Decentralized Search for Shortest Path Approximation in Large-scale Complex Networks

ABSTRACT

Finding approximate shortest paths for extremely large-scale complex graphs is still a challenging problem, where existing work requires too much overhead to achieve accurate results for graphs with hundreds of millions or even billions of edges. In this paper, we develop a decentralized search method based on preprocessed indexes, to approximate shortest path of arbitrary pairs of vertices. We demonstrate that the algorithm achieves very high accuracy with much less index overhead compared to previous work. The algorithm can also achieve differentiated level of accuracies by dynamically controlling the search space to meet various application needs. In order to perform decentralized search more efficiently, we propose a new heuristic index construction algorithm that can greatly increase the approximation accuracy. We further develop support for parallel searches to further reduce the average search time. We implement our algorithm in distributed settings to deal with extreme size graphs. Our algorithm can handle graphs with billions of edges and perform searches for millions of queries in parallel. We evaluate our algorithm on various real world graphs from different disciplines with at least millions of edges.

Categories and Subject Descriptors

E.1 [**Data**]: Graphs and Networks

General Terms

Algorithms, Performance

Keywords

Graphs, Shortest Paths, Decentralized Search

1. INTRODUCTION

Various types of graphs are commonly used as models for real-world phenomenon, such as online social networks, biological networks, the world wide web, among others. As their sizes keep increasing, scaling up algorithms to handle extreme size graphs with billions of vertices and edges remains a challenge that has drawn increased attention in recent years. Specifically, straightforward operations in graph theory are usually too slow or costly when they are applied to graphs at this scale. One problem that has drawn a lot of attention in the past is finding shortest paths in the network. This operation serves as the building block for many other tasks. For example, a natural application for road network is providing driving directions [1]. In social networks, such applications include social sensitive search [24], analyzing influential people [13], among others. Estimating minimum round trip time between hosts without direct measurement is another application in technology networks [21].

Although previous works have studied shortest path problem on large road networks extensively and have very effective approaches, a large category of networks known as the complex networks has very different structures. Approaches that worked on road networks do not perform well on complex networks, as the latter follow power law degree distributions and small diameters. In this paper, we are focusing on the shortest path problem for complex networks in particular, as their extreme sizes and unique topologies make the problem particularly challenging.

A straight-forward solution to this problem in complex networks is to apply online traversals like breadth first search or Dijkstra algorithm for each query. But this approach is very inefficient for large scale networks. The $O(|E|)$ time complexity and $O(|V|)$ space complexity make each search very costly. Another obvious solution is to trade space with time, by performing all-pair shortest path operations offline, and storing the results for each pair of vertices for table lookups. However, the cubic time and quadratic space complexity for preprocessing are not acceptable for large scale networks either. Nevertheless, using offline preprocessing to speed up online queries is very suitable for most applications.

Our design of the engine is motivated by recent studies that combine both offline processing and online queries [17], [23], [3], [18], [12]. In these methods, the step of preprocessing aims to construct indexes for the networks, which are later used in the online query phase to dramatically reduce the query time. Among these approaches, landmark based algorithms [22], [9], [17], [11], [23], [18]are widely used for approximate shortest path/distance between vertices. Usually such algorithms select a small set of landmarks, and construct an index that consists of labels for each vertex, that in turn store distances or shortest paths to landmarks. The approximation accuracy of landmark based algorithms heavily depends on the number of landmarks. Usually to achieve better accuracy, a relatively large set of landmarks is required, which lead to higher preprocessing overheads. Indexes that can answer path queries usually have much larger space overhead than indexes that can only answer distance queries. One goal of our design, therefore, is to provide accurate results while still maintain low overhead for indexing.

We achieve our goal of low-overhead timely query processing with a few design choices. First, instead of estimating shortest path solely from labels of source and target vertices, our algorithm performs a heuristic search during the online query period to explore edges that have not been indexed to achieve higher accuracy with limited index size. The heuristic search that we use is called decentralized search which was introduced by [14]. Here the "decentralized" means that the decision at each step is made based solely on local information which, in our context, is the labels of neighbor vertices.

Compared to other online search algorithms, decentralized search is very light-weighted. The number of visited vertices for decentralized search is bounded by twice of the diameter of the network. Considering that complex networks have relatively a short diameter, decentralized search can finish in very limited steps. Also, the algorithm does not need to mark which vertices have been visited like BFS or A* search, so only small space overhead is required for each search. This property makes it possible for large number of searches running in parallel without reaching the memory limit of machines. For example, in our experiments, we showed that millions of decentralized search can run in parallel. Furthermore, the average search time can be controlled at tens of microseconds or even shorter.

We also introduce several optimizations to control the search space of decentralized search to achieving different level of accuracies. With these optimizations, our algorithm becomes more versatile to meet various application needs without redoing the preprocessing. Specifically, decentralized search works in a way to explore edges that have not been indexed to achieve better accuracy. As its decision for each step relies on information in the index, not all the edges are equally important for estimating the shortest paths. To find out which edges should be stored in the index, we introduce a heuristic index construction algorithm that can increase the approximation accuracy by admitting only those most valuable edges.

Based on our algorithm design, we further develop a query-processing platform based on distributed cloud infrastructure. In this platform, users first submit their graphs for preprocessing needs. The graph processing engine will assign resources according to application's need for accuracy and construct an index for the input graph. Later, users may submit large volumes of queries repeatedly, for which responses will be generated. The light-weighted decentralized search allows a large amount of queries to run in parallel so that queries can be answered in a timely manner. Applications that generate queries (on the client side) can provide their desired accuracy levels and the graph processing engine can dynamically adjust search space of decentralized search to meet differentiated levels of accuracies.

1.1 Contributions

Our contributions can be summarized as follows:

First, as an approximation algorithm, we combine decentralized search with landmark based indexes to achieve a high level of accuracy. We optimize the search space of decentralized search to achieve low online search overheads. By controlling the search space, our algorithm is able to achieve differentiated levels of accuracies.

Second, we propose a more effective heuristic approach for constructing index of the network that can improve the accuracy of the decentralized search without increasing preprocessing and online search overheads.

Third, we implement our algorithm in a distributed manner to handle extremely large graphs and perform the decentralized search in a parallel way to further reduce the average online query time for better scalability.

The rest of this paper is organized as follows. In Section 2, we give notations and definitions used in this paper. We explain decentralized search for shortest path approximation in Section 3. Section 4 shows our index construction algorithm. In Section 5 we show details on our distributed implementation. And we show the evaluations of our algorithm in Section 6. In Section 7 we described previous works on exact and approximate approaches. We conclude our work in Section 8.

2. PRELIMINARIES

2.1 Notations

In our problem, we consider a network modeled as a graph $G = (V, E)$, which represents a vertex set V and edge set E. For a source node s and a target node t, we are interested in finding a path $p(s,t) = (s, v_1, v_2, ..., v_{l-1}, t)$ with a length of $|p(s,t)|$ close to the exact distance between s and t in the graph. Let $P(s,t)$ be the set of all paths from s to t. We can then define the distance between s and t as $d_G(s,t) = min_{p(s,t) \in P(s,t)} |p(s,t)|$.

2.2 Problem formulation

Our problem formulation is as follows: given a graph G, construct an index structure, and perform decentralized search based on the index to answer approximate shortest path queries of arbitrary pair of vertices in large scale networks. We are focusing on a specific kind of networks which is very common in real-world networks called the complex networks.

We will focus on unweighted, undirected graphs in this paper. But all the ideas presented in this paper can be extended for weighted and/or directed graphs.

2.3 Landmark based indexes

Our method is motivated by the idea of using landmarks as the basis for indexes. Specifically, given a graph G and a small (constant) set of landmarks L, $|L|$ BFS traversals are needed to compute shortest paths between each vertex in G

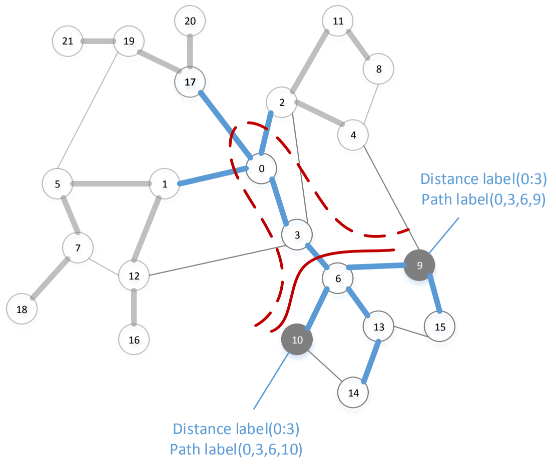


Figure 1: Distance of path estimated by distance-only label and path label. The dashed curved line shows the path estimated by the distance-only label. The solid curved line shows the path estimated by the path label.

to each landmark in L. An index for the graph is constructed such that it contains a label $L(v)$ for each vertex. Each label can store either the distance to each landmark $(l_i, d_G(v, l_i))$ or the shortest path to each landmark $(l_i, p(v, l_i))$ where $|p(v, l_i)| = d_G(v, l_i)$.

Since the shortest-path distance satisfies the triangle inequality, for an arbitrary pair of vertices s and t, we have the following bounds:

$$d_G(s,t) \leq min_{l \in L}\{d_G(s,l) + d_G(l,t)\} \tag{1}$$

$$d_G(s,t) \geq max_{l \in L}|d_G(s,l) - d_G(l,t)| \tag{2}$$

Usually the upper bound can be used as an approximation of the distance of s and t. Note that if s and t are not connected with each other, there will not be a common l from $L(s)$ and $L(t)$.

2.4 The least common ancestor distance

By indexing the shortest paths instead of distances into the label, one can not only derive an estimated path directly from labels of source and target vertices, but also have a better accuracy with the estimated distance. For example, in Fig. 1, the distance estimated from distance-only labels of vertex 9 and 10 is 6, representing the length of path $(9, 6, 3, 0, 3, 6, 10)$ shown in the dashed curved line. Although there is a clearly shorter path $(9, 6, 10)$ shown in solid curved line, with distance-only labels, such a path cannot be found.

More formally, we call the vertex 6 in the previous example as a common ancestor of vertex 9 and 10. When shortest paths rather than distances have been indexed, to get better

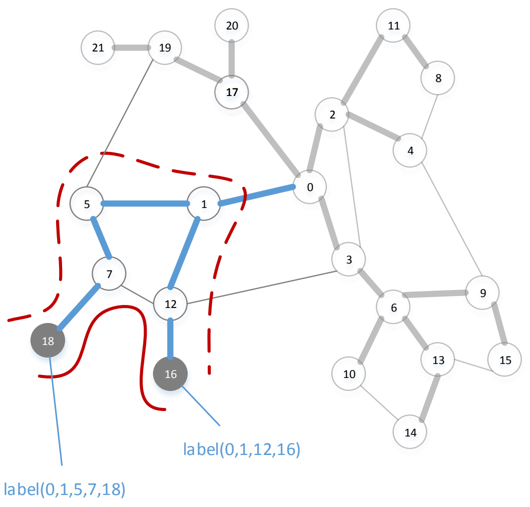


Figure 2: Decentralized search finds edges that are invisible to labels of the source and target vertices. The dashed curved line shows the path found by LCA distance. The solid curved line shows the path found by the decentralized search.

accuracy, the common ancestors closest to the source and target should be derived from the labels first. This common ancestor is called the least common ancestor $c_l(s, t)$ of the source and target vertex. Then instead of calculating upper bounds from the landmark l, the least common ancestor is used:

$$d_G(s,t) \leq min_{l \in L}\{d_G(s, c_l(s,t)) + d_G(c_l(s,t), t)\} \tag{3}$$

We refer to this upper bound as LCA distance $d_{LCA}(s, t)$ in this paper, which is a more accurate estimation of distance of source and target vertices. Note that both LCA distance and related paths can be derived directly from labels. For example, the path $(9, 6, 10)$ can be directly derived from the labels of the vertex 9:$(0 : (0, 3, 6, 9))$ and the vertex 10:$(0 : (0, 3, 6, 10))$ by combining the path from source vertex to LCA $(9, 6)$ and the path from LCA to target vertex $(6, 10)$.

3. DECENTRALIZED SEARCH FOR SHORTEST PATH APPROXIMATION

In this section, we are going to discuss how to perform decentralized searches based on the index structures to achieve a higher accuracy. We will also discuss how to optimize and control the search space of decentralized search to maintain low online search overheads.

3.1 Index guided decentralized search

By indexing the shortest paths from each landmark in the label of each vertex, we are able to answer both distance and path queries by simple LCA computations with labels of source and target vertices. However, paths returned by LCA computations only contain edges that already exist in

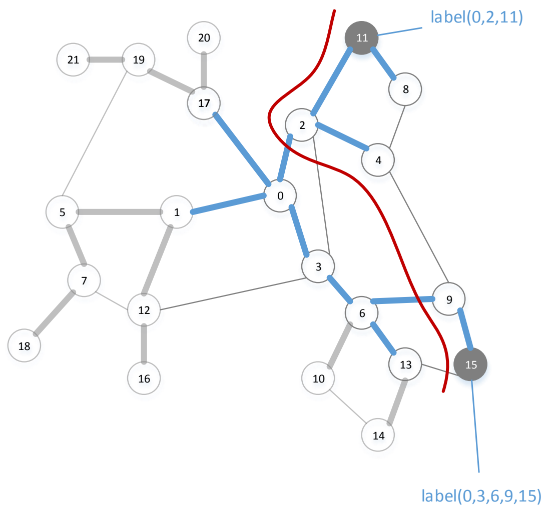


Figure 3: Decentralized search explores the edges that are not directly connected to the indexed shortest path. Edge $(4, 9)$ cannot be found by solely searching circles in the path by LCA distance.

the index structure. Observe that, on the other hand, only relatively a small part of the network, i.e., the edges along the shortest path tree, has been indexed. Therefore, such computations may not lead to a high accuracy. For example, in Fig. 2, using LCA distances to estimate the shortest path from 18 to 16 will return an approximated path $p = (18, 7, 5, 1, 12, 16)$, as represented by the dashed curved line. Note that there is an edge $(7, 12)$ that has not been indexed. Therefore, the path $p = (18, 7, 12, 16)$ cannot be found by computing LCA distance directly from labels of vertex 18 and 16.

To overcome such difficulties, we use a decentralized search in our design, which, at each step, will examine all the neighbors of current visited vertex and select one that is closest to the target. The search will continue this procedure at each step until it reaches the target vertex. In our case, the distance to the target is estimated by the LCA distance. By calculating the LCA distance to the target from each neighbor, the search can explore edges that do not exist in the labels of source and target vertices. For example, if we use decentralized search in Fig. 2 to estimate shortest path from 18 to 16, when examining neighbors of vertex 7, vertex 12 with a LCA distance of 1 to the target will be selected for next step instead of vertex 5 with a LCA distance of 3. In this way, a shorter path can be found by decentralized search through examining edge $(7, 12)$ which is not indexed in the labels of source and target vertex.

Next, observe that the edge $(7, 12)$ shown in Fig. 2 can also be found by searching if there is any edge existing between any pair of vertices on path returned by LCA computation, so that we can avoid traversing neighbors of any vertex. But this could only find edges that have both ends in the labels of the source and target vertex, which is a very small subset of all the edges. Rather than constraining the search space in this way, decentralized search examines all edges adjacent to current visited vertex, which is a more reasonable and larger subset of edges to examine. For example in Fig. 3 from 15 to 11, instead of finding a edge with both ends in labels of vertex 15: $(0, 3, 6, 9, 15)$ and labels of 11: $(0, 2, 11)$, decentralized search can find a edge with only one end in label vertex 15 which has a smaller estimated distance to the target 11. So that the path denoted by solid curved line can be found. Note that if decentralized search not only examines the edges that adjacent to current visited vertex, but also examines edges that are more than one hop away, shorter paths may be found. But due to the huge number of such edges, they have a relatively low possibility leading to a shorter path on average. Examining them may result in too much overhead.

3.2 Early termination

An important question is that when performing decentralized search based on LCA distances, will the search eventually terminate? The answer is that as long as the source vertex s and target vertex t are reachable from each other, which in our case means that s and t have common landmarks, decentralized search will terminate in as much as $2 * max_{u,v} d_G(u, v)$ steps, where $max_{u,v} d_G(u, v)$ is the diameter of the network. Too see this, note that for the LCA distance of arbitrary source and target vertex s and t, the following bound holds:

$$\begin{aligned} d_{LCA}(s, t) &\leq max_{l \in L}\{d_G(s, c_l(s, t)) + d_G(c_l(s, t), t)\} \\ &\leq max_{u,v} d_G(u, v) + max_{u,v} d_G(u, v) \end{aligned} \tag{4}$$

And at each step, as long as the current visited vertex is not the target vertex, there will always be a neighbor with a shorter LCA distance than current visited vertex to the target. So the estimated distance at each step will decrease at least by 1. Therefore, the LCA distance of arbitrary pairs of vertices is bounded by $2 * max_{u,v} d_G(u, v)$ according to equation 4. Decentralized search for arbitrary pairs of reachable vertices will terminate in as most $2 * max_{u,v} d_G(u, v)$ steps.

However, terminating only when the search reaches the target vertex is actually not an ideal stopping criterion, and will result in unnecessary overheads by examining redundant edges. Since the label of each vertex stores the shortest path of each vertex to each landmark, the shortest path follows the optimal substructure. That is, the path between any two vertices along the shortest path is also the shortest path of them. Considering that the label of each vertex follows the optimal substructure, the search can actually terminate once it reaches any vertex in the label of the target vertex. Because when the search reaching any vertex in the label of the target vertex, the label has already contained the shortest path from that vertex to the target vertex and decentralized search cannot find a path shorter than this one. Thus, the remaining path can be directly calculated from the label. Actually, decentralized search do not have to traverse vertices along the label of target vertices because these vertices have already been traversed by breadth first search for the same goal, which is reaching target vertex u, during preprocessing. With this stop criterion, the search space of

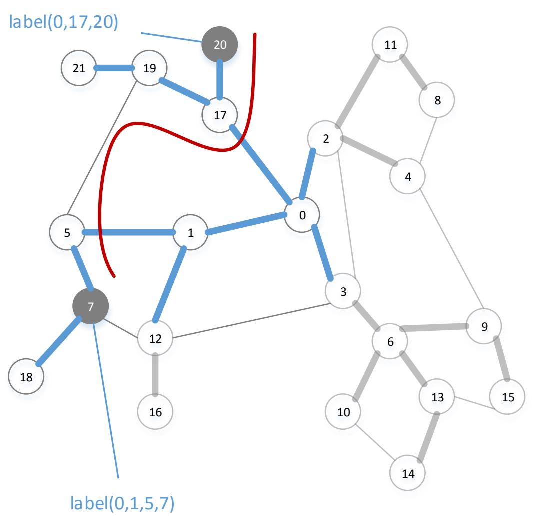


Figure 4: Bidirectional decentralized search can explore different set of edges which may found path at different length. Searches start at vertex 7 will lead to a path shorter than starting at 20 by taking advantage of the edge $(5, 19)$.

decentralized search is significantly reduced and the search can terminate much earlier. The search will only visit at most the number of vertices from source vertex to the least common ancestor of source and target vertices. So the decentralized search can terminate in at most $max_{u,v}d_G(u, v)$ steps. We refer to this optimization as early termination.

3.3 Bi-directional search

As we have shown above, decentralized search is well suited for exploring the edges that connect currently visited vertex to a vertex that has shorter LCA distance to the target. Intuitively, edges that can lead to a shorter path from target vertex to the source vertex are equally important. By performing the decentralized search twice, one from source to target, and the other from target to source, we can achieve better accuracy by traversing more edges that have equal possibilities leading to shorter paths.

Unlike bidirectional BFS, which is aimed at finding shortest path when two search meet, decentralized search does not guarantee that two searches will meet each other. The only purpose for doing bidirectional decentralized search is to explore more edges that have a high possibility leading to a shorter path. When combining the results of bidirectional decentralized search, one can simply select the path with a shorter length. For example in Fig. 4, the search starts from 20 to 7 can find a shorter path $p = (20, 17, 19, 5, 7)$ than the search starts at 7. Due to that 0 has a smaller LCA distance to 7 than 19, the edge $(19, 5)$ cannot be found by the search starts from 20.

3.4 Handle ties in decentralized search

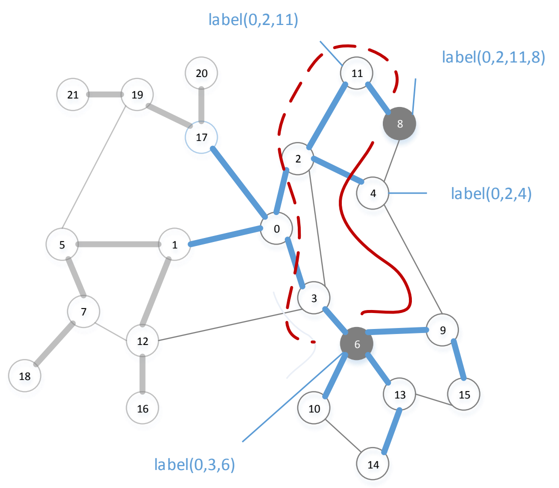


Figure 5: Tie happens during decentralized search. Although LCA distances are the same, selecting different neighbor will search different part of the graph which may lead to paths at different length.

Due to the small world property in complex networks, the shortest path tree of the landmark set usually has a limited depth. Furthermore, it is very common that a vertex connects to multiple vertices in a lower level of shortest path tree. So during decentralized search, there is a good chance that the search will encounter a tie at some point, that is, there are several neighbors of current visited vertex that have same shortest LCA distance to the target. For example in Fig. 5, to find path from 8 to 6, when traversing neighbors of vertex 8, both vertex 11 and 4 have the same LCA distance to vertex 6, but their actual distances to vertex 6 are different due to edges currently invisible to the decentralized search. Labels of each neighbor, on the other hand, provide no clue which one can lead to a shorter path.

Since edges at two hops away are invisible to decentralized search, the search has no way to differentiate neighbors when a tie happens. Edges adjacent to these neighbor vertices have the same possibility leading to a shorter path based on information available to the decentralized search. To increase the possibility of finding shorter paths, multiple neighbors with same shortest LCA distance to the target can be selected as candidates for the next hop. In this way, the decentralized search will traverse multiple or even all neighbors with the same shortest LCA distance at the next step. Apparently, doing so will bring extra overheads due to the larger search space. Note that by selecting multiple neighbors, decentralized search can possibly find multiple paths with the same length. This increases the diversity of approximated path of decentralized search, which is required by some applications.

4. INDEX CONSTRUCTION

As the decentralized search relies on the index structure to find paths, the constructed indexes serve as a crucial factor for accurate approximations. Previous works have studied

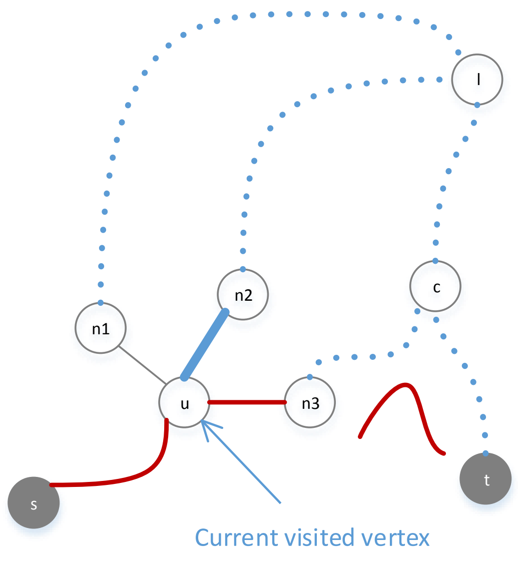


Figure 6: Decentralized search selects vertices with lowest common ancestor with target vertex at each step.

various landmark selection strategies which have a significant impact on the accuracy of online query. In our study, we observe that even with the same landmark set, choosing which edges to be indexed also has a great impact on the accuracy of online search. Because in practical networks, it is very common that there are more than one shortest paths from a vertex to a landmark. Furthermore, each vertex only stores one shortest path in their labels for each landmark. Different shortest paths indicate that different edges are indexed. In this section, we discuss how this will effect the accuracy of online search and propose a heuristic index construction algorithm that can generate an index more suitable for decentralized searches in terms of online query accuracy.

4.1 Impact of index on query accuracy

Although all shortest paths between two vertices are equivalent in terms of lengths, it is not necessary that they have same impact on the decentralized search since they have different vertices along the paths. Recall that in decentralized searches, at each step, the search will traverse neighbors of the currently visited vertex to find one with the shortest LCA distance to the target. Hence, a common ancestor means that the indexed shortest paths of two vertices intersect with each other. By doing this, decentralized search actually finds the neighbor with indexed shortest path that has an intersection with the indexed shortest path of target vertex at a lower level than other neighbors and the current visited vertex. For example, in Fig. 6, the dotted line shows indexed shortest paths to the landmark with vertices along the path. When examining edges adjacent to u, n_3 will be chosen as the next step due to that it has a lower common ancestor c than the other neighbors and u. Note that all the neighbors will have a common ancestor at the root with the target vertex as long as they are in the same connected component. But such common ancestors will not lead to a shorter path, so we do not need to consider them in our algorithm.

For a certain vertex u, it is possible to find the best shortest path from landmark to be indexed so that the shortest approximate path can be found from one vertex v to u. But such a shortest path may lead to a longer approximate path from another vertex w to u. The case will be much more complex when considering all the other vertices in the whole graph. So when constructing label for each vertex, we need to think about average cases. For each vertex, we want to find a shortest path to be indexed that has common ancestors with a larger number of other vertices. This is because during the decentralized search, the more neighbors of current visited vertex has common ancestor with target vertex, the higher the chance there will be a common ancestor at a lower level which suggests a shorter path. As we discussed before, a common ancestor means that two paths have intersections. So we are actually looking for the shortest path which intersects with most other paths to increase the chance for decentralized search to find a shorter path from other vertices to this one.

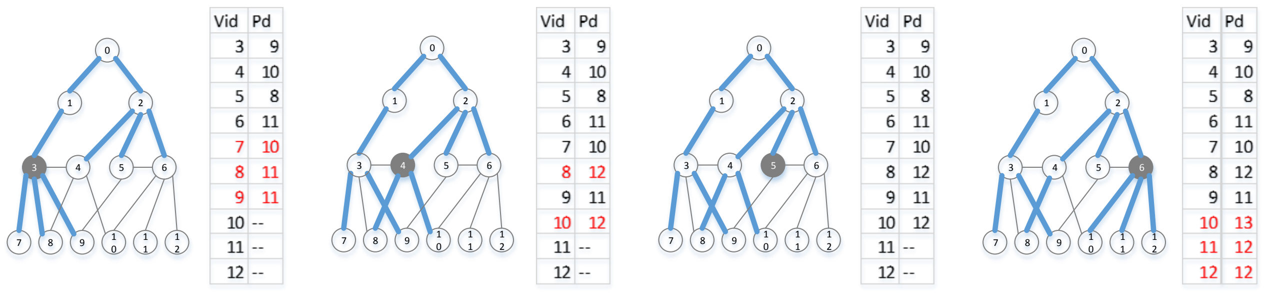
4.2 Heuristic index construction algorithm

We propose a greedy heuristic to construct an index which can lead to better accuracy in online queries. For each vertex u, when generating a label for it to store a shortest path to landmark l, as we discussed above, among all the shortest paths we want to store the one that intersects with most other paths. But it will be quite costly to calculate the number of paths a shortest path intersect with. To achieve this goal, we use a value which is much easier to compute to compare different shortest paths, the degree of all the vertices along each shortest path. We call this value the path degree Pd. The intuition here is that the higher the path degree of a path, the more other paths it may intersect with. Path degree can be easily calculated by summing up the degree of vertices along the path.

Same as the shortest path, the path degree of shortest path also follows optimal substructure. That is, if $(u, .., w, ..., v)$ has the highest path degree among all the shortest path from u to v, then the path degree of $(u, ..., w)$ is also the highest among all the shortest path from u to w. So for each vertex, the shortest path with highest degrees can be calculated easily by comparing path degree of indexed shortest path of its parents and selecting the highest one. Such calculation can be done during breadth first search with little overhead by caching the path degree of the label of each vertex. Fig. 7 shows an example of how to greedily select shortest path with the highest path degree during breadth first search. When traversing vertex 4, even though vertex 8 has already been indexed with a shortest path $(0, 1, 3, 8)$ into its label, due to that $(0, 2, 4, 8)$ has a higher path degree, the label of vertex 8 is updated. The same thing happens to vertex 10 while traversing vertex 6.

5. DISTRIBUTED IMPLEMENTATIONS

To handle extremely large graphs, we implement our algorithm in distributed settings. Due to that decentralized search does not require large volume of data to be cached



Vid	Pd
3	9
4	10
5	8
6	11
7	10
8	11
9	11
10	--
11	--
12	--

(a) Traversing 3: 7, 8, 9 stores the shortest path as they have been reached for first time.

Vid	Pd
3	9
4	10
5	8
6	11
7	10
8	12
9	11
10	12
11	--
12	--

(b) Traversing 4: 8 updates the label for larger path degree. 10 stores shortest path as it's been reached for first time.

Vid	Pd
3	9
4	10
5	8
6	11
7	10
8	12
9	11
10	12
11	--
12	--

(c) Traversing 5: 9 does not update its label since the path degree of shortest path through 5 is smaller.

Vid	Pd
3	9
4	10
5	8
6	11
7	10
8	12
9	11
10	13
11	12
12	12

(b) Traversing 6: 10 updates the label for larger path degree. 11, 12 stores shortest path for first time.

Figure 7: Heuristic algorithm that index shortest path with highest path degree during breadth first search

during the search, it is well suitable for running in a parallel way. Our implementations can handle graphs with billions of edges and running millions of independent queries in parallel.

We build our algorithm on a distributed general graph processing platform - Powergraph[10]. An overview of our system is shown in Fig. 8. Powergraph is designed to handle large-scale graphs with power law degree distributions efficiently by taking advantage of vertex-programs to factor computation over edges. Computation in Powergraph consists of a vertex-program running on a set of vertices. Each vertex-program consist of three phases: Gather, Apply and Scatter. During Gather phase, the *gather* and *sum* functions are used to collect and accumulate data from neighbors. The vertex data is updated in Apply phase by *apply* function through analysis on data collected from Gather phase. In the Scatter phase, *scatter* function is used to spread the new value to the graph and signal neighbors to start new vertex-programs. In this section, we are going to talk about details on how we implement our algorithm as vertex-programs.

5.1 Decentralized search vertex-program

Algorithm 1 shows the vertex-program of decentralized search. In Gather phase, for each query, LCA distance d_L calculated from labels L of each neighbor and target vertex is collected and accumulated by finding the neighbor with the smallest LCA distance. If a tie happens, according to our tie strategy, multiple neighbors may be returned as candidates. Since the decentralized search is a state-less search itself, vertex data does not need to be updated during the Apply phase. Instead, the program will append the candidate(s) to the approximated path $p_{appr.}$ and check whether the stop criterion is satisfied for each query. If so, the results will be recorded and this query will be terminated. Otherwise, program will proceed to Scatter phase to start new vertex-programs on next hop candidate(s) of each query and pass query information to them.

5.2 Index construction vertex-program

Algorithm 1 Algorithm decentralized search vertex program running on u

```
function GATHER($L(v)$, $L(t)$)
    return $d_L(v,t)$
function SUM($d_L(v_1,t)$, $d_L(v_2,t)$)
    return $min(d_L(v_1,t), d_L(v_2,t)$
function APPLY($p_{appr.}(s,t)$, $d_L(v,t)$)
    $p_{appr.}(s,t) += u$
    if termination condition meets then
        store $p_{appr.}(s,t)$
        $term$ = true
    else
        $term$ = false
function SCATTER($p_{appr.}(s,t)$, $term$)
    if $\neg term$ then
        Activate(v, $p_{appr.}(s,t)$)
```

Algorithm 2 illustrates the detailed algorithm to perform the BFS to construct the index. Different from regular BFS, when a vertex is visited by a new parent vertex, the path degree PD of the new path and the indexed path will be compared. The label will be updated if the new path has a higher path degree.

Our heuristic index construction vertex-program does not use the gather function. Instead, parent vertices send out messages containing required variables while activating child vertices. Multiple message from different parent vertices will be combined using a message combiner. The rule used here is to compare the value of path degree in order to select the one with the highest path degree.

5.3 Breaking ties

In the centralized version of our search algorithm, ties can be handled very easily. If a candidate cannot lead to the shortest approximate path at the next step among all candidates, it will be simply discarded. The situation becomes more complicated when implementing decentralized search in a distributed setting. In the distributed version of decentralized search, each candidate will start a new vertex program

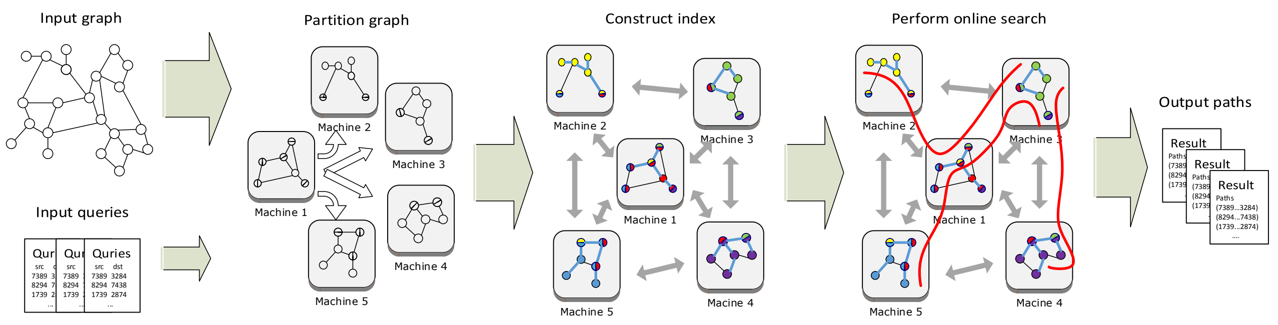


Figure 8: System overview.

Algorithm 2 Algorithm heuristic index construction vertex program running on u

```
function MSG_COMBINER($L(v_1)$, $L(v_2)$, $PD(v_1)$, $PD(v_2)$)
    if $PD(v_1) > PD(v_2)$ then
        return $L(v_1)$, $PD(v_1)$
    else
        return $L(v_2)$, $PD(v_2)$
function APPLY($L(v)$, $PD(v)$)
    $L(u) = L(v)$
    $L(u).push_back(u.id)$
    $PD(u) = PD(v) + u.degree$
function SCATTER($L(u)$, $PD(u)$)
    if $L(v).empty()$ then
        Activate($v$, $L(u)$, $PD(u)$)
```

to approximate shortest path independently. There are no communications among different searches as this would be too costly to implement in a distributed setting. Hence, a candidate does not know the length of the path found by other candidates. Even if a candidate finds a shorter path, it does not have the ability to terminate searches at other candidates.

With this limitation, the search space are quite difficult to control and the search may end up with excessive overhead. A naive to alleviate this problem is to set a hard limit on the number of neighbors selected as candidates for next hop. But the algorithm needs to first decide which ones to select, and there is no available information to help making this decision. Another approach used in our implementation is that at each step, only one candidate will be chosen as a "main" candidate. For candidates that are not "main" candidates, an extra stop criterion will be applied. If it cannot find a shorter path than the one currently found, then the search will stop, and no result will be recorded. This will certainly reduce the chance a shorter path to be found. But it has a much smaller overhead and treats candidates more adaptively than the hard limit approach.

5.4 Hot-spot prevention

Our implementation allows millions of decentralized search to run in parallel. But a hot spot problem may arise when running huge number of queries simultaneously. Since paths have more chance to pass a more *central* vertex, i.e. with a high degree or a high betweenness centrality. When multiple queries running in parallel, a large number of queries will be routed to these *central* vertices. However, those edges adjacent to these vertices are not necessarily evenly distributed to all machines. So computations may congested on several machines which will lead to larger online search overheads. The power law degree distribution of complex network will intensify this problem. Because it has a very small number of vertices with an extremely high degree.

The early termination plays an important role in alleviating the hot-spot problems. As the search will not examine any vertices in the labels of target vertex, the least common ancestor will never be traversed. Due to the way decentralized search approximate shortest paths, the least common ancestor is actually the vertex on lowest level of the shortest path tree rooted at the landmark. Note that landmarks are usually high degree vertices in the center part of the graph. And complex networks have a core-fringe structure [4]. Vertices on the lower level of the shortest path tree are more likely to have higher degree. So the least common ancestor are likely the highest degree vertex along the approximated path. By avoiding traverse the least common ancestor, the hot-spot problem can be greatly reduced.

6. EVALUATIONS

We evaluate our algorithms in distributed settings with 20 Amazon EC2 m1.xlarge machines. Each machine has 4 vCPUs and 15 GiB memory. We use Powergraph [10] as the platform to implement the distributed version. We also implemented a centralized version with Snap [15]. All algorithms are implemented in C++.

6.1 Datasets

We evaluate our algorithm on 9 graphs from different disciplines as shown in table 1 in ascending order on the number of edges. All graphs are complex networks that have power-law degree distribution and relatively small diameter. All datasets have at least millions of edges, the largest one has more than one billion edges. To simplify our experiments, we treat each graph as undirected, un-weighted graphs. We only use the largest weakly connected component of each graph that consist of more than 90% of vertices for all graphs. All datasets are collected from [15] and [19].

Dataset	Type	$\lvert V_{wcc}\rvert$	$\lvert E_{wcc}\rvert$
Google	Web	856K	4.3M
Wiki-talk	Communication	2.4M	4.7M
Skitter	Internet	1.7M	11.1M
Mouse-gene	Biological	43K	14.5M
Baidu	Web	2.1M	17.0M
Facebook	Social	3.1M	23.7M
Livejournal	Social	4.8M	43.4M
Hollywood	Collaboration	1.1M	56.3M
Friendster	Social	65M	1.8B

Table 1: Datasets

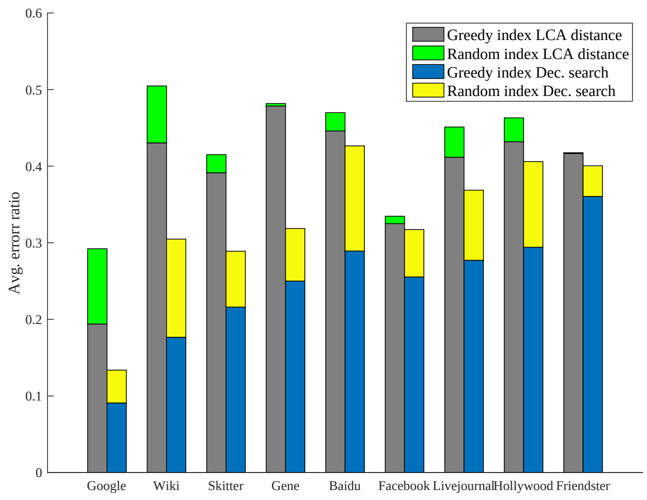


Figure 10: Impact of index quality on the accuracy of both decentralized search and label comparison.

6.2 Approximation Accuracy

In order to evaluate the performance of decentralized search and various optimizations, we compare the distance of estimated path and true distance of vertex pairs. To control the number of exact pairs of shortest path distance we need to find using BFS, we randomly choose 1,000 vertices of each graph as source vertices. For each source vertex, we randomly choose 100 vertices as target vertices. All the results in Fig. 9 and Fig. 10 are averaged with 100,000 queries.

We first compare the accuracy of decentralized search and LCA distance. For the decentralized search, we also compare accuracy of various optimizations including bidirectional search, distributed tie breaking strategy and greedy index construction to achieve different level of accuracies. The accuracy gain is shown in Fig. 9. All experiments are carried on with 2 landmarks except last one using 20 landmarks (note that we cannot construct a 20 landmark index for graph friendster due to the hardware limitations). From the figure, we can see that decentralized search achieves better accuracy for all graphs, and the performance gain varies from 8% to 64%. Various optimizations also have great impact on the accuracy. When all three optimizations have been used, the accuracy are even better than LCA distance computed from 20 landmarks for 7 out of 8 graphs (not including the graph friendster).

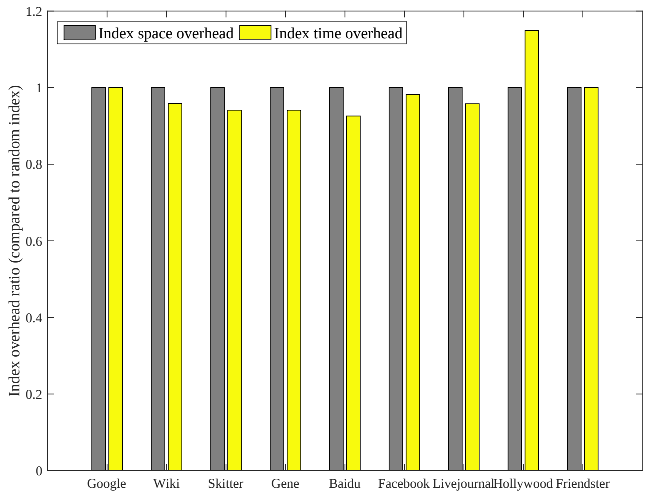


Figure 11: The time and space overhead for preprocessing for a single landmark by heuristic and random index construction

We then evaluate our heuristic index construction algorithm compared to the random index construction algorithm. We compared the results of decentralized search and LCA distance on both kinds of indexes. Fig. 10 shows the performance gain for a single landmark. As we can see that indexes created by greedy heuristic have a much smaller average error rate for both decentralized search and LCA computation. Due to that decentralized search examine more pairs of vertices than LCA computation, it has a better performance gain on 8 out of 9 graphs compared to LCA computation.

6.3 Overhead

We study the preprocessing and online search overhead of decentralized search in this section. First we compare the preprocessing overheads of greedy index construction with random index construction. Then for online search overheads, to show the impact of increased search space, we compare the average search time with various optimizations enabled.

Since whenever the shortest path to the landmark are indexed, they have the same path length, so the space overhead are exactly the same. This can be seen in Fig. 11 that greedy index construction algorithm brings no extra space overhead. Greedy index construction has limited time overhead too. Since the only additional operation of the algorithm is to calculate the sum of degree along each path, which takes constant time in our implementation. Actually, the difference of time overhead mainly comes from the number of times a label has been updated according to path degree or randomly. We can see in Fig. 11 that both preprocessing algorithms have similar time overhead.

Fig. 12 shows the online search overhead in log scale with different optimizations. Note that results for bidirectional search and tie breaking are both with early termination on. We can see that early termination greatly reduces the search overhead from 71.6% to 99.7% by reducing the number of vertices being examined. The time overhead of bidirectional search shown in Fig. 12 is 2.42 to 4.38 times more than one

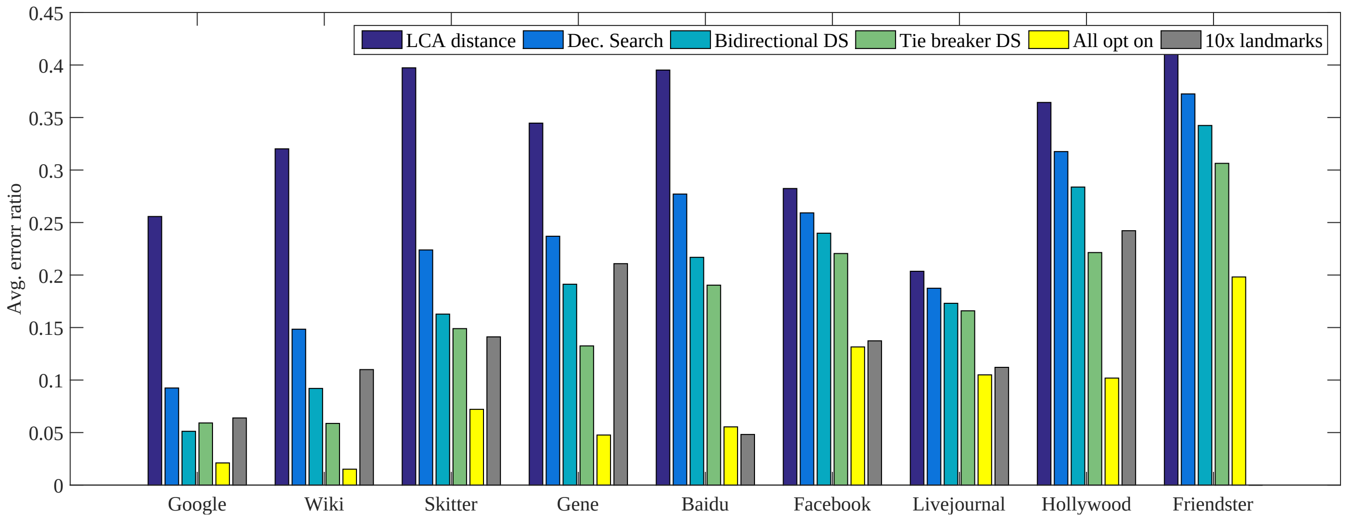


Figure 9: The accuracy of the approximated distance of decentralized search and various optimizations compared to LCA distance.

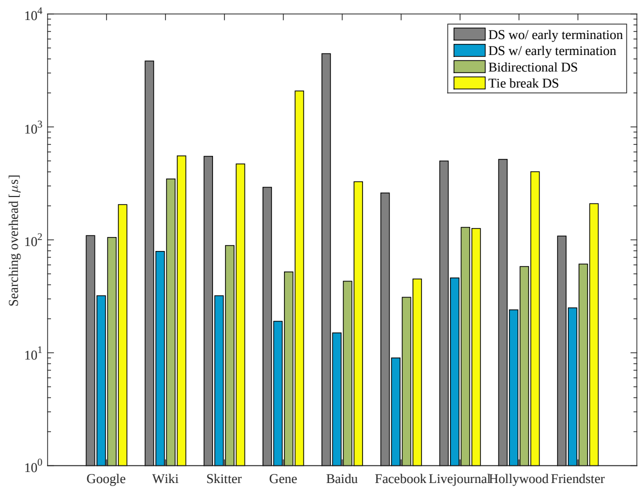


Figure 12: The average search time of decentralized search with various optimizations.

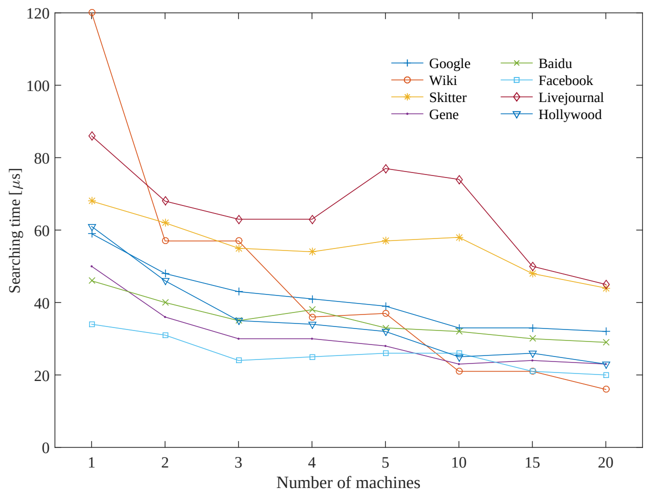


Figure 13: The average search time as the number of machines increase

direction search due to it has to found two paths instead of one path. The situation is a little complicated for tie breaking, since it is more related to the structure of each graph. For most graphs, the time overhead is 2.74 to 16.71 times more than one direction search. The worst case is 109.53 for graph mouse-gene which is the densest graph in our datasets, where too many candidates with same LCA distance exist at each step.

6.4 Scalability

Since our algorithm is designed for large-scale networks, scalability is another major concern of our algorithm. Due to that we implement the algorithm in a distributed setting and execute queries in a parallel way, we study how our algorithm performs when number of machines and queries changes. Moreover, as the size of graph is expected to further increasing in the future, we also studies the impact of the size of the graph to our algorithm.

We first evaluate the search time when number of machines increasing. Results shown in Fig. 13 are averaged of 1,000,000 queries. We can see the trend is that the average run-time decreases as the number of machines increase. The average search time decreases fast when small number of machines are deployed and slow down when large number of machines are used. However there are some spikes in the curve, which might be caused by the hot-spot problem we mentioned before.

We then examine average search time as number of queries running in parallel increases. All experiments are carried on 20 machines in this part. We can see in Fig. 14 that the average search time quickly goes down when the number of queries is small. The reason is that there are some fixed overheads to start and stop the engine for a batch of

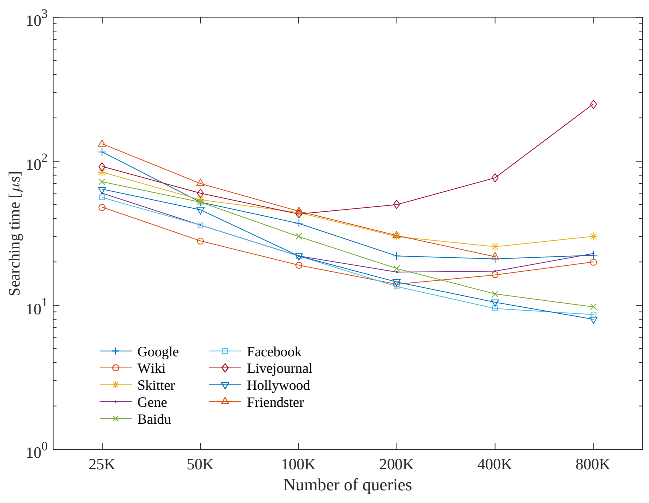


Figure 14: The average search time as the number of queries increase

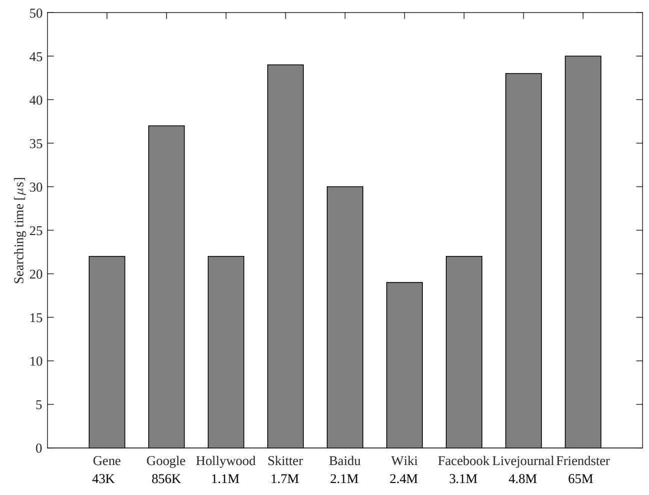


Figure 15: The average search time for different size of graphs

queries. When the number of queries increases, the average overhead for each query will decrease. But when the number of queries becomes larger, the average search time begins to go up. Because too many searches will reach the memory and communication limits, that slow down the search speed.

In the last experiment, we evaluate the search time as the size of graph increases. Unlike BFS which needs to traverse neighbors of $O(n)$ vertices, decentralized search only needs to exam neighbors of $O(log(n))$ vertices for complex networks due to the small world property. We can clearly see in Fig. 15 that as the number of vertices increases, average search time of decentralized search does not follow the same trend. The graph friendster which have sixty-five millions vertices have average search time only 2.05 times more than graph mouse-gene which only have forty-three thousand vertices.

7. RELATED WORKS

Existing work on shortest path/distance can be classified into exact approaches and approximate approaches.

Exact Approaches: Majority of the exact approaches are based on either 2-hop cover [6], [2] or tree decomposition [3], [25]. For the former one, finding optimal 2-hop covers is a challenging problem. [2] takes a different approach that solving 2-hop cover problem with graph traversals which has better scalability. [12] borrowed the highway concept from shortest path algorithms on road networks and construct a spanning tree as a "highway". [8] introduced an effective disk-based label indexing method based on independent set.

Approximate Approaches: Since the exact approaches do not scale very well, approximate algorithms are also well studied for large-scale complex networks. Landmark based algorithms are extensively studied for approximate shortest path/distance [22], [9], [17], [7], [16]. Although theoretical study of such algorithms does not reveal promising results [22], they usually work well in practice. [17], [20] studied various landmark selection strategies for constructing better indexes. A common problem for distance-only indexes is that they do not perform well for close pairs of vertices [3]. Algorithms [11], [23], [18] which index shortest paths are proposed to alleviate this problem. Beside landmark based approaches, there are several other approximate approaches. [5] forms the shortest path problem as a learning problem to predicting pairwise distance. [26] maps vertices to low-dimension Euclidean coordinate spaces to answer distance queries in constant time.

Combining online search with indexes: [9] uses A* search for online query based on indexes constructed by landmark based algorithms. However, the cost of each A* search is still very high for large scale networks. [11] perform BFS on a sub-graph generated by the labels of source and target vertex. Although search space is greatly reduced, it still needs around seconds to handle graphs with millions of vertices.

8. CONCLUSION

In this paper, we describe a novel method to combine online and offline processing to allow approximate searches for extremely large graphs with high accuracy and low overhead. We demonstrate that different accuracy and overhead levels can be achieved by various optimizations controlling the search space of decentralized searches. We also propose a more effective heuristic approach for constructing indexes of the network that can improve the accuracy of the decentralized search without increasing preprocessing and online searching overhead. We implement our algorithm for cloud computing graph processing platforms, and demonstrate that our system can handle extremely large graphs and processing millions of queries in parallel.

9. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th International Conference on Experimental Algorithms*, SEA'11, pages 230–241, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 349–360, New York, NY, USA, 2013. ACM.

[3] T. Akiba, C. Sommer, and K.-i. Kawarabayashi. Shortest-path queries for complex networks: Exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 144–155, New York, NY, USA, 2012. ACM.

[4] D. S. Callaway, M. E. Newman, S. H. Strogatz, and D. J. Watts. Network robustness and fragility: Percolation on random graphs. *Physical review letters*, 85(25):5468, 2000.

[5] M. Christoforaki and T. Suel. Estimating pairwise distances in large graphs. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 335–344, Oct 2014.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[7] V. Floreskul, K. Tretyakov, and M. Dumas. Memory-efficient fast shortest path estimation in large social networks. In *Eighth International AAAI Conference on Weblogs and Social Media*, 2014.

[8] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: An independent-set based labeling scheme for point-to-point distance querying. *Proc. VLDB Endow.*, 6(6):457–468, Apr. 2013.

[9] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[11] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 499–508, New York, NY, USA, 2010. ACM.

[12] R. Jin, N. Ruan, Y. Xiang, and V. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 445–456, New York, NY, USA, 2012. ACM.

[13] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM.

[14] J. Kleinberg. Navigation in a small world. *Nature*, 406:845, 2000.

[15] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[16] M. Maier, M. Rattigan, and D. Jensen. Indexing network structure with shortest-path trees. *ACM Trans. Knowl. Discov. Data*, 5(3):15:1–15:25, Aug. 2011.

[17] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 867–876, New York, NY, USA, 2009. ACM.

[18] M. Qiao, H. Cheng, L. Chang, and J. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. *Knowledge and Data Engineering, IEEE Transactions on*, 26(1):55–68, Jan 2014.

[19] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[20] F. Takes and W. Kosters. Adaptive landmark selection strategies for fast shortest path computation in large real-world graphs. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on*, volume 1, pages 27–34, Aug 2014.

[21] L. Tang and M. Crovella. Virtual landmarks for the internet. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, pages 143–152, New York, NY, USA, 2003. ACM.

[22] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, Jan. 2005.

[23] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1785–1794. ACM, 2011.

[24] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto. Efficient search ranking in social networks. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 563–572, New York, NY, USA, 2007. ACM.

[25] F. Wei. Tedi: Efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 99–110, New York, NY, USA, 2010. ACM.

[26] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao. Orion: Shortest path estimation for large social graphs. In *Proceedings of the 3rd Wonference on Online Social Networks*, WOSN'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.