CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching

Bibek Bhattarai
George Washington University
bhattarai_b@gwu.edu

Hang Liu*
University of Massachusetts Lowell
hang_liu@uml.edu

H. Howie Huang
George Washington University
howie@gwu.edu

ABSTRACT

Subgraph matching finds all distinct isomorphic embeddings of a query graph on a data graph. For large graphs, current solutions face the scalability challenge due to expensive joins, excessive false candidates, and workload imbalance. In this paper, we propose a novel framework for subgraph listing based on Compact Embedding Cluster Index (CECI), which divides the data graph into multiple embedding clusters for parallel processing. The CECI system has three unique techniques: utilizing the BFS-based filtering and reverse-BFS-based refinement to prune the unpromising candidates early on, replacing the edge verification with set intersection to speed up the candidate verification, and using search cardinality based cost estimation for detecting and dividing large embedding clusters in advance. The experiments performed on several real and synthetic datasets show that the CECI system outperforms state-of-the-art solutions on average by 20.4× for listing all embeddings and by 2.6× for enumerating the first 1,024 embeddings.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Theory of computation → Graph algorithms analysis**; *Pattern matching*; *Sorting and searching*; • **Software and its engineering → Massively parallel systems**;

KEYWORDS

subgraph listing; subgraph matching; graph pattern matching; subgraph isomorphism; CECI; extreme cluster

* Work done at George Washington University.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300086

ACM Reference Format:
Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In 2019 International Conference on Management of Data (SIGMOD'19), June 30–July 5, 2019, Amsterdam, Netherlands. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3300086

1 INTRODUCTION

Given a query graph G_q and a data graph G, *subgraph listing* is a problem of finding all the subgraphs of G that are isomorphic to G_q. As an example in Figure 1, subgraph listing for the query graph $\{u_1, u_2, u_3, u_4, u_5\}$ finds two possible matches in the data graph, i.e., $\{v_1, v_3, v_4, v_{11}, v_{12}\}$ and $\{v_1, v_5, v_6, v_{13}, v_{14}\}$. Note that this paper uses the terms subgraph matching and subgraph listing interchangeably. This NP-hard problem is important in many application domains including sub-compound search in chem-informatics [54], analysis of protein-protein interaction networks [44], computer aided design [41], and graph pattern mining [1, 51]. Although many techniques such as search order optimization [19, 46, 59], join cardinality reduction [4, 17, 50], and index based filtering [52, 57–59] have been developed in recent times, subgraph matching for large graphs still faces three main challenges.

First, scalability has been an Achilles heel for existing solutions, the majority of which adopt one of the two approaches, *one embedding at a time listing* [4, 16, 17, 42, 54] or *all embeddings at once listing* [2, 32, 47]. An **embedding** is a unique subgraph of the data graph that matches the query graph. While the former requires small amount of memory for intermediate results, this strictly sequential approach is not viable for larger graphs. Instead of listing all embeddings, most current solutions return the first 1,000 or so embeddings. On the other hand, relying on parallel computing frameworks such as MapReduce, the latter approach aims to process all the embeddings concurrently. However, exponential memory consumption and expensive join operation makes their performance vastly unsatisfactory.

Second, existing algorithms spend a considerable amount of time working on false candidates that yield no embeddings in the end. For example, *join based listings* [2, 32, 50]

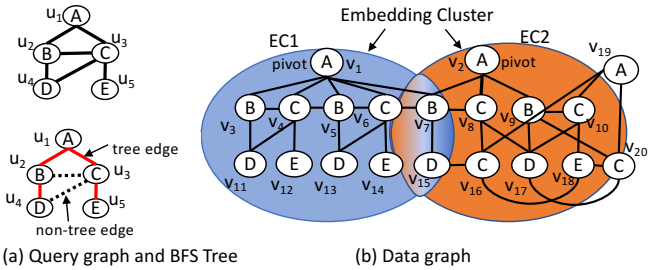


Figure 1: Query graph, data graph and embedding clusters. The solid lines in the BFS tree of the query graph are tree edges and the dotted lines non-tree edges.

decompose the query graph into several sub-patterns, collect the candidates for each sub-pattern, and join them to form the complete embedding. This approach suffers from false cardinality because it cannot limit the join operations to true candidates only [50]. In contrast, *exploration based approaches* [10, 47, 50, 54] list the embeddings by exploring the data graph from predetermined nodes in a specific order. While it avoids the false cardinality associated with joins, this approach suffers from exploring false search paths, leading to incomplete backtracking branches especially when the query graph becomes non-trivial.

The third challenge arises from the power law nature of the real world graphs. This imposes a huge disparity in workload distribution among different machines while parallelizing the solution. Among several projects that have attempted to address this issue, PsgL [47] chooses a worker machine for each intermediate embedding after every expansion, which is an overkill in the sense that it needs to calculate the cost and make a machine selection for every incomplete embedding. In comparison, TTJ [32] uses a node-degree-based threshold to reduce the workload skew. This approach is less costly, however, node degree alone is insufficient to achieve insightful workload estimation. We believe that a workload balancing method that provides a good trade-off between computation overhead and workload balance is of essence to scale subgraph listing.

To address these challenges, we utilize a new concept of **embedding cluster**, which is a special subgraph of the data graph. An embedding cluster contains a group of the embeddings which share the same data node (***pivot node***) as a match of the first query node for a given matching order. For simplicity, we have used the Breadth-first search (BFS) order as the matching order, but our methods work with any other matching orders. In this paper, we use the terms node and vertex interchangeably, and data node and query node are used to refer to a node in the data and query graph, respectively. A data graph may have a number of embedding clusters and each cluster may contain several distinct embeddings. For example, in Figure 1, there are two

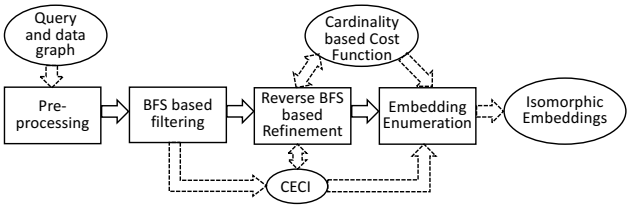


Figure 2: Framework Overview. The solid arrow represents the workflow and the dotted arrow shows the interaction between data structure (in oval) and processing steps (in box).

embedding clusters, $EC1$ and $EC2$, where the embeddings in $EC1$ match the pivot node v_1 to the query node u_1, and in $EC2$, v_2 to u_1.

Traditionally to enumerate the embeddings, one first collects the candidate set for each query node u, i.e. nodes in the data graph that can be matched to u in an isomorphic embedding. For example, $\{v_1, v_2\}$ for u_1, $\{v_3, v_5, v_7, v_9\}$ for u_2, $\{v_4, v_6, v_8, v_{10}\}$ for u_3, and so on. Next, each candidate of the query nodes is joined with the candidates of adjacent query nodes, following the matching order in a backtracking fashion. However, even though node v_1 of the data graph is connected to only three candidates $\{v_3, v_5, v_7\}$ of u_2 and two candidates $\{v_4, v_6\}$ of u_3, the backtrack search tree still tries to join with all four candidates of both u_2 and u_3. This leads to a search cardinality of 32, i.e, $(4 \times 4 \times 2)$.

With the help of embedding clusters, we can now separate the candidates of the query nodes by different pivot nodes, reducing the overall search cardinality of subgraph listing. For the same example, two embedding clusters $EC1$ and $EC2$ yield a cardinality reduction of more than three times to 10, i.e., $(3 \times 2 + 2 \times 2)$. Another benefit of embedding clusters is that such clusters naturally become many smaller local computations, which can be worked upon concurrently.

To incorporate this idea, we design an auxiliary data structure, **Compact Embedding Cluster Index** (CECI)[1], which represents all the embedding clusters and is used to facilitate parallel embedding enumeration. Each node in CECI contains all the data nodes that can be matched with a given query node. The size of CECI is polynomial to the size of the data and query graphs, i.e. $O(|E_q| \times |E_g|)$. CECI not only enables listing all the embeddings in parallel but also reduces the number of unpromising candidates significantly.

Figure 2 shows the workflow of our subgraph listing framework. Two main steps are *logical decomposition of the data graph into embedding clusters* with BFS order based filtering and reverse BFS based refinement to reduce the amount of false candidates, and *parallel embedding enumeration* by searching different embedding clusters concurrently. The

[1]CECI is used to represent the data structure or whole system depending on the context. When needed, the *CECI system* is used to represent the system

work distribution is equipped with proactive workload balancing based on cardinality based cost function. We characterize our approach as *k embeddings at a time listing*, where k different workers enumerate isomorphic subgraphs in parallel from different parts of the data graph, i.e., different embedding clusters.

The evaluation result on a large variety of real and synthetic datasets shows that our method is able to significantly outperform the state-of-the-art algorithms. Specifically, it computes all isomorphic embeddings faster than DualSim [24] and PsgL [47], on average by 6.08× and 34.82× respectively. Also, it outperforms CFLMatch [4] and TurboIso [17] by 2.7× and 2.72× to enumerate the first 1,024 embeddings.

The rest of the paper is organized as follows. Section 2 introduces the problem and the preprocessing steps. Section 3 discusses the CECI creation and refinement, and Section 4 presents parallel embedding enumeration. In Section 6, the experimental results are discussed. Section 7 presents the related work and Section 8 concludes.

2 BACKGROUND

2.1 Problem Statement

A graph can be represented as $G = (V, E, L)$ where V is the set of vertices, $E \subseteq (V \times V)$ is the set of all edges and L is the function that assigns one or more labels to each vertex from the set Σ of labels. A graph $G_{sg} = (V_{sg}, E_{sg}, L_{sg})$ is a subgraph of G if and only if there exists an injective mapping $\mu : V_{sg} \rightarrow V$, such that $(v_1, v_2) \in E_{sg}$ only if $(\mu(v_1), \mu(v_2)) \in E$. Further, a subgraph G_{sg} of data graph G is isomorphic to the query graph $G_q = (V_q, E_q, L_q)$, if and only if there exists a bijective function $f : V_q \rightarrow V_{sg}$ such that $\forall u \in V_q, L_q(u) \subseteq L_{sg}(f(u))$ and $\forall (u_i, u_j) \in E_q, (f(u_i), f(u_j)) \in E_{sg}$. All the query graphs are assumed to be a connected, undirected graph, and the data graph can be directed or undirected. The subgraph listing problem is to find all distinct subgraphs G_{sg} of the data graph G that are isomorphic to a query graph G_q.

2.2 Preprocessing

Preprocessing is crucial to subgraph listing as it can significantly reduce the memory consumption, as well as the total run-time of the algorithm, up to several orders of magnitude. The necessary preprocessing tasks for subgraph listing are described below.

Finding the root query node: Root query node u_s is the query node from which one starts the matching process. Intuitively, one prefers the root node to have the smallest number of matches to minimize the number of embedding clusters. Similar to TurboIso [17], we choose the root vertex u_s based on the cost function $u_s \leftarrow \underset{u}{\operatorname{argmin}} \frac{|candidate(u)|}{degree(u)}$ where $candidate(u)$ is all the candidates of u in the data graph G, and $degree(u)$ is the number of edges of node u. The candidate list of u is obtained by verifying each data node by the label, degree, and neighborhood label count (discussed in Section 3.2). In Figure 1, the costs for u_1, u_2, u_3, u_4, and u_5 are 1, 1.33, 1.25, 2, and 3 respectively. Hence u_1 is chosen as the root query node.

Generating the query tree: we perform a BFS traversal of the query graph starting from the root query node u_s to create a query tree. BFS has been obvious choice for the query tree in several existing works [4, 17] because they represent the topology of the query graph accurately and minimize the diameter of the search space. The edges from the query graph that are also present on the query tree are called **tree edges (TE)**. Each tree edge connects a parent node with a child node, where the former appears earlier than the latter in the BFS order. If an edge is on the query graph but not on the BFS tree, it is called **non-tree edge (NTE)**. The BFS query tree shown in Figure 1(a) has four tree edges, (u_1, u_2), (u_1, u_3), (u_2, u_4), and (u_3, u_5), as well as two non-tree edges (u_2, u_3) and (u_3, u_4).

Determining the matching (visit) order, i.e., the sequence of the query nodes to follow when matching them to the data nodes. Several heuristics have been developed over the years such as the least frequent node first [46], the least frequent path first [59], locally optimized order for each exploration [17], and dense region first [4]. Generally speaking, the order puts more selective query nodes, i.e., the one with fewer candidates earlier to lower the size of intermediate result sets. Also, the next node in order is selected from the neighbors of already selected nodes such that the search space can be limited.

For the simplicity of illustration, BFS traversal order of the query graph from the root node, i.e., $(u_1, u_2, u_3, u_4, u_5)$ in Figure 1 is used. Nonetheless, the techniques we have developed in this work can easily adopt other matching orders without the need for a major modification. In our experiments, adopting edge-ranked visit order [53] or path-ranked order [17] provided up to 34.5% speedup over using naive BFS matching order. The improvement is more significant on larger query graphs.

Breaking automorphism: When multiple query vertices are symmetric to each other, an identical embedding can be listed repeatedly. Formally, such repeated listings of the same embedding are called automorphisms. For example, an embedding of QG1 in Figure 6(a) can be listed for six times:{{0, 1, 2}, {0, 2, 1}, {1, 2, 0}, {1, 0, 2}, {2, 0, 1}, and {2, 1, 0}}.

To list each embedding only once, we have combined the concepts proposed by TurboIso [17] for finding *NEC-equivalence group* with *ordering based symmetry breaking rules* proposed by [16]. First, we explore the equivalence rules advocated by TurboIso to find similar query vertices

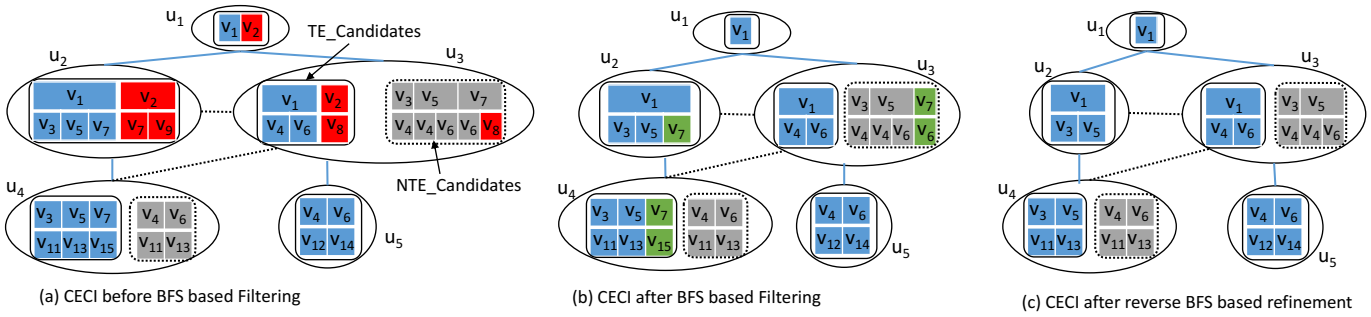


Figure 3: CECI with BFS based filtering and reverse BFS based refinement for the query and data graphs of Figure 1. TE_Candidates and NTE_Candidates are colored in blue and gray respectively. The red and green portions are removed by BFS based and reverse BFS based filtering steps respectively.

and place them in equivalence groups. For example, the vertices 0, 1, and 2 in QG1 in Figure 6(a) are equivalent to each other as they have the same label and are connected to the same set of neighbors or with each other. Second, as [16] suggests, a specific order is enforced within each group to break automorphism. Since the values of the query vertex IDs follow $0 < 1 < 2$ in our equivalence group, a matching order of $map(0) < map(1) < map(2)$ should also be enforced in each embedding to eliminate the automorphisms.

Finding the cluster pivots, i.e., a set $\{v_s\}$ of data nodes that matches with the root query node u_s. Each node in $\{v_s\}$ identifies a distinct embedding cluster. In Figure 1, the cluster pivots for $EC1$ and $EC2$ are v_1 and v_2, from which parallel graph exploration starts to create CECI as we will discuss shortly.

3 CECI CREATION, FILTERING, AND REFINEMENT

This section discusses the techniques to find the candidates, i.e. potential matches of each query node. The candidates undergo several filtering and refinement stages so that the number of false candidates in the verification stage is as small as possible. By traversing the data graph following the BFS traversal order, we create the embedding clusters in form of CECI from the data graph, which accurately captures the topological structure and connectivity of the query graph.

3.1 CECI Structure

Given the query tree T_q of a query graph G_q, CECI represents a data graph G in the structure similar to T_q, where each node in CECI contains the set of data nodes that are the candidates of the corresponding query node u. Each query node u has a tree edge connecting it with its parent u_p. Each CECI node consists of tree edge candidates (TE_Candidates), which are key-value pairs of $<v_p, \{v\}>$, where v_p is the candidate of u_p and $\{v\}$ is the set of candidates of u that are adjacent to v_p.

Figure 3 shows the CECI for the query graph and the data graph from Figure 1. For the tree edge (u_1, u_2), the TE_Candidates consist of $<v_1, \{v_3, v_5, v_7\}>$ and $<v_2, \{v_7, v_9\}>$. In other words, the edges (v_1, v_3), (v_1, v_5), (v_1, v_7), (v_2, v_7), and (v_2, v_9) are candidates of the query edge (u_1, u_2).

If there is a non-tree edge incident upon it, a CECI node contains the non-tree edge candidates (NTE_Candidates), which are also key-value pairs, $<v_n, \{v\}>$. For a non-tree edge (u_n, u), v_n is the candidate of a query node u_n, an NTE neighbor of u, and $\{v\}$ is candidate set of u that are adjacent to v_n. For the non-tree edge (u_2, u_3) in our example, the NTE_Candidates are $<v_3, \{v_4\}>$, $<v_5, \{v_4, v_6\}>$, and $<v_7, \{v_6\}>$.

3.2 CECI Creation and BFS Based Filtering

Following the BFS traversal order, we explore the data graph starting from the pivot nodes level by level to obtain the candidates of each query node. Algorithm 1 presents the steps for creating the TE_Candidates of CECI. The NTE_Candidates can be computed in a similar fashion.

For each node u in the query tree, we first determine the *frontiers*, i.e., the set of nodes from which we expand the CECI. The union of TE_Candidates of node u_p (the parent of u) on CECI generates such frontier set (line 3). In Figure 3(a), the parent of the query node u_4 is u_2, whose TE candidates are $\{v_3, v_5, v_7\}$ and $\{v_7, v_9\}$. The union of these sets produces the frontiers $\{v_3, v_5, v_7, v_9\}$. Note that v_9 is not expanded because it is removed by filtering before we explore u_4. If the parent is the root query node, the frontiers are the cluster pivots. In Figure 3(a), for query node u_2 and u_3, the frontiers for their parent u_1 consist of nodes v_1 and v_2.

For each frontier node v_f, CECI applies four filters to its neighbor nodes $N(v_f)$. First, a *label filter* (LF) collects adjacent nodes $\{v\}$ of v_f that have the label of u (line 5) i.e., the nodes with different labels are removed. In our example, query nodes u_2 and u_3 have labels B and C respectively. For u_2, data nodes $\{v_3, v_5, v_7\}$ adjacent to v_1 and $\{v_7, v_9\}$ adjacent to v_2 are selected as they have label B. Similarly for u_3, data

Algorithm 1: TE_Candidates Construction and Filtering.

```
   Input: T_q, and G
   Output: TE_Candidates and Frontiers of CECI
1  function BFSFilter (T_q, G)
2      forall u ∈ T_q in BFS order do
3          CECI[u_p].Frontiers=∪ CECI[u_p].TE_Candidates
4          forall v_f ∈ CECI[u_p].Frontiers do
               // Label Filter
5              LF(N(v_f), L_q(u))
               // Degree Filter
6              DF(u, v)
               // Neighborhood Label Count Filter
7              NLCF(u, v)
8              CECI[u].TE_Candidates[v_f].add(v)
               // No Tree Edge Candidate
9              if (|CECI[u].TE_Candidates[v_f]| = 0) then
10                 CECI[u_p].Frontiers.delete(v_f)
11                 forall u_c ∈ children of u_p do
12                     CECI[u_c].TE_Candidates.delete(v_f)
13     return TE_Candidates
```

nodes $\{v_4, v_6\}$ adjacent to v_1 and $\{v_8\}$ adjacent to v_2 have label C. Note that v_8 will later be removed by NLC filter.

Next, for each selected candidate v, we apply lightweight *degree filter* (DF) and then more expensive *neighborhood label count filter* (NLCF) on the remaining candidates (line 6 - 8). The former ensures that the degree of data node v is larger than or equal to that of the query node u, and the latter requires that the count $count_v(l)$ of label l nodes in the neighborhood of v must be equal to or greater than the $count_u(l)$ for all distinct labels l in the neighborhood of u. If v passes all these filters, it is added to the value set for the key-value pair $<v_p, \{v\}>$ on the TE_Candidates of the node u. For example, the nodes v_3, v_5, v_7, v_9 on candidate of u_2 have degree no less than that of u_2, i.e., 3 and have at least one adjacent nodes with labels A, C, and D each. Thus, all of them are retained on TE_Candidates. For u_3, the nodes $\{v_4, v_6\}$ have degree 5 and have each A, B, D, and E labeled node in the neighborhood. However, $\{v_8\}$ is filtered out because v_8 does not have label E node among its neighbors.

The last filter (line 9 - 12) is based on the fact that if the TE_Candidates for node u has no entry for key v_p, then there will be no embedding that matches v_p to u_p. Therefore, CECI removes v_p from the candidates of u_p, and the TE_Candidates corresponding to v_p from all children of u_p. In our example, after the first three filters, the TE_Candidates of u_3 for key v_2 becomes empty, thus we remove the node v_2 from the candidates of u_1 and the entry corresponding to v_2 from TE_Candidates of u_2.

For the construction of NTE_Candidates, for each non-tree edge, the node appearing earlier in the matching order acts as the parent and the other as child. The expansion starts from the frontiers of parent node which can be obtained by the union of TE_Candidates and NTE_Candidates for the parent node, and follows the similar steps as in Algorithm 1. For node u_3 in our example, the non tree edge (u_2, u_3) finds its NTE_Candidates by expanding the frontiers of u_2, i.e. $\{v_3, v_5, v_7\}$. The nodes $\{v_4\}$ adjacent to v_3, $\{v_4, v_6\}$ adjacent to v_5, and $\{v_6, v_8\}$ adjacent to v_7 pass the Label filter. Later, v_8 is pruned as it does not pass NLCF. The resulting NTE_Candidates are $<v_3, \{v_4\}>$, $<v_5, \{v_4, v_6\}>$ and $<v_7, \{v_6\}>$.

3.3 Reverse BFS Based Refinement

The BFS order based exploration to create TE_Candidates and NTE_Candidates only filters a portion of the unpromising candidates. This section introduces a reverse BFS ordered refinement to further prune them. The key idea here is to focus on the child nodes, in contrast to the parent node in filtering. Refinement traverses the CECI created in the previous section in the reverse BFS order, e.g., $(u_5, u_4, u_3, u_2, u_1)$ in our example, and removes the disqualified candidates.

To facilitate the refinement, we calculate the *cardinality* for each candidate of a given query node, which also acts as the cost function for the given query and data node pair (u, v). The cardinality for (u, v) represents the maximum possible number of embeddings that can be obtained by matching v with u. The cardinality for the candidates of leaf query nodes, i.e., degree one nodes in the query tree, are one. For other query nodes, the *cardinality* (u, v) for a query node u and a data node v is obtained from the cardinalities of their children.

Formally, cardinality can be represented as

$$cardinality\,(u, v) = \prod_{u_c} \Big[\sum_{v_c} cardinality(u_c, v_c) \Big]$$

Here u_c are the children of u in query tree, while v_c are nodes adjacent to v that are candidates of u_c. For a given u_c, the list $\{v_c\}$ is the list of the nodes that are in TE_Candidates of u_c for key of v and also in the NTE_Candidates of u_c. This algorithm works iteratively from leaf nodes to root query node eventually finding the cardinality for every query node on CECI.

In Figure 3(b), the cardinalities of nodes $\{v_{11}, v_{13}, v_{15}\}$ and $\{v_{12}, v_{14}\}$ in TE_Candidates of u_4 and u_5 are all one, as these two nodes are leaves in T_q. For u_2, the cardinality of v_3 and v_5 are one, as they have one node each as their child i.e., on TE_Candidates of u_4. However, the cardinality of v_7 is zero since its only child v_{15} is not in the NTE_Candidates of u_4.

The process of CECI refinement is illustrated in Algorithm 2. If the cardinality of pair (u, v) is zero, that means the node v is guaranteed not to match u and hence should be removed from TE_Candidates as well as NTE_Candidates

Algorithm 2: Reverse BFS Based Refinement

```
   Input: T_q, and CECI
   Output: refined CECI
1  function revBFSRefine ( G_q, CECI, G )
2      forall u ∈ T_q in reverse-BFS order do
3          score = 0
4          foreach v ∈ CECI[u].TE_Candidates do
5              if v ∉ CECI[u].NTE_Candidates then
6                  cardinality(u, v) = 0
7              score += cardinality(u, v)
               // if v is guaranteed not to match u
8              if cardinality(u, v) == 0 then
9                  CECI[u].TE_Candidates.delete(v)
10                 forall u_c ∈ children of u do
11                     CECI[u_c].TE_Candidates.clear(u)
                       CECI[u_c].NTE_Candidates.clear(u)
12             cardinality(u_p, uList.key) *= score
13     return CECI
```

of u. In addition, the entry of v from TE_Candidates and NTE_Candidates of all children (including non tree edge neighbors) of u also has to be removed. In our example, we remove v_7 from the candidates of u_2. Also, the $<v_7, \{v_6\}>$ entry from NTE_Candidates of u_3 is removed although it has the valid cardinality of one for v_6. Figure 3(c) shows the CECI after refinement.

3.4 Time and Space Complexity

The size of CECI for a data and query graph pair is $O(|E_q| \times |E_g|)$. Intuitively, for each edge in the query graph, at most all the edges in the data graph can be its candidates. The space complexity of index for TurboIso [17] and CFLMatch [4] are $O(|E_g|^{|V_q|-1})$ and $O(|V_q| \times |E_g|)$ respectively. Compared to CFLMatch, CECI incurs higher space cost due to the inclusion of NTE_Candidates, but it accelerates the embedding enumeration process. The time complexity of CECI creation and refinement is $O(|E_q| \times |E_g|)$.

3.5 Correctness

The CECI created is *complete*, i.e., for every pair of nodes u and u_p in the query graph, if there is an embedding of the query graph on the data graph that maps v to u and v_p to u_p, then $< v_p, v >$ is guaranteed to be in either the TE_Candidates or the NTE_Candidates of node u on CECI, contingent upon whether (u_p, u) is a tree or non-tree edge.

Lemma 1*: Given a complete CECI, all embeddings of a query G_q in a data graph G can be computed by traversing all embedding clusters in CECI.*

First, every edge contained on TE_Candidates or NTE_Candidates is also an edge in G. Second, every candidate edge that is removed during CECI creation and refinement is guaranteed not to match the corresponding query edge. The former ascertains that every embedding found by exploring CECI is a true embedding from the data graph, while the latter ensures that no legitimate candidates are eliminated during refinement and filtering.

The complete CECI may not be *minimal*. A minimal CECI is the one where TE_Candidates and NTE_Candidates only contain the candidates that are guaranteed to be part of at least one embedding. Obtaining the complete and minimal candidate CECI is an NP-hard problem [4]. Thus, our work only provides the completeness guarantee, i.e., all the embeddings in the data graph can be found by exploring CECI. Since there can be false candidates for some query vertices in non-minimal CECI, we need further verification while enumerating the embeddings. In particular, Section 4 discusses our set intersection based embedding enumeration technique.

3.6 Implementation

The TE_Candidates and NTE_Candidates for the given query node u are constructed using C++ STL vectors, where each element of the vector holds a pair. The first part of the pair (key) is a scalar, node v_f from frontier of the parent query node u_p. The second part of the pair (value) is an STL vector, which holds all candidates of u that are adjacent to the key. The lists, once constructed are sorted by key so that lookup algorithms like *binary search* and *lower bound* can be used [39].

The filtering and refinement has been parallelized using OpenMP. In the BFS ordered filtering, TE_Candidates and NTE_Candidates are calculated by expanding the frontiers which are dynamically distributed among the threads using traditional pull-based workload distribution model. Allowing each thread to write the TE_Candidates or NTE_Candidates creates serialization in writing. To solve the problem, an intermediate private bin is created for each thread, and they are merged only when all the frontier nodes are expanded. Similarly, during the reverse BFS exploration, the nodes on TE_Candidates are distributed among multiple thread dynamically. To reduce the workload skew, if a certain frontier node has degree more than a predefined threshold (1M by default), it is broken down to multiple pieces and fed to different workers.

4 PARALLEL EMBEDDING ENUMERATION

CECI generated in Section 3 is a collection of embedding clusters, each of which has different size as determined by the cardinality defined in Section 3.3. Figure 4(a) describes the overall workflow of embedding enumeration using CECI. As

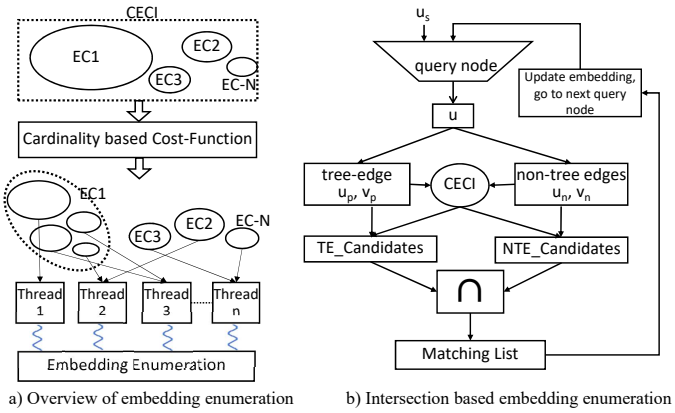


Figure 4: Parallel embedding enumeration with CECI.

Section 4.3 discusses, the large clusters can be further decomposed into smaller sub-clusters. The clusters are assigned to multiple processors and searched upon concurrently. Each processor explores the assigned cluster iteratively starting from its pivot following a specific matching order to generate all embeddings. In the running example, we use the BFS matching order, i.e., $(u_1, u_2, u_3, u_4, u_5)$.

Embedding enumeration in this work is based on set intersection between the TE candidates and NTE candidates. The effectiveness of intersection-based approach for speeding up triangle counting [20, 21] motivated us to adopt such technique. This method is different from existing subgraph enumeration solutions [4, 17] because they only have auxiliary data structure equivalent to TE_Candidates. Without NTE_Candidates, these solutions only obtain the candidate nodes for each query node, which require further verification in order to confirm the presence of non-tree edges.

In the example above, after matching v_1 with u_1 and v_3 with u_2, we get the partial embedding (v_1, v_3). Candidates of u_3 is obtained from TE_Candidates, i.e., $\{v_4, v_6\}$. Each of these candidate nodes has to have an edge with v_3, the match for u_2, in order to get mapped with u_3. This requires verification of edges (v_3, v_4) and (v_3, v_6) on the data graph.

Our enumeration starts by mapping the pivot of the embedding cluster to the root query node, e.g., v_1 to u_1. For a query node u, the TE_Candidates for v_p, where v_p is the data node mapped to u_p, are obtained. Additionally, if node u has non-tree edges, then NTE_Candidates for node v_n matched with u_n are collected. The set intersection between TE_Candidates and NTE_Candidates provides matching nodes $\{v\}$ that can be mapped to u in the current embedding.

In the running example, the matching nodes for u_2 are obtained from its TE_Candidates for v_1, which are $\{v_3, v_5\}$. Mapping v_3 to u_2, the matching nodes for u_3 are obtained by intersection of two lists, TE_Candidates of u_3 for v_1, i.e., $\{v_4, v_6\}$, and NTE_Candidates of u_3 for v_3, i.e., $\{v_4\}$, resulting on the matching node $\{v_4\}$.

Each entry in $\{v\}$ can be matched with u if it has not been matched with another query node already. The matching node is appended to the current embedding, and the process is repeated for the next query node in the matching order until the embedding is complete or the whole embedding cluster is searched upon. In our example, after matching v_4 to u_3, the current embedding becomes $(v_1, v_3, v_4,)$. Repeating the process for u_4 and u_5 gives one complete embedding $(v_1, v_3, v_4, v_{11}, v_{12})$.

When an embedding is complete or it can no longer be expanded, the process backtracks to the previous query node in the visit order and repeats the process for its next matching node. In our example, we backtrack back to u_2, match v_5 to u_2, and advance again to find the second embedding $(v_1, v_5, v_6, v_{13}, v_{14})$. If the process backtracks back to the root query node, the search on the specific embedding cluster is complete. Figure 4(b) shows the workflow of this process.

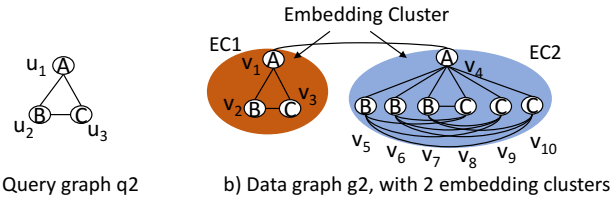
4.1 Benefits of Intersection-Based Method

In addition of having to keep both data graph and auxiliary structure in memory, edge verification is also time costly. Using the adjacency list based graph format, to check whether there is an edge between nodes v_x and v_y, it requires a run-time proportional to the degree of v_x or v_y whichever is the smaller. Even if the adjacency list is sorted, the run-time reduces to log of that degree. To overcome this overhead, CFLMatch [4] uses an adjacency matrix representation (with size $|V| \times |V|$) of the data graph, which limits it to small data graphs only.

Lemma 2: The cost of intersection based enumeration is always less than or equal to using edge verification.

Let's consider a non-tree edge connecting u_i and u_j with m and n candidates respectively. The cost of edge verification is $m \times n$ multiplied with the cost of verifying each edge. With $O(n)$ intersection time (sorted adjacency list), the total time for intersection is $m \times n$. Since the intersection result can be readily added into the embedding, the cost is reduced by a factor of average time for edge verification in data graph. Even with the sparse matrix representation i.e., constant time for edge verification, intersection based approach will outperform edge verification by a constant factor.

Adopting intersection based enumeration provides average improvement of 13% to 170% on run-time for query graphs listed in Figure 6. The speedup is higher for the query graphs with larger number of non-tree edges.



a) Query graph q2 b) Data graph g2, with 2 embedding clusters

Figure 5: Example for imbalanced workload

4.2 Parallel Embedding Enumeration

CECI enables the logical decomposition of the data graph into multiple embedding clusters, where each cluster may contain one or more embeddings of a given query graph. This transforms the problem of subgraph listing on a single large data graph into several smaller subgraphs of that data graph. Naturally, we can now employ multiple machines (workers) to perform subgraph listing on each of these subgraphs, i.e, embedding clusters. These embedding clusters are identified uniquely by their pivot nodes.

Considering each cluster as a single work unit, the naive approach to distribute the total work is to assign an equal number of embedding clusters to each worker. This approach is referred to as **static (ST)** workload distribution, since there is no re-adjustment of the workload once it is assigned. In Figure 5, there are two clusters $EC1$ and $EC2$. If we have two workers available, it will assign one cluster to each worker.

Due to the power law nature of real world graphs, the static distribution suffers from huge disparity among the clusters in term of workload. The obvious solution is to adopt a dynamic workload distribution to supply variable number of embedding clusters to different workers depending on the size of each cluster. We have used a classical pull-based dynamic workload balancing model [11, 27], which allows each worker to start with a certain number of clusters and can request an additional cluster. This approach is called **coarse grained dynamic (CGD)** due to its cluster level granularity.

4.3 Handling Extreme Clusters

In CECI, we aim to identify *ExtremeCluster* which are large embedding clusters that dominate the total subgraph listing time ahead of the time. If assigned to a single machine, these clusters would largely limit the speedup of parallel processing. For example, in Figure 5, cluster $EC2$ contains nine of the total ten embeddings. Assigning $EC2$ to one worker would limit the maximum parallelization speedup to 1.11.

At this point, we advocate the use of cardinality calculated during CECI refinement in Section 3.3. Recall that the cardinality is defined for each pair (u, v) between query node u and data node v. It represents the maximum number of embeddings that can be obtained by matching v with u. Following this convention, the cardinality for the root query node and pivot (u_s, v_s) gives the maximum number of embeddings possible on a given embedding cluster. In Figure 5, the total cardinality for (u_1, v_1) and (u_1, v_4) are 1 and 9 respectively, which reflects the total embeddings on clusters $EC1$ and $EC2$.

We empirically choose a parameter β, and flag a given cluster as ExtremeCluster if the cardinality of that cluster is greater than the threshold of $\beta \times cardinality_{exp}$, where $cardinality_{exp}$ is the expected workload per worker. If a cluster is found to be ExtremeCluster, it is decomposed into several sub-clusters, each of which must have a cardinality less than the threshold $\beta \times cardinality_{exp}$. The sub-clusters are further decomposed if it is an ExtremeCluster.

Algorithm 3: Extreme Cluster Decomposition

```
1  work = { }; i = 0; prev = { }
2  work_unit = CECI[u_s].frontiers
3  Card_old = Σ_{v∈work_unit}(CECI[u_s].cardinality[v])
4  Card_exp = Card_old / worker_count
5  β = 1
6  work = prepare_work(i, prev, work_unit , CECI[0].cardinality,
   Card_old)
7  function prepare_work ( i, prev, work_unit, cardinality, card_old )
8      total = Σ_{v∈work_unit}(cardinality[v])
9      forall w ∈ work_unit do
10         myWork = (card(u_i, w) / total) * card_old
11
12         if myWork ≤ β × Card_exp then
13             work.append(prev ⊕w)
14         else
15             work_unit = CECI[i + 1].TE_Candidates[w]∩
                {∪CECI[i + 1].NTE_candidates}
16             workloads = CECI[i + 1].cardinality
17             prepare_work(i + 1, prev ⊕w, work_unit ,
                workloads, myWork)
18     return work
```

Algorithm 3 describes the routine for decomposing the ExtremeClusters. Conceptually, if we detect an ExtremeCluster while matching node v to query node u (line 8-10), the expansion of v to the next query node u_{next} in the matching order is accomplished using multiple workers. First, the matching nodes for u_{next} adjacent to v are obtained from its TE_candidates and NTE_Candidates (line 13). For each entry v' in matching nodes, effective workload is calculated as $\frac{cardinality(u_{next}, v')}{total(v')} \times cardinality(u, v)$, where $total(v')$ is the sum of cardinalities of all the matching nodes. If workload of v' is greater than $\beta \times cardinality_{exp}$, the function is called again recursively.

In Figure 5, let us assume $\beta = 1$ and the number of workers $k = 2$. Then the threshold $\beta \times cardinality_{exp} = 5$ which means $cardinality(u_1, v_4) = 9$ is greater than the threshold,

therefore $EC2$ is broken down into three sub-clusters with respect to three matching nodes of u_2 in that cluster. After decomposition, all three sub-clusters have a cardinality of 3, i.e, smaller than the threshold. Those three sub-clusters along with $EC1$ is then distributed between two workers. This distributed solution will be referred as **fine-grained distributed (FGD)** in subsequent sections.

The value of β provides a trade-off between workload balance obtained and overhead for doing so. A smaller β increases one time cluster decomposition cost, but provides more balanced workload distribution. To smooth the finishing time of different threads, we sort the clusters and sub-clusters in the work pool by cardinality. This allows the larger clusters to be searched earlier and keeps small clusters in the end.

Note that the cardinality overestimates the total number of embeddings present on each embedding cluster. However, it still serves as an effective reflection of workload because larger embedding clusters in general tend to have bigger number of isomorphic embeddings, and the clusters that do not follow above-mentioned trend contain high false candidate count which needs verification before discarding them thereby increasing the amount of work.

5 DISTRIBUTED IMPLEMENTATION

In addition to a multi-threaded (OpenMP) version, we have implemented the CECI system on a distributed-memory cluster (with OpenMP and MPI). To manage the graph data in the cluster, two methods have been explored: replicating the whole data graph on the memory of each machine (in-memory data graph), and storing the graph on a lustre file system, shared by several compute machines (shared data graph). Efficient decomposition of graph into different machines is a non-trivial problem and is left out for now.

In the first approach, all machines hold the same data and query graph but work on the disjoint set of embedding clusters creating different CECI on each machine. For the second approach, there is only one copy of the data graph shared on the networked storage, in the Compressed Sparse Row (CSR) format [28]. Each machine uses a *beginning_position* array to locate the adjacency list for a given vertex. Both methods distribute the cluster pivots to multiple machines by using MPI library's synchronous communication functions MPI_Send and MPI_Recv.

While distributing pivots $\{v_s\}$ among machines, two major concerns need to be addressed. First, since cardinality is not available before CECI creation, we rely on a light-weighted approximation to balance the workload. In the in-memory setting, the workload for each vertex v is proportional to the sum of degree of v and degree of all neighbors of v. In the second approach, only the degree of a node v is used since the neighbor information is not available. In both cases, the workload is scaled using vertex ID to account for imbalance inflicted by automorphism breaking orders, i.e., effective workload for v is equal to $\frac{|V|-v}{|V|} \times workload(v)$. Since CECI creation takes a small portion (<5%) of the total run-time, even such a coarse method is able to deliver an acceptable performance.

Second, two embedding clusters can overlap with each other (e.g., in Figure 1). When highly overlapping clusters fall in different machines, it incurs significantly redundant exploration. We estimate the similarity between two embedding clusters via Jaccard distance. If v_i and v_j are the pivots of clusters EC_i and EC_j, respectively, the similarity between these two clusters is $J(v_i, v_j)=\frac{N(v_i)\cap N(v_j)}{N(v_i)\cup N(v_j)}$. If $J(v_i, v_j) \geq 0.5$, these two clusters will be handled by the same machine provided that the total workload does not exceed the maximum allowed workload. To reduce the overhead of similarity calculation, CECI system only runs similarity measure on the largest $1,000$ clusters (as defined by light-weighted workload). In our second approach, this method cannot be applied as each machine does not have the whole graph.

During embedding enumeration, the CECI system proceeds as follows: (1) Each machine starts enumeration on embedding clusters from its CECI. These machines also keep record of unexplored clusters in their CECI. (2) When all the embedding clusters in the local CECI are done, it steals the work from the machine with the maximum number of unexplored clusters (victim machine). Work stealing is implemented by using one-directional function MPI_Get. (3) Finally, if needed, the results from all machines are accumulated to a single machine.

6 EXPERIMENTS

We have implemented the CECI system with around 3,000 lines of OpenMP, MPI, and `C++` code and evaluated it on three different machines. (1) **Server 1:** A server with dual-socket Intel Xeon CPU E5-2683 CPUs with 28 cores and 512 GB memory for main memory. (2) **Server 2:** A quad-socket Intel Xeon CPU E7-8857 CPUs with 48 cores and 2 TB memory which is used to run yahoo graph only. (3) **Cluster 1:** 16 8-core nodes cluster with dual-socket Intel Xeon CPU E5-2650 CPUs with 128 GB memory each for distributed implementation. All machines run Linux 3.10.0 kernel with GCC 4.8.5 and compilation flag `-O3`. All reported results are the average of five runs.

Query graph: Five unlabeled query graphs from Figure 6, which are also used by recent projects PsgL, TTJ, and DualSim are used. Section 6.2 experiments on larger, and labeled query graphs.

Data graph: The experiments run on both undirected and directed data graphs listed in Table 1. The first nine are

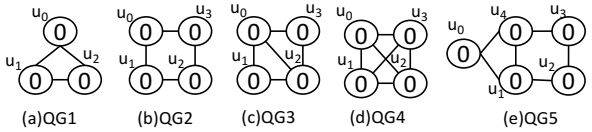


Figure 6: Different simple query graphs used for testing. Note that all the nodes have same label 0.

real graphs while the last one, `rand_500k`, is a synthetic graph generated by Graph500 Kronecker Generator [15]. The numbers of vertices and edges range from 0.5 million to 1.5 billion, and 2 million to 12.9 billion, respectively. All real graphs are obtained from Stanford Large Network Dataset Collection [49], except Yahoo [55].

Table 1: Graph datasets used in the experiments. $|V|$ and $|E|$ are the vertex count and edge count respectively.

Datasets	Abbr.	|V|	|E|	Directed
citPatent	CP	3.77M	16.5M	Y
Friendster	FS	65.6M	1.8B	N
Human	HU	4.6K	0.7M	N
live-journal	LJ	3.99M	34.68M	N
Orkut	OK	3.0M	117.2M	N
Webgoogle	WG	0.9M	8.6M	Y
wiki-talk	WT	2.3M	5.0M	Y
Yahoo	YH	1.4B	12.9B	N
Youtube	YT	1.1M	3.0M	N
rand_500k	RD	0.5M	2.0M	N

6.1 Small Query Graphs

We have run the experiment to find *all embeddings* of query graphs QG1 to QG5 from Figure 6 on eight real world datasets. Only one automorphism of each embedding is listed by utilizing vertex numbering based automorphism breaking technique as described in Section 2.2.

The performance numbers of DualSim are quoted from their paper [24]. Although the DualSim paper reports the result for 6-core machine, it is unlikely to achieve very high performance in a machine with higher core count. Its algorithm is IO bound in nature since DualSim iteratively loads a fixed set of disk pages, each of which contains the adjacency list for one node, and runs isomorphism with that set at each time. To achieve better scalability, DualSim needs to process the whole graph together, which will result in an exponential memory requirement. We implemented PsgL (for optimal setting $\alpha = 0.5$) on shared memory using OpenMP. Hence, the communication overhead for PsgL is minimized.

Figure 7 compares the performance of the CECI system with state-of-the-art solutions – DualSim [24] and PsgL [47] on QG1 and QG4. The run-time reported here includes the time for preprocessing, CECI creation, and enumeration of

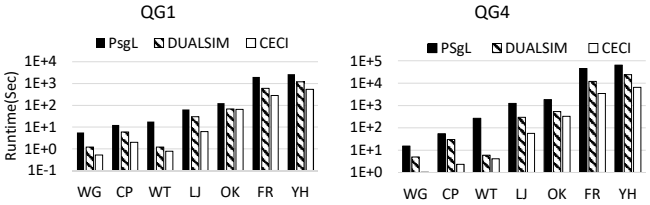


Figure 7: Performance comparison with DualSim and PsgL on query QG1 and QG4. The standard deviation of run-time is less than 2% of the average.

the embeddings. The last step takes more than 95% of the total run-time.

On average, our system outperforms DualSim and PsgL by 1.86× and 4.08× respectively for QG1 and 4.54× and 14.31× respectively for QG4. For QG1, we obtain the highest speedup of 4.8× on LJ and 22.6× on WT against DualSim and PsgL respectively. Similarly, for QG4, the highest speedup is 12.8× on CP and 63.7× on WT.

Figure 8 shows the performance of CECI system against DualSim and PsgL for three remaining query graphs on WG, WT and LJ data graphs. The average speedup of 19.7×, 49.3×, and 86.7× is obtained over PsgL on QG2, QG3 and QG5 respectively. Similarly, the average speedup of 2.5×, 1.7×, and 19.8× is obtained against DualSim. We omit the other bigger data graphs because PsgL cannot finish the query in an extensively long period of time (days) and the DualSim manuscript does not include the results for those data and query combination.

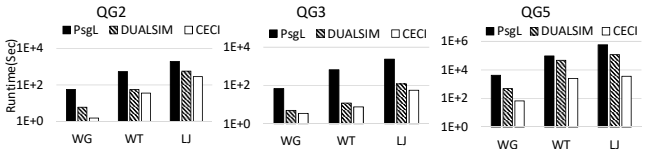


Figure 8: Performance comparison with DualSim and PsgL on query graphs QG2, QG3, and QG5. Standard deviation of runtime is less than 2% of the average.

In general, PsgL is slower because it lacks the ability to prune the unpromising paths prior to exhaustive expansion (Figure 18) and has weaker thread scalability as compared to CECI (Section 6.5) due to exhaustive work distribution method. The speedup over DualSim comes from the better utilization of compute power. Since DualSim loads a set of few slotted pages from graph at a time to run listing algorithm, it incurs multiple IO loads and is able to supply very limited amount of workload in a given time.

6.2 Larger and Labeled Query Graphs

In this section, we evaluate CECI system with larger and labeled query graphs. We perform Depth-first search (DFS)

traversal of data graphs from random source nodes in order to generate connected query graphs of different size from 3 to 50 nodes similar to existing works [4, 17, 50]. Iteratively, a new node is selected and every backward edge from that node to already selected nodes is added to query graph until the required node count is achieved. Thus, at least one isomorphic embedding will be found for each query. For each node count, 100 query graphs are generated.

First, we compare CECI system with CFLMatch [4], a state-of-the-art subgraph matching solution on labeled graphs. However, we restrict the comparison with only two data graphs – RD and HU due to CFLMatch's inability to handle larger data graphs. We randomly inject each node of RD with one of the 100 different labels. HU dataset comes with one or more of 90 different labels on each node. The node labels are transferred to query graph while doing DFS based query generation. If the data node has multiple labels, only the first label is used in the query node. Both CFLMatch and CECI runs single threaded solutions, and only return the first 1,024 embeddings. All of the 100 queries for each size is run for five times and the average of 500 execution is reported.

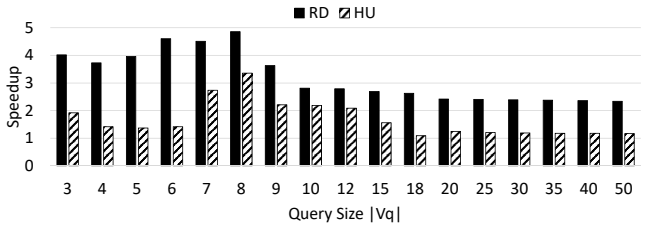


Figure 9: Performance comparison with CFLMatch on larger query graphs. The standard deviation of runtime is less than 5% of the average.

As shown in Figure 9, CECI system outperforms CFLMatch by average 3.5× and 1.9× on RD and HU graph respectively. The speedup comes mainly from intersection based embedding enumeration. The speedup in HU is smaller than in RD because CFLMatch uses only one label per node for HU while CECI system uses multiple labels. The speedup decreases slightly as the query graph gets larger because CECI system has used naive BFS matching order while CFLMatch has their optimized core-forest-leaf matching order, which offers bigger advantage on larger query graphs.

Additionally, we compare our solution with TurboIso[2] on the HU graph to find the first 1024 embeddings. Figure 10 shows that our system is on average 2.71× and 2.52× faster than TurboIso and Boosted-TurboIso respectively. The

[2]We obtain the source code from authors of BoostIso [45], which contains two versions of TurboIso. The first implementation, TurboIso, replicates the works on [17] while the second version, Boosted-TurboIso, speeds up TurboIso further by exploiting the vertex symmetry in data graph.

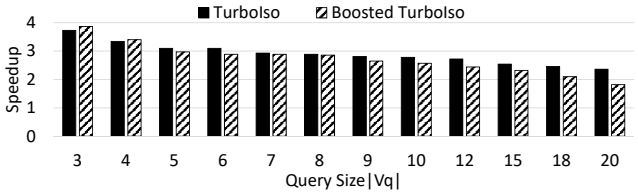


Figure 10: Performance comparison with TurboIso and Boosted TurboIso on larger query graphs. The standard deviation of runtime is less than 5% of the average.

speedup comes from intersection based enumeration, reduction of redundancy in filtering, and better utilization of tree-edge and non-tree edge connectivities to filter the unpromising candidates early.

6.3 Workload Balancing

This section uses QG1, QG3, and QG5 as the query graphs so that CECI system experiences workload imbalance at various backtracking tree depths: 3, 4 and 5 respectively. The speedup obtained by adopting Coarse Grained Dynamic (CGD), and Fine Grained Dynamic (FGD), against static workload distribution (ST) are measured. The value of β is fixed to 0.2, i.e., the embedding clusters that are larger than one fifth of expected cardinality per thread are ExtremeClusters.

From Figure 11, one can see that FGD and CGD techniques clearly outperform ST. FGD is on average 16.8× better than CGD, and CGD is 10.7× faster than ST. In rare cases (WT on QG3), FGD is slightly slower than CGD because those cases do not have ExtremeCluster and treating some clusters as ExtremeCluster increases one time distribution overhead slightly.

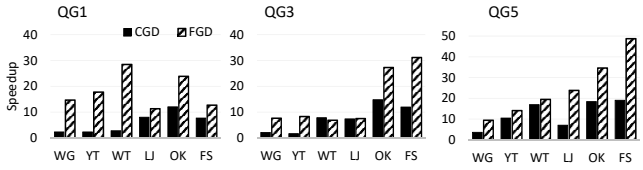


Figure 11: Speedup obtained by adopting dynamic workload balancing method, and ExtremeCluster decomposition over static workload distribution method.

Figure 12 shows the effect of different β on time spent by each worker while running QG3 on FS graph. As we decrease β, the time spent by fastest processor increases, but the high skew at the end is reduced significantly. Scheduling overhead for $\beta = 1, 0.2$, and 0.1 are 14.76, 16.53, and 23.96 Sec respectively.

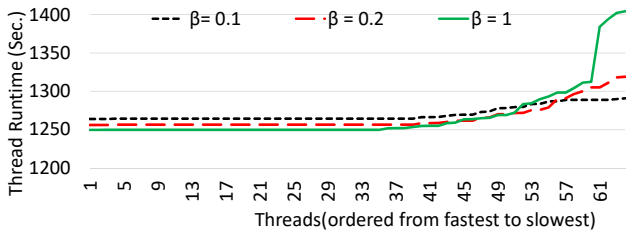


Figure 12: Effect of different β.

6.4 CECI size

Table 2 shows the size of CECI for different data and query graph pairs. Since TE_Candidates and NTE_Candidates only store candidate edges once, CECI system can drastically reduce the space consumption to $O|E_g| \times |E_q|$ as compared to $|E_g|^{|E_q|}$ of aggressive instance expansion approach in PsgL [47]. Here, $|E_q|$ and $|E_g|$ are the query and data graph edge counts.

Table 2: CECI size in GB for different query and data graph combinations. The numbers inside the parentheses and brackets are theoretical memory requirement, and % of space saved by CECI, respectively.

Graph	FS	LJ	OK	WT	YH	YT
QG1	20 (40) [51%]	0.9 (1.5) [45%]	1.3 (2.6) [51%]	0.01 (0.1) [83%]	121 (288) [58%]	0.07 (0.1) [31%]
QG2	26 (54) [52%]	1.1 (2.1) [48%]	1.7 (3.5) [50%]	0.02 (0.2) [88%]	160 (384) [58%]	0.09 (0.2) [45%]
QG3	32 (67) [52%]	1.3 (2.6) [50%]	2.1 (4.4) [52%]	0.03 (0.2) [86%]	225 (481) [53%]	0.09 (0.2) [57%]
QG4	38 (81) [54%]	1.5 (3.0) [49]	2.5 (5.3) [53%]	0.03 (0.2) [85%]	274 (577) [52%]	0.11 (0.3) [66%]
QG5	39 (81) [52%]	1.6 (3.0) [47%]	2.6 (5.3) [52%]	0.03 (0.2) 84%	290 (577) [50%]	0.12 (0.3) [62%]

Filtering and refinement reduces the size of CECI significantly below the theoretical limit. For example, QG5 and YH pair has theoretical size of 624 GB since $|E_q|$ is 6, $|E_d|$ is 12.9 Billion and 8 bytes is used to store each edge. However after usage of BFS filtering and reverse-BFS refinement, the CECI size is only 290 GB, reduced by 2.2×. For larger graphs whose CECI does not fit inside memory, we plan to store it in non-volatile memory [30].

Among existing solution, CFLMatch [4], due to its usage of adjacency matrix based graph representation failed to run data graphs graph larger than 500K nodes on Server 1. Similarly, PsgL [47] due to its exponential intermediate result sets failed to run on Server 1 for YH graphs, i.e., it needed more than 512 GB. DualSim [24] masks the problem of exponential memory requirement by loading only a small portion of graph into memory at a time. On the other hand, TurboIso [17] saves memory by serializing the auxiliary data creation and verification thereby making the solution unsuitable for larger number of embeddings.

6.5 Scalability

This part evaluates the scalability of CECI system under both shared and distributed memory settings. The results presented accounts for the time for both CECI creation and embedding enumeration.

Figure 13 and 14 display the speedup obtained with the increasing number of threads while running QG1 and QG4 respectively on FS and OK on Server 1. We compare the result against speedup on PsgL. CECI system is able to obtain a near linear speedup up to 16 workers. However, the trend slightly flats out beyond 16 threads due to the lack of adequate workload. In general, the better speedup for CECI as compared to PsgL is due to light-weighted workload balancing and better locality on candidate set access.

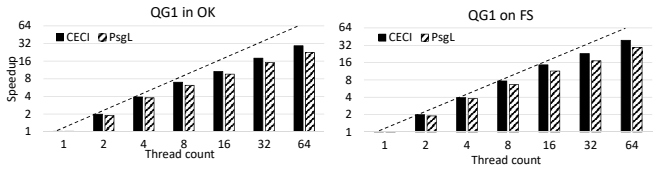


Figure 13: Scalability comparison against PsgL for increasing thread count on Server 1 while running QG1.

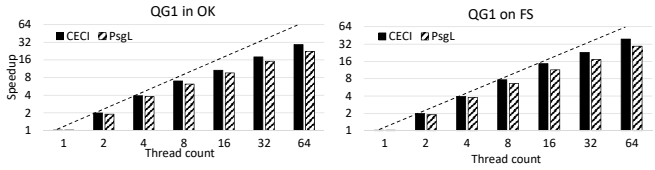


Figure 14: Scalability comparison against PsgL for increasing thread count on Server 1 while running QG4.

We evaluate the CPU usage of CECI system over the lifetime of the program. Initially, CPU usage is low since the process is either heavily serialized or dominated by IO loads. During CECI creation, the cpu usage gets only slightly larger due to massive serialization on creating CECI. During embedding enumeration phase, which constitutes more than 95% of the overall execution time, all cores are used to the maximal resulting in near 100% utilization on each core.

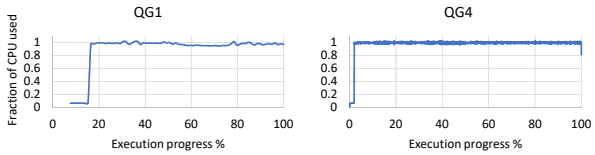


Figure 15: CPU usage while running different query graphs on OK data graph. The results are collected while running 32 OpenMP threads on Server 1.

The scalability of two distributed solutions described in Section 5 is examined on Cluster 1. We vary the machine

count from 1 to 16 while running four OpenMP threads on each machine. In general, the distributed systems have speedup curve that flats out earlier as compared to the shared memory setup because of the higher communication overhead. The reason for having flat line beyond 4 machines (each running 4-OpenMP thread) is due to insufficient workload on smaller graphs. Scalability on smaller graphs YT, WT, and LJ had the same effect on the shared memory framework beyond 16 threads. Note that larger graphs OK and FS have much better speedup with more machines.

Figure 16 shows the speedup of distributed system while the data graph is loaded in the memory of each machine. For 16 machines, we see maximum of 13.72× speedup for QG1 and 14.92 × speedup for QG4 while querying on FS graph. Figure 17 shows the scalability of distributed CECI when the shared data graph is accessed from a lustre file system. Here, the CECI creation overhead increases by a factor up to 100 due to increased IO overhead. However, the parallelization of CECI creation makes the solution scalable and the memory requirement in each compute node is reduced by up to $|E|$. As a result, we are still able to achieve speedup as high as 12.6 × for QG1 and 13.57× for QG4 with 16 machines.

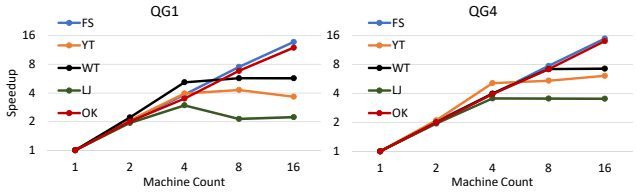


Figure 16: Speedup with increasing machine count for listing QG1 and QG4 on distributed solution that loads whole graph into the memory of each machine.

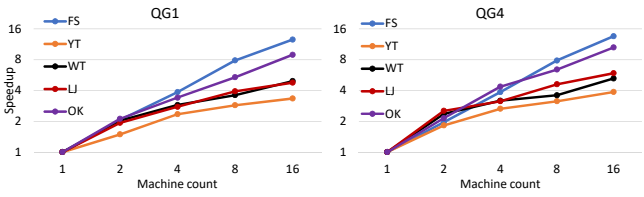


Figure 17: Scalability with increasing node count for listing QG1 and QG4 on distributed solution that uses networked graph storage (lustre).

6.6 CECI: Advantages and Overheads

After filtering and refinement, CECI reduces the number of false search paths significantly, more so when the query graph is complex. We compare the number of recursive calls made by CECI against that by PsgL. In backtracking matching algorithm, a new recursive call to matching routine has to be made every time an intermediate match is expanded by one tree-edge. The total number of recursive call is approximation of total search space [33]. The result in Figure 18 shows up to 44% reduction on the total number of recursive calls. Note that, the benefit increases as the query graph becomes more complex.

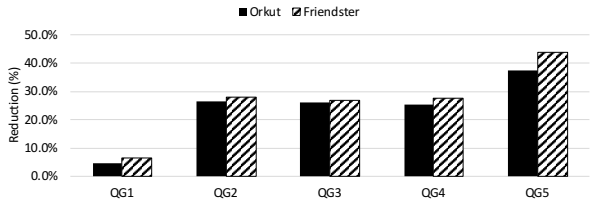


Figure 18: Percentage of reduction on recursive calls by CECI over PsgL for different queries.

The overhead for constructing and using CECI for subgraph listing is insignificant as compared to it's benefits. We implement a baseline parallel subgraph listing solution using graphs only and compared it with CECI based listing. As shown in Figure 19, subgraph listing with CECI, including creation overhead, is up to 2 orders magnitude faster than running listing with bare graph. This speedup comes from several factors including reduced search space, batched filtering and refinement, better locality of candidates.

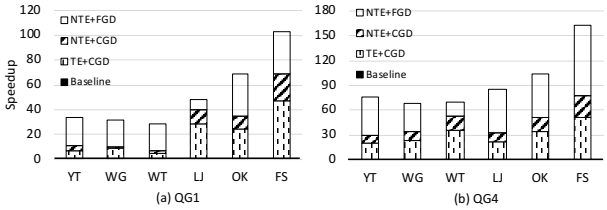


Figure 19: Breakdown of speedup obtained upon baseline technique into several techniques

With the whole graph in memory, CECI construction always took less than 5% of total run-time reported in Section 6.1. Loading graph accounts for less than 15% of total overhead and the majority of overhead is dominated by preprocessing, filtering and refinement. For network stored large graphs, the CECI construction can take up to 40% of the total run-time. In Figure 20, we have presented the breakdown of IO/compute/communication time for constructing CECI on FS graph. The majority of increased overhead comes from loading the partitioned sections of the data graph on demand basis as explained in Section 5.

7 RELATED WORK

Subgraph Matching: The inception of subgraph matching is backtracking based approach [54]. Later works VF2 [10] and QuickSI [46] enhance the matching order by picking the vertex connected to one of the already matched nodes. In addition, QuickSI [46] reduces the search space by selecting

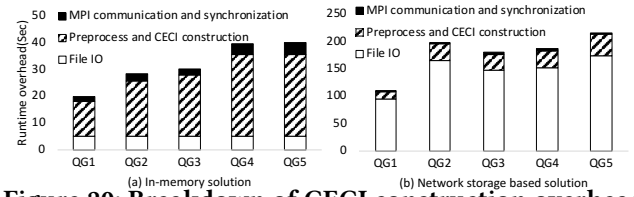


Figure 20: Breakdown of CECI construction overhead into IO, communication and computation.

infrequent nodes and edges first. Later, several solutions such as Spath [59], GraphQL [19], GADDI [57] are developed with the application of several heuristics such as pruning rules, join orders, and auxiliary data.

Lee et al. [33] have carried out a comprehensive survey on subgraph matching and have experimentally demonstrated that there is no universally best visit order and each filtering approach is good for only some datasets. Later, TurboIso [17] exploits the vertex similarity on query graph to reduce the overall workload. BoostIso [45] extends this idea to the data graph to further exploit the symmetries in data and query graphs for reducing the computation cost. CFLMatch [4] removes cardinality cost imposed by the low degree nodes by processing the dense portion of query earlier with the help of core-forest-leaf decomposition.

With the help of an efficient visit order and auxiliary data structures, it is possible to perform subgraph matching in reasonable time for small to medium sized data graphs. Unfortunately, these solutions fail to accommodate large graphs due to their inherent sequential nature – they list embeddings one by one.

Parallel Subgraph Listing solutions have gained popularity in recent years. Prior works [2, 32, 43, 50] decompose the query graph into sub-components, find the matches of each sub-component in parallel machines, and join the result from multiple machines to compute the embedding list. In contrary, [22] decomposes the data graph into distributed machines and applies the ideas from distributed relational query optimizations into subgraph matching. PsgL [47] enumerates all embeddings together by forming new intermediate embedding after each expansion and allocating it to suitable machine. DualSim [24] develops a disk based solution on which the adjacency list of each node is stored as a slotted page and at any time a specific combination of pages is loaded and subgraph listing is performed upon that section using *dual approach*. Despite providing significant parallelism, these solutions suffer from expensive join operation, huge amount of false candidates, and exponential-sized intermediate result. Our system improves upon these methods via a join-free approach, which only expands one embedding at a time on each worker in parallel.

Approximate Subgraph Matching: These solutions [52, 58] are useful for querying noisy and dynamic graph databases because they have better error tolerance. Similarly, approximate subgraph count estimators calculates the number of a given query graph in data graphs [3, 6, 12]. Although these works have better scalability, they do not provide the individual embeddings unlike CECI system.

Subgraph Containment Search is slightly different research area [5, 8, 26, 56, 60, 61] that finds whether a data graph contains at least one isomorphic embedding of given query graph. Although both subgraph containment search and subgraph listing involve subgraph isomorphism, the latter is considered more difficult since it requires enumerating every single embedding.

Subgraph Isomorphism in Streaming Graph is gaining more popularity as most of the real world graph data are continuously evolving. Several works [9, 25, 34] have already provided a solution for performing isomorphic subgraph search in streaming graphs. Recent framework [31] provides a framework to manage and analyze the evolving graph, which can be good platform to run streaming subgraph matching.

Graph data mining exploits subgraph isomorphism to extract the frequent subgraphs from data graphs [1, 8, 51]. Graph mining is essential for several problems involving topological as well as label feature based patterns in the data graphs. Examples of such applications include discovering 3D motifs in protein structures or chemical compounds, extracting network motifs or significant subgraphs from protein-protein network.

Other than subgraph isomorphism, several recent works on optimizing graph exploration [23, 35, 38] and triangle counting [20, 21] have been helpful in designing the CECI system. Finally, graph computing frameworks designed for in-memory [40, 48], distributed [7, 13], external [14, 18, 28, 29, 36], and heterogeneous [37] systems can also be integrated with CECI.

8 CONCLUSION AND FUTURE WORK

In this work, we have introduced a novel approach for subgraph listing with the help of CECI constructed by traversing the data graph. Particularly, with logical decomposition of data graph into embedding clusters we are able to parallelize the subgraph listing problem. In addition, with the help of intersection based embedding enumeration and workload balancing backed by cardinality based cost function, we are able to outperform state of the art parallel solutions, i.e. DualSim and PsgL by 6.08× and 34.82× on average respectively. One possible future research is to translate the logical decomposition into physical decomposition which enables subgraph listing in trillion edge graphs.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their feedback and suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774 to George Washington University.

REFERENCES

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. ScaleMine: Scalable parallel frequent subgraph mining in a single large graph. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 716–727.

[2] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *International Conference on Data Engineering (ICDE), 2013*. IEEE, 62–73.

[3] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Pattern Recognition in Bioinformatics* 24, 13 (2008), i241–i249.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *International Conference on Management of Data*. ACM, 1199–1214.

[5] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. 2010. Enhancing graph database indexing by suffix tree structure. *Pattern Recognition in Bioinformatics* (2010), 195–203.

[6] Ilaria Bordino, Debora Donato, Aristides Gionis, and Stefano Leonardi. 2008. Mining large networks with subgraph counting. In *IEEE International Conference on Data Mining, 2008*. IEEE, 737–742.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 1.

[8] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. 2007. Fg-index: towards verification-free query processing on graph databases. In *international conference on Management of data, SIGMOD, 2007*. ACM, 857–872.

[9] Sutanay Choudhury, Lawrence B. Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proceedings of 18th International Conference on Extending Database Technology*.

[10] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[11] Zhixi Fang, Peiyi Tang, P-C Yew, and C-Q Zhu. 1990. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Trans. Comput.* 39 (1990), 919–929.

[12] Mira Gonen, Dana Ron, and Yuval Shavitt. 2011. Counting stars and other small subgraphs in sublinear-time. *SIAM Journal on Discrete Mathematics* 25, 3 (2011), 1365–1411.

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: distributed graph-parallel computation on natural graphs.. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, Vol. 12. 2.

[14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, Vol. 14. 599–613.

[15] Graph500. [n. d.]. http://www.graph500.org/. Accessed: 2018-08-07.

[16] Joshua A Grochow and Manolis Kellis. 2007. Network motif discovery using subgraph enumeration and symmetry-breaking. In *RECOMB*, Vol. 4453. Springer, 92–106.

[17] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *International Conference on Management of Data, SIGMOD, 2013*. ACM, 337–348.

[18] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 77–85.

[19] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *International Conference on Management of Data, SIGMOD 2008*. ACM, 405–418.

[20] Yang Hu, Hang Liu, and H Howie Huang. 2018. High-Performance Triangle Counting on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–5.

[21] Yang Hu, Hang Liu, and H Howie Huang. 2018. Tricore: Parallel triangle counting on gpus. In *TriCore: Parallel Triangle Counting on GPUs*. IEEE, 0.

[22] Jiewen Huang, Kartik Venkatraman, and Daniel J Abadi. 2014. Query optimization of distributed pattern matching. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 64–75.

[23] Yuede Ji, Hang Liu, and H Howie Huang. 2018. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *2018 SC18: The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 731–742.

[24] Hyeonji Kim, Juneyoung Lee, Sourav S Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath HA Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1231–1245.

[25] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 411–426.

[26] Karsten Klein, Nils Kriege, and Petra Mutzel. 2011. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *International Conference on Data Engineering (ICDE), 2011*. IEEE, 1115–1126.

[27] Clyde P. Kruskal and Alan Weiss. 1985. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software engineering* 10 (1985), 1001–1016.

[28] P. Kumar and H. H. Huang. 2016. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 830–841.

[29] Pradeep Kumar and H. Howie Huang. 2017. Falcon: Scaling IO Performance in multi-SSD Volumes. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 41–53.

[30] Pradeep Kumar and H Howie Huang. 2017. SafeNVM: A Non-Volatile Memory Store with Thread-Level Page Protection. In *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE, 65–72.

[31] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *Proceedings of the 17th Usenix Conference on File and Storage Technologies (FAST'19)*.

[32] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.

[33] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 133–144.

[34] Youhuan Li, Lei Zou, M. Tamer Özsu, and Dongyan Zhao. 2018. Time Constrained Continuous Subgraph Search over Streaming Graphs. *Computing Research Repository* abs/1801.09240 (2018).

[35] Hang Liu and H Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 68.

[36] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-grained IO Management for Graph Computing. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 285–299.

[37] Hang Liu and H. Howie Huang. 2018. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. *CoRR* abs/1812.04070 (2018). https://arxiv.org/abs/1812.04070

[38] Hang Liu, H. Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 403–416.

[39] Scott Meyers. 2001. *Effective STL: 50 specific ways to improve your use of the standard template library*. Pearson Education.

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[41] Miles Ohlrich, Carl Ebeling, Eka Ginting, and Lisa Sather. 1993. Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *international Design Automation Conference*. ACM, 31–37.

[42] Saeed Omidi, Falk Schreiber, and Ali Masoudi-Nejad. 2009. MODA: an efficient algorithm for network motif discovery in biological networks. *Genes & genetic systems* 84, 5 (2009), 385–395.

[43] Todd Plantenga. 2013. Inexact subgraph isomorphism in MapReduce. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 164–175.

[44] N Pržulj, Derek G Corneil, and Igor Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22, 8 (2006), 974–980.

[45] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding Up Subgraph Isomorphism over Large Graphs. *Proceedings of the VLDB Endowment* 8, 5 (Jan. 2015), 617–628.

[46] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[47] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *International Conference on Management of Data, SIGMOD, 2014*. ACM, 625–636.

[48] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[49] SNAP Datasets: Stanford Large Network Dataset Collection. [n. d.]. http://snap.stanford.edu/data. Accessed: 2018-08-07.

[50] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment* 5, 9 (2012), 788–799.

[51] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Symposium on Operating Systems Principles*. ACM, 425–440.

[52] Yuanyuan Tian and Jignesh M Patel. 2008. Tale: A tool for approximate large graph matching. In *International Conference on Data Engineering, 2008*. IEEE, 963–972.

[53] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.

[54] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42.

[55] Yahoo! Search Marketing Advertiser-Phrase Bipartite Graph. [n. d.]. https://webscope.sandbox.yahoo.com/catalog.php?datatype=g. Accessed: 2018-08-07.

[56] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: a frequent structure-based approach. In *international conference on Management of data, SIGMOD 2004*. ACM, 335–346.

[57] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 192–203.

[58] Shijie Zhang, Jiong Yang, and Wei Jin. 2010. Sapper: Subgraph indexing and approximate matching in large graphs. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1185–1194.

[59] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351.

[60] Peixiang Zhao, Jeffrey Xu Yu, and Philip S Yu. 2007. Graph indexing: tree+ delta<= graph. In *international conference on Very large data bases*. VLDB Endowment, 938–949.

[61] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. 2008. A novel spectral coding in a large graph database. In *international conference on Extending database technology*. ACM, 181–192.