Graph Calculus: Scalable Shortest Path Analytics for Large Social Graphs through Core Net

Lixin Fu
Department of Computer Science
University of North Carolina at Greensboro
Greensboro, NC, U.S.A.
lfu@uncg.edu

Jing Deng
Department of Computer Science
University of North Carolina at Greensboro
Greensboro, NC, U.S.A.
jing.deng @uncg.edu

***Abstract*— We focus on the problem of scalable shortest path analytics for large social graphs in this paper. While shortest path distance problem has been investigated extensively, massive graphs on social networks such as Facebook and LinkedIn call for reinvestigation of the problem due to the requirements of scalability and suitability for distributed computing. We propose a core-net based approach to address the problem. In our new core-net algorithm, popular nodes are selected based on their node degrees. Since some of these popular nodes may not be connected, we further include bridge nodes, which connect two or more popular nodes and improve their connectivity, but are not popular nodes themselves. Breadth First Search (BFS) technique is then used to compute shortest paths between any pair of nodes on the core net. When the shortest path between two arbitrary vertices, u and v, is queried, we approximate it with triangulation. We present a graph calculus theory in which the estimated distance goes to the real shortest distance when the degree threshold goes to zero. Analysis and simulation results confirm the superiority of our design, which can easily scale to MapReduce.**

Keywords—shortest-path distance; breadth-first search; core network; graph calculus;

I. Introduction

Shortest path distance query has long been an interesting problem in computer science. Usually such queries need to find the shortest distance between any two vertices in a graph. In the past decade, the huge popularity of social networks such as Facebook, Twitter, and LinkedIn introduces new excitement in the research field. In such massive social networks, it is interesting to identify the number of hops (distance) through friends, friends of friends etc. and the path between two people.

There have been many algorithms addressing the shortest path analytics. For example, both of Dijkstra's algorithm and Bellman-Ford algorithm are efficient in searching for shortest paths on weighted and directed graphs. Breadth-First Search (BFS) has an O(V+E) complexity on unweighted graphs, where V and E are the number of vertices and the number of edges on the graph, respectively. A 2-hop labeling algorithm, TEDI, and a highway labeling algorithm have been proposed recently. These algorithms try to approximate the shortest-path queries in order to be more computationally efficient. In this work, we focus on the problem of shortest path distance query for unweighted and undirected graphs such as the massive Facebook graph.

Extensive applications of such shortest-path analytics are naturally expected, in semantic web, biological networks. For instance, they can help to answer the questions such as how close two users are on social networks as well as how fast/far any interesting post from one user can reach other users. In biological networks, it might be possible to predict how likely two patients with the same disease share similar gene sequences.

In this work, we propose a core-net based approach to address the problem. Based on the intuition that nodes with higher node degrees are more likely on the shortest paths, we first select the nodes whose degrees are above a user-selected threshold. We call these nodes Popular Nodes (PN). Since some of these popular nodes may not be connected by themselves, we further include Bridge Nodes (BN), which connect two or more popular nodes. These bridge nodes are chosen such that they connect those otherwise disconnected popular nodes. The graph containing the popular nodes, the bridge nodes, and the corresponding edges connecting them are called Core Net. Breadth First Search (BFS) technique is then used to compute shortest paths between any pair of nodes on the core net. When the shortest path between two arbitrary vertices, *u* and *v*, is queried, we first search through the core net. If not found in the core net, we approximate it with triangulation, i.e., the shortest path from vertex *u* to core net, the shortest path inside core net, and the shortest path from core net to vertex *v*. We provide detailed analysis and extensive simulations to evaluate the performance of our design, which can easily scale to MapReduce.

Our main contributions in this work are three-fold:

1. We propose a new estimation algorithm based on the concept of core net. Different from other landmark algorithms, our BFS computation are only applied to the core net, making it possible to be executed by in-memory algorithm. This is in constrast to other schemes applying BFS to the entire huge input graphs. Our algorithm is two to three orders faster and feasible for large graphs with much better estimation accuracy.

2. We present a foundation of a new theory of graph calculus where the estimated distance goes to the real shortest distance when the degree threshold goes to zero. This provides us a continuum of trade-off space between computing cost and accuracy.

3. We give parallel strategies for our algorithms to fit in the cloud and distributed computing environment. We have also performed extensive simulations on different data sets to evaluate our scheme.

The rest of the paper is organized as follows. Section II surveys related work in the field and distinguishes our approach from others. Section III provides detailed description of our proposed algorithm. In Section IV, we present our graph calculus theory and parallelization strategies. Analysis and performance evaluation are presented in Section V, followed by Section VI summarizing the work.

II. Related Work

The shortest path distance query problem is a well-studied problem in computer science. However, the unique features of such queries in massive social networks make it pertinent to be re-investigated. In the following, we survey several closely related techniques. We assume that in a general graph G(V, E), n=|V|, m=|E|. For any two nodes u and v, their shortest distance is denoted as δ(u, v) and their path is SP(u, v).

Dijkstra's algorithm [1] and Bellman-Ford algorithm [2] are probably the most well-known algorithm in shortest path distance query. For instance, Dijkstra's algorithm helps to identify an accurate shortest path in a weighted and directed graph G. The algorithm can be implemented with $O(m+n\log(n))$ time; therefore, it is efficient in sparsely connected networks. For unweighted graphs, the Breadth-First Search (BFS) [3] can be used with $O(m+n)$ time. For all pair shortest path (APSP) problem, BFS can be repeatedly called, thus taking $O(n(m+n))$. To solve APSP problem in a densely-connected graph, the Floyd-Warshall algorithm [4] takes $O(n^3)$ time. In summary, for an undirected, unweighted, and sparse graph such as the Facebook net, BFS will be best among these four classic algorithms. However, BFS cannot be directly applied to a large social graph with up to a billion nodes simply due to the storage issue (they are not stored in one machine) or computing costs. Furthermore, BFS is a global algorithm and, therefore, difficult to be parallelized.

Recent research on shortest path query focuses on approximation techniques. The 2-HOP labeling algorithm [5] is one of such techniques that require two steps in solving the problem. In the pre-processing step, it computes, for each node *u*, a list of intermediate vertices OUT(*u*) that represents all the vertices node *u* can reach with a shortest distance. Then for each node *v*, it computes a list of intermediate vertices IN(*v*) that represents all the vertices that can reach node *v* with a shortest distance. In the query processing step, when a directed path from node *u* to node *v* is needed, it only needs to compute a vertex *p* such that $\delta(u,p)+\delta(p,v)$ is minimum for all *p* in OUT(*u*)∩IN(*v*). The algorithm is obviously an approximate. In addition, the pre-processing step is costly. A similar 3-HOP algorithm has also been developed [6].

A local landmark algorithm is recently proposed by Qiao et al. [7]. In a shortest path tree rooted at a global landmark, a local landmark is the least common ancestor of the two query nodes. The local landmark is query dependent and can improve estimation accuracy.

Wei developed an interesting TEDI algorithm [8]. Instead of depending on pre-computation of compressed BFS trees of the graph, TEDI decomposes the graph into a tree in which the node contains more than one vertex, i.e., super node. The shortest paths are stored in these nodes. The search of shortest paths for any given source and target vertices depends on a bottom-up approach based on the shortest paths stored on the nodes as well as the decomposed tree.

Jin et al. proposed a highway-centric labeling approach specifically for sparsely connected graphs [9]. The strategy simulates the usage of highway systems in transportation networks. A highway is constructed with known distance between any two points. When the shortest distance is queried for two nodes *u* and *v*, it only needs to compute the distance from *u* to the highway and from highway to *v*. The labeling and construction of highway is the core issue in the algorithm. Instead, we observe that there are vertices with much higher degrees in social networks than others. In this work, we develop an efficient algorithm based on the usages of these vertices.

Most algorithms of computing shortest distance queries are based on first choosing landmarks that are used to estimate the distances based on triangulation. Potamias et al. gave a proof that the optimal selection of the landmarks is a NP-hard problem [10]. They proposed two heuristics for choosing the landmarks. The first is based on constraint degrees where the large-degree node is chosen first. However, the second largest-degree node may not be chosen if it is too close to the first chosen node (e.g. next to each other). The landmarks are selected in a greedy fashion until the number of landmarks reaches a threshold. The idea behind the unique selection process is that the popular nodes should not be close to each other so that they are more spread with better representations of concentration. The second strategy to choose landmarks is based on global centrality. First, a random sample of nodes is selected as seeds. The one with the smallest average distance to all nodes is selected. The node with next smallest average distance is chosen up to a threshold number of landmarks. When these landmarks are computed, each makes a BFS call to compute the shortest distances to all other nodes to be used to estimate distances. Our algorithm differs from these methods in that we apply BFS only on our core nodes but most of these methods compute the shortest distance to *all* nodes in the input graph. Our algorithm can also return accurate distances if the queried nodes are close.

MapReduce [11, 12] is probably the most popular framework to process large data sets with parallel and distributed algorithms on a cluster of computers. It contains two major procedures: a Map() procedure that filters/sorts a subset of the data; and a Reduce() procedure that summarizes the results from Map() and produce the final result. Hadoop is an open source implementation of MapReduce[13]. On a similar trend, cloud computing [14] allows users of different computer systems to rely on only the servers from cloud service providers, data processing, storage, even virtual computers can be provided by such cloud service providers. Algorithms on large data sets should be converted to MapReduce and can be deployed in a cloud computing environment.

III. The Core-Net Algorithm

In this section, we present the details of our proposed algorithm and explain it with an example graph.

A. *Popular Net*

Our algorithm is first based on the well-known Small World assumption, which is also called Six-Degrees of Separation theory. In this theory, it is claimed that any two people can be connected through friends by at most 6 steps. More recently, it has been identified that the degree of separation is actually 4 [15] in Facebook, meaning that any two nodes in the Facebook graph G(V, E) are connected through a shortest path of at most 4 hops most of the times. Based on this fact, most of times we just need to directly compute the shortest distances between two given nodes online exactly and locally. Our scalable algorithm takes advantage of such an observation.

Another important observation in graph G(V, E) is that some of the nodes have much higher node degrees than others (popularity). With the large number of edges connecting to these popular nodes, it is quite likely that many of the shortest-path queries can be answered by these nodes if the distance is more than four. While there may exist some pair nodes whose shortest paths between them do not contain popular nodes, a) the chance of this happening is small; b) the neighbors of the queried nodes may be among the core nodes.

In our algorithm, if a node's degree is no less than a threshold, ϑ, it is designated as a popular node. We denote the set of popular nodes as PN and with their connections as a popular net graph (PG). For example, in Figure 1, nodes 1, 2, 8, and 12 are popular nodes if the threshold ϑ is 4.

The algorithm to identify the popular net is presented in Algorithm 3.1. The runtime complexity is $O(n)$. The space complexity of cg is $O(|E_{PG}|)$, a small fraction of *ig*.

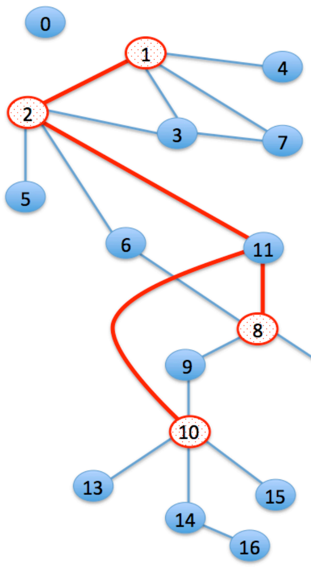


Figure 1, Illustration of a social graph with some popular nodes and bridge nodes. The red-dotted nodes are popular nodes. Node 11 is a bridge node.

```
Algorithm 3.1 Popular_net
Input: the input graph linked lists ig[i], i=1..|V|;
threshold ϑ,
Output: the sub graph that only contains popular
nodes and edges connecting themselves. The
original popular node IDs are stored in array pn and
the link lists are stored in cg[i], the original
neighbor IDs of the ith popular node. Cn_map is
used to map the original input node IDs to the
compact new IDs in 1..|pn|
  1.  k = 0;
  2.  for i=1, |V|
  3.      If deg(i) > ϑ
  4.          pn.add(i)
  5.          cn_map.put(i, k++)
  6.  for i=1..|pn|
  7.          cg[i] = ig[pn[i]] ∩ pn
  8.  return cg;
```

B. *Bridge Nodes and Core Net*

Though the popular nodes are naturally better connected, and therefore helpful in the estimation of other distances, PG is not necessarily a connected graph. The common friend of the celebrities will then play the role of a "bridge." A node that connects two popular nodes is called a bridge node (BN). It may increase the connectivity of the popular net which is otherwise separated or their distances would be larger. So we also add these bridge nodes and their edges connecting the popular nodes (called bridge edges) into PG. The expanded graph is called core graph (CG). Calculate the pair-wise distance matrix for CG using Algorithm dist_matrix (Algorithm 3.2). The matrix will be used in distance query evaluations.

```
Algorithm 3.2  dist_matrix (G)
Input: graph G, |V| the number of nodes in G
Output: dist[i][j]), shortest distance from i to j in V
  1.  for i = 1 to |V| do
  2.          di = bfs (i), di is the distances from i to
      other nodes
  3.          For j = 1 to |V| dist[i][j] = di[j]
```

To compute the bridge nodes, we first calculate all the candidate bridge nodes cn (line 2-6 in Algorithm 3.3). These non-popular nodes should directly connect at least two popular nodes (their indices are stored in array p, line 4 in Algorithm 3.3). We implement it using a bubble algorithm in which the nodes with smallest IDs are repeatedly deleted from a min heap. In Figure 1, for instance, nodes 3, 6, 9, and 11 are the candidate bridge nodes because they are directly connected to two or more popular nodes. Node 6 connects two popular nodes 2, and 8 directly. So does node 11, which will be chosen as a bridge node eventually.

After the candidates are identified, we first compute the distance matrix of the poplar nodes by calling dist_matrix (cg). If adding a candidate node *k* can improve the connectivity among neighboring popular nodes in *p[k]*, *k* will be selected as a bridge node while updating the improved distances (lines 8-9 in Algorithm 3.3). Among the candidates, we first try those with largest degrees. Adding node 3 does not improve the distance between its neighboring popular node 1 and 2, so it is not chosen. Adding node 11's connections to nodes 2 and 8 will shorten their distance from ∞ to 2, so it is chosen. Nodes 6 and 9 are discarded because they do not shorten the distances of popular nodes after node 11 is already selected.

```
Algorithm 3.3 select_bridges
Input: original lists ig[i], for each i in pn
Output: selected bridge nodes stored in array bn

    1.  // find all the candidate bridge nodes cn
    2.  Initialize |pn| queues q[i] = ig[pn[i]], i=1..|pn|
    3.  Using a min heap to find the smallest node m
        among the head notes of queues, delete m from
        heap
    4.  If any, find all the other head nodes that are equal
        to m (stored their queue indexes in p[m] for m),
        deque those lists, and insert the corresponding
        next nodes in queues into heap after deleting each
        min
    5.  if m $\notin$pn and m is shared by pn nodes, cn.add(m)
    6.  Repeatedly perform step 3 to 5 until the queues
        are empty

    7.  // select bridge nodes bn
    8.  for k =0 to |cn| do
    9.   for each pair i, j$\in$ p[k],
        if dist[i][j] >2 then
                dist[i][j] = 2 and   bn.add(k)
    10. Return bn
```

To compute the expanded core graph, all these newly selected bridge nodes are added to pn and the lists of bn are added to cg. In our example in Figure 1, the nodes 1, 2, 8, and 10 are popular nodes and node 11 is a bridge node. The original input lists for newly expanded pn are cleaned to a new cg, so that these lists only contain the popular and bridge nodes. The distance matrix for cg is re-computed for answering queries. CG graph will serve as the intermediates to estimate the distance: $\delta(u, v) \leq \delta(u, p) + \delta(p, q) + \delta(q, v)$, where both *p* and *q* are in CG. The time and space complexity of computing the candidates are O(|pn| *M_G), where M_G is the maximum degree of input graph G. This complexity is linear to the core graph size. The selection of bridge nodes takes O(|cn| * $|pn|^2$), |cn| is the number of candidates generated in lines 2-6.

C. Query Evaluation and Estimation

Once the core graph and its distance matrix are computed, we are ready to answer distance queries online. Given two nodes *u* and *v*, we first compute their neighbor sets N_u and N_v, and neighbor-of-neighbor set N_u^2, N_v^2. The computation of N_u^2

```
Algorithm 3.4 Neighbor-of-neighbor
Input: input graph lists ig, a node u
Output: the neighbor–of-neighbor set $N_u^2$ of u
    1.  for i= 1..|$N_u$|
    2.          $N_u^2 = N_u^2 \cup N_i$
    3.  Sort $N_u^2$
    4.  Eliminate duplicates in $N_u^2$
    5.  Return $N_u^2$
```

and N_v^2 are shown in Algorithm 3.4. Through the intersections of these sets, we can precisely compute the shortest distance from *u* to *v* directly if the distance is no larger than 4. The algorithm is shown in Algorithm 3.5. Both algorithms 3.4 and the lines 1-5 in 3.5 take O(|N_u^2 |+ | N_v^2|), i.e. the total number of the circles (friends of friends) of u and v.

If the shortest distance path between two vertices, *u* and *v*, is longer than 4, two popular nodes[1], *p* and *q*, will be found such that the distance on the path *u-p-q-v* is the shortest. While there may exist shortest paths between a pair of vertices *u* and *v* that may not include any popular nodes, the chance of such event happening is rather small. We evaluate the accuracy of our algorithm in Section V. The estimate time of lines 6-23 in algorithm 3.5 is O(|N_u^2 |+ | N_v^2|+|pn|) since the distance within the core net is constant through pre-computation.

IV. Graph Calculus Theory and Paralelization Strategies Using Map-Reduce

In the context of core net CG as an approximation of the original graph, we develop the following definitions and theorem to form the foundation of a potentially new theoretic field, which we dubbed **Graph Calculus**.

DEFINITION 1. (**Graph Function *f***) f: $\Theta \rightarrow S_\vartheta$. Degree threshold ϑ is the argument (or input variable). Its domain $\Theta = \{0, 1, \ldots M_G\}$, where M_G is the max degree of the input graph G(V, E). For each ϑ in Θ, its image is a core graph G_ϑ (V_ϑ, E_ϑ), a subset of G(V, E) such that $V_\vartheta = \{v | v \text{ in } V, \deg(v) \geq \vartheta\}$ and $E_\vartheta = \{e=(i,j) | e \text{ in } E \text{ and } i, j \text{ in } V_\vartheta\}$. The co-domain is the graph space S_ϑ induced by ϑ, which is defined as the following.

DEFINITION 2. (**Graph Space** S_ϑ) Graph space induced by ϑ is $S_\vartheta = \{G_\vartheta \mid \vartheta \text{ in } \Theta\}$.

DEFINITION 3. (**Graph Limit**) When $\vartheta \rightarrow 0$, $G_\vartheta \rightarrow G$, i.e., $||G_\vartheta - G_0|| \rightarrow 0$. The distance function || . || is defined by users.

DEFINITION 4. (**Shortest Distance Estimate**) $\delta_\vartheta(u, v)$ is the shortest distance between *u* and *v* computed by our algorithm based on G_ϑ (V_ϑ, E_ϑ).

THEREOM 1. (GRAPH CENTRAL LIMIT THEOREM) *As variable ϑ approaches 0, the core graph G_ϑ induced by ϑ approaches the original graph G(V, E):*

$$\lim_{\vartheta \to 0} G_\vartheta = G \qquad (1)$$

[1] Strictly speaking, these two popular nodes can be the same node.

COROLLARY 1. (SHORTEST-DISTANCE CENTRAL LIMIT THEOREM) As ϑ approaches 0, the shortest distance estimate approaches the true shortest distance in G, i.e.

$$lim_{\vartheta \to 0}\, \delta_\vartheta(u, v) = \delta(u, v) \qquad (2)$$

```
Algorithm 3.5, distance (u, v)
Input: two nodes u and v, input graph ig lists, core nodes pn, distance matrix dist of core graph cg.
Output: exact or estimate of u to v distance
// directly and exactly compute the distance of no more than 4
1.  Compute $N_u$ and $N_v$ i.e. the ig[u], and ig[v]
2.  if ( $v \in N_u$)      return 1;  // u - v
3.  if ($N_u \cap N_v \neq \phi$)  return 2;
        // u - x – v, x is common neighbor of u and v
4.  if ($N_u^2 \cap N_v \neq \phi$)  return 3;  // u - u1 - x - v
5.  if ($N_u^2 \cap N_v^2 \neq \phi$)  return 4; // u - u1 - x - v1 - v

// estimate the distance between u and v
6.  // both in core  U --- V,
        upper case: in core,  --- : shortest u to v path
7.  if (u $\in$pn  && v $\in$pn)
        return dist[cn_map.get(u)][cn_map.get(v)];
8.  if (u$\notin$pn) $u_1 = N_u \cap$ pn
9.  if (v$\notin$pn)  $v_1 = N_v \cap$ pn
//u - U1 --- V, u- U1: u directly connects $u_1$ in core (U1)
10. if ( u1 $\neq \phi$ && v $\in$pn )
11.     return 1+ dist[cn_map.get(u1)][ cn_map.get(v)];
// U --- V1 – v:
12. if ( v1 $\neq \phi$ && u $\in$pn )
13.   return 1+ dist[cn_map.get(u)][ cn_map.get(v1)];
// u - U1 --- V1 – v:
14. if ( u1 $\neq \phi$ && v1 $\neq \phi$ )
15.  return 2+ dist[cn_map.get(u1)][ cn_map.get(v1)];
16. if (u1 = $\phi$) u2 = $N_u^2 \cap$ pn
17. if (v1 = $\phi$) v2 = $N_v^2 \cap$ pn
// u - u1 - U2 --- V1 – v
18. if (u2 $\neq \phi$ && v1 $\neq \phi$)
19.     return 3+ dist[cn_map.get(u2)][ cn_map.get(v1)];
// u - U1 --- V2 - v1 - v
20. if (u1 $\neq \phi$ && v2 $\neq \phi$)
21.    return 3+ dist[cn_map.get(u1)][ cn_map.get(v2)];
// u - u1 - U2 --- V2 - v1 - v
22. if (u2 $\neq \phi$ && v2 $\neq \phi$)
23.    return 4+ dist[cn_map.get(u2)][ cn_map.get(v2)];
```

The theorem and corollary are self-explaining. Although as the threshold goes to zero (i.e. choosing more and more popular nodes and bridge nodes) the estimate distance goes to real distance overall, that approaching is by no means monotonous because the graph is of a discrete nature. Furthermore, when the core graph grows the computing cost grows as well.

Our algorithms are not just simple, effective, and easy to implement, their main appeal however is their practicality due to their mostly straightforward ways of parallelization using the MapReduce framework in a cloud computing environment. Due to the sheer sizes of large social graphs, being able to be parallelized in a distributed cloud computation for the web analytics has become a mandate.

Paralelization of offline core construction. The first step is to find the popular nodes (Algorithm 3.1). Given a threshold, the distributed data and computers can independently check the degrees and perform lines 3-4, 6-7; line 5 however needs a global computer to create unique ids for each popular node. Computing the candidates and selecting bridge nodes from them (Algorithm 3.2) is more difficult to parallelize. We can achieve it by an algorithm similar to parallel merging. After the candidates and their popular neighbor arrays *p* are distributed, the selection of bridge nodes (lines 8-9) can be done in parallel as well as cleaning the lists of the core nodes to remove non-core nodes in the lists. For distance matrix computation (Algorithm 3.3), each computing node just independently performs a BFS with its assigned starting node on the core lists which is in the memory of a computing node due to its much smaller size than the input graph.

Paralelization of online query evaluation. It is straightforward to parallelize the computation of the neighbor sets N_u and N_v, and neighbor-of-neighbor set N_u^2, N_v^2 (Algorithm 3.4). So is the local accurate distance computation (Lines 1-5 in Algorithm 3.5) as long as the query node IDs are assigned to the corresponding computing nodes. Again, since the number of core nodes is small, the core node IDs with the mapping function cn_map are broadcast to each computing nodes. Lines 6-21 are inherently parallel as long as the pre-computed core distance matrix is distributed or handled by a global distance oracle on the core net.

V. SIMULATION RESULTS

We used an 8-core Linux server with 64G memory in our simulations. The CPU speed is 3.5 GHz. We implement the algorithms using Java 1.6.

Our first data set is provided by Stanford SNAP [16]. Data set Ego-Facebook contains undirected, unweighted Facebook graph obtained through survey participants. The graph has 4,040 nodes and 88,234 edges. The largest degree of the nodes is 293 and average degree is 41. The input is stored in 10 files, with a total of 170,174 lines. There are 81 singleton nodes and 8 2-node components. Other larger components and the statistics are shown in Table I. There are total 94 components, with the largest one containing 3,927 nodes. Its diameter is 17, meaning that some of the nodes in it are quite far away even if most are close to each other. The pair-wise actual shortest distances among all the nodes were calculated for statistical purposes. Their frequencies are shown in Figure 2. From it we can see that most of the pairs have a distance shorter than 12. In fact, a majority of them have hop distances around 4-8.

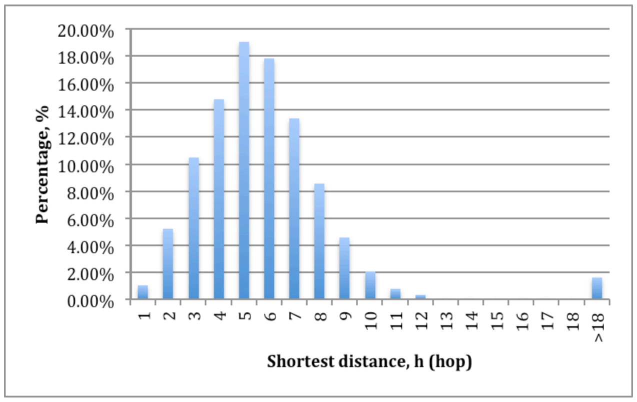


Figure 2, Percentage of pairs of vertices with h-hop distance between themselves. In the graph, x-axis represents the hop distance and y-axis shows the percentage of such hop-distance among all possible pairs.

Table I, Network characteristics, number of nodes and diameter, of larger components.

Component ID	1	2	3	4	5
Number of Nodes	3927	6	4	3	3
Diameter	17	2	1	2	1

Our core net algorithm was evaluated for different factors. We use factors to determine the thresholds for various sizes of input graphs. Threshold ϑ = factor*avg_degree. Runtime of our core-net algorithm with different factors are presented in Figure 3. Gp and Gc times are the times spent in computing the popular graph and the core graph respectively. When the threshold goes to zero, the time increases fast. When the factors are greater than two, our algorithm is very fast.

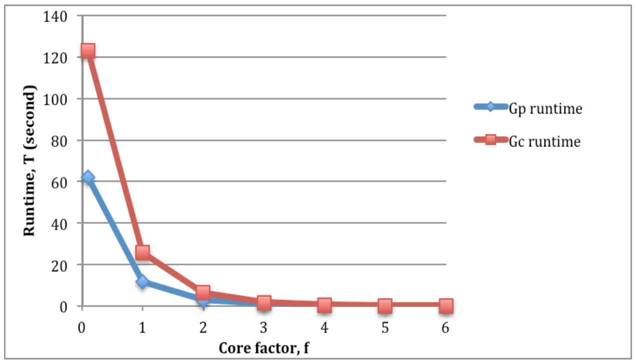


Figure 3: Runtime of our core-net algorithm with different factors. Factor, f, is defined as the ratio between degree threshold and average node degree.

We also evaluated the accuracy of our core-net algorithm. The accuracy is measured by average error rate and miss rate. The average error rate is define as

$$\sum_{i=1}^{k} \frac{(\delta_{\vartheta}{}^{i} - \delta^{i})}{\delta^{i}} / k$$

For each query i, total k queries, $\delta_{\vartheta}{}^{i}$, and δ^{i} are our estimated and true distance of i^th query. Since our evaluation algorithm 3.4 is based on the direct exact calculation of $\delta(u, v)$ using N^1 or N^2 sets, or an estimate if they are within N^2 of any core node. However, if a connected pair is far from each other (more than 4) or from the core net (more than 2), then a "miss" happens. The miss rate is defined as the number of misses divided by total queries. Note that, when we compute the error rate, the missed queries are dropped because their distances are counted as ∞.

The error rates and miss rates of different factors are shown in Figure 4. We use 1000 randomly generated queries (k =1000). Although the error rates vary due to discrete nature of graphs, the miss rates do decrease monotonously with core factor.

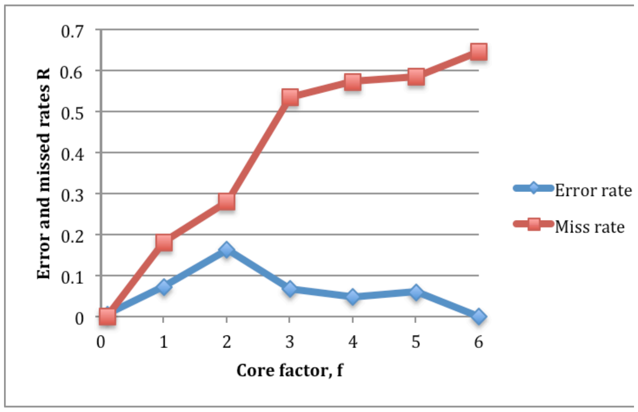


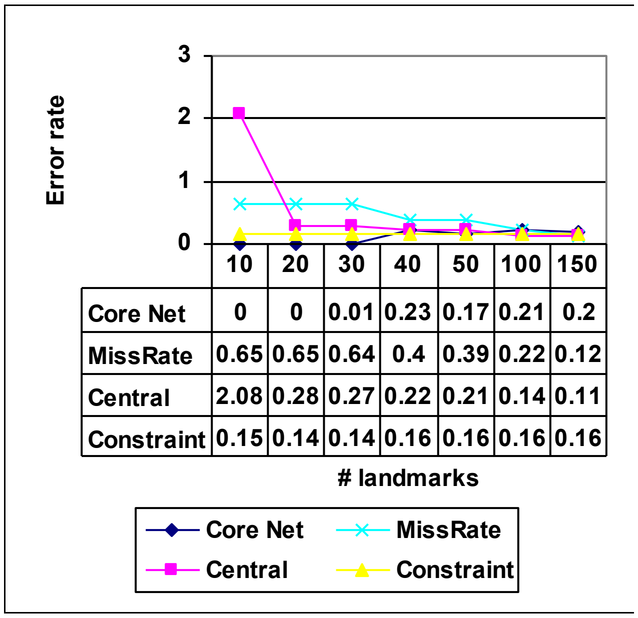
Figure 4: Error rates and miss rates of different factors.

The characteristics of the core net formed with different factors are listed in Table II. These include number of core nodes (#core), number of bridge node candidates (#cand.), number of bridge nodes (#bn), number of disconnected components (#comp), and list of components in the core graphs after our Core-net algorithm has been performed. The components 0 corresponding to the computation of original graph is already shown in Table I.

It can be observed that the largest component (component No. 1) basically contains all of the nodes. Each of the other smaller components contains a couple of nodes each. When the input data grows, the core graph also grows. The internal Core-net computation (in Fig. 4 and Table II) becomes infeasible. We need to compute the core and then write the pre-computed pairwise distances of the core nodes into disk and later retrieve for queries. We implemented the constraint and central algorithms in [10] and compared with our external Core Net. We change the number of landmarks and compare with the offline construction times and average error rate. The results are shown in Figures 5 and 6. Our algorithm takes a fraction of times of Central and Constraint with a similar or better accuracy.

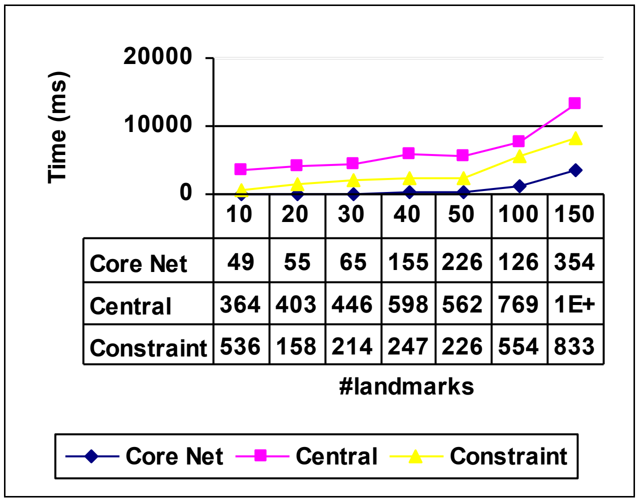
Table II, Core-Net Characteristics with different factors. The "Components" column lists the sizes and diameters of the connected components.

Factor	#core	#cand.	#bn	#comp	Components
0	3959	0	0	13	
1	1335	1558	187	2	(1503, 10) (19, 2)
2	627	1356	87	1	(714, 10)
3	312	877	21	2	(331,6) (2,1)
4	149	676	4	1	(153,5)
5	21	698	3	1	(24,5)
6	3	235	0	2	(1,0) (2,1)



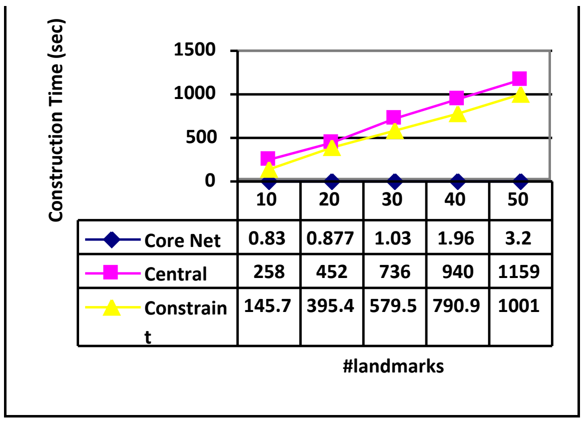
	10	20	30	40	50	100	150
Core Net	0	0	0.01	0.23	0.17	0.21	0.2
MissRate	0.65	0.65	0.64	0.4	0.39	0.22	0.12
Central	2.08	0.28	0.27	0.22	0.21	0.14	0.11
Constraint	0.15	0.14	0.14	0.16	0.16	0.16	0.16

Figure 6, Error rates and missed rates for data set 1.



	10	20	30	40	50	100	150
Core Net	49	55	65	155	226	126	354
Central	364	403	446	598	562	769	1E+
Constraint	536	158	214	247	226	554	833

Figure 5, Offline construction time for data set 1.



	10	20	30	40	50
Core Net	0.83	0.877	1.03	1.96	3.2
Central	258	452	736	940	1159
Constraint	145.7	395.4	579.5	790.9	1001

Figure 7, Offline construction time for data set 2.

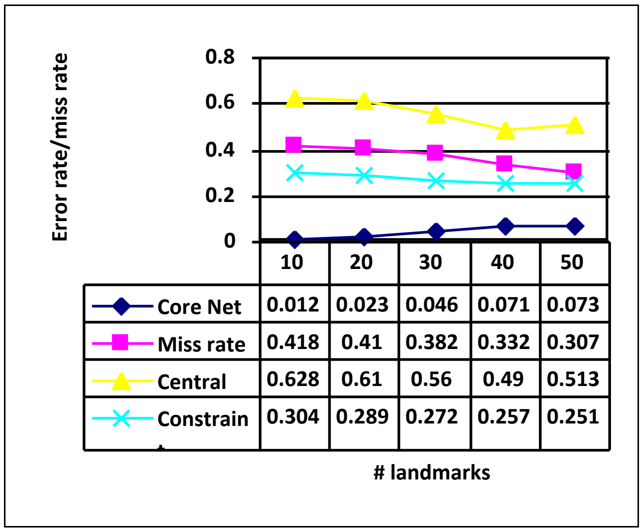
Our second real data set was from [17, 18], called Facebook Social Graph - Breadth First Search data set "bfs-1-socialgraph-release. The file's size is 2.78GB with 1,189,767 lines. The data set was collected in April, 2009 through data scrapping from Facebook.

The graph has 61,876,633 nodes and 170,069,566 edges. Since the nodes with IDs above 1,189,767 do not connect among themselves, we purged these larger IDs from the link lists. This graph is connected. We run external Core Net, Central, and Constraint on this larger data set. Runtimes are shown in Figure 7. From Figure 7, we can see that Core Net is orders faster than the competitors. The error rates of the three competing schemes are shown in Figure 8, which also shows the miss rate of the Core Net scheme. The error rates of the Core Net are 4-10 times lower than the other two schemes.

VI. Concluding Remarks

In this work, we have proposed a shortest-path query algorithm based on core net, which consists popular nodes that have large node degrees, bridge nodes that connect some of these popular nodes that are otherwise disconnected, and the edges connecting these nodes. After a pre-processing phase, the well-known triangulation technique is used to efficiently compute the shortest distance between any pair of nodes. Compared to other related works, our scheme scales well and provides rather accurate approximations for different queries.

In future work, we will investigate different techniques to further improve accuracy and reduce miss rate. In addition, experimentation of our techniques on other large social network data sets are of our interest as well.



	10	20	30	40	50
◆ Core Net	0.012	0.023	0.046	0.071	0.073
■ Miss rate	0.418	0.41	0.382	0.332	0.307
▲ Central	0.628	0.61	0.56	0.49	0.513
✕ Constrain	0.304	0.289	0.272	0.257	0.251

Figure 8, Error rates for data set 2. Core Net, Central and Constraint schemes are shown. The miss rates of the Core Net scheme are presented as well.

REFERENCES

[1] Dijkstra, E.W., *A note on two problems in connexion with graphs.* Numerische Mathematik, 1959. **1**(1): p. 269–271.

[2] Bellman, R., *On a routing problem.* Quarterly of Applied Mathematics 1958(16): p. 87–90.

[3] Cormen, T.H., et al., *Introduction to Algorithms*. Third ed. 2009: The MIT Press.

[4] Floyd, R.W., *Algorithm 97: Shortest Path.* Communications of the ACM, 1962. **5**(6): p. 345.

[5] Cohen, E., et al., *Reachability and distance queries via 2-hop labels.* SIAM J. Comput., 2003. **32**(5): p. 1338–1355.

[6] Jin, R., et al., *3-hop: a high-compression indexing scheme for reachability query*, in *SIGMOD*. 2009.

[7] Qiao, M., et al. *Approximate Shortest Distance Computing: A Query-Dependent Local Landmark Scheme*. in *ICDE*. 2012. Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012.

[8] Wei, F., *TEDI: Efficient Shortest Path Query Answering on Graphs.* Graph Data Management: Techniques and Applications, 2011: p. 214-238.

[9] Jin, R., et al., *A highway-centric labeling approach for answering distance queries on large sparse graphs*, in *SIGMOD*. 2012. p. 445-456.

[10] Potamias, M., et al., *Fast shortest path distance estimation in large networks*, in *Proceedings of the 18th ACM conference on Information and knowledge management*. 2009: Hong Kong, China. p. 867-876.

[11] Dean, J. and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*

[12] Dean, J. and S. Ghemawat, *MapReduce: a flexible data processing tool.* Communications of the ACM, 2010). **53**(1): p. 72-77.

[13] Dittrich, J. and J.-A. Quiané-Ruiz, *Efficient Big Data Processing in Hadoop MapReduce.* PVLDB, 2012. **5**(12): p. 2014-2015.

[14] Agrawal, D., S. Das, and A.E. Abbadi, *Big Data and Cloud Computing: Current State and Future Opportunities* in *EDBT*. 2011: Uppsala, Sweden. p. 530-533.

[15] Backstrom, L., et al., *Four degrees of separation*, in *WebSci*. 2012: Evanston, IL, USA. p. 33-42.

[16] SNAP. 2009; Available from: http://snap.stanford.edu/data/.

[17] Gjoka, M.; *Sampling Online Social Networks,* available from: http://odysseas.calit2.uci.edu/doku.php/public:online_social_networks.

[18] Gjoka, M., et al., *Practical Recommendations on Crawling Online Social Networks.* IEEE J. Sel. Areas Commun. on Measurement of Internet Topologies, 2011. **29**(9): p. 1872-1892.