Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow

Chuangyi Gui, *National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China; Zhejiang Lab, China;* Xiaofei Liao, *National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China;* Long Zheng, *National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China; Zhejiang Lab, China;* Hai Jin, *National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China*

https://www.usenix.org/conference/atc23/presentation/gui

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow

Chuangyi Gui[†‡], Xiaofei Liao[†], Long Zheng[†‡], Hai Jin[†]
[†]*National Engineering Research Center for Big Data Technology and System/ Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China*
[‡]*Zhejiang Lab, China*

Abstract

Graph pattern mining is an essential task in many fields, which explores all the instances of user-interested patterns in a data graph. Pattern-centric mining systems transform the patterns into a series of set operations to guide the exploration and substantially outperform the embedding-centric counterparts that exhaustively enumerate all subgraphs. These systems provide novel specializations to achieve optimum search space, but the inherent redundancies caused by recurrent set intersections on the same or different subgraph instances remain and are difficult to trace, significantly degrading the performance.

In this paper, we propose a dataflow-based graph pattern mining framework named Cyclosa to eliminate the above redundancies by utilizing the concept of computation similarity. Cyclosa is characterized by three features. First, it reorganizes the set operations for a pattern into a set dataflow representation which can elegantly indicate the possibility of redundancies while sustaining the optimal scheduling for high performance. Second, the dataflow-guided parallel execution engine decouples data access and computations to enable efficient results sharing. Third, the memory-friendly data management substrate can automatically manage the computation results with high reuse possibility. Evaluation of different patterns demonstrates that Cyclosa outperforms state-of-the-art pattern-centric systems GraphPi and SumPA by up to 16.28× and 5.52×, respectively.

1 Introduction

Graphs are the *de facto* paths to explore useful information in various fields, including social media analysis [1, 2], financial networks [3, 4], and bioinformatics [5]. Graph pattern mining aims to explore interesting subgraph structures according to the user-given constraints. Typical graph pattern mining applications include subgraph matching [6, 7], clique finding [8, 9], and motifs counting [10–12]. Despite the prevalence of graph pattern mining applications, they have high computational complexity and usually need hours or even days to complete [13–15].

Graph pattern mining systems have emerged in recent years to provide high performance and programmability [16–18]. A common approach is to enumerate all the subgraphs, usually under a certain depth, to check whether the subgraphs satisfy the pattern constraints, which is called the embedding-centric paradigm [16, 19]. This approach is easy to develop and parallelize. However, it results in high memory consumption and wasted computing resources due to a large number of intermediate partial instances [17, 20]. Recently, advanced graph pattern mining systems have adopted a pattern-centric paradigm to overcome inefficiencies [17]. The main idea is to use the structure information of graph patterns to filter intermediates that will not lead to a correct final match. This is achieved by transforming the graph patterns into a series of set operations and executing them in a nested loop following a matching order of pattern vertices. Each loop computes the candidates of corresponding pattern vertex, where the computation is represented as a formula of set intersections on the neighboring lists of previously matched pattern vertices based on the structural connectivity, e.g., $Cand(v_2) = N(v_0) \cap N(v_1)$ for a triangle pattern after matching an edge (v_0, v_1). In this way, only valid partial instances are produced in each loop.

Prior works propose many novel techniques to reduce the search space also the number of partial instances to be explored. AutoMine [17] provides a convenient compiler to generate optimized matching orders automatically (also the order of computations), which will significantly influence the search space. Peregrine [20] and GraphZero [21] introduce a symmetry-breaking method that filters the partial instances leading to the same final mapping by comparing vertex IDs of symmetric vertices. GraphPi [22] further explores an optimal combination of different symmetry-breaking rules and matching orders to minimize the search space. The above systems mainly optimize the mining of a single pattern. SumPA [23] further proposed an abstraction approach to reduce workloads for mining multiple patterns simultaneously.

However, a large number of intrinsic redundant computations remain in the execution of set operations even in an optimized search space, the same set intersection, e.g.,

$N(v_0) \cap N(v_1)$, repeats throughout the processing. Specifically, the redundancies can be classified into two categories, the *explicit redundancy* and *implicit redundancy*, based on whether they rely on the same subgraph instance. In explicit redundancies, one set intersection can be repeatedly used for computing different pattern vertices connected to the same subgraph instances. In implicit redundancies, the same intersection appears in computing on different subgraph instances. The redundant computations usually cost more than 80% of the runtime and severely degrade the performance. Furthermore, these explicit and implicit redundancies spread over the runtime when mining a single pattern or multiple patterns, making it difficult to trace and reuse. Existing systems follow the principle of structural equality and rewrite the exact same set formulas of different vertices into a single one to reduce part of the explicit redundancies [21–23]. However, there are still more than 60% of the redundancies remaining unsolved.

We observe two kinds of computation similarity in the set operations providing the opportunity to help identify and reuse both explicit and implicit redundancies. The first one is the static similarity which reveals the structural similarity among the operands of the set operations in a pattern, that one input operand can be reused in two operators, and the output results of these operators may have latent redundancies. The static similarity can be analyzed before execution. We propose to decouple the operands and operators of all set operations and organize them into a directed flow oriented by the connections among the inputs and outputs, which is called a *set dataflow*, to efficiently exploit the static similarity. Each node in the set dataflow is a set operand or a set operator. The explicit redundancies can be removed by keeping only unique operands in the dataflow, while the implicit redundancies between different operators can be indicated by the overlapped input source nodes of the dataflow.

The second one is the dynamic similarity which reveals the similarity among the inputs of all occurred set intersections during runtime, which can only be analyzed after execution. Specifically, we observe that a small number of high-degree vertices participate in most of the computations, which means redundant computations are concentrated in these vertices. This allows us to cache and reuse implicit redundant results by tracing these high-degree vertices. However, maintaining correctness and efficiency in the execution of the dataflow is challenging. How to manage the intermediate computation results in limited memory space is also a main difficulty.

In this paper, we present Cyclosa, a novel dataflow-based graph pattern mining system to eliminate both explicit and implicit redundancies in the pattern-centric paradigm. Specifically, Cyclosa is characterized by the following key features. First, we propose a novel set dataflow representation and an efficient constructing approach to generate the set dataflow for arbitrary patterns. It maintains optimized cost through a lightweight cost estimation model and introduces the redundancy probability to guide the appropriate reusing of results. Second, we develop a dataflow-based execution model to expose the possibility of capturing and reusing redundancies during runtime. Through the dataflow execution, the data accesses and computations are decoupled, providing the ability to maximize data reuse in parallel. Third, we design a memory-friendly data management substrate to automatically store the computation results with a high possibility of redundancy. It implements smart cache strategies according to the reuse probability of results evaluated, thus achieving a controlled memory consumption. Furthermore, the substrate can efficiently cooperate with the dataflow execution engine to provide the results requested by repeated computations. We implement a prototype of Cyclosa and achieve significant performance improvement over state-of-the-art pattern-centric graph pattern mining systems.

The key contributions of this paper are as follows:

- It introduces a set dataflow representation to explore the fine-grained computation similarity in set operations of pattern-centric graph mining for reducing both explicit and implicit redundancies.
- It proposes a dataflow-guided execution model and a self-managed redundancy storage substrate to efficiently share results among computations, overcoming the challenges of managing and reusing the results.
- It develops Cyclosa, a high-performance graph pattern mining system that eliminates redundant computations for various pattern settings. Experimental results show that Cyclosa outperforms GraphPi and SumPA by up to 16.28× and 5.52×, respectively.

2 Background and Motivation

2.1 Definition of Graph Pattern Mining

Given a data graph $G = \langle V, E \rangle$, where V is the vertex set and E represents the edges, and an input graph pattern p which can be arbitrary graphs, the basic graph pattern mining problem aims to find all the subgraphs of G that are isomorphic to p [20–22]. Each isomorphic subgraph is called an *embedding* of p, and the vertices and edges form an one-to-one mapping between the embedding and p. For example, Figure 1 shows a data graph and a diamond pattern. The subgraphs connected by (2, 3, 0, 4) and (2, 3, 1, 4) are two embeddings of the diamond pattern, and the vertices are mapped correspondingly. As in prior studies, this work focuses on graph pattern mining on the undirected graphs.

Graph pattern mining applications have many variants which may require either a single pattern or multiple patterns to be mined. For example, *subgraph listing* outputs all the instances of a given pattern [24]. The *k-clique finding* counts the complete subgraphs with a certain number of vertices [25]. The *k-motifs counting* counts the number of instances for all

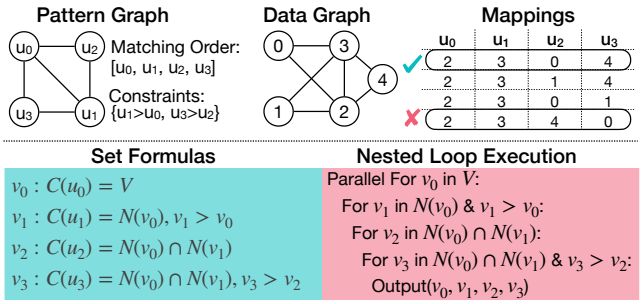


Figure 1: The example of mining a diamond pattern in a pattern-centric system

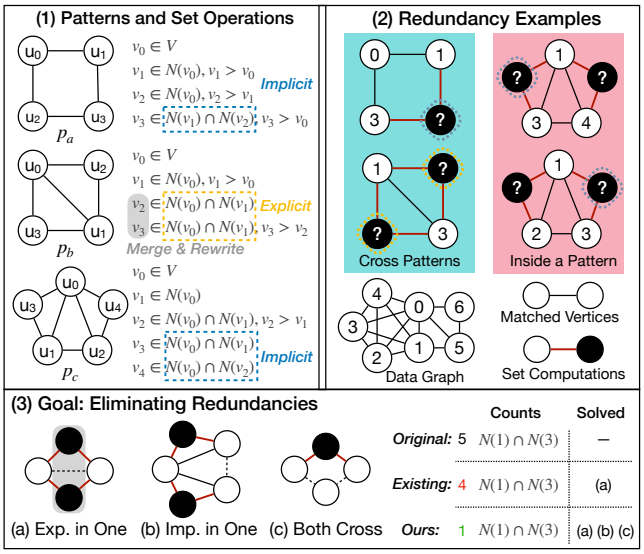


Figure 2: An example of redundant computations in graph pattern mining. The black vertices in dashed cycles represent redundant computations $N(1) \cap N(3)$ in different situations.

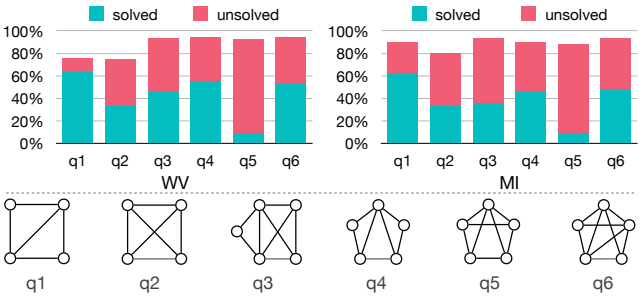


Figure 3: The ratio of redundancy as repeated set intersections for different patterns on *Wiki-Vote* (WV) and *MiCo* (MI) graphs. Solved: the redundancy amount that can be addressed by SumPA against a naïve nested loop execution. Unsolved: the number of remaining redundancies in SumPA.

2.2 Procedure of Graph Pattern Mining

In pattern-centric systems, the mining procedure is typically performed as a series of nested set operations in three steps [17, 22, 27]. Firstly, the graph pattern is analyzed to generate a matching order of pattern vertices. The vertices in the data graph will be computed and mapped to the pattern following the order. Secondly, the set operations required for computing each pattern vertex will be generated based on its prior neighbors in the matching order. Lastly, the set operations will be executed in a nested loop that starts from each data graph vertex. The end of each loop represents that a subgraph instance is found. Usually, the outer loop is executed in parallel. Existing systems focus on minimizing the branches that need to be accessed in the search tree by generating optimized matching orders. In order to further reduce the search space, the symmetry-breaking method is also applied by adding comparison constraints to filter computed results that will lead to an automorphic instance.

For instance, Figure 1 shows how to find the subgraphs of a diamond pattern. In the pattern analysis phase, the matching order of the diamond is defined as $[u_0, u_1, u_2, u_3]$. There are comparison constraints between vertex pairs $<u_1, u_0>$ and $<u_2, u_3>$ because they are symmetric. Following the matching order, we can formulate the set operations for each pattern vertex. Initially, u_0 can be mapped to any of the data graph vertices while u_1 is represented as finding a neighbor of u_0. For vertices u_2 and u_3, they are common neighbors of u_0 and u_1, so that a set intersection is defined respectively. The constraints are checked when $<u_1, u_0>$ and $<u_2, u_3>$ are involved in the computations. These set operations are organized into a nested loop of 4 depths for execution. The first two loops map edges of the data graph to (u_0, u_1). Assuming that the edge (2, 3) has been assigned, the candidates for u_2 and u_3 will be {0, 1, 4}. Due to the symmetry-breaking constraints, we can safely filter (2, 3, 4, 0) and avoid mapping (2, 3, 0, 4) twice because they are the same subgraph.

Our work incorporates existing optimizations of reducing the size of explored search space while enabling high efficiency on computations by eliminating redundancies.

2.3 Problem: Redundant Computations

Despite the pattern-centric mining procedure providing good optimizations on the search space, time-consuming explicit and implicit redundancies exist in the procedure of mining single or multiple patterns. This problem is detailed in Figure 2. The set operations for each vertex corresponding to the matching order and constraints of patterns P_a, P_b, and P_c are given. Following the matching order, consider that a part of

the substructures of these patterns have been mapped to the data graph, which is denoted as white vertices. Next, we will explain the explicit and implicit redundancies in detail.

Explicit Redundancy: A set of intersections that are repeatedly operated upon the neighboring lists of the same vertices from the *same* subgraph instance. For pattern P_b, (1, 3) is partially assigned to (u_0, u_1). In order to compute u_2 and u_3, $N(1) \cap N(3)$ is performed twice. For pattern P_c, the computation $N(1) \cap N(3)$ must be performed in order to map (1, 3, 4) to u_2. To this end, the explicit redundant computation $N(1) \cap N(3)$ is conducted based on the same edge instance (1, 3) inside single pattern P_b and also in the pattern P_c.

Implicit Redundancy: A set of intersections that are repeatedly operated upon the neighboring lists of the same vertices from the *different* subgraph instances. In single pattern P_c, consider two subgraph instances (1, 3, 4) and (1, 2, 3) are already mapped to (u_0, u_1, u_2). The repeated set intersections $N(1) \cap N(3)$ exist for computing u_3 and u_4. They are induced from different subgraph instances. For multiple patterns, when (0, 1, 3) is partially matched to P_a, the computation of $N(1) \cap N(3)$ is induced from different subgraph instances of all three patterns.

These redundancies can be aggravated in a nested loop of set operations as in Figure 1, e.g., resulting in more than twice the number of redundancies for computing u_3 of P_b. As profiled in Figure 3, more than 80% of total computations are redundant in different patterns. However, existing systems can only explore parts of the explicit redundancies because they view each set formula (or a pattern vertex) as a whole in each loop and all follow the principle of structural equality to merge equal set formulas. For example, in Figure 4, GraphPi [22] and GraphZero [21] will merge and rewrite the formulas S_3 and S_4 into one loop. SumPA [23] will reduce v_2 and v_3 into an abstract pattern vertex and then generate a single set formula for two vertices. However, the implicit redundancies remain because they cannot be exposed as structural equality in set formulas. Besides, in parallel execution, explicit redundancies on a reverted edge such as $N(1) \cap N(3)$ and $N(3) \cap N(1)$ will be omitted. As profiled in Figure 3, existing work can only solve less than 40% redundancies. Our work aims to eliminate both explicit and implicit redundancies.

2.4 Insights: Computation Similarity

In order to detect and reuse both the explicit and implicit redundancies, we must explore finer-grained computation similarity rather than the structural equality as in existing work. If the similarity of computations can be identified before the execution, then we can use it to guide the reusing of results. The computation similarity comes from two folds:

- *Static Similarity.* The static similarity originates from the operands level of the set operations for a pattern, exposing the reuse possibility of both inputs and outputs of different computations. Figure 4 presents the set formulas of P_c in Figure 2, other than the formula equality of S_3 and S_4, the input operand $N(v_0)$ is also getting involved in S_2 and S_5. $N(v_0)$ can be shared as inputs for four formulas, and it also results in the implicit redundancies between S_4 and S_5.

- *Dynamic Similarity.* The dynamic similarity lies in the runtime characteristics of the inputs of occurred computations, reflecting which vertices are more likely to be requested for computation. Specifically, we observe *most of the computations are concentrated in a small part of high-degree vertices.* We make an analysis of 4-motifs counting on two graphs as shown in Figure 5. The vertices of the graphs are reordered by a decreasing degree. More than 85% of the computations lie in about 15% of the first high-degree vertices, where the redundancies also concentrate.

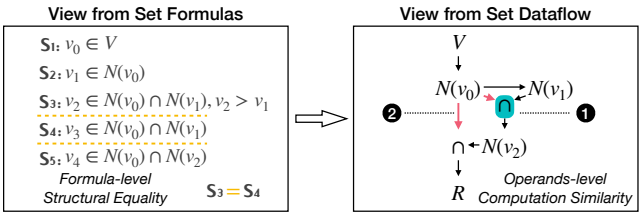


Figure 4: The dataflow view for analyzing the mining procedure of p_c in Figure 2. ❶ S_3 and S_4 are reduced. ❷ Results of $N(v_0)$ are reused in two ∩.

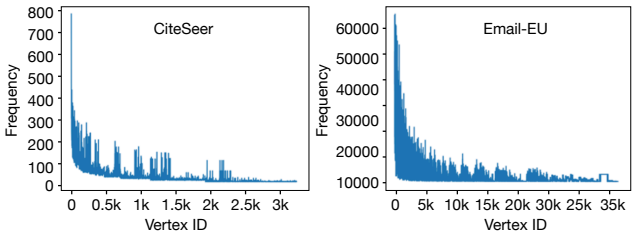


Figure 5: The frequency of vertices getting involved in computations for size-4 motifs counting on two graphs

In this work, we propose a *set dataflow* to use the computation similarity for redundancy elimination, as shown in Figure 4. The set dataflow is a directed graph indicating the procedure of how sets are transferred and computed. The set dataflow decouples the set formulas into individual operands and operators, and the directed edges represent the transfer relation of the input/output data between different operators. It removes explicit and implicit redundancies as follows: ❶ The explicit redundancies can be removed by cutting and maintaining unique operators, e.g., only single $N(v_0)$ and $N(v_1)$ exist. Original two $N(v_0) \cap N(v_1)$ are thus reduced to one, and the set operands are fully shared. ❷ The implicit redundancies between operators are indicated by overlapped inputs, e.g., the results of $N(v_0) \cap N(v_1)$ and $N(v_0) \cap N(v_2)$. Based on the dynamic similarity, we can heuristically cache the computation results of high-degree vertices for reusing.

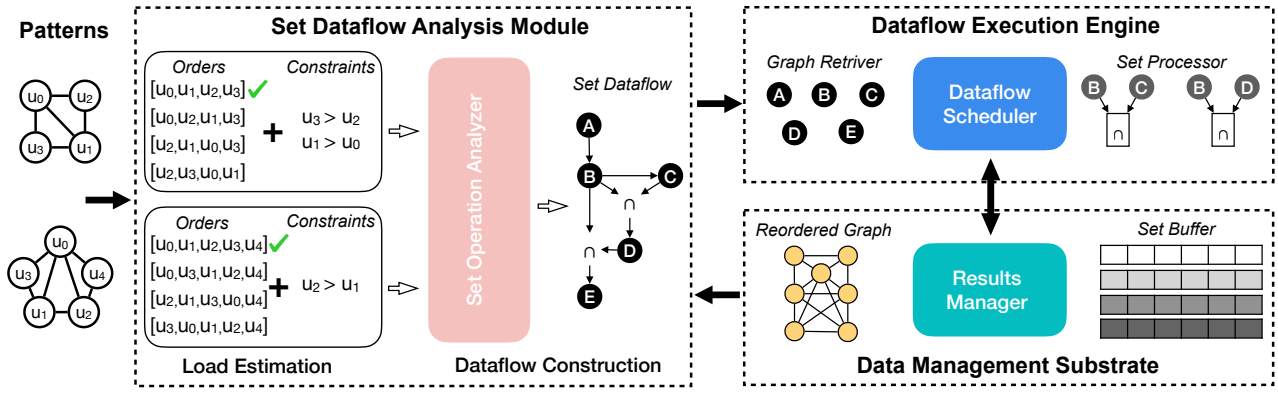


Figure 6: Overview of Cyclosa

Challenges. However, there are still several challenges in constructing an efficient redundancy-free graph pattern mining system. First, maintaining correctness and efficiency is difficult. The set dataflow is only aware of sets during execution, and we need to extract the results correctly while enabling maximum data sharing. Second, managing a large number of redundant results is challenging. Simply storing all the redundant results indicated by set dataflow is inefficient and may consume a large size of memory footprint.

3 System Overview

Cyclosa is architected to solve the above challenges and achieves a redundancy-free execution for graph pattern mining. Given arbitrary graph patterns, Cyclosa can fast analyze the computation similarity via the set dataflow and use the set dataflow to guide the sharing of computations with high parallelisms. Specifically, Cyclosa works with three main modules, as presented in Figure 6.

Set Dataflow Analysis Module. The set dataflow of input patterns is constructed in this module. Each pattern is first analyzed to generate a reuse-aware matching order and constraints with data graph properties. Then, based on the matching order, a set operation analyzer generates a redundancy-reduced set dataflow by keeping unique operands. Through the analysis, the generated set dataflow will maintain sufficient information for correct results and indicate the redundancy probability of different set operators.

Dataflow Execution Engine. The generated set dataflow is fed to the dataflow execution engine for processing. It keeps a decoupled view of the data access and computation that provides the opportunity to manage the results independently. Each node in the dataflow is assigned to a worker to process. The inputs and outputs are managed by the graph retriever communicating with the data management substrate. The set operators are assigned to redundancy-aware set processors that will request redundant results before the computation. A dataflow scheduler controls the processing order by directed edges of the set dataflow to ensure correctness.

Data Management Substrate. Computed results are automatically maintained in the substrate. Once received a result set, a results manager will implement smart cache strategies and maintain results with different reuse possibilities in a proper place of a multi-level set buffer. In this way, the memory footprint is under control. Furthermore, the results management can overlap with the execution engine for high parallelism. The substrate also provides a fast request to the data graph and cached results through efficient data layout.

4 Set Dataflow Analysis

This section first introduces the approach to generate a cost-efficient reuse-aware matching order and then describes the procedure for efficiently constructing a set dataflow.

4.1 Pattern Analysis

Existing approaches enumerate all possible orders and estimate the cost to find the order with the lowest workload [17, 20, 22], but they are oblivious to the redundant computations exposed in runtime characteristics. Besides, the overhead for enumerating all orders will increase when patterns get larger. In this work, we propose a degree-guided two phases analysis, as shown in Figure 7, to solve the above challenges. The main idea is to match the high-degree vertices in a *Depth-First-Search* (DFS) manner to raise the possibility of reusing the results on these vertices. At the same time, we design a lightweight and efficient cost model with graph information for the optimal cost.

DFS Order Enumeration. This phase will generate all degree-first DFS orders of a pattern. Firstly, the constraints of symmetric vertices are generated using the permutation group theory as in GraphPi [22], independent of the order of vertices. In the example of Figure 7, there are two equal constraints because each is sufficient to breaking symmetries, and the $u_0 < u_2$ is selected randomly. Secondly, it traverses

from the pattern vertices with the highest degree and follows a DFS style to get all valid matching orders, e.g., the order starting from u_1 with a degree by 4 is valid while the one from u_0 with a degree by 3 is discarded. These orders will then be estimated for the minimum cost.

Graph-Aware Cost Estimation. We estimate the total number of intermediate subgraph instances as the cost. Given a matching order, we can get the set formulas for computing each pattern vertex. Cyclosa then iteratively estimates the number of instances produced in a nested loop execution of the set formulas. Existing works typically use a fixed metric (i.e., average degree) to predicate the number of newly generated instances from each subgraph [17, 23]. However, this approach omits the filtering effect of set intersections and is inaccurate [28]. Cyclosa incorporates more graph information by combining the vertex degree and triangle-count-per-edge for estimation, because the triangle count enables capturing the reduction information on neighboring lists after an intersection on two or more vertices.

As shown in Figure 7, initially, $|V|$ vertices can be mapped to u_1 after `Loop0`. In `Loop1`, because only one $N(u_1)$ operator is used for matching u_0, we use the average degree deg to estimate the number of newly produced instances from each instance in the previous loop. The total instances in `Loop1` are $|V| * deg$. In `Loop2`, since the results are produced by an intersection on $N(u_0)$ and $N(u_1)$, we use the triangle-count-per-edge $ntri$, instead of deg, to estimate the number of newly generated instances. Note that the constraint $u_0 < u_2$ is applied to `Loop2` so that some produced instances will be filtered. We use a parameter α to reflect the reduction of instances. The cost estimated for subsequent loops is similar. Finally, the total cost of this order is calculated by summarizing the costs of all loops for selection.

Similar to Cyclosa, there is also research [28] considering the data graph properties and using the number of sub-patterns in a sampled graph to estimate the cost of the whole graph. However, it requires the extra sampling and matching phase to get the cost. Different from existing systems, Cyclosa precomputes triangle counts and degrees of the original graph to preserve accuracy without introducing extra steps. Additionally, Cyclosa also considers the factor of instances filtering by symmetry-breaking constraints.

4.2 Dataflow Construction

Given the matching order, a pattern can be transformed into a series of set formulas as in prior work. Based on the set operation, there are mainly two challenges for generating the set dataflow. First, the set dataflow must be aware of existing optimizations on symmetry-breaking. Second, we need a fast approach to save construction time when facing a large number of set operations.

We propose a novel abstraction for representing the set dataflow by introducing three kinds of set operators to provide

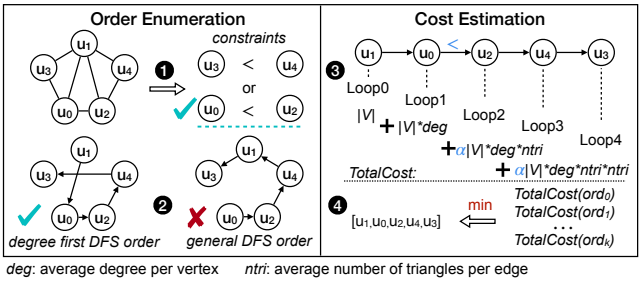


Figure 7: The procedure of finding an appropriate reuse-aware matching order for a pattern by ❶ generating constraints, ❷ enumerating valid DFS orders by degrees, ❸ estimating cost for each order with graph information, and ❹ selecting the order with minimum cost

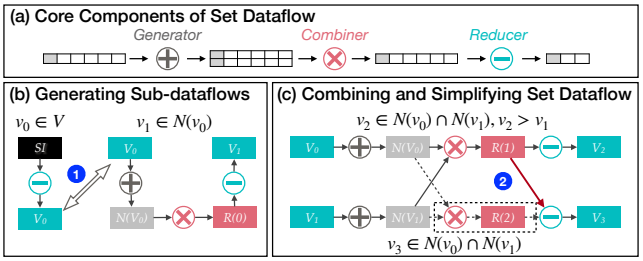
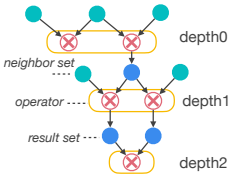


Figure 8: An example of constructing the set dataflow given four set formulas. The sub-dataflow of each set formula is first constructed with three operators, and the input/output sets of each operator are uniquely assigned and identified. The sub-dataflows are then combined into a final set dataflow by ❶ reducing inputs/outputs and ❷ simplifying set operators.

a uniform view and contain all information of set operations including the symmetry-breaking: `Generator`, `Combiner`, and `Reducer`. The `Generator` consumes a valid candidate set of a pattern vertex to generate the neighboring sets. The `Combiner` receives two sets and outputs a single set. The `Reducer` checks a result set and selects valid candidates following filtering rules to produce a new candidate set for certain pattern vertices. Based on these operators, we adopt the idea of divide-and-conquer by first generating the sub-dataflow of each set operation and then combining them to get the final set dataflow, as shown in Figure 8.

Generating Sub-Dataflows. The sub-dataflow of each formula must contain the `Generator`, `Combiner`, and `Reducer` at the same time, except of the initial one because there is no intersection required. In this way, the data required for computation and the operators performed are separated, which are represented in a unified style. For example, in Figure 8, the sub-flow of initial $v_0 \in V$ only contains one `Reducer`. For $v_1 \in N(v_0)$, the $N(v_0)$ is first transformed into a flow with a `Generator`, the output of the `Generator` is then sent to a `Combiner` and is finally checked by a `Reducer`. Similarly, the sub-dataflow for $v_2 \in N(v_0) \cap N(v_1)$ is given, and the con-



Probability. Reusable Results

	NST	HST	RST
Same-Depth	*high*	*middle*	*low*
Diff.-Depth	*middle*	*low*	*none*

Probability Table

	R(0)	R(1)	R(2)
R(0)	1	0.3	0.9
R(1)	0.3	1	0.3
R(2)	0.9	0.3	1

Figure 9: Different levels of redundancy probability. NST, HST, and RST denote operators sharing the same neighboring sets, sharing the same result set and different neighboring sets, and sharing only same result sets, respectively. The redundancy probability of each pair of result sets $(R(i), R(j))$ is stored in a table where higher values denote higher probability to be reused in the future.

straints will be recorded in the metadata of the `Reducer`.

Combining and Simplifying Set Dataflow. Different sub-flows are first combined by connecting same inputs and outputs, e.g., the sub-flows of $v_1 \in N(v_0)$ and initial formula are combined by the output and input candidate sets denoted with V_0. Only one candidate set of V_0 will be maintained after the combination. After the combination, we then traverse the combined set dataflow in a BFS style to cut set operators with the same input nodes. Consider two `Combiners` used for computing the results of v_2 and v_3. They have the same input nodes so that only one `Combiner` remains. Thus, the explicit redundancies of these two operators are eliminated. After the simplification, the final set dataflow is generated.

4.3 Dataflow Evaluation

This evaluation aims to exploit the latent probability of redundant results among different *Combiners* in the set dataflow, which will guide the storing of results. However, providing an exact prediction of the probability before execution is difficult. Therefore, we propose to qualitatively analyze the latent probability of redundancy for different set operators through their depths and input information in the dataflow.

Generally, the operators with at least one shared input source and at the same depth have a higher possibility of producing similar results. Considering the view of pattern structure, this can be understood as they have similar substructures. In Figure 2, the computations for u_3 and u_4 have produced the same results during runtime because these two vertices have similar triangle structures and share the u_0. Figure 9 summarizes the reuse probability under different situations of the combinations of depth and shared input sets. The redundant probabilities of different `Combiners` are recorded in a table together with the set dataflow for execution.

5 Redundancy-Free Pattern Mining

This section introduces an efficient set-centric execution engine and a redundancy-aware data management substrate to enable high performance and optimal results reuse.

5.1 Set-Centric Dataflow Execution

The set formulas are processed in a nested loop in prior works. Despite of the convenience of parallelizing, it lacks the ability of fine-grained data management and reuse. In this work, we present a set-centric dataflow execution engine that decouples data management from computation to maximize results sharing. The core idea is to put the set instead of a subgraph as the basic processed element. The set operator in the set dataflow will start processing whenever the input sets from directed edges are ready. This discrete view enables sharing any results to any of the computations.

To realize the goal, we correspondingly design an executor for each `Generator`, `Combiner`, and `Reducer` together with a *Dataflow Monitor*, as shown in Figure 10(a). Each set contains two parts for identification: the set ID and the elements value, e.g., ID $< 3, -1 >$ for the neighboring set of vertex 3. Set operators coordinate through flow signals. The monitor identifies from the flow signals to know where to move the results and activates the next operator.

Generator Module. It traverses each vertex element from the input candidate set or the initial set to generate related neighboring sets. Each input set is assigned a signal consisting of an operator ID and an instance ID. The output will inherit the operator ID, and a new instance ID is produced to indicate a newly generated subgraph instance for correctness. In the case of Figure 10(a), the signal $nei < 3, -1, op, gid^*, value >$ is sent to the monitor. The monitor will then transfer the output set to a `Combiner` needed.

Combiner Module. In order to support reuse-aware computation, it introduces a *check unit* and a *compute unit*. The check unit first queries whether there are already computed results with the same ID, $req < 3, 5, op, gid >$ in Figure 10(a). The monitor will transfer the request to the data manager. If the request hits, then the computation is omitted. Otherwise, the compute unit is called to generate a new output. The new result $res < 3, 5, gid, value >$ is then transferred to the monitor for the next operator.

Reducer Module. Each input to this module will be a result directly computed from the `Combiner` or fetched from the cached results. The output is a set with valid candidates for a pattern vertex. The constraint information is checked by accessing the metadata of the set dataflow. Once an output set is generated, the related signal, i.e., $ret < op, gid, value >$, is sent to the monitor.

Dataflow Monitor. The monitor coordinates the movement of sets guided by the set dataflow. It has three components: 1) the *Flow Map* storing the metadata of a set dataflow by recording unique IDs for operators, 2) the *Activator* handling the signals of operators and scheduling the sets based on the flow map, and 3) the *Retriever* communicating with the data manager for accessing the graph data and cached results.

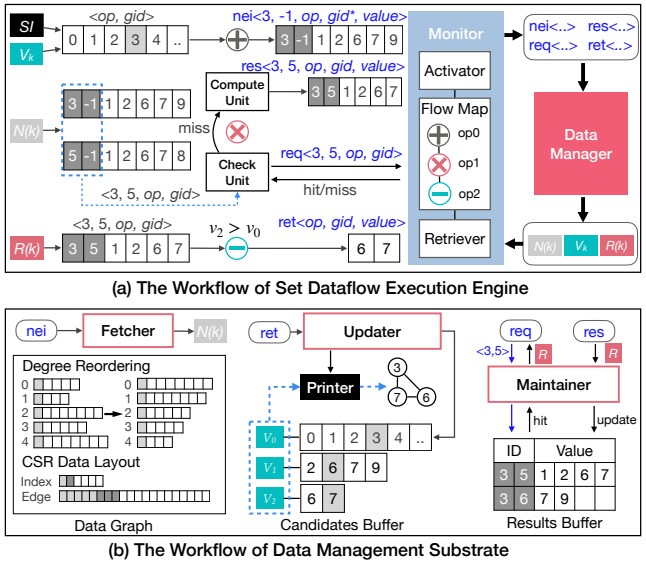


Figure 10: The workflows of set dataflow execution engine and data management substrate

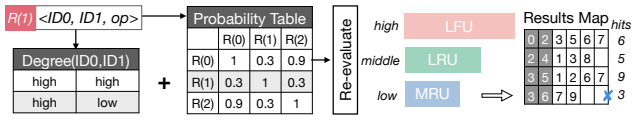


Figure 11: The caching strategy for result sets. The vertex degrees of result $R(1)$ are first checked, and only results with at least one high-degree vertex are maintained. The corresponding row of the probability table is then accessed to further decide the caching strategy.

5.2 Reuse-Aware Data Management Substrate

The dataflow execution engine will continuously produce new result sets. However, due to the large search space, maintaining the results in a limited memory space while maintaining high reuse ratio is challenging. We propose a memory-friendly data management substrate to solve this challenge. The main idea is to selectively store result sets with the highest reuse probability. It also overlaps with the computation phase to embrace correctness and efficiency. The substrate implements three main components: the results maintainer, the candidates updater, and the graph fetcher, as shown in Figure 10(b).

Results Maintainer. The cores of the results maintainer are three fixed-size result buffers under different caching strategies, i.e., the *Least Frequently Used* (LFU), the *Least Recently Used* (LRU), and the *Most Recently Used* (MRU) buffers storing results with high, middle, and low reuse probability, respectively. High reusable results come from high-degree vertices, which are more likely to be generated at the upper levels of the dataflow and are the most frequently requested. Middle reusable results are from parallel computations at the same middle level, yielding better time locality. Low reusable results occurs at the lower levels, which are often infrequently requested by low-degree vertices that are deferred to be processed in Cyclosa. This module dynamically maintains the computed results and responds to the *req* and *res* signals. The sets in the buffer are stored in a <key, value> manner. When a *req* signal arrives, it will search for results by the set ID, update the hit information, and respond to the dataflow monitor. When a *res* signal arrives, the maintainer will estimate the reuse probability with smart caching strategies and store the result in a proper buffer.

The strategy for identifying the reuse probability of a result set is demonstrated in Figure 11. It combines the static and dynamic computation similarities to estimate the reuse probability during runtime. Higher values in the probability table of the set dataflow analysis and higher degrees of the vertices by the result set ID, in particular, will result in a higher reuse probability evaluated. For instance, in Figure 11, the result has a low reuse probability. This is because there is a low degree vertex in the computation, which may not participate in other computations.

Candidates Updater. It is responsible for maintaining the candidate sets produced by the `Reducer` and extracting correct subgraph instances. Each candidate set is allocated with independent memory space. When a *ret* arrives, the *op* information is used for indicating the correct candidate sets. After each update, a *Printer* will consume the candidate sets using the *gid* information once all candidate sets are updated. The *Printer* records current pointers in the candidates set and produces exact subgraph instances. In order to overlap with the updating process, we use a double buffer for extracting the subgraph instances in the *Printer*.

Data Graph Fetcher. It reorganizes the data graph to improve the reuse efficiency and responds to the *nei* signal by returning the neighboring list as a set to the execution monitor. Specifically, based on the degree information, the vertices and edges are reordered so that the vertices with higher degrees will be assigned smaller IDs. In addition, the edges of the high-degree vertices are stored in a contiguous space to improve the cache efficiency because these edges will be frequently accessed during computation. It also conducts a triangle counting process to get the number of triangles of the data graph for pattern analysis.

Discussions. Through the above designs, Cyclosa enables efficient results sharing for redundant computations. In the case of operating intersections upon small sets, directly recomputing may be faster than reusing the results. However, this case rarely happens since most cached results are related to high-degree vertices that have large-size sets. We track the cached intersection results of size-4 motif counting on MiCo graph in Cyclosa and find that only 7.3% of results benefit from recomputations while most prefer the reuse method that can yield higher speedups against the former.

Currently, Cyclosa focuses on mining unlabeled patterns because they generally yield higher computation complex-

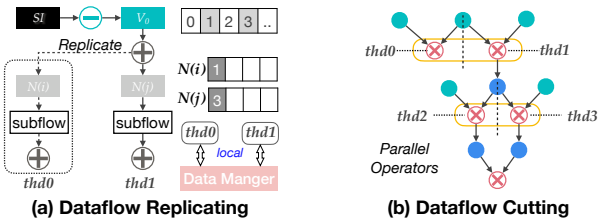


Figure 12: Parallel processing of the set dataflow that exploits multi-level parallelisms

Table 1: Real-World Graphs

Graphs	|V|	|E|	Size
WikiVote (WV)	7.1K	100.8K	0.81MB
MiCo (MI)	96.6K	1.1M	8.24MB
WikiTalk (WT)	2.39M	5.02M	40.16MB
Patents (PA)	3.8M	16.5M	0.12GB
LiveJournal (LJ)	4.0M	34.7M	0.26GB
Orkut (OR)	3.1M	117.2M	0.87GB
Friendster (FR)	65.6M	1.8B	13.46GB

ity than the labeled ones. For handling labeled patterns, we can simply add label information as constraints in `Reducers`. During execution, the structural information of a pattern is first explored for set computation, and the label information is then used only for filtering elements in a set. The labels provide more filtering opportunities that help reduce the total workload amount with fewer redundancies. However, the proposed set dataflow can still benefit labeled patterns by exploring the explicit redundancies in the pattern topology. Note that the dynamic computation similarity may change when the high-degree vertices have different labels.

6 Implementations

Cyclosa is currently built with C++ as an in-memory system on a single machine. This section will introduce the parallel implementations in Cyclosa for mining flexible patterns.

Flexible Pattern Interpretation. Cyclosa provides a convenient interface, `PatternGraph()`, for users to construct flexible pattern graphs by providing exact structures or graph properties, e.g., the edge list of a triangle or clustering coefficient, through *EdgeList* and *Restriction* parameters. The above information will be automatically interpreted into possible patterns. Users only need to interact with the `PatternGraph()` while the underlying runtime is transparent.

Parallel Execution of Set Dataflow. We use OpenMP for parallel execution in Cyclosa. The set dataflow inherently provides multi-level parallelisms, as shown in Figure 12. First, the sub-parts of a dataflow can be replicated and assigned to different threads to exploit the data parallelism. The starting point of the replicas is divided by the `Generator`. Take Figure 12(a) for example. Two threads handle the neighboring sets of vertex 1 and 3, respectively. Second, the `Combiners` at the same depth can be conducted in parallel because there are no dependencies, as shown in Figure 12(b). Note that the thread-local memory maintains an input set and an output set for every dataflow node which are reused throughout the execution. Since the number of dataflow nodes is small and the input/output set size is limited to the maximum degree of the data graph, the thread-local memory is often small.

Load Balancing. The skewness in Cyclosa relates to different numbers of sets produced by `Generators`. We address it in two folds: 1) *Static task assignment.* The input of the first `Generator` is initialized by assigning the reordered graph vertices in a round-robin fashion. This makes the upper `Generators` in a dataflow for different threads produce similar workloads. 2) *Dynamic work stealing.* This can be safely realized by managing the independent thread-local set space. Specifically, Cyclosa identifies the input-set position in the dataflow of the busy thread and replicates its workloads and relevant local set states to idle threads, which launch their corresponding `Generators` with higher parallelism.

Parallel Data Management. The data management substrate maintains the data graph and computation results. The data graph is stored in the *Compressed Sparse Row* (CSR) format. For cliques, the graph is oriented by enforcing a direction between each pair of vertices to reduce workloads. The results buffers are implemented using a concurrent hash map with a fixed-size space. Each candidate set is independently allocated with the size of the maximum degree. In this way, we can keep a controlled memory consumption.

7 Evaluation

In this section, we evaluate the effectiveness of the set dataflow and present the efficiency of Cyclosa.

7.1 Methodology

Patterns and Graph Datasets. The real-world graph datasets in our experiments are shown in Table 1, which are from the Stanford SNAP collection of datasets [29]. They represent typical graphs from different domains and are widely used in previous works [21–23]. The graph pattern mining algorithms evaluated are classified into two categories: 1) single pattern query that includes mining the single non-clique patterns (SM) [24] and *cliques finding* (CF) [25], 2) multiple patterns query that includes mining all patterns with a certain number of vertices, i.e., counting *k-motifs* (*k*-MC) [30] and multiple patterns satisfying specific graph property like *pseudo cliques* (PC) which are constrained with the given density [31]. These applications cover different kinds of representatives of graph pattern mining algorithms in prior works [20–23, 32].

Baseline Systems. Cyclosa is compared with two state-of-the-art pattern-centric systems, GraphPi [22] and SumPA [23].

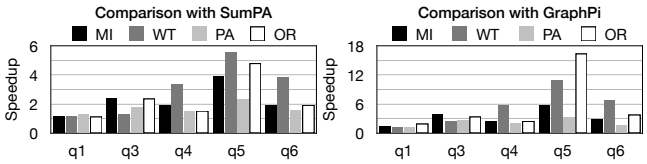


Figure 13: Performance comparison on listing single patterns

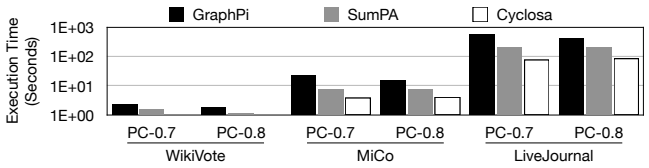


Figure 14: Performance on finding pseudo cliques

Table 2: Execution Time (Seconds) of Clique Finding

Systems	App.	MiCo	Patents	LiveJournal
Cyclosa	4-CF	0.23	0.11	2.27
	5-CF	21.82	0.29	463.02
SumPA	4-CF	1.21	0.37	5.98
	5-CF	50.77	0.45	1182.33
GraphPi	4-CF	1.64	0.44	12.14
	5-CF	60.51	0.52	1625.47

Table 3: Execution Time (Seconds) of Motifs Counting

Systems	App.	WikiVote	MiCo	Patents
Cyclosa	4-MC	1.25	9.56	6.81
	5-MC	257.34	1211.7	336.57
SumPA	4-MC	1.67	14.18	9.34
	5-MC	492.31	4789.5	662.03
GraphPi	4-MC	2.33	17.83	12.99
	5-MC	694.74	5803.97	803.55

Both systems are designed based on the mining procedure of Figure 1. GraphPi is a single pattern matching system that preserves the highest efficiency by finding an optimal combination of matching orders and symmetry constraints for an arbitrary pattern. SumPA is most related to our work and achieves higher performance on multiple patterns than prior works through a novel pattern abstraction.

Hardware Environments. All the experiments are conducted on a single server which is equipped with two 14-core Intel Xeon E5-2680v4 processors, 256GB RAM, and 512GB SSD. It runs a 64-bit Ubuntu 18.04 with kernel 5.4. We use gcc 7.3.0 to compile the applications with optimization under -O3. We use all the physical cores, and hyper-threading is enabled when the threads number exceeds 28.

7.2 Performance Comparison

Single Pattern Query. We first compare the performance of matching single non-clique patterns in Figure 3. These patterns are widely used in prior works [20, 21, 23]. Figure 13 shows the normalized speedups. Compared with GraphPi, Cyclosa achieves a speedup from 1.19× to 16.28×. The lowest speedup is for mining $q1$. Cyclosa can not only eliminate the explicit redundancies for $q1$ but also provide an efficient data graph layout. The highest speedup comes from $q5$ on Orkut. There are more implicit redundancies in $q5$ that cannot be removed by GraphPi, which can be explored in Cyclosa. Compared with SumPA, Cyclosa achieves a speedup from 1.13× to 5.52×. The pattern abstraction of SumPA can only expose parts of the explicit redundancies, while the set dataflow and smart cache in Cyclosa provide more opportunities for handling both explicit and implicit redundancies.

Table 2 shows the execution time for counting size-4 and size-5 cliques on different graphs. Overall, Cyclosa outperforms GraphPi by up to 7.13× and SumPA by up to 5.26× (4-CF on MiCo). GraphPi performs the lowest in all cases because all vertices in a clique are symmetric to each other, and the optimal matching order is unique. SumPA has limited improvement over GraphPi since the pattern abstraction will select a large sub-clique and omit opportunities for data reusing inside. In Cyclosa, the set dataflow can fully exploit the operands level redundancy for clique vertices, and the total workloads are reduced by graph orientation.

Multi-Pattern Query. We compare the performance of Cyclosa with SumPA and GraphPi on motifs counting and pseudo cliques. To support multiple patterns, we add a merging phase in GraphPi as in Automine [17]. Table 3 compares the execution time of counting size-4 and size-5 motifs on different graphs. With a larger size, the number of patterns increases (6 in 4-MC and 21 in 5-MC). Cyclosa outperforms SumPA and GraphPi by up to 3.95× and 4.79× for 5-MC on MiCo, respectively. The average speedups of Cyclosa on 4-MC and 5-MC of all cases are 1.64× and 2.95×. Note that Cyclosa achieves higher speedup when the pattern size and number increase. Existing approaches based on structural equality will face too many divergent branches when processing in-equal parts of these patterns. The set dataflow execution in Cyclosa can explore the computation similarity of both equal and in-equal parts.

Pseudo cliques (PC-k) algorithm finds patterns with a density greater than k. Figure 14 shows the results for mining all pseudo cliques with vertices less than six [23]. Notice that Cyclosa is superior to GraphPi and SumPA by 4.01× $\sim$ 7.52× and 1.48× $\sim$ 2.63×, respectively. Pseudo cliques are denser patterns, where one vertex is usually involved in most computations for other vertices. The set dataflow of Cyclosa can capture this similarity and fully reuse these operands as described in Section 2.4. Besides, the data management substrate can efficiently share results among different patterns.

Note that different mining algorithms can benefit from Cyclosa since the redundancies are highly related to the pattern topology and are independent of algorithm types. For larger patterns, more static similarity opportunities can be exploited. Also, an increased number of total computations amplifies the

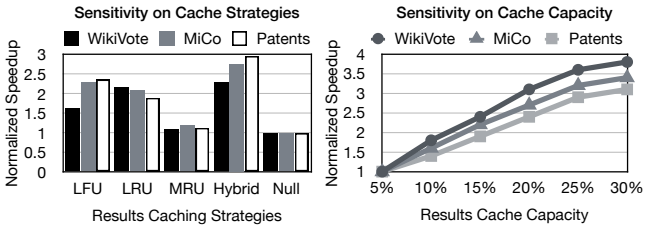


Figure 15: Normalized speedups under various settings of the cache capacity and caching strategy for 4-MC

Table 4: Execution Time (Seconds) on the Large Graph (FR)

App.	Cyclosa	SumPA	GraphPi
3-MC	94.26	143.38	281.42
4-CF	512.97	1491.72	1647.55

reusable computation amount arising from dynamic similarity. However, the size of set dataflow may increase and require more local memory space for parallel processing.

7.3 Sensitivity Study

Cache Strategy. The left chart in Figure 15 investigates the behaviors of different caching strategies for 4-MC. We fix the buffer capacity as 15% size of corresponding graphs. Notice that no single strategy can outperform others in all cases. Among all graphs, the LFU behaves better on Patents graph (2.36×), and the LRU behaves better on the WikiVote (2.17×). This is due to the diverse sparsity of the graphs, and the Patents graph is sparser than WikiVote. The MRU strategy is slowest in all cases, and this is because the results of high-degree vertices in the prior phase will be discarded even though they may be frequently reused. The hybrid strategy can combine the pattern and graph information to select the best buffer and thus achieves the highest performance.

Cache Capacity. The right chart in Figure 15 shows the normalized speedups with various buffer sizes. Initial buffer size is defined as the 5% size of a given data graph. Typically, larger cache capacity brings higher performance gains, e.g., 3.1× improvement from 5% to 20% on WikiVote, because more results can be cached and reused. However, the growth slows down gradually while increasing the buffer size. The main reason is that a larger buffer size induces higher latency in maintaining and querying the results.

7.4 Scalability

Figure 16 compares the performance of different systems by varying the number of threads. The results are normalized to GraphPi with a single thread. Hyperthreading is enabled when the number of threads exceeds 28. For GraphPi and SumPA, each thread is saturated with computations. More threads offer more compute parallelism but have to compete for computation resources. Thus, hyperthreading improves efficiency,

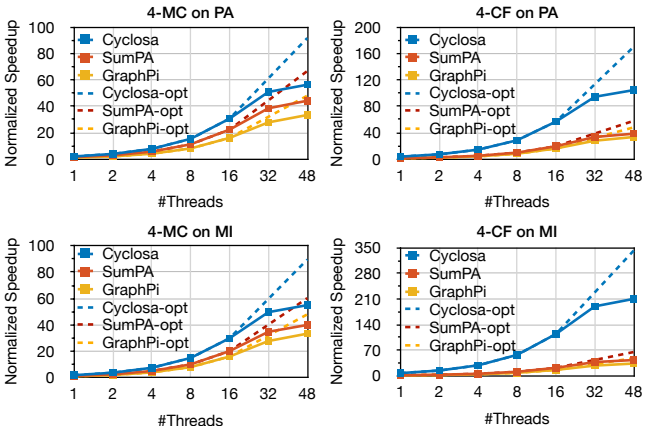


Figure 16: The normalized speedups with various number of threads for different applications

Table 5: Memory Consumption on Orkut with 28 Threads

Systems	4-CF	4-MC	5-MC	PC-0.8
Cyclosa	2.69GB	2.75GB	2.96GB	2.94GB
GraphPi	3.81GB	3.83GB	4.16GB	3.97GB

but the growth slows down gradually. Cyclosa resolves the redundant computation bottleneck, therefore offering more speedups against the above earlier systems. However, memory access contention becomes important when hyperthreading is used, incurring the increasingly-saturated performance improvement as shown in Figure 16. This is because Cyclosa has to maintain and query the results cache.

We also test the ability of Cyclosa to scale to large graphs with billion edges, as shown in Table 4. For size-3 motifs counting, Cyclosa gains 1.52× and 2.99× speedups over SumPA and GraphPi. Cyclosa outperforms SumPA and GraphPi on the Friendster graph for size-4 clique by 2.91× and 3.21×, respectively. Caching the results for large graphs is difficult because of the vast amount of intermediates. The performance improvement proves the efficiency of the set dataflow and caching strategies on larger graphs.

7.5 Overhead

Memory Consumption. Table 5 compares the memory consumption of Cyclosa and GraphPi on the Orkut graph. Cyclosa and GraphPi take an average of 2.84GB and 3.94GB of memory space in these cases. Although Cyclosa needs to maintain some of the results, the memory footprint is still kept small. This is because GraphPi maintains intermediate subgraph instances while Cyclosa shares the same result set for different vertices and executes the dataflow in a DFS style. Each thread in Cyclosa only maintains a local space for each set dataflow node. Besides, the smart caching strategies explore the trade-off between the space efficiency and reuse possibility. The memory space in the data management substrate is

Table 6: Time for Constructing Set Dataflow

App.	WikiVote	Patents	Orkut
5-CF	1.74ms	1.72ms	1.69ms
5-MC	9.32ms	9.44ms	9.37ms
PC-0.8	3.16ms	3.09ms	3.22ms

pre-allocated by a small fixed size.

Dataflow Construction. Table 6 presents the dataflow constructing time for different applications. This includes pattern analysis to get matching order, building the set dataflow, and evaluating redundancy probability. The triangle counting time is excluded because it is only conducted once and can be reused for different algorithms. Notice that the constructing time rises when the number of patterns increases. However, the dataflow construction is only executed once throughout processing and is negligible compared with the computation time. For example, it takes over 250s for 5-MC on WikiVote, while the dataflow construction only takes 9.32ms.

8 Related Work

Early graph pattern mining research focuses on customized improvements for specific given algorithms. For counting cliques, kClist [33] designs an efficient algorithm for processing sparse graphs based on the core value. To handle motifs effectively, G-tries [34] creates a novel tree-like data structure. PGD [26] counts all size-3 and size-4 motifs using partial patterns by some combinatorial rules. There have also been works that use GPUs to speed the subgraph isomorphism problem in finding network motifs and enumerating subgraphs [18, 35–37]. Cyclosa, as opposed to algorithm-specific improvements, focuses on tackling the common redundancy challenges in a more general situation. Prior optimizations can also be integrated into Cyclosa.

General-purpose graph pattern mining systems use expressive and efficient programming paradigms to automatically parallelize a variety of graph mining applications in a consistent manner [16, 38]. Early distributed systems, such as Arabesque [16] and Fractal [19], adopt the embedding-centric model to iteratively extend and enumerate all subgraphs size by size and verify the user-defined constraints for each intermediate embedding. RStream [39] and Kaleido [40] optimize the embedding-centric model in an out-of-core manner on a single machine to alleviate the data shuffling and communication cost. Pangolin [41] and Sandslash [42] provide flexible interfaces that integrate customized algorithmic optimizations to enhance the filtering of intermediate embeddings. Despite the expressiveness and massive parallelisms of these systems, managing a large number of intermediate embeddings becomes the main bottleneck, which suffers from high memory consumption and heavy I/Os [15].

Compared to the embedding-centric model, Cyclosa works in a pattern-aware manner, pioneered by AutoMine [17] and Peregrine [20], to avoid unnecessary storage and process of embeddings under the guide of pattern constraints. The cores of these systems are efficient matching engines that execute fast set operations [27]. AutoMine is a compilation-based system that automatically transforms graph patterns into set programs. GraphZero [21] is an enhanced version of AutoMine that removes explicit redundancies among multiple patterns by introducing the symmetry-breaking optimizations that define a partial order between symmetric vertices. GraphPi [22] further explores the optimal combination of matching orders and symmetry-breaking constraints to speed set programs. Despite the efficiency, the compilation method may induce non-negligible overhead while generating new set programs for newly coming patterns. Dryadic [43] proposes a flexible tree-structured intermediate representation to solve this problem, which supports both compilation and runtime optimizations. DecoMine [28] decomposes a large pattern into smaller ones for faster pattern counting. SumPA [23] merges multiple patterns in a pattern abstraction to minimize workloads. Cyclosa aims to eliminate the inherent computation redundancies, and the data management substrate can also be integrated into the above systems to reduce redundancies.

9 Conclusion

In this work, we present a redundancy-free framework, Cyclosa, that eliminates both explicit and implicit redundancies in pattern-centric graph mining systems. Cyclosa explores the computation similarity through a novel set dataflow representation, which exploits a finer-grained similarity at the operand level instead of the structural equality as in existing works, making it possible to indicate both explicit and implicit redundant computations. Based on the set dataflow, Cyclosa implements an efficient dataflow-guided execution model collaborated with a memory-friendly data management substrate to efficiently reuse computing results, embracing high performance and correctness. The proposed substrate may also be incorporated into the runtime of existing systems to reduce redundancies. Evaluation of a variety of patterns and real-world graphs shows that Cyclosa can significantly outperform state-of-the-art systems GraphPi by up to 16.28× and SumPA by up to 5.52× for various applications.

Acknowledgments

We thank our anonymous reviewers and shepherd for their insightful suggestions. This work is supported by the National Key Research and Development Program of China under (Grant No. 2022YFB4501403), the NSFC (Grant No. 61832006 and 61929103), the Zhejiang Lab (Grant No. 2022PI0AC02), and the Fundamental Research Funds for the Central Universities (Grant No. YCJJ202202011). Long Zheng is the corresponding author.

References

[1] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference*, pages 49–60, 2013.

[2] Vasileios Trigonakis, Jean-Pierre Lozi, Tomás Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost depth-first-search distributed graph-querying system. In *Proceedings of the USENIX Annual Technical Conference*, pages 209–224, 2021.

[3] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[4] Dan Chen, Chuangyi Gui, Yi Zhang, Hai Jin, Long Zheng, Yu Huang, and Xiaofei Liao. Graphfly: Efficient asynchronous streaming graphs processing via dependency-flow. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2022.

[5] Yuhang Wang, Fillia Makedon, James Ford, and Heng Huang. A bipartite graph matching framework for finding correspondences between structural elements in two proteins. In *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 2972–2975, 2004.

[6] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the ACM International Conference on Management of Data*, pages 1447–1462, 2019.

[7] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. RapidMatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2020.

[8] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Parallel k-clique counting on GPUs. In *Proceedings of the ACM International Conference on Supercomputing*, pages 1–14, 2022.

[9] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. In *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms*, pages 135–146, 2021.

[10] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys*, 54(2):28:1–28:36, 2021.

[11] Lorenzo De Stefani, Erisa Terolli, and Eli Upfal. Tiered sampling: An efficient method for counting sparse motifs in massive graph streams. *ACM Transactions on Knowledge Discovery from Data*, 15(5):79:1–79:52, 2021.

[12] Nishil Talati, Haojie Ye, Sanketh Vedula, Kuan-Yu Chen, Yuhan Chen, Daniel Liu, Yichao Yuan, David Blaauw, Alex Bronstein, Trevor Mudge, and Ronald Dreslinski. Mint: An accelerator for mining temporal motifs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 1270–1287, 2022.

[13] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727, 2016.

[14] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An efficient task-oriented graph mining system. In *Proceedings of the ACM European Conference on Computer Systems*, pages 1–12, 2018.

[15] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. A locality-aware energy-efficient accelerator for graph mining applications. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 895–907, 2020.

[16] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 425–440, 2015.

[17] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 509–523, 2019.

[18] Xuhao Chen and Arvind. Efficient and scalable graph pattern mining on GPUs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 857–877, 2022.

[19] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1357–1374, 2019.

[20] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the ACM European Conference on Computer Systems*, pages 1–16, 2020.

[21] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. GraphZero: A high-performance subgraph matching system. *ACM SIGOPS Operating Systems Review*, 55(1):21–37, 2021.

[22] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. GraphPi: High performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[23] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. SumPA: Efficient pattern-centric graph mining with pattern abstraction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 318–330, 2021.

[24] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. HUGE: An efficient and scalable subgraph enumeration system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2049–2062, 2021.

[25] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. Lightning fast and space efficient k-clique counting. In *Proceedings of the ACM Web Conference*, page 1191–1202, 2022.

[26] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *Proceedings of the IEEE International Conference on Data Mining*, pages 1–10, 2015.

[27] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. GraphMineSuite: Enabling high-performance and programmable graph mining algorithms with set algebra. *Proceedings of the VLDB Endowment*, 14(11):1922–1935, 2021.

[28] Jingji Chen and Xuehai Qian. DecoMine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 47–61, 2022.

[29] Jure Leskovec and Andrej Krevl. SNAP datasets: Stanford large network dataset collection. 2014. http://snap.stanford.edu/data.

[30] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. LINC: A motif counting algorithm for uncertain graphs. *Proceedings of the VLDB Endowment*, 13(2):155–168, 2019.

[31] Takeaki Uno. An efficient algorithm for enumerating pseudo cliques. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 402–414, 2007.

[32] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, Wei-Shinn Ku, , and John C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1369–1380, 2020.

[33] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *Proceedings of the International World Wide Web Conference*, pages 589–598, 2018.

[34] Pedro Ribeiro and Fernando Silva. G-Tries: An efficient data structure for discovering network motifs. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1559–1566, 2010.

[35] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiaoli Li. Network motif discovery: A GPU approach. *IEEE Transactions on Knowledge and Data Engineering*, 29(3):513–528, 2017.

[36] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. GSI: GPU-friendly subgraph isomorphism. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1249–1260, 2020.

[37] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. GPU-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1067–1082, 2020.

[38] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. Tesseract: Distributed, general graph pattern mining on evolving

graphs. In *Proceedings of the European Conference on Computer Systems*, pages 458–473, 2021.

[39] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 763–782, 2018.

[40] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. Kaleido: An efficient out-of-core graph mining system on a single machine. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 673–684, 2020.

[41] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proceedings of the VLDB Endowment*, 13(10):1190–1205, 2020.

[42] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: A two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*, pages 378–391, 2021.

[43] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 289–303, 2021.