Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine

Cheng Zhao[1,2], Zhibin Zhang[1], Peng Xu[1,2], Tianqi Zheng[1,2], and Jiafeng Guo[1]

[1]Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
[1]{zhaocheng,zhangzhibin,xupeng16,zhengtianqi,guojiafeng}@ict.ac.cn

***Abstract*—Graph mining is one of the most important categories of graph algorithms. However, exploring the subgraphs of an input graph produces a huge amount of intermediate data. The "think like a vertex" programming paradigm, pioneered by Pregel, cannot readily formulate mining problems, which is designed to produce graph computation problems like PageRank. Existing mining systems like Arabesque and RStream need large amounts of computing and memory resources.**

In this paper, we present Kaleido, an efficient single machine, out-of-core graph mining system which treats disks as an extension of memory. Kaleido treats intermediate data in graph mining tasks as a tensor and adopts a succinct data structure for the intermediate data. Kaleido implements half-memory-half-disk storage for storing large intermediate data, which treats the disk as an extension of the memory. Kaleido adopts a lightweight isomorphism checking strategy which uses an eigenvalue-based algorithm for small graphs and solves tree isomorphism for the other graphs. Comparing with two state-of-the-art mining systems, Arabesque and RStream, Kaleido outperforms them by a GeoMean *13.2*× and *64.8*× respectively.

***Index Terms*—graph mining, exploration, isomorphism, out-of-core**

I. Introduction

Graphs data is ubiquitous in a broad range of fields such as social networks, web networks, financial networks, biological networks, and the analysis of graphs is becoming increasingly important. Generally, we divide graph analysis problems into two major types, graph computation and graph mining. Graph computation aims to compute some meaningful values of vertices in a graph. For example, calculate the PageRank [26] value of a web graph to obtain the top-k valuable web pages; given two vertices in an input graph, calculate the shortest path between them. In contrast, graph mining aims to discover structural patterns to meet the user's interest criteria. For example, mine frequent subgraphs in the biological data to discover highest gene expression [17]; extract the frequency distribution of all motifs that occur in PPI network [28]; discover cliques in financial networks to detect frauds [11].

A. Problem statement

Graph computation problems can be represented through linear algebra over an adjacency matrix-based representation of the graph. Many practical solutions, like PowerGraph [14], Ligra [35], GraphX [15], Chaos [31], Gemini [39], etc., follow a simple "think like a vertex (TLV)" programming paradigm pioneered by Pregel [23], in which each vertex of the input graph is a processing element holding local state and communicating with its neighbors. TLV is suitable for applications which perform value computations around vertices of the input graph, like PageRank.

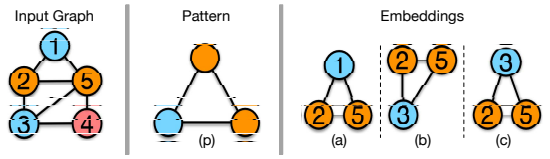


Fig. 1. Graph mining concepts: a graph, a pattern and embeddings. Numbers denote vertex ids; colors represent labels. Pattern p is a template graph. Graph a, b and c are instances of pattern p in the input graph. These instances are called embeddings. Isomorphic embeddings a and b have same pattern p. The same embedding b and c also are called automorphic.

There are several significant graph mining problems, such as frequent subgraph mining, which aim to discover subgraphs of the input graph and collect the statistics of subgraphs which meet the user's criteria. However, the TLV programming paradigm cannot readily formulate these graph mining problems. Given an input graph, these graph mining problems often require exponential combinations around vertices and edges to explore all possible subgraphs. Treating each subgraph as the basic processing element is more convenient, which is also known as the *subgraph-centric model*.

In this paper, we use *pattern* and *embedding* to denote two types of subgraphs in an input graph. A *pattern* is a template, while an *embedding* is an instance. We denote k-embedding for an embedding contains k vertices. Embeddings are *isomorphic* if they contain different vertices and edges but they have the same pattern. Figure 1 illustrates these concepts in graph mining problems. To identify which pattern is frequent in an input graph, we should explore all embeddings, then patternize each embedding and statistic all patterns. The exploration of subgraphs can be executed as *vertex-induced* and *edge-induced*. A *vertex-induced* exploration expands one vertex to an embedding in each iteration, while an *edge-induced* exploration expands one edge.

The first challenge in graph mining applications is *how to build a compact data structure for the intermediate data (embeddings) and process them efficiently*. In querying a k-vertex-pattern in a graph with N distinct vertices, the time complexity is $O(N \cdot \bar{d}^{k-1})$, in which $\bar{d}$ is the average degree

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00064

IEEE computer society

of the graph [2]. Generally, we have up to $O(N \cdot \bar{d}^{k-1})$ different embeddings of size k in this graph. For example, the exploration of 4-embeddings over Patent (3.8 M vertices, 16.5 M edges) [22] produces 13.5 billion embeddings. The second challenge is *how to efficiently test embedding isomorphism in computing patterns*. The graph isomorphism problem (GI) belongs to NP but it is unknown whether GI belongs to NP-complete and no polynomial time algorithm is known [34]. The fastest proven running time for GI has stood at $e^{O(\sqrt{n \log n})}$ [6], in which n is the vertex number of the graph. There are state-of-the-art softwares for GI which solve GI in a quasi-polynomial-time, such as Nauty [24] and Bliss [19]. These softwares use a practical approach named "canonical labeling", in which a graph is relabeled and isomorphic graphs can be identified.

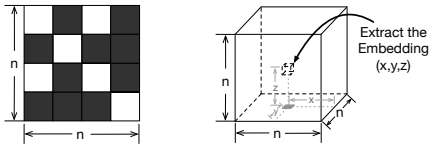
B. Limitations of State-of-the-Art Systems

Recent graph mining systems use declarative models to solve mining problems. Arabesque [36] proposes a natural programming paradigm, "think like an embedding (TLE)", which is also called the *subgraph-centric* model. RStream [37] employs a GRAS programming model that uses a combination of "gather-apply-scatter" (GAS) and relational algebra to support mining algorithms.

Arabesque is a Giraph-based distributed graph mining system. Arabesque designs a prefix-tree-liked embeddings data structure. It stores k arrays for k-embeddings, in which 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 exists at least one canonical embedding with v and u in position i and $i+1$ respectively in the original set. However, in the pattern aggregation phase, an extra canonically checking for each embedding is inevitable. For the experiment of Arabesque, the extra checking still accounts for around 5% of the runtime in mining applications. On the other hand, Arabesque is a distributed system which suffers from the unbalanced graph partitioning and computation. For example, running 3-Motif over Patent, Arabesque uses 60.2s, 47.0s, 46.6s, 46.4s on 1, 2, 4 and 8 nodes respectively. The linear scalability is limited by the unbalanced partition.

RStream is a single-machine graph mining system based on X-Stream [32]. It only supports edge-induced embedding exploration. When solving some vertex-based applications, like the motif counting and the clique discovery, it needs more iterations and more disk I/O. For example, to find 4-motifs in an input graph, RStream iterates 6 times to explore all kinds of 4-motifs ($\binom{4}{2} = 6$). To explore all possible embeddings, RStream executes the operation of all-join in the relational algebra, which produces a huge amount of intermediate data. For example, running the 4-motifs counting on RStream over MiCo (100 K vertices, 1.1 M edges) [12] produces around 1.64 TB intermediate data, while the amount of 4-motifs in MiCo is around 11 billion.

Both Arabesque and RStream use a graph library, Bliss [19], which is an open-source tool for computing graph isomorphism problems. For each input graph, Bliss builds a search tree, then permutes this graph to obtain its canonical form. If two graphs contain the canonical form, they are automorphic. However, building the search tree brings frequently memory allocating and de-allocating which slow down the processing of hashing patterns and consumes a huge amount of memory. For example, the overhead of allocation and de-allocation are more than 53% in running 3-FSM over Patent graph (37 labels) with support 1 and it consumes 16.1 GB memory for total 25,083 patterns. For other graph isomorphism softwares, like Nauty [24] and Trace [27] both of which are implementations of the search tree as well, the memory consumption is even higher. Besides, both Nauty and Trace only suit for the unlabeled graphs.



(a) Adjacency matrix (b) 3-Embedding cube

Fig. 2. Adjacency matrix and the 3-embedding cube. The black blocks in Figure (a) indicate edges in the graph. Figure (b) indicates a cube (tensor) of 3-embeddings and an operation of extracting an arbitrary embedding.

C. Our Approaches

To address the limitations of existing systems, we propose Kaleido, a single-machine, out-of-core graph mining system. Kaleido adopts the subgraph-centric computation model and presents a general programming API which fits most of graph mining applications.

Kaleido treats the intermediate data of the i^{th} exploration as *an i-dimension tensor* and designs *a succinct data structure for the intermediate data.* Intuitively, the set of 1-embeddings is the vertex set of the input graph; the set of 2-embeddings is the edge set without duplicated edges of the input graph, which can be represented by an adjacency matrix (see Figure 2a); the set of 3-embeddings contains 3-chains and triangles, which can be represented by a cube (see Figure 2b). In other words, each vertex-induced expanding of embeddings is equivalent to ascending a dimension for the intermediate data. Inspired by compressed sparse column (CSC) for sparse matrices [33], we design a level-by-level succinct data structure of intermediate embeddings, which is called *compressed sparse embedding* (CSE). Each iteration of the exploration ascends a dimension for the intermediate data and expands a level in CSE.

Kaleido adopts *a hybrid storage for the intermediate embedding* when the scale of the input graph or the exploration depth increases. According to the level-by-level structure of CSE, when the memory is insufficient to afford the whole intermediate data, the hybrid storage stores large levels of CSE on disk. To balance the workload in processing the intermediate data on disk, Kaleido predicts the capacity of embeddings candidate in the next iteration, then divides the exploring tasks to each thread evenly according to the prediction.

Kaleido designs *a lightweight isomorphism checking strategy* to solve the labeled graph isomorphism problem. The

strategy is departed into three parts: (i) eigenvalue-based isomorphism checking algorithm for small embeddings; (ii) specially checking isomorphism for tree-shape-embeddings (k-embedding with $k-1$ edges) and (iii) using Bliss to check the rest embeddings. The key optimization in graph mining applications is to prune as early as possible, therefore Kaleido adopts an efficient algorithm to check isomorphism over small graphs. Harary *et al.* [16] proved that if two k-embeddings ($k < 8$) have the same vertex degrees and the same eigenvalues, they are isomorphic. However, Harary *et al.* only discussed the isomorphism of unlabeled graphs. Kaleido combines the label information and the degree information of vertices in each pattern and eigenvalues of the adjacency matrix to identify isomorphic graphs. The space usage of Kaleido is $O(n)$, while the space usage of Bliss is $O(n+m)$ in which n, m are the number of vertices and edges of the input graph respectively. On the other hand, we observe that the tree-shape-embeddings hold a large proportion in exploring natural graphs (67.8% and 98.7% tree-shape-embeddings in the 10th exploration of Patent in the vertex-induced and the edge-induced exploration respectively). The time complexity of state-of-the-art solution of tree isomorphism is $O(n)$, in which n is the number of vertices in the tree [8]. Kaleido extends the tree isomorphism algorithm to support labeled trees.

To summarize, we make the following contributions:

- We design an API for popular graph mining applications, which enables embedding exploration and pattern aggregation to be expressed effectively (Section III).
- We propose a novel succinct embeddings data structure, compressed sparse embedding. We implement hybrid storage for the large intermediate data in using disks and design a prediction strategy for load balancing (Section IV).
- We propose a lightweight graph isomorphism checking strategy for solving graph mining problems. We design an eigenvalue-based algorithm to solve graph isomorphism for small labeled graphs. We analyze the proportion of tree-shape-embeddings in the exploration and extend the classic tree isomorphism algorithm (Section V).

II. Preliminaries

A *graph* $G = (V, E, L)$ consists of a set of vertices V, a set of edges E and a labeling function L that assigns labels to vertices and edges. A graph $G' = (V', E', L')$ is a *subgraph* of graph $G = (V, E, L)$, i.e., $V' \subseteq V$, $E' \subseteq E$ and $L'(v) = L(v), \forall v \in V'$. A *pattern* is a template graph, while an *embedding* is an instance. In this paper, the vertex-induced embedding is noted as $e = \langle v_1, ..., v_k \rangle$. If an embedding contains k vertices, we say that the size of embedding e is k. The edge-induced embedding is analogous.

Definition 1. *We say that a graph $G_a = (V_a, E_a, L_a)$ is isomorphic to another graph $G_b = (V_b, E_b, L_b)$ if and only if there exists a bijection f_{ab} between G_a and G_b, such that (i) $L_a(v) = L_b(f_{ab}(v)), \forall v \in V_a$, and (ii) $(f_{ab}(u), f_{ab}(v)) \in E_b$ and $L_a(u, v) = L_b(f_{ab}(u), f_{ab}(v)), \forall (u, v) \in E_a$.*

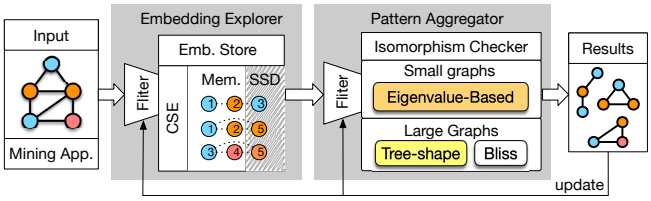


Fig. 3. Architecture of Kaleido.

Two graphs are *automorphic* if and only if they contain the same edges and vertices. As shown by Figure 1, subgraphs b and c contain the same edges and vertices, thus they are automorphic.

Definition 2. *We say an embedding $e = \langle v_1, ..., v_n \rangle$ of graph $G = (V, E)$ is canonical if (i) $\forall i > 1$ it holds $v_i > v_1$; (ii) $\forall i > 1, \exists j < i$ satisfies that $(v_j, v_i) \in E$; (iii) $\forall v_a, v_b, v_c$ if $a < b < c$, $(v_a, v_c) \in E$ and $\nexists d < a$ satisfies that $(v_d, v_c) \in E$, it holds $v_b < v_c$.*

In other words, if an embedding is canonical, it should hold the following three properties. (i) The id of the first vertex in the embedding is the minimum value. (ii) Each vertex in the embedding must be a neighbor of the vertex which is indexed a smaller id, except the first vertex. (iii) There exists an edge in the embedding, $(v_a, v_c), a < c$ and all of vertices before v_a are not neighbor of v_c, therefore if any vertex exists between v_a and v_c, it must holds that $v_b < v_c$.

III. System Architecture and API

We now present the system architecture of Kaleido, demonstrate the API for graph mining problems and introduce implementations of popular graph mining applications.

Kaleido adopts the subgraph-centric computation model and divides the processing of the graph mining problems into the *embedding exploration* phase and the *pattern aggregation* phase. Figure 3 illustrates the architecture of Kaleido. To execute a mining application, Kaleido reads the input graph and mines it in several iterations. In each iteration, *Embedding Explorer* explores and stores all possible embeddings under the user-defined filter firstly; next, *Pattern Aggregator* computes the pattern of each embedding and aggregates them; if the end condition is reached, Kaleido outputs the results; otherwise, *explorer* and *aggregator* update their corresponding filters in the next iteration according to the results and the user's criteria.

The API of Kaleido is illustrated in Listing 1. `EmbeddingFilter` works to exploring $(k+1)$-embeddings from k-embeddings: the `Embedding e` is a k-embedding and the `Vertex v` (`Edge <u, v>` in the edge-induced exploration) is a candidate vertex which is a neighbor of `e` normally. In addition, the canonical filter is built-in. `Pattern Filter` works to aggregating patterns, in order to prune ineligible patterns. `AggregatingMapper` and `AggregatingReducer` (`Mapper` and `Reducer` in short) must be implemented by customized applications. `Mapper` is calculated in `ResultAggregator` concurrently. `Reducer` aggregates `PatternMaps` returned by `Mapper`, and prunes

patterns which are incompatible of `PatternFilter`, then returns results in form of `PatternMap`.

Listing 1. Kaleido API

```
// Optional user defined filter functions
bool EmbeddingFilter(Embedding e, Vertex v)
bool EmbeddingFilter(Embedding e, Edge <u,v>)
bool PatternFilter(Pattern p)

// 2 functions of aggregation phases
PatternMap AggregatingMapper(Embedding e)
PatternMap AggregatingReducer(List<PatternMap>
    pMaps, PatternFilter pFilter)

// Main processing function in applications
List<Embedding> Init(Graph g, int depth)
List<Embedding> EmbeddingsExplorer(Graph g,
    List<Embedding>, EmbFilter eFilter)
PatternMap ResultAggregator(AggregatingMapper
    mapper, AggregatingReducer reducer)
```

Frequent Subgraph Mining. To early prune infrequent patterns, we use the *minimum image-based (MNI) support* [7] as the frequency of each pattern, which counting the minimum number of distinct mappings for any vertex in the pattern. The support measure is anti-monotonic. We implement an edge-induced version of FSM in Kaleido as illustrated in List 2. Given an input graph, the edge number of the query pattern k and a threshold of the support, it returns frequent k-patterns whose support is beyond the threshold. To prune infrequent patterns, we loop `Mapper` and `Reducer` in each iteration. In each iteration of embedding exploration, `EmbeddingExplorer` expands each embedding by adding a frequent edge. `EmbeddingFilter` checks if the candidate edge is frequent. `Mapper` patternizes each embedding, calculates the MNI support for each pattern. `Reducer` prunes infrequent patterns and its corresponding embeddings and returns all the frequent patterns in the last iteration.

Listing 2. FSM (edge-induced exploration)

```
List<Embedding> Init(Graph g, int depth) {
     return frequentEdges;}
bool EmbeddingFilter(Embedding e, Edge <u,v>) {
     return isFrequent(<u,v>);}
bool PatternFilter(Pattern p) {
     return pMap[p] >= threshold;}
PatternMap AggregatingMapper(Embedding e) {
     Pattern p = toPattern(e);
     Support s = MNISupport(p, e);
     return pattern2SupportMap;}
PatternMap AggregatingReducer(List<PatternMap>
    pMaps, PatternFilter pFilter) {
     PatternMap pMap = MergePatternMap(pMaps,
         pFilter);
     return pMap;}
```

Motif Counting. This application counts the frequency of each k-motif in the given graph . As we know exactly each shape of k-motifs like (2 kinds of 3-motifs, 6 kinds of 4-motifs and 21 kinds of 5-motifs, etc.), we stop embeddings generating if $(k-1)$-embeddings are generated. `Mapper` explores all canonical k-embeddings from each $(k-1)$-embedding, then calculates the hash value of each k-embedding. `Reducer` aggregates k-motifs.

Clique Discovery. This application discoveries all k-cliques in the input graph. `EmbeddingFilter(e,v)` checks if the candidate vertex v is neighbor of each vertex in embedding e. `EmbeddingExplorer` prunes illegal embeddings and explores all k-cliques after $k-1$ iterations and returns them. `Reducer` return k-cliques.

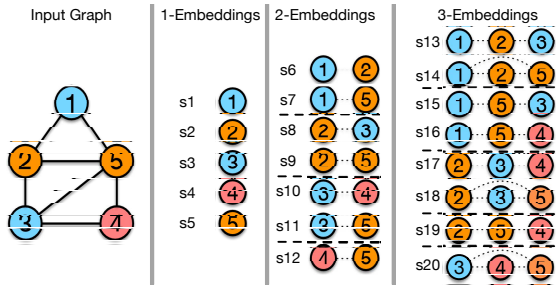


Fig. 4. Procedure of vertex-based generating canonical 3-embeddings. 2-embeddings and 3-embeddings in the figure are arrays of vertices ids, and the order of each embedding is immutable.

IV. Embedding Exploration Phase

In this section, we introduce the procedure of the embedding exploration phase including the design of the compact data structure of embeddings and solutions in facing out of memory.

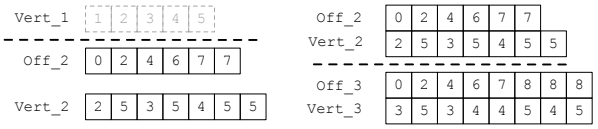
A. The Procedure of the Exploration

Given the size of embedding, the goal is to generate all of the possible and unique embeddings. According to the user's criteria, we eliminate embeddings which are ineligible. We introduce the *canonical filter* which guarantees that embeddings are complete and unique. According to Definition 2, 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 of e with the smallest id that has not been visited yet.

Figure 4 illustrates the process of a series of vertex-based explorations to 3-embeddings. Without loss of generality, consider an exploration of a 2-embedding $s_8 = \langle 2,3 \rangle$. First, neighbors or candidates of s_8 in G are $\{1,4,5\}$, thus possible 3-embeddings generated by s_8 are $\langle 2,3,1 \rangle$, $\langle 2,3,4 \rangle$ and $\langle 2,3,5 \rangle$. This step guarantees the completeness of this exploration. Next, according to Definition 2, $\langle 2,3,1 \rangle$ does not satisfies property (i) of canonical embedding, for $1 < 2$ and 2 is the first vertex of s_8, while both $\langle 2,3,4 \rangle$ and $\langle 2,3,5 \rangle$ are canonical. Therefore s_{17} and s_{18} are generated.

As discussed in Section I, the complexity of exploring k-embeddings is $O(N \cdot \bar{d}^{k-1})$. The goal of storing embeddings is divided into two parts: (i) minimizing memory usage and (ii) extracting an arbitrary embedding as fast as possible. Like an adjacency matrix form of a graph, a k-embedding set can be treated as an adjacency k-dimension tensor (see Figure 2). Kaleido stores the graph structure in *compressed sparse column (CSC)*, which is equivalent to the sparse adjacency matrix of the graph. Inspired by CSC, we design a succinct data structure for embeddings, which is called *compressed sparse embedding (CSE)*. If a k-embedding set is stored in CSE, we call it k-CSE.

As illustrated in Figure 5, Kaleido stores embeddings level-by-level. In each level, the structure of embeddings is stored in two arrays. Vertex array ($vert_l$) indicates the last vertex



(a) 2-Embeddings (b) 3-Embeddings

Fig. 5. The structure of compressed sparse embedding (CSE). The gray array $vert_1$ does not really store in Kaleido; it indicates the relationship between vertex array and offset array. The dotted lines divide CSE into different levels.

of each embedding in level l. Offset array (off_l) indicates the start offset $off_l(i)$ and end offset $off_l(i+1)$ in vertex array of level l. A slice of vertex array, $\big[off_l(i), off_l(i+1)\big)$, indicates that these vertices possesses same embedding prefix. For example, in Figure 5a, the first two elements of offset array are 0 and 2 which indicates a slice of vertex array $\{vert_2(i) | i \geq 0, i < 2\} = \{2, 5\}$. It correspond to s_6 and s_7 illustrated in Figure 4, which possess the same embedding prefix $\{1\}$. Therefore each vertex in vertex array corresponds to a unique embedding in the current level and an embedding prefix of next level. Therefore the length of vertex array in level i is equal to the length of offset array in level $i+1$ minus 1 (to compute conveniently, the last element indicate the length of $vert_i$).

Now given an arbitrary offset of vertex array in level k, we can obtain the k-embedding corresponding to this offset. For example, given offset 5 of vertex array in level 3 in Figure 5b, the goal is to find the corresponded 3-embedding. First, we note the last element of this embedding is $vert_3(5) = 5$, $\langle \cdot, \cdot, 5 \rangle$. Then we find that offset 5 is greater than $off_3(2) = 4$ and less than $off_3(3) = 6$, thus the coordinate of offset 5 in offset array in level 3 is 2. Next, we do this processing again in level 2, and the offset of the vertex array is 2. At last, we obtain the 3-embedding $\langle 2, 3, 5 \rangle$, which corresponds to s_{18} in Figure 4.

Complexity: Each iteration of the embedding exploration extends $O(\bar{d})$ space ($\bar{d}$ is the average of vertex degree and $\bar{d} \propto |E|/|V|$). Thus the space complexity of k-CSE is $O(|E|^{k-1}/|V|^{k-2})$. Given an arbitrary offset of vertex array in level k, the time complexity of obtain the corresponding embedding is $O(k \log \bar{d}) = O(\log(|E|/|V|))$.

B. Hybrid Storage

According to the space complexity of CSE, exploring $(k+1)$-embeddings from k-CSE needs extra $O(|E|^k/|V|^{k-1})$ space. The memory would be insufficient when the exploration depth increases. Thanks to the level-by-level structure of CSE, Kaleido stores large levels of CSE on disk intuitively. We call this half-memory-half-disk storage the hybrid storage.

First, Kaleido partitions $vert_k$ into several parts continuously and evenly and assigns to each thread. Then each thread calculates the $(k+1)^{th}$ elements of each k-embedding and records the offset when all canonical candidates of a k-embedding are enumerated. Finally, each thread appends their part of $vert_{k+1}$ to the writing queue and the writing queue flushes these parts to disk (see Figure 6). If memory is sufficient, Kaleido merges t parts of off_{k+1} in memory,

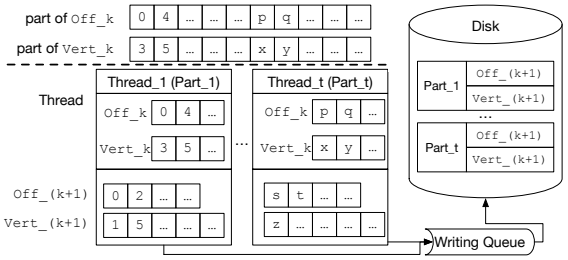


Fig. 6. Exploration on Hybrid CSE. The first k levels are stored in memory. The $(k+1)^{th}$ level is stored on disk in t parts (in this example, t equals to the thread number).

otherwise appends each part of off_{k+1} to the writing queue. When Kaleido explores $(k+2)$-embeddings and constructs the $(k+2)^{th}$ level of embeddings, load the first part of $vert_{k+1}$ and off_{k+1} (if exists) on disk. Then Kaleido executes the former process again and stores $vert_{k+2}$ and off_{k+2} to disk part by part, until it finishes the last part of $vert_{k+1}$.

To explore deeper embeddings or process embeddings in the hybrid storage, Kaleido adopts the sliding window strategy to hide the overhead of I/O. When processing the hybrid storage embeddings, Kaleido maintains h windows for h levels stored on disk. Each window respectively loads two parts of a level of CSE, which are produced by t threads as shown in Figure 6. When all the first parts (main part) of h windows are loaded, Kaleido processes all embeddings in current windows in parallel, while the h windows load the second parts (candidate part) in its corresponding level. If the main part of a window is processed, Kaleido slides this window to the next position (swaps the main part and the candidate part, then abandons the candidate part). Repeat this procedure until all parts on disk are processed.

Load-balance of Hybrid Storage. In each iteration of the embedding exploration, Kaleido expands a neighbor vertex or edge for each embedding. Similar to the definition of the vertex degree, We say an embedding degree is the neighbors' number of the embedding. One of the hallmark properties of natural graphs is their skewed power-law degree distribution [13]. The degree distribution of embeddings is also skewed power-law distribution. When the RAM can afford the embeddings data, Kaleido adopts a work-steal strategy to deal with the load-balance problem in the exploration. However, when the RAM is insufficient, Kaleido stores the high-level embeddings on disk in several parts. The unbalanced partition strategy of the embedding exploration would produce huge parts which cannot load to the memory once. The work-steal strategy can only balance the execution of the exploration but cannot balance the size of each part.

To balance the workload in the exploration of the $(k+1)^{th}$ level, Kaleido predicts the size of $vert_{k+1}$. Figure 7 illustrates an example of the prediction. According to the structure of CSE, the neighbor set of the embedding $\langle 1, 2, 3 \rangle$ is the union of the neighbor set of $\langle 1, 2 \rangle$ and the neighbor set of $\langle 3 \rangle$. From offset arrays in CSE, we easily obtain the degree of $\langle 1, 2 \rangle$ and 3. Kaleido predicts the candidate size accurately by merging the two sources of the candidate. The time complexity

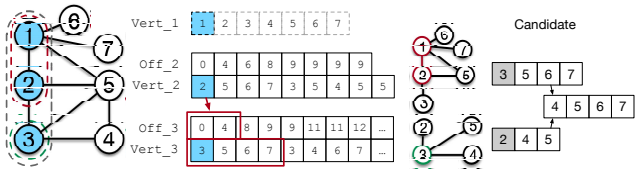


Fig. 7. An example of the prediction of the candidate size of embedding $\langle 1,2,3\rangle$. Candidates of $\langle 1,2,3\rangle$ is the union of the neighbor set of $\langle 1,2\rangle$ and the neighbor set of $\langle 3\rangle$.

of the merging is $O(\bar{d})$. According to the prediction, Kaleido partitions the exploration tasks evenly to each thread.

V. Pattern Aggregation Phase

After k iterations of the embedding exploration, Kaleido collects all possible canonical embeddings in the input graph, whose size is no more than k. Then in the pattern aggregation phase, Kaleido calculates the pattern of each embedding and aggregates them.

In this section, we introduce the design of a compact data structure for the pattern and a lightweight strategy for checking graph isomorphism. The solution of GI problem is departed into three parts: (i) eigenvalue-based isomorphism checking algorithm for small embeddings ($k < 8$); (ii) specially checking isomorphism for tree-shape-embeddings and (iii) using Bliss to check the rest embeddings.

A. The Compact Pattern

Patterns are stored in a simple compact data structure. The data structure contains each pattern's label information and structural information. Generally, we use an adjacency matrix to indicate the structural information of this pattern and a label array to indicate vertex labels. The order of labels matches with an adjacency matrix. Kaleido stores the up-triangle part of adjacency matrix in the form of 1-dimension array and stores it as a bitmap. Obviously, storing a k-pattern in this data structure needs a label array whose size is k, and a bitmap whose size is $\frac{1}{2}(k(k-1))$.

B. Eigenvalue-based isomorphism Checking

Algorithm 1 illustrates the solution of the GI problem in Kaleido where the size of embedding is less than 8. Kaleido maintains the vertex label array L in ascending order (lines 30-31) and the degrees (D) of the same label vertices in ascending order as well (lines 32-33). Note that `Swap` function also maintains the adjacency matrix $\mathbf{A}$, so that the vertex order in $\mathbf{A}$ is consisting with L and D. Then Kaleido builds a weighted adjacency matrix $\mathbf{M}$ (line 34, lines 12-18) whose edge weights is a concatenation of two vertex labels (lines 16-17). Note that label l_i is no more than label l_j after the sorting in lines 29-33. Next, Kaleido calculates eigenvalues of the matrix $\mathbf{M}$. To simplify the calculation of eigenvalues, Kaleido calculates the normalized characteristic polynomial of the matrix $\mathbf{M}$ by the Faddeev-LeVerrier algorithm (line 35, lines 19-26). Finally, Kaleido compares the label array L, the degree array D and the characteristic polynomial P of two input embeddings to check the isomorphism. Note that for unlabeled graphs, Kaleido sets the label of each vertex as a fixed value. For edge labeled graphs, Kaleido sets values of the weighted adjacency matrix as edge labels of the input graph (line 16-17).

Algorithm 1: The graph isomorphism checking in Kaleido

Input: Embeddings e_a, e_b
Output: *true* iff e_a is isomorphic to e_b
1 **Func** `Init(e)`:
2 Label array $L \leftarrow \{l_i = label(v_i) | v_i \in e, \forall i \in [1,k]\}$
3 Adjacency matrix $\mathbf{A} \leftarrow \{a_{i,j} = CheckLink(v_i, v_j) | v_i, v_j \in e, i < j, \forall i,j \in [1,k]\}$
4 Degree array $D \leftarrow \{d_i = \deg(v_i) | v_i \in e, \forall i \in [1,k]\}$
5 **return** $L, \mathbf{A}, D$
6 **Func** `Swap(i, j)`:
7 Swap l_i and l_j
8 **for** $1 \le t \le k$ **do**
9 Swap $a_{i,t}$ and $a_{j,t}$
10 Swap $a_{t,i}$ and $a_{t,j}$
11 Swap d_i and d_j
12 **Func** `WeightedAdjMatrix(L, A)`:
13 $\mathbf{M} \leftarrow \{m_{i,j} = 0 | \forall i,j \in [1,k]\}$
14 **for** $1 \le i < j \le n$ **do**
15 **if** $a_{i,j} = 1$ **then**
16 $m_{i,j} \leftarrow l_i | l_j$
17 $m_{j,i} \leftarrow l_i | l_j$
18 **return** $\mathbf{M}$
19 **Func** `CharPloynomical(M)`:
20 Characteristic polynomial $P \leftarrow \{p_i = 0 | \forall i \in [1,k]\}$
21 $\mathbf{C} \leftarrow \mathbf{M}$
22 **for** $1 \le i \le k$ **do**
23 **if** $i > 1$ **then**
24 $\mathbf{C} \leftarrow \mathbf{M} \cdot (\mathbf{C} + p_{k-i+1}\mathbf{I}_k)$
25 $p_{i-k} = -\frac{tr(\mathbf{C})}{k}$
26 **return** P
27 **Func** `Eigen(e)`:
28 $L, \mathbf{A}, D \leftarrow$ `Init(e)`
29 **for** $1 \le i < j \le n$ **do**
30 **if** $l_i > l_j$ **then**
31 `Swap(i, j)`
32 **else if** $l_i = l_j$ *and* $d_i > d_j$ **then**
33 `Swap(i, j)`
34 $\mathbf{M} \leftarrow$ `WeightedAdjMatrix(L, A)`
35 $P \leftarrow$ `CharPloynomical(M)`
36 **return** L, D, P
37 $L_a, D_a, P_a \leftarrow$ `Eigen(`e_a`)`
38 $L_b, D_b, P_b \leftarrow$ `Eigen(`e_b`)`
39 **return** $L_a = L_b$ *and* $D_a = D_b$ *and* $P_a = P_b$

Theorem 1. *Let $e_a = (V_a, E_a)$ and $e_b = (V_b, E_b)$ be two k-embeddings of an undirected graph G, $k < 8$. Let L, D, P denote the label array, the degree array and the characteristic polynomial of the weighted adjacency matrix of an embedding e respectively and are calculated by `Eigen(e)` in Algorithm 1 (lines 27-36). Embedding e_a is isomorphic to embedding e_b if and only if $L_a = L_b$, $D_a = D_b$ and $P_a = P_b$.*

Proof. **Necessity.** In Algorithm 1, $\mathbf{M}_a$ and $\mathbf{M}_b$ denote the weighted adjacency matrices of the input embedding e_a and e_b respectively. According to Definition 1, the isomorphism leads to $L_a = L_b$, $D_a = D_b$ and $\mathbf{M}_a$ can be transformed to $\mathbf{M}_b$ by a similarity transformation. Thus there exists a permutation matrix $\mathbf{P}$, such that $\mathbf{M}_b = \mathbf{P}\mathbf{M}_a\mathbf{P}^{-1}$. Thus $\mathbf{M}_a$ and $\mathbf{M}_b$ are similar. Similar matrices have the same eigenvalues. Note that,

in this paper, we say that graphs have the same eigenvalues implies that their algebraic multiplicities are the same as well. Therefore $P_a = P_b$.

Sufficiency. According to the necessity, if eigenvalues of two patterns are different, it holds that these patterns are non-isomorphic. Unfortunately, there exist non-isomorphic graphs which have the same eigenvalues. Harary *et at.* [16] listed counterexamples of non-isomorphic graphs with the same eigenvalues.Harary *et at.* also proved that *if two k-embeddings ($k < 6$) have the same eigenvalues, they are isomorphic; if two k-embeddings ($k < 8$) have the same vertex degrees and the same eigenvalues, they are isomorphic*. The proof of Harary *et at.* is under the constraint that input graphs are unlabeled. Next, we prove the sufficiency in labeled graphs.

Note $\epsilon_a = (V_{\epsilon_a}, E_{\epsilon_a})$ as a graph which contains the same vertex permutation with e_a. The vertices of ϵ_a are unlabeled, while edges are labeled and the weighted adjacency matrix is the same as $\mathbf{M}_a$. Note $\epsilon_b = (V_{\epsilon_b}, E_{\epsilon_b})$ symmetrically. According to the conclusion of Harary *et at.*, $D_a = D_b$ and $P_a = P_b$ guarantee that ϵ_a and ϵ_b are isomorphic. Thus, there exists a bijection, $f(u) = v, \forall u \in V_{\epsilon_a}, \forall v \in V_{\epsilon_b}$, between V_{ϵ_a} and V_{ϵ_b}, such that $(f(u), f(v)) \in E_{\epsilon_b}$ and $L(u, v) = L(f(u), f(v)), \forall (u, v) \in E_{\epsilon_a}$. Note that the label of an edge in ϵ_a or ϵ_b indicates labels of vertices which are linked by this edge (Algorithm 1 lines 16-17). Then combining $L_a = L_b$ and $D_a = D_b$, it leads that $L(u) = L(f(u)), \forall u \in V_a$. Therefore e_a is isomorphic to e_b. □

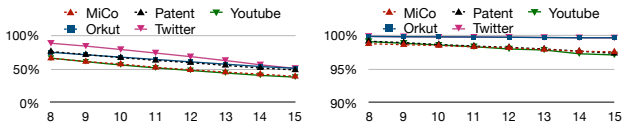
C. Deal with Tree-shape-embeddings

In the procedure of exploring natural graphs, we have observed an interesting fact that *the tree-shape-embeddings account for a large proportion of all explored embeddings*. On the other hand, *the time complexity of state-of-the-art solution of tree isomorphism is $O(n)$, in which n is the number of vertices in a tree-shape-graph*. The solution firstly finds the root of a tree-shape-graph which stands at $O(n)$, then canonizes the rooted tree which stands at $O(\log n)$ [8]. Therefore, solving tree isomorphism is theoretically easier than graph isomorphism. Thus Kaleido specially checks tree isomorphism in exploring embeddings.

In Section I, we discuss the exponential growth of the embeddings. It is very difficult to statistic the precise distribution of the shape of relative large embeddings. To simulate the distribution of the shape of embeddings, we design an experiment to sample embeddings over 4 natural graphs.

We use k steps to generate a k-embedding randomly. Initially, choose a random vertex as a 1-embedding. In the i^{th} step, choose a random neighbor of the $(i-1)$-embedding and unite them as an i-embedding. Once the k^{th} step is finished, a random k-embedding is generated. Then statistic the proportion of tree-shape-embeddings in the whole generated embeddings.

Figure 8 illustrates the proportion of tree-shape-embeddings from the 8^{th} to the 15^{th} iterations of the vertex-induced exploration and the edge-induced exploration over 5 natural graphs (see Table I). In the vertex-induced exploration over Twitter,



(a) vertex-induced (b) edge-induced

Fig. 8. The proportion of tree-shape-embeddings in the exploration. The x-axis indicates the iteration of exploration. The y-axis indicates the proportion of tree-shape-embeddings in all embeddings.

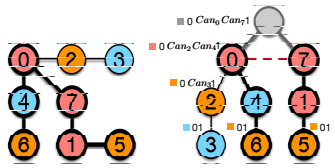


Fig. 9. Encode a tree-shape-embedding. Colors indicate labels of each node. The bold lines and vertices indicate the longest path in the embedding. The right figure indicates the rooted tree. The gray node denotes the fake point.

the tree-shape-embeddings account for 88.9% to 51.1% (the largest proportion over 5 natural graphs). The result over MiCo shows that the proportions are 66.5% to 39.1% (the lowest proportion over 5 natural graphs). Though the result reveals a downtrend of the proportion in the vertex-induced expression, the tree-shape-embeddings still account for a considerable proportion. In the edge-induced explorations, the tree-shape-embeddings account for 99.9% to 99.7% over Twitter and 98.8% to 97.6% over MiCo. As discussed in Section I, the complexity of k-vertex-embeddings is $O(N \cdot \bar{d}^{k-1})$. The k^{th} iteration of the edge-induced exploration generated k-edge-embeddings in which only the tree-shape-embeddings are k-vertex-embeddings. Therefore the tree-shape-embeddings account for the vast majority in the edge-induced exploration.

The tree isomorphism problem is well studied. Kaleido adopts the AHU algorithm [3] to solve the tree isomorphism problem and canonizes a tree-shape-embedding in next steps. Original AHU algorithm solve the unlabeled tree isomorphism problem. To deal with labeled trees, we extend the AHU algorithm (see Figure 9). First, find the longest path in a tree-shape-embedding; if the length of this path is odd, mark the central point as the root, otherwise, break the edge linked the two central points and link this points to a fake point, then mark this fake point as the root (the gray node in Figure 9). Then recursively mark neighbors of the root as the root of the corresponding subtree. A rooted tree of this tree-shape-embedding is built. We note a rooted tree as T, the immediate subtrees as $S_1, ..., S_c$, in which c is the number of the children of the root. The canonical form Can_T is defined as $L_r 0 Can_{S_{i_1}} ... Can_{S_{i_c}} 1$, in which L_r indicates the label of the root, the lexicographical order of $Can(S_1), ..., Can(S_c)$ determines the order of $S_{i_1}, ..., S_{i_c}$. Kaleido compares the canonical form of the rooted tree to check tree isomorphism. For edge labeled trees, assign the edge label to the vertex which is near the leaf in the rooted tree. If there exist two central points, assign both of them the label of the broken edge. Then remove the label on edges and an edge labeled tree transforms to a vertex labeled tree.

TABLE I
DATASET USED IN EVALUATION

Dataset	Vertices	Edges	Labels	Avg. Degree
CiteSeer	3,312	4,536	6	3
MiCo	100,000	1,080,298	29	22
Patent	3,774,768	16,518,948	37	9
Youtube	22,763,734	195,996,204	29	17
Orkut	3,072,441	117,185,083	-	76
Twitter	41,581,361	1,468,365,183	-	73

VI. EVALUATION

In this section, we evaluate Kaleido. First, we compare Kaleido with the state-of-art graph mining systems. Then we compare the graph isomorphic checking algorithm in Kaleido with Bliss. Next, we test the scalability of Kaleido in different applications. Finally, we test the I/O performance in the hybrid storage. Our experiments are performed on a single machine with Intel(R) Xeon(R) Gold 5117 CPU (2 nodes; 56 hyper-threads), 128GB memory, and 1 SSD with 480GB disk space (read bandwidth: 360MB/s, write bandwidth: 480MB/s). The operating system is CentOS 7.

A. Experimental Setup

Datasets: We use 6 datasets as showed in Table I. CiteSeer [12] has publications as vertices, with their Computer Science area as a label, and citations as edges. MiCo [12] models the Microsoft co-authorship and consists of an undirected graph whose nodes represent authors and are labeled with the author's field. Patents [22] includes all citations made by US Patents granted between 1975 and 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 the category of each video. Orkut [25] is a relatively dense user friendship network. Twitter [21] is a relatively large unlabeled graph which represents the Twitter social network.

Applications: We test 4 mining applications, FSM, Motif Counting, Clique Discovery and Triangle Counting. For k-FSM, we mine the frequent subgraphs which $k-1$ edges and at most k vertices. In our experiments, we run 3-, 4-, 5-FSM over several datasets. Motif Counting executions are run with subgraphs whose number of vertices is 3, 4 or 5. Clique Discovery executions are run with subgraphs whose number of vertices is 3, 4 or 5. Triangle Counting counts the number of triangles in the input graph.

B. Comparisons with Mining Systems

We ran all applications on Kaleido and compared it with three state-of-the-art systems, Arabesque [36], RStream [37] and ScaleMine [1]. Note that ScaleMine was designed specially to mine frequent subgraphs, and hence we could only obtain FSM's performance for it. Other mining systems are either not publicly available, such as NScale [29], or do not support FSM and motif counting, such as G-Miner [9]. We ran all these testing cases in a single node server with full usage of 56 threads. As we discussed the limitations of distributed systems in Section I, we focus on the performance of Arabesque and ScaleMine on a single node. We deployed the Hadoop

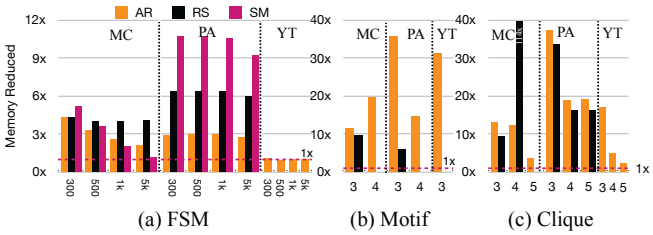


Fig. 10. Comparisons of Memory Consumption with Arabesque, RStream and ScaleMine. These figures indicate the memory reduction factor of the mining algorithms in Table II. Each x-axis indicates the argument of each algorithm. Failed executions are omitted.

2.7.7 in the experimental environment and put datasets on the local hdfs system, then Arabesque reads input graphs from the hdfs system. In testing RStream, the partition number of each algorithm was set to 10, 20, 50, 100 respectively, then chose the fastest result. In testing ScaleMine, the number of worker nodes was set to 1, 2, 4, 8, 16 respectively, then chose the fastest result. Table II reports the running time of the three systems and Figure 10 reports the memory consumptions.

Note that in this set of experiments, Kaleido, Arabesque and ScaleMine run all applications in memory, while RStream writes its intermediate data to disk. Kaleido outperforms both Arabesque, RStream and Scale in all cases. Excluding the small graph CiteSeer, Kaleido outperforms Arabesque by an overall (GeoMean) of **13.2**×, outperforms RStream by an overall of **64.8**× and outperforms ScaleMine by an overall of **36.6**×; the memory consumption of Kaleido is reduced by **7.2**× over Arabesque, **9.9**× over RStream and **2.6**×. Note that RStream uses `std::set` to maintain the graph topological structure, therefore it fails in loading Youtube in our 128 GB memory environment. The triangle counting in RStream uses another data structure of the graph and counting strategy, and it runs normally with the GRAS model. ScaleMine runs out of memory over Youtube as well.

FSM: As discussed earlier, we ran FSM by exploring embeddings in edge-induced strategy and we used the MNI support metric. We explicitly state the support used in each experiment, since this parameter is sensitive to the input graph. Theoretically, the smaller support is, the more computation is needed. However, the calculation of MNI support for each pattern needs much more computation resources. In the implementation of the FSM in Kaleido, we do not statistic the accurate MNI support of each pattern. Instead, when the MNI support of any pattern reaches the threshold given by the user, we mark this pattern a frequent pattern and prune it from the candidate. Therefore the run-time of the FSM computation in Kaleido does not decrease monotonically as the support increases. It will increases to peak time, due to meeting the pruning threshold is getting harder, then decreases normally because the frequent vertices and edges are more and less.

As discussed in Section IV-A, comparing with the intermediate data structure of Arabesque ODAG, the structure of embeddings CSE in Kaleido saves time from the extra canonical checking when travel the embeddings, but it trades some space of the intermediate data to obtain more efficient performance

TABLE II

COMPARISONS OF RUNNING TIME BETWEEN KALEIDO (KA), ARABESQUE (AR) AND RSTREAM (RS) ON FOUR MINING ALGORITHMS, 3-FREQUENT SUBGRAPH MINING (FSM, OPTION: SUPPORT), MOTIF COUNTING (MOTIF, OPTION: k), CLIQUE DISCOVERY (CLIQUE, OPTION: k) AND TRIANGLE COUNTING (TC) OVER THE FORMER DATASETS, CITESEER (CS), MICO (MC), PATENTS (PA), YOUTUBE (YT). EACH RESULT INDICATES THE RUNNING TIME OF THE APPLICATION IN SECOND. 'OOM' INDICATES THE EXECUTION RUNS OUT OF THE MEMORY. '/' INDICATES THE EXECUTION RUNS OUT OF THE SSD. '-' INDICATES THE SYSTEM DOES NOT SUPPORT THE APPLICATION.

		CS				MC				PA				YT			
Apps	Op	KA	AR	RS	SM	KA	AR	RS	SM	KA	AR	RS	SM	KA	AR	RS	SM
FSM	300	**0.04**	23.0	0.14	1.18	**5.6**	101.7	330.7	275.1	**15.3**	139.8	1228	2417	**127.4**	587.0	oom	oom
FSM	500	**0.04**	17.1	0.14	1.13	**6.0**	70.7	326.2	170.1	**16.5**	133.0	1220	2318	**130.3**	504.6	oom	oom
FSM	1000	**0.03**	17.0	0.14	0.16	**5.7**	46.6	316.6	48.3	**18.3**	119.4	1222	2116	**132.2**	498.1	oom	oom
FSM	5000	**0.02**	17.0	0.14	0.16	**2.7**	29.6	261.7	7.4	**20.2**	102.6	1179	779	**160.8**	496.1	oom	oom
Motif	3	**0.03**	23.4	0.11	-	**0.9**	28.3	73.9	-	**4.2**	60.2	100.6	-	**38.9**	443.2	oom	-
Motif	4	**0.06**	26.1	0.42	-	**219.4**	284.7	/	-	**253.2**	634.5	/	-	**6576**	oom	oom	-
Clique	3	**0.02**	23.0	0.02	-	**0.07**	27.9	4.8	-	**0.56**	60.5	95.3	-	**3.79**	342.3	oom	-
Clique	4	**0.03**	27.0	0.03	-	**0.82**	37.7	167.2	-	**1.14**	79.6	196.3	-	**9.39**	573.4	oom	-
Clique	5	**0.04**	29.9	0.04	-	**34.2**	299.0	oom	-	**1.46**	84.8	212.9	-	**17.5**	697.3	oom	-
TC		**0.02**	23.2	0.05	-	**0.07**	25.1	2.7	-	**0.52**	70.1	5.41	-	**2.24**	287.1	39.7	-

since the space complexity of ODAG is $O(|V|^2)$. Even so, Kaleido saves considerable space comparing Arabesque over MiCo and Patent, because Arabesque needs a huge amount of memory to establish its based system, Giraph [5], and graph data structure and the isomorphism checking library Bliss also consumes considerable space.

We found that in the relational phase of RStream [37], the shuffling operation and the aggregating operation produce many memory allocations and deallocations. The shuffling operation turns each tuple into a quick pattern, which allocates and deallocates memory frequently. The aggregating operation builds a hashmap to statistic the support of each pattern in using Bliss. We will discuss the comparison with Bliss in Section VI-C.

ScaleMine gets benefits of its approximate phase and pruning strategy when the number of frequent patterns decreases. However, when frequency support cannot filter most of the explored embeddings, ScaleMine needs more time to figure out frequent patterns.

Motif Counting: 4-Motif in RStream needs 6 iterations to explore all of 4-embeddings and writes too much intermediate data to disk, so that our 480 GB SSD cannot afford it. Thus we tested 4-Motif in RStream over MiCo and Patent on another server, which has an Intel(R) Xeon(R) E5-2640 v4 CPU with a total of 40 threads, 128 GB RAM and 4 TB Seagate ST4000NM0024-1HT HDD disk (read bandwidth: 130MB/s, write bandwidth: 130MB/s). The test was fully used 40 threads and the result shows that RStream produces 1.64 TB and 549.15 GB intermediate data over MiCo and Patent respectively and finishes in 114,917s and 19,740s.

Clique Discovery: To discover k-cliques, RStream uses a tricky solution with only k iterations of the edge-induced exploration. However, it still performs than Arabesque except 3-clique over MiCo and produces many intermediate data. For example, it produces 51.2 GB intermediate data in 4-clique over MiCo.

Triangle Counting: To counting triangles in a given graph, Kaleido treats each edge as a 2-embedding and counts the common adjacency vertices in this embedding. Both RStream and Arabesque scale poorly to larger graphs. The memory consumption of Kaleido is reduced by 22.3× over Arabesque and 7.3× over RStream.

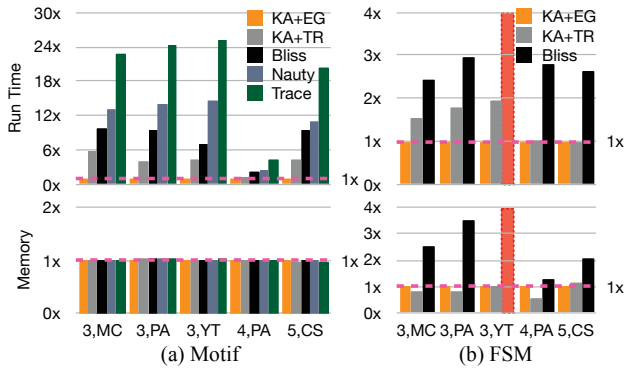


Fig. 11. Comparisons of isomorphism checking algorithms with Bliss, Nauty and Trace. The upper figures compare the run-time; the lower figures compare the memory consumption. Each x-axis indicates the application argument and dataset. The supports of 3-, 4-, 5-FSM are 300, 100k, 100 over corresponding graphs. 3-FSM in Kaleido with Bliss over YT ran out of memory.

C. Comparisons with Isomorphism Checking Algorithms

In this section, we compare our isomorphism checking algorithm with the state-of-the-art softwares, Bliss [19], Nauty [24] and Trace [27]. To test these softwares, we replace the isomorphism checking algorithm in Kaleido with them. To fully evaluate the performance of isomorphism checking algorithms in Kaleido, we compared 3-FSM, 4-, 5-FSM and Motif Counting respectively over different datasets. Note that both Nauty and Trace only suit for unlabeled graphs, therefore we only tested these softwares in Motif Counting. Figure 11 reports result of the comparisons.

Eigenvalue-based. For motif counting, the speedup of Bliss, Nauty and Trace is 6.8×, 9.4× and 16.6× respectively but the memory consumption is similar. For FSM, the speedup of Bliss is 2.7× and the memory consumption reduces by 3.1×. The reason is that the pattern considered by motif counting only contains the structural information of subgraphs, while it contains the label information in FSM. Kaleido builds the weighted adjacency matrix for each pattern, while others build search trees. In FSM, Bliss needs larger hash space and

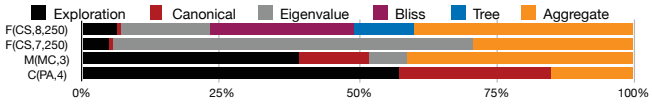


Fig. 12. CPU utilization breakdown during FSM (F), motif counting (M) and clique discovery (C) over CS, MC and PA. Options and dataset are stated in the brackets.

consumes more memory than motif counting. On the other hand, motif counting is a simple statistic of the occurrence of each motif, while FSM calculates the MNI support of each pattern and this calculation needs quite an amount of computation resources.

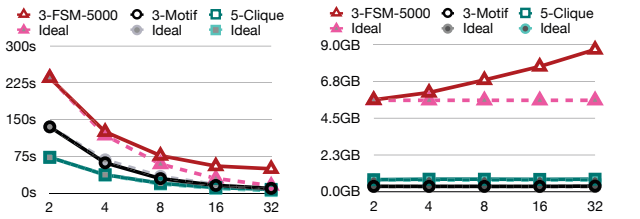
Tree isomorphism. Figure 11 also compares the tree isomorphism strategy with the softwares. It illustrates that the running time and memory usage is negative relative to the proportion of tree-shape-embeddings. To fully test the performance of tree isomorphism in large embeddings, we tested 8-FSM with support 250 over CiteSeer. The tree-shape-embeddings account for 98.9%. Kaleido runs this task in 890.3s and consumes 26GB memory, while it fails in using Bliss for the out-of-memory. Figure 12 shows the CPU utilization breakdown of this execution. Even though the non-tree-shape-embeddings account for 1.1%, running Bliss still takes a significant fraction of the CPU utilization.

D. Execution time breakdown

The CPU utilization breakdown of Figure 12 shows that the embedding exploration phase (exploration and canonical checking) and the pattern aggregation phase (computing support, isomorphism checking and aggregation) occupies a predominant fraction of CPU utilization. Both tests show that aggregation takes a significant fraction of CPU utilization. Kaleido uses `std::unordered_map` to statistic patterns or cliques, therefore the aggregation would be faster if the more efficient hashmap are used. Besides, isomorphism checking takes a significant fraction of CPU utilization in testing FSM, while the embedding exploration phase occupies a predominant fraction in testing motif counting and clique discovery.

E. Scalability

To evaluate the scalability of Kaleido, we tested 3-FSM with 5000 support, 3-Motif and 5-Clique over Patent in varying numbers of threads. Figure 13 illustrates Kaleido's run-time and memory consumption for this experiment. It illustrates that motif counting and clique discovery scale ideally both in the run-time and the memory consumption. While FSM only performs sublinearly in the run-time and the memory consumption increases as the number of threads grows. The implementation of FSM causes this phenomenon. To avoid using concurrent hashmap in the statistic of frequent patterns, we calculate the support of each pattern in every thread independently. It avoids the synchronization over each thread, but it consumes more memory in the pattern computation phase. The overhead of merging aggregating hashmap for FSM in multi-thread is inevitable in our implementation. If we could replace it by an efficient concurrent hashmap, the scalability of FSM in Kaleido would near linear scaling.



(a) Run-Time (b) Memory Consumption

Fig. 13. Scalability of Kaleido in 2, 4, 8, 16, 32 threads. Figure (a) shows the run-time; Figure (b) shows the memory consumption. The dotted lines indicate the ideal run-time and memory consumption respectively.

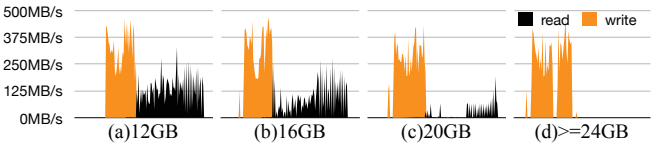


Fig. 14. I/O of 4-FSM over Patent with support 100k. These four figures show the I/O in limiting the memory cache of Kaleido with cgroup. The x-axis indicates the run-time of FSM; the y-axis indicates the reading and writing speed.

F. I/O and Load-balance in Hybrid Storage

To evaluate the performance of hybrid storage of the intermediate data, we ran 4-FSM over Patent with 50k and 100k supports and 4-Motif over Patent and MiCo in memory and on the hybrid storage respectively. In the hybrid storage testing, we stored the last layer of CSE on SSDs. Table III reports the result of these applications. It illustrates that the performance attenuation of using hybrid storage in Kaleido is acceptable (lower than 20% in these applications). For 4-FSM over Patent, the memory consumption reduced by the size of the last layer in CSE. For 4-Motif over MiCo, the memory consumption increases, because we built a buffer in fixed size for each thread (in these applications, 16 MB) and the total size of buffers is larger than the last layer of embeddings. Note that k-Motif only stores $k-1$ layers embeddings in Kaleido.

TABLE III
PERFORMANCE OF KALEIDO ON THE HYBRID STORAGE IN 4-FSM (F) OVER PATENT WITH 50K AND 100K SUPPORTS AND 4-MOTIF (M) OVER PATENT AND MICO.

Applications	F(PA,50K)		F(PA,100K)		M(PA)		M(MC)	
In-Memory	Yes	No	Yes	No	Yes	No	Yes	No
Times(s)	312	363	126	136	253	260	219	229
Memory(GB)	76.7	14.7	32.8	11.4	2.5	1.9	0.6	1.4

For 4-FSM over Patent with 100k support, the memory consumption is 11.4 GB and the size of the intermediate data is less than our experimental server (128 GB). To fully evaluate the design of embedding hybrid storage in Kaleido, we used cgroup in Linux to limit the maximum RAM of Kaleido in our experimental environment.

Figure 14 illustrates the I/O of this application in different limitations of max RAM. When the limitation of maximum RAM is larger than 24 GB, the intermediate data will be fully cached in memory. Figure 15 illustrates the run-time of different limitations of max RAM. When the limitation of maximum RAM is lower than 20 GB, the application reads

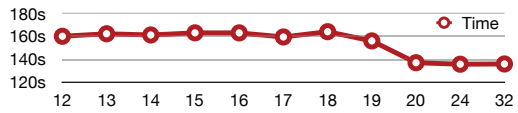
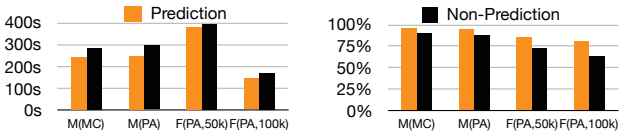


Fig. 15. Run-time of 4-FSM over Patent with 100k support in the different limitation of maximum RAM. The x-axis indicates the limitation of max RAM; the y-axis indicates the run-time.

the intermediate data from the disk and the run-time increases within 20%.

Load-balance. We evaluated the load-balance in hybrid storage by verifying the effectiveness of the prediction of the candidate size. We ran 4-FSM with support 50 K and 100 K over Patent and 4-Motif over Patent and MiCo. The result is illustrated in Figure 16.



(a) Run-Time (b) CPU Utilization Rate

Fig. 16. The comparison of prediction and non-prediction in hybrid storage. The first two columns in Figure (a) and (b) compare the 4-Motif over MiCo and Patent; the last two columns compare the 4-FSM over Patent with supports 50 K and 100 K.

G. Larger Graphs with Kaleido

We complete our evaluation by testing Kaleido over large graphs. We use Orkut and Twitter graphs for this evaluation, both of which are relatively dense graphs with an average degree of 76 and 73. 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 Motif Counting and Clique Discovery.

Table IV reports the running time, the maximum memory used and the number of interesting embeddings that Kaleido processed. For each application, the result reveals that the running time is positive corresponding to the number of embeddings. Overall, the result shows that Kaleido can process large dense graphs with a single commodity server that we use.

TABLE IV
PERFORMANCE OF KALEIDO ON RELATIVE DENSER AND LARGER GRAPHS.

Applications	Times	Memory	Embeddings
3-Motif(Orkut)	580.9s	8.9GB	44×10^9
5-Clique(Orkut)	254.9s	47.4GB	15×10^9
3-Motif(Twitter)	9h 724s	109.5GB	128×10^{12}
3-Clique(Twitter)	1097s	109.5GB	93×10^9

VII. 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.

Graph Mining Algorithms gSpan [38] is an efficient frequent subgraph mining algorithm designed for mining multiple input graphs. However, gSpan is designed for multiple graphs of mining problems. If we have a single input graph, we have to find multiple instances in the same graph, therefore it complexes the problem. Michihiro *et al.* [20] first proposed algorithms to mine patterns from a single graph. They use an expensive anti-monotonic definition of support based on the maximal independent set to find edge-disjoint embeddings. GraMi [12] proposes an effective method in the single large graph and presents an extended version with supporting structural constraints and an approximate version. Pržulj *et al.* [28] introduces the motif counting problem. Ribeiro *et al.* [30] presents G-Tries which is an effective approach for storing and finding the frequency of motifs. Aparício *et al.* [4] designs and implements a parallel version of G-Tries. Maximal clique is a well-studied problem.

Graph Mining Systems Arabesque [36] is a distributed graph mining system which supports popular mining algorithms. Arabesque proposed a graph exploration model with the concept of embeddings. Arabesque explores all the embeddings under constraining of user-defined filters and the developer processes each embedding with a filter-process programming model. Compared with Kaleido, Arabesque needs another canonically checking of each embedding in traveling embeddings. ScaleMine [1] is a parallel frequent subgraph mining system, which computes the approximate solution of frequent patterns firstly and statistics the exact solution by using the results of the first step to prune the search space. NScale [29] is designed to solve graph mining problems using MapReduce framework. It proposes a neighborhood-centric model, in which a k-hop neighborhood subgraph of an interest-point is constructed with k rounds of MapReduce and each round of Map-Reduce extends the 1-hop new neighbors. However, the overhead of MapReduce in processing candidate subgraphs is very high. G-Miner [9] is a distributed graph mining system, which models subgraph processing as independent tasks and designs suitable scheduling for the task pipeline. However, G-Miner does not support FSM and motif counting. ASAP [18] is a distributed, sampling-based approximate computation engine for graph pattern mining. ASAP leverages graph approximation theory and extends it to general patterns in a distributed setting. It allows users to trade-off accuracy for result latency. However, ASAP only counts the interest of the user with an acceptable error, like motif counting and pattern matching, but cannot return the exact result of frequent patterns. RStream [37] is the first single-machine, out-of-core graph mining system. RStream employs a GRAS programming model which combines GAS model and relational algebra to support a wide variety of mining algorithms. However, the join of subgraphs and edges of the input graph in RStream is still an expensive operation. The edge-induced exploration of subgraphs also complexes some mining problems, like motif counting and clique discovery.

Graph Isomorphism Checking Libraries The most common practical approach for the graph isomorphism problems is canonical labeling, a process in which a graph is relabeled in such a way that isomorphic graphs are identical after relabeling. The main strategy of the canonical labeling is building a search tree for the input graph. Nauty [24] is the first program that could handle large automorphism groups; it uses

automorphism to prune the search in testing automorphism. Nauty generates the search tree in depth-first order, while Trace [27] introduces a breadth-first search in generating the search tree. However, these libraries focus on the checking of the automorphism, which only suit for unlabeled graphs. Bliss [19] supports the isomorphism checking of labeled graphs. However, building the search tree brings frequently memory allocating and deallocating which slow down the processing and consume a huge amount of memory. Besides, Bliss is designed for the large graph isomorphic checking, while the eigenvalue checking strategy is sufficient in the mining scenes.

VIII. Conclusion

In this paper, we present Kaleido, a single-machine, out-of-core graph mining system. Kaleido follows the subgraph-centric model and provides a user-friendly simple API that allows non-experts to build graph mining workloads easily. To efficiently store and process the huge amount of intermediate data, Kaleido builds a succinct intermediate data structure and adjusts the storage in memory or out-of-core smoothly according to the scale of intermediate data. Kaleido designs a lightweight isomorphism checking strategy to solve the labeled graph isomorphism problem. Experimental results demonstrate that Kaleido is more efficient than the state-of-the-art graph mining systems in most cases. The isomorphism checking algorithm in Kaleido is more efficient and consumes less memory than state-of-the-art graph libraries.

Acknowledgment

We would like to thank the reviewers for the insightful comments. This work is supported by the Strategic Priority Research Program of CAS (XDA19020400), NSF of China (61425016, 61772498, 91746301), and the Beijing NSF (4172059).

References

[1] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *SC*, pages 61–72. IEEE Press, 2016.

[2] C. C. Aggarwal, H. Wang, et al. *Managing and mining graph data*, volume 40. Springer, 2010.

[3] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[4] D. O. Aparício, P. M. P. Ribeiro, and F. M. A. da Silva. Parallel subgraph counting for multicore architectures. In *IPDPS*, pages 34–41. IEEE, 2014.

[5] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[6] L. Babai, W. M. Kantor, and E. M. Luks. Computational complexity and the classification of finite simple groups. In *SFCS*, pages 162–171. IEEE, 1983.

[7] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *PAKDD*, pages 858–863. Springer, 2008.

[8] S. R. Buss. Alogtime algorithms for tree isomorphism, comparison, and canonization. In *Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 18–33. Springer, 1997.

[9] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *EuroSys*, page 32. ACM, 2018.

[10] X. Cheng, C. Dale, and J. Liu. Statistics and social network of youtube videos. In *IWQoS*, pages 229–238. IEEE, 2008.

[11] W. Eberle, J. Graves, and L. Holder. Insider threat detection using a graph-based approach. *Journal of Applied Security Research*, 6(1):32–81, 2010.

[12] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.

[13] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, volume 29, pages 251–262. ACM, 1999.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX, 2012.

[15] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613. USENIX, 2014.

[16] F. Harary, C. King, A. Mowshowitz, and R. C. Read. Cospectral graphs and digraphs. *Bulletin of the London Mathematical Society*, 3(3):321–328, 1971.

[17] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl_1):i213–i221, 2005.

[18] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. Asap: Fast, approximate graph pattern mining at scale. In *OSDI*, pages 745–761, 2018.

[19] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *ALENEX*, pages 135–149. SIAM, 2007.

[20] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *SDM*, pages 345–356. SIAM, 2004.

[21] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[22] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187. ACM, 2005.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[24] B. D. McKay. Computing automorphisms and canonical labellings of graphs. In *Combinatorial mathematics*, pages 223–232. Springer, 1978.

[25] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, 2007.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[27] A. Piperno. Search space contraction in canonical labeling of graphs. *arXiv preprint arXiv:0804.4881*, 2008.

[28] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[29] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *VLDBJ*, 25(2):125–150, 2016.

[30] P. Ribeiro and F. Silva. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28(2):337–377, 2014.

[31] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424. ACM, 2015.

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488. ACM, 2013.

[33] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[34] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[35] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, volume 48, pages 135–146. ACM, 2013.

[36] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440. ACM, 2015.

[37] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI*, pages 763–782. USENIX, 2018.

[38] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724. IEEE, 2002.

[39] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316. USENIX, 2016.