Random Walk Based Key Nodes Discovery in Opportunistic Networks

http://dx.doi.org/10.3991/ijoe.v12i03.5412

QIN Qin, HE Yong-qiang
Henan Institute of Engineering, Zhengzhou, China

***Abstract*—In opportunistic networks, temporary nodes choose neighbor nodes to forward messages while communicating. However, traditional forward mechanisms don't take the importance of nodes into consideration while forwarding. In this paper, we assume that each node has a status indicating its importance, and temporary nodes choose the most important neighbors to forward messages. While discovering important neighbors, we propose a binary tree random walk based algorithm. We analyze the iteration number and communication cost of the proposed algorithm, and they are much less than related works. The simulation experiments validate the efficiency and effectiveness of the proposed algorithm.**

***Index Terms*—Random Walk, Key Nodes Discovery, Opportunistic Networks, forward mechanisms**

I. INTRODUCTION

While communicating in opportunistic networks, it doesn't require a connected path between the source node and the target node [1]. Opportunistic networks are self-organized, the nodes in this kind of networks are mobile devices, and communication between two nodes is implemented by the opportunity that nodes come across [2]. Assuming that node *a* wants to send a message to node *b*, because there is no connected path between them, *a* has to save the message until it comes across and forwards it to other node *c*; *c* repeats the above 'save and forward' process; and finally, when the message is transmitted to the target node *b*, the communication is finished. This kind of opportunistic networks has been applied in many fields, such as tracking wild animals, organizing handheld and vehicular ad hoc networks, etc.

In opportunistic networks, while a node forwards a message to a temporary node, it usually randomly chooses available neighbor nodes according to some designated mechanism [3]. Classical forwarding mechanisms include direct transmission, location based and estimation based forwarding mechanism. However, the common deficit of these forwarding mechanisms is that they don't take the importance of nodes into consideration. Once a node chooses some inactive or unimportant nodes to forward message, the whole forward process has to be blocked until these nodes become active, and this decrease the efficiency of the whole message transmission process.

In this paper, we take the importance of nodes into consideration while forwarding a message in opportunistic networks, i.e., discovering key nodes for forwarding. We assume that each node has a status indicating its importance. In this paper, we apply PageRank, borrowed from social networks, to present the importance of a node. When a node chooses some neighbors to forward message, it chooses some more important nodes or called key nodes, and at the same time, it divides its status and sends them to all its neighbors. Each neighbor node updates its own status upon receiving an incoming status. We propose an optimized binary tree random walk based algorithm for key nodes discovery. The proposed algorithm can get the statuses of nodes with less forward iterations and communication cost than related works.

This paper is organized as follows. In section 2, we introduce some preliminaries used in this paper. Section 3 introduces related works about random walk based node sampling in networks. Section 4 presents the proposed binary tree random walk based algorithm. In section 5, we give a theoretical analysis about the proposed algorithm. Simulation experiments are given in section 6, and conclusion is given in section 7.

II. PRELIMINARIES

A. Undirected Graph

Assuming an undirected graph $G=(V,E)$, where V is the set of nodes, and E is the set of edges, we denote the number of nodes in G by n, and the number of edges in G by m. The adjacent matrix M of G is

$$m_{i,j}=\begin{cases}1 & (i,j)\in E,\\ 0 & otherwise.\end{cases} \tag{1}$$

As G is undirected, the edge $(u,v)\in E$ **if and only if** $(v,u)\in E$, and thus the adjacent matrix M is symmetric.

An undirected graph G can also be represented as a directed graph $G'=(V,E')$, where $(u,v)\in E$ **if and only if** $(u,v)\in E'$ and $(v,u)\in E'$. So, the adjacent matrix of the directed graph G' is also M. We divide M into two matrixes, upper triangular matrix M^u and lower triangular matrix M^l, where,

$$M^l=\begin{bmatrix}0 & \cdots & 0 & 0\\ m_{2,1} & 0 & (0) & 0\\ \vdots & \ddots & 0 & \vdots\\ m_{n,1} & \cdots & m_{n,n-1} & 0\end{bmatrix}, \text{ and } M^u=(M^l)^T.$$

Given an adjacent matrix M, the transition possibility matrix P of M is

$$p_{i,j}=\frac{m_{i,j}}{\sum_{j=1}^{n}m_{i,j}} \tag{2}$$

where the weights on the outgoing edges of each node sum up to 1.

B. Binary Tree

A **Binary Tree** is a tree data structure, in which each node has at most two child nodes, usually distinguished as 'left' and 'right'. Nodes with children are parent nodes, and child nodes may contain references to their parents. Outside the tree, there is often a reference to the 'root' node (the ancestor of all nodes). A tree, which does not have any node other than the root node, is called a null tree. A tree with n nodes has exactly $n-1$ edges. The **Length** of a tree is its number of nodes.

A **Full Binary Tree** is a tree in which every node other than the leaves has two children. Or, perhaps more clearly, every node in a binary tree has exactly 0 or 2 children.

In a binary tree, there is only one path from each node to the root node, and we define the **Node Depth** is the number of edges in that path. The depth of the root node is 0. We define the **Binary Tree Depth** is the biggest depth for all nodes in the binary tree, that is$\max depth(u), \forall u, u \in V$.

C. Personalized PageRank

The PageRank value of a node in a graph is proportional to its parents' PageRank values, but inversely proportional to its parents' out-degrees [4]. In a random walk or a random surfer model, the PageRank vector π is the stationary distribution, and it can be computed by the following formula:

$$\pi = P^T \cdot \pi \tag{3}$$

Personalized PageRank [5, 6] is that: In a random walk, at each step, with a probability ϵ, usually called the teleport probability, jumps to the source node, and with probability $1-\epsilon$ follows a random outgoing edge from the current node. Personalized PageRank is the same as PageRank, except that with a probability ϵ jumps to the source node, for which we are personalizing the PageRank. The Personalized PageRank vector of graph G, with respect to the source node u, denoted by π_u, satisfies:

$$\pi_u = \epsilon\delta_u + (1-\epsilon)P^T \cdot \pi_u \tag{4}$$

where $\delta_u(v) = 1$ **if and only if** $u = v$, and 0 otherwise.

In some applications, such as friend recommendation or link prediction, it needs to compute all the vectors π_u for all $u \in V$, and this is the fully Personalized PageRank computation problem. However, it only requires the top$k$ values (and corresponding nodes) in each Personalized PageRank vector for some suitable value of k.

D. Monte Carlo Approach

There are two main approaches to compute Personalized PageRank. One approach is to use linear algebraic techniques, such as Power Iteration [4], and the other approach is the Monte Carlo method. The Monte Carlo method, also known as statistical simulation method, solves problems with (pseudo) random number. The Monte Carlo method of Personalized PageRank is to approximate Personalized PageRank scores by directly simulating the corresponding random walks and then estimating the stationary distributions with the empirical distributions of the performed walks.

Fogaras et al. [7] and Avrachenkov et al. [8] proposed the following Monte Carlo method for Personalized PageRank approximation: Starting at each node $u \in V$, do a number R of random walks (called 'fingerprints') starting at u, each having a length geometrically distributed as $Geom(\epsilon)$. Then, the frequencies of visits to different nodes in these fingerprints will approximate the Personalized PageRank values.

III. RELATED WORKS

In opportunistic networks, forwarding mechanisms are classified into direct transmission [9, 10], location based [11, 12] and estimation based [13, 14] forwarding mechanism, and readers can refer to reference [15] for details. In this paper, we proposed a random walk based key node sampling approach for message forwarding, and we mainly focus on the sampling efficiency and message cost of the algorithm, so we review related works about random walk based node sampling.

A. Monte Carlo Baseline

The Monte Carlo approach simulates R random walks from each node $u \in V$. Starting from u, and at each step, appends a node to the end of the fingerprint. As the probability that the random walk stops with is ϵ, the length of the fingerprint is geometrically distributed as $Geom(\epsilon)$, and the expectation of length is $\frac{1}{\epsilon}$.

The above method needs to do the random step many times for long walks. In order to reduce the uncertainty of the geometrical distribution, Bahmani et al. [16] propose to use the constant length $\frac{1}{\epsilon}$ for each fingerprint. In the rest of the paper, we call this method Monte Carlo Baseline, or MCBL for short.

Note: Given a graph G, the MCBL algorithm with parameters λ finishes in λ iterations.

B. The SQRT Algorithm

Das Sarma et al. [17] present an algorithm to efficiently do a single random walk from a source node on the graph in the streaming computation model. In order to construct a random walk of length λ, it first does a short random walk segment of length θ for each node in the graph, and then tries to merge these short segments to form a longer fingerprint. However, to keep the random walk truly random, it cannot use the segment at each node more than once. Hence, if a segment appears twice in a random walk, it gets stuck, and then it needs to bring a number of edges out of a properly defined set of nodes to the main memory to continue the random walk.

Bahmani et al. [18] modify the above algorithm by the following method: To make sure that the algorithm doesn't get stuck when constructing a longer random walk, they do more short random walks at each node. More precisely, if one wants to construct a random walk of length λ, firstly, it needs to do $\lfloor \frac{\lambda}{\theta} \rfloor$ ($\lfloor \cdot \rfloor$ is the floor of $\cdot$) short random walks of length θ, and a short random walk of length $\lambda \% \theta$ ($\lambda \bmod \theta$) if $\lambda \% \theta \neq 0$. While merging, each short segment cannot be used more than once, and this would make sure that the algorithm doesn't get stuck. In the rest of the paper, we call this algorithm the SQRT algorithm. The SQRT algorithm is a simple modification of the algorithm proposed by Das Sarma et al. [17], and can be easily implemented on distributed systems.

The SQRT algorithm consists of two parts, segment generation and segment merging. In the segment generation part, it needs θ iterations to generate the initial seg-

ments, and in the segment merging part, it needs $\lceil \frac{\lambda}{\theta} \rceil - 1$ ($\lceil \cdot \rceil$ is the roof of $\cdot$) iterations to merge these segments.

Note: Given a graph G, the SQRT algorithm with parameters λ and θ finishes in $\theta + \lceil \frac{\lambda}{\theta} \rceil - 1$ iterations, which is optimal when $\theta = \sqrt{\lambda}$, resulting in $2\sqrt{\lambda}$ iterations.

C. The Doubling Algorithm

In order to reduce the number of iterations, Bahmani et al. [18] and Csáji et al. [19] propose the Doubling algorithm. Similar to the SQRT algorithm, the Doubling algorithm also consists of two parts, segment generation and segment merging, where the segment generation part is the same as the SQRT algorithm. In the merging part, the SQRT algorithm performs just one merge per node (per iteration) and grows a single walk per node. In contrast to the SQRT algorithm, the Doubling algorithm merges the short fingerprints with dichotomy, and performs multiple merges and grows multiple walks per node.

After the segment generation part, there are $\frac{\lambda}{2\theta}$ pairs of segments of length θ. In the first iteration of the merging part, the Doubling algorithm merges the $\frac{\lambda}{2\theta}$ pairs of segments to construct $\frac{\lambda}{2\theta}$ fingerprints of length 2θ; in the second iteration, it merges the $\frac{\lambda}{2^2\theta}$ pairs of segments to construct $\frac{\lambda}{2^2\theta}$ fingerprints of length $2^2\theta$; in the k-th iteration, it merges the $\frac{\lambda}{2^k\theta}$ pairs of segments to construct $\frac{\lambda}{2^k\theta}$ fingerprints of length $2^k\theta$, and so on.

Note: Given a graph G, the Doubling algorithm with parameters λ and θ finishes in $\theta + \lceil \log_2 \lceil \frac{\lambda}{\theta} \rceil \rceil$ iterations, which is optimal when $\theta = 1$, resulting in $1 + \lceil \log_2 \lambda \rceil$ iterations.

IV. BINARY TREE RANDOM WALK

The Doubling algorithm is proved to be optimal in a directed graph for constructing a single random walk [18], and the number of iterations is $1 + \lceil \log_2 \lambda \rceil$. However, for undirected graphs, we can reduce the iterations a lot by our defined binary tree random walk.

A. Binary Tree Random Walk

A **Binary Tree Random Walk** is the construction of a binary tree: starting from the source node (called root node in the binary tree), in the first iteration, chooses two nodes from root's neighbors, one randomly chosen from its out-links as its right child node, and another node randomly chosen from its in-links as its left child node; in each of the following iterations, for each leaf node of the tree, chooses two nodes from its neighbors, one randomly chosen from its out-links as its right child node, and another randomly chosen from its in-links as its left child node. Figure 1 is simple binary tree random walk started with node 0.

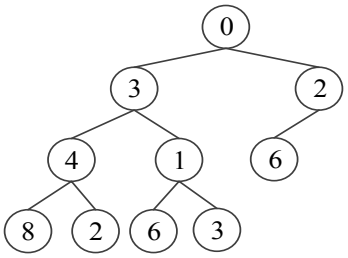


Figure 1. A simple binary tree random walk

B. The Binary Tree Algorithm

Similar to the SQRT and Doubling algorithms, the Binary Tree algorithm has also two parts, segment generation and segment merging.

The segment generation part SegGen(G, λ) is in Algorithm 1. Given a graph G, to construct a binary tree of length λ, we need $\lceil \frac{\lambda-1}{2} \rceil$ independent binary tree segments of depth 2, where $\lfloor \frac{\lambda-1}{2} \rfloor$ of them are independent full binary tree segments, and another binary tree of two nodes if $(\lambda - 1)\%2 = 1$. To keep the binary tree really random, we need to make sure that the binary tree random walk doesn't stuck, so we construct $\lceil \frac{\lambda-1}{2} \rceil$ segments for each node in the graph. Among these segments, the $\lfloor \frac{\lambda-1}{2} \rfloor$ full binary tree segments have both left and right children chosen randomly from the root's neighbors, and the other one (if it exists) has only one child as root's left child node.

Algorithm 1 SegGen(G, λ)

Input: An undirected graph $G = (V, E)$ and the desired length of binary tree random walk λ;

Output: $\lfloor \frac{\lambda-1}{2} \rfloor$ independent full binary tree segments of depth 2 and another binary tree with only two nodes if $(\lambda - 1)\%2 = 1$, rooted at each node;

```
1.  for all v ∈ V do
2.    for i = 1 to ⌊(λ−1)/2⌋ do
3.      Let S[u, i] = [u, i];
4.      //Append two children;
5.      S[u, i]=[u, i;RandomNeighbor(u),
                    RandomNeighbor(u)];
6.    end for;
7.  end for;
8.  //Another binary tree if it exists
9.  If (λ − 1)%2 = 1 then
10.   S[u, i]=[u, i;RandomNeighbor(u),-1];
11. end if;
12. return S;
```

The segment merging part Binarytree(G, λ) is in Algorithm 2. After the segment generation part, there are $\lceil \frac{\lambda-1}{2} \rceil$ independent binary tree segments of depth 2. In the first iteration of the merging part, it merges every $2^{2^0} + 1$ binary trees into a bigger one, and then we have $\lceil \frac{\lceil \frac{\lambda-1}{2} \rceil}{2^1+1} \rceil$ binary trees each with 2^2 leaves nodes; in the second iteration, it merges every $2^{2^1} + 1$ binary trees, and then we have $\lceil \frac{\lceil \frac{\lceil \frac{\lambda-1}{2} \rceil}{2^1+1} \rceil}{2^2+1} \rceil$ binary trees each with 2^{2^2} leaves nodes, and so on, until we have only one binary tree of length λ.

Example 1: Given a graph G and $\lambda = 14$, the algorithm first generates $\lfloor \frac{14-1}{2} \rfloor = 6$ binary tree of length 3,

and a binary tree of length 2 for each $u \in V$, and saves them as files. The merging process is in figure 2. After the generation of segments, each node has 7 segments, and they are mutually independent. In the first iteration of the merging part, constructs a tree of length 7 for each node. For example, for the tree rooted at node 0, whose leaves are v_i and v_j, we append two trees (rooted at v_i and v_j) to it, and thus have 2 trees of length 7 and 1 tree of length 2 for each node. In the second iteration, we append the trees rooted at v_p and v_q to the tree rooted at node 0, and then we get a binary tree random walk of length 14 for node 0.

Algorithm 2 Binarytree(G, λ) //segments merging

Input: An undirected graph $G = (V, E)$ and the desired length of binary tree random walk λ;

Output: One binary tree random walk of length λ starting at each node in G;

1. Let S=SegGen(G, λ);
2. Define $\eta = \lceil \frac{\lambda-1}{2} \rceil$, and $\eta = \lceil \frac{\lambda-1}{2} \rceil$, and $\forall u \in V$, $1 \le i \le \eta$: $W[u, i, \eta]$=$S[u, i]$, $Leaf[u, i, \eta]$=$W[u, i, \eta].LastLine$;
3. for $j = 1$ to $\lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil$ do
4. $\eta' = \lceil \frac{\eta}{2^{2^j-1}+1} \rceil$;
5. for $i = 1$ to η' do
6. for all $u \in V$ do
7. if $i = \frac{\eta}{2^j+1}$ then
8. $W[u, i, \eta'] = W[u, i, \eta]$;
9. $Leaf[u, i, \eta']$=$W[u, i, \eta]$;
10. continue;
11. end if;
12. for all $v \in Leaf[u, i, \eta]$ do
13. $W[u, i, \eta'] = W[u, i, \eta].Append(W[v\ \eta - i + 1, \eta])$;
14. end for;
15. $Leaf[u, i, \eta']$=$W[u, i, \eta'].LastLine$;
16. end for;
17. end for;
18. $\eta = \eta'$;
19. end for;
20. for all $u \in V$ do
21. output $W[u, 1, 1]$;
22. end for;

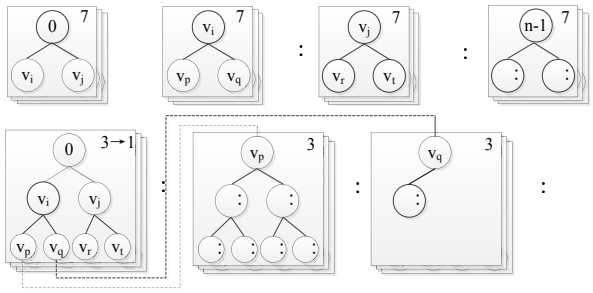


Figure 2. Construction of a binary tree random walk

V. THEORETICAL ANALYSIS

Similar to the proof of the Doubling algorithm, the binary tree random walk is also a random walk. The reason is simply that all segments are generated randomly, and each of the segments isn't used more than once in a binary tree random walk.

A. The Number of Iterations

After the generation of initial segments, the Binary Tree algorithm merges them in a few iterations, until it has one segment per node.

Lemma 1. *Given a set T of binary trees, appends a tree t' to t if the root of t' is the leaf of t, and thus appending all trees ($\forall t', t' \in T \wedge t' \neq t$) to the tree $t (t \in T)$ needs 1 iteration.*

Theorem 1. *Given a graph G, the Binary Tree algorithm with parameter λ finishes in $\lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil$ merging iterations*

Proof: Let Num_t be the number of binary trees after each iteration, Len_t be the length of a binary tree, and $Leaf_t$ be the number of leaf nodes of a binary tree.

In the segment generation part, SegGen(G, λ) finishes in one iteration, after which we have $\lceil \frac{\lambda-1}{2} \rceil$ independent binary tree segments of depth 2 rooted at each node, so$Num_t(0) = \lceil \frac{\lambda-1}{2} \rceil$, $Leaf_t(0) = 2$, and $Len_t(0) = 3$.

In the segment merging part, at the first iteration, it merges every $2^1 + 1$ binary trees into a bigger one, so

$Leaf_t(1) = 2^2 = Leaf_t(0)^2$,
$Len_t(1) = 2 * Leaf_t(1) - 1$, and
$Num_t(1) = \lceil \frac{\lceil \frac{\lambda-1}{2} \rceil}{2^1+1} \rceil = \lceil \frac{Num_t(0)}{2^1+1} \rceil$.

The second iteration merges every $2^2 + 1$ binary trees, so

$Num_t(2) = \lceil \frac{\lceil \frac{\lceil \frac{\lambda-1}{2} \rceil}{2^1+1} \rceil}{2^2+1} \rceil = \lceil \frac{Num_t(1)}{2^2+1} \rceil$,
$Leaf_t(2) = (2^2)^2 = Leaf_t(1)^2$, and
$Len_t(2) = 2 * Leaf_t(2) - 1$.

At the kth iteration, it merges every $Leaf_t(k-1) + 1$ binary trees, and then we have

$Num_t(k) = \lceil \frac{Num_t(k-1)}{Leaf_t(k-1)+1} \rceil$,
$Leaf_t(k) = Leaf_t(k-1)^2$, and
$Len_t(k) = 2 * Leaf_t(k) - 1$.

Above all, the problem can be transformed to following optimization problem:

$$\begin{cases} Leaf_t(0) = 2 \\ Leaf_t(k) = Leaf_t(k-1)^2 \\ Len_t(k) = 2 * Leaf_t(k) - 1 \\ \min k, \textbf{such that } Len_t(k) \geq \lambda \end{cases} \quad (5)$$

Given $Leaf_t(k) = Leaf_t(k-1)^2$, we can have that $Leaf_t(k) = Leaf_t(0)^{(2^k)}$. Moreover, for $Leaf_t(0) = 2$, and thus we have $Leaf_t(k) = 2^{(2^k)}$. If we want $2 * Leaf_t(k) - 1 \geq \lambda$, i.e., $2^{(2^k)} \geq \frac{\lambda+1}{2}$, then it needs that $k \geq \log_2 \log_2 \frac{\lambda+1}{2}$. So, the segment merging part finishes in $\lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil$ iterations. □

From lemma 1 and theorem 1, we can have the following corollary.

Corollary 1. Given a graph G, the Binary Tree algorithm with parameter λ finishes in $\lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil + 1$ iterations.

Table 1 lists the iterations complexity. From the table we can see that our proposed Binary Tree algorithm has the least number of iterations in a graph.

TABLE I.
ITERATION COMPLEXITY COMPARISON

Algorithm	Number of iterations
MCBL	λ
SQRT	$\theta + \lceil \frac{\lambda}{\theta} \rceil - 1$
Doubling	$\theta + \lceil \log_2 \lceil \frac{\lambda}{\theta} \rceil \rceil$
Binary Tree	$\lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil + 1$

B. Communication Cost

In a graph with *n* nodes and *m* edges, we analyze the communication cost of generating a random walk of λ nodes.

During the generation of binary tree segments, the input is the whole network, and the communication cost is $O(m)$; each node has three random walk segments with $\lceil \frac{\lambda-1}{2} \rceil$ nodes as output, and thus the overall output for *n* nodes is $O(n\lambda)$. So, the communication cost for the generation process is $O(m + n\lambda)$.

During the merging process of binary tree segments, at the k-th $(1 \leq k \leq \lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil)$ iteration, and for each node, the input contains $2^{2^{k-1}+1} - 1$ binary tree segments with $\lceil \frac{\lambda-1}{2^{2^{k-1}+1}-2} \rceil$ nodes, and the output are $2^{2^k+1} - 1$ binary tree segments with $\lceil \frac{\lambda-1}{2^{2^k+1}-2} \rceil$ nodes, so the output is $O(\lambda)$. Because the network has *n* nodes, the total output for each iteration is $O(n\lambda)$. For $\lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil$ iterations, the communication cost of the binary tree merging process is $O(n\lambda \log_2(\log_2 \lambda))$.

Above all, the total communication cost for the binary tree algorithm is $O(m + n\lambda \log_2(\log_2 \lambda))$, and table II gives the comparison of communication cost for related random walk based algorithms.

TABLE II.
COMMUNICATION COST COMPARISON

Algorithm	Communication cost
MCBL	$O(m\lambda + n\lambda^2)$
SQRT	$O((m + n\lambda)\theta + n\frac{\lambda^2}{\theta})$
Doubling	$O((m + n\lambda)\theta + n\lambda \log_2 \frac{\lambda}{\theta})$
Binary Tree	$O(m + n\lambda \log_2(\log_2 \lambda))$

C. Approximation Node Importance with Binary Tree Random Walk

In order to approximate the PageRank value for each node, we use the visited frequency of each node. Given an undirected graph $G = (V, E)$, its adjacent matrix M, and its transition possibility matrix P, we can have the corresponding directed graph $G' = (V, E')$, where $(u, v) \in E$ **if and only if** $(u, v) \in E'$ and $(v, u) \in E'$. The adjacent matrix and transition possibility matrix of the directed graph G' is also M and P, and the upper triangular matrix and lower triangular matrixes of M are M^u and M^l, respectively. The transition possibility matrixes of M^u and M^l are P^u and P^l, respectively.

The visited frequency of a leaf node can be divided into two parts by M^u and M^l. That is, we can reach the leaf node v through the last edge either in M^u or in M^l. If we reach v through M^u, then the visited frequency $X(v) = X(v) + 1$; and if we reach v through M^l, then the visited frequency $Y(v) = Y(v) + 1$. In a binary tree of length λ with $\frac{\lambda+1}{2}$ leaves, the rank value of a node $v \in V$ is divided into two parts, $\pi^u(v)$ and $\pi^l(v)$, and they are defined as follows:

$$\begin{matrix} \lim_{\lambda\to\infty} \frac{2X(v)}{\lambda+1} \longrightarrow \pi^u(v) \\ \lim_{\lambda\to\infty} \frac{2Y(v)}{\lambda+1} \longrightarrow \pi^l(v) \end{matrix} \quad (6)$$

Theorem 2. Given an undirected graph $G = (V, E)$, and a binary tree random walk w of length λ on G, the rank vector π (defined as follows) is the PageRank vector.

$$\pi(v) = \lim_{\lambda\to\infty} \frac{2(X(v) + Y(v))}{\lambda + 1} \quad (7)$$

Proof: From equation 6 and 7, we have $\pi = \pi^u + \pi^l$. In the k-th level of the binary tree w, the PageRank vector is $\pi^{(k)}$. While in the $(k+1)$-th level, $P\pi^{(k)} = (P^u + P^l)\pi^{(k)} = P^u\pi^{(k)} + P^l\pi^{(k)} = \pi^{u,(k+1)} + \pi^{l,(k+1)} = \pi^{(k+1)}$, so $\pi^{(k+1)} = P\pi^{(k)}$. That is the PageRank definition in equation 3. □

VI. SIMULATION EXPERIMENTS

In this section, we present the results of the experiments that we have done to test the performance of our method. As discussed by Bahmani et al. [18], the Doubling algorithm is the state-of-the-art approach in the literature, and thus we compare our proposed Binary Tree algorithm with the Doubling algorithm. As efficiency measures, we consider the clock time of the algorithms, and as quality measure, we consider the approximation error for the top k nodes (for suitable values of $k = 10$ or 100).

A. Experimental Setup

In this experiment, in order to validate the overall performance of the algorithms in the whole networks, we test all nodes in a graph and observe the average iteration number and message transmission cost. Because there are always thousands or more nodes in a network, we implement the algorithms in Java on top of the Hadoop platform. Our experiments are executed on a cluster of 20 nodes, where each node is a commodity machine with a 2.16GHz

Intel Core 2 Duo CPU and 1GB of RAM, running CentOS v6.0. As opportunistic network dataset are hard to acquire, we validate the algorithms on two social network dataset, which have similar attributes as opportunistic networks. The two datasets are Tencent and LiveJournal [20]. Summary statistics about these datasets are presented in Table III.

TABLE III.
SUMMARY CHARACTERISTICS OF THE DATASETS

Dataset	*n*	*m*
Tencent	1,944,589	50,655,143
LiveJournal	4,847,571	68,993,773

Here, *n* and *m* represent the numbers of nodes and edges of a graph, respectively.

B. Evaluating Measures and Choosing Parameters

Efficiency Measure: To measure the efficiency of algorithms, we consider the Clock Time.

Clock Time: How long, in terms of wall clock time, it took each job to run from the beginning to the end.

Quality Measure: In friend recommendation, only the users with highest Personalized PageRank values will be recommended. For a node u and a value k, we define Z_u^k to be the set of nodes with the k largest Personalized PageRank values for u, so only the elements in Z_u^k will be returned. Real friend recommendation applications often involve lots of factors, but personalized PageRank usually acts as one of the most important features for these tasks. In this paper, we aim to approximate personalized PageRank, so we take the Personalized RageRank (equation 4) as the standard. In order to measure how well we are approximating the results in Z_u^k and their true numeric values of personalized PageRank, we use:

$$Error(u,k) = \frac{\sum_{v \in Z_u^k} |\pi_u(v) - \hat{\pi}_u(v)|}{\sum_{v \in Z_u^k} \pi_u(v)} \quad (8)$$

where $\hat{\pi}_u(v)$ is the approximate Personalized PageRank by either the Doubling algorithm or the Binary Tree algorithm.

Choosing Parameters: In our experiments, we used the typical value $\epsilon = 0.15$ as our PageRank teleport probability (in equation 4). As stated by Bahmani et al. [18], the Doubling algorithm are not very sensitive to the choice of θ, and the efficiency of the algorithm doesn't change very much for small θ, so we choose constant $\theta = 2$ in our experiments for convenience.

The degree distributions of the two datasets in our experiments all conform to the power law distribution. The nodes in our graphs have widely different degrees. Most of the nodes in the graphs have very small degrees, and only a few nodes have very big degrees. In order to validate the quality of the algorithms for different degrees, we divide the nodes of a graph into 10 buckets, where bucket i $(1 \le i \le 9)$ contains all the nodes whose degree is in the $[2^{i-1}, 2^s)$ interval, and bucket 10 contains all the nodes whose degree is in $[2^9, +\infty)$. We sample 10 nodes uniformly at random from each bucket, use the Average Clock Time and the Average Error of the selected 100 nodes to represent the efficiency measure and the quality measure, respectively. The Average Error is defined as follows:

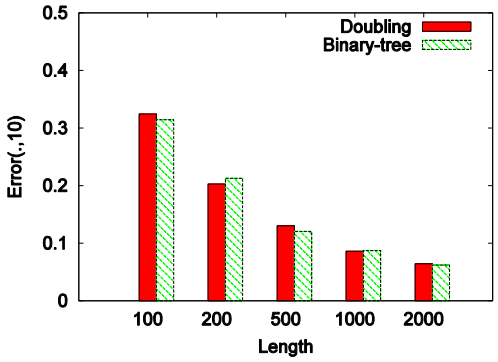
$$\overline{Error}(.,k) = \sum_{i=1}^{100} Error(i,k) \quad (9)$$

C. Experimental Results

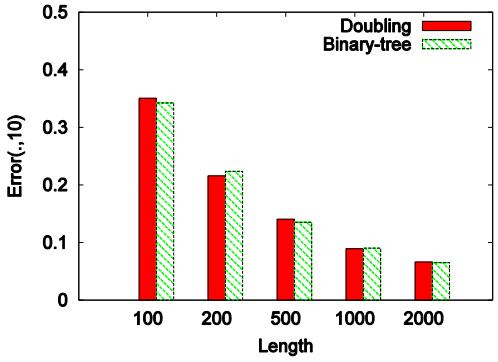
The Doubling algorithm and the Binary Tree algorithm are all approximation methods for Personalized PageRank. We first compare the estimation quality of the two algorithms, and then compare their efficiency.

Quality: We use the Personalized PageRank (equation 4), and let the typical value $\epsilon = 0.15$ as our PageRank teleport probability. As the computation of equation 4 is the process of linear iteration, we assume the results of 100 iterations as standard values. For the quality measure, we sample 10 nodes uniformly at random from each bucket, let variable k be 10 and 100 respectively, and compare the Average Error (equation 9) between the Binary Tree algorithm with the Doubling algorithm. The results, given in 3 and 4, show that the Average Errors of the two algorithms go down while we increase the length of the random walks in both datasets in both $k = 10$ and $k = 100$; and more importantly, our Binary Tree algorithm has nearly the same quality with the Doubling algorithm.

Efficiency: Firstly, we observe the change of Average Clock Time of the two algorithms along with the increase of the random walk length in both datasets, and the results are in Figure 5. From the figure we can see that, both Average Clock Times of the two algorithms increase along with the increase of the random walk length, but our Binary Tree algorithm increase more moderately. When we increase the random walk length, the number of iterations of the Doubling algorithm conforms to $O(\log_2 \lambda)$,

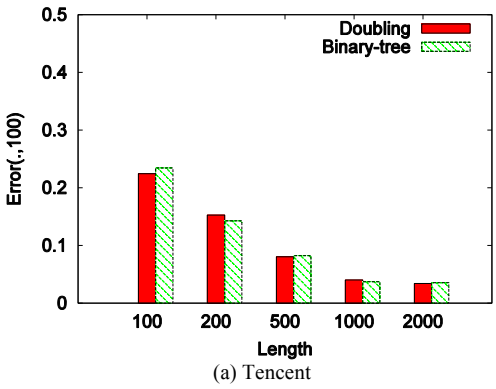


(a) Tencent

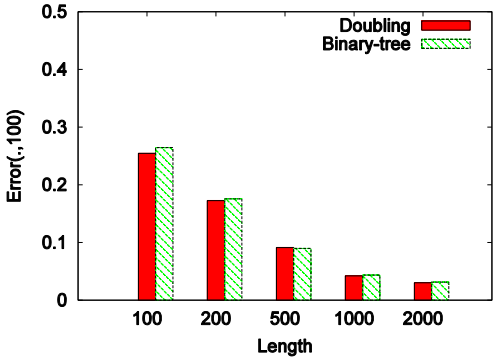


(b) LiveJournal

Figure 3. $\overline{Error}(.,10)$ VS Length



(a) Tencent



(b) LiveJournal

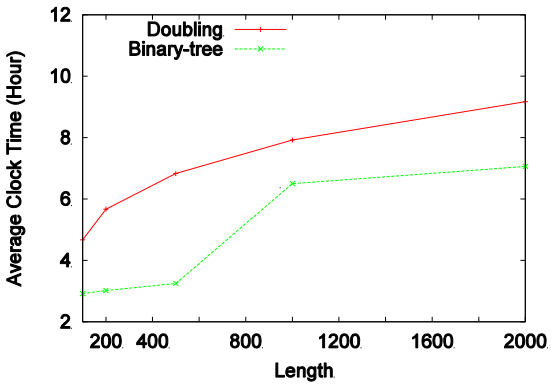
Figure 4. $\overline{Error}(.,100)$ VS Length

but the number of iterations of the Binary Tree algorithm conforms to $O(\log_2 \log_2 \lambda)$. Moreover, when we increase the random walk length from 500 to 1000, the Binary Tree algorithm needs one more iteration ($\lceil \log_2 \lceil \log_2 \lceil \frac{1000+1}{2} \rceil \rceil \rceil$ - $\lceil \log_2 \lceil \log_2 \lceil \frac{500+1}{2} \rceil \rceil \rceil = 1$), but before 500 and after 1000, the number of iterations doesn't change, and only the computation in the iterations themselves increase, that is why there is a sudden rise in the figure.

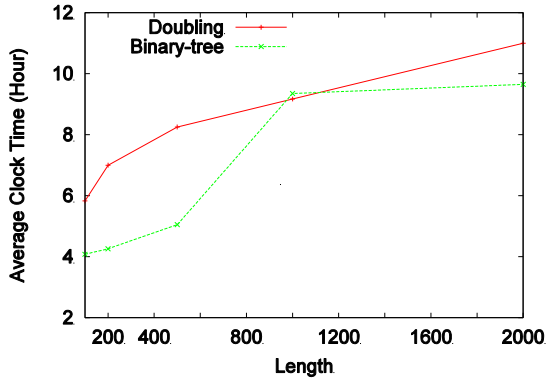
Next, we observe the Average Error VS the Average Clock Time for $k = 10$ or 100 in both of datasets, and the results are in Figure 6 and 7. As it is clear from the figures, the Binary Tree algorithm consistently performs better than the Doubling algorithm in the Tencent dataset; while in the LiveJournal dataset, the Binary Tree algorithm performs better than the Doubling algorithm in most cases except the sudden rise point.

VII. CONCLUSION

In this paper, we studied the importance of nodes while communicating in opportunistic networks. Traditional message forwarding mechanisms don't consider the importance of nodes while choosing neighbors to forward messages, and will be blocked while choosing inactive neighbors. However in this paper, we proposed a binary tree random walk based key node sampling algorithm. According to our algorithm, each node keeps a status (PageRank value), and while forwarding messages, the node can choosing active neighbors. At the same time, each node updates their status dynamically, which makes sure that the whole network is up-to-date. With our method, we can find key nodes in $\lceil \log_2 \lceil \log_2 \lceil \frac{\lambda+1}{2} \rceil \rceil \rceil + 1$ iterations, which greatly reduces the discovery time of key nodes. In addition, the communication cost of the proposed algorithm is much less than related works.

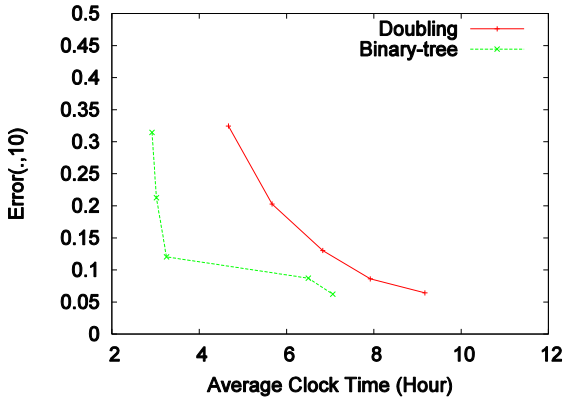


(a) Tencent

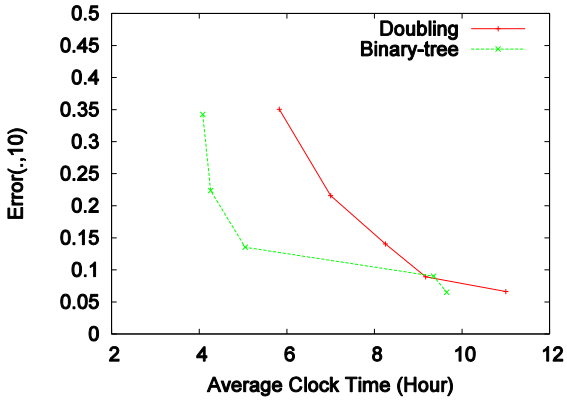


(b) LiveJournal

Figure 5. Average Clock Time VS Length

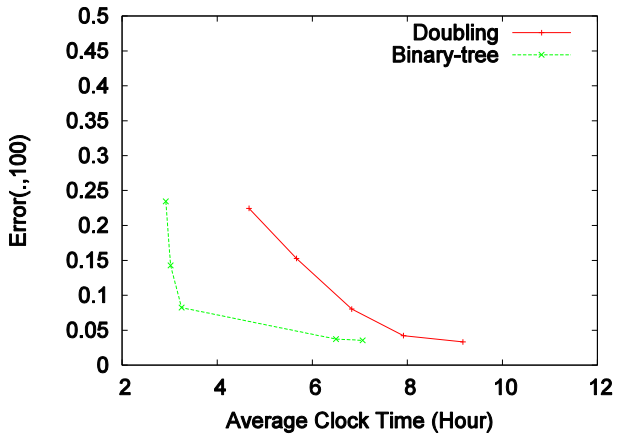


(a) Tencent

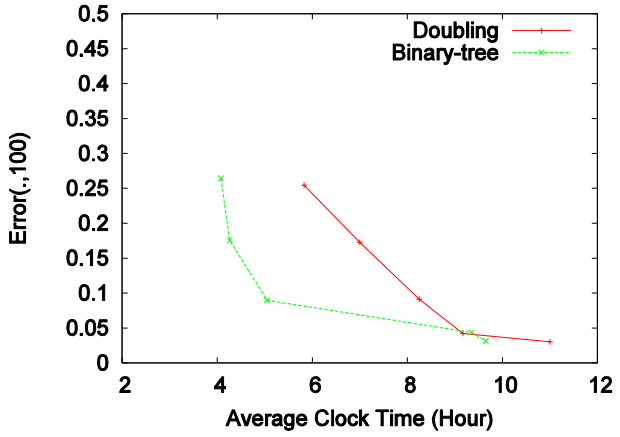


(b) LiveJournal

Figure 6. $\overline{Error}(.,10)$ VS Average Clock Time



(a) Tencent



(b) LiveJournal

Figure 7. : $\overline{Error}(.,100)$ VS Average Clock Time

REFERENCES

[1] Boldrini C, Conti M, Jacopini J, et al. Hibop: a history based routing protocol for opportunistic networks[C]//World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a. IEEE, 2007: 1-12.

[2] Pelusi L, Passarella A, Conti M. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks[J]. Communications Magazine, IEEE, 2006, 44(11): 134-141. http://dx.doi.org/10.1109/MCOM.2006.248176

[3] Lilien L, Kamal Z H, Bhuse V, et al. Opportunistic networks: the concept and research challenges in privacy and security[J]. Proc. of the WSPWN, 2006: 134-147.

[4] Brin S, Page L. The anatomy of a large-scale hypertextual Web search engine[J]. Computer networks and ISDN systems, 1998, 30(1): 107-117. http://dx.doi.org/10.1016/S0169-7552(98)00110-X

[5] Jeh G, Widom J. Scaling personalized web search[C]//Proceedings of the 12th international conference on World Wide Web. ACM, 2003: 271-279.

[6] Haveliwala T, Kamvar S, and Jeh G. An Analytical Comparison of Approaches to Personalizing PageRank. Technical Report, No. 2003-35, Stanford, 2003.

[7] Fogaras D, Rácz B, Csalogány K, et al. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments[J]. Internet Mathematics, 2005, 2(3): 333-358. http://dx.doi.org/10.1080/15427951.2005.10129104

[8] Avrachenkov K, Litvak N, Nemirovsky D, et al. Monte Carlo methods in PageRank computation: When one iteration is sufficient[J]. SIAM Journal on Numerical Analysis, 2007, 45(2): 890-904. http://dx.doi.org/10.1137/050643799

[9] Spyropoulos T, Psounis K, Raghavendra C S. Single-copy routing in intermittently connected mobile networks[C]//Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. IEEE Communications Society Conference on. IEEE, 2004: 235-244.

[10] Grossglauser M, Tse D. Mobility increases the capacity of ad-hoc wireless networks[C]//INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. IEEE, 2001, 3: 1360-1369.

[11] LeBrun J, Chuah C N, Ghosal D, et al. Knowledge-based opportunistic forwarding in vehicular wireless ad hoc networks[C]//Vehicular technology conference, 2005. VTC 2005-Spring. 2005 IEEE 61st. IEEE, 2005, 4: 2289-2293.

[12] Chiou B S, Lin Y J, Hsu Y C, et al. K-hop search based geographical opportunistic routing for query messages in vehicular networks[C]//Next-Generation Electronics (ISNE), 2013 IEEE International Symposium on. IEEE, 2013: 279-282.

[13] Amantea G, Rivano H, Goldman A. A Delay-Tolerant Network Routing Algorithm Based on Column Generation[C]//Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on. IEEE, 2013: 89-96.

[14] Zhu Y, Xu B, Shi X, et al. A survey of social-based routing in delay tolerant networks: positive and negative social effects[J]. Communications Surveys & Tutorials, IEEE, 2013, 15(1): 387-401. http://dx.doi.org/10.1109/SURV.2012.032612.00004

[15] Huang C M, Lan K, Tsai C Z. A survey of opportunistic networks[C]//Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on. IEEE, 2008: 1672-1677.

[16] Bahmani B, Chowdhury A, Goel A. Fast incremental and personalized PageRank[J]. Proceedings of the VLDB Endowment, 2010, 4(3): 173-184. http://dx.doi.org/10.14778/1929861.1929864

[17] Sarma A D, Gollapudi S, Panigrahy R. Estimating pagerank on graph streams[J]. Journal of the ACM (JACM), 2011, 58(3): 13. http://dx.doi.org/10.1145/1970392.1970397

[18] Bahmani B, Chakrabarti K, Xin D. Fast personalized pagerank on mapreduce[C]//Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011: 973-984.

[19] Csáji B C, Jungers R M, Blondel V D. PageRank optimization by edge selection[J]. Discrete Applied Mathematics, 2014, 169: 73-87. http://dx.doi.org/10.1016/j.dam.2014.01.007

[20] Leskovec J, Lang K J, Dasgupta A, et al. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters[J]. Internet Mathematics, 2009, 6(1): 29-123. http://dx.doi.org/10.1080/15427951.2009.10129177

AUTHORS

Qin Qin is with the College of Computer, Henan Institute of Engineering, Zhengzhou, China. (e-mail: qinqinxuezhe@sina.com).

He Yong-qiang is with the College of Computer, Henan Institute of Engineering, Zhengzhou, China.

This work was supported in part by Key scientific research project of Henan Province (15A520054), science and technology project of Henan province(112102310550). Submitted 28 December 2015. Published as resubmitted by the authors 20 february 2016.