Project Report
LSP-267

Exact Subgraph Matching using Massively Parallel Graph Exploration and Linear Algebra: FY19 Information, Computation & Exploitation Line-Supported Program

V. Gleyzer
E.K. Kao

18 November 2019

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Massachusetts Institute of Technology
Lincoln Laboratory

Exact Subgraph Matching using Massively Parallel Graph Exploration and Linear Algebra: FY19 Information, Computation & Exploitation Line-Supported Program

V. Gleyzer
Group 102

E.K. Kao
Group 104

Project Report LSP-267

18 November 2019

Lexington Massachusetts

This page intentionally left blank.

ABSTRACT

Exact subgraph matching, which is a well-studied algorithmic kernel that is critical for many scientific and commercial applications, is an NP-complete problem. Over the years, multiple heuristic-based and approximation-based techniques have been proposed to help alleviate this complexity. In order to apply these techniques for analysis of graphs that contain millions to billions of edges, distributed systems have provided computational scalability through search parallelization. One recent approach for leveraging computational parallelism is through the linear algebra formulation using the matrix multiply operation, which provides a mechanism for performing parallel graph traversal. Benefits of this approach are: 1) a mathematically precise and concise representation, and 2) the ability to leverage specialized linear algebra accelerators and optimized libraries for high-performance analysis of large graph datasets.

In this paper, we design a multi-stage, linear algebra representation and methodology for exact subgraph matching. This formulation provides a stateless discovery mechanism to explore the graph starting at multiple vertices simultaneously and performing this exploration in a massively parallel fashion. We present a preliminary analysis of the approach and demonstrate key advantages of the proposed formulation.

This page intentionally left blank.

ACKNOWLEDGMENTS

The authors would like to thank Michael Yee, Paul Monticciolo, Sanjeev Mohindra, William Song and Robert Bond for their invaluable support and feedback.

This page intentionally left blank.

TABLE OF CONTENTS

This page intentionally left blank.

LIST OF FIGURES

LIST OF FIGURES

(Continued)

1. INTRODUCTION

Given a directed, labeled query graph Q and a data graph G, exact subgraph matching can be defined as finding all non-induced subgraphs, S, of G that are isomorphic to Q and for which corresponding vertices and edges of Q and S share the same attribute values. Subgraph matching is a key kernel found in applications such as bioinformatics, social media analysis and financial transaction analysis. Although general subgraph isomorphism, and by extension subgraph matching, is known to be an NP-Complete problem [1], many parallel, heuristic and approximate algorithms have been proposed to help analyze practical real-world datasets.

In this paper, we demonstrate a subgraph matching algorithm which heavily leverages matrix-based graph traversal for identifying matches. Our key motivation for this approach is the availability of sparse linear-algebra libraries [2,3] that are mapped and optimized for execution of linear algebra primitives on an array of large distributed systems, including a specialized sparse-linear algebra accelerator proposed in [4], as well as traditional high-performance processing systems. These capabilities can enable analysis of graph datasets that would be intractable using alternative methods.

1.1 NOTATION

For parsimony, both the data graph, G, and the query graph, Q, are defined as binary, unlabeled, directed graphs.[1] We use $V(G)$ to represent the vertices of the data graph, and analogously $V(Q)$ to represent the vertices of the query graph. We also define the following functions and operators for any matrix M of real values:

1. $find(M)$ which is a the non-zero entry selector function that returns two vectors, r and c, where r contains the row indices and c contains the column indices of all non-zero values of the matrix M;
2. the $\lfloor M \rfloor$ operator which binarizes the matrix M:
$$\lfloor M \rfloor = \begin{cases} 1 & m_{ij} \neq 0 \\ 0 & otherwise \end{cases};$$
3. the $\triangledown M$ operator which returns the upper-triangular part of a square matrix, excluding the diagonal:
$$\triangledown M = \begin{cases} m_{ij} & i < j \\ 0 & otherwise \end{cases}.$$

1.2 MATRIX-BASED GRAPH TRAVERSAL

The data graph, G has N vertices. Its edges are represented by an $N \times N$ binary adjacency matrix A, where each non-zero element a_{ij} indicates the presence of an edge from vertex i to vertex

[1] Support for vertex and edge labels are discussed in more detail in Section 3

j. W_k is defined to be a $N \times N$ walk matrix, where each element w_{ij} indicates the number of paths from vertex i to vertex j after k steps. Following these definitions, W_k can be computed using the powers of the adjacency matrix,

$$W_k = A^k, \tag{1}$$

and can be expressed recursively using

$$W_k = \begin{cases} I & k = 0 \\ W_{k-1} \cdot A & k \geq 1 \end{cases}. \tag{2}$$

With I being equal to the identity matrix, each row, $w_{i:}$, in W_k represents an independent k-hop *breadth-first-search* (BFS) traversal of the graph starting at vertex i. This formulation provides a mechanism to explore the graph starting at multiple vertices simultaneously and performing this exploration in a massively parallel fashion.

1.3 RELATED WORK

Because of mathematical rigor and inherent parallelism, there has been recent interest in leveraging linear algebraic primitives to define and implement graph algorithms [5–12]. From the perspective of exact pattern matching, however, most previous work has focused specifically on triangular queries [8, 10] or cycles [13]. This paper leverages this work as a starting point and expands the linear algebra approach to define an end-to-end algorithm to support enumeration of arbitrary subgraph queries.

Subgraph matching is a NP complete problem; therefore, various approaches utilize different fundamental primitives in an attempt to address the inherent complexity. Many successful algorithms, such as presented in [14–17], leverage tree-based backtracking techniques with different heuristic methods to quickly prune the search space. These approaches have been demonstrated to work well on a single, shared-memory processing system. However, since the tree search requires substantial communication for traversal of the graph, which can be distributed across tens of thousands of processors, these techniques have not been successfully implemented to analyze large-scale, distributed datasets. There has been recent work to help address these challenges, such as [18, 19]. Since the main focus of this paper is to define a linear algebra formulation, we will leave a more rigorous comparison to how well these techniques compare to our approach for future work. However, one of the benefits of the linear algebra primitive is the ability to leverage inherent matrix parallelism to lower communication requirements for multi-path traversal.

An alternative approach for performing subgraph matching is utilizing a database join-based primitive [20–22]. The query graph is treated similar to a standard database query and is decomposed into multiple small subgraphs or subqueries. Based on an optimization strategy and an execution plan, these components are individually enumerated to find partial matches, and are subsequently joined in a specific order to find complete matches for the original query graph. The join order of the components can have a considerable effect on the number of intermediate results, and therefore has significant impact on the overall runtime of the algorithm. Because of the inherent similarity between matrix multiplication and a database join operation, the approach presented in this paper

will be able to leverage many concepts introduced by the database community, including join-order optimization and general query planning.

Pruning techniques, such as the ones presented in [23] generate a set of attribute and topological constraints based on the requirements defined by the query graph. Then, utilizing these constraints to pre-filter the original graph, they can potentially eliminate a substantial portion of the vertices, thus, dramatically reducing the subgraph search space. These approaches are complimentary to the work proposed in this paper since they can be implemented using a pre-processing and filtering step which can be a prerequisite to any exact graph matching algorithm. Furthermore, since most of these constraints (e.g., topological constraints) require explicit traversal of the graph, they can potentially be accelerated using the same massively parallel traversal primitives presented in this paper.

1.4 CONTRIBUTIONS

- Designed an end-to-end linear algebra representation and methodology for exact subgraph matching by joining detected subgraph components
- Developed efficient and massively parallel walks on graph to detect subgraph components, using matrix-matrix multiplies
- Integrated symmetry-breaking and revisit-detection techniques into the parallel walks to remove invalid paths from consideration
- Demonstrate the advantages of the proposed subgraph component detection, over a range of realistic graph topologies

This page intentionally left blank.

2. ALGORITHM DESCRIPTION

As illustrated in Figure 1, the algorithm is divided into two phases: *planning* and *execution*. During the *planning* phase, the data and the query graphs are analyzed in order to identify the best execution plan. Specific tasks include:

1. decomposition of the query into multiple smaller subgraphs, which we will refer to as components,
2. proposal of the join order with which these partial matches are combined, and
3. analysis of symmetry and generation of symmetry-breaking constraint for all intermediate steps.

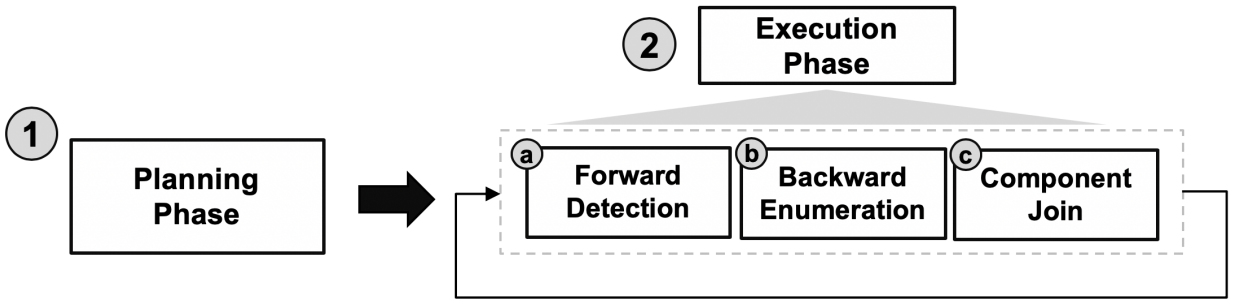


Figure 1. Algorithm overview.

Even though planning plays a critical role in subgraph matching algorithms, the work associated with this phase is typically several orders of magnitude smaller than the actual execution. In this paper, our main focus is to develop a linear algebra framework to accelerate the subgraph matching execution while leveraging all relevant techniques utilized for traditional query planning and optimization, similar to the ones proposed in [20–22, 24]. Since these techniques are secondary to our thesis, we will refer the reader to the references for further details.

The *execution* phase is responsible for incrementally searching and joining components until all of the subgraphs matching the query graph are found. The phase consists of three steps:

1. *component detection* : performs parallel walks to detect components and stores state necessary to reconstruct enumeration for valid matches;
2. *component enumeration* : identifies and enumerates all of the vertices participating in the component matches found in the detection step;
3. *multi-component join* : joins newly found components with enumerated partial matches from previous iterations according to the execution plan.

Algorithm 1 Top-level Pseudocode

Output: R: enumerated result set
Input: Q: query graph, A: data graph adjacency matrix
  $R \leftarrow \emptyset$
  $P \leftarrow generateQueryPlan(Q, A)$
  for all $p \in P$ **do**
    $W \leftarrow componentDetection(p, A)$
    $C \leftarrow componentEnumeration(W, A)$
    $R \leftarrow componentJoin(R, C)$
  end for

As illustrated by Algorithm 1, this procedure is repeated until the entire execution plan (i.e., all individual steps p) has been completed. Note, for the sake of clarity, we will assume that the execution plan is implicitly available during all steps of the execution.

2.1 COMPONENT DETECTION

In order to avoid enumerating every partial match during the graph traversal, we propose to utilize a matrix-based forward search for 2, 3 or 4-vertex components. As we will demonstrate in Section 4, the number of enumerated paths for a single component can be significantly smaller than the total number of explored paths, and thus, by performing an efficient look-ahead operation, we can lower the overall cost of the subsequent enumeration step.

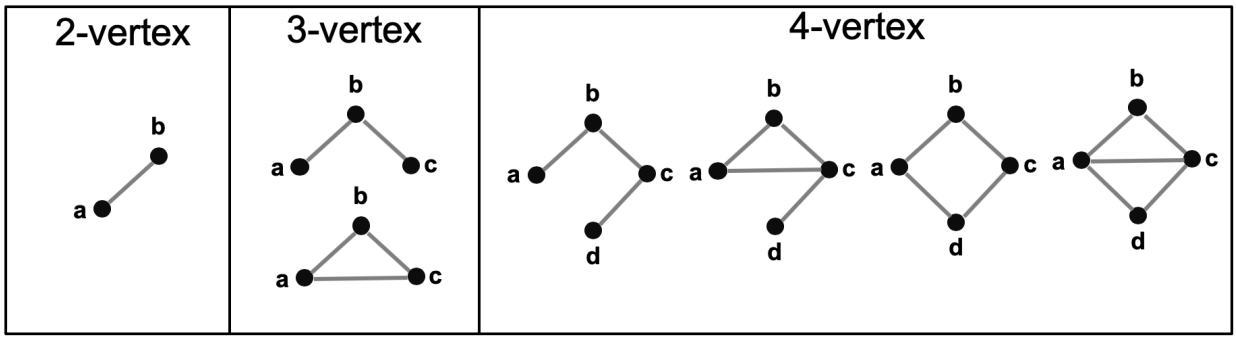


Figure 2. Example hop-sequences for components which can be traversed without a multi-component join.

Several components that can be detected using this matrix-based search are depicted in Figure 2. For example, Algorithm 2 provides a specific implementation for a directed, 4-node cycle query component illustrated in Figure 3. This example contains several key ideas, which are introduced in the subsequent subsections, that can be leveraged to detect all 4-node components except a 4-clique subgraph. This limitation is further addressed in Section 3.

Algorithm 2 Detection of directed 4-cycle component

Output: W_k: walk matrices

Input: A: data graph adjacency matrix; W_0: diagonal matrix with 1s indicating all of the starting vertices

1: $W_1 = \triangledown(W_0 \cdot A)$
2: $W_2 = \triangledown(W_1 \cdot A)$
3: $W_{a\underline{b}c\underline{b}} = (\lfloor W_2 \rfloor \cdot (A \circ A^T)) \circ W_1$
4: $W_3 = \triangledown(W_2 \cdot A) - W_{a\underline{b}c\underline{b}}$
5: $W_4 = diag(W_3 \cdot A)$

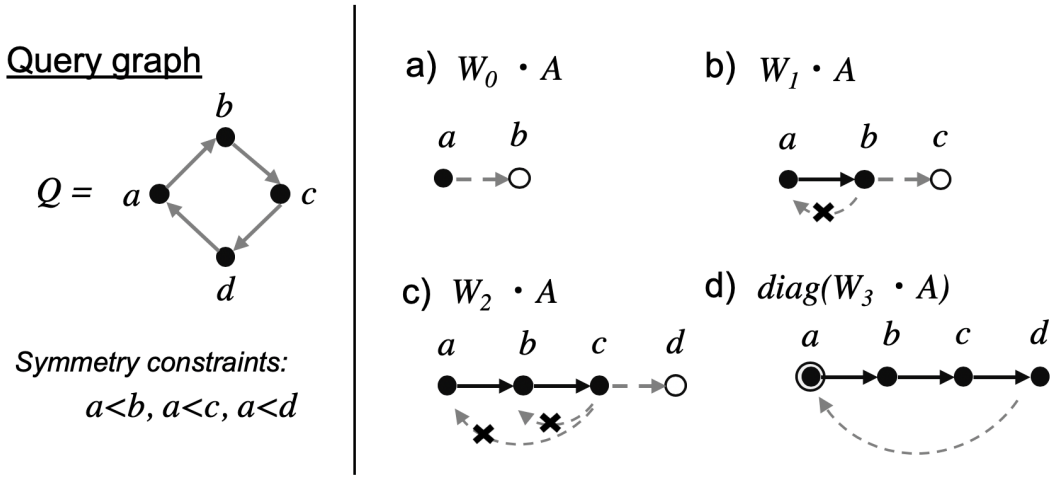


Figure 3. Visualization of the individual hops taken on the query graph Q for Algorithm 2. Crossed out arrows indicate invalid paths which can be traversed as part of each operation.

2.1.1 Basic Graph Traversal

Matrix multiplication of the walk matrices, W_k, by the adjacency matrix or its transpose allows the walk to progress forward and traverse all of the outgoing or incoming edges, respectively. As such, W_0 initializes the starting vertices for each parallel exploration and contains all of the candidates for the query graph vertex a. This approach mirrors the rudimentary walk primitive introduced Eq. 2. However, even though the basic matrix multiplication provides a way to perform a massively parallel walk, by itself, it does not define a mechanism to filter or discriminate degenerate or redundant paths. The ability to filter these paths early can potentially have significant impact on the runtime performance, since traversing these paths requires both communication and processing resources.

2.1.2 Cycle removal

As mentioned earlier, simple BFS walks, such as the ones described by Eq. 2, do not discriminate between the paths that are taken. So, paths, such as cycles, are continuously traversed. For example, given a graph that contains a simple 2-cycle subgraph, the traversal would continue to oscillate between the two vertices in the cycle as k increased. Specifically in the case of subgraph matching, this would indicate a degenerate path, since the vertices in the final subgraph must be unique and non-repeating. As demonstrated in Algorithm 2, cycles that return back to the starting vertex candidates are represented by the diagonal entries of the walk matrices and can be simply ignored by removing these entries from each W_k matrix. This function is achieved as part of the upper triangular operation which is performed after each intermediate BFS hop.

Detecting and removing cycles rooted at the b candidates requires additional steps demonstrated on line 3 of Algorithm 2. Intuitively, the algorithm takes a forward step on all of the *bi-directional* edges starting at its c candidates, W_2, and masks out any path that did not reach a vertex which is a potential b candidate, W_1. This newly computed term, $W_{a\underline{bcb}}$, provides the count of all 2-hop paths which start at a b candidate, pass through a c candidate and return back to the original b candidate. By removing $W_{a\underline{bcb}}$, W_3 accounts for all the the non-cyclical paths connecting the starting vertex candidates a and any of their respective d candidates.

2.1.3 Symmetry-breaking constraints

Symmetric query graphs have multiple automorphisms, and therefore, generate a multitude of redundant matches rotated around the symmetrical vertices. As explored in depth in [25], by applying constraints that ensure symmetric vertex candidates can only be considered in a certain order, the redundant matches can be ignored during the traversal. This concept is realized in Algorithm 2 using the upper-triangular operator. Since i and j matrix indices represent numerical identifiers, they provide a total order between all of the graph vertices. In the provided example on line 1 of Algorithm 2, the upper triangular operator enforces a less-than constraint between the rows and the columns; therefore, it specifies a simple way of only considering paths where the index of the starting vertex a is smaller than any of the considered candidates b. Applying this operator after each BFS hop, realizes the constraints necessary to remove all such redundant matches for a 4-cycle component.

As discussed in more detail in Section 3, this concept can be extended to more complicated query graphs constraints; however, it does require special consideration during the join and query planning phases.

2.2 COMPONENT ENUMERATION

After completing the component detection phase, W_k matrices contain the state necessary to reconstruct all of the unique paths matching the component subgraph. Specifically, the procedure begins at the last hop, W_K, which contains the count and end-vertices of all of the valid paths for each independent frontier. First, it works backwards to remove any forward path in all W_k matrices that did not directly contribute to a detection in W_{k+1}. Subsequently, for each hop, it creates a list

of the remaining vertices and leverages this state to expand and enumerate every individual path. Algorithm 3 and Algorithm 4 illustrate the pseudocode for the two-part reconstruction process for 4-cycle example presented in the previous section.

Algorithm 3 Component Enumeration (Part I): *Backwards Filter and Expansion of Partial Paths*

Output: row_k: association between partial paths from hop k to hop $k+1$; col_k: id of participating vertex for each partial path at hop k

Input: A: data graph adjacency matrix; W_k: walk matrices generated by the detection phase

1: $\{row_K, col_K\} = find(W_K)$
2: $f = row_K$
3: **for** $k \leftarrow \{K...1\}$ **do**
4: $\quad E = W_{k-1}(f, :) \circ A(:, col_k)$
5: $\quad \{row_{k-1}, col_{k-1}\} = find(E)$
6: $\quad f = f(row_{k-1})$
7: **end for**

Algorithm 4 Component Enumeration (Part II): *Forward Enumeration of Complete Paths*

Output: C: $P \times |V(Q)|$ matrix, where P is the number of overall matches. Each column in the matrix corresponds to a vertex of the query graph, and the values are the IDs of the vertices in the data graph which match the corresponding query vertex

Input: row_k: association between partial paths from hop k to hop $k+1$; col_k: id of participating vertex for each partial path at hop k

1: $P = [col_0]$
2: $r = row_0$
3: **for** $k \leftarrow \{1...K\}$ **do**
4: $\quad P = [P, col_k(r)]$
5: $\quad$ **if** $k < K$ **then**
6: $\quad\quad r = row_k(r)$
7: $\quad$ **end if**
8: **end for**

A full example for both parts of the enumeration algorithms is presented in Figure 4. It's important to note since the backward filter and expansion algorithm only considers the matrix W_k to inform the W_{k-1} expansion, it has the potential of generating invalid paths with repeating vertices. These paths can easily be detected and filtered in the second part of the enumeration.

2.3 MULTI-COMPONENT JOIN

Up to this point, the presented algorithms considered detection and enumeration of small components. In order to generalize this approach for support of arbitrary queries, multiple individual

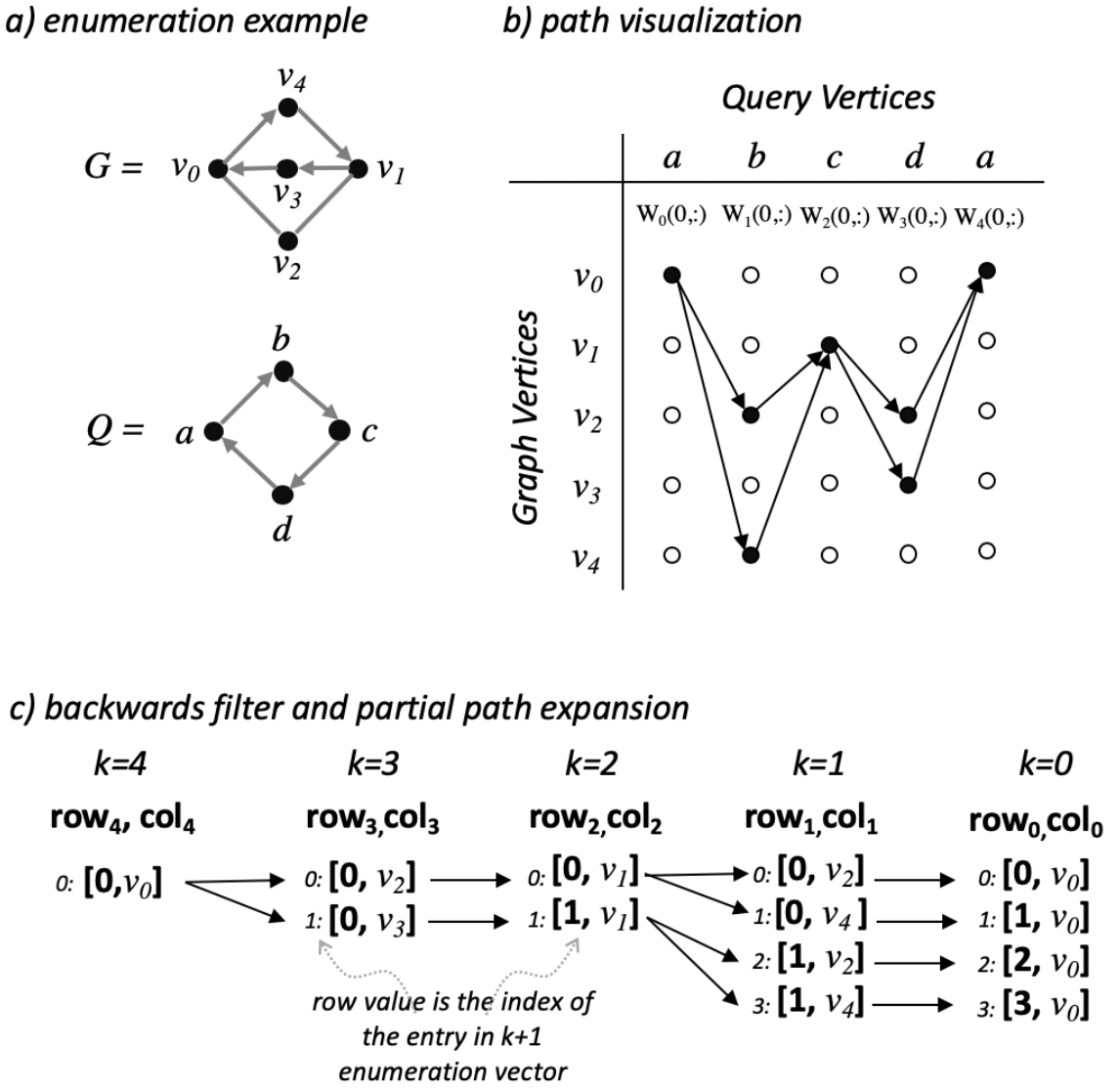


Figure 4. Complete enumeration example for all path starting at v_0. (NOTE: Other paths ignored for illustration purposes)

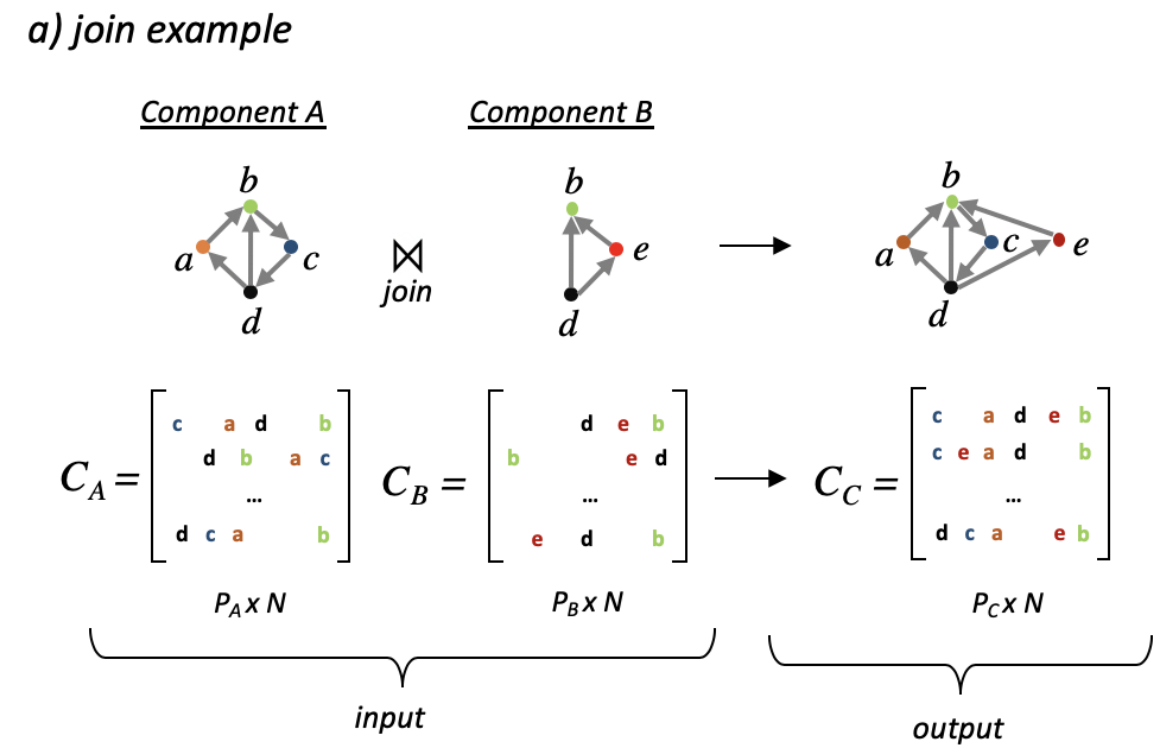
components can be joined together. As discussed earlier, the order and the component decomposition is identified during the planning phase. Once individual components are successfully enumerated, the join operation described in Algorithm 5 can be used to merge the results to realize the complete subgraph enumeration.

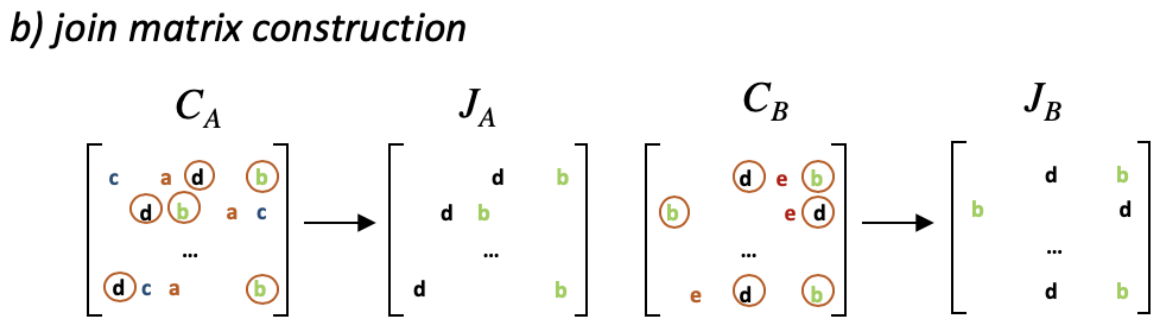
Algorithm 5 Component Join

Output: C_C: $P_C \times N$ matrix, where P_C is the number of overall matches after the join operation
Input: C_A: $P_A \times N$ matrix, where P_A is the number of overall matches for component A. C_B: $P_B \times N$ matrix, where P_B is the number of overall matches for component B.

1: $s = identifySharedVertices(C_A, C_B)$
2: $J_A = buildJoinMatrix(C_A, s)$
3: $J_B = buildJoinMatrix(C_B, s)$
4: $J_C = J_A * J_B^T$ // matrix multiply (+,*) operators are overloaded for (+, ==)
5: $k \leftarrow 0$
6: **for all** $J_C(i, j) ==$ # of shared vertices **do**
7: $\quad C_C(k, :) = join(s, C_A(i, :), C_B(j, :))$
8: $\quad k = k + 1$
9: **end for**

As illustrated in Figure 5, the output of the enumeration step is first converted to a $P \times N$ matrix where each row represents an individual match, each column indicates vertex participation in the match, and the value provides a mapping between the vertex in the data graph and its associated vertex in the original query subgraph. For example, for a four node component, such as C_A, each row would have four non-zero values for each query vertex match. Then, in order to perform the join, the algorithm first identifies the query vertices that are shared between two enumerated components and filters the input matrices to only contain candidates for the shared vertices (query vertices b and d in the provided example). The main objective of the join is to identify all the partial matches from the two components that share the data graph vertices and combine these matches to form the larger Q_C component. This operation is equivalent to a traditional database natural join and can be implemented using a simple matrix multiply. The matrix multiply provides a mathematical operation for an all-to-all comparison, the results of which are then used to perform the actual individual row-based joins for each resulting match. By replacing the traditional element-wise operator with an equivalency check, the matrix multiply ensures that the order of vertices is preserved during the match consideration. It is important to note that partial matches that are identified by the preceding operation only consider shared vertices, and therefore, the individual row-join operation may identify results which contain candidate vertices which are mapped to multiple query vertices. Since these matches would be considered invalid by definition, they are filtered by the join operation.





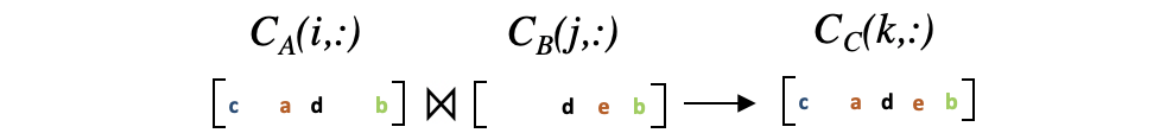


Figure 5. Visualization of the join operations and steps described in Algorithm 5.

3. DISCUSSION

This section discusses the fundamental advantages of the massive parallel walks using the matrix representation. It also provides thoughts and considerations that are relevant to the applicability of the presented algorithms to the general subgraph matching problem.

3.1 EFFICIENT STATELESS REPRESENTATION OF POTENTIAL MATCHES

The approach described in Section 2 is designed to leverage the highly efficient representation of walk matrices. Under this representation and using linear algebra operators, components are detected in a massively parallel fashion without enumerating the partial paths being considered, as typically done in subgraph matching using a tree representation. At each k step in the parallel walks, the partial paths being considered are represented minimally as counts in the walk matrix where each non-zero element $W_k(i, j)$ represents the number of such partial paths starting from vertex i and ending at vertex j. Note that a single count in this representation is able to denote multiple paths without enumerating them, increasing the efficiency even more.

This matrix-based representation is stateless in the sense that each walk matrix does not store information on all the vertices previously visited by the partial paths, except the starting vertex. However, the sequence of walk matrices, along with the adjacency matrix contain all the necessary information for enumerating the paths at a later stage (see Section 2.2). As demonstrated empirically in Section 4, most partial paths will not complete as detected components at the end of the walk, so there is much advantage to enumerate only the small fraction of paths that complete. This is the advantage under the stateless representation of walk matrices.

3.2 COMPONENT SIZE CONSIDERATIONS

The empirical results described in Section 4.2 suggest that a 3-hop walk to detect 4-vertex components provides a practical trade-off between component size and the density of the intermediate walk matrices. Nonetheless, an interesting consideration is whether the detection algorithm can be extended indefinitely to detect larger components. The simple answer is that the current approach is limited to only detecting 3-hop paths. The limitation stems from its greatest benefit which is the fact that it is stateless. W_k matrices aggregate path information, which means that compensation terms, such $W_{a\underline{b}c\underline{b}}$ in Algorithm 2, to remove all cycles need to be computed in order to continue the detection. Thus, for a 5-node component, terms $W_{\underline{a}bcd\underline{a}}$, $W_{a\underline{b}cd\underline{b}}$ and $W_{ab\underline{c}d\underline{c}}$ are required. Since there is only one candidate G vertex for the a query vertex per frontier (i.e., a row of W_k), removing all of the diagonal entries would compensate for the $W_{\underline{a}bcd\underline{a}}$ cycles. Similar to $W_{a\underline{b}c\underline{b}}$, $W_{ab\underline{c}d\underline{c}}$ can be computed using

$$W_{ab\underline{c}d\underline{c}} = (\lfloor W_3 \rfloor \cdot (A \circ A^T)) \circ W_2. \tag{3}$$

However, in order to compute $W_{a\underline{b}cd\underline{b}}$, the algorithm has to be able to determine which b candidates in W_1 contributed to which d candidate in W_3. Because of the intrinsic aggregation of the

matrix multiplication, this information cannot be recovered from the walk matrices without explicitly enumerating the individual paths. We could introduce compensation terms that leverage multi-hop association matrices, which can be computed through powers of the adjacency matrix, to help identify these relationships. However, as the walk length increases, the density of the compensation matrices will increase, ultimately making storage and computation of these terms intractable.

3.3 HOP-SEQUENCE ORDER

The component detection process requires that the traversal consists of a continuous, uninterrupted walk starting at the candidates for the initial query vertex and ending at the candidates for the final vertex. Each matrix multiply by the adjacency matrix, A, represents an edge traversal from the previously discovered candidates to identify potential new candidates for the next vertex in the query. In the example illustrated in Algorithm 2, we can set W_0 to the identity matrix, which would mean that all vertices of the data graph are potential candidates for being a match for the query vertex a. Through matrix multiplication, we can, in parallel, traverse the edges connected to each of these vertices to discover potential candidates for the b vertex. Consequently, by extending this 1-hop-based process, we can identify candidates for all of the vertices in the query.

As discussed in Section 3.2, any edge that connects candidates between multiple hops would require multi-hop association matrices. However, by making the observation that there is never an ambiguity for which a candidate contributed to which c candidates, since there is only one a candidate considered by each parallel frontier, the adjacency matrix captures all of the required state to identify these edges. Thus, by simply changing the traversal order to allow 2-hop association edges to only connect to the a candidate, this approach can be applied to find all, but one, 4-node components. Figure 6 demonstrates several potential hop sequences that *cannot* be supported using a single walk. Specifically, each of these hop-sequences requires information that disambiguates which b candidate contributed to which d candidate. Other than the 4-clique subgraph, all other components can be detected through simple adjustment of the hop sequence order, as illustrated previously in Figure 2.

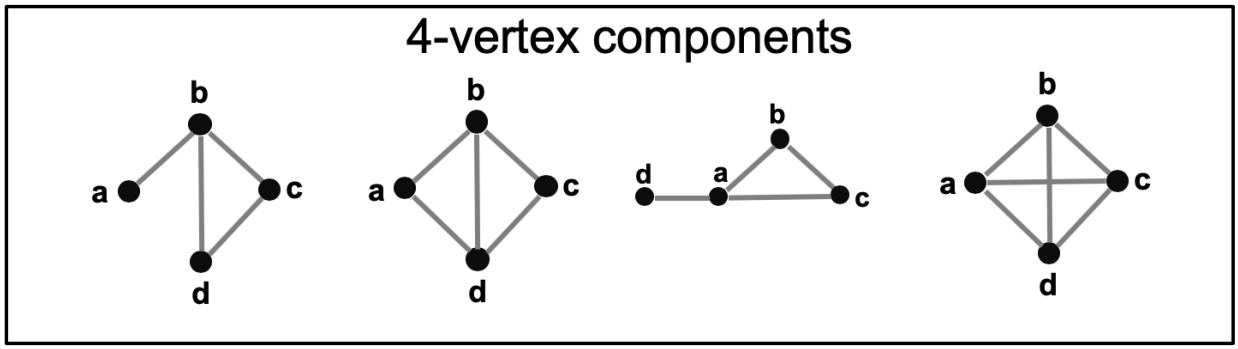


Figure 6. Example of hop-sequence that can be supported only with a multi-component join.

No hop-sequence combination can remove the discussed ambiguity from a 4-clique subgraph; therefore, it is not a detectable component. It is important to note that it does not prevent a clique from being a supported query graph, since it can be easily decomposed into any number of supported components which can be joined to find any arbitrary subgraph.

3.4 UTILIZING STATE FROM MULTIPLE COMPONENTS

Component joins mitigate the limitation on the size and complexity of the component detectable using the matrix-based parallel walks, but treating each component detection independently may cause redundant explorations, because it is not fruitful to search for later components where earlier components already failed to match. This can be mitigated by seeding the parallel walks for detecting later components with only the starting vertices that have already matched in earlier components. In this way, as more of the subgraph components have been detected and joined, fewer starting vertices will be considered for later component detection, focusing the search only on parts of the graph with potential complete matches.

3.5 LEVERAGING ATTRIBUTES

A general challenge for subgraph matching is the overwhelming number of potential matches, especially early in the process when the matched parts are not very unique. Our approach is not immune to this fundamental challenge either. Other than speeding up the component detection with efficient massive parallel walks, our approach can be further enhanced with the availability of vertex and edge attributes. Matching with graph attributes can eliminate a large fraction of potential matches by filtering with attribute constraints to reduce the qualifying edges and nodes at each step of the parallel walk.

3.6 SYMMETRY-BREAKING FOR LARGER QUERY GRAPHS

Applying symmetry-breaking constraints is an effective technique for detecting redundant matches. Several important considerations have to be taken into account in order to leverage these ideas for a subgraph matching methodology which decomposes the query graph into smaller partial subqueries. First, the individual subqueries can be symmetric. This property can be identified during the planning phase, and thus, specialized constraints can be generated and applied in the intermediate matching steps in order to minimize the storage and computation requirements for the partial results. At the same time, during the join phase, all rotations would still need to be considered and expanded in order to verify the existence of any valid join opportunities between the component and any other partial match.

Second, similar to component path exploration discussed in Section 3.3, applying multi-hop constraints requires specialized planning and query decomposition. As demonstrated in Algorithm 2, during the detection phase, applying the upper-triangular operator to the walk matrices, W_k, after each hop allows a simple way to apply constraints that involve the a candidate for each component.

Applying constraints involving other vertices may be achieved during or after any completed join operation on fully-enumerated partial matches.

4. EVALUATION

The primary workhorse of the proposed graph matching approach is the linear-algebra-based component detection using massively parallel walks. This section evaluates and demonstrates the advantages of the proposed algorithm to efficiently detect component matches without enumerating partial matches, over a range of realistic graph topologies via simulation. Quantitative evaluation is performed to reveal the computational and storage requirements as well as savings on the densest walk matrix (i.e., W_3 in Algorithm 2) which dominates the requirements. Specifically, statistics are reported on: 1) the number of partial products which correspond to the number of edge traversals to extend partial paths in the parallel walk, 2) the density of the densest walk matrix, 3) the fraction of partial paths completed as detected components, and 4) the scale of compression by representing partials paths as counts in the walk matrix.

4.1 SIMULATING REALISTIC GRAPH TOPOLOGIES

We use the degree-corrected stochastic blockmodel [26] with an additional density parameter [27] to simulate networks across a range of realistic topological characteristics: 1) network density, 2) Power-Law degree distribution, and 3) strength of community structure. For rapid exploration over the graph topological features, evaluation is done on small data graphs of $N = 1000$ vertices. We will scale up in the identified topological regime of interest as future work. The query component used for this evaluation is the directed 4-cycle, with the exact detection steps shown in Algorithm 2.

Under this generative model, each edge, a_{ij}, is drawn from a Poisson distribution of rate λ_{ij} governed by the equations below:

$$a_{ij} \sim \text{Poisson}(\lambda_{ij}) \tag{4}$$

$$\lambda_{ij} = I_{ij} \times \theta_i \theta_j \times \Omega_{b_i b_j} \tag{5}$$

Each edge a_{ij} represents the count or strength of interaction, which is binarized to render the data graph for subgraph matching. The edge switch $I_{ij} \sim \text{Bernoulli}(s)$ is parameterized by s which controls the overall density of the network. Note, however, that the density parameter s is not the realized graph density (i.e., edge count divided by N^2) which is affected also by the other parameters. The vertex degree term θ_i adjusts node i's expected degree, drawn from a Power-Law distribution with an exponent α between -3 and -2 to capture the degree distribution of realistic, scale-free graphs [28]. The strength of community structure is governed by the block matrix Ω where strong communities are formed when the diagonal elements (i.e., within-community strength) are much larger than the off-diagonal elements (i.e., between-community strength). The community (i.e., block) assignment for each node b_i is drawn from a multinomial distribution with equal probabilities. The probabilities of community assignment can be drawn from a Dirichlet distribution for varying levels of community size heterogeneity, for future evaluations. These parameters serve as "knobs" that can be dialed to capture a rich range of realistic network topologies. Figure 7 shows an example data graph generated with the baseline parameter setting: density parameter $s = 0.3$, Power-Law

exponent $\alpha = -0.3$, and 5 times more within-community edges than between-community edges. For each parameter setting, results are reported on 10 realizations of data graphs for statistical support (i.e., standard deviation bars in Figure 8 through 16).

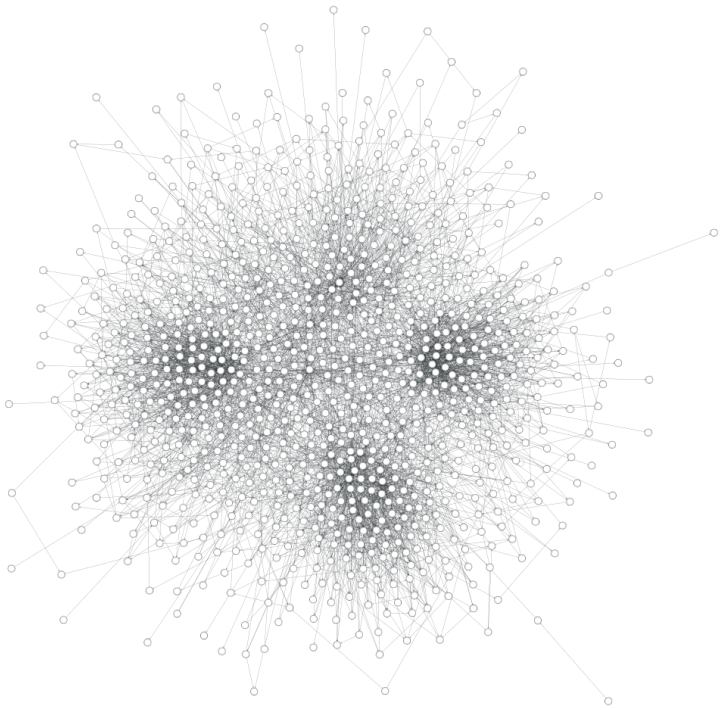
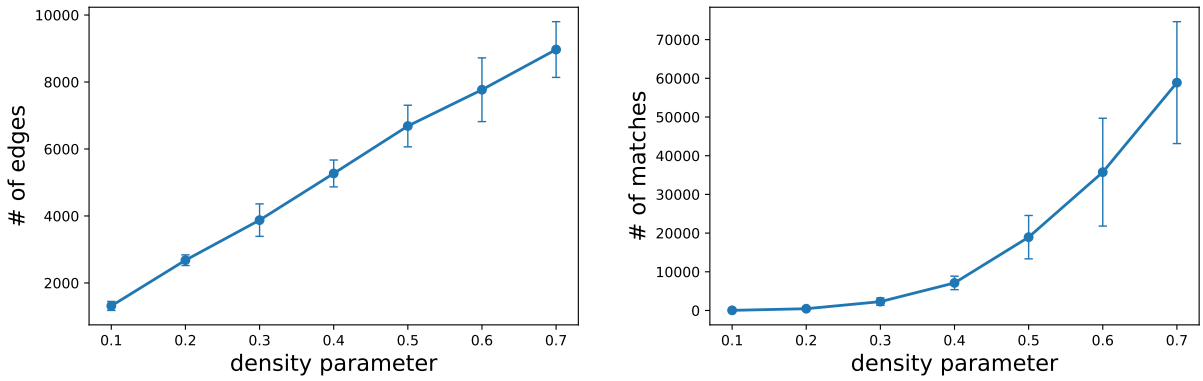


Figure 7. Baseline simulated data graph for evaluation, with density parameter $s = 0.3$, Power-Law exponent $\alpha = -0.3$, and 5 times more within-community edges than between-community edges.

4.2 EVALUATION OVER GRAPH DENSITY

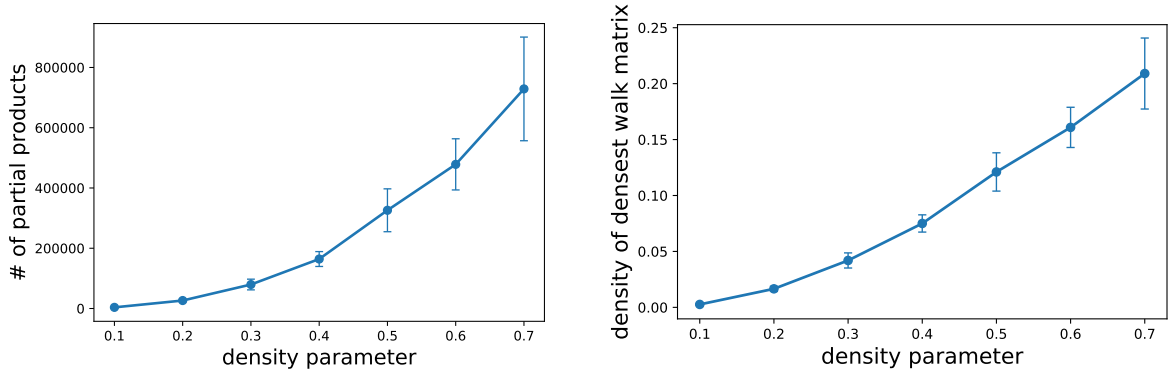
We first evaluate the behaviors of the proposed detection algorithm over varying levels of data graph density, for a directed 4-cycle component (see Algorithm 2). As captured by the subsequent figures, we demonstrate the rate with which the density and computational requirement increase for the realization of the densest walk matrix. The resulting statistics highlight the key advantages of the algorithm to efficiently detect component matches without enumerating partial results.



(a) (b)

Figure 8. Growth on the number of edges and matches as the density parameter increases.

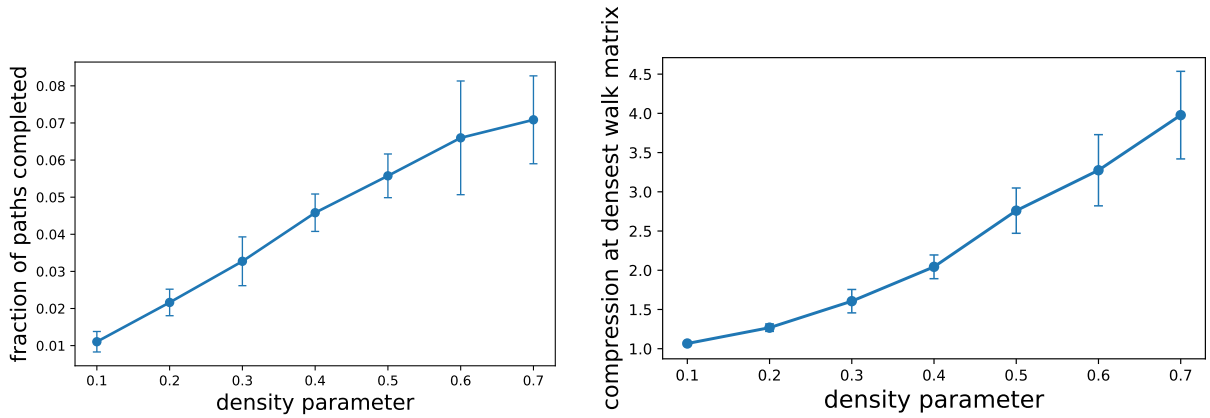
Figure 8a shows the linear growth on the number of edges as the density parameter increases. Not surprisingly, the number of matches for the 4-cycle component grows exponentially as the dense regions produce a large number of matches, shown in Figure 8b. This exponential growth in matches also results in much higher numbers of partial products required for computing the walk matrices and their density, as shown in Figure 9a and 9b for the densest walk matrix (i.e., W_3 in Algorithm 2). This increased computational and storage demand makes it difficult for any matching algorithm to scale up for large data graphs. We will focus our evaluation going forward at the graph density produced by density parameter $s = 0.3$ in this simulation. Note that for scaling up evaluation on larger graphs, this density parameter will be adjusted for a graph of N vertices to grow on the order of $N \log N$ in the number of edges, commonly known as the edge growth rate for the graph to remain connected [29].



(a) (b)

Figure 9. Number of partial products and density of the densest walk matrix as the graph gets denser.

A key advantage of our proposed component detection algorithm using stateless walk matrices is that it avoids enumerating partial paths, unlike conventional tree-based matching algorithms. Paths are enumerated later only on completed matches. Figure 10a shows the significance of this saving, as only a small fraction of partial paths at the densest walk matrix (i.e., W_3 in Algorithm 2) are completed. At the operating graph density of $s = 0.3$, our algorithm avoids enumeration of about 97% of partial paths that do not complete a 4-cycle. Another significant advantage is the compression of multiple paths sharing the same source and destination in the walk matrices as a single count, as discussed in Section 3.1. Figure 10b shows the scale of compression (i.e., average number of paths compressed in a single count) at the densest walk matrix (i.e., W_3), which is around $\times 1.6$ at the operating graph density of $s = 0.3$.

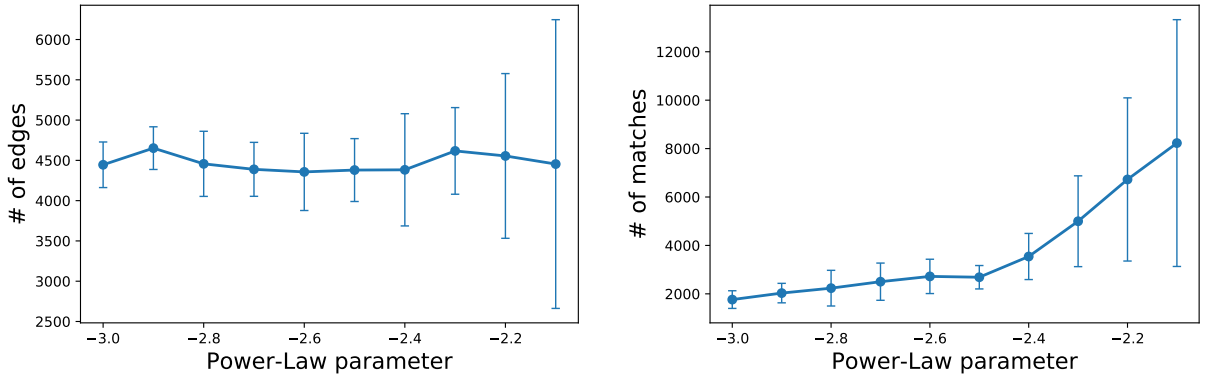


(a) (b)

Figure 10. Fraction of partial paths completed and the compression scale at the densest walk matrix as the density parameter increases.

4.3 EVALUATION OVER VERTEX DEGREE DISTRIBUTION

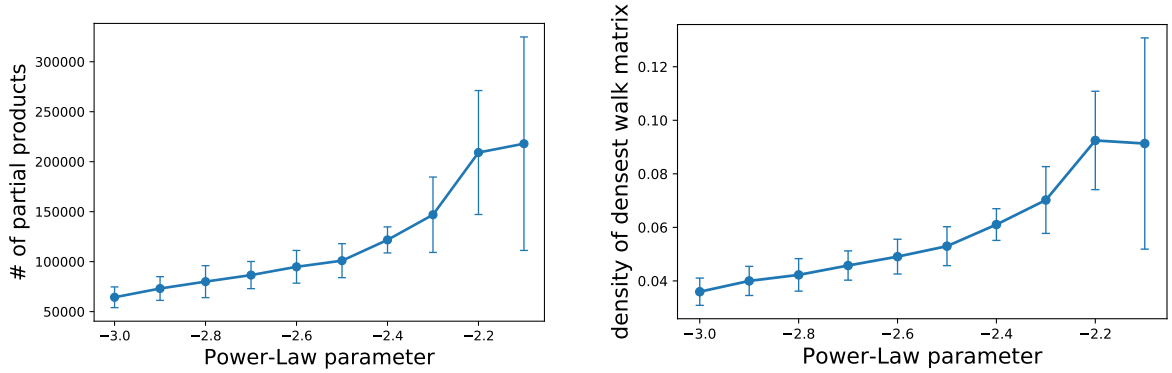
Another important graph topological feature to vary is the number of high degree nodes, accomplished by varying the Power-Law exponent α from the realistic range of -3 to -2 [28]. With a higher exponent, the degree distribution decays slower, resulting in more high degree nodes. For meaningful comparison, the average number of edges (i.e., graph density) is fixed, as shown in Figure 11a. Nevertheless, due to the increased variance in the heavy tail degree distribution at higher α, there is more variations in the number of edges. Overall, Figure 11b shows a graceful, non-exponential, increase in the number of matches from more concentrated edges around the super vertices.



(a) (b)

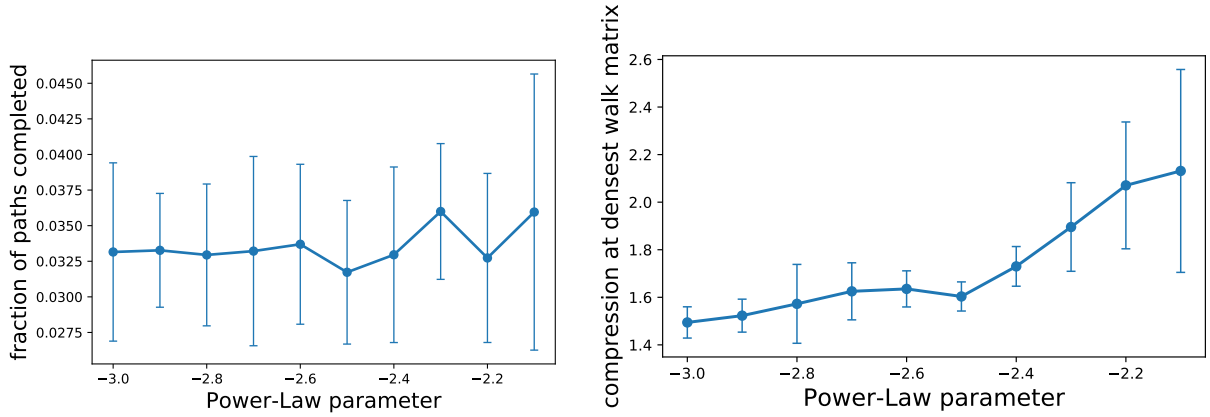
Figure 11. The number of edges and component matches as the Power-Law exponent increases to have more high degree vertices.

The computational and storage demand also increases gracefully, as shown in Figure 12a and 12b. At the highest exponent, the growth actually slows down due to the increased compression scale shown in Figure 13b. This effect is not surprising, because more higher degree nodes will generate more paths that share the same sources and destinations, and they are compressed as counts in the walk matrices representation. Lastly, Figure 13a shows the completion rate of partial paths remains quite low around 3.4%, demonstrating again the advantage of the proposed component detection algorithm. For evaluation on graph density and strength of community structure, we use the baseline value of $\alpha = -2.5$ on the Power-Law exponent.



(a) (b)

Figure 12. Number of partial products and density of the densest walk matrix as the Power-Law exponent increases to have more high degree vertices.

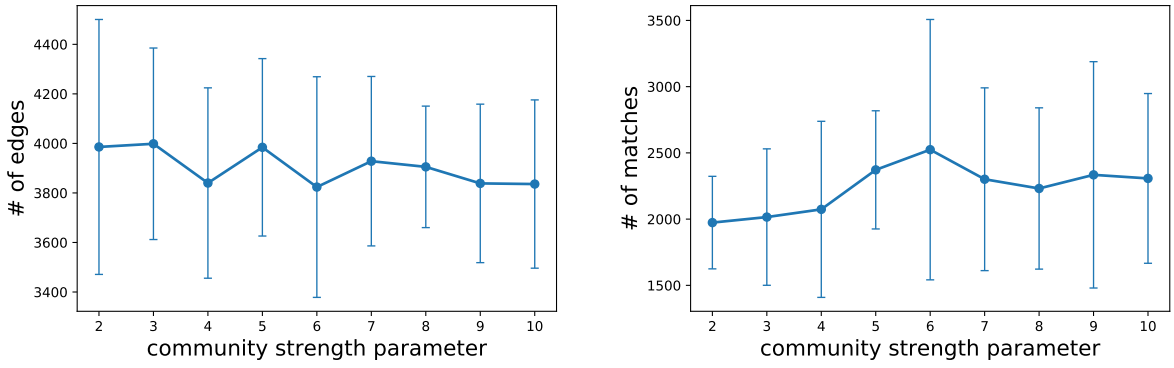


(a) (b)

Figure 13. Fraction of partial paths completed and the compression scale at the densest walk matrix as the Power-Law exponent increases to have more high degree vertices.

4.4 EVALUATION OVER STRENGTH OF COMMUNITY STRUCTURE

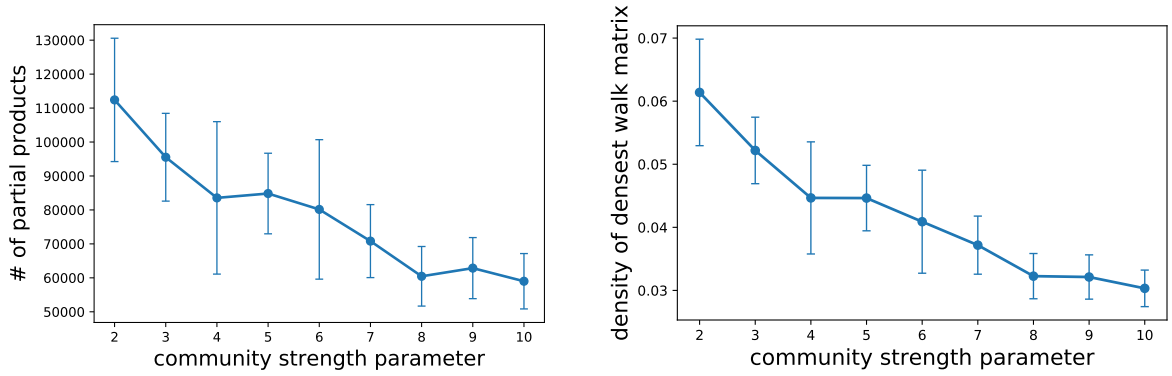
Community structure is another important characteristics in real-world graphs, as vertices from the same communities interact more strongly with one another than those between different communities. Here we evaluate the effects of the community structure on the behaviors of the component detection algorithm. The strength of the community structure is increased gradually, parameterized by the ratio of within-community edges to between-community edges. For meaningful comparison, the average number of edges (i.e., graph density) is fixed, as shown in Figure 14a. The number of matches in Figure 14b has a gradual but statistically insignificant increase, likely due to more cycles being formed with more concentrated edges within communities.



(a) (b)

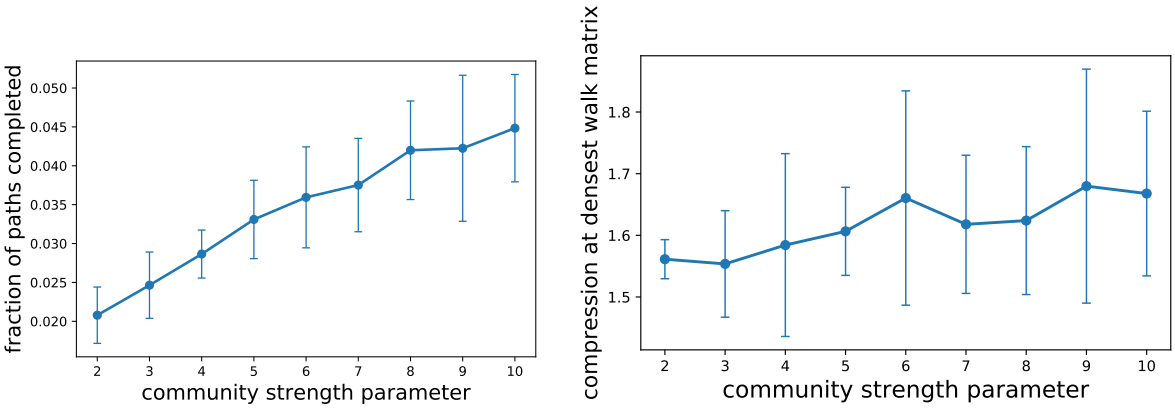
Figure 14. The number of edges and component matches as the strength of community structure increases.

Figure 15a and 15b show decreased computational and storage requirement on the walk matrices with stronger community structure, from a lower number of partial paths being generated and slightly higher compression scale as shown in Figure 16b. With stronger community structure, the paths being considered tend to be contained within-communities, resulting in fewer and more concentrated partial paths. These more concentrated partial paths also have a higher completion rate, with stronger community structure, as shown in Figure 16a. The completion rate remaining below 5% (see Figure 16a), together with a compression scale at around 1.65, demonstrate the advantages of the proposed component detection algorithm.



(a) (b)

Figure 15. Number of partial products and density of the densest walk matrix as the strength of community structure increases.



(a) (b)

Figure 16. Fraction of partial paths completed and the compression scale at the densest walk matrix as the strength of community structure increases.

This page intentionally left blank.

5. SUMMARY

In this paper, we introduce and motivate a new, multi-stage subgraph matching algorithm that leverages matrix-based graph traversal for identifying matches. This formulation provides a stateless discovery mechanism to explore the graph starting at multiple vertices simultaneously and performing this exploration in a massively parallel fashion. We present a preliminary analysis of the approach and demonstrate key advantages of the proposed formulation. In the future, we will be looking to map the algorithm onto different state-of-the-art linear algebra accelerator technologies, as well as extend the analysis and evaluation to much larger graph datasets.

This page intentionally left blank.

REFERENCES

[1] S.A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, New York, NY, USA: ACM (1971), STOC '71, pp. 151–158, URL http://doi.acm.org/10.1145/800157.805047.

[2] T. Davis, "Algorithm 9xx: Suitesparse: Graphblas: graph algorithms in the language of sparse linear algebra," *ACM Trans. on Mathematical Software* (2019).

[3] A. Buluç and J.R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.* 25(4), 496–509 (2011), URL http://dx.doi.org/10.1177/1094342011403516.

[4] W.S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE (2016), pp. 1–7.

[5] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*, vol. 22, SIAM (2011).

[6] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J.D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (2016), pp. 1–9.

[7] A. Azad and A. Buluç, "Distributed-memory algorithms for maximal cardinality matching using matrix algebra," in *2015 IEEE International Conference on Cluster Computing* (2015), pp. 398–407.

[8] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop* (2015), pp. 804–811.

[9] T.M. Low, D.G. Spampinato, A. Kutuluru, U. Sridhar, D.T. Popovici, F. Franchetti, and S. McMillan, "Linear algebraic formulation of edge-centric k-truss algorithms with adjacency matrices," in *2018 IEEE High Performance extreme Computing Conference (HPEC)* (2018), pp. 1–7.

[10] M.M. Wolf, M. Deveci, J.W. Berry, S.D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (2017), pp. 1–7.

[11] T. Mattson, T.A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "Lagraph: A community effort to collect graph algorithms built on top of the graphblas," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE (2019), pp. 276–284.

[12] F. Jamour, I. Abdelaziz, Y. Chen, and P. Kalnis, "Matrix algebra framework for portable, scalable and efficient query engines for rdf graphs," in *EuroSys* (2019).

[13] G. Sundaram and S.S. Skiena, "Recognizing small subgraphs," *Networks* 25(4), 183–191 (1995).

[14] J. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)* 23, 31–42 (1976).

[15] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26(10), 1367–1372 (2004).

[16] H. He and A. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2018* (2008), pp. 405–418.

[17] I. Almasri, X. Gao, and N. Fedoroff, "Quick mining of isomorphic exact large patterns from large graphs," *IEEE International Conference on Data Mining Workshops, ICDMW* 2015, 517–524 (2015).

[18] X. Ren, J. Wang, W.S. Han, and J.X. Yu, "Fast and robust distributed subgraph enumeration," *Proc. VLDB Endow.* 12(11), 1344–1356 (2019), URL https://doi.org/10.14778/3342263.3342272.

[19] V. Carletti, P. Foggia, P. Ritrovato, M. Vento, and V. Vigilante, "A parallel algorithm for subgraph isomorphism," in D. Conte, J.Y. Ramel, and P. Foggia (eds.), *Graph-Based Representations in Pattern Recognition*, Cham: Springer International Publishing (2019), pp. 141–151.

[20] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proceedings of the VLDB Endowment* 8(10), 974–985 (2015).

[21] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proceedings of the VLDB Endowment* 10(3), 217–228 (2016).

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

[23] T. Reza, M. Ripeanu, N. Tripoul, G. Sanders, and R. Pearce, "Prunejuice: Pruning trillion-edge graphs to a precise pattern-matching solution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, Piscataway, NJ, USA: IEEE Press (2018), SC '18, pp. 21:1–21:17, URL https://doi.org/10.1109/SC.2018.00024.

[24] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing surveys (CsUR)* 16(2), 111–152 (1984).

[25] J.A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *RECOMB* (2007).

[26] B. Karrer and M.E. Newman, "Stochastic blockmodels and community structure in networks," *Physical Rev. E* 83(1), 016107 (2011).

[27] E.K. Kao, S.T. Smith, and E.M. Airoldi, "Hybrid mixed-membership blockmodel for inference on realistic network interactions," *IEEE Transactions on Network Science and Engineering* (2018).

[28] A.L. Barabási, "Scale-free networks: a decade and beyond," *science* 325(5939), 412–413 (2009).

[29] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci* 5(1), 17–60 (1960).

This page intentionally left blank.