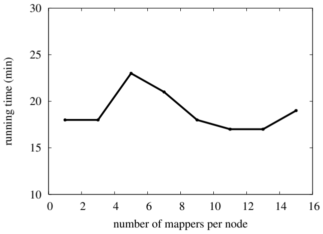

Figure 13. Total running time versus number of mappers per node on EC2.

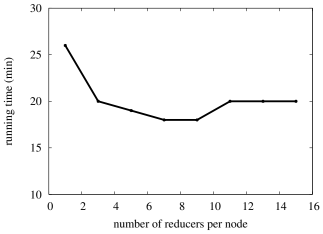

Figure 14. Total running time versus number of reducers per node on EC2.

c. Reducers' running time distribution

Figure 15, 16, 17 and 18 show the distribution of the reducers' running time on Athena. We observe that when we increase the number of reducers per node, the distribution becomes more volatile; for example, when we concurrently run 15 reducers per node, the reducers' completion time vary from 20 minutes to 120 minutes. This also indicates the bad I/O performance on Athena for concurrent accessing.

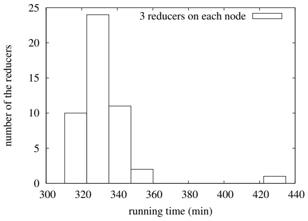

Figure 15. 3 reducers per computing node.

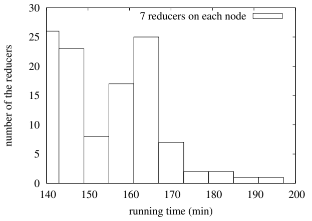

Figure 16. 7 reducers per computing node.

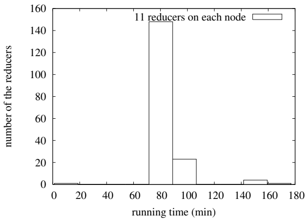

Figure 17. 11 reducers per computing node.

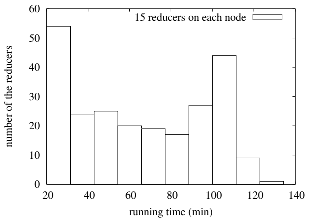

Figure 18. 15 reducers per computing node.

3) Sizes of the files on HDFS for each sub-template: Figure 19 shows how the sizes of the files that store the counts of colorful embeddings vary for different subtemplates when we run SAHAD on *U10-1*. It is consistent with the theoretical bound presented in Lemma 5.2.

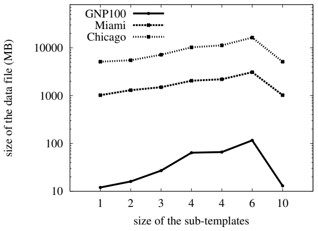

Figure 19. Size of the data for $\forall T_i \in \mathcal{T}$. There are totally 7 templates in $\mathcal{T}$, two of which are with the same size 4.

D. Illustrative applications

In this section, we illustrate the performance on 3 different kinds of queries. We use Athena and assign 16 nodes as the data nodes; for each node, we assign a maximum of 4 mappers and 3 reducers per node. Our experiments on EC2 for some of these queries are discussed later in Section VII-E.

1. Unlabeled subgraph queries: Here we compute the counts of templates *U5-1*, *U7-1* and *U10-1* on GNP100 and Miami, as shown in Figure 20. We also plot how the running time scales to different templates and networks, as shown in Figure 21 – we observe that for unlabeled template with up to 10 nodes on the Miami graph, SAHAD runs in less than 25 minutes.

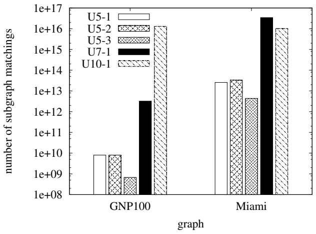

Figure 20. The counts of unlabeled subgraphs in GNP100 and Miami.

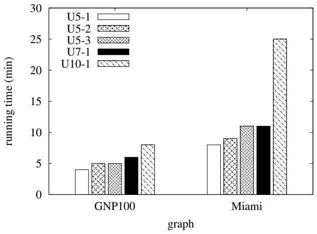

Figure 21. Running time for counting unlabeled subgraph in GNP100 and Miami.

2. Labeled subgraph queries: Here we count the total number of embeddings of template *L7-1*, *L10-1* and *L12-1* in Miami and Chicago. Figure 23 shows that the running time for counting templates up to 12 nodes is around 15 minutes on Miami, less than 35 minutes needed for Chicago. The running time is much less for the labeled subgraph queries than that for the unlabeled subgraph queries. It is due to the fact that that labeled templates have much less number of embeddings due to the label constraints.

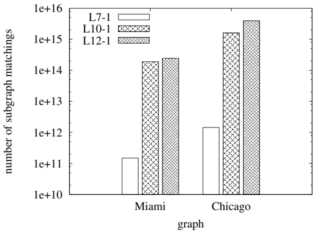

Figure 22. The counts of labeled templates in Miami and Chicago.

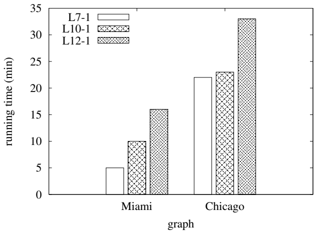

Figure 23. Running time for counting labeled templates in Miami and Chicago.

3. Computing graphlet frequency distribution: Figure 24 and 25 show the graphlet frequency distribution of the networks of Miami and Chicago, respectively. The template is *U5-1*. It takes 15 minutes and 35 minutes to compute graphlet frequency distribution on Miami and Chicago, respectively.

E. SAHAD *on Amazon EC2*

On EC2, we run unlabeled and labeled subgraph queries on Miami and GNP100 for templates *U5-1*, *U7-1*, *U10-1*, *L7-1*, *L10-1* and *L12-1*. We use the same 4 EC2 instances as discussed previously, and each node runs up to a maximum of 2 mappers and 8 reducers concurrently. As shown in Figure 26 and 27, the running time on EC2 is comparable to that on Athena, except for *U10-1* on Miami, which takes roughly 2.5 hours to finish on EC2, but only 25 minutes on Athena. This is because for such a large template and graph as large as Miami, input/output data as well as the I/O pressure on disks are tremendous. EC2 uses virtual disks as local storage, which hurt overall performance when dealing with such a large amount of data.

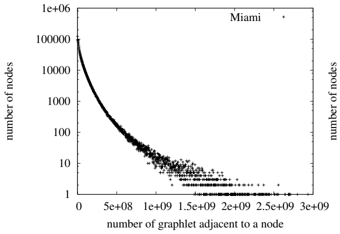

Figure 24. Graphlet frequency distribution of Miami.

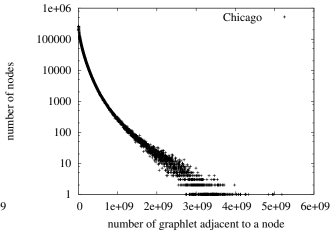

Figure 25. Graphlet frequency distribution of Chicago.

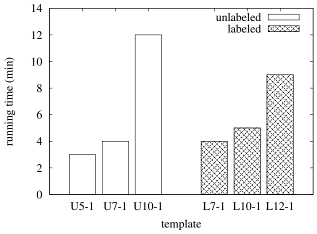

Figure 26. Running time for various templates on GNP100.

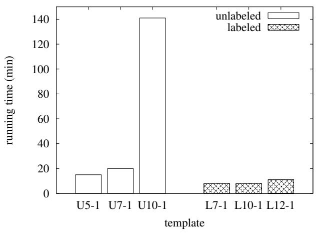

Figure 27. Running time for various templates on Miami.

VIII. Conclusion

In this paper, we develop SAHAD, an algorithm for a broad class of subgraph isomorphism problems in very large networks involving queries in the form of trees. It is the first such MapReduce based algorithm with work complexity asymptotically close to the best sequential algorithm of [1]. SAHAD scales well to very large templates (up to size 12) on very large graphs (with over 500M edges) and on very diverse computing resources, especially including the Amazon EC2. We find the disk I/O to be the main bottleneck in scaling SAHAD to much larger instances, and we posit that employing multiple disks per node (a rising trend in Hadoop) or using I/O caching will help mitigate this bottleneck and boost performance even further.

Acknowledgements. The work of Zhao Zhao, Maleq Khan, Anil Kumar S. Vullikanti and Madhav V. Marathe has been partially supported by NSF PetaApps Grant OCI-0904844, NSF NETS Grant CNS-0831633, NSF Grant CNS-0845700, NSF Netse Grant CNS-1011769, NSF SDCI Grant OCI-1032677, DOE Grant DE-SC0003957, NIH MIDAS project 2U01GM070694-7, and DTRA grants HDTRA1-09-1-0017, HDTRA1-11-1-0016 and HDRTA1-11-D-0016-0001. The work of Guanying Wang and Ali R. Butt is partially supported by the following NSF grants: CCF-0746832, CNS-1016793 and CNS-1016408.

References

[1] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241, 2008.

[2] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):856, 1995.

[3] Amazon. Elastic Computing Cloud (EC2). http://aws.amazon.com/ec2.

[4] C. Barrett, R. Beckman, M. Khan, V. Kumar, M. Marathe, P. Stretz, T. Dutta, and B. Lewis. Generation and analysis of large synthetic social contact networks. In *Winter Simulation Conference*, 2009.

[5] E. Bloedorn, N. J. Rothleder, D. DeBarr, and L. Rosen. Relational graph analysis with real-world constraints: An application in irs tax fraud detection. In *AAAI*, 2005.

[6] C. Borgelt and M. R. Berhold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, 2002.

[7] M. Bröcheler, A. Pugliese, and V. Subrahmanian. Cosi: Cloud oriented subgraph identification in massive social networks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 248–255. IEEE, 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering*, pages 1036–1050, 2005.

[10] F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326, 2004.

[11] L. Getoor and C. P. Diehl. Link mining: a survey. *SIGKDD Explor. Newsl.*, 7:3–12, 2005.

[12] J. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Research in Computational Molecular Biology*, pages 92–106. Springer, 2007.

[13] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: Mining maximal frequent subgraphs from graph databases. In *ACM KDD*, 2004.

[14] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 52(2):114–132, 2008.

[15] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *ECML-PKDD*, 2000.

[16] M. Kowaluk, A. Lingas, and E. Lundell. Counting and detecting small subgraphs via equations and matrix multiplication. In *ACM SODA*, 2011.

[17] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271, 2005.

[18] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. *Advances in Knowledge Discovery and Data Mining*, pages 380–389, 2006.

[19] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. *Advanced Parallel Processing Technologies*, pages 341–355, 2009.

[20] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824, 2002.

[21] R. Pagh and C. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 2011.

[22] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177, 2007.

[23] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *ICDE*, 2003.

[24] R. Ronen and O. Shmueli. Evaluating very large datalog queries on social networks. 2009.

[25] S. Sakr. Graphrel: A decomposition-based and selectivity-aware relational framework for processing sub-graph queries. 2009.

[26] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[27] C. Tsourakakis, U. Kang, G. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

[28] V. Vassilevska and R. Williams. Finding, minimizing, and counting weighted subgraphs. In *ACM STOC*, 2009.

[29] T. White. *Hadoop: The definitive guide*. Yahoo Press, 2010.

[30] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *KDD*, 2005.

[31] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM Trans. Database Syst.*, 32, June 2007.

[32] Z. Zhao, M. Khan, V. S. A. Kumar, and M. Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 594–603, 2010.