Arabesque: A System for Distributed Graph Mining

Carlos H. C. Teixeira *, Alexandre J. Fonseca, Marco Serafini,
Georgos Siganos, Mohammed J. Zaki, Ashraf Aboulnaga

Qatar Computing Research Institute - HBKU, Qatar

Abstract

Distributed data processing platforms such as MapReduce and Pregel have substantially simplified the design and deployment of certain classes of distributed graph analytics algorithms. However, these platforms do not represent a good match for distributed graph mining problems, as for example finding frequent subgraphs in a graph. Given an input graph, these problems require exploring a very large number of subgraphs and finding patterns that match some "interestingness" criteria desired by the user. These algorithms are very important for areas such as social networks, semantic web, and bioinformatics.

In this paper, we present Arabesque, the first distributed data processing platform for implementing graph mining algorithms. Arabesque automates the process of exploring a very large number of subgraphs. It defines a high-level *filter-process* computational model that simplifies the development of scalable graph mining algorithms: Arabesque explores subgraphs and passes them to the application, which must simply compute outputs and decide whether the subgraph should be further extended. We use Arabesque's API to produce distributed solutions to three fundamental graph mining problems: frequent subgraph mining, counting motifs, and finding cliques. Our implementations require a handful of lines of code, scale to trillions of subgraphs, and represent in some cases the first available distributed solutions.

1. Introduction

Graph data is ubiquitous in many fields, from the Web to advertising and biology, and the analysis of graphs is becoming increasingly important. The development of algorithms for graph analytics has spawned a large amount of research, especially in recent years. However, graph analytics has traditionally been a challenging problem tackled by expert researchers, who can either design new specialized algorithms for the problem at hand, or pick an appropriate and sound solution from a very vast literature. When the input graph or the intermediate state or computation complexity becomes very large, scalability is an additional challenge.

The development of graph processing systems such as Pregel [25] has changed this scenario and made it simpler to design scalable graph analytics algorithms. Pregel offers a simple "think like a vertex" (TLV) programming paradigm, where each vertex of the input graph is a processing element holding local state and communicating with its neighbors in the graph. TLV is a perfect match for problems that can be represented through linear algebra, where the graph is modeled as an adjacency matrix (or some other variant like the Laplacian matrix) and the current state of each vertex is represented as a vector. We call this class of methods *graph computation* problems. A good example is computing PageRank [6], which is based on iterative sparse matrix and vector multiplication operations. TLV covers several other algorithms that require a similar computational architecture, for example, shortest path algorithms, and over the years many optimizations of this paradigm have been proposed [17, 26, 36, 42].

Despite this progress, there remains an important class of algorithms that cannot be readily formulated using the TLV paradigm. These are *graph mining* algorithms used to discover relevant patterns that comprise both structure-based and label-based properties of the graph. Graph mining is widely used for several applications, for example, discovering 3D motifs in protein structures or chemical compounds, extracting network motifs or significant subgraphs from protein-protein or gene interaction networks, mining attributed patterns over semantic data (e.g., in Resource Description Framework or RDF format), finding structure-content relationships in social media data, dense subgraph

* Currently a PhD student at the Federal University of Minas Gerais, Brazil. This work was done while the author was at QCRI.

SOSP'15, October 4–7, 2015, Monterey, CA, USA.

http://dx.doi.org/10.1145/2815400.2815410

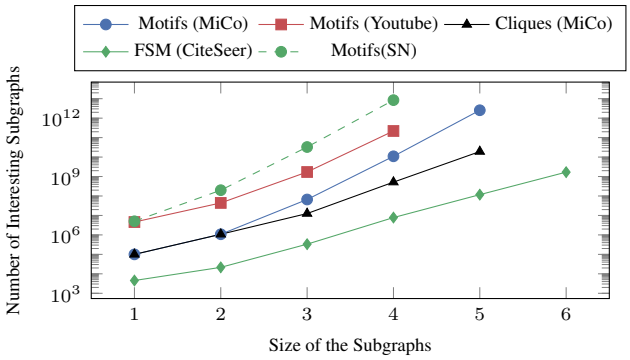


Figure 1: Exponential growth of the intermediate state in graph mining problems (motifs counting, clique finding, FSM: Frequent subgraph mining) on different datasets.

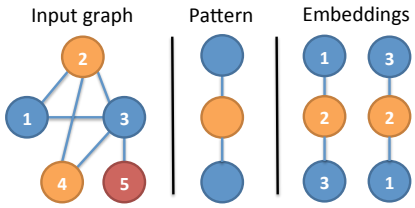


Figure 2: Graph mining concepts: an input graph, an example pattern, and the embeddings of the pattern. Colors represent labels. Numbers denote vertex ids. Patterns and embeddings are two types of subgraphs. However, a pattern is a *template*, whereas an embedding is an *instance*. In this example, the two embeddings are *automorphic*.

mining for community and link spam detection in web data, among others. Graph mining algorithms typically take a labeled and immutable graph as input, and mine patterns that have some algorithm-specific property (e.g., frequency above some threshold) by finding all instances of these patterns in the input graph. Some algorithms also compute aggregated metrics based on these subgraphs.

Designing graph mining algorithms is a challenging and active area of research. In particular, scaling graph mining algorithms to even moderately large graphs is hard. The set of possible patterns and their subgraphs in a graph can be exponential in the size of the original graph, resulting in an explosion of the computation and intermediate state. Figure 1 shows the exponential growth of the number of "interesting" subgraphs of different sizes in some of the graph mining problems and datasets we will evaluate in this paper. Even graphs with few thousands of edges can quickly generate hundreds of millions of interesting subgraphs. The need for enumerating a large number of subgraphs characterizes graph mining problems and distinguishes them from graph computation problems. Despite this state explosion problem, most graph mining algorithms are centralized because of the complexity of distributed solutions.

In this paper, we propose automatic *subgraph exploration* as a generic building block for solving graph mining problems, and introduce Arabesque, the first embedding exploration system specifically designed for distributed graph mining. Conceptually, we move from TLV to "think like an *embedding*" (TLE), where by embedding we denote a subgraph representing a particular instance of a more general template subgraph called a *pattern* (see Figure 2).

Arabesque defines a high-level *filter-process* computational model. Given an input graph, the system takes care of automatically and systematically visiting all the embeddings that need to be explored by the user-defined algorithm, performing this exploration in a distributed manner. The system passes all the embeddings it explores to the application, which consists primarily of two functions: *filter*, which indicates whether an embedding should be processed, and *process*, which examines an embedding and may produce some output. For example, in the case of finding cliques the filter function prunes embeddings that are not cliques, since none of their extensions can be cliques, and the process function outputs all explored embeddings, which are cliques by construction. Arabesque also supports the pruning of the exploration space based on user-defined metrics aggregated across multiple embeddings.

The Arabesque API simplifies and thus democratizes the design of graph mining algorithms, and automates their execution in a distributed setting. We used Arabesque to implement and evaluate scalable solutions to three fundamental and diverse graph mining problems: frequent subgraph mining, counting motifs, and finding cliques. These problems are defined precisely in Section 2. Some of these algorithms are the first distributed solutions available in the literature, which shows the simplicity and generality of Arabesque.

Arabesque's embedding-centered API facilitates a highly scalable implementation. The system scales by spreading embeddings uniformly across workers, thus avoiding hotspots. By making it explicit that embeddings are the fundamental unit of exploration, Arabesque is able to use fast coordination-free techniques, based on the notion of embedding canonicality, to avoid redundant work and minimize communication costs. It also enables us to store embeddings efficiently using a new data structure called Overapproximating Directed Acyclic Graph (ODAG), and to devise a new two-level optimization for pattern-based aggregation, which is a common operation in graph mining algorithms.

Arabesque is implemented as a layer on top of Apache Giraph [3], a Pregel-inspired graph computation system, thus allowing both graph computation and graph mining algorithms to run on top of the same infrastructure. The implementation does not use a TLV approach: it considers Giraph just as a regular data parallel system implementing the Bulk Synchronous Processing model.

To summarize, we make the following contributions:

- We propose embedding exploration, or "think like an embedding", as an effective basic building block for graph mining. We introduce the filter-process computational model (Section 3), design an API that enables embedding exploration to be expressed effectively and succinctly, and present three example graph mining applications that can be elegantly expressed using the Arabesque API (Section 4).
- We introduce techniques to make distributed embedding exploration scalable: coordination-free work sharing, efficient storage of embeddings, and an important optimization for pattern-based aggregation (Section 5).
- We demonstrate the scalability of Arabesque on various graphs. We show that Arabesque scales to hundreds of cores over a cluster, obtaining orders of magnitude reduction of running time over the centralized baselines (Section 6), and can analyze trillions of embeddings on large graphs.

The Arabesque system, together with all applications used for this paper, is publicly available at the project's website: www.arabesque.io.

2. Graph Mining Problems

In this section, we introduce the graph-theoretic terms we will use throughout the text and characterize the space of graph mining problems addressed by Arabesque.

Terminology: Graph mining problems take an input graph G where vertices and edges are labeled. Vertices and edges have unique ids, and their labels are arbitrary, domain-specific attributes that can be null. An *embedding* is a subgraph of G, i.e., a graph containing a subset of the vertices and edges of G. A *vertex-induced* embedding is defined starting from a set of vertices by including all edges of G whose endpoints are in the set. An *edge-induced* embedding is defined starting from a set of edges by including all the endpoints of the edges in the set. For example, the two embeddings of Figure 2 are induced by the edges $\{(1,2),(2,3)\}$. In order to be induced by the vertices $\{1,2,3\}$ the embeddings should also include the edge $(1,3)$. We consider only *connected* embeddings such that there is a path connecting each pair of vertices.

A *pattern* is an arbitrary graph. We say that an embedding e in G is *isomorphic* to a pattern p if and only if there exists a one-to-one mapping between the vertices of e and p, and between the edges of e and p, such that: (i) each vertex (resp. edge) in e has one matching vertex (resp. edge) in p with the same labels, and vice versa; (ii) each matching edge connects matching vertices. In this case, we say informally that e *has pattern* p. Equivalently, we say that there exists a *subgraph isomorphism* from p to G, and we call e an *instance* of pattern p in G. Two embeddings are *automorphic* if and only if they contain the same edges and vertices, i.e., they are equal and thus also isomorphic (see for example Figure 2).

Characterizing Graph Mining Problems: Arabesque targets graph mining problems, which involve subgraph enumeration. Given an immutable input graph G with labeled vertices and edges, graph mining problems focus on the enumeration of all *patterns* that satisfy some user-specified "interestingness" criteria. A given pattern p is evaluated by listing its matches or embeddings in the input dataset G, and then filtering out the uninteresting ones. Graph mining problems often require subgraph isomorphism checks to determine the patterns related to sets of embeddings, and also graph automorphism checks to eliminate duplicate embeddings.

We consider graph mining problems where one seeks connected graph patterns whose embeddings are vertex- or edge-induced. There are several variants of these problems. The input dataset may comprise a collection of many graphs, or a single large graph. The embeddings of a pattern comprise the set of (exact) isomorphisms from the pattern p to the input graph G. However, in inexact matching, one can seek inexact or approximate isomorphisms (based on notions of edit-distances, label costs, etc.). There are many variants in terms of the interestingness criteria, such as frequent subgraph mining, dense subgraph mining, or extracting the entire frequency distribution of subgraphs up to some given number of vertices. Also related to graph mining is the problem of graph matching, where a query pattern q is fixed, and one has to retrieve all its matches in the input graph G. Solutions to graph matching typically use indexing approaches, which pre-compute a set of paths or frequent subgraphs, and use them to facilitate fast matching and retrieval. As such graph mining encompasses the matching problem, since we have to both enumerate the patterns and find their matches. Further, any solution to the single input graph setting is easily adapted to the multiple graph dataset case. Therefore, in this paper we focus on graph mining tasks on a single large input graph. See [1] for a state-of-the-art review of graph mining and search.

Use Cases: Throughout this paper we consider three classes of problems: *frequent subgraph mining*, *counting motifs*, and *finding cliques*. We chose these problems because they represent different classes of graph mining problems. The first is an example of explore-and-prune problems, where only embeddings corresponding to a frequent pattern need to be further explored. Counting motifs requires exhaustive graph exploration up to some maximum size. Finding cliques is an example of dense subgraph mining, and allows one to prune the embeddings using local information. We discuss these problems below in more detail.

Consider the task of *frequent subgraph mining (FSM)*, i.e., finding those subgraphs (or patterns, in our terminology) that occur a minimum number of times in the input dataset. The occurrences are counted using some anti-monotonic function on the set of its embeddings. The anti-monotonic property states that the frequency of a supergraph should not

exceed the frequency of a subgraph, which allows one to stop extending patterns/embeddings as soon as they are deemed to be infrequent. There are several anti-monotonic metrics to measure the frequency of a pattern. While their differences are not very important for this discussion, they all require aggregating metrics from all embeddings that correspond to the same pattern. We use the *minimum image-based support* metric [7], which defines the frequency of a pattern as the minimum number of distinct mappings for any vertex in the pattern, over all embeddings of the pattern. Formally, let G be the input graph, p be a pattern, and $\mathcal{E}$ its set of embeddings. The pattern p is frequent if $sup(p, \mathcal{E}) \geq \theta$, where $sup()$ is the minimum image-based support function, and θ is a user-specified threshold. The FSM task is to mine *all* frequent subgraph patterns from a single large graph G.

A *motif* p is defined as a connected pattern of vertex-induced embeddings that exists in an input graph G. Further, a set of motifs is required to be non-isomorphic, i.e., there should obviously be no duplicate patterns. In motif mining [30], the input graph is assumed to be unlabeled, and there is no minimum frequency threshold; rather the goal is to extract all motifs that occur in G along with their frequency distribution. Since this task is inherently exponential, the motifs are typically restricted to patterns of order (i.e., number of vertices) at most k. For example, for $k = 3$ we have two possible motifs: a chain where ends are not connected and a (complete) triangle. Whereas the original definition of motifs targets unlabeled and induced patterns, we can easily generalize the definition to labeled patterns.

In *clique mining* the task is to enumerate all complete subgraphs in the input graph G. A complete subgraph, or clique, p with k vertices is defined as a connected subgraph where each vertex has degree $k - 1$, i.e., each vertex is connected to all other vertices (we assume there are no self-loops). The clique problem can also be generalized to maximal cliques, i.e., those not contained in any other clique, and frequent cliques, if we impose a minimum frequency threshold in addition to the completeness constraint.

3. The Filter-Process Model

We now discuss the filter-process model, which formalizes how Arabesque performs embedding exploration based on application-specific *filter* and *process* functions.

3.1 Computational Model

Arabesque computations proceed through a sequence of *exploration steps*. At a conceptual level, the system performs the operations illustrated in Algorithm 1 in each exploration step. Each step is associated with an initial set I containing embeddings of the input graph G. Arabesque automates the process of exploring the graph and expanding embeddings. Applications are specified via two user-defined functions: a *filter function* ϕ and a *process function* π. The application

Algorithm 1: Exploration step in Arabesque.

```
input  : Set I of initial embeddings for this step
output: Set F of extended embeddings for the next step (initially
        empty)
output: Set O of new outputs (initially empty)
foreach e ∈ I such that α(e) do
    add β(e) to O;
    C ← set of all extensions of e obtained by adding one incident
    edge / neighboring vertex;
    foreach e' ∈ C do
        if φ(e') and there exists no e'' ∈ F automorphic to e' then
            add π(e') to O;
            add e' to F;
run aggregation functions;
```

can optionally also define two additional functions that will be described shortly.

Arabesque starts an exploration by generating the set of *candidate* embeddings C, which are obtained by expanding the embeddings in I. The system computes candidates by adding one incident edge or vertex to e, depending on whether it runs in edge-based or vertex-based exploration mode. In edge-based exploration, an embedding is an edge-induced subgraph; in vertex-based exploration, it is vertex-induced (see Section 2). In the first exploration step, I contains only a special "undefined" embedding, whose expansion C consists of all edges or vertices of G, depending on the type of exploration. The application can decide between edge-based or vertex-based exploration during initialization.

After computing the candidates, the filter function ϕ examines each candidate e' and returns a Boolean value indicating whether e' needs to be processed. If ϕ returns true, the process function π takes e' as input and outputs a set of user-defined values. By default, e' is then added to the set F. After an exploration step is terminated, I is set to be equal to F before the start of the next step. The computation terminates when the set F is empty at the end of a step.

Arabesque runs the outer loop of Algorithm 1 in parallel by partitioning the embeddings in I over multiple servers each running multiple worker threads. This distribution is transparent to applications. Each execution step is executed as a superstep in the Bulk Synchronous Parallel model [38].

Optional aggregation functions: The filter-process model described so far considers single embeddings in isolation. A common task in graph mining systems is to aggregate values across multiple embeddings, for example grouping embeddings by pattern. To this end, Arabesque offers specific functions to execute user-defined aggregation for multiple embeddings. Aggregation can group embeddings by an arbitrary integer value or by pattern, and is executed on candidate embeddings at the end of the exploration step in which they are generated.

The optional *aggregation filter* function α and *aggregation process* function β can filter and process an embedding e in the exploration step following its generation. At that

time, aggregated information collected from all the embeddings generated in the same exploration step as e becomes available. The α function can take filtering decisions before embedding expansion based on aggregate values. For example, in the frequent subgraph mining problem we can use aggregators to count the embeddings associated with a given pattern, and then filter out embeddings of infrequent patterns with α. Similarly, the β function can be used to output aggregate information about an embedding, for example its frequency. By default, α returns *true* and β does not add any output to O.

Guarantees and requirements: Embedding exploration processes every embedding that is not filtered out. More formally, it guarantees the following *completeness* property: for each embedding e such that $\phi(e) = \alpha(e) = \textit{true}$, embedding exploration must add $\pi(e)$ and $\beta(e)$ to O.

Completeness assumes some properties of the user-defined functions that emerge naturally in graph mining algorithms. The first property is that the application considers two embeddings e and e' to be equivalent if they are automorphic (see Section 2). Formally, Arabesque requires what we call *automorphism invariance*: if e and e' are automorphic, then each user-defined function must return the same result for e and e'. Arabesque leverages this natural property of graph mining algorithms to prune automorphic embeddings and substantially reduce the exploration space.

The second property Arabesque requires is called *anti-monotonicity* and is formally defined as follows: if $\phi(e) = \textit{false}$ then it holds that $\phi(e') = \textit{false}$ for each embedding e' that extends e. The same property holds for the optional filter function α. This is one of the essential properties for any effective graph mining method and guarantees that once a filter function tells the framework to prune an embedding, all its extensions can be ignored.

3.2 Alternative Paradigms: Think Like a Vertex and Think Like a Pattern

We can now contrast our embedding-centric or "Think Like an Embedding" (TLE) model with other approaches to build graph mining algorithms. We empirically compare with these approaches in Section 6.

The standard paradigm of systems like Pregel is *vertex-centric* or "Think Like a Vertex" (TLV), because computation and state are at the level of a vertex in the graph. TLV systems are designed to scale for large input graphs: the information about the input graph is distributed and vertices only have information about their local neighborhood. In order to perform embedding exploration, each vertex can keep a set of local embeddings, initially containing only the vertex itself. Then, to expand a local embedding e, vertices push it to the "border" vertices of e that know how to expand e by adding its neighbors. Each expansion results in a new embedding, which is sent again to border vertices and so on. With this approach, highly connected vertices must take on a disproportionate fraction of embeddings to expand. The approach also creates a significant number of duplicate messages because each new embedding must be sent to all its border vertices. These limitations significantly affect performance. In our experiments, we observed that TLV-based embedding exploration algorithms can be two orders of magnitude slower compared to TLE.

The current state-of-the-art centralized methods for solving graph mining tasks typically adopt a different, *pattern-centric* or "Think Like a Pattern" (TLP) approach. The key difference between TLP and the embedding-centric view of the filter-process model is that it is not necessary to explicitly materialize all embeddings: state can be kept at the granularity of patterns (which are much fewer than embeddings) and embeddings may be re-generated on demand. The process starts with the set of all possible (labeled) single vertices or edges as candidate patterns. It then processes embeddings of each pattern, often by recomputing them on the fly. After aggregation and pattern filtering, the valid set of patterns are extended by adding one more vertex or edge. Subgraph or pattern mining proceeds iteratively via recursive extension, processing and filtering steps, and continues until no new patterns are found. Parallelizing the computation via partitioning it by pattern can easily result in load imbalance, as our experiments show. This is because there are often only few patterns that are highly popular – indeed, finding these few patterns is the very goal of graph mining. These popular patterns result in hotspots among workers and thus in poor load balancing.

4. Arabesque: API, Programming, and Implementation

We now describe the Arabesque Java API and show how we use it to implement our example applications.

4.1 Arabesque API

The API of Arabesque is illustrated in Figure 3. The user must implement two functions: `filter`, which corresponds to ϕ, and `process`, which corresponds to π. The process function in the API is responsible for adding results to the output by invoking the `output` function provided by Arabesque, which prints the results to the underlying file system (e.g. HDFS). The optional functions α and β correspond, respectively, to `aggregationFilter` and `aggregationProcess`. These application-specific functions are invoked by the Arabesque framework as illustrated in Algorithm 1. All these functions have access to a local read-only copy of the graph.

For performance and scalability reasons, a MapReduce-like model is used to compute aggregated values. Applications can send data to reducers using the `map` function, which is part of the Arabesque framework and adds a value to an aggregation group defined by a certain key. Many applications use the pattern of an embedding as the aggregation key. Arabesque detects when the key is a pattern and uses

Mandatory application-defined functions:

```
boolean filter (Embedding e)
void process (Embedding e)
```

Optional application-defined functions:

```
boolean aggregationFilter (Embedding e)
void aggregationProcess (Embedding e)
Pair<K,V> reduce (K key, List<V> values)
Pair<K,V> reduceOutput (K key, List<V> value)
boolean terminationFilter (Embedding e)
```

Arabesque functions invoked by applications:

```
void output (Object value)
void map (K key, V value)
V readAggregate (K key)
void mapOutput (K key, V value)
```

Figure 3: Basic user-defined functions in Arabesque.

```
boolean filter(Embedding e) { return true; }
void process(Embedding e){
    map (pattern(e), domains(e)); }
Pair<Pattern,Domain> reduce
             (Pattern p, List<Domain> domains){
    Domain merged_domain = merge(domains);
    return Pair (p, merged_domain); }
boolean aggregationFilter(Embedding e){
    Domain m_domain = readAggregate(pattern(e));
    return (support(m_domain)>=THRESHOLD); }
void aggregationProcess(Embedding e) {
    output(e); }
```

(a) **Frequent subgraph mining** (edge-based exploration)

```
boolean filter(Embedding e){
    return (numVertices(e) <= MAX_SIZE);}
void process(Embedding e){
    mapOutput (pattern(e),1); }
Pair<Pattern,Integer> reduceOutput
       (Pattern p, List<Integer> counts){
    return Pair (p, sum(counts)); }
```

(b) **Counting motifs** (vertex-based exploration)

```
boolean filter (Embedding e){
    return isClique(e); }
void process (Embedding e){ output(e); }
```

(c) **Finding cliques** (vertex-based exploration)

Figure 4: Examples of Arabesque applications.

specific optimizations to make this aggregation efficient, as explained in Section 5.4. The application specifies the aggregation logic through the `reduce` function. This function receives all values mapped to a specific key in one execution step and aggregates them. Any method can read the values aggregated over the previous execution step using the `readAggregate` method.

Output aggregation is a special case where aggregated values are sent directly to the underlying distributed filesystem at the end of each exploration step and are not made available for later reads. It can be used through the methods `mapOutput` and `reduceOutput`. Their logic is similar to the aggregation functions described previously, but aggregation is only executed when the whole computation ends.

The `terminationFilter` function can halt the computation of an embedding when some pre-defined condition is reached. Arabesque applies this filter on an embedding e after executing $\pi(e)$ and before adding e to F. This is just an optimization to avoid unnecessary exploration steps. For example, if we are interested in embeddings of maximum size n, the termination filter can halt the computation after processing embeddings of size n at step n. Without this filter, the system would have to proceed to step $n+1$ and generate all embeddings of size $n+1$ just to filter all of them out.

4.2 Programming with Arabesque

We used the Arabesque API to implement algorithms that solve the three problems discussed in Section 2. The pseudocode of the implementations is in Figure 4. Each of the applications consists of very few lines of code, a stark contrast compared to the complexity of the specialized state of the art algorithms solving the same problems [8, 30, 43].

In frequent subgraph mining we use aggregation to calculate the support function described in Section 2. The support metric is based on the notion of domain, which is defined as the set of distinct mappings between a vertex in p and the matching vertices in any automorphism of e. The `process` function invokes `map` to send the domains of e to the reducer responsible for the pattern p of e. For example, in Figure 2 the domain of the blue vertex on the top of the pattern is vertex 1 in the first embedding and 3 in the second. The function `reduce` merges all domains: the merged domain of a vertex in p is the union of all its aggregated mappings. Since expansion is done by adding one edge in each exploration step, we are sure that all embeddings for p are visited and processed in the same exploration step. The `aggregationFilter` function reads the merged domains of p using `readAggregate` and computes the support, which is the minimum size of the domain of any vertex in p. It then filters out embeddings for patterns that do not have enough support. Finally, the `aggregationProcess` function outputs all the embeddings having a frequent pattern (those that survive the aggregation-filter). The implementation of this application consists of 280 lines of Java code. Of these, 212 are related to handling domains and computing support, which are basic tasks required in any algorithm for frequent subgraph mining to characterize whether an embedding is relevant. By comparison, the centralized baseline we use for evaluation, GRAMI [14], consists of 5,443 lines of Java code.

For motif frequency computation, we perform an exhaustive exploration of all embeddings until we reach a given maximum size and count all embeddings having the same pattern. Since the input graph is not labeled in this case, a pattern corresponds to a motif. The function `mapOutput` sends a value to the output reducer responsible for the pattern of e. The `reduceOutput` function outputs the sum of the counts for each motif p. Our implementation consists of

18 lines of code, very few compared to the 3,145 lines of C code of our centralized baseline (Gtries [31]).

In finding cliques we do a local pruning: if an embedding is not a clique, none of its extensions can be a clique. Since we visit only cliques, the evaluation function outputs the embeddings it receives as input. The `isClique` function checks that the newly added vertex is connected with all previous vertices in the embedding. This application consists of 19 lines of code, while our centralized baseline (Mace [37]) consists of 4,621 lines of C code.

In all these examples it is easy to verify that the evaluation and filter functions satisfy the anti-monotonic and automorphism invariance properties required by the Arabesque computational model.

4.3 Arabesque implementation

Arabesque can execute on top of any system supporting the BSP model. We have implemented Arabesque as a layer on top of Giraph. The implementation does not follow the TLV paradigm: we use Giraph vertices simply as *workers* that bear no relationship to any specific vertex in the input graph. Each worker has access to a copy of the whole input graph whose vertices and edges consist of incremental numeric ids. Communication among workers occurs in an arbitrary point-to-point fashion and communication edges are not related to edges in the input graph. Giraph computations proceed through synchronous supersteps according to the BSP model: at each superstep, workers first receive all messages sent in the previous superstep, then process them, and finally send new messages to be delivered at the next superstep. The operations of the workers are described in Section 5. Aggregation functions, if specified, are executed using standard Giraph aggregators. Optional output workers are used for applications that aggregate output values. The input values for output aggregation persist over supersteps. Once the computation is over, output workers aggregate all their input values and output them.

5. Graph Exploration Techniques

We now describe in more detail how Arabesque performs graph exploration. We first discuss the coordination-free exploration strategy used by workers to avoid redundant work. We then introduce the techniques we use to store and partition embeddings efficiently.

5.1 Coordination-Free Exploration Strategy

When running exploration in a distributed setting, multiple workers can reach two "identical", i.e., automorphic (see Section 2), embeddings through different exploration paths. Consider for example the graph of Figure 2. Two different workers w_1 and w_2 may reach the two embeddings in Figure 2, one starting from edge $(1, 2)$ by adding $(2, 3)$ and the other starting from edge $(3, 2)$ by adding $(2, 1)$. Since all user-defined functions are automorphism-invariant (see Section 3) we can avoid redundant work by discarding all but one of the identical, automorphic embeddings.

Arabesque solves this problem using a novel coordination-free scheme based on the notion of *embedding canonicality*. Informally, we need to select exactly one of the redundant automorphic embeddings and elect it as "canonical". In our example, before w_1 and w_2 execute the filter and process functions on a new embedding e, Arabesque executes an embedding canonicality check to verify whether e can be pruned (see Algorithm 1). This check runs on a single embedding without requiring coordination, as we now discuss.

A sound canonicality check must have the property of *uniqueness*: given the set S_e of all embeddings automorphic to an embedding e, there is exactly one canonical embedding e_c in S_e. We call e_c the *canonical automorphism* of e. In Arabesque we also need an additional property for canonicality checks called *extendibility*, which we define as follows. Let e be a candidate embedding obtained by extending a parent embedding e' by one vertex or edge. The parent embedding e' is canonical because it has not been pruned. Let e_c be the canonical automorphism of e. Extendibility requires that e_c is one of the extensions of e'. This allows Arabesque to prune the automorphisms of a parent e' while still exploring the canonical automorphism of each child e.

Algorithm 2: Arabesque's incremental embedding canonicality check (vertex-based exploration)

```
input : Input graph G
input : Canonical parent embedding ⟨v_1, ..., v_n⟩
input : Extension vertex v
output: true iff ⟨v_1, ..., v_n, v⟩ is canonical
if v_1 > v then
    return false;
foundNeighbour ← false;
for i = 1 ... n do
    if foundNeighbour = false and v_i neighbor of v in G then
        foundNeighbour ← true;
    else if foundNeighbour = true and v_i > v then
        return false;
return true;
```

Arabesque checks embedding canonicality for each candidate before applying the filter function. There can be a huge number of candidates, so it is essential that the check is efficient. We developed a linear-time algorithm, which is based on the following definition of canonical embedding.

Consider the case of vertex-based exploration (the edge-based case is analogous). An embedding e is canonical if and only if its vertices were visited in the following order: start by visiting the vertex with the smallest id, and then recursively add the neighbor in e with smallest id that has not been visited yet. For better performance, Arabesque performs this canonicality check in an incremental fashion, as illustrated in Algorithm 2. When a worker processes an embedding $e \in I$, it can already assume that e is canonical because Arabesque prunes non-canonical embeddings before passing them on

to the next exploration step. Arabesque, characterizes an embedding as the list of its vertices sorted by the order in which they have been visited – the embedding is vertex-induced so the list uniquely identifies it. When Arabesque checks the canonicality of a new candidate embedding obtained by adding a vertex v to a parent canonical embedding e, the algorithm scans e in search for the first neighbor v' of v, and then vertifies that there is no vertex in e after v' with higher id than v.

The extended version of this paper [35] includes proofs showing that Algorithm 2 satisfies the uniqueness and extendibility properties of canonicality checking. We also show that these two properties, together with anti-monotonicity and automorphism invariance, are sufficient to ensure that Arabesque satisfies the completeness property of embedding exploration (see Section 3).

5.2 Storing Embeddings Compactly

Graph mining algorithms can easily generate trillions of embeddings as intermediate state. Centralized algorithms typically do not explicitly store the embeddings they have explored. Instead, they use application-specific knowledge to rebuild embeddings on the fly from a much smaller state.

The only application-level logic available to Arabesque consists of the filter and process functions, which are opaque to the system. After each exploration step, Arabesque receives a set of embeddings filtered by the application and it needs to keep them in order to expand them in the next step. Storing each embedding separately can be very expensive. As we have seen also in Algorithm 2, Arabesque represents embeddings as sequences of numbers, representing vertex or edge ids depending on whether exploration is vertex-based or edge-based. Therefore, we need to find techniques to store sets of sequences of integers efficiently.

Existing compact data structures such as prefix-trees are too expensive because we would still have to store a new leaf for each embedding in the best case, since all canonical embeddings we want to represent are different. In contrast, Arabesque uses a novel technique called Overapproximating Directed Acyclic Graphs, or ODAGs. At a high level, ODAGs are similar to prefix trees where all nodes at the same depth corresponding to the same vertex in the graph are collapsed into a single node. This more compact representation is an overapproximation (superset) of the set of sequences we want to store. When extracting embeddings from ODAGs we must do extra work to discard spurious paths. ODAGs thus trade space complexity for computational complexity. This makes ODAGs similar to the representative sets introduced in [2], which have a higher compression capability but require more work to filter out spurious embeddings. In addition, it is harder to achieve effective load balancing with representative sets. We now discuss the details of ODAGs and show how they are used in Arabesque.

The ODAG Data Structure: For simplicity, we focus the discussion on ODAGs for vertex-based exploration; the edge-based case is analogous. The ODAG for a set of canonical embeddings consists of as many arrays as the number of vertices of all the embeddings. The i^{th} array contains the ids of all vertices in the i^{th} position in any embedding. Vertex v in the i^{th} array is connected to vertex u in the $(i+1)^{th}$ array if there is at least one canonical embedding with v and u in position i and $i+1$ respectively in the original set. An example of ODAG is shown in Figures 5 and 6.

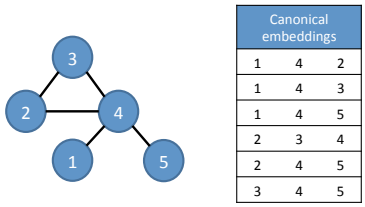


Figure 5: Example graph and its set of S of canonical vertex-induced embeddings of size 3.

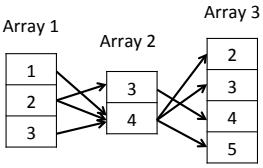


Figure 6: ODAG for the example of Figure 5. It also encodes spurious embeddings such as $\langle 3, 4, 2 \rangle$.

Storing an ODAG is more compact than storing the full set of embeddings. In general, in a graph with N distinct vertices we can have up to $O(N^k)$ different embeddings of size k. With ODAGs we only have to keep edges between k arrays, where k is the size of the embeddings, so the upper bound on the size is $O(k \cdot N^2) = O(N^2)$ if k is a constant much smaller than N.

It is possible to obtain all embeddings of the original set by simply following the edges in the ODAG. However, this will also generate spurious embeddings that are not in the original set. Consider for example the graph of Figure 5 and its set of canonical embeddings S. Expanding the ODAG of Figure 6 generates also $\langle 3, 4, 2 \rangle \notin S$. Filtering out such spurious embeddings requires application-specific logic.

ODAGs in Arabesque: In Arabesque, workers produce new embeddings in each exploration step and add them to the set F (see Algorithm 1). Workers use ODAGs to store F at the end of an exploration step, and extract embeddings from ODAGs at the beginning of the next step.

Filtering out spurious embeddings, as discussed, requires application-specific logic. Applications written using the filter-process model give Arabesque enough information to perform filtering. In fact, workers can just apply the same filtering criteria as Algorithm 1: the canonicality check and the user-defined filter and aggregate filter functions. If any of these checks is negative for an embedding, we know that the embedding itself or, thanks to the anti-monotonicity property, one of its parents, was filtered out. In addition, since

our canonicality check is incremental, we do not need to consider embeddings that extend a non-canonical sequence, so we can prune multiple embeddings at once.

After every exploration step, Arabesque merges and broadcasts the new embeddings, and thus the ODAGs, to every worker. In order to reduce the number of spurious embeddings, workers group their embeddings in one ODAG per pattern. For each pattern, workers merge their local per-pattern ODAGs into a single per-pattern global ODAG. Each per-pattern global ODAG is replicated at each worker before the beginning of the next exploration step.

Merging ODAGs requires merging edges obtained by different workers. For example, consider the first two arrays in Figure 6. One worker might have explored the edge $\langle 2, 3\rangle$ and another worker $\langle 2, 4\rangle$. Edges of ODAG arrays are indexed by their initial vertex, so the two arrays from the two workers will have two different entries for the element 2 of the first array that need to be merged. Edge merging is very expensive because of the high number of edges in an ODAG. Therefore, Arabesque uses a map-reduce step to execute edge merging. Each entry of each array produced by a worker is mapped to the worker associated to its index value. This worker is responsible for merging all edges for that entry produced by all workers into a single entry. The entry is then broadcast to all other workers, which computes the union of all entries in parallel.

5.3 Partitioning Embeddings for Load Balancing

After broadcast and before the beginning of the next exploration step, every worker obtains the same set of ODAGs, one for each pattern mined in the previous execution step. The next step is to partition the set I of new embeddings (see Algorithm 1) among workers. This is achieved by partitioning the embeddings in each pattern ODAG separately.

Workers could achieve perfect load balancing by using a round-robin strategy to share work: worker 1 takes the first embedding, worker 2 the second and so on. However, having workers iterate through all embeddings produced in the previous step, including those that they are not going to process, is computationally intensive. Therefore, workers do round robin on large blocks of b embeddings. The question now is how to identify such blocks efficiently.

Arabesque associates each element v in every array with an estimate of how many embeddings, canonical or not, can be generated starting from v. To this end, Arabesque assigns a cost 1 to every element of the last array, and it assigns to an element of the i^{th} array the sum of the costs of all elements in the $(i+1)^{th}$ array it is connected to. Load can then be balanced by having each worker take a partition of the elements in the first array that has approximately the same estimated cost. While partitioning, it could happen that the cost of an element v of the first array needs to be split among multiple workers. In this case, the costs associated to the elements of the second array connected to v are partitioned. The process continues recursively on subsequent arrays until a balanced load is reached.

5.4 Two-Level Pattern Aggregation for Fast Pattern Canonicality Checking

Arabesque uses a special optimization to speed up per-pattern aggregation, as discussed in Section 4. The optimization was motivated by the high potential cost of this type of aggregation, as we now discuss.

Consider again the example of Figure 2 and assume that we want to count the instances of all single-edge patterns. The three single-edge embeddings $(1, 2)$, $(2, 3)$, and $(3, 4)$ should be aggregated together since they all have a blue and a yellow endpoint. Therefore, their two patterns $(blue, yellow)$ and $(yellow, blue)$ should be considered equivalent because they are isomorphic (see Section 2). The aggregation reducer for these two patterns is associated to a single *canonical pattern* that is isomorphic to both. Mapping a pattern to its canonical pattern thus entails solving the graph isomorphism problem, for which no polynomial-time solution is known [16]. This makes pattern canonicality much harder than embedding canonicality, which is related to the simpler graph automorphism problem.

Identifying a canonical pattern for each single candidate embedding would be a significant bottleneck for Arabesque, as we show in our evaluation, because of the sheer number of candidate embeddings that are generated at each exploration step. Arabesque solves this problem by using a novel technique called *two-level pattern aggregation*.

The first level of aggregation occurs based on what we call *quick* patterns. A quick pattern of an embedding e is the one obtained, in linear time, by simply scanning all vertices (or edges, depending on the exploration mode) of e and extracting the corresponding labels. The quick pattern is calculated for each candidate embedding. In the previous example, we would obtain the quick pattern $(blue, yellow)$ for the embeddings $(1, 2)$ and $(3, 4)$ and the quick pattern $(yellow, blue)$ for the embedding $(2, 3)$. Each worker locally executes the reduce function based on quick patterns. Once this local aggregation completes, a worker computes the canonical pattern p_c for each quick pattern p and sends the locally aggregated value to the reducer for p_c. Arabesque uses the *bliss* library to determine canonical patterns [20].

In summary, instead of executing graph isomorphism for a very large number of candidate embeddings, two-level pattern aggregation computes a quick pattern for every embedding, obtains a number of quick patterns which is orders of magnitude smaller than the candidate embeddings, and then calculates graph isomorphism only for quick patterns.

6. Evaluation

6.1 Experimental Setup

Platform: We evaluate Arabesque using a cluster of 20 servers. Each server has 2 Intel Xeon E5-2670 CPUs with a

total of 32 execution threads at 2.67GHz per core and 256GB RAM. The servers are connected with a 10 GbE network. Hadoop 2.6.0 was configured so that each physical server contains a single worker which can use all 32 execution threads (unless otherwise stated). Arabesque runs on Giraph development trunk from January 2015 with added functionality for obtaining cluster deployment details and improving aggregation performance. These modifications amount to 10 extra lines of code.

	Vertices	Edges	Labels	Av. Degree
CiteSeer	3,312	4,732	6	2.8
MiCo	100,000	1,080,298	29	21.6
Patents	2,745,761	13,965,409	37	10
Youtube	4,589,876	43,968,798	80	19
SN	5,022,893	198,613,776	0	79
Instagram	179,527,876	887,390,802	0	9.8

Table 1: Graphs used for the evaluation.

Datasets: We use six datasets (see Table 1). CiteSeer [14] has publications as vertices, with their Computer Science area as label, and citations as edges. MiCo [14] has authors as vertices, which are labeled with their field of interest, and co-authorship of a paper as edges. Patents [18] contains citation edges between US Patents between January 1963 and December 1999; the year the patent was granted is considered to be the label. Youtube [10] lists crawled video ids and related videos for each video posted from February 2007 to July 2008. The label is a combination of the video's rating and length. SN, is a snapshot of a real world Social Network, which is not publicly available. Instagram is a snapshot of the popular photo and video sharing social network collected by [28]. We consider all the graphs to be undirected. Note that even if some of these graphs are not very large, the explosion of the intermediate computation and state required for graph exploration (see Figure 1) makes them very challenging for centralized algorithms.

Applications and Parameters: We consider the three applications discussed in Sections 2, which we label FSM, Motifs and Cliques. By default, all Motifs executions are run with a maximum embedding size of 4, denoted as MS=4, whereas Cliques are run with a maximum embedding size of MS=5. For FSM, we explicitly state the support, denoted S, used in each experiment as this parameter is very sensitive to the properties of the input graph.

6.2 Alternative Paradigms: TLV and TLP

We start by motivating the necessity for a new framework for distributed graph mining. We evaluate the two alternative computational paradigms that we discussed in Section 3.2. Arabesque (i.e., TLE) will be evaluated in the next subsection. We consider the problem of frequent subgraph mining (FSM) as a use case. Note that there are currently no distributed solutions to solve FSM on a single large input graph in the literature.

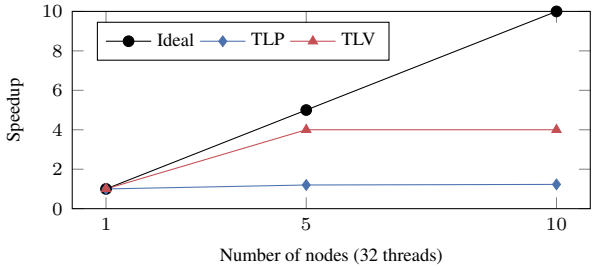


Figure 7: Scalability Analysis of Alternative Paradigms: FSM (S=300) on CiteSeer.

The Case of TLV: Our TLV implementation globally maintains the set of embeddings that have been visited, much like Arabesque. The implementation adopts the TLV approach as described in Section 3.2 and uses the same coordination-free technique as Arabesque to avoid redundant work. The TLV implementation also uses application-specific approaches to control the expansion process. Our TLV implementation of FSM uses this feature to follow the standard depth-first strategy of gSpan [43].

In Figure 7, we show the scalability of FSM with support 300 using the CiteSeer graph. As seen from the figure, TLV does not scale beyond 5 servers. A major scalability bottleneck is that each embedding needs to be replicated to each vertex that has the necessary local information to expand the embedding further. In addition, high-degree vertices need to expand a disproportionate fraction of embeddings. CiteSeer is a scale-free graph thus affecting the scalability of TLV.

Overall TLV performance is two orders of magnitude slower compared to Arabesque. TLV requires more than 300 seconds to run FSM on the CiteSeer graph, while Arabesque requires only 7 seconds for the same setup. The total messages exchanged for this tiny graph is 120 million, versus 137 thousand messages required by Arabesque. Due to the hot-spots inherent to the graph structure, or the label distribution, and the extended duplication of state that the TLV paradigm requires, we conclude that TLV is not suited for solving these problems.

The Case of TLP: The TLP implementation is based on GRAMI [14], which represents the state of the art for centralized FSM. GRAMI keeps state on a per-pattern basis, so few relatively straightforward changes to the code-base were sufficient to derive a TLP implementation where patterns are partitioned across a set of distributed workers.

GRAMI uses a number of optimizations that are specific to FSM. In particular, it avoids materializing all embeddings related to a pattern, a common approach for TLP algorithms. Whenever a new pattern is generated, its instances are recalculated on the fly, stopping as soon as a sufficient number of embeddings to pass the frequency threshold is found. GRAMI thus solves a simpler problem than the TLV and

Arabesque implementations of FSM: it does not output all frequent embeddings but only their patterns.

The TLP version of GRAMI is significantly faster than TLV: GRAMI runs in 3 seconds for the same input graph and support compared to the hundreds of seconds for TLV. However, TLP suffers from extremely limited scalability, thus the performance can't be improved compared to the centralized algorithm as seen in Figure 7. This is again because of hot-spots: it is quite common that only a few frequent patterns exist. Thus, irrespective of the size of the cluster, only a few workers (equal to the number of these frequent patterns) will be used. In addition, due to the skewed nature of many graphs, load is unlikely to be well balanced among these patterns. Some patterns will typically be much more popular than others and the corresponding workers will be overloaded. Spreading the load by using techniques like work stealing is not viable since there exists no straightforward way to split the work associated with a pattern in GRAMI. The same problem holds for the other example applications we consider. For instance, in the case of Motifs, for depth equal to 3, there are only 2 patterns to process. While TLP can provide the best performance for a single thread (centralized) scenario, its lack of scalability limits the usefulness in distributed frameworks.

6.3 Arabesque: The TLE Paradigm

We now focus on evaluating the performance of Arabesque and its optimizations. While Arabesque is generic enough to describe easily most graph mining algorithms, the internals hide a powerful optimized engine with a number of innovations that allows the system to efficiently process the trillions of embeddings that graph exploration generates.

Single Thread Execution Performance: Arabesque is built from scratch as a generic distributed graph mining system. Since it has been observed that a centralized implementation can outperform distributed frameworks that utilize thousands of threads [27], we next show that the performance of Arabesque running on a single thread is comparable to the best available centralized implementations. For Motifs we use G-Tries [31] as the centralized implementation. For Cliques we use Mace [37]. For FSM we use GRAMI [14] to discover the frequent patterns and then VFLib [11] to discover the embeddings. The centralized implementations are highly optimized C/C++ implementations with the exception of GRAMI which uses Java.

For these experiments, we run Arabesque on a single worker with a single thread. We report the total time excluding the start-up and shutdown overhead to run a Giraph job in Hadoop (on average 10 seconds).

Table 2 shows a comparison between baseline single-threaded implementations and Arabesque. Even when running on a single thread, Arabesque has comparable performance or is even faster than most centralized implementations. The only exception is GRAMI, which, as discussed, uses extra application-specific TLP optimizations and solves a simpler problem by not outputting all frequent embeddings. The performance advantage of GRAMI disappears when we require discovery of the actual embeddings as we see from the running time of VFLib.

Application	Centralized Baseline	Arabesque (1 thread)
Motifs (MS=3)	G-Tries: 50s	37s
Cliques (MS=4)	Mace: 281s	385s
FSM (S=300)	Grami+VFLib: 3s + 1.8s	5s
FSM (S=220,MS=7)	Grami+VFLib: 13s + 1,800s	8,548s

Table 2: Execution times of centralized implementations and Arabesque running on a single thread. Motifs and Cliques were run with the MiCo graph, FSM with CiteSeer.

These results are a clear indicator of the efficiency of Arabesque. Despite being built with a generic framework running over Hadoop (and Java), Arabesque can achieve performance comparable, and sometimes even superior, to application-specific implementations. The main contributing factor, as we show later in this section, is that the user-defined functions in Arabesque consume an insignificant amount of CPU time. The user-defined functions steer the exploration process through a high-level API. The API abstracts away the details of the actual exploration process, which are under the control of Arabesque and can thus be efficiently implemented. This is in stark contrast to the graph processing systems analyzed in [27], where the user-defined functions perform the bulk of the computation, leaving little room for system-level optimizations.

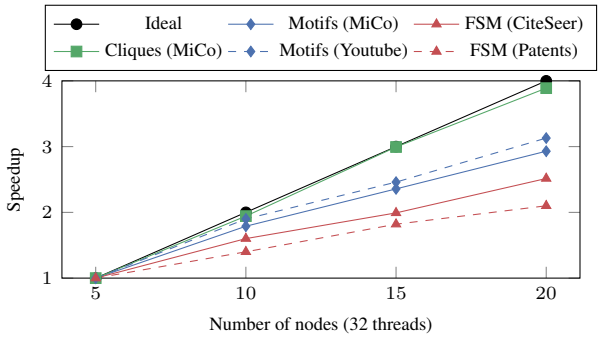


Figure 8: Scalability of Arabesque:Speedup relative to the configuration with 5 servers.

Scalability: The TLE approach of Arabesque makes it easy to scale the system to a large number of servers. We ran all three algorithms on datasets that allow computation to terminate in a feasible time on a single server while leaving sufficient work to be executed on a large cluster. Table 3 reports execution times, thus excluding setup and shutdown times, with a growing number of servers and, for reference, the execution time of the centralized baselines. Figure 8

Application - Graph	Centralized Baseline	Arabesque - Num. Servers (32 threads)				
		1	5	10	15	20
Motifs - MiCo	G-Tries: 8,664s	328s	74s	41s	31s	25s
FSM - CiteSeer	Grami+VFLib: 13s + 1,800s	431s	105s	65s	52s	41s
Cliques - MiCo	Mace: 14,901s	1,185s	272s	140s	91s	70s
Motifs - Youtube	G-Tries: *Fail*	8,995s	2,218s	1,167s	900s	709s
FSM - Patents	Grami+VFLib: 1,147s + >19h	548s	186s	132s	102s	88s

Table 3: Scalability of Arabesque - For FSM - CiteSeer, the chosen support was 220 and the search was terminated at embedding size 7, while for FSM - Patents the chosen support was 24k with no maximum embedding size.

illustrates the same results in terms of speedup, comparing distributed settings among each other.

The results show that Arabesque scales to a large number of servers. Different applications show different scalability factors. In general, applications generating more intermediate state and more patterns scale less. For example, FSM scales less because it generates many patterns and transmits a large number of embeddings that are discarded by the aggregation filter at the beginning of the next step, when aggregated metrics become available. By contrast, in Cliques we have a single pattern at each step (a clique) and fewer embeddings. The behavior of Motifs is in between. This trend is due to the characteristics of ODAGs. Arabesque constructs one ODAG per pattern, and thus as the number of patterns grows, so does the number of ODAGs. Considering the same number of embeddings, the more ODAGs they are split into, the smaller the potential for compression. Furthermore, since ODAGs are broadcast, the communication cost of transmitting embeddings increases as more servers are added, and the per-server computational cost of de-serializing and filtering out embeddings remains constant.

Despite these scalability limitations, ODAGs typically remain advantageous. We have tested the scalability of the system without ODAGs and the slope of the speedup is closer to the ideal speedup than in Figure 8. Nevertheless, this better scalability is greatly outweighed by a significant increase in the overall execution time, as we will see shortly.

ODAGs: Arabesque introduced ODAGs to compress embeddings and make it possible to mine large graphs that generate trillions of embeddings (see Section 5.2). Figure 9 shows the efficacy of ODAGs by comparing the space required to store intermediate embeddings with and without ODAGs at different exploration steps. We report the sizes of the structures listing all the embeddings at the end of each superstep. This represents the minimum space required for the embedding: after de-serialization one can expect this value to grow much larger. The results clearly show that ODAGs can reduce memory cost by several orders of magnitude even in relatively small graphs such as CiteSeer.

As with any compression technique, ODAGs trade space for computational costs, so one might wonder whether using ODAGs results in longer execution times. Indeed, the

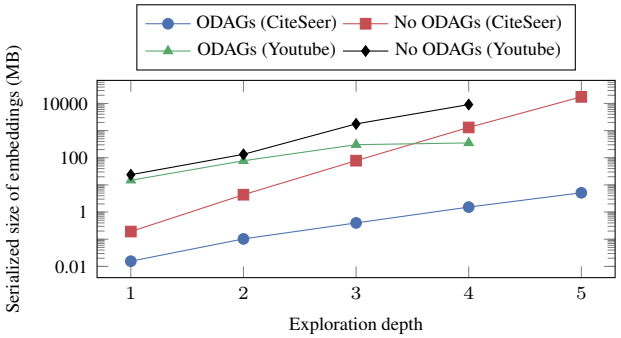


Figure 9: Compression effect of ODAGs processed at each depth. Data captured from executions of FSM on CiteSeer (S=220, MS=7) and Youtube (S=250k).

opposite holds: ODAGs significantly speed up computation, especially when running many exploration steps. Figure 10 reports the normalized execution time slowdown when ODAGs are disabled, compared to the results of Table 3. Removing ODAGs can increase execution time up to 4 times in these experiments. A more compact representation of the embeddings, in fact, results in less network overhead to transmit embeddings across servers, lower serialization costs, and less overhead due to garbage collection. We have observed, however, that in the first exploration steps with very large and sparse graphs, the overhead of constructing ODAGs outweights the cost of sending individual embeddings, because ODAGs achieve very little compression. In such cases, we can revert to using embedding lists.

Two-Level Pattern Aggregation: In Section 5.4, we introduced our novel two-level pattern-key aggregation technique to reduce the number of pattern canonicality checks, i.e., graph isomorphism, run by the system. Table 4 compares the number of checks without the optimization, which is equal to the number of embeddings, and with the optimization, which is equal to the number of quick patterns. The results show a reduction of several orders of magnitude using the optimization. For instance, for Motifs with the Youtube graph the optimization allows Arabesque to run graph isomorphism only 21 times instead of 218 billion times. The number of quick patterns is also very close to the number of actual canoni-

	Motifs-MiCo MS=3	FSM-CiteSeer S=300	FSM-CiteSeer S=220,MS=7	Motifs-MiCo MS=4	FSM-Patents S=24k	Motifs-Youtube MS=4
Embeddings	66,081,419	2,890,024	1,680,983,703	10,957,439,024	1,910,611,704	218,909,854,429
Quick patterns	3	116	1,433	21	1,800	21
Canonical patterns	2	28	97	6	1,348	6
Reduction factor	*22,027,140*x	*24,914*x	*1,173,052*x	*521,782,810*x	*1,061,451*x	*10,424,278,782*x

Table 4: Effect of two-level pattern aggregation. The results refer to the deepest exploration level.

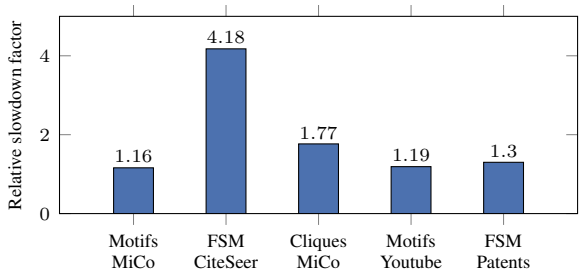


Figure 10: Slowdown factor when storing full embedding lists compared to the results of Table 3 with 20 servers.

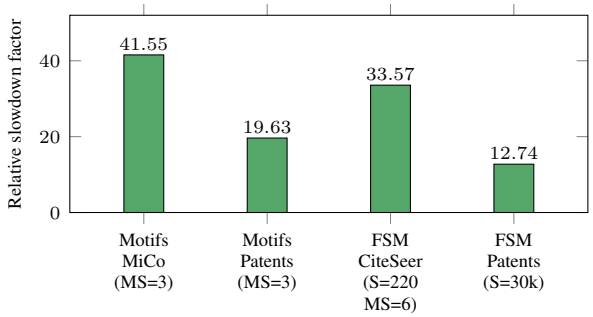


Figure 11: Slowdown factor when removing two-level pattern aggregation (not applicable to Cliques).

cal patterns, thus minimizing the required number of graph isomorphism checks.

The actual savings in terms of execution time depend not only on how often we compute graph isomorphism but also on the cost of the computation itself, which in turn depends on the complexity of the pattern. In order to take this into account, Figure 11 reports the relative execution time slowdown when two-level aggregation is disabled. We consider smaller instances than in Table 3 to keep execution times manageable; the slowdown grows with larger instances. The results show that without the optimization the system can be more than one order of magnitude slower, since it spends most of its CPU cycles on computing graph isomorphism.

Execution time breakdown: The CPU utilization breakdown of Figure 12 shows that storing, sharing, and extracting embeddings occupies a predominant fraction of CPU utilization. Embedding canonicality and pattern canonicality checking still take a significant fraction of CPU cycles even after our optimizations, showing that executing these checks efficiently is critical. Note that Cliques does not use pattern aggregation. Interestingly, the user-defined functions consume an insignificant amount of CPU, although their logic is fundamental in determining the exploration process and thus the overall system load.

Application	Time	Memory	Embeddings
Motifs-SN (MS=4)	6h 18m	110 GB	$8.4 * 10^{12}$
Cliques-SN (MS=5)	29m	50 GB	$3 * 10^{10}$
Motifs-Inst (MS=3)	10h 45m	140 GB	$5 * 10^{12}$

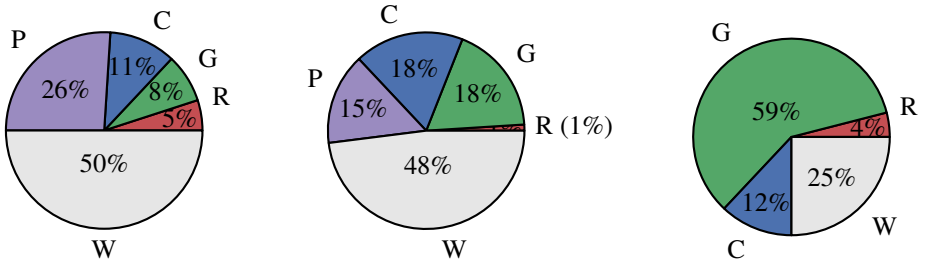
Table 5: Execution details with large graphs and 20 servers.

6.4 Large Graphs with Arabesque

We complete our evaluation by running Arabesque on large graphs and checking the limits in terms of required resources. We use the SN and the Instagram graph for this evaluation. SN is both a large and dense graph with an average degree of 79, while Instagram has close to one billion edges but is significantly less dense compared to SN. For these two graphs we don't have real world labels, so we focus on graph mining problems that look for structural patterns, such as Motifs and Cliques, rather than more inherently label-dependent problems such as FSM.

In Table 5, we report the running time, the maximum memory used and the number of interesting embeddings that Arabesque processed. For Motifs-SN, the application analyzed 8.4 trillion embeddings and ran for 6 hours 18 minutes. Cliques, as expected, posed a smaller load on the system, and it ran in half an hour, analyzing 30 billion embeddings. Instagram is a large and sparse graph, so ODAGs do not have high compression efficiency in the first exploration steps. In fact, we could not run Motifs with ODAGs and MS = 4 because it exceeds the memory resources of our servers (256 GB). Table 5 thus reports the results for MS = 3 using regular embedding lists.

Overall, the results show that even with the commodity servers that we utilize, Arabesque can process graphs that are dense and have hundreds of millions of edges and tens of million of vertices.



(a) FSM CiteSeer (S=220,MS=7)

(b) Motifs MiCo (MS=4)

(c) Cliques MiCo (MS=5)

Figure 12: CPU utilization breakdown during the superstep preceding the last one. W = Writing embeddings (ODAG creation, serialization, transfer); R = Reading embeddings (ODAG extraction); G = Generating new candidates; C = Embedding canonicality checking; P = Pattern aggregation.

7. Related Work

Over the last decades graph mining has emerged as an important research topic. Here we discuss the state-of-the-art for the graph mining problems tackled in this paper.

Centralized Algorithms: Among the most efficient methods for frequent subgraph mining is gSpan [43]. However, gSpan is designed for mining multiple input graphs and not a single large graph. When there are multiple graphs, the frequency of a pattern is simply the number of input graphs that contain it, so finding only one instance of a pattern in a graph is sufficient to make this determination. If we instead have a single input graph, we have to find multiple instances in the same graph, and this makes the problem more complex. One of the first algorithms to mine patterns from a single graph was proposed in [22]. It uses a level-wise edge growth extension strategy, but uses an expensive anti-monotonic definition of support based on the maximal independent set to find edge-disjoint embeddings. For the single large input graph setting GRAMI [14] is a recent approach that is very effective. The motif problem was introduced in [30]. The work in [31] proposes an effective approach for storing and finding motif frequencies. Listing all the maximal cliques is a well studied problem, with the Bron-Kerbosch algorithm [8] among the most efficient ones in practice. See [15] for a recent method that can handle large sparse real-world graphs.

Distributed and Parallel Approaches: Recently there have been several papers on both parallel and distributed FSM using MPI or the MapReduce framework [5, 13, 19, 23, 24, 39] as well as GPUs [21]. However, all these methods focus on the case of multiple input graphs, which is simpler as we have previously discussed. Some existing work targets graph matching, a subset of graph mining problems: given a query q, it finds all its embeddings in a distributed manner. The work in [33] uses a Pregel-based approach for graph matching in a single graph, while [44] proposes a Hadoop-based solution. For motifs, [29] proposes a multicore parallel approach, while [34] develops methods for approximate motif counting on a tightly coupled HPC system using MPI. An early work on parallel maximal clique enumeration is [12], which proposed a parallel CREW-PRAM implementation. A more recent parallel algorithm on the Cray XT4 machine was proposed in [32]. The work in [9] uses an MPI-based approach, whereas MapReduce based implementations are given in [40, 41]. The work in [4] focuses on the related problem of finding dense subgraphs using MapReduce.

8. Conclusions

In this paper, we showed that distributing graph mining tasks is far from trivial. Focusing on optimizing centralized algorithms and then considering how to convert them to distributed solutions using a TLV or TLP approach can result to scalability issues. Distributing these tasks requires a mental shift on how to approach these problems.

Arabesque represents a novel approach to graph mining problems. It is a system designed from scratch as a distributed graph mining framework. Arabesque focuses simultaneously on scalability and on providing a user-friendly simple programming API that allows non-experts to build graph mining workloads. This follows the spirit of the MapReduce and Pregel frameworks that democratized the processing and analysis of big data. We demonstrated that Arabesque's simple programming API can be used to build highly efficient distributed graph mining solutions that scale and perform very well.

Acknowledgments

We would like to thank Landon Cox for shepherding the paper, the SOSP reviewers for their valuable feedback, Ehab Abdelhamid for the GRAMI implementation, Niloth-pal Talukder for some of the datasets, Tommi Junttila for changing the license of his jbliss library, Lori Paniak and Kareem El Gebaly for the cluster setup. Carlos H. C. Teixeira would like to thank CNPq, Fapemig and Inweb for the travel support to attend the conference.

References

[1] AGGARWAL, C. C., AND WANG, H. *Managing and mining graph data*. Springer, 2010.

[2] ANCHURI, P., ZAKI, M. J., BARKOL, O., GOLAN, S., AND SHAMY, M. Approximate graph mining with label costs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2013).

[3] AVERY, C. Giraph: Large-scale graph processing infrastructure on Hadoop. *Proceedings of the Hadoop Summit* (2011).

[4] BAHMANI, B., KUMAR, R., AND VASSILVITSKII, S. Densest subgraph in streaming and MapReduce. *Proceedings of the VLDB Endowment 5*, 5 (2012).

[5] BHUIYAN, M. A., AND HASAN, M. An iterative MapReduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering 27*, 3 (March 2015).

[6] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems 30*, 1 (1998).

[7] BRINGMANN, B., AND NIJSSEN, S. What is frequent in a single graph? In *Proceedings of the Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining* (2008), Springer-Verlag.

[8] BRON, C., AND KERBOSCH, J. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM 16*, 9 (1973).

[9] CHENG, J., ZHU, L., KE, Y., AND CHU, S. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining* (2012).

[10] CHENG, X., DALE, C., AND LIU, J. Dataset for "statistics and social network of youtube videos". http://netsg.cs.sfu.ca/youtubedata/.

[11] CORDELLA, L., FOGGIA, P., SANSONE, C., AND VENTO, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26*, 10 (2004).

[12] DAHLHAUS, E., AND KARPINSKI, M. A fast parallel algorithm for computing all maximal cliques in a graph and the related problems. In *Proceedings of SWAT*, Lecture Notes in Computer Science. Springer, 1988.

[13] DI FATTA, G., AND BERTHOLD, M. R. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Transactions on Parallel and Distributed Systems 17*, 8 (2006).

[14] ELSEIDY, M., ABDELHAMID, E., SKIADOPOULOS, S., AND KALNIS, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment* (2014).

[15] EPPSTEIN, D., AND STRASH, D. Listing all maximal cliques in large sparse real-world graphs. In *Experimental Algorithms*. Springer, 2011.

[16] GAREY, M. R., AND JOHNSON, D. S. *Computers and intractability*, vol. 29. WH Freeman, 2002.

[17] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2012).

[18] H., H. B., JAFFE, A. B., AND TRAJTENBERG, M. The NBER patent citation data file: Lessons, insights and methodological tools. http://www.nber.org/patents/, 2001.

[19] HILL, S., SRICHANDAN, B., AND SUNDERRAMAN, R. An iterative MapReduce approach to frequent subgraph mining in biological datasets. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine* (2012).

[20] JUNTTILA, T., AND KASKI, P. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the SIAM Workshop on Algorithm Engineering and Experiments* (2007).

[21] KESSL, R., TALUKDER, N., ANCHURI, P., AND ZAKI, M. J. Parallel graph mining with GPUs. In *Proceedings of the International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications* (2014).

[22] KURAMOCHI, M., AND KARYPIS, G. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery 11*, 3 (2005).

[23] LIN, W., XIAO, X., AND GHINITA, G. Large-scale frequent subgraph mining in MapReduce. In *Proceedings of the IEEE International Conference on Data Engineering* (2014).

[24] LU, W., CHEN, G., TUNG, A. K., AND ZHAO, F. Efficiently extracting frequent subgraphs using MapReduce. In *Proceedings of the IEEE International Conference on Big Data* (2013).

[25] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2010).

[26] MCCUNE, R. R., WENINGER, T., AND MADEY, G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *arXiv:1507.04405* (2015).

[27] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems* (2015).

[28] MEJOVA, Y., HADDADI, H., NOULAS, A., AND WEBER, I. #Foodporn: Obesity patterns in culinary interactions. In *Proceedings of the ACM conference on Digital Health 2015* (2015).

[29] OLIVEIRA APARICIO, D., PINTO RIBEIRO, P. M., AND DA SILVA, F. M. A. Parallel subgraph counting for multicore architectures. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications* (2014).

[30] PRŽULJ, N. Biological network comparison using graphlet degree distribution. *Bioinformatics 23*, 2 (2007).

[31] RIBEIRO, P., AND SILVA, F. G-Tries: A data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery 28*, 2 (2014).

[32] SCHMIDT, M. C., SAMATOVA, N. F., THOMAS, K., AND PARK, B.-H. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing 69*, 4 (2009).

[33] SHAO, Y., CUI, B., CHEN, L., MA, L., YAO, J., AND XU, N. Parallel subgraph listing in a large-scale graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2014).

[34] SLOTA, G. M., AND MADDURI, K. Complex network analysis using parallel approximate motif counting. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium* (2014).

[35] TEIXEIRA, C. H. C., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: A system for distributed graph mining - Extended version. Technical Report, Qatar Computing Research Institute, 2015.

[36] TIAN, Y., BALMIN, A., CORSTEN, S. A., TATIKONDA, S., AND MCPHERSON, J. From "think like a vertex" to "think like a graph". *Proceedings of the VLDB Endowment 7*, 3 (2013).

[37] UNO, T. MACE: Maximal clique enumerator, ver 2.0. http://research.nii.ac.jp/~uno/code/mace.html.

[38] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990).

[39] WANG, C., AND PARTHASARATHY, S. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the Annual International Conference on Supercomputing* (2004).

[40] WU, B., YANG, S., ZHAO, H., AND WANG, B. A distributed algorithm to enumerate all maximal cliques in MapReduce. In *Proceedings of the International Conference on Frontier of Computer Science and Technology* (2009).

[41] XIANG, J., GUO, C., AND ABOULNAGA, A. Scalable maximum clique computation using MapReduce. In *Proceedings of the IEEE International Coference on Data Engineering* (2013).

[42] YAN, D., CHENG, J., LU, Y., AND NG, W. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment 7*, 14 (2014).

[43] YAN, X., AND HAN, J. gSpan: Graph-based substructure pattern mining. In *Proceedings of the IEEE International Conference on Data Mining* (2002).

[44] ZHAO, Z., WANG, G., BUTT, A. R., KHAN, M., KUMAR, V. A., AND MARATHE, M. V. Sahad: Subgraph analysis in massive networks using Hadoop. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium* (2012).