Fast and Robust Distributed Subgraph Enumeration

Xuguang Ren
Griffith University, Australia
x.ren@griffith.edu.au

Junhu Wang
Griffith University, Australia
j.wang@griffith.edu.au

Wook-Shin Han
POSTECH, Public of Korea
wshan@dlab.postech.ac.kr

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

ABSTRACT

We study the subgraph enumeration problem under distributed settings. Existing solutions either suffer from severe memory crisis or rely on large indexes, which makes them impractical for very large graphs. Most of them follow a synchronous model where the performance is often bottlenecked by the machine with the worst performance. Motivated by this, in this paper, we propose RADS, a Robust Asynchronous Distributed Subgraph enumeration system. RADS first identifies results that can be found using single-machine algorithms. This strategy not only improves the overall performance but also reduces network communication and memory cost. Moreover, RADS employs a novel region-grouped multi-round expand verify & filter framework which does not need to shuffle and exchange the intermediate results, nor does it need to replicate a large part of the data graph in each machine. This feature not only reduces network communication cost and memory usage, but also allows us to adopt simple strategies for memory control and load balancing, making it more robust. Several optimization strategies are also used in RADS to further improve the performance. Our experiments verified the superiority of RADS to state-of-the-art subgraph enumeration approaches.

PVLDB Reference Format:
Xuguang Ren, Junhu Wang, Wook-Shin Han, Jeffrey Xu Yu. Fast and Robust Distributed Subgraph Enumeration. PVLDB, 12(11): 1344-1356, 2019. DOI: https://doi.org/10.14778/3342263.3342272

Keywords
Distributed System, Asynchronous, Subgraph Enumeration

1. INTRODUCTION

Subgraph enumeration is the problem of finding all occurrences of a query graph in a data graph. Its solution is the basis for many other algorithms and it finds numerous applications[11]. This problem has been well studied under single machine settings [9][17]. However in the real world, the data graphs are often fragmented and distributed across different sites. This phenomenon highlights the importance of distributed systems of subgraph enumeration. Also, the increasing size of modern graph makes it hard to load the whole graph into memory, which further strengthens the requirement of distributed subgraph enumeration.

In recent years, many approaches have been proposed [1, 20, 12, 13, 5, 4, 16]. However all existing approaches are facing one or more of the following problems: (1) Memory crisis. Huge numbers of intermediate results are generated in [20, 12, 13]. Lacking of effective pruning and compression techniques results in a memory crisis to those approaches. (2) Heavy network communication. The large intermediate results of [20, 12, 13] need to be shuffled through the network. A big network latency may be generated when the data graph is distributed across different sites. (3) Heavy index. The index files of [16] can be many times larger than the data graph as shown in Table 2 of Section 8, and computing/maintaining such big indexes can be very expensive. (4) Synchronization delay. Most of the current systems are synchronous, hence they suffer from synchronization delay, making the overall performance equivalent to that of the slowest machine.

Different approaches may be facing different problems but may complement each other. One may argue that we can design a better system by simply stacking their successful ideas together. However this is very challenging across different system designs. For example, the compression strategy proposed in [16] cannot be used in [13] since [13] requires the intermediate results to be grouped by joining keys. The asynchronous method used in [4] is not suitable for join-based approaches [12, 13].

In this paper, we present RADS, a Robust Asynchronous Distributed Subgraph enumeration system. RADS is asynchronous, index free, with light communication cost, and also significantly reduces and compresses intermediate results. RADS is a solution which employs a multi-round framework and comprises several key ideas and optimization strategies. To be specific, we make the following contributions:

(1) We propose a method to identify embeddings that can be found on each local machine independent of other machines, and use single-machine algorithm to find
them. This strategy not only improves the overall performance, but also reduces network communication and memory cost.

(2) RADS employs a framework of region-grouped multi-round expand verify & filter where the key ideas are (i) to communicate undetermined edges and verification results instead of exchanging intermediate results, where the size of former is much smaller than that of the latter; (ii) to exchange parts of data graph in a region-grouped multi-round manner where the peak size of data graphs replicated in each machine is reduced.

(3) We propose effective memory control strategies to minimize the chance of memory crash, making our system more robust. Our strategy also facilitates workload balancing.

(4) We propose optimization strategies to further improve the performance. These include: (i) a set of rules to compute an efficient execution plan; and (ii) a dynamic data structure to compactly store intermediate results.

(5) We conduct extensive experiments which demonstrate that our system is not only faster than existing solutions (except for some queries using [16], which relies on heavy indexes), but also more robust.

Paper Organization We first introduce the preliminaries in Section 2. In Section 3, we present the architecture of our system RADS. The core framework R-Meef of RADS is given in Section 4. In Section 5, we present algorithms for computing the execution plan. In Section 6, we present the embedding trie data structure to compress our intermediate results. Our memory control strategy is given in Section 7. We present our experiments in Section 8, and discuss related works in Section 9. We conclude the paper in Section 10.

2. PRELIMINARIES

Data Graph & Query Graph Both the data graph and query graph (a.k.a query pattern) are assumed to be unlabeled, undirected, and connected graphs. We use $G = (V_G, E_G)$ and $P = (V_P, E_P)$ to denote the data graph and query graph respectively, where $V_G$ and $V_P$ are the vertex sets, and $E_G$ and $E_P$ are the edge sets. We will use data (resp. query) vertex to refer to vertices in the data (resp. query) graph. Generally, for any graph $g$, we use $V_g$ and $E_g$ to denote its vertex set and edge set respectively, and for any vertex $v$ in $g$, we use $adj(v)$ to denote $v$’s neighbour set and use $deg(v)$ to denote the degree of $v$.

Subgraph Isomorphism Given a data graph $G$ and a query pattern $P$, $P$ is subgraph isomorphic to $G$ if there exists an injective function $f: V_P \rightarrow V_G$ such that for any edge $(u_1, u_2) \in E_P$, there exists an edge $(f(u_1), f(u_2)) \in E_G$. The injective function is also known as an embedding of $P$ in $G$, and it can be represented as a set of vertex pairs $(u, v)$ where $u \in V_P$ is mapped to $v \in V_G$. We will use $\mathbb{E}_G(P)$ to denote the set of all embeddings of $P$ in $G$.

The problem of subgraph enumeration is to find the set $\mathbb{E}_G(P)$. In the literature, subgraph enumeration is also referred to as subgraph isomorphism search [14][9][17] and subgraph listing [11][20].

Partial Embedding A partial embedding of graph $P$ in graph $G$ is an embedding in $G$ of a vertex-induced subgraph of $P$.

Symmetry Breaking A symmetry breaking method based on automorphism is conventionally used to reduce duplicate embeddings [7]. As a result the data vertices in the final embeddings should follow a preserved order of the query vertices. We apply this technique in this paper by default and we will specify the preserved order when necessary.

Graph Partition & Storage Given a data graph $G$ and $m$ machines $\{M_1, \ldots, M_m\}$ in a distributed environment, a partition of $G$ is denoted $\{G_1, G_2, \ldots, G_m\}$ where $G_i$ is the partition located in the $i^{th}$ machine $M_i$. In this paper, we assume each partition is stored as an adjacency-list. For any data vertex $v$, we assume its adjacency-list is stored in a single machine $M_i$ and we say $v$ is owned by $M_i$ (or $v$ resides in $M_i$). We call $v$ a foreign vertex of $M_i$ if $v$ is not owned by $M_i$.

For any $v$ owned by $M_i$, we call $v$ a border vertex if at least one of its neighbors resides in other machines than $M_i$. Otherwise we call it a non-border vertex. We use $V_G^b_i$ to denote the set of all border vertices in $M_i$.

3. RADS ARCHITECTURE

In this section, we present the architecture of RADS as shown in Figure 1.

![Figure 1: RADS Architecture](image)

Given a query pattern $P$, within each machine, RADS first launches a process of single-machine enumeration (SM-E) and a Daemon thread, simultaneously. After SM-E finishes, RADS launches a R-Meef thread subsequently. Note that the R-Meef threads of different machines may start at different time.

- Single-Machine Enumeration The idea of SM-E is to try to find a set of local embeddings using a single-machine algorithm, such as TurboIso[9], which does not involve any distributed processing. The subsequent distributed process only has to find the remaining embeddings. This strategy can not only boost the overall enumeration efficiency but also significantly reduce the memory cost and communication cost of the subsequent distributed process. Moreover the local embeddings can be used to estimate
the space cost of a region group, which will help to effectively control the memory usage (see Section 7).

We first define the concepts of border distance and span, which will be used to identify embeddings that can be found by SM-E.

Definition 1. Given a graph partition $G_t$ and a data vertex $v$ in $G_t$, the border distance of $v$ w.r.t. $G_t$, denoted $BD_{G_t}(v)$, is the minimum shortest distance between $v$ and any border vertex of $G_t$, that is

$$BD_{G_t}(v) = \min_{v' \in V^b_{G_t}} \text{dist}(v, v') \tag{1}$$

where $\text{dist}(v, v')$ is the shortest distance between $v$ and $v'$.

Definition 2. Given a query pattern $P$, the span of query vertex $u$, denoted $\text{Span}_P(u)$, is the maximum shortest distance between $u$ and any other vertex of $P$, that is

$$\text{Span}_P(u) = \max_{u' \in V_P} \text{dist}(u, u') \tag{2}$$

Proposition 1. Given a data vertex $v$ of $G_t$ and a query vertex $u$ of $P$, if $\text{Span}_P(u) \leq BD_{G_t}(v)$, then there will be no embedding $f$ of $P$ in $G$ such that $f(u) = v$ and $f(u')$ is not owned by $M_t$, where $u' \in P, u' \neq u$.

Proposition 1 states that if the border distance of $v$ is not smaller than the span of query vertex $u$, there will be no cross-machine embeddings (i.e., embeddings where the query vertices are mapped to data vertices residing in different machines) which map $u$ to $v$. The proof of Proposition 1 can be found in the extended version of this paper [18].

Let $u_{\text{start}}$ be the starting query vertex (namely, the first query vertex to be mapped) and $C(u_{\text{start}})$ be the candidate vertex set of $u_{\text{start}}$ in $G_t$. Let $C_1(u_{\text{start}}) \subseteq C(u_{\text{start}})$ be the subset of candidates whose border distance is no less than the span of $u_{\text{start}}$. According to Proposition 1, all embeddings that map $u_{\text{start}}$ to a vertex in $C_1(u_{\text{start}})$ can be found using a single-machine subgraph enumeration algorithm over $G_t$, independent of other machines. In RADS, the candidates in $C_1(u_{\text{start}})$ will be processed by SM-E, and the other candidates will be processed by the subsequent distributed process. The SM-E process is simple, and we will next focus on the distributed process. For presentation simplicity, from now on we say a candidate vertex of $u_{\text{start}}$, we mean a candidate vertex in $C(u_{\text{start}}) - C_1(u_{\text{start}})$, unless explicitly stated otherwise.

The distributed process consists of a daemon thread and a subgraph enumeration thread R-Meef:

- Daemon Thread listens to requests from other machines and supports four functionalities:
  1. verifyE is to return the edge verification results for a given request consisting of vertex pairs. For example, given a request $\{(v_0, v_1), (v_2, v_3)\}$ posted to $M_1$, $M_1$ will return \{true, false\} if $(v_0, v_1)$ is an edge in $G_t$ while $(v_2, v_3)$ is not.
  2. fetchV is to return the adjacency-lists of the requested vertices of the data graph. The requested vertices sent to machine $M_t$ must reside in $M_t$.
  3. checkR is to return the number of unprocessed region groups (which is a group of candidate data vertices of the starting query vertex, see Section 4) of the local machine (i.e., the machine on which the thread is running).
  4. shareR is to return an unprocessed region group of the local machine to the requester machine. shareR will also mark the region group sent out as processed.

- R-Meef Thread is the core subgraph enumeration thread. When necessary, the local R-Meef thread sends verifyE requests and fetchV requests to the Daemon threads of other machines, and the other machines respond to these requests accordingly. We present the details of R-Meef in next Section.

Once a local machine finishes processing its own region groups, it will broadcast a checkR request to the other machines. Upon receiving the numbers of unfinished region groups from other machines, it will send a shareR request to the machine with the maximum number of unprocessed region groups. Once it receives a region group, it will process it on the local machine. checkR and shareR are for load balancing purposes only, and they will not be discussed further in this paper.

4. THE R-Meef FRAMEWORK

Before presenting the details of the R-Meef framework, we need the following definitions.

Definition 3. Given a partition $G_t$ of a data graph $G$ located in machine $M_t$, and a query pattern $P$, an injective function $f_{G_t}: V_P \to V_G$ is called an embedding candidate (EC) of $P$ w.r.t. $G_t$ if for any edge $(u, u') \in E_P$, there exists an edge $(f_{G_t}(u), f_{G_t}(u')) \in E_G$, provided either $f_{G_t}(u) \in V_G$, or $f_{G_t}(u') \in V_G$.

We use $\overline{E}_{G_t}(P)$ to denote the set of ECs of $P$ w.r.t. $G_t$. Note that for an EC $f_{G_t}$ and a query vertex $u$, $f_{G_t}(u)$ is not necessarily owned by $G_t$. That is, the adjacency-list of $f_{G_t}(u)$ may be stored in other machines. For any query edge $(u, u')$, an EC only requires that the corresponding data edge $(f_{G_t}(u), f_{G_t}(u'))$ exists if at least one of $f_{G_t}(u)$ and $f_{G_t}(u')$ resides in $G_t$. Therefore, an EC may not be an embedding. Intuitively, the existence of the edge $(f_{G_t}(u), f_{G_t}(u'))$ can only be verified in $G_t$ if one of its end vertices resides in $G_t$. Otherwise the existence of the edge cannot be verified in $M_t$, and we call such edges undetermined edges.

Definition 4. Given an EC $f_{G_t}$ of query pattern $P$, for any edge $(u, u') \in E_P$, we say $(f_{G_t}(u), f_{G_t}(u'))$ is an undetermined edge of $f_{G_t}$ if neither $f_{G_t}(u)$ nor $f_{G_t}(u')$ is in $G_t$.

Example 1. Consider a partition $G_t$ of a data graph $G$ and a triangle query pattern $P$ where $V_P = \{u_0, u_1, u_2\}$. The mapping $f_{G_t} = \{(u_0, v_0), (u_0, v_1), (u_0, v_2)\}$ is an EC of $P$ in $G$ w.r.t. $G_t$ if $v_0 \in V_{G_t}$, $v_1 \in \text{adj}(v_0)$ and $v_2 \in \text{adj}(v_0)$ and neither $v_1$ nor $v_2$ resides in $G_t$. $(v_1, v_2)$ is an undetermined edge of $f_{G_t}$.

Obviously if we want to determine whether $f_{G_t}$ is actually an embedding of the query pattern, we have to verify its undetermined edges in other machines. For any undetermined edge $e$, if its two end vertices reside in two different machines, we can use either of them to verify whether $e \in E_G$ or not. To do that, we need to send a verifyE request to one of the machines.
Note that it is possible that an undetermined edge is shared by multiple ECs. To reduce network traffic, we do not send verifyE requests once for each individual EC, instead, we build an edge verification index (EVI) and use it to identify ECs that share undetermined edges. We assume each EC is assigned an ID (to be discussed in Section 6).

**Definition 5.** Given a set $\tilde{R}_{G_i}(P)$ of ECs, the edge verification index (EVI) of $\tilde{R}_{G_i}(P)$ is a key-value map $I$ where

1. for any tuple $(e, IDs) \in I$, $e$ is a vertex pair $(v, v')$; and the value $I$ is the set of IDs of the ECs in $\tilde{R}_{G_i}(P)$ of which $e$ is an undetermined edge.

2. for any undetermined edge $e$ of $f_{G_i} \in \tilde{R}_{G_i}(P)$, there exists a unique tuple in $I$ with $e$ as the key and the ID of $f_{G_i}$ in the value.

Intuitively, the EVI groups the ECs that share each undetermined edge together. It is straightforward to see:

**Proposition 2.** Given data graph $G$, query pattern $P$, and an edge verification index $I$, for any $(e, IDs) \in I$, if $e \notin E_G$, then none of the ECs corresponding to $IDs$ can be an embedding of $P$ in $G$.

Like SEED [13] and TwinTwig [12], we decompose the pattern graph into small decomposition units.

**Definition 6.** A decomposition of query pattern $P$ is a sequence of decomposition units $DE = (dp_0, \ldots, dp_i)$ where every $dp_i \in DE$ is a subgraph of $P$ such that

1. The vertex set of $dp_i$ consists of a pivot vertex $v$ and a non-empty set of leaf vertices, all of which are vertices in $V_P$; and for every $u' \in LF$, $(v, u') \in E_P$.

2. The edge set of $dp_i$ consists of two parts, $E^{star}_{dp_i}$ and $E^{sib}_{dp_i}$; where $E^{star}_{dp_i} = \{ (dp_i, piv, u') \}$ is the set of edges between the pivot vertex and the leaf vertices, and $E^{sib}_{dp_i} = \{ (u, u') \in EF_i \}$ is the set of edges between the leaf vertices.

3. $\bigcup_{dp_i \in DE} (V_{dp_i}) = V_P$, and for $i < j$, $V_{dp_i} \cap V_{dp_j} = \emptyset$.

Note condition (3) in the above definition says the leaf vertices of each decomposition unit do not appear in the previous units. Unlike the decompositions in SEED [13] and TwinTwig [12], our decomposition unit is not restricted to stars and cliques, and $\bigcup_{dp_i \in DE} (E_{dp_i})$ may be a proper subset of $E_P$.

**Example 2.** Consider the query pattern in Figure 2 (a), we may have a decomposition $(dp_0, dp_1, dp_2, dp_3)$ where $dp_0, piv = u_0, dp_0, LF = \{ u_1, u_2, u_3 \}$, $dp_1, piv = u_1, dp_1, LF = \{ u_4, u_5 \}$, and $dp_3, piv = u_0, dp_3, LF = \{ u_8, u_9 \}$. Note that the edge $(u_4, u_5)$ is not in any decomposition unit.

In the above example, the edge $(u_4, u_5)$ is between vertices that belong to different units. We call such edges cross-unit edges. More formally, let $PL = (dp_0, \ldots, dp_i)$ be a decomposition. For each $i \in [0, l]$, we define $E^{cro}_{dp_i} = \{ (u_1, u_2) \in E_P | u_1 \notin \bigcup_{j < i} V_{dp_j} \} \{ dp_i, piv \}, u_2 \in dp_i, LF$. And call the edges in $E^{star}_{dp}, E^{sib}_{dp}$ and $E^{cro}_{dp}$ the expansion edges, sibling edges, and cross-unit edges respectively. The sibling edges and cross-unit edges are both called verification edges. Note that the expansion edges of all the units span a spanning tree of $P$, and the verification edges are the edges not in the spanning tree. Consider $dp_2$ in Example 2, we have $E^{star}_{dp_2} = \{ (u_2, u_3), (u_2, u_6) \}$, $E^{sib}_{dp_2} = \{ (u_3, u_4) \}$.

![Figure 2: Running Example](image)

**Algorithm 1: R-Meef Framework**

**Input:** Query pattern $P$, partition $G_i$ on machine $M_i$, execution plan $PL$

**Output:** $\tilde{R}_{G_i}(P)$

1. $RG = \{ r_g \ldots g_k \} \gets regionGroups(C(dp_0, piv, M_i))$
2. for each region group $rg \in RG$
   1. init embedding trie $ET$ with size $|V_P|$  
   2. init edge verification index $I$
   3. for each data vertex $v \in rg$
      1. $f \gets (dp_0, piv, v)$
      2. $l \gets expandEmbedTrie(f, M_i, dp_0, ET)$
      3. $R \gets verifyForeignE(l)$
      4. $R \gets filterFailedEmbed(R, I, ET)$
   4. for Round $i = 1$ to $|PL|$
      1. $l \leftarrow ET$
      2. fetchForeignV(i)
      3. for each $f \in I$
         1. $ET = expandEmbedTrie(f, M_i, dp_0, ET)$
         2. $R \gets verifyForeignE(l)$
         3. $R \leftarrow filterFailedEmbed(R, I, ET)$
      4. $R \leftarrow R \cup ET$

Within each machine, we group the candidate data vertices of $dp_0, piv$ within $M_i$ into region groups (Line 1). For each region group $rg$, a multi-round mapping process is conducted (Line 2 to 18). Within each round, we use a data
structure $E_T$ (embedding trie) to save the generated intermediate results, i.e., embeddings and embedding candidates (Line 3). The EVI $I$ is initialized in Line 4, which will be reset for each round of processing (Line 11).

(1) First Round (round 0). Starting from each candidate $v$ of $rg$, we match $v$ to $dp_0.piv$ in the execution plan. After the pivot vertex is matched, we find all the ECs of $dp_0$ with respect to $M_1$ and compress them into $E_T$. We use a function expandEmbedTrie to represent this process (Line 7). For each EC compressed in $E_T$, its undetermined edges need to be verified in order to determine whether this EC is an embedding of $dp_0$. We record this information in the edge verification index $I$, which is constructed in the expandEmbedTrie function. After we have the EVI $I$ in $M_1$, we send a verifyE request to verify those undetermined edges within $I$ in the machine which has the ability to verify it (function verifyForeignE in Line 8). After the edges in $I$ are all verified, we remove the failed ECs from $E_T$ (Line 9).

(2) Other Rounds. For each of the remaining rounds of the execution plan, we first clear the EVI $I$ from the previous round (Line 11). In the $i$th round, we want to find all the ECs of $P_i$ based on the embeddings in $RG_i(P_{i-1})$ (where $dp_i.piv$ has been matched). The process is to expand every embedding $f$ of $RG_i(P_{i-1})$ with each embedding candidate of $dp_i$ within the neighbourhood of $f(dp_i.piv)$. If not all the data vertices matched to $dp_i.piv$ by the ECs in $RG_i(P_{i-1})$ reside in $M_i$, we will have to fetch the adjacency-lists of those foreign vertices from other machines in order to expand from them. A sub-procedure fetchForeignE is used to represent this process (Line 12). After fetching, for each embedding $f$ of $RG_i(P_{i-1})$, we find all the ECs of $P_i$ by expanding from $f(dp_i.piv)$ (Line 14). The found ECs are compressed into $E_T$. Then verifyForeignE and filterFailedEmbed are called to make sure that the failed ECs are filtered out from the embedding trie, which will only contain the actual embeddings of $P_i$, i.e., $RG_i(P_i)$ (Line 15, 16).

After all the rounds of this region group have finished, we have a set of embeddings of $P$ compressed into $E_T$. The results obtained from all the region groups are put together to obtain the embeddings found by $M_i$.

One important thing to note is that if a foreign vertex is already cached in the local machine, for the undetermined edges attached to this vertex, we can verify them locally without sending requests to other machines. Also we do not re-fetch any foreign vertex if it is already cached previously.

Example 3. Consider the data graph $G$ in Figure 2, where the vertices marked with dashed border lines reside in $M_1$ and the other vertices reside in $M_2$. Consider the pattern $P$ and execution plan $PL$ given in Example 2. We assume the preserved orders due to symmetry breaking are: $u_1 < u_2$, $u_3 < u_5$, $u_4 < u_6$, and $u_7 < u_8$.

There are two vertices $\{v_0, v_2\}$ in $M_1$ and two vertices $\{v_1, v_{10}\}$ in $M_2$ with a degree not smaller than that of $dp_0.piv$. Therefore in $M_1$, we have $C(dp_0.piv) = \{v_0, v_2\}$ and in $M_2$ we have $C(dp_0.piv) = \{v_1, v_{10}\}$. After grouping, assume we have $RG = \{rg_0, rg_1\}$ where $rg_0 = \{v_0\}$ and $rg_1 = \{v_2\}$ in $M_1$, and $RG = \{rg_0\}$ where $rg_0 = \{v_2, v_{10}\}$ in $M_2$.

Consider the region group $rg_0$ in $M_1$. In round 0, we first match $v_0$ to $dp_0.piv$. Expanding from $v_0$, we may have ECs including by not to limit to (we lock $u_7$ to $v_7$ for easy demonstration):

$$f_{g_0} = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_7, v_7)\}$$

$$f_{g_1} = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_7, v_7)\}$$

$$f_{g_2} = \{(u_0, v_0), (u_1, v_0), (u_2, v_1), (u_7, v_7)\}$$

These ECs are compressed into $E_T$. Note that a mapping such as $\{(u_0, v_0), (u_1, v_1), (u_2, v_11), (u_7, v_7)\}$ is not an EC of $dp_0$ w.r.t $M_1$ since $(v_1, v_{11})$ can be locally verified to be non-existent. Since the undetermined edge $(v_1, v_{10})$ of $f_{g_0}$ cannot be determined in $M_1$, we put $\{(v_1, v_{10}), < f_{g_0} >\}$ into the EVI $I$. We then ask $M_2$ to verify the existence of the edge. $M_2$ returns false, therefore $f_{g_0}$ will be removed from $E_T$.

In round 1, we have two embeddings $RG_1(P_0) = \{f_{g_1}, f_{g_2}\}$ to start with. To expand $f_{g_1}$ and $f_{g_2}$, we need to fetch the adjacency-lists of $v_1$ and $v_{10}$ respectively. We send a single fetchV request to fetch the adjacency-lists of $v_1$ and $v_{10}$ from $M_2$. After expansion from $v_1$, we get a single embedding $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_7, v_7)\} \in RG_1(P_1)$. There is no embedding of $P_1$ expanded from $v_{10}$. Hence $f_{g_2}$ will be removed from the embedding trie.

In round 2, we expand from $v_2$ to get the ECs of $P_2$. $dp_2.piv$ was already mapped to $v_2$ as seen above, and $v_2$ has neighbors $v_5$ and $v_{10}$ that are not matched to any query vertices. Since there are sibling edge $(u_5, u_6)$ and cross-unit edge $(u_4, u_5)$ in $P_2$, we need to verify the existence of $(v_5, v_7)$ and $(v_6, v_9)$ if we want to map $u_5$ to $v_5$ and map $u_6$ to $v_9$. The existence of both $(v_4, v_5)$ and $(v_6, v_9)$ can be verified locally. Similarly if we want to map $u_5$ to $v_5$, $u_6$ to $v_{10}$, we will have to verify the existence of $(v_5, v_{10})$, and so on. It can be locally verified that $(v_5, v_{10})$ does not exist, and remotely verified that $(v_6, v_9)$ does not exist. Therefore, at the end of this round, we will get a single embedding for $P_2$, which extends the embedding for $P_1$ by mapping $u_5$ to $v_5$ and $u_6$ to $v_9$ respectively. We expand the embedding trie accordingly.

Following the above process, after we process the last round, we have an embedding of $P$ starting from region group $rg_0$ in machine $M_1$, which will be saved in $E_T$:

$$f_{g_0} = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_5), (u_6, v_6), (u_7, v_7), (u_8, v_9), (u_9, v_{11})\}$$

In order to achieve the best performance, each component of Algorithm 1 should be carefully designed. In the following sections, we address the issues one by one.

5. Computing Execution Plan

It is obvious that we may have multiple valid execution plans for a query pattern and different execution plans may have different performance. The challenge is how to find the most efficient one among them? In this section, we present some heuristics to find a good execution plan.

5.1 Minimizing Number of Rounds

Given query pattern $P$ and an execution plan $PL$, we have $|PL| + 1$ rounds for each region group, and once all the rounds are processed we will get the set of final embeddings. Also, within each round, the workload can be shared. To be specific, a single undetermined edge $e$ may be shared by multiple ECs. If these embedding candidates are generated in the same round, the verification of $e$ can be shared by all of them. The same applies to the foreign vertices where
the cost of fetching and memory space can be shared among multiple embedding candidates if they happen to be in the same round. Therefore, our first heuristic is to minimize the number of rounds (namely, the number of decomposition units) so as to maximize the workload sharing.

Here we present a technique to compute a query execution plan, which guarantees a minimum number of rounds. Our technique is based on the concepts of maximum leaf spanning tree [6] and minimum connected dominating set.

Definition 8. A maximum leaf spanning tree (MLST) of pattern \( P \) is a spanning tree of \( P \) with the maximum number of leaves (a leaf is a vertex with degree 1). The number of leaves in an MLST of \( P \) is called the maximum leaf number of \( P \), denoted \( l_P \).

Definition 9. A connected dominating set (CDS) of \( P \) is a subset \( D \) of \( V_P \) such that (1) \( D \) is a dominating set of \( P \), that is, any vertex of \( P \) is either in \( D \) or adjacent to a vertex in \( D \), and (2) the subgraph of \( P \) induced by \( D \) is connected. A minimum connected dominating set (MCDS) is a CDS with the smallest cardinality among all CDSs. The number of vertices in a MCDS is called the connected domination number, denoted \( c_P \).

It is shown in [3] that \( |V_P| = c_P + l_P \).

Theorem 1. Given a pattern \( P \), any execution plan of \( P \) has at least \( c_P \) decomposition units, and there exists an execution plan with exactly \( c_P \) decomposition units.

Proof. Suppose \( \{dp_0, \ldots, dp_k\} \) is an execution plan with \( k + 1 \) decomposition units. The pivot vertices of the decomposition units form a connected dominating set of \( P \). Therefore, \( k + 1 \geq c_P \). This proves any execution plan has at least \( c_P \) decomposition units.

Suppose \( T \) is a MLST of \( P \). From \( |V_T| = c_T + l_T \) we know the number of non-leaf vertices in \( T \) is \( c_T \). We can construct an execution plan by choosing one of the non-leaf vertices \( v_0 \) as \( dp_0.piv \), and all neighbors of \( v_0 \) in \( T \) as the vertices in \( dp_0.LF \). Regarding \( v_0 \) as the root of the spanning tree \( T \), we then choose each of the non-leaf children \( v_i \) of \( v_0 \) in \( T \) as the pivot vertex of the next decomposition unit \( dp_i.piv \), and all children of \( v_i \) as the vertices in \( dp_i.LF \). Repeat this process until every non-leaf vertex of \( T \) becomes the pivot vertex of a decomposition unit. This decomposition has exactly \( c_T \) units, and it forms an execution plan. This shows that there exists an execution plan with \( c_T \) decomposition units. \( \square \)

Theorem 1 indicates that \( c_P \) is the minimum number of rounds of any execution plan. The above proof provides a method to construct an execution plan with \( c_T \) rounds from a MLST.

Example 4. Consider the pattern \( P \) in Figure 2, it can be easily verified that the tree obtained by removing the edges (\( u_1, u_2 \), (\( u_3, u_4 \)), (\( u_4, u_5 \), (\( u_5, u_6 \)) and (\( u_6, u_7 \)) is a MLST of \( P \). Choosing \( u_7 \) as the root, we will get a minimum round execution plan \( PL_1=\{dp_0, dp_1, dp_2\} \) where \( dp_0.piv = u_0 \), \( dp_0.LF = \{u_1, u_2, u_7, u_6, u_5\} \), \( dp_1.piv = u_1 \), \( dp_1.LF = \{u_0, u_3, u_4\} \), \( dp_2.piv = u_2 \), \( dp_2.LF = \{u_5, u_6\} \).

5.2 Minimizing the span of \( dp_0.piv \)

Given a pattern \( P \), multiple execution plans may exist with the minimum number of rounds, while their \( dp_0.piv \) can be different. Our second heuristic is to choose the plan(s) whose \( dp_0.piv \) have the smallest span. This strategy will maximize the number of embeddings that can be found using SM-E. Recall the RADS architecture where \( dp_0.piv \) is the starting query vertex \( u_{start} \), based on Proposition 1, we know that the more candidate vertices of \( dp_0.piv \) can be processed in SM-E, the more workload can be separated from the distributed processing, and therefore the more communication cost and memory usage can be reduced.

Consider the pattern in Figure 3, the bold edges demonstrate a MLST based on which both \( u_3 \) and \( u_4 \) can be chosen as \( dp_0.piv \). The execution plans from them have the same number of rounds. However, \( Span_{\mu}(u_3) = 2 \) while \( Span_{\mu}(u_4) = 3 \). Therefore we choose the plan with \( u_3 \) as the \( dp_0.piv \).

5.3 Maximizing Filtering Power

Given a pattern \( P \), multiple execution plans may exist with the minimum number of rounds and their \( dp_0.piv \) have the same smallest span. Here we use the third heuristic which is to choose plans with more verification edges in the earlier rounds. The intuition is to maximize the filtering power of the verification edges as early as possible. To this end, we propose the following score function \( SC(PL) \) for an execution plan \( PL = \{dp_0, \ldots, dp_l\} \):

\[
SC(PL) = \sum_{dp_i \in PL} \frac{1}{(i+1)^p} \times (|E_{dp_i}^{vb}| + |E_{dp_i}^{vr}|) \tag{3}
\]

\( |E_{dp_i}^{vb}| + |E_{dp_i}^{vr}| \) is the number of verification edges in round \( i \), and \( \rho \) is a positive parameter used to tune the score function. In our experiments we use \( \rho = 1 \). The function \( SC(PL) \) calculates a score by assigning larger weights to the verification edges in earlier rounds (since \( \frac{1}{(i+1)^p} > \frac{1}{(j+1)^p} \) if \( i < j \)).

Example 5. Consider the query plans \( PL_1 \) and \( PL_2 \) in Example 4. The total number of verification edges in these plans are the same. In \( PL_1 \), the number of verification edges for the first, second and third round is 2, 1, 2 respectively. In \( PL_2 \), the number of verification edges for the three rounds is 1, 2, and 2 respectively. Therefore, we prefer \( PL_1 \). Using \( \rho = 1 \), we can calculate the scores of the two plans as follows:

\[
SC(PL_1) = 2/1 + 1/2 + 2/3 \approx 3.2 \quad SC(PL_2) = 1/1 + 2/2 + 2/3 \approx 2.7
\]

When several minimum-round execution plans have the same score, we use another heuristic rule to choose the best one from them: the larger the degree of the pivot vertex, the earlier we process the unit. The pivot vertex with a larger degree has a stronger power to filter unpromising candidates.

6. EMBEDDING TREE

In this section we present the data structure used for compressing the intermediate results as well as the algorithms to maintain it. We first define a matching order, following which the query vertices are matched. It is also the order the nodes in the embedding trie are organized.
Definition 10. Given a query execution plan $PL = \{dp_0, \ldots, dp_l\}$ of pattern $P$, the matching order w.r.t $PL$ is a relation $\prec$ defined over the vertices of $P$ that satisfies the following conditions:

1. $dp_i.piv \prec dp_j.piv$ if $i < j$;
2. For any two vertices $u_1 \in dp_i.LF$ and $u_2 \in dp_j.LF$, $u_1 \prec u_2$ if $i < j$.
3. For $i \in [0, l]$:
   
   (i) $dp_i.piv \prec u$ for all $u \in dp_i.LF$;
   
   (ii) for any vertices $u_1, u_2 \in dp_i.LF$ that are not the pivot vertices of other units, $u_1 \prec u_2$ if $deg(u_1) > deg(u_2)$, or $deg(u_1) = deg(u_2)$ and the vertex ID of $u_1$ is less than that of $u_2$;
   
   (iii) if $u_1 \in dp_i.LF$ is a pivot vertex of another unit, and $u_3 \in dp_i.LF$ is not a pivot vertex of another unit, then $u_1 \prec u_2$.

It is easy to verify $\prec$ is a strict total order over $V_P$. Following the matching order, the vertices of $P$ can be arranged into an ordered list. Consider the execution plan $PL_1$ in Example 4. The vertices in the query can be arranged as $(u_0, u_1, u_2, u_7, u_8, u_9, u_3, u_4, u_5, u_6)$ according to the matching order.

Let $PL = \{dp_0, \ldots, dp_l\}$ be an execution plan, $PL_i$ be the subgraph of $P$ induced from the vertices in $dp_0 \cup \cdots \cup dp_i$ (as defined in Section 4), and $R$ be a set of results (i.e., embeddings or embedding candidates) of $P$. For easy presentation, we assume the vertices in $PL_i$ have been arranged into the list $u^0, u^1, \ldots, u^n$ by the matching order, that is, the query vertex at position $j$ is $u^j$. Then each result of $PL_i$ can be represented as a list of corresponding data vertices.

Next we formally define embedding trie and present the algorithms for the maintenance of the embedding trie.

6.1 Structure of the Embedding Trie

Definition 11. Given a set $R$ of results of $P$, the embedding trie of $R$ is a collection of trees used to store the results in $R$ such that:

1. Each tree represents a set of results that map $u^0$ to the same data vertex.
2. Each tree node $N$ has
   
   • $w$: a data vertex
   
   • parent$N$: a pointer pointing to its parent node (the pointer of the root node is null).
   
   • childCount: the number of child nodes of $N$.
3. If two nodes have the same parent, then they store different data vertices.
4. Every leaf-to-root path represents a result in $R$, and every result in $R$ is represented as a unique leaf-to-root path.
5. If we divide the tree nodes into different levels such that the root nodes are at level 0, the children of the root nodes are at level 1 and so on, then the tree nodes at level $j$ ($j \in [0, l]$) store the set of values $\{f(u^j) | f \in R\}$.

Example 6. Consider $P_0$ in Example 3, where the vertices are ordered as $u_0, u_1, u_2$ according to the matching order. There are three ECs of $P_0$: $(v_0, v_1, v_2)$, $(v_0, v_1, v_9)$ and $(v_0, v_9, v_11)$. These results can be stored in a tree shown in Figure 4(a). When the second EC is filtered out, we have $\mathcal{R}_C(P_0)$ compressed in a tree as shown in Figure 4(b). The first EC can be expanded to an EC of $P_1$ (where the list of vertices of $P_1$ are $(u_0, u_1, u_2, u_3, u_4)$, which is as shown in Figure 4(c).

Although the structure of embedding trie is simple, it has some nice properties: (1) Compression Storing the results in the embedding trie saves space than storing them as a collection of lists. (2) Unique ID For each result in the embedding trie, the address of its leaf node in memory can be used as the unique ID. (3) Retrieval Given a particular ID represented by a leaf node, we can easily follow its pointer parent$N$ step-by-step to retrieve the corresponding result. (4) Removal To remove a result with a particular ID, we can remove its corresponding leaf node and decrease the childCount of its parent node by 1. If ChildCount of this parent node reaches 0, we remove this parent node. This process recursively affects the ancestors of the leaf node.

6.2 Maintaining the embedding trie

Recall that in Algorithm 1, given an embedding $f$ of $P_{l-1}$, the function expandEmbedTrie is used to search for the ECs of $dp_l$ within the neighbourhood of the mapped data vertex $v_{piv}$, where $v_{piv} = f(dp_l.piv)$. Moreover, the expandEmbedTrie function handles the task of expanding the embedding trie $ET$ by concatenating $f$ with each newly found EC of $dp_l$. If an EC is filtered out or if an embedding cannot be expanded to a final result, the function must remove it from $ET$.

The details of the expandEmbedTrie function are given in Algorithm 2. Lines 3 to 7 set the candidate set for each $u \in dp_l.LF$ as the intersection of the neighbour set of $v_{piv}$, where $v_{piv} = f(dp_l.piv)$ and the neighbour set of each $f(u')$, where $(u', u'')$ is a cross-unit edge and $f(u')$ is in $Mt$. If any of the candidate sets is empty, it removes the corresponding trie node $N'$ from $ET$. Otherwise it picks the first leaf vertex $u$ and calls a recursive subroutine adjEnum (Line 11, 12). By expanding any $f$ of $P_{l-1}$, adjEnum finds the ECs of $P_l$ within the neighbourhood of $v_{piv}$. If $f$ cannot be expanded into an EC of $P_l$, we will remove $N$ from $ET$ (Line 13).

The details of the adjEnum function are presented in Algorithm 3. In each round of adjEnum, we try to match
Algorithm 2: EXPAND-EMBED-Trie

Input: an embedding \( f \) of \( P_{i-1} \), local machine \( M_i \), unit \( c_{dp} \), embedding trie \( ET \)

Output: expanded \( ET \) and an edge verification index \( I \)

1. \( v_{prev} \leftarrow f(dp_0.piv) \)
2. get \( N \) corresponding to \( f \)
3. for each \( u \in dp_i.LF \) do
   4. \( C(u) \leftarrow adj(v_{prev}) \)
   5. for each \( (u,u') \in E_{\text{root}} \) do
      6. if \( f(u') \) resides in \( M_i \) then
         7. \( C(u) \leftarrow adj(f(u')) \cap C(u) \)
   8. if \( C(u) = \emptyset \) then
      9. remove \( N \) from \( ET \)
   10. return
11. \( u \leftarrow \) first leaf vertex in \( dp_i.LF \)
12. if \( \text{adjEnum}(N,u) \) is false then
   13. remove \( N \) from \( ET \)

\( u' \) to a candidate vertex \( v \), where \( u' \) is a query vertex in \( dp_i.LF \). If \( v \) can pass the local verification (Line 4.5), we add \((u,v)\) to \( f \) and create a trie node \( N' \) for \( v \) (Line 6 to 9). If all leaf vertices have been verified, we add all undetermined edges of \( f \) to \( I(v) \) (Line 11, 12). Otherwise, we call a deeper \( \text{adjEnum} \). If the current \( f \) can be expanded to an EC, we chain \( N' \) to the corresponding node \( N' \) of \( ET \) (Line 17, 19).

Algorithm 3: \( \text{adjEnum} \)

Input: Trie node \( N \) representing embedding \( f \) of \( P_{i-1} \), leaf vertex \( u \) of \( dp_i \)

Output: true or false

1. \( F_{\text{current}} \leftarrow \) false
2. for each \( v \in C(u) \) do
   3. \( E \leftarrow \) true
   4. if \( v \) resides in \( M_i \) then
      5. \( E \leftarrow \text{isJoinable}(u,v) \)
   6. if \( E \) is true then
      7. add \((u,v)\) to \( f \)
      8. create a trie node \( N' \)
      9. \( N'.v \leftarrow v, N'.\text{parent}N \leftarrow N \)
   10. if \( |f| = |V_p| \) then
      11. for each undetermined edge \( e \) of \( f \) do
         12. add \( N' \) to \( I(e) \)
      13. \( F_{\text{current}} \leftarrow \) true
   14. else
      15. \( u' \leftarrow \) next vertex in \( dp_i.LF \)
      16. \( F_{\text{deeper}} \leftarrow \text{adjEnum}(N',u') \)
   17. if \( F_{\text{deeper}} \) is true then
      18. \( N.\text{childCount} \leftarrow \) add \( N' \) as a child node of \( N \) in \( ET \)
      19. remove \((u,v)\) from \( f \)
20. return \( F_{\text{current}} \)

7. MEMORY CONTROL STRATEGIES

This section focuses on the robustness of R-Meef. Since R-Meef still caches fetched foreign vertices and intermediate results in memory, memory consumption is still a critical issue when the data graph is large. We propose a grouping strategy to keep the peak memory usage under the memory capacity of the local machine.

Our idea is to divide the candidate vertices of the first query vertex \( dp_i.piv \) into disjoint groups and process each group independently. In this way, the overall cached data on each machine will be divided into several parts, where each part is no larger than the available memory \( \Phi \).

A naive way of grouping the candidate vertices is to divide them randomly. However, random grouping of the vertices may put vertices that are “dissimilar” to each other into the same group, potentially resulting in more network communication cost. Consider the data graph in Figure 5. Suppose the candidate vertex set is \( \{v_0,v_1,v_2,v_3\} \). If we divide it into two groups \( \{v_0,v_1\} \) and \( \{v_2,v_3\} \), then because \( v_0 \) and \( v_1 \) share most neighbours, there is a good chance for the ECs of \( dp_0 \) generated from \( v_0 \) and \( v_1 \) to share common verification edges, and share common foreign vertices that need to be fetched (e.g., if \( dp_0.piv \) is mapped to \( v_0 \) by ECs originated from \( v_0 \) and \( v_1 \), and \( v_0 \) is not on the local machine). However, if we partition the candidate set into \( \{v_0,v_2\} \) and \( \{v_1,v_3\} \), then there is little chance for such sharing.

Our goal is to find a way to partition the candidate vertices into groups so that the chance of edge verification sharing and foreign vertices sharing by the results in each group is maximized.

Let \( C \equiv C(dp_0.piv) \) be the candidate set of \( dp_0.piv \), and \( \Phi \) be the available memory. Our method is to generate the groups one by one as follows. First we pick a random vertex \( v \in C \) and let \( rg = \{v\} \) be the initial group. If the estimated memory requirement of the results originated from \( rg \), denoted \( \phi(rg) \), is less than \( \Phi \), we choose another candidate vertex in \( C - rg \) that has the greatest proximity to \( rg \) and add it to \( rg \); if \( \phi(rg) > \Phi \) we remove the last added vertex from \( rg \). This generates the first group. For the remaining candidate vertices we repeat the process, until all candidate vertices are divided into groups. Here an important concept is the proximity of a vertex \( v \) to a group of vertices, and we define it as the percentage of \( v \)’s neighbors that are also neighbors of some vertex in \( rg \), that is,

\[
\text{proximity}(v,rg) = \frac{|\text{adj}(v) \cap \bigcup_{v' \in rg} \text{adj}(v')|}{|\text{adj}(v)|} \quad (4)
\]

Intuitively the vertices put into the same group are within a region - each time we will choose a new vertex that has a distance of at most 2 from one of the vertices already in the group (unless there are no such vertices). Therefore we call the group a region group.

Estimating memory usage In our system, the main memory consumption comes from the intermediate results and the fetched foreign vertices. The space cost of other data structures is trivial.

Consider the set of intermediate results originated from the group \( rg \subseteq C \). Recall that all results originated from the same candidate vertex of \( dp_0.piv \) are stored in the same tree, while any results originated from different candidate vertices are stored in different trees. Therefore, if we know the space cost of the results originated from every candidate vertex, we can add them together to obtain the space cost of all results originated from \( rg \).

To estimate the space cost of the results originated from a single vertex, we use the average space cost of local embeddings of a candidate vertex \( v \in C_i(u_{start}) \) in embedding trie format, which can be obtained when we conduct SM-E.
Recall that for each \( v \) of \( C_1(\text{use}_\text{start}) \) in SM-E, we find the local embeddings originated from \( v \) following a backtracking approach. In each recursive step of the backtracking approach, we may record the number of candidate vertices that are matched to the corresponding query vertex. The sum of all steps will be the number of trie nodes if we group the those local embeddings into embedding trie. Based on the sum, we know the space cost of local embeddings originating from \( v \) in the format of embedding trie.

Next, we consider the space cost of the fetched foreign vertices in each round. Recall that when expanding the embeddings of \( P_{i-1} \) to ECs of \( P_i \), we only need to fetch vertex \( v \) if there exists \( f \in R_{G_i}(P_{i-1}) \) such that \( f(dp_.piv) = v \). In the worst case, for every candidate vertex \( v \) of \( dp_.piv \), there exists some \( f \in R_{G_i}(P_{i-1}) \) which maps \( dp_.piv \) to \( v \), and none of these candidate vertices of \( dp_.piv \) resides locally. Therefore the number of data vertices that need to be fetched equals to \( |C(dp_.piv)| \) in the worst case.

8. EXPERIMENTS

In this section, we present our experimental results. We focus on performance comparison with four state-of-the-art distributed subgraph enumeration approaches:

- **PSgL** [20], the algorithm using graph exploration originally based on Pregel.
- **TwinTwig** [12], the algorithm using joining approach originally based on MapReduce.
- **SEED** [13], an upgraded version of **TwinTwig** while supporting clique decomposition unit.
- **Crystal** [16], the algorithm relying on clique-index and compression and originally using MapReduce.

We will also compare the communication cost. In addition, we will test the impact of our execution plan and our compression strategy. For scalability, we will follow the approach of [13] and test the effect on query time by varying the number of machines in the distributed cluster, the number of data vertices, and the average degree of the data vertices.

**Environment** We conducted our experiments in a cluster platform where each machine is equipped with Intel CPU with 16 Cores and 16G memory. In our performance comparison we used 10 machines in the cluster. For scalability test on varying graph size and vertex degree we used a larger cluster of 20 machines since the time taken on larger denser graphs is much longer. The operating system of the clusters is Red Hat Enterprise Linux 6.5.

We implemented our approach in C++ with the help of Mpi2h [8] and Boost library [19]. We used Boost.Asio to achieve the asynchronous message listening and passing. We used TurboIso [9] as our SM-E processing algorithm. We implemented **PSgL**, **TwinTwig** and **SEED** using C++ with MPI library as well. For **Crystal**, we used the code provided by the authors.

**Dataset & Queries** We used five real datasets: DBLP, RoadNet, LiveJournal, UK2002 and Friendster, whose profiles are given in Table 1. DBLP is a small data graph which is to test whether the approaches can fully utilize the memory when there is enough space available. RoadNet is a sparse data graph, which is used to illustrate whether a subgraph enumeration solution has good filtering power to filter out false embeddings early. Two denser data graphs, LiveJournal and UK2002, are used to test the algorithms’ ability to handle denser graphs with larger numbers of embeddings. Friendster is a super large data graph (32.4GB), where many queries will fail to complete in a reasonable time for all algorithms. We only used it in the scalability test against the graph size and average degree.

| Dataset \((G)\) | \(|V|\) | \(|E|\) | Avg. degree | Diameter |
|----------------|------|------|------------|----------|
| RoadNet        | 1.9M | 2.7M | 2.81       | 849      |
| DBLP           | 0.3M | 1.0M | 6.62       | 21       |
| LiveJournal    | 4.8M | 42.9M| 15         | 17       |
| UK2002         | 18.3M| 288.1M| 32         | 24       |
| Friendster     | 0.6M | 1.8B | 55         | 92       |

On disk, our data graphs are stored in plain text format where each line represents an adjacency-list of a vertex. We used Metis [10] to partition the data graphs and each machine randomly picks up one partition and no duplication is allowed. The approach of **Crystal** relies on the clique-index of the data graph which should be pre-constructed and stored on disk. In Table 2, we present the disk space cost of the index files generated by the program of **Crystal**. The index file for Friendster is not included since it takes too long to compute. As can be seen, the index files are more than 10 times larger than the original data graph. The queries we used are shown in Figure 6. Experiments using additional queries can be found in the long version of this paper [18].

<table>
<thead>
<tr>
<th>Dataset ((G))</th>
<th>Data Graph File Size</th>
<th>Index File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoadNet</td>
<td>43.1MB</td>
<td>569MB</td>
</tr>
<tr>
<td>DBLP</td>
<td>13MB</td>
<td>210MB</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>501MB</td>
<td>6.5GB</td>
</tr>
<tr>
<td>UK2002</td>
<td>4.1GB</td>
<td>60GB</td>
</tr>
</tbody>
</table>

Figure 6: Query Set

8.1 Performance Comparison

We compared the performance of the five subgraph enumeration approaches by measuring the time elapsed (in seconds) and the volume of exchanged data during the processing of each query. The results of RoadNet, DBLP, LiveJournal and UK2002 are given in Figures 7, 8, 9 and 10, respectively. We mark the result as empty when the test fails due to memory crash. When any bar reaches the upper bound, it means the corresponding values is beyond the upper bound value shown in the chart.

**Exp-1: RoadNet** The results over the RoadNet dataset are given in Figure 7. As can be seen from the figure, RADS and PSgL are significantly faster than the other three methods (by more than 1 order of magnitude). RADS and PSgL are using graph exploration while the others are using join-based
methods. Therefore, both RADS and PSgL demonstrated efficient filtering power for this sparse graph. The join-based methods need to group the intermediate results based on keys so as to join them together, their performance was significantly dragged down when dealing with sparse graphs compared with RADS and PSgL.

It is worth noting that PSgL was shown to be slower than SEED and TwinTwig in [12],[13]. This may be because the datasets used in TwinTwig and SEED are much denser than RoadNet, hence a huge number of embeddings will be generated. The grouped intermediate results of TwinTwig and SEED significantly reduced the cost of network traffic. Another interesting observation is that although Crystal has heavy indexes, its performance is much worse than PSgL and RADS. The reason is that the number of cliques in RoadNet is relatively small considering the graph size. Moreover, there are no cliques with more than two vertices in queries q1, q3, q6, q7 and q8. In such cases, the clique index cannot help to improve the performance.

As shown in Figure 7(b), the communication cost is not large for any of the approaches (less than 5M for most queries). In particular, for RADS, the communication cost is almost 0, which is mainly because most data vertices can be processed by SM-E, as such little network communication is required.

Exp-2:DBLP The result over DBLP is shown in Figure 8. As aforementioned, DBLP is smaller but much denser than RoadNet. The number of intermediate results generated in DBLP are much larger than that in RoadNet, as implied by the data communication cost shown in Figure 8 (b). Since PSgL does not consider any compression or grouping of intermediate results, the communication cost of PSgL is much higher than that of the other approaches (more than 200M for queries after q4). Consequently, the time delay due to shuffling the intermediate results caused bad performance for PSgL. However, PSgL is still faster than SEED and TwinTwig. This may be because the time cost of grouping intermediate results of TwinTwig and SEED is high as well. It is worth noting that the communication cost of our RADS is quite small (less than 5M). The time efficiency of RADS is better than Crystal even for queries q2, q4 and q8 where the triangle crystal can be directly loaded from index without any computation.

Exp-3:LiveJournal As shown in Figure 9, for LiveJournal, SEED, TwinTwig and PSgL start becoming impractical for queries from q4 to q8. Due to the huge number of intermediate results generated, the communication cost increased significantly as well, especially for PSgL where communication cost was beyond control when the query vertices reach 6. The method of Crystal achieved good performance for queries q2, q4 and q8. This is mainly because Crystal simply retrieved the cached embeddings of the triangle to match the vertices (u0, u1, u2) of those 3 queries. However, when dealing with the queries with no good crystals (q6, q7 and q8), our method significantly outperformed Crystal. One important thing to note is that the other three methods (SEED, TwinTwig and PSgL) are sensitive to the end vertices, such as u5 in q5. Both time cost and communication cost increased significantly from q4 to q8. RADS processes those end vertices last by simply enumerating the combinations without caching any results related to them. The end vertices within Crystal will be bud vertices which only requires simple combinations. As indicated by query q5, where their processing time increased slightly from that of q4, RADS and Crystal are nicely tuned to handle end vertices.
Exp-4:UK2002 As shown in Figure 10, TwinTwig, SEED and Psgl failed the tests of queries after q2 due to memory failure caused by huge number of intermediate results. The communication cost of all other methods are significantly larger than RADS (more than 2 orders of magnitude), we omit the chart for communication cost here. Similar to that of LiveJournal, the processing time of Crystal is slightly better than that of RADS for q4 and q5 (which have triangles). This is because Crystal directly retrieves the embeddings of the triangles from the index. However, for q2 and queries without good crystals, our approach demonstrates better performance. Note that the index file of Crystal is more than 10 times larger than the original data graph, as shown in Table 2.

Another advantage of RADS over Crystal is our memory control strategies ensures it is more robust: we tried to set a memory upper bound of 8G and test query q6, Crystal starts crashing due to memory leaks, while RADS successfully finished the query.

8.2 Effectiveness of Compression
To show the effectiveness of our compression strategy, we conducted an experiment to compare the space cost of the simple embedding-list (EL) with that of our embedding trie (ET). We use the RoadNet and DBLP data sets for this test. The queries are as shown in Figure 6. We omit the test over the other two data sets because the uncompressed volume of the results are too large.

Table 3: Compression on RoadNet(Mb)

<table>
<thead>
<tr>
<th>Query</th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>q4</th>
<th>q5</th>
<th>q6</th>
<th>q7</th>
<th>q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL</td>
<td>264</td>
<td>13</td>
<td>65</td>
<td>81</td>
<td>146</td>
<td>183</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ET</td>
<td>163</td>
<td>5</td>
<td>33</td>
<td>40</td>
<td>63</td>
<td>73</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4: Compression on DBLP (Gb)

<table>
<thead>
<tr>
<th>Query</th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>q4</th>
<th>q5</th>
<th>q6</th>
<th>q7</th>
<th>q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL</td>
<td>0.3</td>
<td>0.2</td>
<td>4.5</td>
<td>3.2</td>
<td>17.6</td>
<td>7.6</td>
<td>5.3</td>
<td>4</td>
</tr>
<tr>
<td>ET</td>
<td>0.05M</td>
<td>0.06</td>
<td>1.1</td>
<td>0.7</td>
<td>3.8</td>
<td>1.5</td>
<td>0.9</td>
<td>0.8</td>
</tr>
</tbody>
</table>

The results are as shown in Table 3 and Table 4, respectively. For RoadNet the intermediate results generated by Queries 7 and 8 are negligible, therefore they are not listed. The results for both datasets demonstrate a good compression ratio. It is worth noting that the compression ratios of all queries over RoadNet are smaller than that over DBLP. This is because the embeddings of RoadNet are very diverse and they do not share a lot of common vertices.

8.3 Effectiveness of Query Execution Plan
To validate the effectiveness of our strategy for choosing query execution plan, we compared the processing time of RADS with two other baseline plans which are generated by replacing the execution plan of RADS with the execution plans RanS and RanM, respectively. RanS represents a plan consisting of random star decomposition units (no limit on the size of the star) and RanM represents plan with minimum number of rounds without considering the strategies in Sections 5.2 and 5.3. In order to cover more random query plans, we run each test 5 times and report the average. The queries are as shown in Figure 6. For queries q1 to q8, the query plans generated in the above three implementations are almost the same. Therefore, we omit the data for those three queries.

8.4 Scalability Test
Varying graph size and average degree Following the method of [13], we generated 5 subgraphs of Friendster by extracting 20, 40, 60, 80 and 100% of the vertices. By fixing 100% of the vertices, we randomly sample 20, 40, 60, 80 and 100% of the edges to get another 5 subgraphs whose average degrees range from 11 to 55. We compared the results of Psgl, SEED, TwinTwig and RADS by testing two queries q2 and q4, the results are as shown in Figure 12. The index of Crystal is too large on Friendster, therefore Crystal is not included in this experiment.

As we can see in Figure 12, the time cost of all four methods significantly increases when we increase the graph size and the average degree. However, RADS is consistently better than the other 3 methods. For q4, the three previous methods all failed to finish the processing on the complete Friendster graph due to memory crash.

Varying the number of nodes in the cluster We compared the five approaches by varying the number of nodes in the cluster from 5 to 10 and 15. Instead of reporting the...
processing time, here we report the ratio between the total processing time of all queries using 5 nodes and that of the other two cases, which we call scalability ratio. Because TwinTwig, SEED and PSgL failed some queries for LiveJournal and UK2002, we omit them in the two datasets. The result is shown in Figure 13. The most important thing to observe is that our approach demonstrates linear speed-up when the number of nodes is increased for Roadnet and DBLP. For LiveJournal and UK2002 the difference between Crystal and RADS is not much while RADS is better for both.

![Figure 12: Results of varying graph size & avg. degree](image)

(c) q2 vary avg. degree  
(d) q4 vary avg. degree

Figure 12: Results of varying graph size & avg. degree

9. RELATED WORK

The works most closely related to ours are TwinTwig [12], SEED [13] and PSgL [20]. Both [12] and [13] use multi-round two-way joins. [12] uses the same data partitioning as in our work, and it decomposes the query graph $P$ into a set of small trees $d p_0, \ldots, d p_k$ such that the union of these trees is equal to $P$. The union of their embeddings on all machines is the set of all embeddings of $d p_i$ over $G$. [13] is similar to [12], except that it allows decomposition units to be cliques as well as trees, and it uses bushy join rather than left-deep joins\(^2\). Both TwinTwig and SEED may generate huge intermediate results, and shuffling, re-distribution and synchronization cost a lost of time. Instead of using joins, we follow expand-verify-filter on each machine, as such we generate less intermediate results, and we do not need to re-distribute them to different machines.

PSgL [20] is based on Pregel [15]. It maps the query vertices one at a time following breath-first traversal. However, there are important differences between PSgL and our system (RADS). (1) In each step of expansion, PSgL needs to shuffle and send the partial matches (intermediate results) to other machines, while RADS does not need to do so. (2) PSgL stores each (partial) match as a node of a static result tree, while RADS stores the results in a dynamic and compact data structure. (3) There is no memory control in PSgL.

Also closely related to our work are [5] and [4], which introduce systems for parallelizing serial graph algorithms, including (but not limited to) subgraph isomorphism search. The work [1] treats the query pattern as a conjunctive query, where each predicate represents an edge, and computes the results as a multi-way join in a single round of map and reduce. The problem with [5][4][1] is that a large part of data graph have to be duplicated over all the machines which limits their practicality when the query pattern is complex.

Qiao et al [16] represent the set $I_P$ of all embeddings of pattern $P$ in a compressed form, code($I_P$), based on a minimum vertex cover of $P$. It decomposes the query graph $P$ into a core $core(P)$ and a set of so-called crystals $\{p_1, \ldots, p_k\}$, such that code($I_P$) can be obtained by joining the compressed results of $core(P)$ and $\{p_1, \ldots, p_k\}$. The compressed results of $core(P)$ and the crystals can be obtained from the compressed results of components of $P$. It needs to build an index of all cliques of the data graph, as shown in Table 2. Although no shuffling of intermediate results is required, the indexes of [16] can be many times larger than the data graph.

BigJoin, one of the algorithms proposed in [2], treats a subgraph query as a join of $|E_P|$ binary relations where each relation represents an edge in $P$. Similar to RADS and PSgL, it generates results by expanding partial results a vertex at a time, assuming a fixed order of the query vertices. Different from our work, it still needs to shuffle and exchange intermediate results, and therefore synchronization before that.

10. CONCLUSION

We presented a novel approach for distributed subgraph enumeration. By processing the data vertices far away from the border using the single-machine algorithms, we isolated a large part of vertices which does not have to be involved in the distributed process. By passing edge verification requests/results and adjacency lists of foreign vertices, RADS significantly reduced the network communication cost. We also proposed a compact format to store the generated intermediate results. Our query execution plan and memory control strategies helped to improve the efficiency and robustness. Our experiments verified the superiority of our approach compared with several other approaches.

Acknowledgement Xuguang Ren and Junhu Wang were supported by the Australian Research Council (ARC) discovery grant DP130103051; Wook-Shin Han was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.NRF-2017R1A2B2S007116) and Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1401-53. Jeffrey Xu Yu was supported by the Research Grants Council of Hong Kong, China, No. 14203618 and No. 14221716, and the Australian Research Council discovery grant DP130103051.
11. REFERENCES


