Research Article

Shortest Paths Based Web Service Selection in Internet of Things

Xiangdong Yin and Jie Yang

Department of Computer and Communication Engineering, Hunan University of Science and Engineering, Yongzhou 425100, China

Correspondence should be addressed to Xiangdong Yin; yinxiangdongx@163.com

Received 22 March 2014; Accepted 3 May 2014; Published 26 June 2014

Academic Editor: Junjie Chen

Copyright © 2014 X. Yin and J. Yang. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The connecting of things to the Internet makes it possible for smart things to access all kinds of Web services. However, smart things are energy-limited, and suitable selection of Web services will consume less resources. In this paper, we study the problem of selecting some Web service from the candidate set. We formulate this selection of Web services for smart things as single-source many-target shortest path problem. We design algorithms based on the Dijkstra and breadth-first search algorithms, propose an efficient pruning algorithm for breadth-first search, and analyze their performance of number of iterations and I/O cost. Our empirical evaluation on real-life graphs shows that our pruning algorithm is more efficient than the breadth-first search algorithm.

1. Introduction

Recently, development of RFID, sensor, and networking technologies make it possible for various physical world things connecting to the internet, and people usually call it Internet of Things (IoT). In IoT, more and more devices are getting connected to the Internet, and the next step is to use the World Wide Web and its associated technologies, such as Web services, as a platform for smart things.

In Internet, there are a lot of Web services. These Web services connect with each other and construct a topology of graph. In IoT, all smart things are connecting to the Internet, so they can access all kinds of Web services. However, smart things are usually energy-limited, so the selection of suitable Web service is extremely important. Guinard et al. [1] give an overall view of Web service access in Internet of Things. They propose a process and a suitable system architecture that enables developers and business process designers to dynamically query, select, and use running instances of real-world services.

In this paper, we study the problem of Web services selection in IoT setting; that is, given a smart thing and a graph of Web services, how to select several suitable Web services from the candidate set while providing satisfactory quality of service. If the selected Web services are nearest, the smart thing would take less accessing and waiting time and then consume less power. Here, we formulate the selection

of Web services as single-source many-target shortest path problem.

The shortest path problem can be classified as SSSTSP (single-source single-target shortest path), SSMTSP (single-source many-target shortest paths), APSP (all-pairs shortest paths), and k SP (k shortest paths). The MSSTSP (many-sources single-target shortest paths) is the same as SSMTSP if we reverse all edges in the graph.

1.1. Our Contribution. In this paper, we study the problem of SSMT k SP (single-source many-target k shortest paths) on MapReduce, which is finding k shortest paths from a candidate set for one source node on MapReduce. In our work, we do not care about the exact k paths, but rather the k nearest neighbors of the source node, so the SSMT k SP problem can also be considered as single-source many-target k nearest neighbors problem. The SSMT k SP can be used in recommending friends or ads in social networks, and in searching malls or hotels in road networks.

We design algorithms based on the Dijkstra and breadth-first search algorithms, propose an efficient pruning algorithm for breadth-first search, and analyze their performance of number of iterations and I/O cost. Our empirical evaluation on real-life graphs and Hadoop platform shows that the pruning algorithm is more efficient than the breadth-first search shortest path algorithm.

The rest of the paper is organized as follows. Section 2 gives the background for shortest path for graphs, MapReduce, Dijkstra algorithm, and breadth-first search. The SSMTkSP problem and its corresponding algorithms are presented in Section 3. We analyze the performance of our algorithms in Section 4, show experimental results in Section 5, and review the related work in Section 6. Finally, conclusion and future work is given in Section 7.

2. Background

In this section, we provide some background for shortest path for graphs, MapReduce, Dijkstra algorithm, and breadth-first search.

2.1. Shortest Path for Graphs. We consider a Weighted Directed Graph $G = (V, E)$, where V is the set of nodes, E is the set of edges, and the number of nodes and edges is $|V|$ and $|E|$, respectively. We use D to represent the diameter of G . For an edge $(u, v) \in E$, the weight is denoted with $w_{u,v}$, and u is v 's parent node and v is u 's child node. A path P in a graph G is a sequence of nodes $\langle v_1, v_2, \dots, v_j \rangle$, where $v_i \in V$ ($1 \leq i \leq j$) and $(v_i, v_{i+1}) \in E$ ($1 \leq i < j$). v_1 and v_j are called *source* and *target* of P and linked by P . The length of P between *source* and *target* is the sum of weights of edges in it; that is, $\text{len}(P) = \sum_{i=1}^{j-1} w_{i,i+1}$, and the number of hops from *source* to *target* in P is the number of edges, that is, $h(P) = j - 1$.

Definition 1. A path P in graph G from start s to target t ($s, t \in V$) is a shortest path, if there does not exist any path P' between s and t , such that $\text{len}(P') < \text{len}(P)$.

For the sake of simplifying the presentation, we assume for the rest of the paper that the weight on each edge of G equals 1; that is, $\text{len}(P) = h(P)$. This degrades G to an unweighted graph but does not affect the result of our algorithms.

2.2. MapReduce. MapReduce [2], proposed by Google, is a programming model for processing huge amounts of data in parallel using a large number of commodity machines, and its open-source implementation is Hadoop (<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html>). By automatically handling the lower level issues, such as job distribution, data storage, and fault tolerance, it allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

In the MapReduce-like systems, programs are done iteratively in three phases: *Map*, *Shuffle*, and *Reduce*. In the *Map* phase, they read a collection of values or key/value pairs from input sources in parallel and emit zero or more key/value pairs for each input element by invoking a user defined *Mapper* function. In the *Shuffle* phase, they group together all the Mapper-emitted key/value pairs sharing the same key and output all groups to the next phase. In the *Reduce* phase, they invoke a user defined *Reducer* function for each distinct group independently and in parallel and emit zero or more key/value pairs that will be written on disks or be the input

FIGURE 1: Graph G as our Running Example.

of the next *Map* phase in the following iteration. In our work, we will use two basic primitives: *Map* and *Reduce*.

2.3. Dijkstra Algorithm. Dijkstra algorithm [3] is a classical algorithm for single-source shortest path problem for graphs with positive weights. It computes the shortest paths between the source s and any other node in a graph. The main idea is that starting from s , expand one node that has the shortest path to s ; repeat the expanding process until all nodes are reached from s . In each expanding process, the algorithm expands a nearest neighbor, and it requires that the weight of each edge is positive.

Example 2. Given the graph G in Figure 1, if we choose v_0 as the source, the distances between v_0 and other nodes will be $\{[v_1, 1], [v_2, 1], [v_3, 2], [v_4, 2], [v_6, 2], [v_5, 3]\}$, which can be done in 6 expanding processes.

2.4. Breadth-First Search. To find the shortest paths between the source s and other nodes in graph G , we can conduct a breadth-first search (BFS for short) starting from s and obtain a BFS-tree T_s [4]. In order to construct BFS-tree of G , we first initialize T_s as a tree having only the root node s and then add children of s in G to T_s . That is, if $(s, x) \in G$, we add x to T_s as a child of s . Then iteratively, for each leaf node x in current T_s , we search all its children y , such that $(x, y) \in E$ and y has not been added to T_s yet and add y to T_s as a child of x .

Example 3. Also considering the graph G in Figure 1, the BFS-tree of G can be seen in Figure 2. If we choose v_0 as the source, the distances between v_0 and other nodes are also $\{[v_1, 1], [v_2, 1], [v_3, 2], [v_4, 2], [v_6, 2], \text{ and } [v_5, 3]\}$, but the BFS can finish in only 3 expanding processes.

3. SSMTkSP

3.1. Problem Statement

Definition 4. Given a (weighted) directed graph $G = (V, E)$ and a candidate node set C which is chosen from V , the SSMTkSP problem is that, for a source node $u \in V$, find k

FIGURE 2: BFS-tree of Graph G.

nearest neighbors with shortest paths, such that those nearest neighbors are in C .

In our work, we do not care about the k exact paths, but rather the k target nodes, or called nearest neighbors, so the SSMTkSP problem can also be considered as single-source many-target k nearest neighbors problem. The candidate set C could be chosen from V according to nodes' importance on the graph, such as PageRank scores [5].

3.2. DijkstraKNN Algorithm on MapReduce. On MapReduce, we keep a root set R for source s . At the beginning, R contains only $(s, 0)$, and at each iteration, we add a nearest neighbor u and the shortest distance between s and u to R . As long as R contains k nodes coming from C , or there is no out-link from R , the expanding process terminates. Then we can get the k nearest neighbors through another Map phase. Details of the DijkstraKNN algorithm can be seen in Algorithms 1 and 2.

3.3. BFSKNN and PruningBFSKNN Algorithms on MapReduce. The DijkstraKNN algorithm expands one nearest neighbor in each iteration for the source s . However, we can expand all its neighbors of the same hops at the same time, that is, BFSKNN algorithm. In BFSKNN algorithm, iteratively we expand all neighbors of the same hops from s

and update the shortest distances between s and these nodes. While all distances do not change any more, we obtain the shortest distances between s and other nodes in a graph. Details of the BFSKNN algorithm can be seen in Algorithm 3.

Different from the Dijkstra algorithm, which expands the nearest neighbor in each iteration, the BFSKNN algorithm expands all nodes that have the same hops from s (see Algorithm 4 for details). The conditions that terminate them are also different. In the Dijkstra algorithm, if we have found k nearest neighbors or R has not any out-link, the algorithm terminates. However, in the BFSKNN algorithm, the termination condition is that all distances between s and other nodes in G do not change.

Theorem 5. *The BFSKNN Algorithm terminates in a limited number of iterations, and the output is a valid solution to the SSMTkSP problem.*

Proof. Given the graph G , construct a BFS-tree T rooted at source s in the following steps:

- (1) let s be the root of T ;
- (2) find all out-links for each leaf node of T in G and add them to T as children of that leaf node;
- (3) repeat Step 2 until that the path from root to leaf node forms a loop in G .

Input: A (weighted) graph $G = (V, E)$, candidate set C , k nearest neighbors, and source node s ;
Output: A list l of $(node, distance)$ from s , where $|l| \leq k$;

```

(1) let  $R = \{(s, 0)\}$ ;
(2) let  $R.NODE = \{r.node \mid r \in R\}$ ;
(3) /*****
(4) Here we redefine “ $\cap$ ” as follows:
(5)  $R \cap C = \{(v_i, d_i) \mid (v_i, d_i) \in R \text{ and } v_i \in C\}$ 
(6) *****/
(7) while  $|R \cap C| < k \wedge \exists(u, v)(u \in R.NODE \wedge v \in (V(G) - R.NODE) \wedge (u, v) \in E(G))$  do
(8)   DijkstraMapper( $R, G$ );
(9)   //find a nearest neighbour
(10)   $R = R \cup \text{DijkstraReducer}()$ ;
(11) end while
(12)  $l = C \cap R$ ;
(13) return  $l$ 
```

ALGORITHM 1: DijkstraKNN(G, C, k, s).

DijkstraMapper (R, G):

```

(1) for all  $(from, to)$  in  $G$  do
(2)   if  $from \in R.NODE \wedge to \notin R.NODE$  then
(3)     emit( $from, to$ );
(4)   end if
(5) end for;
DijkstraReducer (output of DijkstraMapper):
(6) let  $result(key, value) = (0, +\infty)$ ;
(7) for all  $(key, value)$  do
(8)   //find a nearest neighbour not in  $R$ ;
(9)   if  $key \notin R.NODE \wedge value < result.value$  then
(10)     $result = (key, value)$ ;
(11)  end if
(12) end for;
(13) return  $result$ 
```

ALGORITHM 2: Expanding a nearest neighbor.

So, we have that there is a simple path from s to v in G if and only if there is a path from root to leaf node v in T . The above steps finish in a limited number i ($i \leq |V|$) of iterations, so the BFSKNN algorithm terminates in a limited number of iterations. After i iterations, we can find all the shortest paths between s and other nodes in G , so since the $(i + 1)$ -th iteration, all the shortest paths between s and other nodes will not change any more, and the k nearest neighbors from C construct a valid solution to the SSMT k SP problem. $\square$

However, in each expanding process of BFSKNN, all nodes expand along with their out-links, and this causes the problem of path expansion because all edges have to be accessed. In order to prune edges that are not necessarily accessed, we keep a list $cList$ for each node, which records nodes from the candidate set for the current path. In $cList$, if a path contains k nodes from C , then we prune paths derived from that path, because they do not contain any

useful information about k nearest neighbors. Details can be seen in Algorithms 5 and 6.

Example 6. Given the graph G in Figure 1, candidate set $C = \{v_0, v_2, v_5, v_6\}$ and $k = 2$, we can prune the BFS tree of G in Figure 2, and the results are in Figure 3.

Theorem 7. *The PruningBFSKNN algorithm terminates in a limited number of iterations, and the output is a valid solution to the SSMT k SP problem.*

Proof. Constructing a BFS-tree T the same as Proof of Theorem 5, we have that there is a simple path from source s to target v in G if and only if there is a path from root s to leaf node v in T . Prune paths have k candidate nodes from C after the k th candidate node. There are two kinds of pruned nodes, normal nodes (not belonging to C) and candidate nodes. If a node is a candidate node, it cannot be one of the k nearest neighbors in the pruned paths, because we have found k nearest neighbors in those paths. Otherwise, if a node is a normal node, then whether it appears in pruned paths or not does not affect the termination of the algorithm and does not affect the correctness of the algorithm either. So we have that the PruningBFSKNN algorithm terminates in a limited number of time, and the output is a valid solution to the SSMT k SP problem. $\square$

4. Performance Analysis

In this section, we analyze and compare the number of iterations and I/O cost of the DijkstraKNN, BFSKNN, and PruningBFSKNN algorithms. Realistic analysis of the efficiency of MapReduce algorithms is not straightforward, because the algorithms' efficiency in practice depends on many other factors, such as distribution of data, scheduling of jobs, and proximity of the communicating machines. However, these factors are all controlled by the system, and

Input: A (weighted) graph $G = (V, E)$, candidate set C , k nearest neighbors, and source node s ;
Output: A list l of $(node, distance)$ from s , where $|l| \leq k$;

- (1) let $R = null$;
- (2) let $R' = \{(v_i, d_i) \mid v_i == s?0 : +\infty, 0 \leq i \leq |G|\}$;
- (3) **while** $R' \neq R$ **do**
- (4) $R = R'$;
- (5) BFSMapper(R, G);
- (6) $R' = \text{BFSReducer}()$;
- (7) **end while**;
- (8) //“ $\cap$ ” means the same as Algorithm 1
- (9) $l = C \cap R$;
- (10) **return** l

ALGORITHM 3: BFSKNN(G, C, k, s).

BFSMapper (R, G):

- (1) **for all** $(node, distance)$ in R **do**
- (2) $\text{emit}(node, distance)$;
- (3) **end for**;
- (4) **for all** $(from, to)$ in G **do**
- (5) $\text{emit}(to, from.distance + w_{from,to})$;
- (6) **end for**;

BFSReducer (output of BFSMapper):

- (7) **for all** key **do**
- (8) let $result(key, value) = (0, +\infty)$;
- (9) **for all** value **do**
- (10) **if** $value < result.value$ **then**
- (11) $result = (key, value)$;
- (12) **end if**
- (13) **end for**;
- (14) $\text{emit}(result)$;
- (15) **end for**

ALGORITHM 4: Expanding neighbors with the same hops in BFSKNN.

only the number of iterations and I/O cost can be controlled by users' MapReduce programs.

4.1. The Number of Iterations

Theorem 8. The Dijkstra algorithm with parameters $G(V, E)$, C, k and s finishes in $O(k \times (|V|/|C|))$ MapReduce iterations.

Proof. Since C is chosen from V , so when we expand a node, the probability that it belongs to C is $(|C|/|V|)$. In order to obtain k nearest neighbors, we need $O(k \times (|V|/|C|))$ expanding processes, so the Dijkstra algorithm finished in $O(k \times (|V|/|C|))$ MapReduce iterations. $\square$

Theorem 9. The BFSKNN and PruningBFSKNN algorithms with parameters $G(V, E), C, k$ and s finish in k ($k \leq D$) MapReduce iterations.

Proof. Construct a BFS-tree T the same as Proof of Theorem 5, so the depth of T equals the diameter D of G , and the BFSKNN and PruningBFSKNN algorithms terminate in at most D expanding processes, so the BFSKNN and PruningBFSKNN algorithms finish in at most D MapReduce iterations. $\square$

4.2. I/O Cost Analysis. We now start by analyzing the I/O cost of the DijkstraKNN algorithm. The DijkstraKNN algorithm terminates in $O(k \times (|V|/|C|))$ expanding processes. In each expanding process, the input is R and G , so the total I/O cost is

$$O\left(\frac{k \cdot |V|}{|C|} (|V| + |E|)\right). \quad (1)$$

Next, we analyze the I/O cost of the BFSKNN algorithm. The BFSKNN algorithm terminates in at most D expanding processes. In each expanding process, the input is R and G , so the total I/O cost is

$$O(D (|V| + |E|)). \quad (2)$$

At last, we analyze the I/O cost of the PruningBFSKNN algorithm. The PruningBFSKNN algorithm also terminates in at most D expanding processes. In the i th ($1 \leq i < k$) expanding process, the input is the same as BFSKNN, but in the i th ($k \leq i \leq D$) expanding process, we prune $2^i(1 - \prod_{j=0}^{i-k}(1 - C_{k-1+j}^{k-1}(|C|/|V|)^k(1 - (|C|/|V|))^j))$ unnecessary paths, so the total I/O cost that we save is

$$O\left(\sum_{i=k}^D 2^i \left(1 - \prod_{j=0}^{i-k} \left(1 - C_{k-1+j}^{k-1} \left(\frac{|C|}{|V|}\right)^k \left(1 - \frac{|C|}{|V|}\right)^j\right)\right)\right). \quad (3)$$

5. Experiments

In this section, we present the results of the experiments that we did to test the performance of our algorithms.

Input: A (weighted) graph $G = (V, E)$, candidate set C , k nearest neighbors, and source node s ;
Output: A list l of $(node, distance)$ from s , where $|l| \leq k$;

```

(1) let  $R = null$ ;
(2) let  $R' = \{(v_i, cList_i, d_i) \mid v_i == s \wedge 0 : +\infty, cList_i = 0, 0 \leq i \leq |G|\}$ ;
(3) while  $R' \neq R$  do
(4)    $R = R'$ ;
(5)   PruningBFSMapper( $R, G$ );
(6)    $R' = \text{PruningBFSReducer}()$ ;
(7) end while;
(8) /* * * * * *
(9) Here we redefine " $\cap$ " as follows:
(10)  $R \cap C = \{(v_i, d_i) \mid (v_i, cList_i, d_i) \in R \text{ and } v_i \in C\}$ 
(11) /* * * * * */
(12)  $l = C \cap R$ ;
(13) return  $l$ 

```

ALGORITHM 5: PruningBFSKNN(G, C, k, s).

PruningBFSKNNMapper (R, G):

```

(1) for all  $(node, cList, distance)$  in  $R$  do
(2)   emit( $node, cList, distance$ );
(3) end for;
(4) for all  $(from, to)$  in  $G$  do
(5)   if  $from.cList \cap C \geq k$  then
(6)     return;
(7)   else if  $from \in C$  then
(8)      $cList = cList \cup \{from\}$ 
(9)   end if;
(10)  emit( $to, cList, from.distance + w_{from,to}$ );
(11) end for;
PruningBFSKNNReducer (output of PruningBFSKNNMapper):
(12) for all  $key$  do
(13)  Let  $result(key, cList, value) = (0, 0, +\infty)$ ;
(14)  for all  $value$  do
(15)    if  $value < result.value$  then
(16)       $result = (key, cList, value)$ ;
(17)    else if  $value == result.value \wedge |key.cList| > |result.cList|$  then
(18)       $result = (key, cList, value)$ ;
(19)    end if;
(20)  end for;
(21)  emit( $result$ );
(22) end for

```

ALGORITHM 6: Expanding neighbors with the same hops in PruningBFSKNN.

TABLE 1: Summary characteristics of datasets.

Dataset	$ V $	$ E $	D
Epinions	196,591	950,327	13
LiveJournal	4,847,571	68,993,773	18

Here $|V|$, $|E|$, and D represent the number of nodes, the number of edges and the diameter of a graph respectively.

5.1. Experimental Setup

Dataset. In order to demonstrate the robustness of our methods and to show their performance on realistic data, we

present experiments with two real-world datasets, Epinions social network [6] and LiveJournal social network [7, 8]. Summary statistics about these datasets are presented in Table 1.

Experimental Platform. We implement the Dijkstra, BFSKNN, and PruningBFSKNN algorithms in Java on top of Hadoop platform. Our experiments are executed on a cluster of 20 nodes, where each node is a commodity machine with a 2.16 GHz Intel Core 2 Duo CPU and 1 GB of RAM, running CentOS v6.0.

FIGURE 3: Pruned BFS-tree of graph G , candidate set $C = \{v_0, v_2, v_5, v_6\}$ and $k = 2$.

FIGURE 4: PageRank for graphs.

5.2. Experimental Results. For choosing the candidate sets, we compute the PageRank scores [5] of each graph using Pegasus [9] and select 5,000 top ones as the candidate set. As can be seen from Figure 4 that PageRank scores of the two graphs all conform to the power law, that means only a small number of nodes are important. In the ads, the candidate

sets usually contain the entities in which we gain profit from business.

We choose top 50, 500, and 5000 PageRank score nodes from Epinions social network and top 5000 PageRank score nodes from LiveJournal social network and compare the performance of the above algorithms. We select randomly

FIGURE 5: Execution time of Epinions.

20 source nodes from each graph and compute the average execution time.

As you can see from Figure 5, the execution time of the DijkstraKNN algorithm changes greatly as we change the size of the candidate set, and bigger candidate set has smaller execution time. Moreover, the BFSKNN and PruningBFSKNN algorithms have more stable execution time. We also print all results for candidate sets size of 500 and 5000 in Figures 6(a) and 6(b), which show that the execution time grows linearly as the growth of iterations. This is because the inputs are R and G in each iteration, and the processing time is about 37 seconds. However, the execution time also grows nearly linearly as the growth of k , which is against our intuition that it should grow exponentially as the number of nodes grows exponentially. The reason is that when we search in a graph, we usually find high PageRank score nodes instead of those fringe nodes.

We also compare the stability of the above algorithms; details are in Figure 7. The execution time of the DijkstraKNN algorithm changes greatly as we choose different sources because some nodes having more influential neighbors need less iterations. The BFSKNN and PruningBFSKNN algorithms have more stable execution time because their execution time depends on the radius of a graph for the source, and the radius of a graph is more stable in practice.

For LiveJournal social network, we choose the top 5000 PageRank score nodes as the candidate set and compare the performance of the above algorithms. The result is in Figure 8. The DijkstraKNN algorithm is efficient for small k , but if the candidate set is smaller and k is bigger, the PruningBFS algorithm is a better choice.

6. Related Work

In this section, we review some related work about MapReduce, the shortest paths and entity recommendation for graphs.

MapReduce. MapReduce algorithms have been designed or proposed in the literature for a wide range of applications, such as machine learning [10], text processing [11], and bioinformatics [12, 13]. MapReduce provides an excellent tool for large graph processing as well; a number of graph processing systems have been designed on it in the literature, such as Pegasus [9] and Giraph (<http://incubator.apache.org/giraph/>). In this paper, we study one of the most well-known graph computation problems, that is, computing the shortest paths or the nearest neighbors.

The Shortest Paths. Computing the shortest paths is a basic and important primitive that lies at the core of graph related problems. The shortest path problem can be classified as SSSTSP, SSMTSP, APSP, and k SP and classical algorithms such as Dijkstra algorithm [3] and Floyd-Warshall algorithm [14] can only handle small graphs in serial computation model. As the graphs become larger and larger, in order to answer questions timely, researchers deal with this problem by approximation [15–17], preprocessing [18, 19], or parallelization. Current parallel algorithms for the shortest path problem are mainly based on searching the BFS-tree of the graph [18, 20, 21], but they either study the problem of k SP or focus on APSP. In this paper, we study the problem of SSMTSP, which is a composition of the above two, design BFS-based parallel algorithm on MapReduce, and propose corresponding pruning strategy.

Entity Recommendation. As complex networks, such as social networks and web-page graphs, become more and more popular, entity recommendation on graphs becomes a hot-spot among researchers. Current entity recommendation methods are mainly content-based or link-based. Content-based entity recommendation algorithms consider only the content of entities and recommend the most similar ones for some entity (refer to [22, 23] for details). Rank-based entity recommendation algorithms consider the structure of the graphs and recommend the most influential ones for some entity. The influence of entities can be defined as PageRank [5], HITS [24], SimRank [25], or some methods that derive from them. In this paper, from the perspective of influence, we consider the nearest neighbors as the most influential entities and recommend the nearest neighbors from the selected candidate set for some entity.

7. Conclusion and Future Work

In this paper, we study the problem of single-source many-target k shortest paths for graphs, which is finding k nearest neighbors from a candidate set for one source. To the best of our knowledge, this is the first study of recommending k nearest neighbors from selected candidate set. As MapReduce programming model becomes more and more popular in

FIGURE 6: Dijkstra algorithm.

FIGURE 7: Stability comparison.

FIGURE 8: Execution time of LiveJournal.

data intensive computing, we evaluate algorithms on its open-source implementation-Hadoop. We design algorithms based on the Dijkstra and breadth-first search algorithms, propose an efficient pruning algorithm for breadth-first search, and evaluate their performance.

As graphs become bigger and bigger, precise algorithms need much time to compute several nearest neighbors and some fringe nodes may even waste more computing sources. In the future, we plan to seek approximation algorithms that can handle this problem.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

- [1] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-based internet of things: discovery, query, selection, and on-demand provisioning of web services,"

- IEEE Transactions on Services Computing*, vol. 3, no. 3, pp. 223–235, 2010.
- [2] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
 - [3] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
 - [4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, Cambridge, Mass, USA, 2001.
 - [5] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: bringing order to the web,” Tech. Rep., 1999.
 - [6] M. Richardson, R. Agrawal, and P. Domingos, “Trust management for the semantic web,” in *The Semantic Web—ISWC 2003*, vol. 2870 of *Lecture Notes in Computer Science*, pp. 351–368, 2003.
 - [7] 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 (KDD ’06)*, pp. 44–54, ACM, 2006.
 - [8] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, “Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters,” *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.
 - [9] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “PEGASUS: a peta-scale graph mining system—implementation and observations,” in *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM ’09)*, pp. 229–238, IEEE, Miami, Fla, USA, December 2009.
 - [10] D. Gillick, A. Faria, and J. DeNero, “Mapreduce: distributed computing for machine learning,” Tech. Rep., 2006.
 - [11] J. Lin and C. Dyer, “Data-intensive text processing with mapreduce,” *Synthesis Lectures on Human Language Technologies*, vol. 3, no. 1, pp. 1–177, 2010.
 - [12] S. Leo, F. Santoni, and G. Zanetti, “Biodoop: bioinformatics on hadoop,” in *Proceedings of the 38th International Conference Parallel Processing Workshops (ICPPW ’09)*, pp. 415–422, IEEE, September 2009.
 - [13] R. Taylor, “An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics,” *BMC Bioinformatics*, vol. 11, supplement 12, article S1, 2010.
 - [14] R. Floyd, “Algorithm 97: shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.
 - [15] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec, *HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop*, Citeseer, 2008.
 - [16] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, “Fast shortest path distance estimation in large networks,” in *Proceedings of the ACM 18th International Conference on Information and Knowledge Management (CIKM ’09)*, pp. 867–876, ACM, November 2009.
 - [17] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Zhao, “Orion: shortest path estimation for large social graphs,” in *Proceedings of the 3rd conference on Online social networks*, p. 9, USENIX Association, 2010.
 - [18] Y. Xiac, W. Wu, J. Pei, W. Wang, and Z. He, “Efficiently indexing shortest paths by exploiting symmetry in graphs,” in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT ’09)*, pp. 493–504, ACM, March 2009.
 - [19] F. Wei, “TEDI: efficient shortest path query answering on graphs,” in *Proceedings of the International Conference on Management of Data (SIGMOD ’10)*, pp. 99–110, ACM, June 2010.
 - [20] D. A. Bader and K. Madduri, “Designing multithreaded algorithms for breadth-first search and si-connectivity on the Cray MTA-2,” in *Proceedings of the International Conference on Parallel Processing (ICPP ’06)*, pp. 523–530, IEEE, August 2006.
 - [21] J.-H. Wen, H.-L. Jiang, Z. Mei, and J.-L. Song, “Scene shortest path solutions based on the breadth first search,” *Advanced Materials Research*, vol. 267, pp. 25–29, 2011.
 - [22] S. Debnath, N. Ganguly, and P. Mitra, “Feature weighting in content based recommendation system using social network analysis,” in *Proceedings of the 17th International Conference on World Wide Web (WWW ’08)*, pp. 1041–1042, ACM, New York, NY, USA, 2008.
 - [23] O. Phelan, K. McCarthy, M. Bennett, and B. Smyth, “Terms of a feather: content-based news recommendation and discovery using twitter,” *Advances in Information Retrieval*, vol. 6611, pp. 448–459, 2011.
 - [24] J. M. Kleinberg, “Authoritative sources in a hyperlinked environment,” *Journal of the ACM*, vol. 46, no. 5, pp. 604–632, 1999.
 - [25] G. Jeh and J. Widom, “SimRank: a measure of structural-context similarity,” in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’02)*, pp. 538–543, July 2002.