Data Sci. Eng. (2017) 2:2–21
DOI 10.1007/s41019-016-0023-z

Distance-Aware Selective Online Query Processing Over Large Distributed Graphs

Xiaofei Zhang[1] · **Lei Chen**[2]

Received: 8 November 2016 / Accepted: 12 December 2016 / Published online: 18 January 2017

Abstract Performing online selective queries against graphs is a challenging problem due to the unbounded nature of graph queries which leads to poor computation locality. It becomes even difficult when a graph is too large to be fit in the memory. Although there have been emerging efforts on managing large graphs in a distributed and parallel setting, e.g., Pregel, HaLoop and etc, these computing frameworks are designed from the perspective of scalability instead of the query efficiency. In this work, we present our solution methodology for online selective graph queries based on the shortest path distance semantic, which finds various applications in practice. The essential intuition is to build a distance-aware index for online distance-based query processing and to eliminate redundant graph traversal as much as possible. We discuss how the solution can be applied to two types of research problems, distance join and vertex set bonding, which are distance-based graph pattern discovery and finding the structure-wise bonding of vertices, respectively.

Keywords Graph processing · Shortest path distance · Graph partition

✉ Xiaofei Zhang
xiaofei.zhang@uwaterloo.ca

Lei Chen
leichen@cse.ust.hk

[1] University of Waterloo, Waterloo, Canada

[2] Hong Kong University of Science & Technology, Clear Water Bay, Hong Kong

1 Introduction

The tremendous size of real-world graph data raises series of challenges in efficient graph management and query processing. With the generic vertex-centric model [22] applied to practices, there has been a huge advancement in large-scale graph analytic tasks, e.g. PageRank, SCC, subgraph listing and etc. More efficient generic graph processing frameworks, like the subgraph-centric model [33, 35], are emerging to accelerate the graph analytic task. However, online graph query which finds various applications in real practice does not attract much research effort, especially the highly selective queries which has an limited output size. In this work, we study the problem of answering shortest path distance-based selective graph queries in a online fashion, such that *ad hoc* queries of such type can be answered promptly.

Graph queries using the shortest path or shortest path distance semantic are widely used in practices. For example, one popular way to define the similarity of two vertices in the network would be the similarity of their distance vector to a number of pre-selected vertices. Other examples like influential maximization, or adaptive betweenness calculation and etc., are all based on the shortest path semantic. Therefore, effective distance estimation as well as efficient shortest path retrieving are essential for online graph queries. However, one fundamental issue is the unbounded nature of graph queries which often leads to poor computation locality. For example, to find out the shortest path(s) from vertex u to v, a naive BFS would access a large number of vertices in the graph to answer the query. In an extreme case, if graph G is a social network, and the distance between u and v is five, a shortest path query evaluation much likely leads to an entire graph visit due to the six-separation law [13].

Various graph indexing techniques have been proposed to speed up the query evaluation, like the embedding technique introduced in GStore [39] for efficient SPARQL query processing, independent set-based labeling [10], the distance oracle approach [28] and etc. However, effective generic index structures for *adhoc* graph queries are space-consuming and involve a long setup time. Therefore, answering online selective graph queries with a combination of light-weight indices and fast graph exploration makes it a feasible solution with respect to both time and space efficiency.

In this work, we study both effective distance estimation and efficient graph exploration for shortest path discovery. We apply our methods to two types of online graph queries: distance join and vertex set bonding, which captures interesting graph patterns and structure-wise prominent vertices, respectively. We highlight the contribution of our work as follows:

- We propose a novel partition strategy for web-scaled graphs in the shared-nothing distributed environment to efficiently support the pairwise shortest path estimation;
- We develop a vertex filtering scheme to effectively support guided graph exploration, such that the cost on redundant vertex accessing could be significantly saved;
- We show how our technique can be applied to two types of powerful graph queries;
- We discuss our prototype implementation with extensive experiments over both real and synthetic web-scaled graphs on an in-door cluster.

The rest of this paper is organized as follows: we first formally define the problem in Sect. 2, then we introduce our partition-based distance estimation index in Sect. 3 and guided graph exploration in Sect. 4, respectively. We show how our proposed technique can be applied to accelerate the evaluation of two types of queries in Sect. 5. We report all experiments in Sect. 6 and briefly review the related works in Sect. 7, followed by Sect. 8 which concludes the paper.

2 Problem Definition

In this section, we first clarify the problem definition and notations used in this paper. For comprehensiveness, we present an overview of our solution before diving into the any technical details.

Given graph $G = \langle V, E\rangle, E \subset V \times V$, a path from vertex u to v is a sequence P of edges: $e(x_0, x_1)$,...,$e(x_{i-1}, x_i)$, where $e(x_i, x_j) \in E$ and $x_0 = u, x_i = v$. Intuitively, the shortest path from u to v, denoted as $SP(u, v)$, is the edge sequence of the minimum length. The shortest path distance, denoted as $d(u, v)$, is the length of a shortest path. Although one may pre-compute all pairwise shortest path distance by brute-force and materialize all the results, the space cost would be at least $O(n^2)$, which is prohibited in practical usage. On the contrary, we would like to find a quality guaranteed pairwise distance estimation with as less space cost as possible.

Definition 1 (*Distance estimation*) Given a pairwise distance query $Q_D(u, v)$, returning $\hat{d}(u, v)$ in $O(1)$ time using at most $O(c|V|)$ storage, having

$$\hat{d}(u, v) \leq (1 + \epsilon)d(u, v)$$

where $\epsilon \in (0, 1), c$ is a constant factor.

As shown in the definition, all pairwise shortest path distance should be estimated in constant time with quality guarantees. The fundamental challenge is to minimize the space consumption as much as possible. Clearly, if we let ϵ be 0, then the problem can only be solved by pre-compute all pairwise distances. Thus, the problem essentially asks for the space lower bound of a parameter-adjustable solution for distance estimation.

Although finding the shortest path is a well established problem, we define the optimal shortest path computing problem under an exploration semantic as follows:

Definition 2 (*Graph exploration strategy*) Given graph $G = \langle V, E\rangle$, a vertex u is explored only if $\exists e(u, v) \in E$ and v is explored.

Definition 3 (*Atomic graph exploration cost*) Given vertex $u \in V$, the atomic graph exploration cost of u is $|\{e(u, *)\}|$, where $\{e(u, *)\}$ denotes the set of edges going out from u.

Definition 4 (*Optimal shortest path computing*) Given a pairwise shortest path query $Q_{SP}(u, v)$, let $\mathcal{V}$ be a set of vertices to access to answer the query in an exploration manner, then the optimal goal is $\underset{\mathcal{V}}{\operatorname{argmin}} \sum_{|e(v,*|)|}$, where $v \in \mathcal{V}$.

Intuitively, the optimal shortest path computing is to locate the minimum set of vertices to access in order to answer the query. Unfortunately, the exploration based path computing has been proven to be untractable in terms of the number of vertex access. Thus, we shall study effective heuristics that eliminate redundant vertex access as much as possible.

The solution we introduced for these two problems are somehow correlated. We first introduce a partition-based distance estimation index, which effectively estimates $d(u, v)$ in constant time. Essentially, we partition the graph

into a set of subgraph pairs, such that vertices that are far away enough from each other would be grouped into different partitions. Thus, we can use the distance between partitions to estimate the true pairwise distance. Based on the lightening fast yet accurate (with error guarantee) distance estimation, we can perform a guided graph exploration. Moreover, we introduce the landmark-based guided graph exploration and probe-based graph exploration which requires much less space overhead.

Algorithm 1: Computing $Vor(G_{V'})$

```
Data: G, V'
Result: Vor(G_{V'})
Initialize every vertex v ∈ V/ V' to be unmarked;
while ∃v ∈ V/ V' does not report termination do
    if v is unmarked and receives a message (id,len) from an incident edge then
        Mark v with (id,len);
        v sends (id,len + 1) to all its incident edges;
        v reports termination;
    if v is unmarked and receives a set of messages (list(id),len) from incident edges simultaneously then
        Mark v with (id_min,len);
        v sends (id_min,len + 1) to all its incident edges;
        v reports termination;
    Bulk Synchronization;
for each v ∈ V/ V' do
    v sends (id) to all its incident edges;
    Bulk Synchronization;
for each v ∈ V/ V' do
    if v's id is different from the received id then
        Mark v as a boundary vertex;
```

3 Partition-Based Distance Index

Being a crucial criteria of online queries, the latency of query processing should be reduced as much as possible. To address this efficiency issue, we partition graph G based on its summary graph extracted from the Delaunay triangulation of a set of selected vertices. A *Well-Separated-Subgraph Decomposition* method is employed to guarantee a distance-aware partition. Since the partition task is beyond the capability of a single stand-alone server, we first randomly partition G to all computing nodes and compute the summary graph in a distributed manner. As the summary graph is small, a partition schema can be derived in one server. Afterward, a re-partition of G is performed among all computing nodes.

This process includes three steps, selecting a subset of vertices V' from G, building the Voronoi diagrams $Vor(G_{V'})$ and extracting a graph which is the corresponding Delaunay triangulation of V', i.e., $DT(G_{V'})$.

Step 1. Selecting V'. We adopt the betweenness approximation method proposed in a series of work [2, 4, 21] to sample V'. The cardinality of V', however, is considered as a customizable parameter in our solution. It would be great to have a larger $|V'|$, but it would be inefficient to compute the partition of $DT(\mathcal{G}_{V'})$ in the main memory. Therefore, we make $|V'|$ reasonably large as long as a $O(|V'|^2)$ matrix can be completely loaded in a server's main memory.

Step 2. Computing Vor $(G_{V'})$ The pseudo-code given in Algorithm 1 illustrate the computation of *Vor* $(G_{V'})$. Notice that there could be many different ways to handle conflicts, i.e., a vertex can be of equal distance to different reference points. In our solution, we specifically assign a conflicting vertex to the reference point with a smaller *id* value.

Step 3. Constructing $DT(G_{V'})$. $DT(G_{V'})$ can be easily constructed after *Vor* $(G_{V'})$ is obtained. $DT(G_{V'})$ is a weighted undirected graph. There is an edge between $u, v \in V'$ *iff* u and v reside in two adjacent Voronoi cells in *Vor* $(G_{V'})$. And the weight of this edge is $diam(u) + diam(v)$, where $diam(\cdot)$ denotes the graph diameter.

3.1 c-WSSD Partition

To partition a graph in a distance-aware fashion, we actually employ a two-layer hierarchical partition method. The bottom layer is constructed with the identification of numbers of distance-preserving Voronoi cells. The top layer is to partition the Delaunay graph $DT(G_{V'})$, which is a summary of the bottom layer. Different from most existing graph partition techniques that aim at reducing cross partition cuts, we would like to have any pairwise distance query be evaluated as quickly as possible. The

intuitions are simple: 1) the vertices that are far away from each other should be partitioned into different subgraphs; 2) the vertices that are close to each other should be located in the same or adjacent subgraphs. For clear illustration purpose, we first introduce the concept of *Well-Separated-Subgraph Decomposition*, which serves as a partition constraint for $DT(G_{V'})$.

Definition 5 (*c-WSSD*) Let S be a connected graph of n vertices. Let $\mathcal{F} = \{(A, B) : A \subseteq S, B \subseteq S\}$ be a collection of pairs of subgraphs of S. For any constant $c \geq 1$, we call $\mathcal{F}$ a c-Well Separated Subgraph Decomposition, *c*-WSSD in short, if the following conditions are satisfied:

1. For any $x, y \in S$, there exists a unique pair $(A, B) \in \mathcal{F}$ such that $x \in A$ and $y \in B$.
2. $d(A, B) \geq c \times Max\{diam(A), diam(B)\}$.

In the definition, $diam(A)$ refers to the diameter of subgraph A; $d(A, B)$ denotes the shortest path distance between two subgraphs A and B. We reason the *c*-WSSD graph partition from two aspects. First, such a definition is query oriented. As stated in the first condition, given any two vertices from $DT(G_{V'})$, the distance constraint can be examined on just one machine that holds the corresponding pair of graph partitions. Second, it restricts the distance between partitions to ensure a straightforward query constraint verification. The second condition given in *c*-WSSD is to guarantee that two clusters of vertices from $\mathcal{G}$ are far apart from each other. As we elaborate later in this section, a *c*-WSSD graph partition ensures an error-bounded immediate pairwise distance estimation.

We solve the *c*-WSSD construction problem by employing a comparison-based binary tree traversing procedure presented in Algorithm 2. To be specific, let $R(\cdot)$ denote the minimal spanning tree structure of a given subgraph; u, v denote subgraphs of $DT(G_{V'})$; l_u and l_v denote the diameter value of $R(u)$ and $R(v)$, respectively. To elaborate, we first compute $R(DT(G_{V'}))$ and assign it as the root of a binary tree T. T is constructed as follows. For each non-leaf internal (or root) node $u \subseteq V'$, we split it into two children by removing the edge on which the center of $R(u)$ resides. Binary tree T is built along with the subgraph pair extraction process. An internal node u splits only if certain conditions hold, as shown in the *if...then* clauses of Algorithm 2.

Algorithm 2: Constructing c-WSSD of $DT(G_{V'})$

```
Data: DT(G_V'), queue Q, binary tree T
Result: F
1  F ← ∅;
2  T_root ← R(DT(G_V'));
3  Add pair (T_root, T_root) to Q;
4  while Q ≠ ∅ do
5      Extract (u, v) from Q;
6      if d(R(u), R(v)) ≥ c × Max{l_v, l_u} then
7          Add pair (u, v) to F;
8      if l_u ≥ l_v then
9          Equal split u into u' and u'' ;
10         Add pairs (u', v) and (u'', v) to Q if they are not in Q;
11     if l_v ≥ l_u then
12         Equal split v into v' and v'' ;
13         Add pairs (u, v') and (u, v'') to Q if they are not in Q;
14     Report F;
```

Essentially, Algorithm 2 constructs the *c*-WSSD by traversing T with the help of a queue Q. We first initialize Q to store the ordered pair $(T_{\text{root}}, T_{\text{root}})$, where $T_{\text{root}} = R(G_{V'})$. Then we incrementally grow T in a BFS-traversal manner, during which process we compare and discover qualified ordered subgraph pairs and have them be stored in $\mathcal{F}$. We shall first prove the correctness of Algorithm 2 and then explain its running time complexity.

Lemma 1 *$\mathcal{F}$ constructed with Algorithm 2 is c-WSSD.*

Proof Take any two vertices $x, y \in DT(G_{V'})$, since Q contains $(T_{\text{root}}, T_{\text{root}})$ initially, the traversal algorithm must eventually put a pair of nodes (u, v) into Q such that $R(u) \cap R(v) = \emptyset, x \in u$, and $y \in v$. Afterward, the traversal algorithm expends upon (u, v) and finally put a pair (u', v') into $\mathcal{F}$ such that $x \in R(u')$ and $y \in R(v')$. Notice that (u', v') is inserted into Q exactly once in the algorithm. This guarantees that x and y are not covered by another pair in $\mathcal{F}$. Thus, $\mathcal{F}$ satisfies the first constraint of *c*-WSSD's definition. Moreover, it is clear that if $(u, v) \in \mathcal{F}, d(R(u), R(v)) \geq c \times Max\{diam(R(u)), diam\ (R(v))\}$. Thus, $\mathcal{F}$ satisfies the second constraint given in the definition. □

Lemma 2 *The cardinality of $\mathcal{F}$, i.e. $|\mathcal{F}|$, is $O(|V'|)$.*

Proof It suffices to prove that given any internal node u, there are at most $O(1)$ nodes v' such that (u, v') appears in Q. Suppose that (u, v') appears in Q. It holds if $(u, v')=(r, r)$. Otherwise, (u, v') was put into Q because we split the parent of u or the parent of v'. Without loss of generality, assume that we split the parent v of v'. Thus, $l_v \geq l_u$ and $d(R(u), R(v)) < c \times Max\{l_u, l_v\}$. In other words, $R(v)$ lies inside a spanning tree R with the same center of $R(u)$ and length equal to $4c \times l_v$. Then there are no more than $16c^2$ disjoint binary tree segments of width l_v inside R, each of which may generate two children that appear with u in Q. Hence, u appears with at most $32c^2$ nodes in Q. □

Since each pair in $\mathcal{F}$ must appear in Q too, $|\mathcal{F}| = O(n)$. It takes $O(n \log n)$ time to construct $\mathcal{T}(V')$. Then the traversal algorithm runs in $O(m \log m)$, where m is the total number of pairs that appear in Q. Lemma 2 has already shown that $m = O(n)$. Therefore, Algorithm 2 constructs $\mathcal{F}$ in $O(n \log n)$ time ($n = |V'|$). To achieve the $(1 + \epsilon)$ approximation of the pairwise shortest path distance, given $0 < \epsilon < 1$, we make $\mathcal{F}$ an error-bounded partition of $DT(G_{V'})$ by making $c = \frac{2(1+\epsilon)}{\epsilon}$. Then we can have the following result:

Lemma 3 *Given any two vertices $x, y \in DT(G_{V'})$, the shortest path distance approximation between x, y is at most $(1 + \epsilon)d(x, y)$.*

Proof Let (u, v) be the pair in $\mathcal{F}$ such that $x \in u$ and $y \in v$. Let path $ab, a \in u$ and $b \in v$, be the shortest path between u and v. Thus, $d(x, y) \geq d(R(u), R(v)) \geq \frac{2(1+\epsilon)}{\epsilon} \times Max\{l_u, l_v\} \geq \frac{2(1+\epsilon)}{\epsilon} \times Max\{d(a, x), d(b, y)\}$. One can inductively assume that a path connecting a and x in u whose length is at most $(1 + \epsilon)d(a, x) \leq \frac{\epsilon}{2} d(x, y)$. The same inductive assumption holds for b and y in v. Thus, the path distance from x to y is at most

$$\begin{aligned} & d(a,b) + (1+\epsilon)d(a,x) + (1+\epsilon)d(b,y) \\ & \leq d(x,y) + (2+\epsilon)d(a,x) + (2+\epsilon)d(b,y) \\ & \leq d(x,y) + \epsilon d(x,y). \end{aligned} \tag{1}$$

□

It is worth pointing out that the partition strategy illustrated above also significantly reduces the computation overhead across different storage nodes. Although the c-WSSD computation is sequential and handled centrally, the data re-partition can be easily conducted in parallel.

4 Guided Graph Exploration

In addition to fast and accurate pairwise distance estimation, finding the exact pairwise shortest path efficiently, in terms of minimizing the redundant edge visit, raises a grand challenge as well. In this section, we study a landmark-based guided graph exploration strategy.

4.1 Landmark Selection

Although the technique presented in Sect. 3 gives error-bounded distance estimation in $O(1)$ time, the storage cost of subgraph pair index depends on the underlying graph topology structure. If a graph is extremely dense, the constant factor c could be over 100 which makes it an infeasible solution. Therefore, we consider a more generic strategy to estimate pairwise distance, which is sufficient to perform guided graph exploration.

Selecting landmarks or reference points to facilitate the shortest path distance computation has been adopted in many works [26, 28, 29]. 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 $|p_{st}|$[1] 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, $O = \{o_1, o_2, \ldots, o_d\}$, 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 3 gives a deterministic solution of finding the minimal d, which also helps decide the selection of landmark vertices.

In the first line of Algorithm 3, 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 3 is $O(h|G|)$, because on

[1] For the rest of this paper, we use $|p_{st}|$ and $d(s, t)$ interchangeably, they both denote the shortest path distance from s to t.

each level of recursion the entire graph is traversed. We can save the computation cost using a random algorithm given in Algorithm 4. It is worth pointing out that Algorithm 4 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 4 *Algorithm 4 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 3. However, according to Algorithm 4, 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 3. Therefore, Algorithm 4 reports at most $\frac{3}{2} \times 2^h = 3 \times 2^{h-1}$ landmarks.

With the set of landmarks O determined, we are able to associate each vertex a label vector denoting its distance to all landmarks. Let $l(v)$ be 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.

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 $|p_{uo'}| + |p_{o'o}| - \sigma$, 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.

Algorithm 3: 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 4: 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;
```

4.2 Guided Graph Exploration

To explore the shortest path from s to t, at least $|p_{st}|$ super steps are necessary using a vertex-centric model. 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 whether 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 $1 \le i \le d$. 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 5. 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 5 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.

Apparently, Algorithm 5 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 5 *Given vertices s and t, a vertex v possibly resides on p_{st} if $Max\{|l(v)_i - l(s)_i| + |l(v)_i - l(t)_i|\} \le UB(|p_{st}|)$, where $1 \le i \le d$.*

Proof Let vertices u and v be directly adjacent to each other. Then $Max\{|l(v)_i - l(u)_i|\} = 1$, where $1 \le i \le d$, 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 5 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 6 *Given s and t, a vertex v possibly resides on p_{st} if for $p_{st} \in [LB(p_{st}), UB(p_{st})]$ and $1 \le i \le d$, assuming $l(s)_i \le l(t)_i$, then*

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

Algorithm 5: 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|;
```

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 6 can be easily proved following the cosine law and the Heron's formula. By applying the filtering criteria suggested in Lemmas 5 and 6, 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 6.

Algorithm 6 employs a range label to check whether 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.

Correctness. There are only two cases where v does not reside on p_{st}. One is that v reaches both s and t from a same vertex u. In this case, according to Algorithm 6, 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}. 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.

Complexity. Obviously, Algorithm 6 takes the space complexity of up to $O(|g_{st}|)$, and the total iteration step of Algorithm 6 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 6 reduces the communication cost at each superstep. While comparing with Algorithm 5, Algorithm 6's total number of iteration steps is fixed. It makes Algorithm 6 more generic for all possible workloads.

Note that it is trivial to add a global counter in Algorithm 6 to record each vertex's shortest path distance to s and t. Then the exact $|p_{st}|$ can be obtained after the program execution. The difference between Algorithm 5 and 6 is that the former aims at fast validation of $|p_{st}|$ with as least vertex access as possible under the help of vertex labeling. Algorithm 6 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.

Algorithm 6: 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;
```

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.

5 Apply to Online Graph Queries

The distance estimation and guided graph exploration technique elaborated above can serve as fundamental building blocks for various online graph queries. In this section, we present two different types of selective online queries that can benefit from the proposed technique.

5.1 Distance Join Query

Given a query graph Q and the data graph G, a distance join query returns all the subgraphs from G that satisfy every pairwise distance constraint in Q. Such a kind of query is handy and expressive in social network analysis and Biochemical network investigation [8, 14, 38]. It captures not only the structure information about the query graph, but also implies strong connectivity constraints, i.e. the pairwise distance between any two given vertices. Clearly, a distance join query is more flexible than the subgraph search and especially useful in graph analytic tasks that target on co-relationship discovery. For comprehensiveness, we first define the *distance join query* as follows:

Definition 7 (*distance join*) Given a query graph Q of n vertices $\{v_1, \ldots, v_n\}$ and m edges of weights $\{w(v_i, v_j) | (v_i, v_j) \in Q, 1 \leq i \neq j \leq n\}$, let S denote a set of n vertices selected from a data graph G, we define S is a distance match of Q **iff** the following bijective function f holds:

1. $\forall v_{i,1 \leq i \leq n}, f(v_i) \in S$;
2. $\hat{d}(f(v_i), f(v_j)) \leq w(v_i, v_j)$, if $(v_i, v_j) \in Q (1 \leq i \neq j \leq n)$, where function $\hat{d}(\cdot, \cdot)$ denotes the pairwise shortest path distance of two vertices in $\mathcal{G}$.

Then the distance join of Q and G, denoted as $DJ(Q, G)$, is to find all the distance matches of Q in G.

According to the definition, function f can be defined as any bijective mapping, e.g. label matching, similarity matching and etc. And $\hat{d}(\cdot, \cdot)$ can be any distance metric

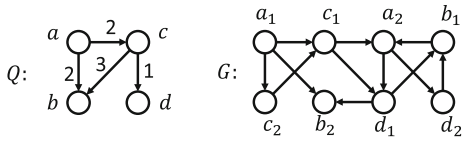

Fig. 1 An example of $DJ(Q, G)$

depending on the application scenario. In this work, we adopt the exact label matching as function f, and the shortest path distance as the distance function $\hat{d}(\cdot, \cdot)$, which satisfies the triangle inequality. An example of $DJ(Q, G)$ is given in Fig. 1. In the example, the weight of each edge in G is one. Based on the query, a vertex c must be adjacent to a vertex d, which makes c_1 and d_1 the only option. Considering $\hat{d}(a_1, b_1) > 2$ and a_2 does not reach c_1, $\{a_1, b_2, c_1, d_1\}$ is the only valid result.

Note that $DJ(Q, G)$ allows different pairwise distance constraints, which makes the query introduced in [38] a special case of our study (as the query in [38] restricts all pairwise distance constraints to a same value Δ). Such a generalized query semantic implies at least the same computational complexity as the query defined in [38], which is reported to be $\#\mathcal{P}$-complete.

5.1.1 Evaluation Overview

Intuitively, it is the join order selection problem to generate a good query plan for $DJ(Q, G)$. First of all, we need a cost model to evaluate different query plans. There have been many literatures studying the cost metric of a distributed jobs *w.r.t* various constraints. We omit the details on cost model construction as it is beyond the scope of this paper. For the rest of this paper, we use $C^*(\cdot)$ to denote the cost function.[2]

Given the cost-driven query evaluation plan, which is a sequence of subqueries to evaluate, the join condition validation falls into two categories: (1) validate two vertices that do not reside on the same machine; (2) validate two vertices that are co-located on the same machine. As elaborated in Sect. 3, the distance-aware graph partition particularly favors the distance approximation of vertices that are far away from each other. Thus, it only takes constant time to justify a join condition for the case (1). However, for the query inputs of two vertices that are *not far away* from each other, i.e., they fall into the same set of Voronoi cells after partition, we need further computation on each computing node to answer the query. The essential challenge is to minimize the I/O operation as much as

[2] The cost model we employed is elaborated in [36]. As a matter of fact, any off-shelf cost models can be applied.

possible such that queries can be answered more efficiently.

5.1.2 Data Block Construction

We discuss how we organize graph data locally according to the *c-WSSD* property. Let $F_i \subset \mathcal{F}$ be the set of subgraph partitions distributed to a computing node i. As it is infeasible to keep F_i completely in the main memory, F_i must be written back to the file system in a certain manner. Since we employ HDFS as the underlying file system, challenge rises because HDFS, like other Cloud file systems, is managed on the block basis. First, the data loading from disk is at the level of blocks. Second, HDFS demonstrates unsatisfactory performance in random block access. Therefore, it is not a trivial task to write F_i back to HDFS. A data block needs to be carefully constructed.

Since $F_i = \{(A, B), ...\}$ is a set of subgraph pairs, where both A and B are sets of Voronoi cells. Therefore, the storage structure should be designed on the basis of Voronoi cells. Without loss of generality, we consider the storage of subgraph A from $(A, B) \in F_i$. Let A compose r Voronoi cells, i.e., $A = \{vc_1, vc_2, \ldots, vc_r\}$. Assume A needs s disk blocks. It matters the way to assign r Voronoi cells to s data blocks. Because if two query vertices fall into A, intuitively we would not want to load all s data blocks to explore A for the answer. Our solution is as follows. We first compute the group betweenness of each Voronoi cell, and select the Voronoi cells of high betweenness than all adjacent neighbors, which are named as *peaks*. Starting from these *peaks*, we expand the region of each *peak* by progressively including its adjacent Voronoi cells to form a *mountain*. After all the Voronoi cells are covered by this process, we derive a partition of this subgraph. Each *mountain* corresponds to one or more consecutive data blocks. Like the visual example shown in Fig. 2, we simplify each Voronoi cell as a square, and the number within each cell represents its group betweenness in the subgraph. We consider a 6×8 matrix of Voronoi cells. Clearly, there are some *peak* cells (marked with circles) that have high group betweenness, surrounded by Voronoi cells of relatively low group betweenness, which visually forms three *mountains* in Fig. 2a.[3] Given any two query points p and q, intuitively, the shortest path between them are most likely to reside on a path passing through some of the peaks, as shown in the figure. Therefore, if we group the Voronoi cells according to the *mountain* areas, we can achieve more efficient disk I/O on average.

Moreover, we introduce redundancies for cell clustering in our implementation, as the three large overlapping rectangles shown in Fig. 2(b), such that it reduces the probability of involving more groups of Voronoi cells for the query evaluation. Although the factor of redundancy itself can be a research topic if more data statistics and query patterns are presented, in our prototype system, we only include the two-hop neighbor cells. For each group of Voronoi cells, we write them back to the disk on the basis of Voronoi cells. For each cell, it is organized as the (*key*, *value*) data model, although we need to add special links from *values* to *keys*, such that when it is loaded in the main memory, it serves as an auxiliary index structure for shortest path computing.

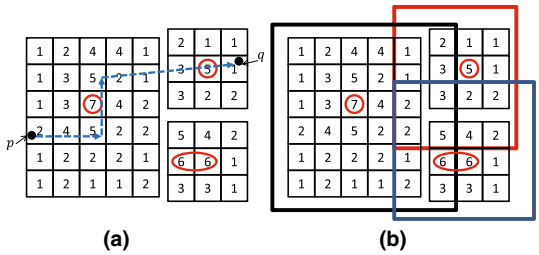

Fig. 2 An example of Voronoi cell grouping

As elaborated above, we now have the power to validate the distance join conditions effectively for both cases: (1) vertices that are far away from each other; (2) vertices that are close by each other. Although to generate the *optimal* query plan remains an open question, fast and accurate distance estimation always serves as a fundamental building block.

5.2 Vertex Set Bonding

Vertex Set Bonding query (VSB query for short) 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. Such type of query finds various applications in practice, for example, network monitoring [7], community bonding [5, 13] and etc. In this section, we formally define the VSB query and then show how the query evaluation can benefit from our guided graph exploration. For comprehensiveness, we first introduce the vertex-path and vertex–vertex dominance concept.

Definition 8 (*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.

[3] Figure 2 is to provide a visual aid, and the boundaries are not specifically defined.

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.

Definition 9 *(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 10 *(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 11 *(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.

Definition 12 *(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.

Definition 13 *(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, where

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

In the above definition, $\sigma_{xy}(v)$ denotes the number of shortest paths between x and y that pass through v, σ_{xy} denotes the total number of shortest paths between x and y.

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.

Our general solution framework works as follows. 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[4]. To show how guided graph exploration helps with the query evaluation, we highlight two building blocks: path sharing and probe-based communication.

5.2.1 Naive Plan Versus Path Sharing

A naive query plan is to 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 7. 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 7: 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;
```

[4] As a matter of fact, we can perform approximated betweenness ranking on exploration as presented in work [37].

Theorem 1 *Algorithm 7 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 7 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 7 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.

In contrast to naively compute the pairwise shortest path between two sets of input vertices X and Y, an optimization opportunity lies in taking the advantage of vertex distribution in X and Y. As the VSB query can be applied to find the bonding between communities, where a community must be composed of vertices that are close to each other. It implies the potential of shortest path sharing property. Thus, 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 discussed in Sect. 4), 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 7 *Given a set of vertices X', a virtual vertex x_v of label $l(x_v)_i$, having*

$$l(x_v)_i = \left\lceil \frac{Max\{l(x')_i\} - Min\{l(x')_i\}}{2} \right\rceil$$

where $x' \in X'$, guarantees $\forall x' \in X'$ s.t.: 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_vy} must passes x', implying $p_{x'y} \subset p_{x_vy}$. Similarly, the case when $|p_{x'y}| > |p_{x_vy}|$ can be easily verified. □

By employing the path sharing, we could greatly save the concurrent exploration cost of Algorithm 7, as well as the space and communication cost on D_{xy}, since the total number of such dominant vertex set are reduced.

5.2.2 Probe-Based Communication

A main bottleneck of Algorithm 7 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. Let each graph vertex be associated with a set of independent hash functions, denoted by H. 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 7), 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 whether 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 7) could be reduced. Following the same context of Algorithm 7, we show how the probe-based communication is employed to evaluate a VSB query in Algorithm 8.

Algorithm 8: 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 8, 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 7, 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 8 achieves the same approximation ratio on $AB(R)$ as Algorithm 7, 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.

In this section we show how to apply the distance estimation and guided graph exploration to two types of graph queries. As a matter of fact, there are other technical challenges in evaluating such queries. For example, adaptive query plan generation, as well as taking the query structure feature into consideration would yield better performance of distance join processing. As VSB query use the betweenness semantic for bonding vertices, online betweenness approximation and ranking technique would greatly contribute to efficient query processing. However, the two techniques discussed in Sects. 3 and 4 are orthogonal research problems and applicable to any distance-aware graph queries.

6 Experiments

We report two sets of experiments in this section: 1) testing the effectiveness of c-WSSD partition method on reducing the computation cost of pairwise join validation for distance join queries; 2) how guided graph exploration accelerates shortest path computing and the VSB query evaluation.

6.1 Setup

6.1.1 c-WSSD Partition for Distance Join

Testbed We built up the test bed on a cluster of 16 servers. Every server has 4 Intel(R) Xeon(R) CPU E5-2650 of 2.0GHz, each of which has two cores and supports 16 threads, 12 GB memory and 1 TB hard disk storage. The running operating system is 2.6.35-22-server #35-Ubuntu SMP.

Datasets Brief statistics of the four employed data sets are summarized in Table 1. In the table $d_{\max}$ denotes the diameter of a graph. Data set A is the US patents data. Data set B is the web graph of the TREC 2009 Category B data set, which is the set of the first 50 million English pages

Table 1 Graph data used for *c*-WSSD partition and distance join

ID	# of nodes	# of edges	# of labels	$d_{\max}$
A	3,774,768	16,522,438	481	22
B	428,136,613	454,075,638	1,325	78
C	500,000,000	4,287,029,468	1,000,000	1,052
D	10,000,000,000	23,946,452,156	1,000,000	1,927

Table 2 General statistics of employed graph data sets

ID	No. of nodes (million)	No. of edges (million)	$d_{\max}$	Size (GB)
A	~428	~454	78	3.6
B	~1,825	~65,219	5328	869.2
C	~33	~1108	7	25.6
D	10,000	23,946	2,927	42.2

collected in January and February 2009 by the Language Technologies Institute at CMU. Synthetic data sets C and D are random graphs generated with the *igraph*[5] package.

Query Workload Given a data set, we randomly select $10 \sim 20$ labels and generate three types of query graphs: star-shaped, path-shaped and circled graph. Meanwhile, we randomly assign the pairwise distance constraints. We generate 300 distance queries for each data set and evaluate the batch one by one. We run every job batch with 3 cold-start and report the average execution time.

6.1.2 Guided Graph Exploration for VSB Queries

Testbed 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 [19] 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).

Datasets We employ four data sets of different scales and topologies in the experiments, as briefly summarized in Table 2. Data set A describes the web graph of the TREC 2009 Category B data set. Data set B comes from the WebGraph 2012 project [23], 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 [18]. 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 *igraph*.

Query Workload For each VSB query, we randomly select $10 \sim 100$ 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.

6.2 c-WSSD Partition

c-WSSD partition method provides a distance-aware partition of a large graph, which makes it possible to estimate the pairwise shortest path distance in constant time. In the experiments we study from two aspects: (1) the effectiveness of *c*-WSSD method in terms of query evaluation time cost as well as the I/O and network cost; (2) the scalability of *c*-WSSD method under different scales of data and parameter.

Effectiveness To validate the effectiveness of *c*-WSSD, we randomly generate 100 pairwise shortest path distance queries for each data set. In Fig. 3, we report the experiment results on the largest real and synthetic data set B and D, respectively. We measure the query evaluation cost in terms of disk I/O (swap) volume, network volume and the time efficiency. We employed two other intuitive graph partition strategies for comparison: random partition and the *k*-minimal cut partition using **METIS** [16]. Note that on each individual computing node, we set up the same data block layout and in-memory index structure. Therefore, only the graph data distribution matters in this experiment.

As shown in Fig. 3a, *c*-WSSD and random partition have about the same cost, while *k*-minimal cut can introduce high I/O cost. The rationale behind is that *k*-minimal cut tends to group large number of connected vertices in one partition. Therefore, the queried two vertices are very much likely fall into the same storage node where graph exploration method needs to be adopted. It is not surprising that, as shown in Fig. 3b, *k*-minimal cut greatly saves the network traffic. However, *c*-WSSD is the winner of query evaluation time, as shown in Fig. 3c. Because it does not

[5] http://igraph.sourceforge.net/index.html.

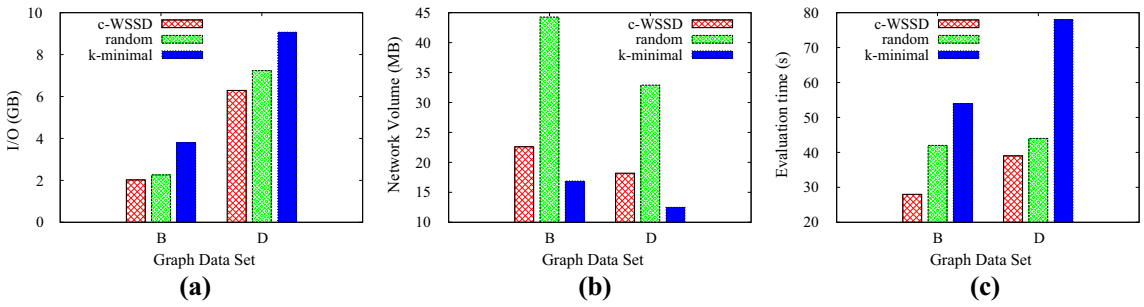

Fig. 3 Pairwise shortest path distance computation: *c*-WSSD versus random versus *k*-minimal cut. **a** I/O cost. **b** Network cost. **c** Time cost

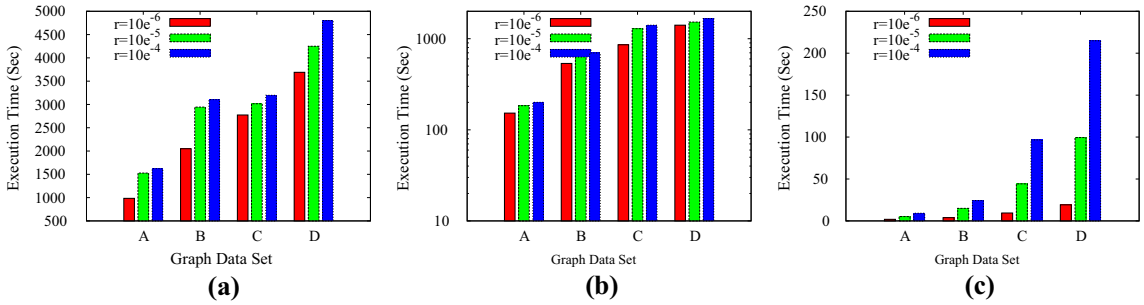

Fig. 4 Scalability test: preprocessing cost of the Voronoi diagram construction and the partition computation. **a** Cost of V' selection. **b** Cost of Voronoi diagram computation. **c** Cost of computing $\mathcal{F}$

introduce much network traffic or I/O cost during the query evaluation. The extra cost of *c*-WSSD is paid during the preprocessing stage of graph partition.

Scalability As introduced in Sect. 3, the preprocessing steps include selecting the initial vertices of high global betweenness, computing the Voronoi diagram, and re-partition the graph which involves large volumes of data copying over the network. As discussed before, the cardinality of initial vertex set serves as a trade-off point between query efficiency and system complexity. Therefore, we conduct experiments based on different sizes of initial vertex sets to demonstrate its affection on the final solution.

Figure 4a gives the time cost evaluation on four data sets with respect to the initial vertices selection based on different selection ratios, where $r = \frac{|V'|}{|V|}$ denotes the percentage of employed vertices to partition the graph. We have two main observations. First, for a given r, along with the size of a graph grows, the selection cost increases dramatically. Second, given a data set, when r increases in the order of magnitude, the selection cost also increases, however, following a Logarithm level growth.

Figure 4b demonstrates the time cost of computing Voronoi diagram with Algorithm 1 given in Sect. 3. Given a data set, when r increases in the order of magnitude, the time cost to compute the Voronoi diagram does not grow in the same pace. Since the computing process is essentially in the BSP (*Bulk Synchronous Processing*) style; therefore, more initial vertices actually help to explore the entire graph faster. However, extra cost to maintain the boundary vertices cannot be neglected.

The time cost to compute the partition set $\mathcal{F}$ is presented in Fig. 4c. The results are obtained when ϵ is set to 0.05, similar trends of results are observed when $\epsilon = 0.01$ and $\epsilon = 0.1$. As proved in Sect. 3, the cardinality of final $\mathcal{F}$ is only subjected to the size of initial vertex set V'. We observe the same trend in the experiment that the time cost to compute $\mathcal{F}$ is closely related to the selection of r.

6.3 Guided Graph Exploration for VSB Query

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.

Preprocessing As presented in Sect. 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 3, we report the

Table 3 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	2231	3029	549	227	4216	4248
	16	1429	2574	531	189	4094	4225
	32	879	1782	492	141	2709	2799
C	1	126	145	329	124	1139	1178
	2	10	26	69.2	36.4	95.6	105.4
	4	3	59	2.3	78.5	78.9	131.7
D	8	1576	2109	421	179	1465	1509
	16	1206	2005	392	164	1437	1486
	32	457	1324	354	139	1128	1305

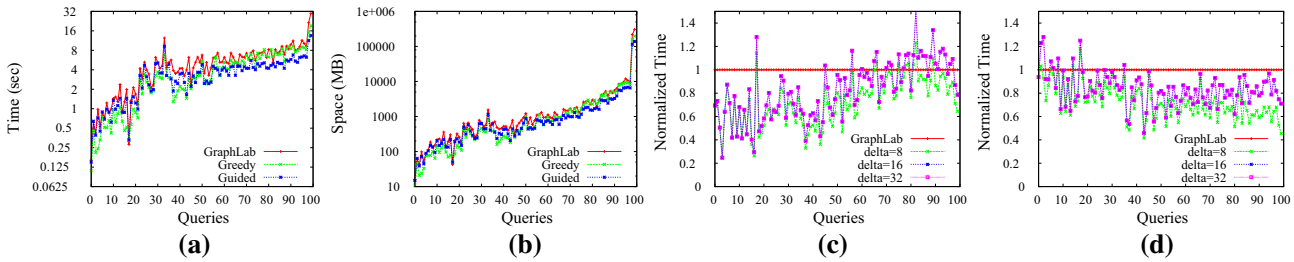

Fig. 5 The speed up of evaluating p_{st} queries. **a** Time ($\delta = 8$). **b** Space ($\delta = 8$). **c** (T) Greedy versus GraphLab. **d** (T) Guided versus GraphLab

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 refvsb:datasets are the ones with vertex label compression applied, as elaborated in Sect. 3.1.2. A straightforward 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.

Fast shortest path computing To validate the guided graph exploration for shortest path, 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 [2]. Due to the space limit, we highlight our findings on graph B.

Figure 5 shows how our methods, greedy (Algorithm 5) and guided exploration (Algorithm 6), compare to the GraphLab's shortest path in p_{st} evaluation. Figure 5a, b shows the time and space cost respectively. Space cost is the total size of data access on the distributed storage. Note

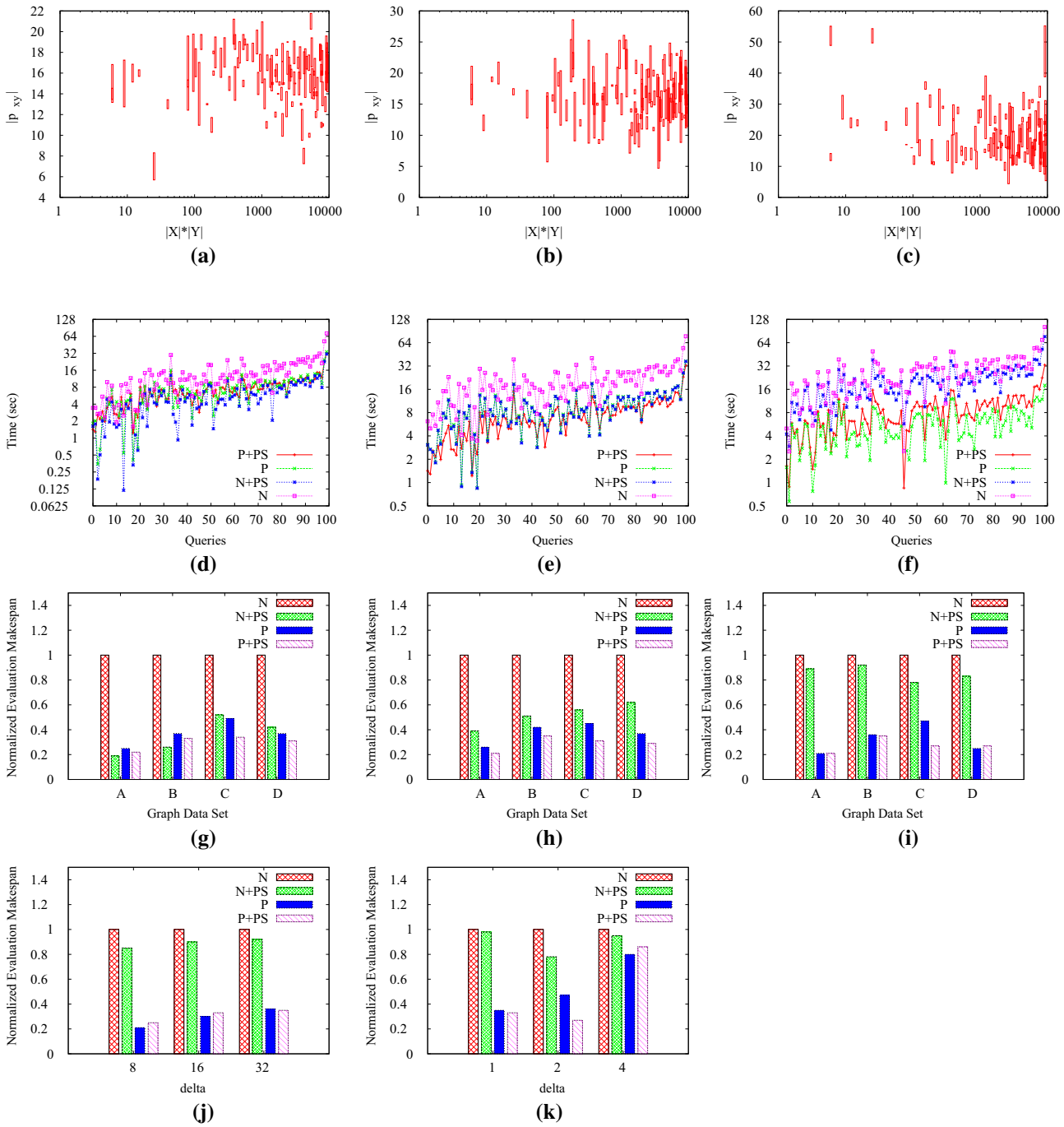

Fig. 6 **a–i** Different query workloads test; **j–k** evaluation speedup using different δ (Graph B & C). **a** XLYL query distribution. **b** XLYR query distribution. **c** XRYR query distribution. **d** XLYL query time cost. **e** XLYR query time cost. **f** XRYR query time cost. **g** Q of XLYL workload. **h** Q of XLYR workload. **i** Q of XRYR workload. **j** δ & efficiency (Graph B). **k** δ & efficiency (Graph C)

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 Fig. 5a, 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 5 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 Fig. 5c, 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 before making a decision on the next hop. Clearly, as shown in Fig. 5d, when $|p_{st}|$ is large, the guided exploration is more sensitive to the setting of δ. Although smaller δ works better for the path query, there is the greater extra space overhead to trade off.

VSB query evaluation We evaluate our proposed solution from the efficiency 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. In Sect. 5.2, we introduce a naive VSB query processing solution (Algorithm 7) 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 Fig. 6. Figure 6a–c show the distribution of random queries we generated, where queries are sorted according to their input size ($|X| \times |Y|$ as x axis). Figure 6d–f shows 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 8), PS stands for path sharing. Apparently, if input vertices are close to each other, path sharing would achieve great time saving, as shown in Fig. 6d. On the contrary, when query inputs are randomly selected, as shown in Fig. 6f, probe-based method performs better.

We report the normalized query processing makespan of different methods on all data sets in Fig. 6g–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 Fig. 6g, 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 8, 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 Fig. 5d. We highlight our findings using the results from graph B and C. For graph B, as shown in Fig. 6j, 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 Fig. 6k, 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.

7 Related Work

Distributed graph processing models and systems General purpose large graph management has drawn great research interest. Early work [20] illustrates the challenging issues of large graph management. Proposals in [3] and [22] are two well recognized models for parallel large graph processing, which are MPI(*Message Passing Interface*)-based and BSP, respectively. Although MPI usually gains more time efficiency, it is relatively complicated and puts a heavy burden of system implementation on programmers. Pregel [22] is a vertex-centric computing model, which is more flexible and relatively easy to program. However, it has to sacrifice time efficiency due to inevitable synchronization costs at each iteration step. Work [24] conducts an empirical comparison of three computing diagrams for large graph processing, which are RDBMS-like

approaches, data parallel approach (e.g. Pregel) and in-memory graph exploration approach. Improvement works over Pregel, like Pregelix [6], Blogel [35] targets on network cost reduction to yield better performance. Trinity [31] and GBase [15] are two other state-of-art distributed general purpose graph management systems with substantially different designs. GBase models graphs using adjacent matrices. It transforms nearly all the graph analytic functions into matrix manipulations using iterative MapReduce jobs. On the contrary, Trinity employs an in-memory *key-value* store in a distributed shared memory environment. It models graph data following the vertex-centric model, i.e., each vertex is associated with its one-hop neighbor(s), such that all the classical graph exploring algorithms can be directly plugged in.

Distance-based query over distributed graph Employing landmarks to approximate the shortest path distance is a widely adopted technique [17, 25, 27]. 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 [27] 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 [9]. 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. [32] 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. Follow-up works, like graph simplification [30], shortest path discovery over road network [11, 34], 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 [12]. Several special graph families, including trees or graphs with bounded tree-width, have distance labeling schemes with $O(log_2 n)$ bit labels [1]. 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.

8 Conclusion

In this paper, we study two fundamental building blocks for distance-aware online graph query: fast and accurate distance estimation, as well as guided graph exploration. A c-WSSD partition method is introduced to generate the index structure to produce error-bounded shortest path distance estimation in $O(1)$ time with space complexity of $O(c|V|)$, where c is a constant factor. Furthermore, we discuss how to perform guided graph exploration with landmark referencing. We validate the proposed technique with distance join and VSB query workload over both real and synthetic graph data in real Cloud environment.

References

1. Alstrup S, Bille P, Rauhe T (2005) Labeling schemes for small distances in trees. SIAM J Discrete Math 19(2):448–462
2. Bader DA, Madduri K (2006) Parallel algorithms for evaluating centrality indices in real-world networks. In: ICPP, pp 539–550
3. Bader DA, Madduri K (2008) Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In: IPDPS, pp 1–12
4. Bader DA, Kintali S, Madduri K, Mihail M (2007) Approximating betweenness centrality. In: WAW, pp 124–137
5. Brandtzæg PB, Heim J, Kaare BH (2010) Bridging and bonding in social network sites—investigating family-based capital. IJWBC 6(3):231–253
6. Bu Y, Borkar VR, Jia J, Carey MJ, Condie T (2014) Pregelix: big(ger) graph analytics on a dataflow engine. PVLDB 8(2):161–172
7. Castro M et al (2003) Future directions in distributed computing. In: Topology-aware routing in structured peer-to-peer overlay networks, pp 103–107
8. Cheng J, Yu JX, Yu PS (2011) Graph pattern matching: a join/semijoin approach. IEEE Trans Knowl Data Eng 23(7):1006–1021

9. Cohen E, Halperin E, Kaplan H, Zwick U (2003) Reachability and distance queries via 2-hop labels. SIAM J Comput 32(5):1338–1355
10. Fu AW, Wu H, Cheng J, Wong RC (2013) IS-LABEL: an independent-set based labeling scheme for point-to-point distance querying. PVLDB 6(6):457–468
11. Funke S, Nusser A, Storandt S (2014) On k-path covers and their applications. PVLDB 7(10):893–902
12. Gavoille C, Peleg D, Pérennes S, Raz R (2004) Distance labeling in graphs. J Algorithms 53(1):85–112
13. Guille A, Hacid H, Favre C, Zighed DA (2013) Information diffusion in online social networks: a survey. SIGMOD Rec 42(2):17–28
14. Jin W, Yang J (2011) A flexible graph pattern matching framework via indexing. In: SSDBM, pp 293–311
15. Kang U, Tong H, Sun J, Lin CY, Faloutsos C (2011) Gbase: a scalable and general graph management system. In: KDD, pp 1091–1099
16. Karypis G et al (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
17. Kleinberg JM, Slivkins A, Wexler T (2004) Triangulation and embedding using small sets of beacons. In: FOCS 17–19, pp 444–453
18. Kwak H, Lee C, Park H, Moon SB (2010) What is twitter, a social network or a news media? In: WWW, pp 591–600
19. Low Y, et al (2010) Graphlab: a new framework for parallel machine learning. In: UAI, pp 340–349
20. Lumsdaine A, Gregor D, Hendrickson B, Berry JW (2007) Challenges in parallel graph processing. Parallel Process Lett 17(1):5–20
21. Madduri K, et al (2009) A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: IPDPS, pp 1–8
22. Malewicz G, et al (2010) Pregel: a system for large-scale graph processing. In: SIGMOD, pp 135–146
23. Meusel R, et al (2014) Graph structure in the web - revisited: a trick of the heavy tail. In: WWW, pp 427–432
24. Najork M, et al (2012) Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In: WSDM, pp 103–112
25. Ng TSE, Zhang H (2002) Predicting internet network distance with coordinates-based approaches. In: INFOCOM
26. Potamias M, Bonchi F, Castillo C, Gionis A (2009a) Fast shortest path distance estimation in large networks. CIKM, pp 867–876
27. Potamias M, Bonchi F, Castillo C, Gionis A (2009b) Fast shortest path distance estimation in large networks. In: CIKM, pp 867–876
28. Qi Z, Xiao Y, Shao B, Wang H (2013) Toward a distance oracle for billion-node graphs. PVLDB 7(1):61–72
29. Qiao M, Cheng H, Yu JX (2011) Querying shortest path distance with bounded errors in large graphs. In: SSDBM, pp 255–273
30. Ruan N, Jin R, Huang Y (2011) Distance preserving graph simplification. In: ICDM, pp 1200–1205
31. Shao B, Wang H, Li Y (2012) The trinity graph engine. Technical Report 161291, Microsoft Research
32. Tao Y, Sheng C, Pei J (2011) On k-skip shortest paths. In: SIGMOD, pp 421–432
33. Tian Y, Balmin A, Corsten SA, Tatikonda S, McPherson J (2013) From "think like a vertex" to "think like a graph". PVLDB 7(3):193–204
34. Yan D, Cheng J, Ng W, Liu S (2013) Finding distance-preserving subgraphs in large road networks. In: ICDE, pp 625–636
35. Yan D, Cheng J, Lu Y, Ng W (2014) Blogel: a block-centric framework for distributed computation on real-world graphs. PVLDB 7(14):1981–1992
36. Zhang X, Chen L, Wang M (2015a) Efficient parallel processing of distance join queries over distributed graphs. IEEE Trans Knowl Data Eng 27(3):740–754
37. Zhang X, Cheng H, Chen L (2015b) Bonding vertex sets over distributed graph: a betweenness aware approach. PVLDB 8(12):1418–1429
38. Zou L, Chen L, Özsu MT (2009) Distancejoin: pattern match query in a large graph database. PVLDB 2(1):886–897
39. Zou L, Özsu MT, Chen L, Shen X, Huang R, Zhao D (2014) gstore: a graph-based SPARQL query engine. VLDB J 23(4):565–590