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

Cheng Zhao
ICT, CAS
zhaocheng@ict.ac.cn

Zhibin Zhang
ICT, CAS
zhangzhibin@ict.ac.cn

Peng Xu
ICT, CAS
xupeng@ict.ac.cn

Tianqi Zheng
ICT, CAS
zhengtianqi@ict.ac.cn

Xueqi Cheng
ICT, CAS
cxq@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 utilizes the eigenvalue of the adjacency matrix of a subgraph to efficiently solve the subgraph isomorphism problems with an acceptable constraint that the vertex number of a subgraph is less than 9. Kaleido implements half-memory-half-disk storage for storing large intermediate data, which treats the disk as an extension of the memory. Comparing with two state-of-the-art mining systems, Arabesque and RStream, Kaleido outperforms them by a GeoMean **12.3**× and **40.0**× respectively.

1. 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, we calculate the PageRank [23] value of a web graph to obtain the top-k valuable web pages; give two vertices in an input graph, we calculate the shortest path between them. While graph mining aims to discover structural patterns to meet the user's interest criteria. For example, we mine frequent subgraphs in the biological data to discover highest gene expression [16]; we extract the frequency distribution of all motifs that occur in PPI network [25]; we discover cliques in financial networks to detect frauds[10].

1.1 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 [13], Ligra [32], GraphX [14], Chaos [28], Gemini [36], etc., follow a simple "think like a vertex (TLV)" programming paradigm pioneered by Pregel [21], which is a perfect match for linear algebra problems.

However, the TLV programming paradigm cannot readily formulate graph mining problems. Given an input graph, graph mining problems often require exploring a very large number of subgraphs and finding patterns that match some interesting criteria desired by the user. 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 an example of pattern matching, which is also a step of the frequent subgraph mining. 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 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 explo-

Figure 1: An example of pattern matching. 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. Same embedding b and c also are called automorphic.

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

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

ration of 4-embeddings over Patent (3.8 M vertices, 16.5 M edges) [20] produces 13.5 billion embeddings. The second challenge is how to efficiently test embedding isomorphism in computing patterns. The graph isomorphism problem (GI) is an NP hard problem and no polynomial time algorithm is known [31]. The fastest proven running time for GI has stood at $e^{O(\sqrt{n \log n})}$ [5].

1.2 Limitations of State-of-the-Art Systems

Recent graph mining systems use declarative models to solve mining problems. Arabesque [33] proposes a natural programming paradigm, "think like an embedding (TLE)", which is also called *subgraph-centric* model. RStream[34] 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 ith array contains the ids of all vertices in the ith position in any embedding. Vertex v in the ith 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 run-time in mining applications.

RStream is a single-machine graph mining system based on X-Stream[29]. It only supports the 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) [11] 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[18], which is an open source tool for computing graph isomorphism problems. Bliss builds a search tree and calculates a hash value for each pattern. If two patterns contain the same hash value, they are automorphic. However, building the search tree brings frequently memory allocating and

deallocating which slow down the processing of hashing patterns and consumes a huge amount of memory. For example, the overhead of allocation and deallocation are more then 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.

1.3 Our Approaches

To address the limitations of existing systems, we propose Kaleido, a single-machine, out-of-core graph mining system. Inspired by Arabesque, Kaleido adopts the embedding-centric computation model and presents a general programming API which fits most of graph mining applications. 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 is the set of 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. Thus Kaleido treats the intermediate data of the ith exploration as an i-dimension tensor. Inspired by compressed sparse column (CSC) for sparse matrices [30], 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.

Calculating large embeddings of graph mining problems is hard to implement because of the exponential growth of the intermediate data in graph mining problems. For example, the sizes of embeddings in 7 iterations of the embedding explorations over a small graph CiteSeer (3.3 K vertices, 4.7 K edges) are 3.3 K, 4.5 K, 24.5 K, 352.2 K, 7.7 M, 168.2 M, 3.5 G respectively. Kaleido focuses on efficiently solving the GI problems in small embeddings (less than 9 vertices). From the property of isomorphism, we explain that isomorphic embeddings have the same eigenvalues. Harary *et at.* [15] proved that the smallest non-isomorphic unlabeled graphs contain 6 vertices with the same eigenvalues and the smallest non-isomorphic unlabeled graphs with the same eigenvalues and the same vertex degrees contain 9 vertices. To solve the labeled graph isomorphism problem, Kaleido combines the label information and the degree information of vertices in each pattern and eigenvalues of the adjacency matrix to check isomorphism over embeddings.

To store more intermediate data when the scale of the input graph or the exploration depth increases, we design a hybrid storage for the intermediate embedding. 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 work load 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. When processing the intermediate data on disk, Kaleido adopts a slide-window strategy to guarantee the performance of parallel computation and hide the overhead of I/O.

To summarize, we make the following contributions:

- We introduce the computation model of Kaleido in Section 3. We propose a novel succinct embeddings data structure, compressed sparse embedding, which treats embeddings as a sparse tensor.

- We propose a lightweight graph isomorphism checking algorithm in using eigenvalues of the adjacency matrix of each pattern (Section 3).

- We implement a hybrid storage for the large intermediate data, which treats disks as an extension of the memory and hides the overhead of I/O by the computation of the embedding exploration and pattern aggregation (Section 4).

- We design an API for popular graph mining applications, which enables embedding exploration and pattern aggregation to be expressed effectively. We present four popular graph mining applications which are expressed in Kaleido API (Section 5).

- We compare Kaleido with Arabesque and RStream in four graph mining applications. We compare the performance of GI problems in Kaleido with bliss. We demonstrate the scalability of Kaleido in different applications. We demonstrate the I/O performance in the hybrid storage (Section 6).

The rest of paper are organized as follows. Section 2 formalizes the problem. Section 3 presents the computation model of Kaleido, the design of CSE and the eigenvalue-based isomorphism checking algorithm. Section 4 presents the storage strategy of the large intermediate data. Section 5 introduce the API of Kaleido and the implementations of popular graph mining applications. Section 6 presents the experimental evaluation. Section 7 surveys related works and Section 8 concludes.

2. 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 subgraph $G_a = (V_a, E_a, L_a)$ of graph G is *isomorphic* to another subgraph $G_b = (V_b, E_b, L_b)$ of G 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 $L_a(u, v) = L_b(f_{ab}(u), f_{ab}(v))$, $\forall (u, v) \in E_a$.

In Figure 1, both subgraphs $a = (V_a, E_a, L_a)$ and $b = (V_b, E_b, L_b)$ have pattern p. There exists a bijection f between subgraphs a and b, $f_{a \leftrightarrow b} : \{1 \leftrightarrow 3, 2 \leftrightarrow 2, 5 \leftrightarrow 5\}$, which satisfies constraining of subgraphs isomorphism. Thus subgraph a is isomorphic to subgraph b. Two subgraphs 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 $v_b < v_c$.

3. COMPUTATION MODEL

In this section, we describe the computation model of Kaleido and the design of data structure of intermediate data and patterns. The procedure of graph mining applications in Kaleido are mainly divided into two phases, the embedding exploration phase and the pattern aggregation phase.

3.1 Embedding Exploration

In the phase of embedding generation, given the size of embedding, our goal is to generate all of the possible and unique embeddings. Then 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.

We follow Arabesque's idea of checking embedding canonicality for each candidate. From 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 3 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 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.

Figure 3: 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; the dotted lines between two embeddings, like a dotted line between s_7 and s_8, which means s_6 and s_7 are generated by s_1 and s_8 and s_9 are generated by s_2; note that the dotted lines between vertices do not really exist in the embedding array, instead they only represent there exist an edge in input graph between these two vertices.

3.1.1 Embeddings Data Structure

The goal of storing embeddings is divided into two parts: (i) minimizing memory usage and (ii) obtaining 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.

Figure 4: The structure of compressed sparse embedding (CSE). This figure shows a 2-CSE and a 3-CSE of 2-embeddings and 3-embeddings illustrated in Figure 3 respectively. 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.

As illustrated in Figure 4, 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 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, $[off_l(i), off_l(i+1))$, indicates that these vertices possesses same embedding prefix. For example, in Figure 4a, 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 3, which possess the same embedding prefix $\{1\}$. Therefore each vertex in vertex array corresponds to a unique embedding in 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, 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 4b, 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 3.

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 $O(k \log \bar{d}) = O(\log(|E|/|V|))$.

3.2 Pattern Aggregation

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 int the pattern aggreation phase, Kaleido calculates the pattern of each embedding and aggregates them. The challenge of the aggregation is how to efficiently test embedding isomorphism and obtain the pattern. As it is well studied, GI is known as an NP-hard problem [31]. The state-of-art algorithm is to solve the problem has run-time $e^{O(\sqrt{n \log n})}$ for graphs with n vertices [5]. Existing algorithms or libraries build search tree for each pattern to solve the GI problem, like Bliss.

In Kaleido, patterns are stored in a simple compact data structure, as illustrated in Figure 5. 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 (gray area in Figure 5b) in form of 1-dimension array and stores it as a bitmap. Obviously, storing an k-pattern in this data structure needs a label array whose size is k, and a bitmap whose size is $\frac{1}{2}(k(k-1))$.

In pattern aggregation phase, Kaleido directly transforms each embedding to this pattern data structure. Then Kaleido should identify non-isomorphic patterns and automorphic patterns. Note that the relationship between the pattern and the data structure is 1 to n. One data structure represents a unique pattern but one pattern can be represented in different data structures. These different data structures represent automorphic patterns. In Kaleido, the GI problem is solved by utilizing the relationship between the adjacency matrix of the pattern and its eigenvalues.

THEOREM 1. *Let G_a and G_b be two k-subgraphs of an undirected graph G, A and B be adjacency matrices of G_a and G_b respectively, $\Lambda_a = \{\lambda_{a1}, ..., \lambda_{an}\}$ and $\Lambda_b = \{\lambda_{b1}, ..., \lambda_{bn}\}$ be eigenvalues of A and B respectively. If G_a is isomorphic to G_b, it holds that $\Lambda_a = \Lambda_b$.*

(a) Input Graph (b) Adj. Matrix (c) Pattern Structure

Figure 5: Pattern structure. Figure 5a indicates an undirected graph. Figure 5b is adjacency matrix of the graph. Figure 5c consists 2 parts of pattern structure. Colors in label list indicate different labels in the graph. Adjacency matrix is stored in form of bitmap.

$$\lambda^6 - 7\lambda^4 - 4\lambda^3 + 7\lambda^2 + 4\lambda - 1 \qquad \lambda^9 - 8\lambda^7 + 19\lambda^5 - 14\lambda^3 + 2\lambda$$

Figure 6: The smallest counterexamples and corresponding characteristic polynomials.

PROOF. Let G_{pa} and G_{pb} be patterns of G_a and G_b respectively. A and B are also adjacency matrices of G_{pa} and G_{pb} respectively. According to Definition 1, if G_a is isomorphic to G_b, G_{pa} is automorphic to G_{pb}. It leads that A can be transformed to B by a similarity transformation. Thus there exists an invertible n-by-n matrix P, such that

$$B = P^{-1}AP$$

Thus A and B are similar. Similar matrices have the same eigenvalues and their algebraic multiplicities are the same. □

According to Theorem 1, if eigenvalues of two patterns are different, it holds that these patterns are non-isomorphic. Unfortunately, not all graphs that have the same eigenvalues [1] are necessarily isomorphic. However, Harary *et at.* [15] proved that the smallest non-isomorphic graphs with the same eigenvalues contain 6 vertices and the smallest non-isomorphic graphs with the same eigenvalues and the same vertex degrees contain 9 vertices (see Figure 6). Thus we get a corollary.

COROLLARY 1. *If two k-embeddings (k < 6) have the same eigenvalues, they are isomorphic. Further, if two k-embeddings (k < 9) have the same vertex degrees and the same eigenvalues, they are isomorphic.*

When vertices of the input graph have different labels, the GI problem becomes a little complex. Algorithm 1 illustrates the solution of the GI problem in Kaleido where the size of embedding is less than 9. Kaleido maintains the vertex label array L in an ascending order (lines 30-31) and the degrees (D) of the same label vertices in an ascending order as well (lines 32-33). Note that Swap function also maintains the adjacency matrix A, so that the vertex order in A is consisting with L and D. Then Kaleido builds a

[1] In this paper, if graphs have the same eigenvalues, their algebraic multiplicities are the same as well.

Algorithm 1: Kaleido's graph isomorphic check

Input: Embedding $e = \langle v_1, ..., v_k \rangle$
Output: Hash value $hash\_value$ of Embedding e

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** $1 \leq t \leq k$ **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($L, \mathbf{A}$):
13    $\mathbf{M} \leftarrow \{m_{i,j} = 0|\forall i, j \in [1, k]\}$
14   **for** $1 \leq i < j \leq n$ **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($\mathbf{M}$):
20   Characteristic polynomial $P \leftarrow \{p_i = 0|\forall i \in [1, k]\}$
21    $\mathbf{C} \leftarrow \mathbf{M}$
22   **for** $1 \leq i \leq k$ **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** EigenHash(e):
28    $L, \mathbf{A}, D \leftarrow$ Init(e)
29   **for** $1 \leq i < j \leq n$ **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($L, \mathbf{A}$)
35    $P \leftarrow$ CharPloynomical($\mathbf{M}$)
36    $hash\_value \leftarrow hash(L) \oplus hash(D) \oplus hash(P)$
37   **return** $hash\_value$

---

weighted adjacency matrix 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 the eigenvalues of the matrix M. However, the calculation of the accurate eigenvalues is redundant, while Kaleido calculate the characteristic polynomial of the matrix M by the Faddeev-LeVerrier algorithm (line35, lines 19-26). Finally, Kaleido calculates the hash value of each embedding by combining the hash value of the label array L, the degree arrary D and the characteristic polynomial P in XOR ($\oplus$).

THEOREM 2. *Let e_1 and e_2 be two k-embeddings of an undirected graph G, $k < 9$. Let h_1 and h_2 be hash values of e_1 and e_2, which are calculated by Algorithm 1. Embedding e_1 is isomorphic to embedding e_2 if and only if $h_1 = h_2$.*

PROOF. For embeddings e_1 and e_2, hash values $h_1 = h_2$ is equivalent to that label arrays $L_1 = L_2$, degree arrays $D_1 = D_2$ and characteristic polynomials $P_1 = P_2$. From Definition 1, the isomorphism leads to $L_1 = L_2$ and $D_1 =$

Figure 7: 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).

Figure 8: 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 $\langle 3 \rangle$.

D_2. Theorem 1 satisfies $P_1 = P_2$. The necessity is proved. Note M_1, M_2 as the weighted adjacency matrix of e_1 and e_2 respectively. Note ϵ_1 as a graph which is no vertex label but the adjacency matrix is weighted and equals to M_1. Note ϵ_2 symmetrically. From Corollary 1, $D_1 = D_2$ and $P_1 = P_2$ guarantee that ϵ_1 and ϵ_2 are isomorphic. Thus, there exists a bijection $f(u) = v, \forall u \in V_{\epsilon_1}, \forall v \in V_{\epsilon_2}$ between V_{ϵ_1} and V_{ϵ_2}, such that $(f(u), f(v)) \in E_\epsilon$ and $L(u, v) = L(f(u), f(v)), \forall (u, v) \in E_{\epsilon_1}$. Then combining $L_1 = L_2$ and $D_1 = D_2$, it leads to that $L(u) = L(f(u)), \forall u \in V_{e_1}$. The sufficiency is proved. □

4. IMPLEMENTATION

In this section, we introduce the storage strategy of larger intermediate data in Kaleido. Then we introduce the load-balance strategy of Kaleido in facing the insufficient RAM.

4.1 Embedding 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 an 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 7). If memory is sufficient, Kaleido merges t parts of off_{k+1} in memory, 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 7. 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.

4.2 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 [12]. The degree distribution of embeddings is also skewed power-law distribution. When the RAM can afford the embeddings data, Kaleido utilizes 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 work load in the exploration of the $(k+1)^{th}$ level, Kaleido predicts the size of $vert_{k+1}$. Figure 8 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 of the merging is $O(\bar{d})$. According to the prediction, Kaleido partitions the exploration tasks evenly to each threads.

5. KALEIDO API

In this section, we demonstrate the API for graph mining problems in Kaleido. Finally, we introduce implementations of popular graph mining applications in Kaleido.

The API of Kaleido is illustrated in Listing 1. **Embedding-Filter** and **PatternFilter** are two optional filters in the embedding exploration phase and the pattern aggregation phase respectively, which will return **true** in default. For the vertex-induced embedding exploration, **EmbeddingFilter** works in exploring $(k+1)$-embeddings from k-embeddings: the **Embedding e** is a k-embedding and the **Vertex v** is a

candidate vertex which is a neighbor of `e` normally. It is analogous to edge-induced embedding exploration, in which the candidate of each embedding is an `Edge <u,v>`. Actually, we eliminate a default embedding filter, the canonical embedding filter. The `vertex v` could be appended to `Embedding e` if and only if they satisfy constraints of both user-defined filter and the default canonical filter. `Pattern Filter` works in the aggregation of patterns, in order to prune ineligible patterns. `AggregatingMapper` and `AggregatingReducer` (`Mapper` and `Reducer` in short) must be implemented by customized applications. `Mapper` is an operator for calculating the pattern of an embedding `e` and adding the pattern to `PatternMap`. `Mapper` is calculated in `ResultAggregator` concurrently. `Reducer` aggregates `PatternMap`s 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)
```

The processes of graph mining application in Kaleido are described as follows. Initially, in the vertex-induced applications, `Init` treats the vertex set of the input graph G as 1-embeddings or the edge set as 2-embeddings. The user should determine a terminated condition of `EmbeddingExplorer`. While in the edge-induced application, `Init` treats the whole edge set of Graph G as the set of 1-embeddings. Generally, the user gives the maximum number k of different vertices in each embedding. Then `EmbeddingExplorer` iterates $k-1$ times to explore all possible k-embeddings with restriction of `EmbeddingFilter`. If embeddings are larger than memory, Kaleido stores them to disks intermediately. Next `PatternComputer` calculates patterns of each k-embedding and aggregates patterns with restriction of `PatternFilter`. Finally, Kaleido returns results in the form of `PatternMap`.

5.1 Popular Mining Applications in Kaleido

Frequent Subgraph Mining. To early prune infrequent patterns, we use the *minimum image-based (MNI) support* [6] 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. Given an input graph, edges 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. Initially, `Init` calculates the MNI support for each edge (1-embedding) and eliminates infrequent edges according to the threshold. In each iteration of embedding ex-

ploration, `EmbeddingExplorer` expands each embedding by adding a frequent edge. `EmbeddingFilter(e,<u,v>)` checks if the candidate edge `<u,v>` is frequent. Then `Mapper` patternizes each embedding, calculates the MNI support for each pattern. Next, `Reducer` prunes infrequent patterns and its corresponding embeddings. `Pattern Filter` eliminates infrequent patterns. Finally, in the last iteration, `Reducer` statistics frequent patterns and returns the results.

Motif Counting. This application counts the frequency of each k-motif in the given graph. The `EmbeddingFilter` and `PatternFilter` can be set as default. 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. Then `Mapper` explores all canonical k-embeddings from each $(k-1)$-embedding, then calculates the hash value of each k-embedding. Finally `Reducer` aggregates k-motifs. For example, consider counting 3-motifs over graph in Figure 3. After the embedding exploration phase, `EmbeddingExplorer` returns 2-embeddings of the graph. In the pattern aggregation phase, considering embedding s_6 without loss of generality. The `Mapper` explores two 3-embeddings, s_{13} and s_{14}, for s_6. The `Mapper` obtains a 3-chain and a triangle. Finally `Reducer` aggregates results: 5 3-chains and 3 triangles.

Figure 9: Example of 3-Clique Discovery. The dotted lines between vertices do not really exist in the embedding array. Gray embeddings indicates that they are eliminated by clique checking filter.

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. Therefore generating cases like embedding $\langle 1, 2 \rangle$ and candidate 3 will be eliminated, cause vertex 3 is not neighbor of vertex 1. After `Init`, `EmbeddingExplorer` prunes illegal embeddings and explores all k-cliques after $k-1$ iterations and returns them. In this example, `Mapper` is unnecessary to calculate pattern of each embedding, cause all embeddings have a same pattern. Finally, `Reducer` return k-cliques.

Triangle Counting. This application counts the number of triangles in the input graph. Initially, `Init` generates 2-embeddings from the edge set of the input graph. Then `Mapper` counts the number of common neighbors of two vertices in each 2-embedding canonically. Finally, `Reducer` aggregates counting results. For embedding $s_6 = \langle 1, 2 \rangle$ illustrated in Figure 3, vertex 5 satisfies former restrictions. Analogously, cases like vertex 5 over s_8 and s_{10} satisfy the restrictions. The answer of triangle counting over graph in Figure 3 are 3.

6. EVALUATION

In this section, we evaluate Kaleido. First we compare Kaleido with the state-of-art graph mining systems, Arabesque

and RStream. 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 evaluated on a single machine with Intel(R) Xeon(R) Gold 5117 CPU, 128GB memory, and 1 SSD with 480GB disk space. The operating system is CentOS 7.

6.1 Experimental Setup

Table 1: 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	7,065,219	59,811,883	29	17

Datasets: We use 5 datasets as showed in Table 1. CiteSeer [11] has publications as vertices, with their Computer Science area as a label, and citations as edges. MiCo [11] models the Microsoft co-authorship and consists of an undirected graph whose nodes represent authors and are labeled with the author's field. Patents [20] 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 [8] 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.

Applications: We test 4 mining applications discussed in Section 5, 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.

6.2 Comparisons with Mining Systems

We compared Kaleido with two state-of-the-art systems, Arabesque [33] and RStream [34]. We ran all these testing cases in a single node server. In testing Arabesque, we deployed the Hadoop 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. We ran algorithms mentioned in Section 6.1 over labeled graphs, CiteSeer, MiCo, Patent and Youtube on Kaleido, Arabesque and RStream.

Table 2 reports the running time of the three systems and Figure 10 reports the memory consumptions. Note that in this set of experiments, Kaleido and Arabesque run all applications in memory, while RStream writes its intermediate data to disk. Kaleido outperforms both Arabesque and RStream in all cases. Excluding the small graph CiteSeer, Kaleido outperforms Arabesque by an overall (GeoMean) of **12.3**× and outperforms RStream by an overall of **40.0**×; the memory consumption of Kaleido is reduced by **7.2**× over Arabesque and **9.9**× over RStream.

Arabesque is a Giraph-based system and implemented by Java. Each iteration of Arabesque is mapped to a superstep of Giraph [4]. Arabesque needs extra time to boost the system and fit the basic Giraph API. Therefore, for the small graph, CiteSeer, Arabesque is not as efficient as the other two systems. Table 3 also indicates that Arabesque allocates a huge amount of memory for the lower layer system.

RStream is an X-Stream-based system. In the preprocessing phase, RStream partition the input graph into several parts according to an optional partition number given by the user. 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.

Figure 10: Comparisons of Memory Consumption with Arabesque and RStream. These figures indicate the memory reduction factor of the mining algorithms in Table 2. Each x-axis indicates the argument of each algorithm. The Gray bars indicate the algorithm runs out of the memory.

FSM: As discussed earlier, we ran FSM by exploring subgraphs in edge-induced strategy and we used the minimum image-based support metric [6], which defines the frequency of a pattern as the minimum number of distinct mappings for any vertex in the pattern, over all instances of the pattern. 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, as illustrated in Figure 11. 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.

Table 2: Comparisons of running time between Kaleido (KA), Arabesque (AR) and RStream (RS) on four mining algorithms, frequent subgraph mining (3-FSM, option: support), motif counting (k-Motif, option: k), clique discovery (k-Clique, option: k) and triangle counting (TC) over the former datasets, CiteSeer (CS), MiCo, Patents (PA), Youtube (Ytb). Each result indicate the running time of the application in second. '-' indicates the execution runs out of the memory. '/' indicates the execution runs out of the SSD.

Dataset		CS			MiCo			PA			Ytb		
Apps	Options	KA	AR	RS	KA	AR	RS	KA	AR	RS	KA	AR	RS
3-FSM	300	**0.04**	23.03	0.14	**7.35**	101.77	330.72	**25.47**	139.8	1228	**132.59**	426.67	-
	500	**0.04**	17.05	0.14	**8.19**	70.74	326.29	**26.41**	133	1220	**133.31**	409.23	-
	1000	**0.03**	17.01	0.14	**7.84**	46.65	316.68	**28.71**	119.4	1222	**136.24**	397.19	-
	5000	**0.02**	17.01	0.14	**3.97**	29.67	261.70	**31.51**	102.6	1179	**155.04**	396.11	-
Motif	3	**0.03**	23.42	0.11	**1.39**	28.37	73.92	**4.74**	79.71	100.61	**35.47**	246.24	-
	4	**0.06**	26.10	0.42	**198.17**	284.79	/	**152.28**	634.47	/	**4988.96**	-	-
Clique	3	**0.02**	23.01	0.02	**0.46**	27.92	4.75	**0.56**	60.55	95.34	**2.16**	195.63	-
	4	**0.03**	27.03	0.03	**3.88**	37.71	167.22	**1.14**	79.68	196.37	**7.83**	461.81	-
	5	**0.04**	29.99	0.04	**183.63**	299.01	-	**1.46**	84.81	212.95	**18.99**	505.9	-
TC		**0.02**	23.18	0.05	**0.17**	25.05	2.74	**0.52**	70.17	5.4	**2.24**	287.04	39.68

Table 3: Comparisons of memory consumption (MB) between Kaleido, Arabesque (AR) and RStream (RS) over CiteSeer. Each result indicate the memory consumption of the application in Mega-byte.

Apps	Options	Kaleido	AR	RS
3-FSM	300	205.8	1916	**97.1**
	500	194.2	1888	**97.1**
	1000	130.9	1864	**97.1**
	5000	**23.3**	1889	97.1
Motif	3	**26.9**	1802	196.7
	4	**166.8**	1812	482.5
Clique	3	**25.8**	1890	97.1
	4	**29.7**	1932	97.1
	5	**27.2**	1940	97.1
TC		**26.9**	1819	198.3

(a) Run-Time   (b) Memory Consumption

Figure 11: The run-time and the memory consumption trends over the increasing of support in 3-FSM. The x-axis indicates the support of 3-FSM.

As shown in Table 2 and Figure 10a, Kaleido outperforms Arabesque by a GeoMean of 8.5×, 4.4× and 2.9× over MiCo, Patent and Youtube respectively, while the memory consumption reduces a GeoMean of 3.1× and 2.9× over MiCo and Patent respectively and increases 1.2× over Youtube. As discussed in 3.1.1, 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 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 and graph data structure and the isomorphism checking library bliss also consumes considerable space.

Comparing with RStream, Kaleido outperforms it by a GeoMen of 46.7× over MiCo and 43.4× over Patent, while the memory consumption reduces a GeoMean of 4.2× over MiCo and 6.3× over Patent. We found that in the relational phase of RStream [34], 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 6.3.

Motif Counting: Motif counting assumes the input graph is unlabeled and explores all of the embeddings exhaustively until the subgraph reaches the maximum size. Table 2 and Figure 10b respectively report the comparison of the run-time and the memory consumption with Arabesque and RStream. Comparing with Arabesque, Kaleido outperforms by a GeoMean of 6.8× and the memory consumption reduces by a GeoMean of 20.7×. Comparing with RStream, Kaleido outperforms by a GeoMean of 33.6× and the memory consumption reduces by a GeoMean of 7.7×. 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. RStream produces 1.64 TB and 549.15 GB intermediate data over MiCo and Patent respectively and finishes in 114917s and 19740s.

Clique Discovery and Triangle Counting: Clique Discovery is to enumerate all complete subgraphs in the input graph. Triangle Counting is to count the number of triangles in the input graph. Table 2 and Figure 10c and Figure 10d respectively report the comparison of the run-time and the memory consumption with Arabesque and RStream. Comparing with Arabesque, Kaleido outperforms by a Ge-

Figure 12: Comparisons of isomorphism checking algorithms with bliss. Figure a and b compare 3-Motif and 3-FSM over Patent, MiCo and Youtube. Figure c compares 4-Motif and 4-FSM over Patent. Figure d compares 5-Motif and 5-FSM over CiteSeer. The upper figures compare the run-time; the lower figures compare the memory consumption

oMean of 34.1× and the memory consumption reduces by a GeoMean of 10.6×. Arabesque has a good performance in running 5-cliques over MiCo. Because MiCo is a denser but smaller graph. The ODAG saves a huge amount of memory while CSE must store many repeating vertices in deeper layers. Comparing with RStream, Kaleido outperforms by a GeoMean of 72.0× and the memory consumption reduces by a GeoMean of 25.0×. 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.

6.3 Comparisons with Isomorphism Checking Algorithms

In this section, we compare our isomorphism checking algorithm with the state-of-the-art library, bliss [18]. To test bliss, we replace the isomorphism checking algorithm in Kaleido with bliss.

Figure 12 illustrates the comparison with bliss. To fully evaluate Kaleido's isomorphism checking algorithm, we compared 3-FSM, 4-, 5-FSM and Motif Counting respectively over different datasets. For motif counting, the speedup is 5.8× but the memory consumption is similar. For FSM, the speedup is 2.1× 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 Bliss builds search trees. In FSM, Bliss needs larger hash space and consumes more memory than motif counting. On the other hand, the counting of motifs 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. Therefore the speedup of replacing Bliss in FSM is not as high as motif counting.

Figure 13: Comparisons of Kaleido and bliss in running 3-FSM and 4-FSM when the number of vertex labels changes in 7-label Patent (PA-7) and 37-label Patent (PA-37). Figure a and b show results of 3-FSM; Figure c and d show results of 4-FSM. The x-axis indicates the support of FSM applications.

Since the graph Patent possesses two levels of vertex labels, the category and the sub-category of each patent, we tested the performance of 3-FSM and 4-FSM over Patent in Kaleido and bliss with several supports, when the number of vertex labels changes. Figure 13 illustrates the result. When the number of vertex labels increases, both Kaleido and Bliss needs more time. The reason is the implementation of FSM in Kaleido, in which Kaleido does not statistic the accurate MNI support of each pattern but prunes patterns which reach the support threshold from the candidate instead. When the support threshold is fixed, the 37-label Patent (PA-37) ought to return less frequent patterns than the 7-label Patent (PA-7). As Figure 11 illustrates, the run-time trend of the support in 3-FSM is to increase first and then decrease. Note that Figure 11 illustrates that the run-time trend is to decrease when the support is larger than 10,000. Thus Figure 13b shows more run-time in PA-37 than Figure 13a in PA-7. The number of frequent embeddings in PA-7 and PA-37 is close (335,781,273 and 335,035,665 respectively in support 300). It almost needs no extra space in Kaleido, while bliss needs extra memory to build more complexity search trees and maps to a larger hash space. In testing 4-FSM, the intermediate embeddings account for the major memory consumption, while Bliss needs longer run-time to check the isomorphism over embeddings. The number of frequent embeddings 4-FSM with support 500 K over PA-7 and 4-FSM with support 100 K over PA-37 is close (1,303,911,410 and 1,490,970,608). The run-time and the memory consumption of these two applications is close too. The reason is the support constrains that frequent patterns contain only one kind of vertex label. It concludes that Bliss is more sensitive with the vertex label information of the input graph than Kaleido; when the number of vertex labels increases, Bliss needs more space to calculate the hash value of each pattern.

6.4 Scalability

We tested 3-FSM with 5000 support, 3-Motif and 5-Clique over Patent in varying numbers of threads. Figure 14 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 multithread is inevitable in our implementation. If we could replace it by a efficient concurrent hashmap, the scalability of FSM in Kaleido would near linear scaling.

(a) Run-Time (b) Memory Consumption

Figure 14: Scalability of Kaleido in 2, 4, 8, 16, 32 threads. Figure 14a shows the run-time; Figure 14b shows the memory consumption. The dotted lines indicate the ideal run-time and memory consumption respectively.

6.5 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 4 reports the result of these applications. It illustrates that the performance attenuation of using hybrid storage in Kaleido is acceptable (lower than 30% in these applications). For 4-FSM over Patent, the memory consumption reduced by the size of the last layer in CSE. For 4-Motif, 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 4: Performance of Kaleido on the hybrid storage in 4-FSM over Patent with 50k and 100k supports and 4-Motif over Patent and MiCo.

Apps	In-Memory	Time(s)	Memory (GB)
4-FSM(PA,50k)	Yes	312.1	76.7
	No	362.7	15.8
4-FSM(PA,100k)	Yes	125.7	32.8
	No	135.8	11.4
4-Motif(PA)	Yes	152.2	2.5
	No	249.2	2.7
4-Motif(MC)	Yes	198.1	0.6
	No	247.5	1.4

Figure 15: 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.

Figure 16: 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.

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 [2] in Linux to limit the maximum RAM of Kaleido in our experimental environment.

Figure 15 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 16 illustrates the run-time of different limitations of max RAM. When the limitation of maximum RAM is lower than 20 GB, the application reads the intermediate data from the disk and the run-time increases within 20%.

6.5.1 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 17. The prediction of the candidate size saves the run-time in the exploration over hybrid storage and it outperforms 1.2× over non-prediction. Figure 18 illustrates the CPU utilizing rate of 4-FSM over Patent with supports 50 K and 100 K and the prediction improve the efficiency of the exploration significantly.

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.

Graph Mining Algorithms gSpan [35] is an efficient frequent subgraph mining algorithm designed for mining

---

[2] A cgroup is a collection of processes that are bound to a set of limits or parameters defined via the cgroup filesystem. http://man7.org/linux/man-pages/man7/cgroups.7.html

Figure 17: 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.

Figure 18: Comparisons of the CPU utilizing rate in 4-FSM over Patent with supports 50 K and 100 K. The dotted boxes indicate the embedding exploration phase in FSM.

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.* [19] 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 [11] 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.* [25] introduces the motif counting problem. Ribeiro *et al.* [27] presents G-Tries which is an effective approach for storing and finding the frequency of motifs. Aparício *et al.* [3] designs and implements a parallel version of G-Tries. Maximal clique is well studied problem.

Graph Mining Systems Arabesque [33] 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 [26] 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 Map-Reduce 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 [7] 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 [17] 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 [34] 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 mainly strategy of the canonical labeling is building search tree for the input graph. Nauty [22] 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 [24] introduces a breadth-first search in generating the search tree. Saucy [9] is an implementation of the Nauty system by utilizing the sparsity and particular construction of colored graphs. However, these libraries focus on the checking of the automorphism, which only suits for the unlabeled graphs. Bliss [18] 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. In addition, bliss is designed for the large graph isomorphic checking, while the eigenvalue checking strategy is sufficient in the mining scenes.

8. 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 an lightweight and efficient graph isomorphism checking algorithm for small graphs in which the number of vertices is less than 9. Experimental results demonstrates 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 the state-of-the-art graph library.

9. 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] 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.

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

[5] 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.

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

[7] 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.

[8] X. Cheng, C. Dale, and J. Liu. Statistics and social network of youtube videos. In *2008 16th Interntional Workshop on Quality of Service*, pages 229–238. IEEE, 2008.

[9] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *DAC*, pages 530–534. ACM, 2004.

[10] 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.

[11] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB 14*, 7(7):517–528, 2014.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 345–356. SIAM, 2004.

[20] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD 15*, pages 177–187. ACM, 2005.

[21] 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.

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

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

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

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

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

[27] 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.

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

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

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

[31] 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.

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

[33] 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 15*, pages 425–440. ACM, 2015.

[34] 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.

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

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