Fast and Distributed Approximation of Shortest Paths in Large-scale Complex Networks

***Abstract*—Finding near-shortest path in a fast and distributed way for large complex networks are challenging. Existing work for approximate shortest path are neither too slow or require too much storage and time overhead for networks with millions of edges. In this paper, we study methods for finding approximate results, where we hope to generate fast and distributed algorithms for finding near-shortest path based on graph embeddings. Specifically, the algorithm pre-computes and stores a digest of graph topology in each node, so that any point to point path query can be answered by finding a routing procedure from the source to destination, approximately but with high accuracy. We evaluate our algorithm on networks with different kinds of topology, such as scale free networks, random networks and sensor networks.**

I. Introduction

Introduced by the famous experiment conducted by Stanley Milgram, the small-world phenomenon, that most vertices can be reached from each other by a small number of hops, is found to be an fundamental characteristic in most real world networks. Although this property assure us the existence of such short paths between vertices in small world networks, due to the ever increasing number of vertices, this seemingly straightforward operation has become challenging. For example, for graph derived from LiveJournal social networks with 4 million vertices and 50 million edges [1], online BFS/Dijkstra traversals will take around a minute to finish on a single commercial computer, even for A* search which can significantly reduce the number of vertices visited, it still require to access an average of 20K vertices for each query [2].

Various networks which are commonly used in modern information technologies to model real-world phenomenon, such as online social networks (e.g., LinkedIn or Facebook), biological networks, road networks, etc have millions of vertices and billions of edges that even a cluster of machines can hardly deal with them. For example, in Figure 1, we plot the runtime of BFS running on a Amazon EC2 cluster with 10 machines for networks of various size. Networks are derived from Facebook, Twitter and LiveJournal respectively [3], [1]. We can see that even for a cluster of machines, a single BFS traversal takes around 3.5 seconds, and the time required for the traversal increases quickly as the scale of the network increases.

In this review, we are focusing on one of the most common variants of the shortest path/distance problem, to find a point-to-point shortest path/distance in a graph. This operation serves as the building block for many other tasks, and has many applications. For example, a natural application for road network is providing driving directions [4]. In social networks, such applications include social sensitive search [5], analyze influential people [6]. Estimating minimum round trip time between hosts without direct measurement is another application in technology networks [7].

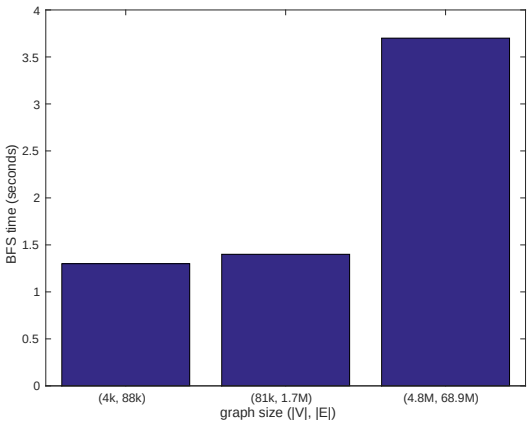


Fig. 1. This figure shows the runtime of a single BFS on a Amazon EC2 cluster with 10 machines for graphs of various size.

Usually in these applications, point-to-point shortest path/distance problem will be solved repeatedly for different source/target vertex pairs. A straight forward solution to this problem is to apply online traversal like BFS/Dijkstra for each query. But clearly, this approach is very inefficient due to the scalability issue discussed above. Due to the nature of applications' needs, most point-to-point shortest path/distance algorithms will use preprocessing in order to speed up the following online searching operations. One obvious way for preprocessing is to perform all-pair shortest paths operation such as Floyd-Warshall and store the results for each pair of vertices. When a query arrives, only constance time is required to return the results. However, this approach is also unacceptable due to the $O(V^3)$ runtime and $O(V^2)$ storage space. To balance between preprocessing and online query has become a key challenge in this research area [2], [8], [9], [10], [11].

The point-to-point shortest path/distance problem has already been well studied in road networks, distance queries can be answered in less than a microsecond for the complete USA road network [4]. Most of the methods on road networks take advantage of the spatial and planar-like properties a road network has [12]. Especially, Abraham et al. discovered that several fastest distance computation algorithms actually rely on a graph measure called highway dimension. However, these

existing algorithms for road networks can hardly be applied to complex networks, due to the fact that the structure of most complex networks is nowhere near planar, and they usually do not have low highway dimensions like road networks. Actually, the locality of real-world complex networks is very poor, since the number of edges crossing clusters of the network is very large [13].

To further increase the scalability to handle large-scale complex networks, approximate methods have also been studied. Instead of returning an exact shortest path/distance between query vertex pairs, a near optimal path/distance is returned. Compared to online traversals for finding exact shortest path in seconds, it usually takes milliseconds for an algorithm to return an approximated shortest path [12], [8], and microseconds to approximate shortest path distance [14].

A. Problem Formulation

Our problem formulation is as follows: we investigate how to find exact or approximate point-to-point shortest paths/distance between any two vertices in extremely large networks. By extremely large, we are concerned with networks of million vertices, but in practice, such networks may even contain hundreds of millions of vertices. We are focusing on a specific kind of networks which is very common in real-world networks called complex network. They usually do not exhibit the same properties as simple ones such as road networks. For example, it is well known that these networks have their vertices' degrees conforming to the power law, and that vertices have relatively short distances between themselves. In most cases, the locality of these networks are very poor.

B. Contributions

In this review, we provide a comprehensive state-of-the-art study in algorithms on point-to-point shortest path/distance problems in large-scale complex networks. Existing algorithms usually contain two phases, distance calculation and online searching. After preprocessing, algorithms can return exact or estimated shortest path distance of two given vertices that can be used in applications where the underlying path is not important. The later one return an actual exact or approximate shortest path, and is essential for several applications require path information. Usually, online searching needs distance estimates to avoid large number of visited vertices, which is the main benchmark of these kind of algorithms.

C. Paper organization

The remaining of this review is organized as follows. In Section **??**, we present several fundamental preprocessing methods being used by most of the algorithms and describes algorithms that calculate exact or estimated shortest path distance for any given vertex pair. We discuss online searching algorithms that finding actual paths in Section **??**. We conclude the whole review in Section VII.

II. Motivational Case Study

In this section, we first present a motivational case study, showing that there are critical differences between planar graphs and social networks. In particular, we show the performance of an algorithm applied to both planar graphs and social network graphs.

A. Preliminaries

In our problem, we consider a graph $G = (V, E)$, which represents a graph of vertex set V and edge set E. For a source node s and destination node t, we are interested in finding a path p in the graph, which is an ordered sequence of n vertices, such that,

$$p = (v_0, v_1, ..., v_{n-1})$$

where $(v_i, v_{i+1}) \in E$. We can write that $|p| = n$ to denote that length of p is n. Given a pair of source and destination nodes, there may be many paths connecting them, which we denote as P. We can then define the distance between s and t as:

$$distance(s, t) = min_{p \in P(s,t)} |p|$$

Next we define path approximation ratio for a possibly non-optimal path q as:

$$ratio(q) = \frac{|q|}{distance(s, t)}$$

Note that for any path q, the ratio defined here must be at least 1. If $ratio(q)$ is equal to 1, this path must be an optimal path by definition.

Based on this definition, suppose that given an approximation scheme that returns a path for every pair of nodes s and t (denoted as $p(s, t)$), we can easily define the average approximation ratio of all node pairs for a graph $G = (V, E)$ with n nodes as:

$$averageratio(G) = \frac{1}{n(n-1)} \sum_{s \in V, t \in V} ratio(p(s, t))$$

B. Existing Graph Embedding Approaches

Previous methods based on graph embeddings have been developed under different names, such as landmark-based routing, coordinate based routing, among others, for planar graphs such as wireless sensor networks. To find a path between s and t, this algorithm takes a two-staged approach: a pre-computation step that assigns each node with a coordinate vector, which is composed of distances from all vertices to a few selected landmark nodes, and a routing step that uses this pre-computed data to find paths between nodes with a complexity that is only proportional to the path distance. The overall procedure is shown in Figure 2, where a configuration of three landmarks and six vertices are illustrated.

The precomputation step, a set of landmarks is first chosen in the graph, based on the relative distances of nodes.

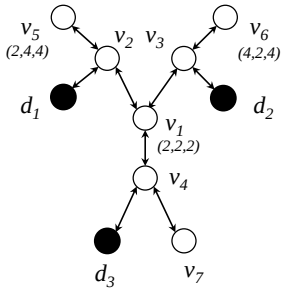


Fig. 3. This figure shows an example of coordinate distance may not represent real distance of two nodes.

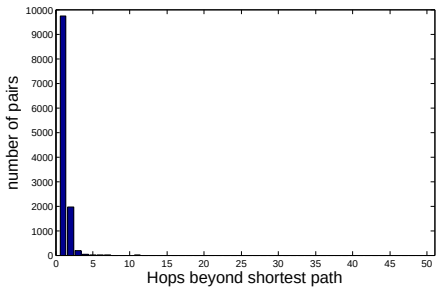


Fig. 4. This figure shows distribution of hop difference between path found for the sensor network.

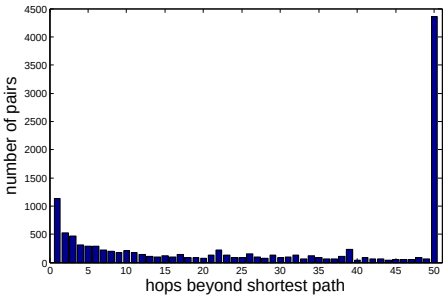
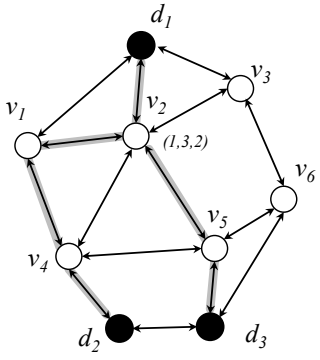


Fig. 5. This figure shows distribution of hop difference between path found for a social network.



Vertex	Coordinates
d_1	(0,3,3)
d_2	(3,0,1)
d_3	(3,1,0)
v_1	(1,2,3)
v_2	(1,2,2)
v_3	(1,3,2)
v_4	(2,1,2)
v_5	(2,2,1)
v_6	(2,2,1)

Fig. 2. This figure shows three landmarks, $\{d_1, d_2, d_3\}$, and six additional vertices. The coordinates for each node are shown in the table on the right.

Recent studies have pointed out that choosing the best set of landmarks for graph embeddings as NP-Hard, therefore, we assume they are chosen based on simple heuristics such as degrees of nodes. Next, for every node that is not a landmark, we compute the shortest path that connects each landmark to this node, by using the breadth-first search method. For a total of m landmarks, there will be a total of m breadth-first search procedures.

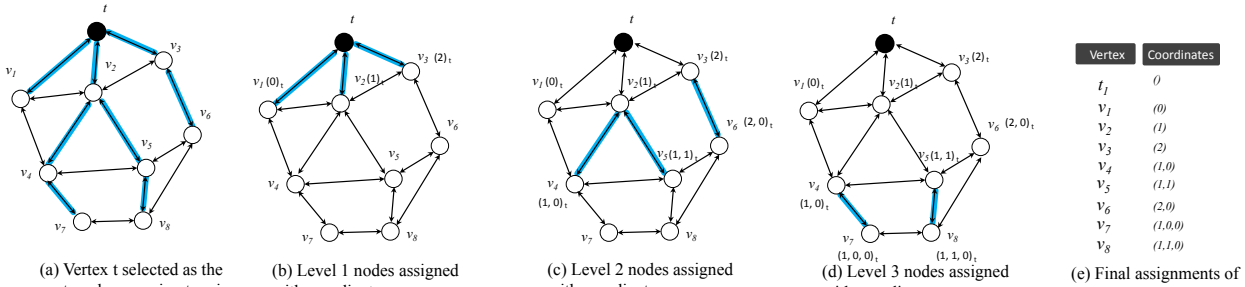
To perform routing and find paths, the Euclidean distance of the vectors are used to find the next hop based on the pre-computed coordinates. Note that here, the coordinates only contain distance information. Usually this works pretty well with relatively simple topologies. The key advantage is that the storage overhead of maintaining coordinate vectors is only proportional to the number of landmarks, which is usually sufficiently small compared to the total number of nodes.

To measure the quality of returned paths, we calculate the average approximation ratio of this figure. This is done by performing routing for every two pairs of nodes. For example, from v_1 to v_3, we can find that the distance between them $d(v_1, v_3) = \sqrt{5}$. Among the neighbors of v_1, v_2 has the shortest distance to v_3 as 1. Hence, v_2 is chosen as the next hop. Therefore, the path found is v_1, v_2, v_3, which is optimal. One can verify that for this graph, the average approximation ratio is precisely 1, meaning that all paths found are optimal.

C. Comparison of Graph Embedding Performance on Complex Network vs Network with simple topology

Although the coordinate based algorithm works well for networks with a simple topology, we find that it has poor performance for graphs with more complex topologies, such as social networks where a large range of degree distributions and long-distance edges dominate. The root cause is that the algorithm only encodes hop counts to a few landmarks, therefore, in many cases, the coordinates can no longer correctly represent the real distances of two nodes. Fig. 3 illustrates an example. To find a path from v_2 to v_6, using a greedy search will encounter problems between v_5 and v_1. In fact, both $distance(v_5, v_6)$ and $distance(v_1, v_6)$ are 2, but intuitively, v_1 clearly has a shorter distance to v_6.

We next show a numerical study to illustrate the performance difference of coordinates applied to simple topologies and complex topologies. The first network is a social network which has power law degree distribution and relatively short diameter. The other network is a randomly generated network which has a topology similar to sensor networks. Each node has similar degree, following by a normal distribution. There are no long distance edges, nodes are only connected to nearby neighbors, so this network has a relatively long diameter. Both nodes have around $4,000$ nodes and around $176,000$ edges. Due the size of the graph, there are approximately $4000 * 4000$ different ordered node pairs. We only randomly choose $10,000$ pairs out of them and using coordinate routing to find path between each pair. Since the average shortest path of this two graph are different (approximate average length of shortest path for the social network is 3, while it is 40 for the generated random sensor network), instead of path approximation ratio, we show the distribution of length of path found by graph embedding approach minus shortest path length in Fig. 4 and Fig. 5. From these two figure, we can see that compared to the network with simple topology, coordinate routing algorithm really performs poorly on complex networks. Actually. the average approximation ratio of randomly chosen $10,000$ pairs for the social network is 37.983 compared to 1.014 for random sensor network.



(a) Vertex t selected as the root, and a spanning tree is built using the BFS search

(b) Level 1 nodes assigned with coordinates

(c) Level 2 nodes assigned with coordinates

(d) Level 3 nodes assigned with coordinates

(e) Final assignments of coordinates

Fig. 6. This figure shows forming a tree structure and then assigning coordinates to nodes based on their relative orderings in the tree.

III. Algorithm Design

In this section, we describe the design of our algorithm. We first present the overall algorithm. Then we describe a few optimizations to improve its performance.

A. Algorithm Overview

Our algorithm design is inspired by the previous work in that it is also built on top of a landmark-based framework. Specifically, it consists of the precomputation stage and the path-finding stage. We next describe these two stages separately.

1) Precomputation: The precomputation step involves choosing a few nodes as landmarks, and then computing for every node a "topology digest" based on this landmark. These digests will be used in the approximate path finding step later. The preprocessing, illustrated in Figure 6, works as follows:

Root Selection: In this phase, we select a few nodes as roots, denoted as t_0, t_1, etc. These nodes will be used to construct trees. Usually, we select roots based on their degrees: those nodes that have the highest degrees will be chosen as the roots. On Figure 6, however, we intentionally select the root as t for illustration purpose, even though it does not have the highest degree. Specifically, Figure 6(a) shows that a tree is constructed with the node t as the root to all other nodes in the graph.

Coordinate Assignments: In this phase, we perform a BFS operation from the root to every other nodes in the tree structure. Therefore, for each node in the tree, we can find its children nodes, based on which we can assign them with coordinates. Figure 6(b) to Figure 6(d) show the children node coordinates. Specifically, one node assigns coordinates to its children nodes as follows: suppose this node's current coordinate is (x) and it has k children nodes. Each children node is assigned with $(x, 0)$, $(x, 1)$, up to $(x, k-1)$. Note that such assignments have to propagate from level to level: first at the root, then propagates until the farthest leaves are reached. The coordinates assigned to each node is shown in Figure 6(e).

2) Shortest Path Approximation: In this stage, the algorithm generates approximations for shortest paths between a pair of nodes, s and t. The goal is to get the a small error, that is, the number of hops should be similar to the optimal path. Based on the pre-computed coordinates, the algorithm performs the following steps to generate such a path:

1) Suppose that the path digests for node s and t are vectors $< a >$ and $< b >$, respectively, calculate their common ancestor $< c >$.
2) Starting from the source node s, select the next hop as its parent until the last node of the common ancestor $< c >$ is reached.
3) Starting from the last node of $< c >$, select the next hop using $< b >$'s coordinates until the destination node t is reached.
4) Return the path found in step 3.

Clearly, for many cases, such a path is not optimal: it will have to go back to a root before making progress to the destination node. For example, in Figure 6, we observe that from node v_4 to node v_6, a total of 4 hops is taken (v_4,v_1,t,v_3,v_6), as opposed to the optimal path (v_4,v_5,v_6), which is 2 hops. Next we describe a few optimizations to dramatically improve the performance of this approach.

B. Design Optimizations

In this section we explain our improvements to the original algorithm to obtain better approximations. As a first step, we try to combine it with the coordinate based approach to find existing shortcuts within the paths. In the second step, we develop an ensemble of trees, and develop a hybrid method to calculate the distance, hence reducing the total path lengths. These improvements provide considerable improvements in terms of the approximation quality, as we will show in the experimental evaluation.

1) Shortcutting through Greedy Search on Weighted Coordinates: We first describe the improvement called path shortcutting. The idea is that we try to go through the neighbors that directly lead to the destination node instead of going back to the root of the tree structures. The key idea is shown in Figure 7, where we hope to obtain a path from s to t. Using the tree structure alone, we will end up with a path on the left through the root. On the other hand, a shorter path exists along the "leaves" of the tree through u. Therefore, our goal

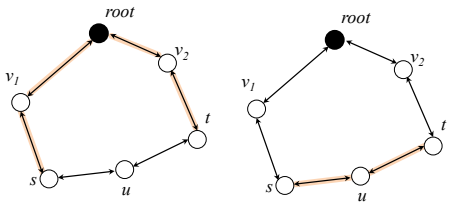


Fig. 7. This figure shows how to use shortcuts to reduce the length of found paths.

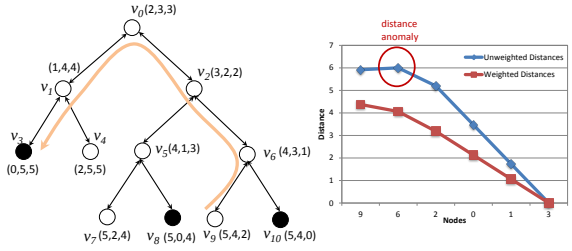


Fig. 8. This figure shows how weighted coordinates avoid local maximum along the path.

is to search for such paths as shortcuttings, and return them whenever possible.

The basis of this design is that we use a modified version of the graph embedding based coordinates, called "weighted coordinates" as the key building block. The reason that we do not choose the original design of coordinates is that usually, coordinates do not reflect the real topologies in a perfect manner. An example of this is observed in Figure 8, which shows a case study where we would like to find a path from v_9 to v_3, illustrated with the orange line. The distances of each node to the destination are shown on the right half of this figure. Note that for the first hop from v_9 to v_6, the distance to the destination actually *increases*, i.e., there is a local maximum reached at this hop. One solution to solve this problem is by observing that each dimension of the coordinate vectors should not be treated equally to contribute to the distance calculation. Instead, those coordinates that change *the most* from the source to the destination should be given a larger weight, because they provide more information on the relative changes of path distances. In Figure 8, for example, the first dimension of the coordinate vectors changes from 5 to 0, which is the largest. Therefore, it should be given the highest weight.

Mathematically, we assign weights to dimensions based on the delta of the source and destination coordinates. Suppose that the source node s has a vector of $(s_0, s_1, ..., s_{k-1})$, and the destination node t has a vector of $(t_0, t_1, ..., t_{k-1})$, the weight coefficients w_i $(i \in [0, k-1])$ are calculated as follows:

$$w_i = \frac{|s_i - t_i|}{\sum_{j=0}^{k-1} |s_j - t_j|}$$

Note that weight coefficients are only computed once at the source, and do not change along the path. The distance between an intermediate node u to t is then calculated as:

$$d(u,t) = \sqrt{\sum_{j=0}^{k-1} w_j \times (u_j - t_j)^2}$$

The results for the path in Figure 8 are plotted as comparison. Observe that the local maximum no longer exists in the path.

2) Tree Ensembles: In this section we describe our second optimization to improve the performance, by using multiple trees instead of a single tree. We call this approach "tree ensembles". Recall that each tree is a collection of paths that connects the root with every other node in the graph without loops. Our improvement then constructs multiple trees, such that every node is assigned with more than one set of encodings. To build these trees, the algorithm starts breadth-first searches from different roots. To avoid reusing the same edges as much as possible, different BFS procedures will follow a different order of inserting nodes into the BFS search queue, so that those nodes that are used in one tree will be visited in a different order in a second tree.

After multiple trees are constructed, each node then has more than one encodings to use to lead to the destination nodes. In the algorithm, it will always pick the nearest one with the shortest distance such that only one tree is used.

3) Putting Things Together: We next describe how to put the two optimizations together. Specifically, every time a new path request is received, a packet delivery will switch between the "greedy mode" and "tree mode". The key idea is that if the greedy mode is making smooth progress, we can try to find shortcuts with this approach. On the other hand, if local maximum areas are reached, the packet will now switch to the tree mode to route around the local maximum areas such that it will always be delivered to the destination. Therefore, the delivery ratio is always 100% as long as the graph is not partitioned.

IV. System Implementation

A. Implementation Overview

We implemented all methods including the classical Dijkstra's online shortest path algorithm and our proposed methods within the SNAP framework, a recently proposed high-performance graph processing engine for handling graph related computational tasks. The reason that we choose SNAP is that it has great ability to store and process large scale graphs in a highly efficient manner. We next give an overview of the SNAP architecture.

B. Implementation Details

V. System Evaluation

A. Evaluation Overview

B. Datasets

C. Methodology

D. Evaluation Results

VI. Related Work

In this section, we describe related work in three parts: first, we briefly survey existing methods for calculating shortest paths. Second, we describe methods on embeddings for graphs. Finally, we describe the applications of this work.

Existing Methods: Existing work on calculating shortest distances can be classified into two categories: exact approaches and approximate approaches. The exact approach such as the Dijkstra algorithm has a computing complexity of $O(n^2)$ in general, and therefore not scalable for all-pair shortest distance calculations in large graphs. Another algorithm by Floyd-Warshall leverages dynamic programming to solve the all-pairs shortest paths in $O(n^3)$. On the approximate algorithm side, recent state of the art algorithms also tried to combine bidirectional Dijkstra with A* algorithms to prune the search space. However, such algorithms are still too slow for large-scale graphs. Another interesting trend of research is to provide estimations on the path lengths rather than finding the actual paths. Such works typically avoid any kind of online Dijkstra/BFS traversals, but are considerably different from our work.

Our work falls into the category of trying to find approximate paths through preprocessing. Our goal is to preprocess a graph so that point-to-point queries can be answered approximately and at the computational overhead proportional to the path itself, not to the size of the network. Key to the performance of such algorithms is the quality of found paths, i.e., the length to the optimal path. Previous theoretical research tends to get much more relaxed bounds than ours, as our empirical studies show that bounds as low as 1.1 can be obtained in real-world graphs. In a small-world network, such as the graph described in the evaluation section, the exact distance is only guaranteed to lie within the interval. Therefore, the generated path is at most one hop longer.

Graph Embedding Methods: Our work on using the tree structure is closely related to general embedding methods, which aim to transform graphs into alternative representations. By doing so, we can obtain significant speed-ups by embedding nodes into a different space and using a more efficient distance functions. Several landmark based approaches have been proposed in the literature. For example, Kleinberg discussed the problem of approximating network distances in real networks using landmarks and their theoretical implications. Our techniques are unique in the sense that trees are used as opposed to coordinates only. On another topic, it has been widely started to perform routing in geographic networks. Our work is different since we focus on graphs that exhibit complex social network behavior.

Applications:

VII. Conclusions

References

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 44–54. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150412

[2] 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*, ser. CIKM '09. New York, NY, USA: ACM, 2009, pp. 867–876. [Online]. Available: http://doi.acm.org/10.1145/1645953.1646063

[3] J. Mcauley and J. Leskovec, "Discovering social circles in ego networks," *ACM Trans. Knowl. Discov. Data*, vol. 8, no. 1, pp. 4:1–4:28, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2556612

[4] 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*, ser. SEA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 230–241. [Online]. Available: http://dl.acm.org/citation.cfm?id=2008623.2008645

[5] 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*, ser. CIKM '07. New York, NY, USA: ACM, 2007, pp. 563–572. [Online]. Available: http://doi.acm.org/10.1145/1321440.1321520

[6] 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*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/956750.956769

[7] L. Tang and M. Crovella, "Virtual landmarks for the internet," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '03. New York, NY, USA: ACM, 2003, pp. 143–152. [Online]. Available: http://doi.acm.org/10.1145/948205.948223

[8] 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*. ACM, 2011, pp. 1785–1794.

[9] 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*, ser. EDBT '12. New York, NY, USA: ACM, 2012, pp. 144–155. [Online]. Available: http://doi.acm.org/10.1145/2247596.2247614

[10] 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*, vol. 26, no. 1, pp. 55–68, Jan 2014.

[11] 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*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 445–456. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213887

[12] 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*, ser. CIKM '10. New York, NY, USA: ACM, 2010, pp. 499–508. [Online]. Available: http://doi.acm.org/10.1145/1871437.1871503

[13] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[14] 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*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 349–360. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465315