Bonding Vertex Sets Over Distributed Graph: A Betweenness Aware Approach

Xiaofei Zhang[†], Hong Cheng[‡], Lei Chen[†]
[†]Department of Computer Science & Engineering, HKUST
[‡]Department of Systems Engineering & Engineering Management, CUHK
{zxfei,leichen}@cse.ust.hk, hcheng@se.cuhk.edu.hk

ABSTRACT

Given two sets of vertices in a graph, it is often of a great interest to find out how these vertices are connected, especially to identify the vertices of high prominence defined on the topological structure. In this work, we formally define a ***V**ertex **S**et **B**onding* query (shorted as VSB), which returns a minimum set of vertices with the maximum importance *w.r.t* total betweenness and shortest path reachability in connecting two sets of input vertices. We find that such a kind of query is representative and could be widely applied in many real world scenarios, e.g., logistic planning, social community bonding and etc. Challenges are that many of such applications are constructed on graphs that are too large to fit in single server, and the VSB query evaluation turns to be NP-hard. To cope with the scalability issue and return the near optimal result in almost real time, we propose a generic solution framework on a shared nothing distributed environment. With the development of two novel techniques, guided graph exploration and betweenness ranking on exploration, we are able to efficiently evaluate queries for error bounded results with bounded space cost. We demonstrate the effectiveness of our solution with extensive experiments over both real and synthetic large graphs on the Google's Cloud platform. Comparing to the exploration only baseline method, our method achieves several times of speedup.

1. INTRODUCTION

In this work, we study a novel graph query, namely the ***V**ertex **S**et **B**onding* query (VSB query for short), which extracts the most prominent vertices, called bonding agents, in connecting two sets of input vertices. The prominence of a vertex is defined on its contribution to the shortest path connectivity between input vertex sets. Intuitively, given two input sets of vertices X and Y, the desired bonding agents are the minimum set of vertices to remove in order to enlarge every pair of shortest path distance between X and Y. We find this query could be widely applied in various real world applications. We elaborate with two following examples.

Example 1.1 (Network Flow Monitoring). *To identify potential bottlenecks for large volume data transfer initiated randomly on a P2P network is critical to improve the overall network performance [11]. For example, Figure 1 shows that the data transfer plan among two vertex sets X and Y, where edges are the data transfer paths, and v_1,...,v_9 are clients contributing to the data forwarding. For illustration purpose, we assume that all clients have the same capacity and all links are identical. The VSB query will return the minimum set of clients that dominates the overall data transfer performance. More importantly, the returned bonding agents will be the minimum set of vertices to monitor in order to obtain a complete statistics on the transfer between X and Y.*

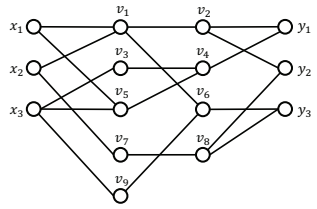


Figure 1: Vertex set bonding example

Example 1.2 (Community Bonding). *A common interest of social network study is to find the "bonding" of communities [8], which is vital in understanding information propagation and hidden correlations [23]. Two people are usually considered to be tightly connected by the ones residing on the shortest path between them. Intuitively, an ideal bonding agent would reside on as many cross group pairwise shortest paths as possible, and meanwhile connect as large portion of two groups as possible. Such agents could best serve the message passing between two groups.*

The above two examples imply an essential need to efficiently discover the most valuable bonding agents between two sets of vertices, which could be given at *ad hoc*. As a matter of fact, such a demand is common in many real world applications, like ad hoc logistics planning [38], or extracting and querying correlations in knowledge graphs on input facts [41], and etc.

There is a rich literature on structure oriented graph queries, such as shortest path, reachability, subgraph matching, influential maximization [28] and etc. However, we find that none of these existing graph queries could be directly applied to the scenarios we discussed above. We briefly review the most relevant graph queries and elaborate the novelty of the VSB query. As VSB is defined on the shortest path semantic, it makes betweenness centrality a reasonable metric to employ. Top-k betweenness computing or k-betweenness are classic queries employed to find important vertices in a network. However, due to the local dominance property of the betweenness metric, such queries cannot serve the vertex sets bonding properly. For example, in Figure 1, v_2 has a much higher betweenness than v_4. However, v_2 is completely dominated by v_1, meaning v_1 resides on every shortest path between X and Y that

Proceedings of the VLDB Endowment, Vol. 8, No. 12

passes v_2. On the contrary, without v_4, the shortest path distance between x_3 and y_1 would increase. It makes v_4 an indispensable bonding agent.

Influential maximization is a cool idea to identify a set of important vertices (referred as *seeds*) to maximize the expected number of vertices to be influenced following certain cascading model. However, a practical cascading model is the key to guide the seed selection, which cannot be directly transferred to our applications scenarios. Moreover, a widely adopted heuristic is to select seeds that are far away from other vertices as long as the cascading function allows. Therefore, such a query cannot return the crucial set of vertices in connecting two vertex sets topologically.

Minimum cut [25] finds the minimum set of edges to remove to turn a graph into two disjoint subgraphs. However, it does not offer any insight on how other vertices contributes to the connection between X and Y. Moreover, minimum cut does not help to find the bonding agents. Because a bonding agent could reside on every pairwise shortest path between vertex sets X and Y, and not be incident to any edge from the minimum cut.

We believe that the VSB query is a novel query to explore, given none of existing well studied graph queries could properly serve the application scenarios discussed above. The VSB query ranks a vertex's prominence in bonding two vertex sets with two factors taken into consideration: betweenness and the shortest path connectivity. Although the metric for a vertex's prominence varies depending on application scenarios, such a vertex bonding query pattern is generic. In this work, we evaluate VSB queries based on betweenness, as it is a popular centrality measurement in practice. Although there are other centrality metrics, like closeness, eigenvector, percolation and etc., we consider other semantics as a future extension of this work.

Moreover, we find that the VSB query inherently asks for a much more efficient evaluation strategy which has never been explored in the existing literature. To evaluate VSB queries efficiently, considering the fact that graph G could be stored in a shared nothing distributed setting, simply a combination of existing techniques may not be the optimal choice. To elaborate, a straightforward solution to answer a VSB query is to first extract the subgraph $\mathcal{G}$ that contains all pairwise shortest paths between two input vertex sets X and Y, then assess the bonding vertices based on a ranking of betweenness and shortest path connectivity. However, identifying $\mathcal{G}$ on arbitrary input vertex sets is computational costly based on any BSP model developed for generic graph processing, as many redundant vertices would be accessed during the computation. Recent research shows that such an exploration-based path computation cannot be bounded in terms of vertex accessing [17]. As the best known betweenness computation algorithm [6] requires $\mathcal{G}$ to be available in the first place, which is highly impractical. Moreover, computing the all vertices' exact or approximated betweenness for further selection would introduce redundant computational cost. Because only a partial ranking of some *important* vertices would be sufficient to answer the query. Therefore, Not only the state-of-art graph processing framework like Pregel [34] needs be employed to explore the maximum parallelism of query evaluation, we also need delicate and lightweight index structures to support efficient filtering and scheduling. It is nontrivial to combine the distributed index and generic graph processing framework to perform selective queries over graph data.

To address the challenge, we propose two novel building blocks for the efficient VSB query evaluation: guided graph exploration and betweenness ranking on-exploration. The essential idea behind is to consider every vertex as a high dimensional vector derived from its distance to a set of landmarks. Thus, we could guide the graph exploration to reduce redundant vertex access and have the vertex-centric computation run simultaneously to achieve the most parallelism. As we only ask for the minimum set of vertices of the highest accumulative betweenness as the bonding vertices, instead of computing the exact betweenness value, we simply rank the betweenness of vertices during graph exploration to save the computation cost. To summarize, we highlight our contribution of this work as follows:

- We propose and formalize a novel vertex set bonding query which could be widely applied in real practices. We show that VSB problem is NP-hard and develop approximation algorithms for error bounded query evaluation (Section 2);
- We develop a vertex filtering scheme to effectively support guided graph exploration, such that the cost on redundant vertex accessing could be significantly saved (Section 3);
- We develop an effective betweenness ranking algorithm simply based on graph exploration, such that VSB query can be evaluated as quickly as possible by saving the cost on exact betweenness computation (Section 3);
- We develop a generic VSB evaluation framework and validate our solution with extensive experiments in a real Cloud environment (Section 4 and 5).

In addition, we discuss the most recent related works in Section 6 and make the conclusion in Section 7.

2. PROBLEM DEFINITION

In this section, we first introduce the preliminaries of betweenness centrality and define the terminology adopted in this work. We shall formally define the VSB query and discuss the computational complexity of the VSB problem. Then we give an overview of our solution and present the technical road map.

Being an important centrality metric, a vertex v's betweenness value is defined as the fraction of all pairwise shortest paths that passes v. Let $C_B(v)$ denote the betweenness of v, $\sigma_{st}(v)$ denote the number of shortest paths between two vertices s and t that passes v, σ_{st} denote the total number of shortest paths between s and t, then $C_B(v)$ is defined as follows:

$$C_B(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

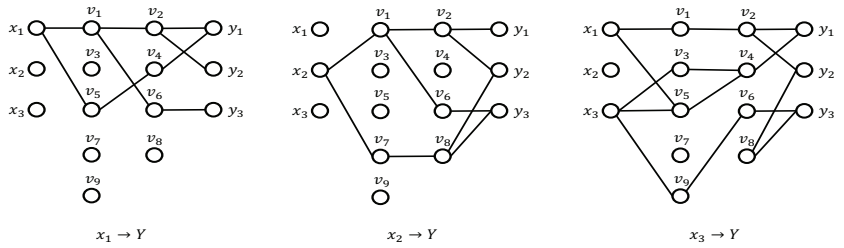
The best known sequential algorithm for computing betweenness was proposed by Brandes [6], and it has been the basis for many parallelization approaches [4][13][16][32]. Brandes's algorithm firstly defines the *dependency* of a source vertex s on a vertex v as: $\delta_s(v)$=$\sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. Thus, $C_B(v)$=$\sum_{s \neq v \in V} \delta_s(v)$. The key insight is that $\delta_s(v)$ satisfies the recurrence given in Eq.2, where $pred(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w.

$$\delta_s(v) = \sum_{w: v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \tag{2}$$

Given the recurrence derived in Eq.2, Brandes's algorithm is able to compute every vertex's betweenness with $O(m + n)$ space complexity and $O(mn)$ computational complexity for unweighted graph, where m and n are the number of edges and vertices in G.

We define two types of dominance relation for comprehensive illustration of the VSB query evaluation. Given an undirected graph G=$\langle V, E \rangle$, let p_{st} denote the shortest path(s) between vertex s and t, and $|p_{st}|$ be the length of p_{st}. We define the vertex-to-path dominance as follows:

DEFINITION 1 (V-P DOMINANCE). *A vertex v dominates a path p_{st}, denoted as $v \vdash p_{st}$, **iff** $|p_{st}|$ increases by removing v from the graph. $\{v_\vdash\}^P$ denotes the set of shortest paths dominated by v.*



(a)

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
x_1y_1	0.5	0.5	0	0.5	0.5	0	0	0	0
x_1y_2	1	1	0	0	0	0	0	0	0
x_1y_3	1	0	0	0	0	1	0	0	0
x_2y_1	1	1	0	0	0	0	0	0	0
x_2y_2	0.5	0.5	0	0	0	0	0.5	0.5	0
x_2y_3	0.5	0	0	0	0	0.5	0.5	0.5	0
x_3y_1	0	0	0.5	1	0.5	0	0	0	0
x_3y_2	0.25	0.75	0.25	0.5	0.25	0.25	0	0.25	0.25
x_3y_3	0	0	0	0	0	1	0	0	1
$C_B(v_i)$	4.75	3.75	0.75	2	1.25	2.75	1	1.25	1.25

(b)

Figure 2: Decomposed $X \to Y$ paths of Figure 1 for betweenness and shortest path connectivity analysis.

If there exists multiple shortest paths between s and t, then p_{st} may not be dominated by any single vertex. Instead, p_{st} is dominated by a vertex set U, denoted as $U \vdash p_{st}$, where $|p_{st}|$ increases if U is removed from the graph. For example, in Figure 1, as there are two parallel shortest paths between x_1 and y_1, $p_{x_1y_1}$ is not dominated by a single vertex. On the other hand, $p_{x_1y_1}$ is dominated by a set of vertices, e.g. $\{v_1, v_5\}$. Following the same assumption context, we define the dominance of vertex to vertex as follows:

DEFINITION 2 (U-V DOMINANCE). *A vertex u dominates another vertex v, denoted as $u \vdash v$, **iff** $\{v_\vdash\}^P \subseteq \{u_\vdash\}^P$. The set of vertices dominated by vertex u is denoted as $\{u_\vdash\}^V$.*

Given two sets of vertices X and Y, let P_{XY}=$\{p_{xy} | x \in X, y \in Y\}$ denote the set of all pairwise shortest paths between the elements of X and Y, we further define closed dominance and minimum closed dominance as follows:

DEFINITION 3 (CLOSED DOMINANCE). *A vertex set U is said to be a closed dominance of P_{XY}, **iff** $P_{XY} \subset \bigcup_{u \in U} \{u_\vdash\}^P$.*

DEFINITION 4 (MINIMUM CLOSED DOMINANCE). *A vertex set U is a minimum closed dominance of P_{XY} **iff** U is no longer a closed dominance of P_{XY} after removing any element in U.*

To elaborate, consider the example given in Figure 1. We decompose the pairwise shortest path connection between each $x_i \in X$ and Y, as shown in Figure 2(a). Regarding each $v_i \in V$, we show its betweenness credit earned on every pairwise shortest path between X and Y in Figure 2(b), computed with Eq.3,

$$C_B(v_i|X,Y) = \sum_{x \in X, y \in Y} \frac{\sigma_{xy}(v_i)}{\sigma_{xy}} \tag{3}$$

Obviously $\{v_1, ..., v_9\}$ forms a closed dominance of P_{XY}. But it is not a minimum one. Both $\{v_1, v_4, v_6, v_7\}$ and $\{v_2, v_3, v_5, v_6, v_7\}$ form a minimum closed dominance of P_{XY}. However, it is easy to examine that v_1 dominates v_2. Because v_1 is v_2's precedent on every shortest path from X to Y that passes v_2. Similarly, v_4 dominates v_3 and v_5. Therefore, we further define the optimal minimum closed dominance as follows:

DEFINITION 5 (OPTIMAL MINIMUM CLOSED DOMINANCE). *A vertex set U is an optimal minimum closed dominance of P_{XY} **iff** U is a minimum closed dominance of P_{XY}, $\nexists U'$ which is another minimum closed dominance of P_{XY} that $\exists u' \in U'$, $\exists u \in U$ having $u'_\vdash u$.*

Based on the terminology introduced above, now we formally define the vertex set bonding query, *a.k.a* the VSB query.

Problem Definition 2.1 (VSB Query). *Given an undirected graph G=$\langle V, E \rangle$ and two input sets of vertices X and Y, a vertex set bonding query Q=$\langle G, X, Y, R \rangle$ asks for a set of vertices $R \subset V$-$\{X, Y\}$, such that 1) R forms an optimal minimum closed dominance of P_{XY}; 2) $AB(R)$=$\sum_{v \in R} C_B(v|X,Y)$ is maximized.*

To elaborate, consider the example shown in Figure 2(a). We can find two vertex sets which are optimal minimum closed dominance of P_{XY}, $\{v_1, v_4, v_6, v_7\}$ and $\{v_1, v_4, v_6, v_8\}$. Apparently, the later set contributes more in the betweenness centrality and therefore should be returned as the answer. It is worth pointing out that although here we focus on the VSB query defined on the shortest path based dominance, any other dominance semantic could be employed to meet different application scenarios. For example, in heterogeneous information networks, the dominance could be defined upon the cover of keywords or meta-path patterns. Although exploring the evaluation strategy for different dominance functions is beyond the scope of this work, we argue that the VSB query is generic and could be employed for various real world applications.

From the problem definition, one can easily tell that the VSB problem is a variation of the weighted set cover problem, which has been proven to be NP-hard. However, one upfront problem is that X and Y are given at *ad hoc*, no vertex-path dominance relation is determined until the run time. In other words, for any vertex $v \in V$, $\{v_\vdash\}^V$ is unpredictable until X and Y are determined and $\mathcal{G}$ is extracted. More importantly, the essential difficulty of the VSB problem is that there could be exponential number of vertex sets for the minimum closed dominance verification, which makes none of the existing solutions for weighted set cover problem applicable. Therefore, we must develop novel techniques to reduce the possible vertex-path dominance combination in order to answer the query with precision guarantees as quickly as possible.

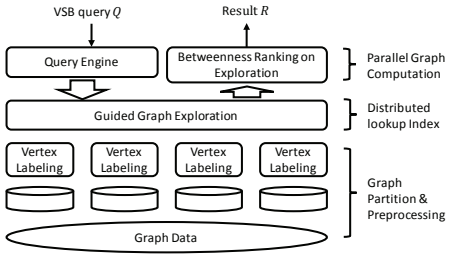


Figure 3: Solution Framework of the VSB query evaluation

We show our solution framework in Figure 3. Let graph data be stored on a shared nothing distributed environment. We label all vertices according to their distances to selected landmarks. Then the guided graph exploration building block would effectively filter unnecessary vertices when a VSB query is submitted to the query engine. Later, we shall perform the betweenness ranking computation on exploring only the valid vertices. Note that the core technique for guided graph exploration lies in a light weighted distributed lookup index. While the betweenness ranking function is designed to take the advantage of parallel graph processing (basically the vertex centric model). In the following sections, we shall first explain how we label all vertices and the methodology of guided graph exploration. Then we show our distributed lookup index design that effectively filters unnecessary vertices upon a coming query, and a parallel vertex centric computation model for betweenness ranking on exploration. After that, we elaborate our generic VSB query processing strategy.

3. EVALUATION BUILDING BLOCKS

In this section, we first introduce the underlying data model we adopt for query evaluation, as well as the preprocessing steps to prepare the data. Then we elaborate two essential building blocks to our generic solution, guided graph exploration and betweenness ranking on exploration.

3.1 Data Model & Preprocessing

Following the vertex centric computing model proposed by Google [34], we model each vertex as an independent functioning unit. Given an undirected graph G=$\langle V, E\rangle$, we define each vertex v as follows:

$$v = \begin{cases} l(v) & \text{vertex label} \\ H & \text{a set of independent hash func.} \\ B(v) & \text{blocks of adjacent vertices} \end{cases}$$

$l(v)$ is derived from v's distance to all the landmarks, which we shall elaborate soon. Function set H is the same for all vertices, which shall be employed for constructing a bloom filter to serve the query evaluation. Note that all of v's one-hop neighbors are hash partitioned into a set of data blocks, denoted as $B(v)$. The size of a data block is of one cache line read. Instead of using adjacent list, we adopt a variant of the CRS (Compressed Row Storage) format. It is a compromise of the improved cache read locality and the ability to adapt dynamic graph update.

3.1.1 Landmark Selection

Selecting landmarks or reference points to facilitate the shortest path distance computation has been adopted in many works [43] [39][42]. Existing landmark selection criteria are quite biased according to different graph structures and applications. In our solution, we select landmarks not only based on the consideration of graph partition and pairwise shortest distance estimation, but an *evenly coverage* property is desired. To elaborate, we find that given two vertices s and t, the landmark best serves $|p_{st}|$ computation is the one closest to p_{st}. Therefore, we define a set of landmarks of evenly coverage as follows:

DEFINITION 6 (δ-EVENLY COVERAGE). *Given a graph G =$\langle V, E\rangle$, a set of landmarks, O=$\{o_1, o_2, ..., o_d\}$, is said to be an evenly coverage of G, **iff** $\forall v \in V$, $\exists o_i \in O$ such that $|p_{vo_i}| \leq \delta$, where δ is a customizable parameter.*

According to the definition, an interesting question is how to decide an evenly coverage O of a given graph G. Intuitively, if δ is small, the cardinality of O, denoted with parameter d would be large. As a matter of fact, it is easy to derive that in an extreme case, d needs to be at least as large as $\frac{n-1}{2\delta}$. On the other hand, at most 3 landmarks are sufficient if the diameter of G is smaller or equal to 2δ. In practice, we would like to select the minimal number of landmarks that satisfy a δ-evenly coverage of G in order to save index space and computation costs. Algorithm 1 gives a deterministic solution of finding the minimal d, which also helps decide the selection of landmark vertices.

In the first line of Algorithm 1, G_{diam} denotes the diameter of G. We consider G_{diam} as a given input as it can be easily computed following the super step based message passing model. Apparently, the above algorithm is to recursively partition G into a set of small graphs with diameter smaller than 2δ, and report the center vertices of these small graphs as landmarks. Let the level of recursions is h, then the total number of landmarks is $d = 2^h$. The computation cost of Algorithm 1 is $O(h|G|)$, because on each level of recursion the entire graph is traversed. We can save the computation cost using a random algorithm given in Algorithm 2. It worths pointing out that Algorithm 2 does not need G_{diam} to be pre-computed. On the other hand, as shown on line 3 of the algorithm, we randomly select a path (simply using graph exploration) of length 2δ at each iteration, and filter out all vertices that could be evenly covered in δ-hops from the middle vertex of this selected path, until all vertices from G are covered.

LEMMA 1. *Algorithm 2 runs at the complexity of $O(|G|)$ and returns an evenly coverage of G with at most $3 \times 2^{h-1}$ landmarks.*

PROOF. Consider an uncovered subgraph g with a diameter falls in $(2\delta, 4\delta]$, it takes two landmarks to evenly cover g according to Algorithm 1. However, according to Algorithm 2, a subgraph $g' \in g$ could be selected, leaving the remaining part to be sufficiently covered by at most 2 landmarks. Therefore, it takes three landmarks to cover any two adjacent small graphs after partition in Algorithm 1. Therefore, Algorithm 2 reports at most $\frac{3}{2} \times 2^h$=$3 \times 2^{h-1}$ landmarks. □

Algorithm 1: δ-evenly coverage landmarks computation

```
   Data: G=⟨V, E⟩,δ,G_diam
   Result: O={o_1, o_2, ..., o_d}
   Procedure LandMark()
1      while G_diam > 2δ do
2          LandMark(HalfSplit(G));LandMark(G-HalfSplit(G));
3      o ← the middle vertex of G's diameter path;
4      return o;
   Procedure HalfSplit()
5      e(s, t) ← the middle edge of G's diameter path;
6      G=G-e(s, t) ;
7      s.color ← c_1; t.color ← c_2 ;
8      Mark all vertices active;
9      while ∃v ∈ V is active do
10         if v receives a color message c_i then
11             v.color ← c_i;
12         if v is active and has a color then
13             v broadcasts its color to all neighbors;
14             v ← inactive;
15     return the graph colored with c_1;
```

Algorithm 2: Fast δ-evenly coverage landmarks computation

```
   Data: G=⟨V, E⟩,δ
   Result: O={o_1, o_2, ..., o_d}
1  O ← ∅;
2  while G ≠ ∅ do
3      p ← randomly select a path of length 2δ from G;
4      g ← the graph that can be reach from o within δ-hops;
       /* o is the middle point of p                    */
5      G ← G − g ;
6      O ← O ⋃ {o};
7  return O;
```

Note that for simplicity we do not show the case that there does not exist a path of length 2δ in G any more. As it is straightforward to introduce one landmark for each remaining connected component in G. An interesting opening problem is the selection of δ. We study how different δ affects the data preprocessing and query evaluation efficiency and report our findings in the experiment section.

3.1.2 Preprocessing

There are two major preprocessing tasks: one is to partition the graph to a shared nothing distributed environment; the other one is to assign each vertex a label for query evaluation. Note that our solution does not depend on a particular graph partition format, therefore, any general graph partition technique can be applied. In our solution, we follow a typical graph partition strategy that promises better data locality and workload balance. We first randomly partition G to all computing nodes. After the selection of O (the set of landmarks), G is conceptually composed of d small graphs $\{g_1, ..., g_d\}$, where g_i is the small graph covered by landmark o_i. Thus, we can obtain an abstraction of G, namely G_A=$\langle O, W\rangle$, where each vertex is the landmark o_i of weight $|g_i|$,

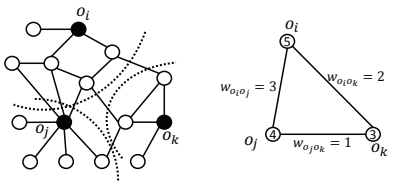


Figure 4: Computing an abstraction of G based on the landmarks

and an edge (o_i, o_j) of weight $w_{o_io_j}$ denoting g_i and g_j are connected with accumulatively $w_{o_io_j}$ edges in G. A simple example shown in Figure 4 shows how G_A is constructed. By applying a balanced minimum cut partition of G_A with METIS [27] (a well proven graph partition toolkit), we can obtain a partition of G such that the number of vertices stored on each computing node is balanced and the communication cost across physical machines could be greatly saved. Moreover, with the set of landmarks O determined, we are able to decide each vertex's label according to its distance to all landmarks. Therefore, vertex v's label $l(v)$ is a d-dimensional vector, where $l(v)_i$ denotes v's distance to landmark o_i. Starting from the d landmarks, with one time graph exploration, every $l(v)$ can be determined. Thus, our data preprocessing cost is at the computational and space complexity of $O(|G|)$.

It is worth pointing out that associating each vertex with a d dimensional vector ideally trades off space cost to empower filtering on graph exploration. However, in real world scenarios d could be very large if δ is set to a small value, which could impose infeasible space overhead for graph storage. As a matter of fact, given a vertex u and a landmark o, their shortest path distance can be denoted as $|p_{uo'}|$+$|p_{o'o}|$-σ, where $|p_{uo'}|$ is the distance from u to its nearest landmark o'. As dist(o,o') can be pre-computed during preprocessing, then only the adjusted value σ needs to be stored. Note that the employed graph partition strategy potentially promises a locality-based landmark clustering. It results in the value locality of σ in u's label, where a simple value based compression technique can be applied to reduce the total space cost significantly.

3.2 Guided Graph Exploration

Graph exploration is an essential tool to path query evaluations. Performance of the simple vertex centric model is highly dominated by high degree vertices and the total number of super steps to run. Intuitively, to explore the shortest path from s to t, at least $|p_{st}|$ super steps are necessary. Starting from s, a naive graph exploration method like BFS would access all vertices within a distance of $|p_{st}|$ to s. Thus, we would like to investigate a guided graph exploration approach to significantly reduce the redundant vertex access.

Our design is simple and straightforward. Let v_k resides on the shortest path between s and t. Assume $|p_{st}|$ is given, v_k is a k-hop vertex from s, then according to the cosine law, the distance from v_k to a landmark o_i is solely determined on $l(s)_i$, $l(t)_i$ and k. And such a condition must be hold between every landmark and v_k, which could greatly help filtering out possible candidates for future examination. Plus, as v_k's label has been computed during the preprocessing phase, it is easy to verify if v_k exists. If negative, it only shows that the assumption on $|p_{st}|$ is wrong. Given vertex s and t, we can simply bound the $|p_{st}|$ using the triangle inequality. It is easy to verify that $|p_{st}| \in [Max(|l(s)_i - l(t)_i|), Min(|l(s)_i + l(t)_i|)]$, where $1 \leq i \leq d$. For comprehensive presentation, the notation $|p_{st}| \in [LB(|p_{st}|), UB(|p_{st}|)]$ is employed for the rest of this paper. An observation on the determination of $|p_{st}|$ is that, an assumption of $|p_{st}|$ is correct **iff** $\forall k \in [1, |p_{st}|)\ \exists v_k$, such that $\forall o_i \in O$, $l(v_k)_i$ is valid according to the cosine law. Based on this observation, given a range of possible $|p_{st}|$, a brute-force solution is to check all possible values of $|p_{st}|$ in an ascending order and report the first valid result as the correct $|p_{st}|$, as described in Algorithm 3. Note that the loop given on line 3 indicates an iterative exploration process. In each iteration, we identify a set of valid vertices to be explored according Observation 1. The benefit of Algorithm 3 is that we can get exact p_{st} as a side product. However, the worst case happens when some landmark resides on p_{st}, meaning we get correct $|p_{st}|$ only after checking all the possible values.

Algorithm 3: A brute-force validation of $|p_{st}|$

```
Data: |p_st| ∈ [r_MIN, r_MAX]
Result: |p_st|
1 for i ∈ [r_MIN, r_MAX] do
2     |p_st| = i;
3     for k ∈ [1, i] do
4         Let S_k be the set of vertices that are k-hop neighbors of s;
5         if ∄v_k ∈ S_k is valid then
6             continue;
7 return |p_st|;
```

Apparently, Algorithm 3 is efficient only for the scenarios where $|p_{st}|$ is very close to its lower bound. In the worst case, it takes $O(|p_{st}|^2)$ iterations to find p_{st}. Therefore, we would like to propose another algorithm which has strict performance guarantees on all possible conditions. The intuition is that by starting from a set of vertices possibly residing on p_{st}, which must be a superset of p_{st}, we perform a guided exploration that iteratively prunes all candidates that do not belong to p_{st}.

LEMMA 2. *Given vertices s and t, a vertex v possibly resides on p_{st} if $Max\{|l(v)_i - l(s)_i| + |l(v)_i - l(t)_i|\} \leq UB(|p_{st}|)$, where $1 \leq i \leq d$.*

PROOF. Let vertices u and v be directly adjacent to each other. Then $Max\{|l(v)_i - l(u)_i|\} = 1$, where $1 \leq i \leq d$, because jumping from u to v, the distances between u and all landmarks alter by at most one. Therefore, given any two vertices u and v, $Max\{|l(v)_i - l(u)_i|\}$ indicates a lower bound of the pairwise shortest path distance between them. Thus, if the sum of lower bounds of a vertex v's distance to s and t is greater than an upper bound of $|p_{st}|$, denoted as $UB(|p_{st}|)$, then v must not reside on p_{st}. □

Although Lemma 2 indicates a filter on the possible vertices to explore, the cost to examine the entire graph set remains unacceptable. We could rule out some candidate vertices based on their distances to all landmarks, as guaranteed by the following:

LEMMA 3. *Given s and t, a vertex v possibly resides on p_{st} if for $p_{st} \in [LB(p_{st}), UB(p_{st})]$ and $1 \leq i \leq d$, assuming $l(s)_i \leq l(t)_i$, then*

$$l(v)_i \in \begin{cases} [l(s)_i, l(t)_i] & \text{if} \quad \arccos \frac{l(s)_i^2 + l(t)_i^2 - |p_{st}|^2}{2l(s)_i l(t)_i} \leq \frac{\pi}{2} \\ [h, l(t)_i] & \text{else} \end{cases}$$

where $h = \frac{2(\alpha(\alpha - l(s)_i)(\alpha - l(t)_i)(\alpha - p_{st}))^{\frac{1}{2}}}{p_{st}}$, $\alpha = l(s)_i + l(t)_i + p_{st}$.

Lemma 3 can be easily proved following the cosine law and the Heron's formula. By applying the filtering criteria suggested in Lemmas 2 and 3, we could obtain a subgraph of G, denoted as g_{st}, which must be a superset of p_{st}. Note that $\forall v \in g_{st}$, v's degree is at least 2 and all of v's neighbors belong to g_{st}. This is easy to prove by contradiction. Then, we start an iterative validation process on g_{st} to obtain p_{st} by filtering out unnecessary vertices step by step, as described in Algorithm 4.

Algorithm 4 employs a range label to check if a vertex resides on the path p_{st}. Each vertex that receives a lower(upper) bound of the range label, it sets up the list to watch if any upper(lower) bound would be sent from the same vertex, e.g. *v.swatch* and *v.twatch* in lines 9 and 14 respectively. Initially, s and t are only half bounded, and they pass on the range to its neighbors. Iteratively, if a vertex

v finds that it receives both the lower and upper range bounds from the same vertex, as examined in the two **IF** clauses on lines 7 and 11, v definitely does not reside on p_{st}. Therefore, v can be marked as inactive, and it will not participate in any further computation. Finally, all vertices that remains active and closely bounded shall be returned.

Algorithm 4: Graph exploration for p_{st}

```
   Data: g_st
   Result: p_st
1  for v ∈ g_st do
2      v.state← active; v.range← (−∞, +∞);
3  s.range← (s, +∞); t.range← (−∞, t);
4  s and t broadcast their range to all neighbors;
5  repeat
6      if v receives lower range update (s, +∞) then
7          if the message source vertex is not in v.twatch list then
8              v.range← (s, ⋆);
9              add the message source vertex to v.swatch list;
10             v forwards the lower range to neighbors that are not in v.swatch;
           else
               v.state←inactive;
11     if v receives upper range update (−∞, t) then
12         if the message source vertex is not in v.swatch list then
13             v.range← (⋆, t);
14             add the message source vertex to v.twatch list;
15             v forwards the upper range to neighbors that are not in v.twatch;
           else
               v.state←inactive;
   until s and t are closely bounded;
16 p_st ← all active vertices in g_st that are closely bounded;
17 return g_st;
```

Correctness. There are only two cases where v does not reside on p_{st}, as shown in Figure 5. One is that v reaches both s and t from a same vertex u, as shown in Figure 5(a). In this case, according to Algorithm 4, v would receive range updates from u only, thus it will be pruned. The other case is that the sum of two shortest path distances $|p_{uv}|+|p_{u'v}|$ is larger than $|p_{uu'}|$, where u and u' resides on p_{st}, as shown in Figure 5(b). Thus, the algorithm terminates before all vertices on the path p_{uv} and $p_{u'v}$ get closely bounded, and these paths would be removed eventually.

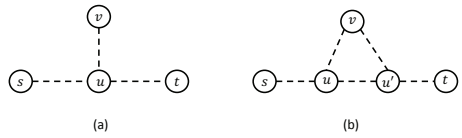


Figure 5: Two cases where v does not reside on p_{st}

Complexity. Obviously, Algorithm 4 takes the space complexity of up to $O(|g_{st}|)$, and the total iteration step of Algorithm 4 is the same as $|p_{st}|$. And within each step, only vertices with range updates would send out messages to selected neighbors. Therefore, comparing to the naive exploration method, Algorithm 4 reduces the communication cost at each superstep. While comparing with Algorithm 3, Algorithm 4's total number of iteration steps is fixed. It makes Algorithm 4 more generic for all possible workloads.

Note that it is trivial to add a global counter in Algorithm 4 to record each vertex's shortest path distance to s and t. Then the exact $|p_{st}|$ can be obtained after the program execution.

Both Algorithms 3 and 4 proposed in this section take the advantage of the data model introduced in Section 3.1. The difference is that Algorithm 3 aims at fast validation of $|p_{st}|$ with unnecessary vertex access eliminated as much as possible under the help of vertex labeling. Algorithm 4 first uses vertex labeling to identify a super set of p_{st} to explore, then conduct the exploration in a way that eliminate communication as much as possible.

Our guided graph exploration method could serve as a building block to evaluate other distance aware queries. For example, in the network field, there are common requests like routing a package from s to t that must pass or must not pass some given node within a transfer budget. Our vertex label method makes it straightforward to estimate the cost to include or exclude a vertex on the shortest path exploration. Therefore, cost aware solutions can be easily constructed to discover such a constraint routing path efficiently.

3.3 Betweenness Assessing On-exploration

Recall that our problem definition is to find an optimal minimum closed dominance set of vertices of the highest accumulative betweenness centrality upon two sets of input vertex sets X and Y given at *ad hoc*. The guided graph exploration method introduced above is to help reduce the unnecessary vertex access and communication. Although it is possible to run the existing betweenness computation algorithm after $\mathcal{G}$ is obtained, which contains exactly all pairwise shortest paths between X and Y, we find that an on-exploration betweenness assessing or ranking method is worth investigating as no exact betweenness value is required.

Following the context of v-p dominance and u-v dominance given in Section 2, we investigate how to extract such dominance correlations during the graph exploration. As betweenness assessing and ranking is orthogonal to the guided graph exploration part, we have no assumption on the graph to be explored. Given a graph G and input vertex sets X and Y, we employ the vertex centric model to do the exploration for betweenness assessing. After the exploration is terminated, we are able to answer the following query, given two vertices $u, v \in V$, how is $C_B(u|X,Y)$ comparing to $C_B(v|X,Y)$. Note that here u and v should at least satisfy v-p dominance, otherwise they are not possible to be returned in the final answer.

Obviously, if $u_{\vdash}v$, then $C_B(u|X,Y) \geq C_B(v|X,Y)$. The challenge is that if u does not dominate v, how do we compare their betweenness. Intuitively, $C_B(u|X,Y)$=$|\{u_{\vdash}\}^P|$+f, where f denotes u's contribution to other pairwise shortest paths that it does not dominate. We address the problem by transforming pairwise shortest paths into a set of two level tree structures to record the dominance correlation to help compute f efficiently. We elaborate with a simple example as shown in Figure 6. The shortest paths between s and t are shown in the left part of the figure. p_{st} is broken into three two level trees, where each tree's root is a vertex that dominates multiple vertices along the exploration path from s to t. The dash line pointing from v_4 to v_1 indicates that v_4 connects to all of v_1's children in p_{st}. The dash line pointing from v_6 as a child to v_6 as a root indicates that v_6 does not dominate p_{st}. Given such a transformation, it is now easy to compare two vertices betweenness without computing the exact value. For example, to compare the betweenness of v_5 and v_9, we find that v_6 is on the same level with two vertices v_5 and v_7, yet it dominates two other vertices including v_9, thus v_5 and v_9 contribute equally to p_{st}. Let T_{st} be the set of trees generated from p_{st}, then we could employ the following Lemma for vertex betweenness assessing:

LEMMA 4. *If u and v are recursively dominate by the same vertex in T_{st}, then $C_B(u|s,t) \geq C_B(v|s,t)$ **iff** u recursively dominates more vertices than v does in T_{st}; If u and v are not recursively dominated by the same vertex in T_{st}, then $C_B(u|s,t) \geq C_B(v|s,t)$ **iff** $\frac{rcc(u)}{rcc(u_{root})} \geq \frac{rcc(v)}{rcc(v_{root})}$, where function $rcc(\cdot)$ denotes the count of recursively dominated children, u_{root} (v_{root}) denotes the vertex that recursively dominates u (v) and not be dominated by any other vertex.*

Lemma 4 can be proved by the definition of betweenness, as given in Section 2. As a matter of fact, the generation of T_{st} during graph exploration is straightforward. No extra space needs to

be allocated for T_{st}. We simply keep the count of children that a vertex dominates, and link the root vertices which connect to the same children, like v_1 and v_4 shown in Figure 6.

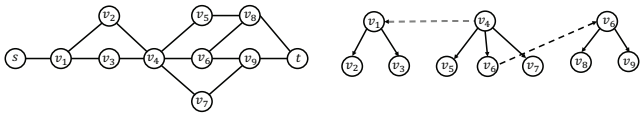


Figure 6: Breaking p_{st} into a set of two level trees

A great advantage of this on-exploration betweenness assessing is that it perfectly fits a parallel execution mode under a distributed setting. As all partial results can be directly assembled for betweenness assessing. Then, given input vertex sets X and Y, we could generate (or mark more precisely, as no extra space is consumed) $T_{x_iy_j}, \forall x_i \in X, y_j \in Y$. Then only the vertices which are not dominated by others in $T_{x_iy_j}$ are saved for further computation.

So far, we have elaborate the two essential building blocks for VSB query evaluation, which are guided graph exploration and betweenness ranking on-exploration. However, to evaluate a VSB query in the most time efficient way, a careful query evaluation plan needs to be considered.

4. QUERY PROCESSING

To evaluate the VSB query Q in the most time efficient manner, it is worth investigating an efficient query planning strategy that promises bounded network communication and memory footprint. Recall that the answer to a VSB query is a set of vertices that forms an optimal minimum dominance of P_{XY} and has the highest accumulative betweenness. Given the two building blocks introduced in Section 3, a naive query plan can be easily developed. We simply apply the same computation procedure on each p_{xy}, where $x \in X$, $y \in Y$, and assemble the final results based on a reduction of every p_{xy}'s dominance vertices, as described in Algorithm 5. Note that a temporary data set D_{xy} is employed in the algorithm to store all the dominant vertices of p_{xy} (on Line 4). As Line 1-3 applies the same computing procedure to all pairwise paths between X and Y, this part can be executed in parallel. The reduction process on Line 4 is to reduce the dominance vertices of each path to a single set $\mathcal{D}$ and then compute centrally.

Algorithm 5: Naive VSB query evaluation

```
   Data: ,Q = ⟨G, X, Y, R⟩
   Result: R
1  for x ∈ X, y ∈ Y do
2      D_xy ← ∅; Explore for p_xy;
3      D_xy ← D_xy + {u}, ∀u ∈ p_xy and u⊢p_xy;
4  Reduction D = ⋃{D_xy};
5  for D_xy ∈ D do
6      for D_x'y' ∈ D,x' ≠ x, y' ≠ y do
7          if ∃u ∈ D_xy ∩ D_x'y' then
8              Push(u, q); /* q is a priority queue based on
                   the size of {u⊢}^P                    */
9      u ← Pop(q);
10     R ← R + {u};
11     D ← D − ⋃{D_x'y'},where u⊢p_x'y';
12 return R;
```

THEOREM 1. *Algorithm 5 takes up $O(|X||Y||p_{xy}|)$ space, communicates at $O(|X||Y||p_{xy}|)$ volume of data, and runs at the time complexity of $O(|X|^2|Y|^2)$, returns an optimal minimum dominance of P_{XY} R having $AB(R) > \frac{1}{2}AB(R^*)$, where R^* is the optimal answer,.*

PROOF. For each pairwise shortest path p_{xy}, the temporary dominance vertex set D_{xy} computed on Line 4 can be as large as $|p_{xy}|$, which explains the space and communication complexity. The nested loop structure indicates a comparison between a path against every other path, which is of complexity $O(|X|^2|Y|^2)$.

We prove R is an optimal minimum dominant set of P_{XY} by contradictory. Assume there exists another vertex $u \in p_{xy}$ that dominates $v \in R$. As $v \vdash p_{xy}$ holds, therefore, both u and v are pushed into the priority queue (Line 9). However, v is returned only if it is the vertex of the largest dominance in the priority queue, meaning $\{v\vdash\}^P \supseteq \{u\vdash\}^P$, which indicates the assumption must be invalid. As we elaborated before, a vertex u's betweenness $C_B(u)$ equals to $|\{u\vdash\}^P| + f$, where f is u's contribution to other shortest paths that it resides on but does not dominate. Clearly f cannot exceed 1, therefore, at each step a returned result's betweenness is at least $\frac{1}{2}$ of the optimal choice. Accumulatively, the final $AB(R) \geq AB(R^*)$. □

Algorithm 5 is straightforward and easy to implement, and it works for all query workload. However, its efficiency can suffer from the all-to-one large volume of data copy in the reduction step (Line 4). Meanwhile, the efficacy of Algorithm 5 can be further improved if we take the f part of a vertex's betweenness estimation into consideration. Thus, we develop several optimization techniques to improve the performance of VSB query processing.

4.1 Path Sharing

First of all, we optimize the query evaluation by taking the advantage of vertex distribution in X and Y. As introduced in the motivation example, a VSB query can be very useful to find the bonding between communities, where a community must be composed of vertices that are close to each other. This is crucial for the query optimization as it implies the potential of shortest path sharing property. The example shown in Figure 1 represents such kind of workload. In this case, there are two problems to solve: 1) how to quickly decide the input vertex distribution, as X and Y are given at *ad hoc*; 2) how to make the best of path sharing.

We solve the first problem with group prediction using vertex labels. Note that all vertices are labeled by their distances to landmarks. Graph G is partitioned into a number of small graphs that have a diameter restriction. Thus, given two vertices u and v, if both $l(u)_i$ and $l(v)_i$ is smaller or equal to δ (the graph partition parameter defined in Section 3.1), it is certain that u and v are in the same partition graph. Intuitively, if u and v share similar distances to multiple landmarks, they are close to each other. Given a VSB query $Q = \langle G, X, Y, R\rangle$, we first partition vertices in X and Y according to their labels. With so many distance based clustering algorithms off-the-shelf, we choose the simplest one. We groups vertices of the same small graph partition together to obtain long shared paths, such that the exploration cost can be greatly saved.

The second problem essentially concerns how to identify the shared paths when vertices are grouped. As a matter of fact, such shared paths can only be determined during the runtime. Sometimes, vertices that are close to each other may not share an single edge to destinations at all. Therefore,, we only need to identify the region or the boundary of shared paths to save the exploration cost. Given a set of grouped vertices, denoted as X', we simply add a virtual node x_v to the graph to represent X'. The trick is how we decide $l(x_v)$.

LEMMA 5. *Given a set of vertices X', a virtual vertex x_v of label $l(x_v)_i = \lceil \frac{Max\{l(x')_i\} - Min\{l(x')_i\}}{2} \rceil$, where $x' \in X'$, guarantees $\forall x' \in X'$: 1) if $|p_{x'y}| < |p_{x_vy}|$, $p_{x'y} \subset p_{x_vy}$; 2) if $|p_{x'y}| > |p_{x_vy}|$, $p_{x_vy} \subset p_{x'y}$.*

PROOF. Consider the case when $|p_{x'y}| < |p_{x_vy}|$. Clearly, two adjacent vertex's label difference on every dimension is at most one, where the label is a d-dimensional vector. Thus, the label of $l(x_v)$ defined in the Lemma indicates the center of X', which reaches

every vertex x' in X' with minimum hops. Therefore, the path $p_{x_v y}$ must passes x', implying $p_{x'y} \subset p_{x_v y}$. Similarly, the case when $|p_{x'y}| > |p_{x_v y}|$ can be easily verified. □

By employing the path sharing, we could greatly save the concurrent exploration cost of Algorithm 5, as well as the space and communication cost on D_{xy}, since the total number of such dominant vertex set are reduced.

4.2 Probe-based Communication

A main bottleneck of Algorithm 5 is its all-to-one communication at the reduction part, which brings about a burst of data copying over network. Instead of such a brute-force solution, we develop a probe-based lookup strategy which could greatly save the overall communication cost. As introduced in Section 3.1, each graph vertex is associated with a set of independent hash functions, denoted by H. Thus, we could use H to build up a bloom filter for the element-in-set test, which is essential to our probe-based communication. To elaborate, instead of directly copying D_{xy} over network (Line 5 in Algorithm 5), we first compute the bloom filter of each D_{xy} for p_{xy}, denoted as $\mathcal{F}_{xy}$, which is a m_f bits vector. Then we pass m_f to threads examining other pairwise shortest paths. In this way, each thread can check if any dominant vertex it finds could also be dominant vertex on other paths. Although the bloom filter may introduce false positive, it greatly reduces the size of data to transfer for verification. Another benefit of using probe-based communication is that most computation is local, such that the centralized computing workload (Lines 5-11 in Algorithm 5) could be reduced. Following the same context of Algorithm 5, we show how the probe-based communication is employed to evaluate a VSB query in Algorithm 6.

Algorithm 6: VSB query evaluation with probe-based comm.

```
   Data: ,Q = ⟨G, X, Y, R⟩
   Result: R
1  for x ∈ X, y ∈ Y do
2      D_xy ← ∅; Explore for p_xy;
3      D_xy ← D_xy + {u}, ∀u ∈ p_xy and u⊢p_xy;
4      F_xy ← BloomFilter(D_xy, H); Broadcast(F_xy);
5      for u ∈ D_xy do
6          Push(u, q); /* q is a priority queue based on the
               number of filters u hits                    */
7      while q ≠ ∅ do
8          u ← Pop(q);
9          if u is valid then
10             if ∄r ∈ R, r⊢u&&{R⊢}^P ⊇ {u⊢}^P then
11                 R ← R + {u};Break;
12 return R;
```

In Algorithm 6, we eliminated the centralized computation. Although R is a shared variable, a distributed lock can be employed for synchronous updates. As the algorithm shows, it is easy to be executed in parallel, e.g., each computing thread computes for each pairwise shortest path. Apparently, the communication cost is much reduced comparing to Algorithm 5, since only the bloom filter vector is transferred in the first place. The verification later on (Line 9) transfers one vertex's label at a time. More importantly, each thread aborts as soon as it contributes a dominance vertex to R, or finds out that a residing path is already in R. This early stop property leads to a fast convergence of the final answer. It is worth pointing out that Algorithm 6 achieves the same approximation ratio on $AB(R)$ as Algorithm 5, as long as R greedily chooses a vertex u of the largest $|\{u_\vdash\}^P|$ at each synchronous update. Comparing to the path sharing technique, which only benefits when input vertices tend to be close to each other, this probe-based solution is generic for all kinds of workloads.

4.3 Degree-based Approximation

The above two optimization techniques, path sharing and probe-based communication improve query processing efficiency by reducing redundant data access and data copy over network. But neither of them helps return more accurate results. To improve the efficacy of Algorithm 5, we develop an a (1-ϵ) approximation algorithm, where $\epsilon \in (0,1)$ is a user specified parameter.

Recall that the approximation ratio of Algorithm 5 being $\frac{1}{2}$ is because for each path p_{xy} we select a dominant vertex u, another vertex v which contributes to almost 100% of shortest paths connecting x and y and does not dominated by u may not be selected. Thus, to achieve the (1-ϵ) approximation ratio, we need to search for such kind of vertex like v whenever a vertex u is adding to R.

Our method is to add a vertex filtering process to the generation of D_{xy} for each p_{xy} in Algorithm 5, which can be easily modified to fit Algorithm 6 as well. For path p_{xy}, to add u as a dominant vertex to D_{xy}, we search for a vertex v that belongs to p_{xy} and takes up at least $\frac{\epsilon}{1-\epsilon}$ fraction of shortest paths between p_{xy}, and add all such v to D_{xy}. To check if a vertex v fits the filtering criteria, we need to estimate its betweenness based on the two-level tree structure we introduced in Section 3.3. Let T denote the two-level tree structure generated on-exploration of p_{xy}. Thus, each vertex in T is associated with a time step. We iteratively select the vertices of the same time stamp with accumulative highest degrees, e.g., $v_{i=0,\ldots,l}$ with degree $dgr_{i=0,\ldots,l}$ accordingly, we report v_i that has $\frac{dgr_i}{\sum_{j=0,\ldots,l} dgr_j} \geq \frac{\epsilon}{1-\epsilon}$, as well as v_i's ancestors in T. In this way, we do not need to compute the exact betweenness for each vertex in p_{xy}. Since we always start from the vertex of the minimal betweenness in p_{xy}, the betweenness recurrence given in Eq.2 guarantees that our method is false negative free. Note that the time stamp based degree ranking in our method takes $O(|p_{xy}|log|p_{xy}|)$ time, which does not become the dominate cost for Algorithms 5 and 6. Although it may populate the size of D_{xy} to exact p_{xy}, the space complexity of these two algorithms remains unchanged.

So far, we have present our solution for VSB query processing with efficiency and efficacy guarantees. Three optimization techniques introduced above are in fact orthogonal to each other, as they improve the query evaluation performance from different perspectives. Later on in the experiment section, we shall study how these optimization techniques would affect the query processing performance on different types of workloads.

5. EXPERIMENTS

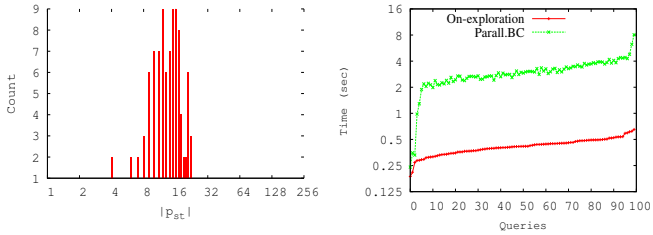
Our experiment study mainly includes three parts: 1) how the proposed guided graph exploration and betweenness ranking on-exploration help VSB query processing; 2) how different query processing algorithms work under different query workloads.

5.1 Setup

We build up the test bed using the Google Cloud platform, using 6 servers of the n1-highmem-8 type. Each server has 8 virtual CPUs, 52GB memory and 1TB persistent disk, running Debian 7 of Linux kernel 3.2.0-4-amd64. We choose GraphLab [31] to build the prototype system, as it supports both BSP based graph computation model and the message passing model. Our program is written in C++ and compiled with gcc 4.7.2(switch O3 is on).

Table 1: General statistics of employed graph data sets

ID	# of Nodes (Million)	# of Edges (Million)	G_{diam}	Size (GB)
A	∼428	∼454	78	3.6
B	∼1,825	∼65,219	5,328	869.2
C	∼33	∼1,108	7	25.6
D	10,000	23,946	2,927	42.2



(a) $|p_{st}|$ *distr.* of 100 random queries (b) Time cost on betweenness ranking

Figure 7: Betweenness ranking on different workloads

Experimental Data Sets We employ four data sets of different scales and topologies in the experiments, as briefly summarized in Table 1. Data set A describes the web graph of the TREC 2009 Category B data set, which is the set of the first 50 million English pages collected in January and February 2009 by the Language Technologies Institute at CMU. Data set B comes from the WebGraph 2012 project [35], which is extracted from the Web cropus released by the Common Crawl Foundation in August 2012. Data set C is a crawled social graph from twitter [30]. Note that we only employ the largest connected component of graph data B and C, and make the graphs undirected. Synthetic data sets D is a random graph generated with the *igraph*[14] package.

Query Workloads For each VSB query, we randomly select 10∼100 vertices as input X and Y. We generate three types of queries, which essentially represent different kinds of workloads: 1) $\forall x \in X$, $l(x)_i \leq \delta$, $\forall y \in Y$, $l(y)_j \leq \delta$, where $i, j \in [1, d]$ are randomly selected, i.e., both vertices in X and Y are close to each other, denoted as XLYL; 2) $\forall x \in X$, $l(x)_i \leq \delta$, where $i \in [1, d]$ is randomly selected, $\forall y \in Y$ is randomly selected, denoted as XLYR; 3) both X and Y are randomly generated from G, denoted as XRYR. We would like to show that our solution works well for all kinds of workloads, and the optimization techniques we proposed would be very useful for certain kind of workload. We generate 100 VSB queries for each type of workload, and evaluate the batch one by one. We run every job batch with 3 cold-start and report the average execution time.

Evaluation Plan We first show our experiments on the graph preprocessing part and validate the effectiveness of the two essential building blocks proposed in Section 3. We compare our method with GraphLab's shortest path utility implementation and show the differences *w.r.t.* time efficiency. Then we evaluate the effectiveness and efficiency of Algorithms 5 and 6.

5.2 Preprocessing & Two Building Blocks

As presented in Section 3.1, we can select d landmarks using either a deterministic or a random algorithm. δ is the crucial parameter to choose. Intuitively, the larger δ is, the number of vertices covered by a single landmark gets larger, which leads to a smaller d. Experiments also validate this point. In Table 2, we report the time efficiency of graph preprocessing and the value of d accordingly, as well as the total disk space cost after preprocessing.

Regarding time efficiency, we have two observations from the results. First, by increasing δ, d drops more significantly if a deterministic algorithm is employed comparing to using a random algorithm. For example, when δ increases from 16 to 32 in graph B, d drops from 1429 to 879, which almost drops a half using the deterministic algorithm. On the contrary, by using the random algorithm, it only drops from 2574 to 1782. Second, although a random algorithm always generates more partitions, it is still a winner *w.r.t.* time efficiency. Meanwhile, as shown in Table 2, the extra space cost of vertex labeling turns to be manageable even δ is set to a small value. Although each vertex is presented with a d bytes vector during query processing, the label vectors are initially compressed and recovered only upon data access. The reported data sizes in Table 2 are the ones with vertex label compression applied, as elaborated in Section 3.1.2. A straight forward observation is that if G is power-law graph with large G_{diam}, like graph B and D, smaller δ promises better compression ratio. For example, the sizes of graph B with δ set to 8 and 16 are very close. This property is guaranteed by the characteristic of value based compression. Moreover, if the data graph is extremely dense with a small diameter, like graph C, the extra space cost on vertex labeling drops significantly when δ increases, as the number of landmarks would be very limited.

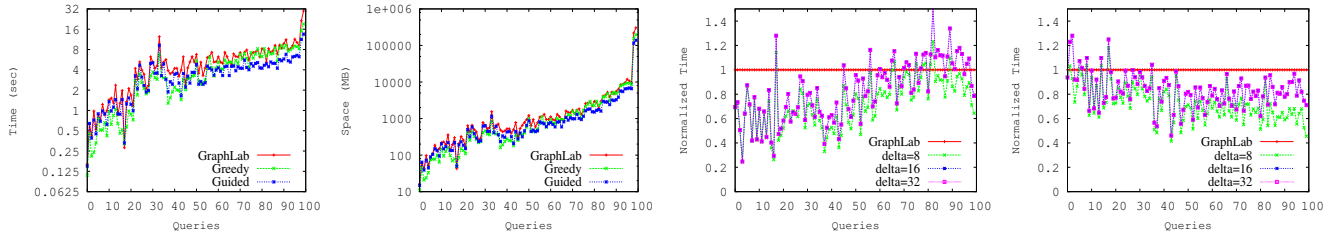
Table 2: Graph preprocessing using different algorithms

ID	d			T(sec)		Size(GB)	
	δ	dm.	rd.	dm.	rd.	dm.	rd.
A	4	64	98	146	39	33.2	57.7
	8	36	78	129	32	22.4	41.3
	16	8	42	89	27	8.2	23.6
B	8	2,231	3,029	549	227	4,216	4,248
	16	1,429	2,574	531	189	4,094	4,225
	32	879	1782	492	141	2,709	2,799
C	1	126	145	329	124	1,139	1,178
	2	10	26	69.2	36.4	95.6	105.4
	4	3	59	2.3	78.5	78.9	131.7
D	8	1,576	2,109	421	179	1,465	1,509
	16	1,206	2,005	392	164	1,437	1,486
	32	457	1,324	354	139	1,128	1,305

To validate the guided graph exploration and betweenness assessing on-exploration, we randomly pick 100 pairs of vertices from each graph and ask for p_{xy}, and rank the betweenness of two random vertices from p_{xy}. As a comparison, we employ the GraphLab's shortest path utility implementation and the parallel betweenness computing algorithm introduced in [4]. Due to the space limit, we highlight our findings on graph B. Figure 7(a) shows the $|p_{st}|$ distribution of 100 random queries. Figure 7(b) compares the time cost to rank two random vertices's betweenness using parallel betweenness computation and our on-exploration method. Clearly, on-exploration betweenness ranking achieves significant time saving as it does not need to consider the entire graph and compute the exact betweenness value.

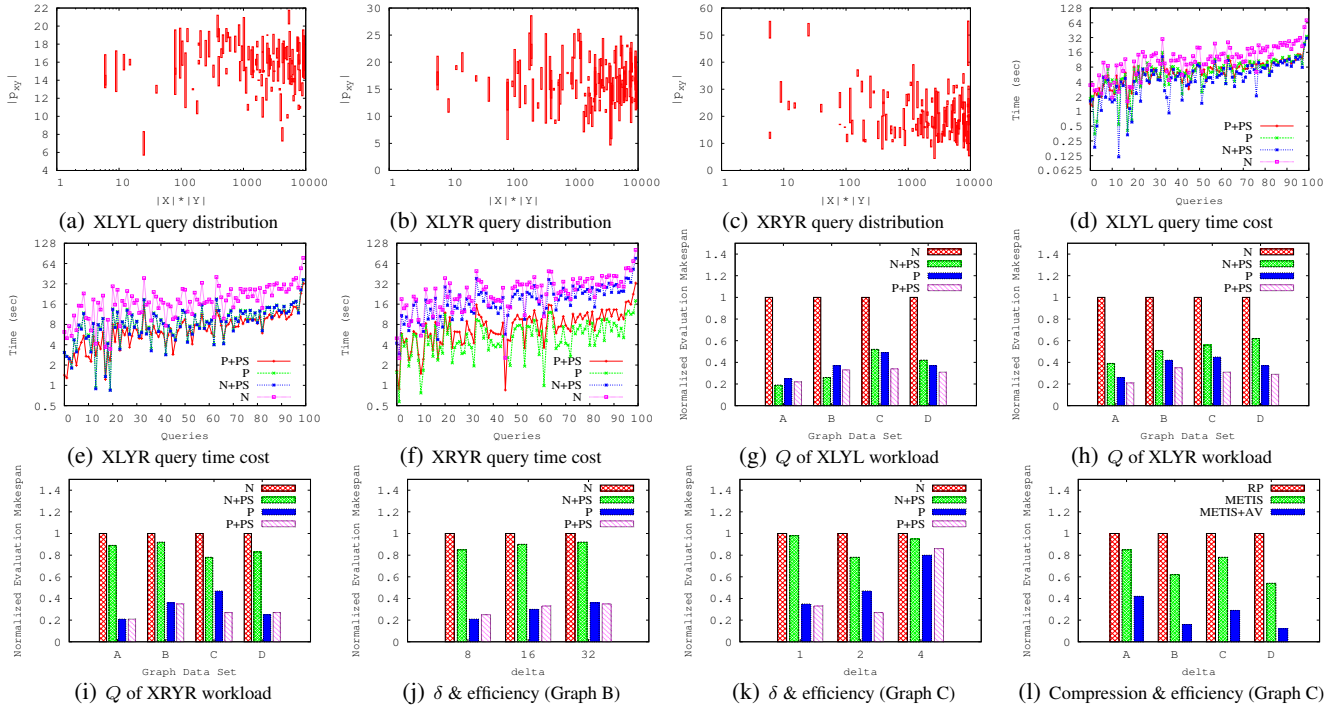
Figure 8 shows how our methods, greedy (Algorithm 3) and guided exploration (Algorithm 4), compare to the GraphLab's shortest path utility in p_{st} evaluation under the same workload shown in Figure 7(a). Figure 8(a) and 8(b) show the time and space cost respectively. Space cost is the total size of data access on the distributed storage. Note that the queries of x axis are sorted in an ascending order of $|p_{st}|$, and y axis is presented in logarithm scale. As shown in Figure 8(a), when $|p_{st}|$ is small, the greedy method's execution time is only about half of the GraphLab's method. The reason is that Algorithm 3 terminates quickly with less vertex access. With the increasing of $|p_{st}|$, the greedy algorithm's efficiency drops and sometimes even performs worse than the GraphLab's method. Because when $|p_{st}|$ grows, it takes the greedy algorithm more iterations to guess the correct $|p_{st}|$. On the contrary, guided exploration performs stable and achieves more time saving when $|p_{st}|$ grows.

To investigate how δ affects our algorithm, we present time cost of greedy and guided exploration with different δ setting on the same workload in Figure 8(c) and 8(d). Note that we normalize all the time cost using GraphLab's result, as it does not rely on the setting of δ. Apparently, our algorithm achieves more speedup when δ is smaller, which is reasonable as it promises better pruning power. Another observation from the result is that, comparing to the guided exploration, the greedy algorithm is more resistent to different δ. It is because greedy algorithm uses vertex label pruning in a passive way, while the guided exploration employs the pruning actively be-



(a) Time cost to find p_{st} ($\delta = 8$) (b) Space cost to find p_{st} ($\delta = 8$) (c) (T) Greedy v.s. GraphLab with diff. δ (d) (T) Guided v.s. GraphLab with diff. δ

Figure 8: The speed up of evaluating p_{st} queries



(a) XLYL query distribution (b) XLYR query distribution (c) XRYR query distribution (d) XLYL query time cost

(e) XLYR query time cost (f) XRYR query time cost (g) Q of XLYL workload (h) Q of XLYR workload

(i) Q of XRYR workload (j) δ & efficiency (Graph B) (k) δ & efficiency (Graph C) (l) Compression & efficiency (Graph C)

Figure 9: (a)-(i) Different query workloads test; (j)-(k) Evaluation speedup using different δ (Graph B & C); (l) Compression v.s. efficiency

fore making a decision on the next hop. Clearly, as shown in Figure 8(d), when $|p_{st}|$ is large, the guided exploration is more sensitive to the setting of δ. Although smaller δ works the best for the path query, there is the greater extra space overhead to trade off.

5.3 VSB Query Evaluation

We evaluate our proposed solution from both the efficiency and efficacy perspectives. We first set δ for the four data sets as 8, 32, 2 and 32 respectively to compare the effectiveness of our proposed query processing solution.

Query Efficiency In Section 4.1, we propose a naive VSB query processing solution (Algorithm 5) and two optimization techniques to improve the time efficiency. To validate the proposed solution, we report how the combination of optimization techniques serve the query evaluation, particularly, on different query workloads. Due to the space limit, we highlight our results on graph B in Figure 9. Figures 9(a) to 9(c) show the distribution of random queries we generated, where queries are sorted according to their input size ($|X| \times |Y|$ as x axis). Figures 9(d) to 9(f) show the time costs for different query evaluation methods over different workloads, where N stands for the naive algorithm, P stands for the probe-based communication solution (Algorithm 6), PS stands for path sharing. Apparently, if input vertices are close to each other, path sharing would achieve great time saving, as shown in Figure 9(d). On the contrary, when query inputs are randomly selected, as shown in Figure 9(f), probe-based method usually performs better.

We report the normalized query processing makespan of different methods on all data sets in Figures 9(g) to 9(i). We have made two observations from the efficiency experiments. First, given the same query workload, the underlying graph structure would greatly affect algorithm performance. Take graphs A for example, it is much more sparse than graph C. As shown in Figure 9(g), over the same query workload, the best evaluation strategy for graph A is path sharing while for graph C it is a combination of path sharing and probe-based communication. Clearly, reducing network communication as much as possible for a dense network brings more benefits than packing shared paths. Second, path sharing clearly helps a lot when the vertices in X or Y are close to each other. For example, for the XLYL workload, comparing to the naive algorithm, we can obtain almost 5x speed up on graph A by applying path sharing. On the contrary, the probe-based communication method performs more stable on different workloads. One thing to notice is that combining path sharing with probe-based solution does not double the speedup. The reason is that path sharing reduces the concurrent computing threads itself, but makes the computing workload of each thread unbalanced. Note that in Algorithm 6, R is updated with synchronization, which could easily suffer from unbalanced current computing workloads.

Another critical concern is that how δ affects the query evaluation performance. As the algorithms we proposed are based on the

guided graph exploration method, therefore, we observe the similar trend of efficiency improvement when δ decreases as shown in Figure 8(d). We highlight our findings using the results from graph B and C. For graph B, as shown in Figure 9(j), the path sharing optimization method is closely bounded with the total number of vertices to explore. Therefore, the probe-based method is essential to the performance improvement. For graph C, as shown in Figure 9(k), due to the density property, path sharing is desirable when δ is set to a proper value, like δ=2. When $delta$ equals to 1, there is not much optimization space left after a probe-based method is employed, as the pruning power on vertex exploration is sufficiently strong. On the contrary, when δ equals to 4, almost the entire graph needs to be considered to extract the shared path, which would result in severe performance decay. The hints we learn from the results are that if the query workload is unknown, smaller δ is preferred for fast query processing as long as the extra space cost is manageable; the crucial performance optimization lies in reducing the total number of vertices to access and compute; path sharing does not help with the speedup if δ is too small or too large.

Another observation is that compression always helps. In Figure 9(l), we show how compression affects the overall query processing efficiency. As elaborated in Section 3.1, we employ METIS to partition graph and only store the adjusted value σ, denoted as METIS+AV. Compared to random partition (RP), or METIS partition only, METIS+AV achieves the best compression ratio, normally around 0.026. RP gives the worst compression ratio (around 0.17) since it hurts the locality of a vertex label values. Only METIS could achieve a compression ratio of about 0.09. Clearly, for large graphs, like B and D, better compression ratio brings about significant time saving, since the I/O operation is inevitable and usually becomes the dominance cost.

Query Efficacy Other than the efficiency optimization to the naive algorithm, we also present a degree-based vertex filtering process to help improve the approximation ratio of Algorithm 5 from 0.5 to (1-ϵ), where $\epsilon \in (0, 1)$. We verify the effectiveness of this method over different graph data with random query workloads. Due to the space limit, table 3 summarize our findings when different ϵ over graph B with 3 random VSB queries. In the table, the second column is the accumulative betweenness of returned result R using the algorithm 5, and T in the third column indicates its computation time. For the rest columns, we check how different ϵ would affect the returned results, denoted as $AB(R')$ and the time cost to pay. Apparently, when ϵ gets smaller, we can obtain a better result. And the extra time cost is not that expensive, which validate our claim that the degree-based vertex filtering processing would not dominate the computation complexity.

Table 3: Improving the efficacy of VSB query processing

Q	$AB(R)$	T(sec)	$AB(R')$ and Time					
			ϵ=0.5	T(sec)	ϵ=0.25	T(sec)	ϵ=0.125	T(sec)
1	5.75	2.32	6.25	3.29	7.0	5.28	7.0	5.88
2	21.25	93.46	24.92	126.7	26.24	159.27	26.24	172.45
3	8.45	9.28	8.45	12.49	8.45	13.52	8.87	23.47

6. RELATED WORK

Distributed Graph Processing Models & Systems State-of-art distributed systems for general purpose graph processing like Trinity [45] or GraphX [22], all support Pregel [34], a vertex-centric computing model of proven scalability and flexibility. Since our work is based on the native vertex-centric model, our implementation can directly serve as a plugin on these systems. Improvement works over Pregel, like Pregelix [9], Blogel [47], are orthogonal to our solution and can be applied to yield better overall performance.

Distance Based Query over Distributed Graph Employing landmarks to approximate the shortest path distance is a widely adopted technique[29][40][37]. The basic idea is to pre-compute the shortest distances between all the nodes and selected landmarks, and then apply the triangle inequality to help estimate the shortest path distance. Work [40] investigates finding the optimal set of landmarks. In particular, they target on answering the pairwise shortest path distance query. They introduce the LandMark-Cover problem, which is to find a minimum number of points such that given any pair of vertices u and v, there exists at least one landmark residing on the shortest path from u to v. This problem is proven to be closely related with the 2-hop labeling scheme [12]. Landmark based methods do not aim to provide the exact distance. Instead, they use a small number of landmarks to do estimation. Tao et al. [46] introduce the k-skip shortest path, which is a natural substantial of returning the exact shortest path. Intuitively, it reports a set of vertices V that consecutively reside on a shortest path from s to t, having every vertex on this path is at most k-hop away from at least one vertex in V. Following up works, like graph simplification[44], shortest path discovery over road network[18][48], employ similar concepts to perform a distance preserving graph partition. The δ-evenly coverage landmark selection defined in this work, however, is orthogonal to the k-skip concept. Because shortest path is not the substantial concern in our problem. We select landmarks to serve online graph exploration. There is no sequence semantic of our landmarks. In other words, k-skip returns more vertices residing on the shortest path of two query points when k decreases. On the contrast, given a smaller δ, the δ-evenly coverage serves better in reducing redundant vertex access on exploration step by step. Vertex labeling is another line of research to answer distance queries. Gavoille et al. show that general graphs support an exact distance labeling scheme with labels of $O(n)$ bits [19]. Several special graph families, including trees or graphs with bounded tree-width, have distance labeling schemes with $O(log_2 n)$ bit labels [2]. However, it is infeasible to directly apply these theory results to a large graph of billion nodes, as the space overhead of labeling would be unaccepted. Our solution, on the other hand, simply targets on vertex pruning using distance labels. And due to the δ-evenly coverage landmark selection scheme, the locality of vertices' label vectors is well preserved. Therefore, a simple value based compression could greatly help to reduce the overall space cost on vertex labeling.

Parallel Betweenness Computation Parallel betweenness computation has long been established as a research problem. Early works like [10][26][49][4] achieve excellent performance, but the size of the input graph is very small or a big distributed cluster is used [10]. Subsequent work by Madduri et al. [32] improves the algorithm by using successors instead of predecessors in the computation of the DAG D, which produces a more efficient, locality-friendly algorithm. Edmonds et al. [16] present an approach that targets on the fine-grained parallelism in a distributed memory environment. It initially employs a label-correcting single-source shortest-path algorithm to compute the shortest path distances and predecessor lists. Then, nodes' successors are computed using the predecessor lists. These algorithms compute the exact value of betweenness centrality. To reduce the computational cost, a number of approximation algorithms have been proposed [3][7] [20]. All these works assume that the graph structure is given in advance. On the contrast, our solution is based on a betweenness ranking semantic, which is computed along *ad hoc* graph exploration.

There is also a rich literature on the betweenness based analysis in social network research, particularly in the field of community detection [21][15] and information diffusion [36][24]. Girvan - Newman algorithm [21] iteratively removes the edge of the highest betweenness to compute communities. It re-calculates betweenness

of all affected edges in each iteration. Work [5] explores the sparsity nature of social networks and achieves considerable improvement over Brande's algorithm. Instead of exacting betweenness computing, Maiya et al. [33] propose online sampling algorithm to approximate individual betweenness in a social network. Following the same intuition, k-path centrality [1], defined on a k-hop random walk semantic, is proposed to categorize nodes' centrality rather than to rank all the nodes *w.r.t.* the exact betweenness centrality. To our best knowledge, we are the first to study the vertex set bonding query over distributed graphs. Comparing to the community correlation and information diffusion studies, which are commonly built on various assumptions on graph traversing (or diffuse) models, we define vertex set bonding as a generic optimization problem and propose a complete solution framework. By replacing the bonding metric with other path based centrality measurements, our solution can be applied in various application contexts. It is worth pointing out that the two novel techniques, guided graph exploration and betweenness ranking on-exploration could be directly applied as fundamental building blocks for graph analysis in general purpose.

7. CONCLUSION

In this work, we formally define a ***V**ertex **S**et **B**onding* query, which returns a minimum set of vertices with the maximum importance *w.r.t* total betweenness and shortest path reachability in connecting two sets of vertices. With the development of two novel techniques, guided graph exploration and betweenness ranking on exploration, as well as several optimization techniques, we evaluate VSB queries with both efficiency and efficacy guarantees. Experiments over both real and synthetic large graphs on the Google's Cloud platform validate the effectiveness of our method.

Acknowledgement This work is supported in part by the Hong Kong RGC Project N_HKUST637/13, National Grand Fundamental Research 973 Program of China under Grant 2014CB340303 , NSFC Grant No. 61328202, NSFC Guang Dong Grant No. U1301253, Microsoft Research Asia Gift Grant and Google Faculty Award 2013. This work is also supported by the Hong Kong Research Grants Council (RGC) General Research Fund (GRF) Project No. CUHK 411211, and the Chinese University of Hong Kong Direct Grant No. 4055015 and 4055048.

8. REFERENCES

[1] T. Alahakoon, R. Tripathi, N. Kourtellis, R. Simha, and A. Iamnitchi. K-path centrality: a new centrality measure in social networks. In *Proceedings of the 4th Workshop on Social Network Systems*, page 1, 2011.

[2] S. Alstrup, P. Bille, and T. Rauhe. Labeling schemes for small distances in trees. *SIAM J. Discrete Math.*, 19(2):448–462, 2005.

[3] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, pages 124–137, 2007.

[4] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, pages 539–550, 2006.

[5] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. Fast exact computation of betweenness centrality in social networks. In *ASONAM*, pages 450–456, 2012.

[6] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[7] U. Brandes and C. Pich. Centrality estimation in large networks. *I. J. Bifurcation and Chaos*, 17(7):2303–2318, 2007.

[8] P. B. Brandtzæg, J. Heim, and B. H. Kaare. Bridging and bonding in social network sites - investigating family-based capital. *IJWBC*, 6(3):231–253, 2010.

[9] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[10] A. Bulu and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *IJHPCA*, (4):496–509.

[11] M. Castro and et. al. Future directions in distributed computing. chapter Topology-aware Routing in Structured Peer-to-peer Overlay Networks, pages 103–107. Springer-Verlag, Berlin, Heidelberg, 2003.

[12] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[13] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on a multi-threaded, multi-core platform. In *IPDPS*, pages 688–697, 2011.

[14] G. Csardi and T. Nepusz. The igraph software package for complex network research. 2006.

[15] L. Despalatovic, T. Vojkovic, and D. Vukicevic. Community structure in networks: Girvan-newman algorithm improvement. In *MIPRO, Opatija, Croatia, May 26-30, 2014*, pages 997–1002.

[16] N. Edmonds and et. al. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *HiPC*, pages 1–10, 2010.

[17] W. Fan, X. Wang, and Y. Wu. Querying big graphs within bounded resources. In *SIGMOD*, pages 301–312, 2014.

[18] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. *PVLDB*, 7(10):893–902, 2014.

[19] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, 2004.

[20] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, pages 90–100, 2008.

[21] M. Girvan and M. E. J. Newman. Community structure in social and biological network. In *In Proceedings of the National Academy of Science USA*, pages 99:8271–8276, 2002.

[22] J. E. Gonzalez and et. al. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[23] A. Guille, H. Hacid, C. Favre, and D. A. Zighed. Information diffusion in online social networks: a survey. *SIGMOD Record*, 42(2):17–28, 2013.

[24] A. Guimarães, A. B. Vieira, A. P. C. da Silva, and A. Ziviani. Fast centrality-driven diffusion in dynamic networks. In *WWW, Companion Volume*, pages 821–828, 2013.

[25] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 27-29 January 1992, Orlando, Florida.*, pages 165–174, 1992.

[26] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, pages 1–7, 2010.

[27] G. Karypis and et. al. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[28] D. Kempe, J. M. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *SIGKDD*, pages 137–146, 2003.

[29] J. M. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and embedding using small sets of beacons. In *(FOCS), 17-19,*, pages 444–453, 2004.

[30] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[31] Y. Low and et. al. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[32] K. Madduri and et. al. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS*, pages 1–8, 2009.

[33] A. S. Maiya and T. Y. Berger-Wolf. Online sampling of high centrality individuals in social networks. In *PAKDD Part I*, pages 91–98, 2010.

[34] G. Malewicz and et. al. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[35] R. Meusel and et. al. Graph structure in the web - revisited: a trick of the heavy tail. In *WWW*, pages 427–432, 2014.

[36] A. Mochalova and A. Nanopoulos. On the role of centrality in information diffusion in social networks. In *ECIS*, page 101, 2013.

[37] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM*, 2002.

[38] D. Oliver and et. al. Significant route discovery: A summary of results. In *Geographic Information Science*, volume 8728 of *Lecture Notes in Computer Science*, pages 284–300. Springer International Publishing, 2014.

[39] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. CIKM, pages 867–876, 2009.

[40] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[41] J. Pujara and et. al. Knowledge graph identification. In *ISWC Proceedings, Part I*, pages 542–557, 2013.

[42] Z. Qi, Y. Xiao, B. Shao, and H. Wang. Toward a distance oracle for billion-node graphs. *PVLDB*, 7(1):61–72, 2013.

[43] M. Qiao, H. Cheng, and J. X. Yu. Querying shortest path distance with bounded errors in large graphs. In *SSDBM*, pages 255–273, 2011.

[44] N. Ruan, R. Jin, and Y. Huang. Distance preserving graph simplification. In *ICDM*, pages 1200–1205, 2011.

[45] B. Shao, H. Wang, and Y. Li. The trinity graph engine. *Technical Report 161291, Microsoft Research*, 2012.

[46] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *SIGMOD*, pages 421–432, 2011.

[47] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[48] D. Yan, J. Cheng, W. Ng, and S. Liu. Finding distance-preserving subgraphs in large road networks. In *ICDE*, pages 625–636, 2013.

[49] Q. Yang and S. Lonardi. A parallel algorithm for clustering protein-protein interaction networks. In *CSB Workshops*, pages 174–177, 2005.