GraphZero: A High-Performance Subgraph Matching System

Daniel Mawhirter†* *Sam Reinehr*†* *Connor Holmes*† *Tongping Liu*§ *Bo Wu*†
Colorado School of Mines† *University of Massachusetts at Amherst*§

Abstract

Subgraph matching is a fundamental task in many applications which identifies all the embeddings of a query pattern in an input graph. Compilation-based subgraph matching systems generate specialized implementations for the provided patterns and often substantially outperform other systems. However, the generated code causes significant computation redundancy and the compilation process incurs too much overhead to be used online, both due to the inherent symmetry in the structure of the query pattern.

In this paper, we propose an optimizing query compiler, named GraphZero, to completely address these limitations through symmetry breaking based on group theory. GraphZero implements three novel techniques. First, its schedule explorer efficiently prunes the schedule space without missing any high-performance schedule. Second, it automatically generates and enforces a set of restrictions to eliminate computation redundancy. Third, it generalizes orientation, a surprisingly effective optimization that was only used for clique patterns, to apply to arbitrary patterns. Evaluation on multiple query patterns shows that GraphZero outperforms two state-of-the-art compilation and non-compilation based systems by up to 40X and 2654X, respectively.

1 Introduction

Subgraph matching identifies all embedddings of specific query patterns in input graphs, which is important for bioinformatics [22, 55], social networks [39, 56], and fraud detection [5, 12]. Triangle counting represents a simple subgraph matching task, where the pattern is a triangle and the goal is to count all its instances in an input graph. While triangle counting is much more costly to solve than many graph traversal problems, matching larger patterns (e.g., size-7 cliques) involves even more complex algorithms and may need hours or days to finish.

Subgraph matching problems have attracted significant attention in the data analytics community. A common approach is to design efficient algorithms for individual patterns [4, 13, 60]. IEEE, Amazon and MIT organize an annual challenge to rank the submitted implementations for triangle counting [1]. However, this approach is not scalable for general subgraph matching, as the number of patterns increases exponentially in the pattern size. For example, there exist 112 size-6 connected patterns but 853 size-7 connected patterns.

Many software systems have been proposed to perform subgraph matching for arbitrary patterns [6, 8, 11, 19, 41, 50, 53, 59]. Despite their generality, these systems leave much performance on the table due to two reasons. First, they implement a generic algorithm that handles arbitrary patterns but does not perform particularly well for any of them. It is why manual implementations of specialized algorithms for specific patterns are still popular. Second, some of them need to run isomorphism check online to verify whether a produced embedding uniquely matches the pattern of interest, which may incur substantial overhead for non-trivial patterns.

The compilation approach for subgraph matching has advantages as shown in systems like EmptyHeaded [3] and AutoMine [37]. They generate specialized algorithms for the given pattern and compile it to efficient low-level code. The code has a nested-loop structure, each level growing the embedding by one vertex towards the pattern through set operations on the neighbor lists (e.g, set intersection). If an embedding is identified by the innermost loop, it guarantees to match the pattern and avoids online isomorphism test. Several non-compilation based systems, represented by RADS [19] and GraphFlow [26], find an optimized matching order, which leverages the given pattern to improve performance in a similar manner.

While compilation-based subgraph matching systems often substantially outperform other systems [3, 37], they have three problems. The first problem is that their generated code incurs significant computation redundancy due to identifying the same embedding multiple times. For a size-7 near-clique, which is a clique with one absent edge, they identify each

[*] equal contribution

embedding 120 times. The second problem is their slow compilation speed. For example , AutoMine uses a brute-force approach to enumerate all possible schedules (i.e., vertex matching orders). As a result, its search algorithm may unnecessarily traverse schedules that generate the same code. If used as an offline compiler, the compilation overhead may be acceptable especially because most real-world applications only match small patterns. But it may be problematic if AutoMine is used as a just-in-time compiler for dynamic queries where the overhead lies on the critical path. Finally, these systems can only apply the orientation optimization [51] to clique patterns by enforcing edge traversals from lower-degree vertices to higher-degree vertices. Although this optimization alone can produce up to tens of times performance improvement for triangle counting [24, 47], most patterns can not enjoy this benefit.

Addressing the three problems faces substantial challenges. First, patterns and their inner structures have symmetry. Consider the rectangle pattern. All four vertices are initially equivalent due to the four-way rotational symmetry. After fixing one vertex, there is still a two-way mirrored symmetry between the two vertices attached to the first, leading to a total multiplicity of 8. The symmetry leads to different ways to map the query pattern to the vertices in the graph. Although symmetry breaking has been used in prior systems [18, 19], it is unclear how to completely eliminate symmetry-related redundancy correctly and efficiently in a compilation-based system. Second, the schedule space can be enormous, and different schedules may produce programs that differ dramatically in performance. However, pruning the schedule space naively (e.g., setting an upper bound on the number of explored schedules) may miss high-performance schedules. Third, the orientation optimization can be applied to clique patterns because they are perfectly symmetric, meaning that any pair of vertices can be exchanged yet still preserving the pattern. Most patterns do not have this property, so it is unclear whether they can also benefit from this optimization.

In this paper, we present GraphZero, an optimizing query compiler for subgraph matching that systematically addresses these challenges to enable zero redundancy in both compilation and execution. GraphZero contains three key components: a schedule explorer, a redundancy optimizer, and an orientation optimizer. Given the patterns of interest, the schedule explorer only searches for schedules of potentially different performance. The redundancy optimizer automatically generates a set of restrictions for the found schedule and enforces the restrictions on it to completely eliminate computation redundancy. The orientation optimizer successfully generalizes the application of the orientation optimization to arbitrary patterns.

The fundamental idea for GraphZero to break symmetry is to use more than the topology information, such as vertex ID or degree. We observe that the differences between different ways to identify the same embedding are essentially automorphisms, which are isomorphisms from the embedding to itself. The number of automorphisms determines the degree of redundancy. By enforcing a set of automatically generated restrictions between discovered vertices, GraphZero reduces the number of automorphisms to one. Consequently, the generated code identifies each instance exactly once. It is worth mentioning that different from existing symmetry breaking techniques [18], GraphZero enforces a high-performance schedule and at most one restriction for each matched vertex. Moreover, GraphZero uses automorphisms to prune the schedule space and only discovers one of the equivalent schedules that correspond to the same automorphism group. It still finds the optimal schedule because all the equivalent schedules order the set operations in the same way and hence have the same performance. Finally, GraphZero enjoys a byproduct of the enforced restrictions, which is that for each restriction it can also restrict an edge traversal order that generalizes the orientation optimization.

We have extensively evaluated all the three components of GraphZero by comparing it with two state-of-the-art subgraph matching systems. Our experimental results demonstrate performance improvements of up 40X for single pattern matching over AutoMine as well as up to 2654X improvement over RADS [19]. GraphZero generates high performance schedules up to 197X faster than AutoMine for multiple complex patterns. Moreover, GraphZero's generalized orientation optimization produces up to 88.6X performance improvement over the unoriented version.

We make the following contributions in this paper: 1) We reveal that the inherent symmetry in graph patterns is a fundamental reason for computation redundancy and the slow compilation speed of existing compilation-based subgraph matching systems; 2) We propose to use group theory to break symmetry through automatically generated and enforced restrictions, which not only completely eliminates redundancy in the generated code but also substantially prunes the schedule search space; 3) We generalize the orientation optimization to arbitrary patterns by leveraging the generated restrictions; 4) We present an optimizing compiler that integrates the proposed techniques to substantially outperform two state-of-the-art systems with fast compilation.

2 Background and Motivation

2.1 Definition of Subgraph Matching

A subgraph $S = (V_S, E_S)$ of an input graph $G = (V_G, E_G)$ is vertex-induced if $V_S \subset V_G$ and E_S contains all edges in E_G where the source and destination vertices are in V_S. A vertex-induced subgraph S matches the query graph Q if S is isomorphic to Q. Subgraph matching, as explored in this paper, discovers *all distinct* subgraphs that match the query graph. The query graphs are assumed to be connected and undirected with or without vertex labels. The input graph is directed or undirected with or without vertex labels.

2.2 Basics of Compilation-Based Subgraph Matching Systems

Figure 1 (a) demonstrates the basic workflow of compilation-based subgraph matching systems. Given the pattern of interest, they find a matching order of its vertices, making sure that any vertex except the first one is connected to at least one vertex matched before it. A schedule can then be naturally generated following this order with present edges encoded by set intersections and absent edges by set differences. For example, D is connected to both B and C but not A. Thus, D should be in the intersection of B's and C's neighborhood but not in A's neighborhood. Finally, these systems compile the schedule into a nested loop structure, the inner-most loop of which identifies the embeddings of the pattern. Since the matching order determines a schedule, we use a total order on the label set to represent the schedule. For example, the schedule shown in Figure 1 (a) can be represented by $[A,B,C,D]$. This approach readily generalizes to any query pattern, as long as the compilation time is reasonable.

2.3 Problems and Challenges

2.3.1 Computation redundancy

Every subgraph matching system needs a way to test isomorphism, that is, determine what pattern a particular subgraph matches. Compilation-based systems embed the isomorphism test in the generated code itself, using the nested loop structure to filter embeddings down to those that match the desired structural pattern. Doing so discovers the vertices in a particular order, what we call a *mapping* from the pattern's labeled vertices to the embedded vertices in the graph. However, the label set may be mapped to the vertices of the same embedding in different ways. Figure 1 (b) shows that there exists eight different mappings for a rectangle instance, leading to eight times over-counting (also called multiplicity) and computation redundancy.

AutoMine partially solves the problem by introducing the concept of root symmetry. It checks whether a schedule's first two vertices are equivalent and if so only processes that first “root" edge in one direction. For many patterns, this successfully cuts the computation redundancy in half. However, as Figure 2 shows, the number of possible mappings explodes with the number of vertices in the pattern. Ideally, the system should only identify each instance exactly once and so completely eliminate computation redundancy.

Challenges. The scheduling approach of existing compilation-based systems is fundamentally unequipped to handle this multiplicity problem. All the mappings correspond to the same topology and cannot be differentiated by only set operations. As such, the symmetries in a pattern induce the same multiplicity in any possible schedule for that pattern. The symmetry may express itself in different ways in different schedules, as the vertices that can satisfy a particular mapping depend on both the schedule and the previously explored vertices. This diversity is a key reason the problem is both challenging to handle and valuable to solve. Our approach must find all the symmetries and a way to break them to make the mapping unique.

2.3.2 Brute-force Schedule Generation

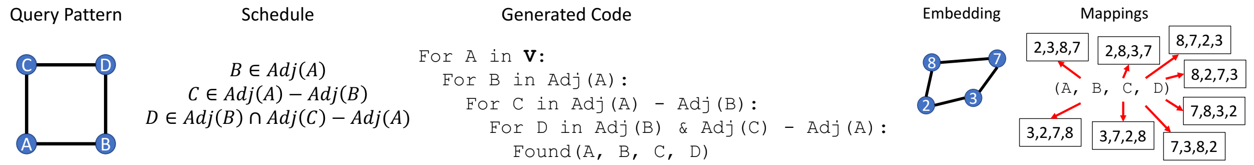
Existing systems, such as AutoMine and GraphFlow, use a naive approach to schedule generation. It first enumerates all possible schedules and uses a performance model to rank them. This process is inefficient because many schedules correspond to the same code and thus have the same performance. AutoMine takes over 147 seconds to generate schedules for 7-vertex motif counting. When used as a just-in-time compiler, this latency can be problematic because the schedule search overhead lies on the critical path.

Challenges. The compilation process introduces a chicken-and-egg problem: we want to avoid an exhaustive search of the scheduling space while still identifying a high-performance schedule, which is however unknown until after exploring the space. We can run isomorphism test to filter out schedules that produce the same code, but the testing itself involves an exponential algorithm. Moreover, estimating the performance of schedules is more complex when computation redundancy is eliminated.

2.3.3 Generalization of Orientation Optimization

Orientation is a popular optimization for triangle counting [24, 47, 51]. By only allowing higher-degree vertices to be discovered after lower-degree vertices to form triangles, the optimization effectively prunes half of the edges (i.e., the ones from higher-degree vertices to lower-degree vertices) but still preserves the structure of all triangle embeddings. However, it can achieve significant speedups much larger than 2X because the time complexity of triangle counting grows super-linearly with the maximum degree. Existing systems can apply orientation to clique patterns thanks to the perfect symmetry. However, most patterns cannot enjoy the benefit from the orientation optimization in these systems.

Challenges. Generalizing the orientation optimization to cover arbitrary patterns is difficult due to their arbitrary topology. Enforcing an order for vertex discovery based on degree conceptually prunes edges from the graph data. Doing so may miss some embeddings of interest when it is necessary to discover a vertex from a higher-degree vertex due to the chosen schedule but the edge is not present. While manual application of the optimization for specific patterns is possible, automating it in a compiler demands a systematic approach.



```
For A in V:
  For B in Adj(A):
    For C in Adj(A) - Adj(B):
      For D in Adj(B) & Adj(C) - Adj(A):
        Found(A, B, C, D)
```

(a) Compilation Phase

(b) Matching Phase

Figure 1: The Workflow of Compilation-Based Subgraph Matching Systems.

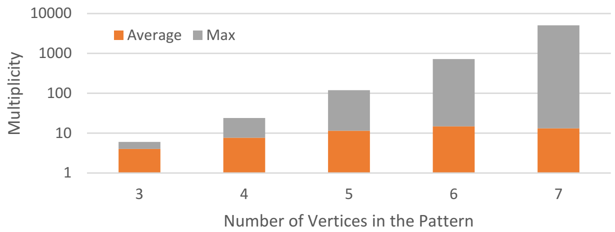


Figure 2: The Computation Redundancy Explosion Problem

3 Overview of GraphZero

The overall workflow of GraphZero has four stages as shown in Figure 3. Taking a pattern as the input, the schedule explorer searches only part of the schedule space while guaranteeing that the optimal schedule is discovered. Based on a performance model, the schedule explorer estimates the performance of the searched schedules and produces a high-performance schedule for the redundancy optimizer. The redundancy optimizer then generates a set of restrictions and enforces them on the schedule to perform redundancy-free computation. The orientation optimizer, which is optional, reindexes the vertices in the input graph based on decreasing degree and leverages the generated restrictions to conceptually prune the corresponding edges from higher-degree to lower-degree vertices while keeping the entire graph. Finally, the optimized schedule can run on the reindexed graph to match the input pattern. Although the discussion focuses on one pattern for simplicity, if provided multiple patterns GraphZero merges the schedules to only generate one matching program.

4 Redundancy-Free Code Generation

In this section, we first explain the essential reason for computation redundancy and build the connection between redundancy and automorphisms, which is the key concept from group theory to address the problem. We next describe the approach to completely eliminating computation redundancy through automatically generated and enforced restrictions, followed by a method to minimize the overhead to implement the restrictions in the generated schedule. Finally, we show what the output looks like with and without these restrictions.

4.1 Computation Redundancy and Automorphism

The computation redundancy problem caused by over-counting is rooted in the symmetry in the pattern. For example, given an embedding of the rectangle pattern, each of the four vertices in the embedding can be indistinguishably mapped to the first label in the schedule. Once the first vertex in the embedding is mapped, both its neighbor vertices can be mapped to the second label of the schedule. The multiplicative mapping choices result in the eight times over-counting. Note that selecting a better schedule does not solve the problem because the symmetry cannot be broken by only set operations.

A mapping represents a one-to-one relationship between the labels in the pattern and the vertices in the embedding. If we give a total ordering to the embedding vertices, each mapping can be represented by a total ordering of the labels. Consider the example in Figure 1. We have $[A,B,C,D]$ to denote the schedule, so a total ordering of the embedding vertices can be $[2,3,8,7]$. The mapping from $[A,B,C,D]$ to $[2,8,3,7]$ can then be represented by the total ordering of the labels: $[A,C,B,D]$. We call a total ordering of the labels that corresponds to a mapping a valid ordering. Not all total orderings correspond to a mapping. For instance, $[A,C,D,B]$ is not a valid ordering because C and D are connected in the pattern, but the embedding vertices 3 and 8 are not.

Each valid ordering is essentially a permutation of the total ordering representing the schedule [1]. Each such permutation function is called an automorphism in group theory, which is an isomorphism from the pattern to itself and hence preserves the structure of the pattern. If an automorphism repositions a label A to the original position of another label B (e.g., $[A,B,C,D] \to [B,A,C,D]$), we say that the automorphism moves A to B for simplicity. All the automorphisms for the same pattern form an automorphism group. The size of the automorphism group determines the number of times for over-counting. Ideally, we want to reduce the size of the group to one to completely eliminate computation redundancy.

[1] A total ordering is a set plus the relation on the set, but we treat it as a label array to simplify the discussion

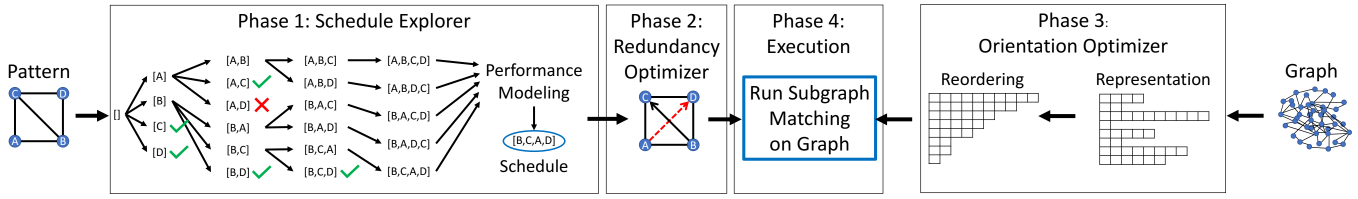


Figure 3: Overview of GraphZero

4.2 Breaking Symmetry through Restrictions

The key idea is that we can enforce restrictions on the IDs of the embedding vertices to break the symmetry inherent in the pattern. Vertices are not reused within the same embedding, so there is a natural total ordering on the IDs. Consider the rectangle pattern in Figure 1 again. We have observed that the first label in the schedule can be mapped to any of the four embedding vertices. The restriction we enforce is that the first mapped vertex should have the highest ID among the four embedding vertices. Doing this effectively reduces the over-counting by four times because only two mappings, namely $[A,B,C,D] \rightarrow [8,7,2,3]$ and $[A,B,C,D] \rightarrow [8,2,7,3]$ respect the restrictions. We then enforce another restriction that the ID of the second mapped vertex should be larger than that of the third mapped vertex. The end result is that the only way to identify the embedding is through the mapping $[A,B,C,D] \rightarrow [8,7,2,3]$, which completely solves the over-counting problem.

We make two observations on the example. First, the restrictions are each a transitive binary relation (i.e., larger than) between the IDs of two embedding vertices. Second, we can use a binary ordering relation on the labeled vertices of the pattern to represent the restriction on the embedding vertices. As such, the partial ordering $\{(A,B),(A,C),(A,D),$ $(B,C)\}$ should be sufficient to generate all restrictions.

Algorithm 1 generalizes the idea and generates such a partial ordering for an arbitrary pattern with a given schedule. The algorithm at the beginning computes the automorphism group by trying all permutations of the vertex labels and filtering out invalid ones, which do not preserve the pattern. It then iterates through all labeled vertices in the schedule. In each iteration, it determines all the labeled vertices indistinguishable from the traversed vertex v after prior partial ordering. For each of of these labeled vertices $x(v)$, which the automorphism x maps v to, the algorithm adds a binary relation $(v,x(v))$ to the resultant partial ordering. At the end of the for loop, only automorphisms that do not move the traversed vertices are used to generate binary relations for the next traversed vertex.

The following theorem allows us to use these relations to uniquely discover each embedding exactly once.

Algorithm 1: Restriction generation algorithm

```
input : $P$ : the pattern.
input : $S$ : the schedule.
output : $L$ : a partial ordering on the labels
begin
    $Aut \leftarrow$ all the automorphisms of $P$
    $L \leftarrow$ an empty partial ordering.
    // iterate through these in order,
        $S[0], S[1], \cdots$
    for $v$ in $S$ do
        // stabilized_aut contains all
            automorphisms that do not move
            the vertices we have iterated
            over
        $stabilized\_aut \leftarrow []$
        for $x \in Aut$ do
            if $x(v) = v$ then
                // x does not move v
                add $x$ to $stabilized\_aut$
            else
                // $x$ moves v and indicates a
                    binary ordering relation
                add $(v, x(v))$ to L, if not present
        $Aut \leftarrow stabilized\_aut$
```

Theorem 1 *: For an arbitrary pattern P and its schedule S, let L be the partial ordering on the label set in P generated by*

Algorithm 1. Given an instance of P, denoted by e, in a graph G there exists exactly one mapping M from P to e if for each binary ordering relation $(S[i], S[j])$ in L, we add a restriction $M(S[i]).id > M(S[j]).id$.

This proof is split into two components. The first shows that at most one mapping exists that follows the ordering. The second shows that any instance that matches a mapping has at least one mapping that follows the restrictions.

To prove that at most one mapping exists, we perform a proof by contradiction. Assume two distinct mappings, M_1 and M_2. Then, M_1 and M_2 must differ in at least one vertex. Let the first vertex in which they differ be d. Then, there is an automorphism that does not moves any of the vertices of S before d, which maps M_1 to M_2. There is therefore an automorphism A such that $M_1(A(x)) = M_2(x)$. Note that $M_2(d) \neq M_1(d)$, so $M_2(d) = M_1(A(d)) < M_1(d)$ is a restriction that is enforced. However, note that symmetrically, we also have $M_1(d) < M_2(d)$, which is a contradiction, so there cannot be two distinct mappings that both obey the partial orderings.

To prove that there is at least one mapping that satisfies the restrictions imposed by the binary relations, we first present the following lemma, which shows an important property of these automorphisms that allow us to apply them without changing any relations.

Lemma 1 *Consider any automorphism A, which maps $S[i]$ to $S[i]$, for $i < k$. A does not move the set of binary relations $(S[a], S[b])$ where $a < k$, if we apply the automorphism to each binary relation (X,Y) to become $(A(X), A(Y))$, over all automorphisms. As a consequence, applying such an automorphism does not break the relations.*

First, for any $b < k$, the binary relations do not move, so they are automatically satisfied. Now, consider $(S[a], S[b])$ as a binary relation, where $b \geq k$. Then, there exists an automorphism, which does not move $S[j]$ for $j < a$ and maps $S[a]$ to $S[b]$. There is then an automorphism obtained by composition, which maps $S[a]$ to $A(S[b])$ but does not move $S[j]$ for $j < a$. $(S[a], A(S[b]))$ is thus a binary relation in the original set. Therefore, the set of binary relations after the automorphism is a subset of the binary relations before the automorphism. Then, symmetrically, since the automorphism is invertible with an automorphism, the set of relations is the same before and after the automorphism is applied.

That is, we can apply a sequence of N automorphisms, and make a corresponding sequence of labelings: $M_0, M_1, M_2, \cdots M_N$. Each member of these labelings will satisfy the following: M_j satisfies all relations of the form $(S[z], S[k])$, where $k < j$ and z can be any index. Note that we can allow M_0 to be M. We shall prove that we can construct this sequence iteratively.

We consider M_k and construct M_{k+1}. Let z be the index such that $M_k(S[z])$ is maximized, and where $(S[z], S[k])$ is a relation, or $z = k$. Note that if $z = k$, we can just take $M_{k+1} = M_k$, as M_k satisfies the restrictions because $M_k(S[z]) < M_k(S[k])$ for all z where $(S[z], S[k])$ is a relation we have. Otherwise, there is an automorphism A that does not move $M_k(S[x])$ for $x < k$, but moves $M_k(S[z])$ to $M_k(S[k])$. This is required for the restriction to exist.

We shall prove that letting $M_{k+1} = M_k A$ (i.e., $M_{k+1}(x) = M_k(A(x))$) satisfies the restrictions we desired. By the lemma, all relations $(S[a], S[b])$ where $b < k$ are still satisfied. To prove that M_{k+1} satisfies the restrictions of the form $(S[x], S[k])$. Note that $S[z] > S[d]$ for any d where there is an automorphism moving $S[k]$ to $S[d]$. For any x where such an automorphism exists, $(A(S[x]), S[k])$ is also a restriction because there is an automorphism moving $S[k]$ to $A(S[x])$. Therefore, $M_{k+1}(S[x]) = M_k(A(S[x])) < M_k(S[z]) = M_{k+1}(S[k])$. Thus, M_{k+1} satisfies the restrictions we desired. This completes the recursive step.

We can then construct M_N, which is a mapping that satisfies all the binary relations in the partial ordering. Therefore, we have proved that the algorithm will find every distinct mapping at least once and at most once, so it finds each one exactly once, as desired.

Algorithm 1 and Theorem 1 demonstrate the possibility to completely remove computation redundancy. Implementing the restrictions in the code generator is simple. The compiler needs to insert bound checks when generating a for loop as demonstrated in Figure 4 (a).

4.3 Minimizing the Overhead of Enforcing Restrictions

Figure 4 (a) shows that when discovering one vertex the code may need to perform multiple checks, which incurs non-trivial overhead especially for large patterns. We want to minimize the number of checks to increase performance. For example, within Figure 4 (a), the $v_0 < v_2$ check is made redundant by the $v_0 < v1$ check and the $v_1 < v_2$ check. These correspond to the (A,C), (A,B) and (B,C) relations generated by the schedule, respectively. The following theorem generalizes this process, and shows that we only need to generate one relation per matched vertex.

Theorem 2 *Given the set of binary relations generated by Algorithm 1, for each k $(0 \leq k < |S|)$ where S is the input schedule, we only keep $(S[z], S[k])$ where z is maximized for k, we still properly enforce all the original binary relations.*

We omit the complete proof but describe its basic idea based on proof by contradiction. Assume that we require two binary relations (X,Z) and (Y,Z) in the partial ordering without having (X,Y) or (Y,X). Due to Algorithm 1, we have two automorphisms mapping X and Y to Z, respectively, which implies that we must have an automorphism to map between X and Y. We therefore must have either (X,Y) or (Y,X) in the partial ordering, which contradicts the assumption.

We can utilize Theorem 2 to modify Algorithm 1 to store a map of chosen relations. Instead of adding $(v,x(v))$ as a relation, we set the value in the map for $x(v)$ to be v, representing that the only relation that needs to be checked for $x(v)$ is $(v,x(v))$. Figure 4 (b) shows the generated code for rectangle counting after applying this optimization.

5 Fast Schedule Generation

In this section, we first explain the reason schedules of the same pattern have dramatically different performance. We describe the schedule generation algorithm used in existing systems and show why it incurs tremendous overhead, followed by the algorithm used in GraphZero's schedule explorer to efficiently prune the search space. We finally present GraphZero's method of estimating performance of schedules bounded with restrictions.

5.1 Performance Differences

The performance of different schedules can vary significantly. To illustrate this, we compare two schedules for tailed triangle. As shown in Figure 5, the two schedules search for the vertices in $[A,B,C,D]$ and $[C,D,B,A]$ order according to the labels in the diagram. The key difference between these schedules is intuitive: the schedule $[C,D,B,A]$ executes its innermost loop once for each triangle embedding in the graph, while the schedule $[A,B,C,D]$ does it once for every wedge (i.e., a two-edge path). In some of the graphs we evaluate for this paper, wedges may appear over 500X more frequently than triangles. So if the amount of work done inside the innermost loop is comparable, the schedule $[C,D,B,A]$ should have much better performance.

5.2 Brute-Force Schedule Generation

To avoid missing high-performance schedules, existing compilation-based systems take a conservative approach and tests if each permutation of the labeled vertices is a valid schedule as shown in Algorithm 2. Recall that a schedule is considered valid if each vertex, except for the first, is directly connected to at least one vertex that comes before it. This allows it to be described as a member of a composition of set differences and intersections of the neighbor sets of previous vertices.

We can easily improve the schedule generation using Algorithm 3, which recursively searches only valid schedules. However, the algorithm is still inefficient for non-trivial patterns. To understand the complexity of traversing all valid schedules, we consider a path of length n. There are n starting vertices to choose from and picking any vertex except the two end vertices leaves the remaining task as finding the valid schedules for the two sub-paths on the two sides. We omit the proof but just show that there exist a total of 2^{n-2} valid schedules. In another example, a clique of n vertices, all $n!$ possible schedules are valid.

Algorithm 2: Automine's schedule generation

```
input  : P : the pattern.
output : schedules : the list of all valid schedules
begin
    for permutation S of the vertices in P do
        if S is a valid schedule then
            add S to schedules
```

Algorithm 3: Schedule generation algorithm - smart brute force

```
input  : P : the pattern.
output : schedules : list of all valid schedules for P
schedules ← []
call recursive_generate_0([])
Procedure recursive-generate_0
    input  : sched - a partially built schedule
    if sched covers all vertices in the pattern then
        add sched to schedules
    else
        if sched is empty then
            valid_next ← P's vertices
        else
            valid_next ← all vertices in P connected
              to sched but not in sched
        for Vertex v ∈ valid_next do
            call recursive_generate_0(sched+[v])
```

5.3 Pruning the Valid Schedule Space

The vast scheduling space described previously is expensive to search exhaustively, so we need an efficient method to prune the space without missing any potentially high-performance schedule. A key observation we have is based on the clique example. Although it has $n!$ schedules, all of them generate the same sequence of set intersection operations and hence generate the same code. We call such schedules equivalent schedules. If two schedules are not equivalent, we call them distinct schedules. In general, any two schedules are equivalent if and only if there is an automorphism that maps one to the other. As an example, $[A,B,C,D]$ and $[A,B,D,C]$ are equivalent schedules for the tailed triangle in Figure 5. Recall that the multiplicity, M, of all valid schedules is the same because it is an inherent property of the pattern. Therefore, if K schedules are valid for a pattern, there exist K/M distinct schedules. The goal of the schedule explorer is to only

```
input: $G$ the graph
output: $count$ the number of induced q2 in $G$
begin
  $count$ ← 0
  for $v_0 \in G$ do
    for $v_1 \in N(v_0)$ do
      if $v_0 < v_1$ then
        break
      for $v_2 \in N(v_0) - N(v_1)$ do
        if $v_0 < v_2$ then
          break
        if $v_1 < v_2$ then
          break
        for $v_3 \in N(v_1) \cap N(v_2) - N(v_0)$ do
          if $v_0 < v_3$ then
            break
          $count$ ← $count$ + 1
```

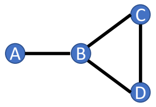
(a) Applying All Restrictions

```
input: $G$ the graph
output: $count$ the number of induced q2 in $G$
begin
  $count$ ← 0
  for $v_0 \in G$ do
    for $v_1 \in N(v_0)$ do
      if $v_0 < v_1$ then
        break
      for $v_2 \in N(v_0) - N(v_1)$ do
        if $v_1 < v_2$ then
          break
        for $v_3 \in N(v_1) \cap N(v_2) - N(v_0)$ do
          if $v_0 < v_3$ then
            break
          $count$ ← $count$ + 1
```

(b) One Restriction per Vertex

Figure 4: Matching rectangle with restrictions

```
  begin
    count ← 0
C   for $v_0 \in V$ do
D     for $v_1 \in N(v_0)$ do
B       for $v_2 \in N(v_0) \cap N(v_1)$ do
A         s ← $N(v_2) - N(v_0) - N(v_1)$
          count ← count + |s|
    count ← count / 2
```



```
  begin
    count ← 0
A   for $v_0 \in V$ do
B     for $v_1 \in N(v_0)$ do
C       for $v_2 \in N(v_1) - N(v_0)$ do
D         s ← $N(v_1) \cap N(v_2) - N(v_0)$
          count ← count + |s|
    count ← count / 2
```

Figure 5: Tailed triangle counting with two different schedules

generate these distinct schedules.

Algorithm 4 describes GraphZero's schedule search approach which efficiently prunes the search space without missing high-performance schedules. The optimizations appear in two places: 1) valid schedule generation and 2) equivalent schedule pruning.

Each recursive invocation of the procedure *recursive_generate* tries to include one more vertex into the input schedule until the schedule covers all vertices in the pattern. To avoid generating invalid schedules, after the first vertex has been selected only vertices adjacent to at least one vertex already in the schedule are considered for extending the schedule (lines 9-12,23). These are the vertices contained within *valid_next*.

The insight to prune the schedule space is similar to the idea used in Algorithm 1. When extending the partial schedule to include a vertex from *valid_next*, selecting different vertices may produce equivalent partial schedules due to automorphisms. Algorithm 4 leverages the automorphisms to partition *valid_next* into disjoint sets, such that for any two vertices x, y in a set, any schedule generated by considering x next is equivalent to one considering y next. This property exists if and only if there is an automorphism remaining that moves x to y. Hence, the algorithm only expands the partial schedule by including the first vertex in each set and marks the rest as *processed* (line 22). A *processed* vertex is never considered to extend the partial schedule, thus pruning the schedule space.

Theorem 3 *For a given pattern, Algorithm 4 generates all distinct schedules, and generates no two equivalent schedules.*

We first prove that it generates no two equivalent schedules. Say that it produces two equivalent schedules, S_1 and S_2. Let the first vertex at which S_1 and S_2 differs be v_1 in S_1 and v_2 in S_2. There is necessarily an automorphism mapping S_1 to S_2, as they are equivalent. When the algorithm produces the two schedules, at the point where v_1 and v_2 are processed, neither must be marked by the other. However, since there is an automorphism mapping S_1 to S_2, v_1 would mark v_2 and v_2 would mark v_1. This is a contradiction. Therefore it is impossible for the algorithm to generate both S_1 and S_2.

We then prove that every valid schedule is equivalent to one generated by the algorithm. Suppose, for the sake of contradiction, that a schedule S is not equivalent to any generated schedule. Then there must exist a smallest $i (0 < i \leq |S|)$ where the subschedule formed by the first i vertices (denoted as $S[0:i]$) are not equivalent to any generated subschedule of the same length. Note that i cannot be 0, as all empty schedules are equivalent. The algorithm should generate a schedule T such that $T[0:i-1]$ is equivalent to $S[0:i-1]$. The automorphism x_1 under which the two subschedules are equivalent moves $S[i]$ to some vertex v. If the algorithm extends $T[0:i-1]$ by including v as the next vertex, $T[0:i]$ is equivalent to $S[0:i]$, which is a contradiction. Otherwise, we must have a schedule T' such that $T'[0:i-1]$ is identical to $T[0:i-1]$ and an automorphism x_2 to move $T'[i]$ to v for v to be marked. Then the automorphism composed by $x_2 x_1^{-1}$ moves $T'[i]$ to $S[i]$, which means $T'[0:i]$ is equivalent to $S[0:i]$. Therefore, every valid schedule is equivalent to one generated by the algorithm.

Algorithm 4: Schedule generation algorithm

```
input  : $P$ : the pattern.
output : $schedules$ : the list of all distinct valid
         schedules for $P$
begin
    $schedules \leftarrow []$
    $Aut \leftarrow$ all the automorphisms of $P$
    call recursive_generate([],Aut.{}, schedules)
Procedure $recursive\text{-}generate$
    input  : sched - a partially built schedule
    input  : aut - automorphisms that move $v$ to $v$ for
             $v \in sched$
    input  : valid_next - list of potential next vertices
    input  : schedules - list of distinct schedules
    if $sched$ covers all vertices in the pattern then
        add $sched$ with partial ordering $relations$ to
          schedules
        End this procedure call
    if $sched$ is empty then
        $iterate\_over \leftarrow P.vertices$
    else
        $iterate\_over \leftarrow valid\_next$
    for $v \in iterate\_over$ do
        if $v$ is marked as processed then
            continue
        else
            $stabilized\_aut \leftarrow []$
            for $x \in Aut$ do
                if $x(v) = v$ then
                    add $x$ to $stabilized\_aut$
                else
                    mark $x(v)$ as processed
            call recursive-generate(sched+[v],
              $stabilized\_aut$,
              $valid\_next \cup N(v) - \{v\}$,schedules)
```

5.4 Performance Model

Now that we have successfully generated all distinct schedules, we need a performance model to select a high-performance schedule. Recall that once a schedule is generated, it is naturally mapped to a nested loop structure with restrictions as explained in Section 4. Figure 4 (b) shows the nested loop structure with one restriction per nested for loop for rectangle matching. The performance model needs to estimate for each nested for loop 1) the number of iterations and 2) the number of iterations in which the restriction is satisfied. For the third for loop in the example, the performance model should estimate the number elements in $N(v_0) - N(v_1)$ and the number of times $v_1 < v_2$ holds.

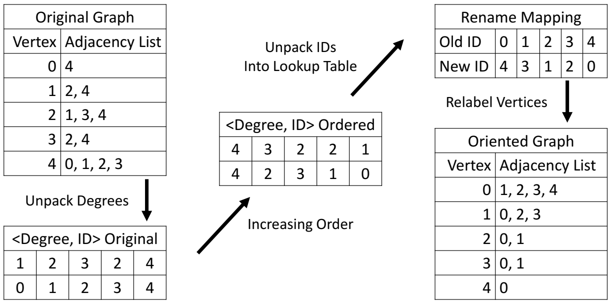
The absolute numbers depend on the graph because in general the larger the graph is the bigger those numbers are. We therefore build a probabilistic model, assuming n vertices in the graph and a probability p for an edge to exist. For a for loop, given that its number of iterations is determined by $k1$ set intersections and and $k2$ set differences, the number of iterations is estimated as $np^{k1+1}(1-p)^{k2}$ by assuming that edge edge is equally likely to occur in a set operation.

Note that given the estimation above, we only need to model the probability of satisfying the restriction. However, it is more difficult because the probability to satisfy the restriction in one for loop depends on the restrictions in all the for loops above it. We therefore model the probability of satisfying all checks up to the restriction of the considered for loop. We omit the formulas to derive these probabilities due to limited space, but will include them in the final version. Finally, we sum up the cost of all for loops to estimate the performance of the schedule.

6 Generalizing Orientation-Based Optimization

In this section, we first explain why the orientation optimization is important for subgraph matching and the problems of generalizing it for arbitrary patterns. We then present GraphZero's method to leverage the automatically generated restrictions for the generalization.

Orientation optimization enforces an order to discover vertices by only allowing higher-degree vertices to be discovered after lower-degree vertices. It works well for triangle counting because every instance of the triangle pattern can be represented by a DAG, starting from the lowest-degree vertex and ending with the highest-degree vertex. The optimization allows pruning all edges from a higher-degree vertex to a lower-degree vertex but still guarantees that every triangle instance can be identified.



Original Graph	
Vertex	Adjacency List
0	4
1	2, 4
2	1, 3, 4
3	2, 4
4	0, 1, 2, 3

<Degree, ID> Original				
1	2	3	2	4
0	1	2	3	4

<Degree, ID> Ordered				
4	3	2	2	1
4	2	3	1	0

Rename Mapping					
Old ID	0	1	2	3	4
New ID	4	3	1	2	0

Oriented Graph	
Vertex	Adjacency List
0	1, 2, 3, 4
1	0, 2, 3
2	0, 1
3	0, 1
4	0

Figure 6: Graph Reindexing Approach

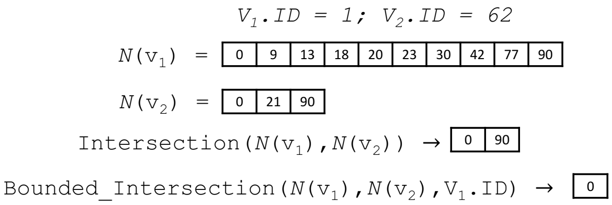

V1.ID = 1; V2.ID = 62
N(v1) = 0 9 13 18 20 23 30 42 77 90
N(v2) = 0 21 90
Intersection(N(v1),N(v2)) → 0 90
Bounded_Intersection(N(v1),N(v2),V1.ID) → 0

Figure 7: Bounded Intersection Benefits

While the optimization only reduces the number of edges to process by a factor of 2, it substantially reduces the neighbor set size of "hot" vertices, which appear more frequently in patterns. Consider a star topology graph of $N+1$ vertices with the center "hot" vertex connected to each of the other vertices. With orientation optimization, its neighbor set becomes empty. But in the original graph, the neighbor set of size N needs to be accessed N times.

Generalizing the orientation optimization faces two problems. First, the direction of the pruned edge may interfere with the discovery order determined by the schedule. For instance, a restriction of the schedule may require that two mapped vertices v_i and v_j should satisfy $v_i.ID > v_j.ID$, but v_i's degree is larger than v_j's degree. In this case, the pruned edge is necessary for the schedule to discover v_j from v_i. Second, the orientation optimization must prune edges to gain benefit, but the same graph may also be used by other applications which may not work with the modified dataset.

GraphZero generalizes the orientation optimization on arbitrary patterns without pruning any edge by implementing two techniques. The first technique reindexes the vertices such that the higher-degree vertices must have a smaller ID than that of the lower-degree vertices, as shown in Figure 5. It uses the original IDs for tie-breaking when vertices have the same degree. Recall that when GraphZero enforces a restriction $(v_i.ID > v_j.ID)$, it also respects the discovery order from lower-degree vertices to higher-degree vertices by only allowing v_j to be discovered after v_i. The second technique extends the set operations to leverage the restrictions for early exit (i.e., reducing the accessed elements in neighbor sets). Given a restriction $(v_i.ID > v_j.ID)$, when discovering v_j we are only interested in the neighbors in v_i's neighbor set whose ID is less than $v_i.ID$. We can hence use $v_i.ID$ as a bound to perform the set operations. Because the neighbor sets are stored as sorted lists of integers, a linear scan finds the output vertices in sorted order, and can terminate whenever a bound condition is met. The technique is particularly useful for large-degree vertices as shown in Figure 7. After the reindexing, v_1, a large-degree vertex, has a small ID as 1. If there exists a restriction involving v_1, the intersection can use $v_1.ID$ as the bound. In the example, the original intersection needs to traverse all elements in $N(v_1)$ while the bounded intersection only traverses three elements in total.

Graphs	#Vertices	#Edges	Description
CiteSeer [15]	3264	4536	Publication citations
Wiki-Vote [33]	7115	100762	Wiki Editor Voting
MiCo [15]	96638	1080156	Co-authorship
DBLP [9, 10]	317080	1049866	Co-authorship
Patents [34]	3.8M	16.5M	US Patents
LiveJournal [7]	4.8M	42.9M	Social network
Orkut [2]	3.1M	117.2M	Social network
UK-2002 [9, 10]	18.5M	261.8M	Web graph
Twitter [9, 10, 29]	41.7M	1.2B	Social network

Table 1: Graph Datasets

7 Evaluation

In this section, we evaluate GraphZero and compare it with RADS, a state-of-the-art non-compilation based system, and AutoMine, a state-of-the-art compilation-based system. The highlights of the results are as follows: 1) GrapZero running on a single machine is up to 2654X faster on a small graph and up to 57.8X faster on two large graphs than RADS running on a 10-machine cluster. 2) For 10 different workloads on real-world graphs, GraphZero is up to 40X faster than AutoMine running on the same machine. 3) The compilation time reduction opens up the potential for a just-in-time compilation process with up to 197X speedup over the AutoMine compiler. 4) The generalized orientation optimization obtains the benefits of AutoMine's clique-specific optimization for arbitrary patterns yielding up to 88.6X speedup.

7.1 Methodology

7.1.1 Experimental Setup

Table 1 shows the 9 real-world graphs used in the experiments. AutoMine uses 5 of them to demonstrate that it outperforms prior matching systems including Arabesque [50] and RStream [53] by up to 4 orders of magnitude. We hence also use these graphs to experiment with GraphZero. We include DBLP and UK2002 used in the evaluation of RADS [19]. We also include Wiki-Vote to observe scalability to large patterns and Twitter to evaluate scalability to large graphs. We run

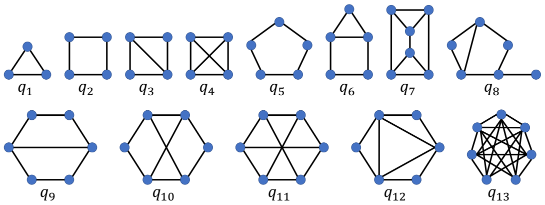


Figure 8: Query Patterns

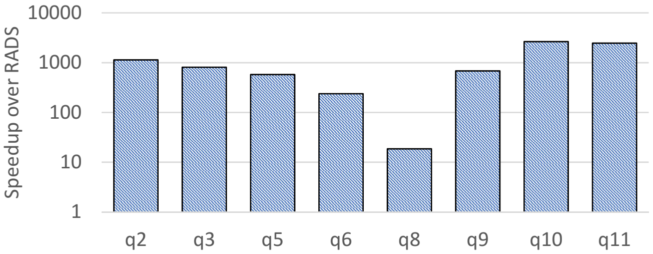


Figure 9: Speedup versus RADS on DBLP

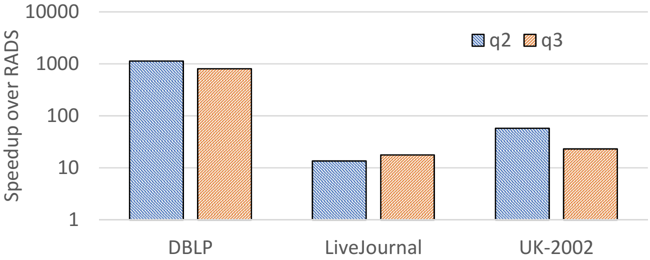


Figure 10: Speedup versus RADS on Two Queries

Graph	App.	AutoMine	GraphZero
CiteSeer	3-MC	1.6ms	0.9ms
	4-MC	11.9ms	2.4ms
	5-MC	537ms	38ms
Wiki-Vote	3-MC	34.5ms	9.2ms
	4-MC	11.5s	1.7s
	5-MC	5300s	500s
MiCo	3-MC	230ms	60ms
	4-MC	45.2s	15.2s
	5-MC	5.56h	1.2h
Patents	3-MC	1.9s	0.74s
	4-MC	82.1s	10.2s
	5-MC	117m	12.7m
LiveJournal	3-MC	13.4s	4.04s
	4-MC	367m	54m
Orkut	3-MC	82.2s	23.1s
	4-MC	43.7h	7.4h
Twitter	3-MC	31.3h	3.9h

Table 2: Motif Counting Performance

experiments on machines with 2 8-core Intel Xeon E5-2670 CPUs (hyperthreading disabled) and 64GB of memory. Each machine runs Red Hat Enterprise Linux 6.9 with Linux kernel version 2.6 and gcc version 4.4.7, which we use with optimization level O3. Figure 8 shows the 13 patterns we focus on for this evaluation, which range in size from 3 to 7 vertices. We also perform motif counting, a popular application used to evaluate many other subgraph matching systems [25, 37, 53], on up to 5 vertices using aggregate schedules to efficiently count all motifs on a particular number of vertices.

7.2 Performance Comparison

7.2.1 Individual Patterns

Since the RADS system is not released, we refer to the performance results collected on a 10-machine cluster (each with 16 cores) reported in the paper [19]. As Figure 9 shows, on the DBLP graph, GraphZero outperforms RADS by 1131X on q1 and up to 2654X on q7, a complex 6-node pattern. The worst case for GraphZero, q5, still reduces the execution time from 269s down to 14s. On the larger graphs, the greater computational resources in the cluster become a larger advantage for RADS as seen in Figure 10. Even still, the q1 runtime on UK-2002 is reduced from 11700s in RADS to 202s in GraphZero.

To compare with AutoMine, we run 6 matching applications, corresponding to the non-clique target patterns in Figure 8, on the 5 smaller graphs. This directly showcases the improvements that multiplicity reduction offers. Notice that in Figure 11, GraphZero outperforms AutoMine by up to 40X for q13 and 22X for q5, running on Wiki-Vote and Patents respectively. The best case scenario for those is expected to be 120X and 5X if the multiplicity reduction provided linear speedup, but GraphZero's generalized orientation optimization allows us to exceed that in the case of q5.

7.2.2 Motif Counting

Motif Counting finds all connected patterns of a specified number of vertices, for our purpose between 3 and 5. Both the complexity of each pattern and the number of patterns increase quickly with the number of vertices, with only 2 patterns on 3 vertices, but 21 on 5 vertices. We run Motif Counting for 3, 4, and 5 vertices on all 7 graphs with a 72 hour timeout. Table 2 shows all of the results that completed within the time limit. We observe that GraphZero outperforms AutoMine for all workloads with the smallest speedup of 1.75X and largest speedup of 14X. Notice that as the motif size increases, there is a corresponding sharp increase in the computational costs. We therefore only successfully finish the experiments on Motif-3 and Motif-4 with LiveJournal and Orkut and Motif-3 with Twitter. For the five smallest graphs, where the speedup is relatively small for Motif-3 (up to 3.8X), we see a trend of increasing speedup relative to AutoMine.

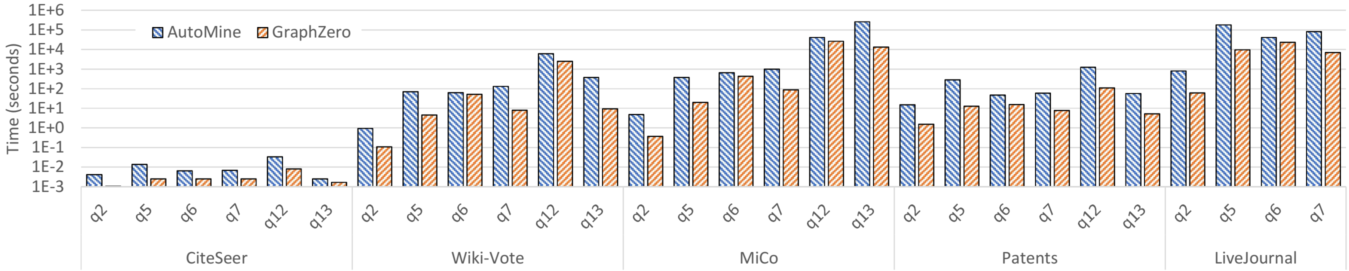


Figure 11: Individual Pattern Performance

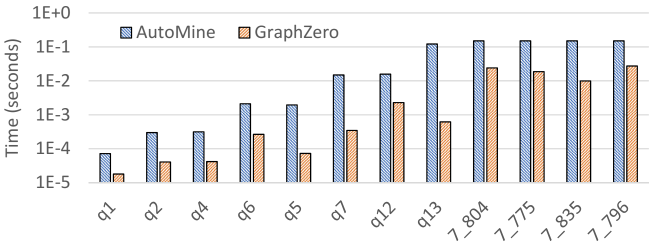


Figure 12: Individual Pattern Compilation Time

On Motif-5, the speedup ranges from 4.7X to 14X.

7.3 Schedule Generation

7.3.1 Compilation Speed Comparison for Individual Patterns

Considering patterns individually provides a direct comparison between the compilers in a pattern query scenario. Figure 12 shows that where the AutoMine compiler takes up to 121ms for q13, the GraphZero compiler takes just 0.6ms, a 197X speedup. Other 7-vertex patterns may take even longer. Of the 853 patterns of 7 vertices, the 4 most expensive (shown as 7_804, 7_775, 7_835 and 7_796) take over 149ms on their own in AutoMine. GraphZero completes all of these in under 30ms each, with an average speedup of over 8.7X and a maximum of 15.1X on the 7_775 pattern (meaning index 775 in pattern discovery order). The results demonstrate that GraphZero's compilation technique has promise for use in a just-in-time compiler.

7.3.2 Compilation Speed Comparison for Motifs

Combining the schedules for multiple patterns improves performance through data reuse, but can be expensive at compilation time. For Motif-7, each individual pattern is expensive to compile, and the aggregate compilation for all 853 patterns takes AutoMine over 2.5 minutes to complete. The speedup that GraphZero achieves on individual patterns naturally benefits the combined case, with a total compilation time of just 37 seconds, a 4X performance improvement. This trend will only continue with larger patterns, and with this performance, GraphZero has the potential to consider even larger patterns in the future.

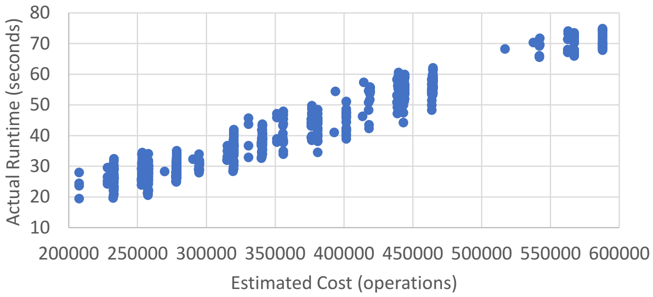


Figure 13: Estimated versus Actual Performance on Patents

7.3.3 Performance Modeling

The performance model, as discussed in Section 5, estimates the number of operations each schedule needs to perform. It does not have to estimate real runtime, as its only purpose is to find the highest performance schedule. The estimates are computed in terms of number of operations performed in a uniform random graph with 1000 vertices and an average degree of 5. We show in Figures 13 that the estimated cost and real performance are strongly correlated. The Coefficient of Determination describes the strength of statistical correlation, with 1 being a correlation that is perfectly defined by the data. We observe that for Patents this value comes out as 0.94, demonstrating the strong relative predictive capabilities of the performance model. The selected schedule according to the heuristic described in Section 5.4 was the best schedule on Patents, though run-to-run variation can be up to 5%. According to these results, we conclude that the performance model and heuristic successfully ensure that GraphZero selects a high-performance schedule.

7.4 Orientation Optimization

7.4.1 Cliques

The AutoMine work generalizes the orientation optimization to any-size cliques, but requires manually pruning all

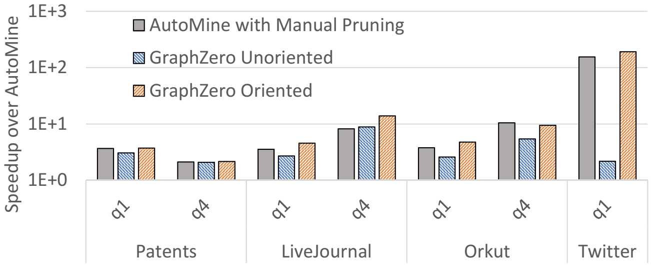


Figure 14: Clique Performance Comparison

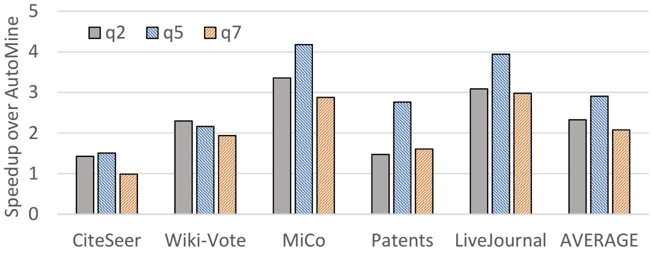


Figure 15: Speedup from Orientation

edges from higher-degree vertices to lower-degree vertices. GraphZero's schedules automatically implement the optimization when running on the oriented graph. Figure 14 uses the automatically generated code from AutoMine as the baseline and shows the speedups of AutoMine with manual pruning and GraphZero with both unoriented and oriented graphs on q1 and q4 (triangles and size-4 cliques). We only show the four largest graphs in which q1 matching takes at least 100ms. The manual pruning makes AutoMine much faster than the baseline, producing on average a 24.9X speedup for q1 and 7.7X for q1. GraphZero without orientation achieves up to 94% of its performance on MiCo q1, but falls behind in most other cases. The oriented graph, however, allows GraphZero to achieve up to 1.4X and up to 2X speedup over AutoMine with Manual Pruning on q1 and q4, respectively, both with MiCo. We point out that GraphZero with orientation and AutoMine with manual pruning process the same amount of graph data (i.e., the edges from lower-degree vertices to higher-degree vertices). The extra performance benefit from GraphZero with orientation is from better load balance because of the degree-based sorting.

GraphZero's final performance beats default AutoMine by an average of 30.7X on q1 and 9.4X on q4. It is especially interesting that GraphZero achieves a 192X speedup over the baseline for Twitter from using the orientation technique for q1. A plausible reason is that because Twitter is the largest graph, its high-degree vertices contribute to a greater portion of the runtime than for other graphs.

CiteSeer	Wiki-Vote	MiCo	Patents	LiveJournal	Orkut	Twitter
3.3ms	10.2ms	32ms	0.88s	1.7s	2.8s	77.7s

Table 3: Graph reindexing overhead

7.4.2 General Patterns

We evaluate the speedup directly attributed to the orientation in GraphZero for three queries – q2, q5, and q7, on which AutoMine cannot apply the optimization. Figure 15 demonstrates the speedup of GraphZero with orientation over GraphZero without orientation. The performance improves by up to 4.2X, with q5 seeing the most benefit at an average of 2.9X and q7 seeing the lowest benefit at an average of 2.1X. These results are expected because the sparsity of the q2 and q5 patterns makes them more likely to include high-degree vertices, whereas the dense subpatterns (q1 subgraphs) in q7 filter out many of those high-degree vertices. But the benefit from orientation optimization is maximized on high-degree vertices.

7.4.3 Cost Benefit Analysis

Using the oriented graph typically improves performance, but not necessarily for free. Applying the orientation optimization in GraphZero incurs overhead for the reindexing process. If the processing happens offline, there is no runtime overhead since the resultant graph is structurally equivalent to the original. We also consider the possibility of reindexing the graph just-in-time, as the operations have low complexity compared to large pattern matching. Table 3 reports the reindexing overhead for all the graphs used for evaluation. On CiteSeer, the smallest graph with only 4536 edges, even the 3.3ms time is large compared to the total mining time, so it is not worth applying the optimization. However, as the graph size increases, the benefit of orientation substantially outweighs the overhead. For the largest graph Twitter, reindexing takes 77.7 seconds, but saves over 4 hours of processing for q1. while the absolute cost to orient the graph indeed grows with the graph, the performance benefit scales up far faster.

8 Related Work

General graph analytics systems. Many graph processing systems, including GraphLab [35], Graph [30], Gemini [61], Pregel [36], GridGraph [62], XStream [42], and Ligra [46], expose a think-like-a-vertex or think-like-an-edge abstraction, which makes it easy to express graph traversal algorithms such as breadth first search. The distributed systems focus on optimizing communication [45], locality [17], and load balance [27], whereas the single-machine systems heavily optimize I/O scheduling [57], minimize data loading [52], or trade off accuracy for performance [28]. However, none of these systems can be easily used to compose subgraph matching applications, because of the gap between the low-level abstraction and the structural patterns.

Subgraph matching systems. Single-machine subgraph matching systems focus on optimizing the matching order and avoiding redundant computation. Turbo$_{\text{iso}}$ [21] uses a spanning tree of the query graph to speed up matching, but it enumerates numerous false positive matches and fails to enforce high-performance matching orders. Han et al. [20] propose adaptive matching orders to improve performance and pruning by failing set to reduce redundancy. Sun et al. [48] design the LIGHT system, which defers the materialization of pattern vertices and converts the candidate set computation into finding the minimum set cover to eliminate redundancy. Distributed subgraph matching systems strive to optimize pattern decomposition for matching [6, 31, 41, 44, 49], minimize inter-machine communication [19, 54], or improve load balance [8].

Almost all subgraph matching systems use the symmetry-breaking technique proposed by Grochow and Kellis [18]. GraphZero is different in three ways. First, GraphZero completely eliminates redundancy in schedule search, which is never explored before. Second, GraphZero can generate restrictions for a given high-performance schedule, while previous systems can not simultaneously enforce a schedule and break symmetry. Third, GraphZero guarantees to enforce a minimum number of restrictions.

Graph mining systems. Graph mining systems can also performs subgraph matching because they can search for structural patterns in graphs. Arabesque [50] is the first distributed graph mining system that supports high-level interfaces for user to easily specify and mine patterns. A number of graph mining systems are then proposed with optimized memory consumption [11], depth-first search [14], out-of-core processing [53, 59], or approximate mining support [25, 38]. As pointed out in [37], these systems implement generic but inefficient mining algorithms and incur unnecessary global synchronizations.

AutoMine is a unique graph mining system built upon a set-based representation. Given a list of patterns, its compiler can automatically generate a nested loop structure of set intersection and subtraction operations to identify all of them. It substantially outperforms prior graph mining systems and is hence used as the baseline to evaluate GraphZero. EmptyHeaded [3] and GraphFlow [26] are similar systems, but they focus on optimizing set intersection operations and cannot handle missing edges in patterns. For example, they may consider the two-edge path in a triangle as an instance of the wedge pattern, which is unacceptable in many applications. In contrary, GraphZero, like AutoMine, supports arbitrary patterns and completely eliminates redundancy in both schedule search and code execution.

Optimizing compilers for graph processing. Many compilers for graph analytics perform sophisticated optimizations once the graph algorithm is clearly expressed using the provided domain-specific language. GraphIt [58] enables user to describe the graph algorithm in the algorithm language and how the algorithm should be optimized in the scheduling language. This separation allows the user to focus on algorithm design and offload the optimization tasks to the compiler. Green-Marl [23], SociaLite [43], and Abelian [16] can automatically parallelize and optimize graph algorithms but the optimization space they can explore is significantly smaller compared with GraphIt. Pai and Pingali [40] propose a set of compiler optimization techniques to efficiently map graph algorithms to the GPU architecture. None of these compilers can generate efficient graph mining algorithms, let alone remove redundancy, which is one of GraphZero's most important contributions.

Orientation optimization. The orientation optimization was first proposed by Latapy [32] for triangle counting. Shun et al. [47] and Voegele et al. [51] extend it for parallel triangle counting. Hu et al. [24] applies it to triangle counting on GPUs. AutoMine is the first work that generalizes the optimization for any-size clique patterns. To our knowledge, no prior work could apply the optimization to arbitrary patterns, which is made possible in GraphZero.

9 Conclusion

We proposed an optimizing compiler, GraphZero, that systematically addresses the limitations of existing compilation-based subgraph matching systems. GraphZero breaks symmetry in patterns, which causes serious performance and compilation overhead problems in other systems, through automatically generated and enforced restrictions. It leverages the generated restrictions and generalizes an important optimization for subgraph matching to arbitrary patterns based on a reindexing technique. The experiments showed that GraphZero substantially outperformed both RADS and AutoMine for multiple patterns on real-world graphs.

Acknowledgements

This project was supported in part by NSF grant CCF-1823005 and an NSF CAREER Award (CNS-1750760).

References

[1] graphchallenge. https://graphchallenge.mit.edu/, November 22, 2019.

[2] Orkut social network. http://snap.stanford.edu/data/com-Orkut.html, November 22, 2019.

[3] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[4] N. K. Ahmed, J. Neville, R. A. Rossi, and N. G. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 1–10, 2015.

[5] L. Akoglu, H. Tong, and D. Koutra. Graph-based anomaly detection and description: A survey. *CoRR*, abs/1404.4679, 2014.

[6] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *PVLDB*, 11(6):691–704, 2018.

[7] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[8] B. Bhattarai, H. Liu, and H. H. Huang. CECI: compact embedding cluster index for scalable subgraph matching. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1447–1462. ACM, 2019.

[9] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[10] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[11] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 32:1–32:12, 2018.

[12] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, pages 157–168, 2015.

[13] S. Chu and J. Cheng. Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data*, 6(4):17:1–17:32, Dec. 2012.

[14] V. V. dos Santos Dias, C. H. C. Teixeira, D. O. Guedes, W. M. Jr., and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1357–1374, 2019.

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

[16] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In *European Conference on Parallel Processing*, pages 249–264. Springer, 2018.

[17] G. Gill, R. Dathathri, L. Hoang, and K. Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *PVLDB*, 12(4):321–334, 2018.

[18] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In T. P. Speed and H. Huang, editors, *Research in Computational Molecular Biology, 11th Annual International Conference, RECOMB 2007, Oakland, CA, USA, April 21-25, 2007, Proceedings*, volume 4453 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

[19] M. Han, H. Kim, G. Gu, K. Park, and W. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1429–1446. ACM, 2019.

[20] M. Han, H. Kim, G. Gu, K. Park, and W. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1429–1446. ACM, 2019.

[21] W. Han, J. Lee, and J. Lee. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013,*

New York, NY, USA, June 22-27, 2013, pages 337–348. ACM, 2013.

[22] T. Hocevar and J. Demsar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[23] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. *ACM SIGARCH Computer Architecture News*, 40(1):349–362, 2012.

[24] Y. Hu, H. Liu, and H. H. Huang. Tricore: parallel triangle counting on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 14:1–14:12, 2018.

[25] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, 2018. USENIX Association.

[26] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698. ACM, 2017.

[27] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[28] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016*, pages 245–257, 2016.

[29] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. AcM, 2010.

[30] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, 2012.

[31] L. Lai, L. Qin, X. Lin, Y. Zhang, and L. Chang. Scalable distributed subgraph enumeration. *PVLDB*, 10(3):217–228, 2016.

[32] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.

[33] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650. ACM, 2010.

[34] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187, 2005.

[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[36] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[37] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523. ACM, 2019.

[38] D. Mawhirter, B. Wu, D. Mehta, and C. Ai. Approxg: Fast approximate parallel graphlet counting through accuracy control. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, pages 533–542, 2018.

[39] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003.

[40] S. Pai and K. Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *ACM SIGPLAN Notices*, volume 51, pages 1–19. ACM, 2016.

[41] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *PVLDB*, 11(2):176–188, 2017.

[42] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, 2013.

[43] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE, 2013.

[44] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 625–636. ACM, 2014.

[45] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 317–332, 2016.

[46] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[47] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 149–160, 2015.

[48] S. Sun, Y. Che, L. Wang, and Q. Luo. Efficient parallel subgraph enumeration on a single machine. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 232–243. IEEE, 2019.

[49] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[50] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 425–440, 2015.

[51] C. Voegele, Y. Lu, S. Pai, and K. Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*, pages 1–7, 2017.

[52] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

[53] 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 *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 763–782, 2018.

[54] Z. Wang, R. Gu, W. Hu, C. Yuan, and Y. Huang. BENU: distributed subgraph enumeration with backtracking-based framework. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 136–147. IEEE, 2019.

[55] S. Wernicke and F. Rasche. Fanmod: A tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, May 2006.

[56] C. Yang, M. Liu, V. W. Zheng, and J. Han. Node, motif and subgraph: Leveraging network functional blocks through structural convolution. In *IEEE/ACM 2018 International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2018, Barcelona, Spain, August 28-31, 2018*, pages 47–52, 2018.

[57] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 608–621, 2018.

[58] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe. Graphit: a high-performance graph DSL. *PACMPL*, 2(OOPSLA):121:1–121:30, 2018.

[59] C. Zhao, Z. Zhang, P. Xu, T. Zheng, and X. Cheng. Kaleido: An efficient out-of-core graph mining system on A single machine. *CoRR*, abs/1905.09572, 2019.

[60] R. Zhu, Z. Zou, and J. Li. Fast rectangle counting on massive networks. In *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*, pages 847–856, 2018.

[61] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, Savannah, GA, 2016. USENIX Association.

[62] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, 2015. USENIX Association.