PAPER *Special Section on Computational Intelligence and Big Data for Scientific and Technological Resources and Services*

A Fast Algorithm for Liquid Voting on Blockchain

Xiaopin ZHOU[†a)], **Peng LI**[†b)], **Yulong ZENG**[††c)], **Xuepeng FAN**[††d)], **Peng LIU**[†††e)], ***Nonmembers***,
and* Toshiaki MIYAZAKI**[†f)], ***Member

SUMMARY Blockchain-based voting, including liquid voting, has been extensively studied in recent years. However, it remains challenging to implement liquid voting on blockchain using Ethereum smart contract. The challenge comes from the gas limit, which is that the number of instruction for processing a ballot cannot exceed a certain amount. This restricts the application scenario with respect to algorithms whose time complexity is linear to the number of voters, i.e., $O(n)$. As the blockchain technology can well share and reuse the resources, we study a model of liquid voting on blockchain and propose a fast algorithm to eliminate the restriction. The key idea behind our algorithm is to shift some on-chain process to off-chain. In detail, we first construct a Merkle tree off-chain which contains all voters' properties. Second, we use *Merkle proof* and interval tree to process each ballot with $O(\log n)$ on-chain time complexity. Theoretically, the algorithm can support up to 2^{1000} voters with respect to the current gas limit on Ethereum. Experimentally, the result implies that the consumed gas fee remains at a very low level when the number of voters increases. This means our algorithm makes liquid voting on blockchain practical even for massive voters.

key words: *liquid voting, blockchain, smart contract, gas limit, Merkle tree*

1. Introduction

Since the inception of Bitcoin, blockchain has shown great promises in implementing decentralized cryptocurrencies that have attracted significant attentions of academia, industry and government. Essentially, a blockchain is a decentralized and immutable public ledger ensured by cryptography and peer-to-peer networking technologies. The ledger share the transactions and reuse the technological resources across many participants so that any involved record cannot be altered retroactively, without the alteration of all subsequent blocks. This allows participants to verify and audit transactions independently and relatively inexpensively [1]. The early blockchain implementation, represented by Bitcoin, can store simple data formats in the form of money-transferring transactions. Later, Ethereum and other emerging blockchain projects have enhanced the blockchain by involving smart contracts, which are programs running on blockchain nodes for storing and operating on-chain data. With smart contracts, we can implement various complex decentralized applications with strong security and trustiness guarantee, e.g., insurance, supply chain management and copyright protection.

Voting is one of the most popular blockchain applications, and plays an important role in many decision-making scenarios. Liquid voting (or liquid democracy) is a form of delegative democracy [2], which lies between direct and representative democracy [3–6]. Voters can either vote directly or delegate their vote to other participants. Voters delegate some others who may further elect other representatives, which forms a delegation graph. Since voters may change their delegation by the voting deadline, frequent graph operations, like graph traversing, will be executed.

Graph traversing operations are common and various efficient algorithms are available on our traditional computing systems. However, it becomes a big challenge for a unique limitation called *gas limit* when we conduct such operations on blockchain. We take Ethereum as an example. Executing a single instruction of smart contracts consumes a certain amount of *gas fee*, which varies from several to 10 thousand[†]. Ethereum has a parameter `block_gas_limit`, usually about 10 million, which determines the total gas that can be consumed within a block(that is *gas limit*). The total gas fee for invoking a smart contract can not exceed `block_gas_limit`, because the corresponding transaction can only be included in one block, which means that the number of instructions cannot exceed a certain amount. Otherwise, the transaction will be reverted.

Many companies/parties have started practicing applications of liquid voting, such as Google Votes [7] and Pirate Parties (software: liquid feedback) [8]. However, they are centralized solutions that suffer from black-box operations and statistical errors. Blockchain is promising in solving these issues, but an efficient blockchain-based implementation of liquid voting is still an open challenge.

The difficulty lies in the self-tally requirement. Self-tally means that after all ballots have been cast, anyone can compute the result without external help. This is a natural requirement for a distributed voting scheme [9]. Naive

[†] The author is with the School of Computer Science and Engineering, The University of Aizu.
[††] The author is with Beijing YeeZTech Co., Ltd.
[†††] The author is with the College of Computer Science and Technology, Hangzhou Dianzi University.
a) E-mail: d8211110@u-aizu.ac.jp
b) E-mail: pengli@u-aizu.ac.jp
c) E-mail: cengyulong@yeez.tech
d) E-mail: xuepeng.fan@yeez.tech
e) E-mail: perryliu@hdu.edu.cn
f) E-mail: miyazaki@u-aizu.ac.jp

DOI: 10.1587/transinf.E0.D.1

[†] According to `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md`, one `storage_modify` instruction costs 5000 gas, and one `storage_add` instruction costs 20000 gas.

Copyright © 200x The Institute of Electronics, Information and Communication Engineers

algorithms usually compute the result by traversing the delegation graph on-chain, of which the time complexity is $O(n)$, where n is the number of voters. Especially, when the delegation graph is chain-like, they are undesirable due to the gas limit, since application scenarios are limited to less than one thousand voters (usually with millions of voters required).

Some discussions try to rise to the challenge by adding restrictions to the delegation graph, i.e., set an upper limit to the depth of the graph (`max_depth`). Yet, this would open the door to potential griefing attacks. In this case, delegators can block voters from delegating by delegating a chain of depth as `max_depth-current_depth`[†]. Other solutions abound but are all unreasonable when meet with the gas limit.

In this paper we propose an algorithm that reduces the on-chain time complexity to $O(\log n)$ for processing each voting information, which essentially solves the on-chain liquid voting problem. Our algorithm does not add any restriction to voters: any voter can delegate arbitrarily. Our algorithm's off-chain time complexity is also acceptable, which is only $O(n)$.

Our algorithm solves the liquid voting problem with the following aspect:

- At the beginning of a voting, each voter obtains the delegation graph by snapshotting the current height of Ethereum, then executes an $O(n)$ off-chain initialization to get his initialization data.
- Voters directly vote to a candidate by sending a voting message, attached with his initialization data. Upon receiving a voting message, the voting contract use the Merkle proof to check the correctness of the initialization data with $O(\log n)$ time complexity.
- After that, the voting contract process the voting message to update the voting status and storage realtimely through the interval tree structure which further achieves $O(\log n)$ time complexity.

The rest of this paper is organized as follows. In Section 2, we describe the background technologies and further elicits the motivation of this paper. In Section 3 we discuss the problem with an example. In Section 4, we present our algorithm for the liquid voting on blockchain problems. We do some theoretical analysis and prove some properties of our algorithm in Section 5. In Section 6 we show our experiment results. Finally, we conclude the paper in Section 7.

2. Background and Motivation

2.1 Voting and Liquid Democracy

Theories and algorithms of liquid voting have been extensively studied in recent years. Blum et al. [10] give an overview of liquid voting, including concepts, history and issues. The latest research progress of liquid voting can be found at The *Liquid Democracy Journal* [††].

Recently, a series of literature studies the implementation of blockchain-based voting systems [11, 12], some of which also refer to liquid voting but not consider the self-tally requirement. The introduction of self-tally can be found in [13], which states that the property of self-tally and perfect ballot secrecy can not be satisfied simultaneously. Thus in this paper the privacy is compromised in favor of realtime self-tallying. Yang et al. [14] introduce a self-tallying voting system by Ethereum smart contract, but do not consider the liquid voting scenario. McCorry et al. [15] also implement a distributed and self-tally electronic voting scheme using the Ethernet blockchain, while the core is to maximize the protection of voter privacy.

There are some applications for liquid voting, as Google vote and liquid feedback. The algorithms in them work in following ways: Google vote's algorithm mainly bases on the work of Schulze's [16], which is a m^3 method for electing a winner, where m is the number of candidates. They also demonstrate that the system can implement liquid voting on a social network in a scalable manner with a gradual learning curve. The basics of Liquid Feedback's algorithm come from Harmonic Weight[†††], Proportional Runoff[††††] and Schulze method, whose proposes are to determine the weights of candidates. Though both algorithms can be applied to liquid voting, the self-tallying requirement and gas limit are not taken into consideration.

2.2 Merkle Tree

Merkle tree [17] is a commonly used data structure for storing and verifying data on blockchain. The blockchain stores only the root of the Merkle tree (called *Merkle root*). A leaf node together with its correct *Merkle proof* (also known as *Merkle path*, which is defined as a sequence of nodes in the Merkle tree that corresponds to brother of each node on path from the leaf node to the root) can recover the root of the Merkle tree. Merkle proof is used to concisely proof and ensure the validity of a certain data set being inclusive in a larger data set without revealing either the complete data set or its subset. The length of the Merkle path and the time complexity for recovering the Merkle root are all logarithmic to the number of leaf nodes of the Merkle tree.

Besides, the one-wayness of the hash function guarantees that it is hard to construct a correct Merkle proof for any data that does not belong to any leaf nodes of the Merkle tree.

2.3 Interval Tree

Interval tree is a binary tree that each node holds an interval. The interval of any node is uniformly distributed to its two child nodes, until the interval becomes a singular, to be a leaf node. If the root is indexed by 1. For any non-leaf node

[†] `https://forum.aragon.org/t/open-challenges-for-on-chain-liquid-democracy/161`

[††] `https://liquid-democracy-journal.org/`

[†††] `https://en.wikipedia.org/wiki/Harmonic_mean`

[††††] `http://www.magnetkern.de/prop-runoff/prop-runoff.html`

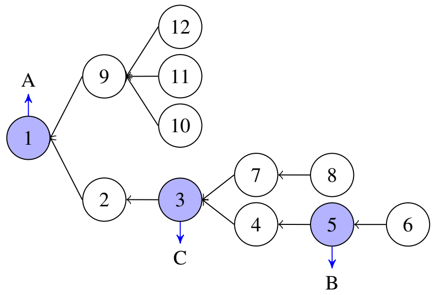

Fig. 1 Delegation tree. We ignore the virtual node with index 0 here.

k, its child nodes are indexed by $2k$ and $2k + 1$, respectively.

Interval tree supports finding and updating operation. For updating operation, usually not every leaf node is updated since the updating information may stop at an intermediate node, recorded as lazy-tag, and will be executed in subsequent operations. Finding operations need to be executed recursively starting from the root, and trigger the pass-down operation of all lazy-tags until the leaf node is reached. The complexity is $O(\log n)$ with respect to interval tree for both operations. We refer [18], Chapter 10.1 to readers for more detail.

2.4 Motivation

The security and transparency of the blockchain, combined with the freedom and flexibility of liquid democracy, make it possible to truly achieve democracy for everyone. However, due to the gas limit on Ethereum, the application scenario is restricted with respect to algorithms whose time complexity is linear to the number of voters, i.e., $O(n)$. Therefore, we hope to be able to implement an $O(\log n)$ algorithm to eliminate the restriction, as the total gas fee will nerver exceed the gas limit. There by we can make liquid voting on blockchain practical, that without any restriction to voters: no limit to the number of voters and voters can delegate arbitrarily.

3. Problem Description

In this section, we first define the process of liquid voting on blockchain. Second, we formalize the liquid voting problem. Finally, we breakdown the problem into subproblems.

In brief, liquid voting on blockchain can be separated into three periods:

(1) Spare period

The election organizer allocates voting powers to all voters, initializes voters identity if necessary, and deploys the delegate contract on blockchain. Each voter can arbitrarily delegate or undelegate by sending transactions to the delegate contract. In our model, we assume that each voter is allowed to appoint at most one delegation. During this period, no voting is casted. And a voter's delegation is not allowed to change after the spare period.

Table 1 Expamle of self-tallying the ballots for each voting

inputs	outputs		
1 A	A 78	B 0	C 0
5 B	A 67	B 11	C 0
3 C	A 45	B 11	C 22

(2) Prepare period

In this period, the election organizer needs to deploy a voting contract with the delegation graph from the spare period. Notice that the delegation graph could be too large to reconstruct on blockchain. We provide more details in Section 4.

(3) Voting period

After the voting begins, each voter can directly vote to a candidate by sending a voting message to the voting contract, with all his delegators' voting powers also casting to the same candidate. It is notable that, although our algorithm does not allow voters to change their delegations during the voting period, they can accomplish the same purpose by casting a direct voting. Changing delegation to a voter that has voted is equivalent to casting a direct voting, and changing delegation to a voter that has not voted is rare in practice.

We introduce delegation graph to describe voters' delegation behaviors. A delegation graph G is a direct graph, where each node represents a voter, and a direct edge (u, v) represents that voter v delegates his voting power to voter u. It is intuitive to assume that a delegation graph G contains no loop, thus a forest (multiple trees). Additionally, we can add a virtual node that is pointed by the root of each connected branch. So a delegation graph G can be transferred to a delegation tree T. If G contains only one connected branch, the transfermation can be ingored since it is already a delegation tree. Fig. 1 shows an example of delegation tree for 12 voters, in which voters are indexed by numbers $1, 2, \ldots, n$, while candidates are indexed by capital letters $A, B, C, \ldots$.

The liquid voting problem can be described as tallying all candidates' ballots with a subset of voters in a delegation tree voting to the candidates. Given that a delegation tree could contain thousands of nodes, it is difficult, if not impossible, to traverse it on-chain due to the gas limitation. Notice that the involved nodes of each voting is quite less than the total number of nodes, we further convert our goal to *self-tallying the ballots of all candidates for each voting message*. Instead of self-tallying the ballots after all votes are casted, our new goal is more applicable for the blockchain scenario. We again take Fig. 1 as an example, and further assume each voter's voting power equals to his index for convenience. At the beginning, nobody votes. When voter 1 votes for candidate A (as the first voter), A obtains $1 + 2 + \cdots + 12 = 78$ ballots. After voter 1 votes, suppose voter 5 (as the second voter) votes for candidate B. Then B obtains $6 + 5 = 11$ ballots. A's ballots decrease by 11, turning into 67. Further, voter 3 (as the third voter) votes for candidate C, then C obtains $3 + 4 + 7 + 8 = 22$ ballots, A's ballots become 45, and B's ballots don't change. Table 1 shows the process.

It's obvious that in order to tally the ballots after each voting, the key point is to maintain each node's *"lost voting power"* (the total ballots of his delegators who vote directly, initially valued zero). Since a certain delegation tree is established, the *total voting power* (the total ballots the voter and all his delegator hold) of each voter is determined. When a voter votes, his actual voting power is his total voting power minus his lost voting power. Meanwhile, some other voters' lost voting power should be updated after he votes. As long as each voter's lost voting power can be updated within $O(\log n)$, the liquid voting problem can be solved.

Take Fig. 1 for example, when voter 5 votes, his nearest voted parent is voter 1, so all nodes on the path $(5) \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow (1)$ are to be updated. When voter 3 votes, nodes on the path $(3) \rightarrow 2 \rightarrow (1)$ are to be updated.

So the main two goals of our algorithm are: first, find the voter's nearest voted parent; second, update lost voting powers of nodes on the path from the voter to the voter's nearest voted parent. However, to meet both these two goals we need to traverse the graph in traditional method, whose time complexity is $O(n)$.

4. Algorithm

The basic idea of our algorithm is to trade off-chain processing time for on-chain processing time. Specifically, we 1) snapshot the delegation graph and construct a Merkle tree in which a leaf represents each voter's information, like voting power and delegation; and 2) process the voting message on-chain by leveraging both Merkel proof and interval tree, which further achieves $O(\log n)$ time complexity.

4.1 Overview

Table 2 Variables used in the algorithm

Notation	Description
`T`	The delegation tree, which is generated and stored off-chain.
n	Number of nodes, as well as the length of the preorder sequence (called preorder index for short).
n_0	Length of the bracket sequence.
`node`	A type, representing the voters.
`node.weight`	Node's voting power, which is given from the snapshot.
`node.index`	Index of the node in the pre-order sequence.
`node.address`	The Ethereum address of the node, which is an inherent parameter.
`b[]`	Mapping from a node's preorder index to the node.
`nearestparent[]`	Mapping from a node's preorder index to its nearest voted parent's preorder index.
`s[]`	The score of the bracket sequence (shown in the following).
`node.endpoint`	The maximum preorder index among the node's children (include multi-level).
`node.left`	The first index where the node appears in the bracket sequence.
`node.right`	The second index where the node appears in the bracket sequence.
`node.power`	Node's total voting power (including its children's).
`node.candidate`	The candidate that the voter votes.
`v[]`	Recording the ballots of candidates.
`lazy_1[]`	Lazy-tag of the interval tree with respect to the preorder sequence, which also reflects the index of the nearest voted parent.
`lazy_2[]`	Lazy-tag of the interval tree with respect to the bracket sequence, which also reflects the "score" of the sequence.

We introduce the overall procedure of liquid voting on blockchain through three periods. As mentioned in Section 3, the delegations are constructed in the spare period, we describe the main algorithm in the steps from (A) to (F). Specifically, steps (A) to (C) describe the prepare period, while steps (D) to (F) describe the voting period.

Step A) At the beginning of the prepare period, all on-chain information are snapshotted by the current height of the blockchain, mainly, each voter's delegation and weight. Then, each voter involved in the voting locally constructs the delegation tree `T` and gets all voters' voting powers according to the following two rules:

First, for each voter, get his last delegating operation from the snapshot and add the corresponding direct edge to the delegation graph. For all edges that are on a cycle, delete the latest edge. Then repeat the deletion until the delegation graph has no cycle[†]. After that, add an edge from each zero-outdegree node to the virtual node, resulting in `T`. Since the rule is deterministic, all voters obtain a same delegation tree.

Second, all voters' voting powers can be obtained from the snapshot (`node.weight` in Table 2). There are many off-chain methods in practice to distribute voting powers, which is not critical of our paper as long as all voters can reach an agreement.

Step B) After the construction of `T`, we use `T.root` to denote the root of `T` (the virtual node). Then, each voter locally call `Preorder(T.root)` (the procedure is shown in Algorithm 1, n and n_0 are initialized with 0 to exclude the virtual node) to obtain initialization data.

Step C) Each voter constructs a Merkle tree according to the initialization data. The information of each leaf node is the hash of the node's initialization data (`data=(node.address,node.power,node.index,node.endpoint,node.left,node.right)`). The election organizer deploys a new smart contract, called the *voting contract*, which will be introduced later. The Merkle root is hard-coded in the voting contract by the election organizer.

After step C, the prepare period is over and the voting period begins. In the following steps, we describe the voting contract which processes each direct voting message.

[†] All transactions in Ethereum are attached with a timestamp. The time order on the blockchain is defined as that if a transaction's block height is larger than another trasaction, the former is later than the latter. If two transactions have the same block height, the transaction with the larger timestamp is later. The rule of Etherum guarantees that the timestamp can not be forged too far from the actual time otherwise the transaction is infeasible.

Algorithm 1: Procedure of `Preorder(root)`

```
$n \leftarrow n + 1$
$n_0 \leftarrow n_0 + 1$
root.left $\leftarrow n_0$
root.index $\leftarrow n$
root.power ← root.weight
for all node in root's direct child do
    Preorder(node)
    root.power ← root.power+node.power
end for
root.endpoint $\leftarrow n$
$n_0 \leftarrow n_0 + 1$
root.right $\leftarrow n_0$
```

Algorithm 2: Procedure of `Vote`, upon receiving a voting message

```
Input: node,voter, data,proof,node.candidate
Output: C[]
 1: if not check(RootHash,proof,data) then
 2:     return
 3: end if
 4: b[node.index] ← node
 5: FAULVP(node.left,node.left,1,$2n$,1,0)
 6: FAULVP(node.right,node.right,1,$2n$,1,0)
 7: $t$ ← node.power − s[node.left] + s[node.right]
 8: C[node.candidate] ← C[node.candidate] $+t$
 9: FAUNVP(node.index,node.index,1,$n$,1,0)
10: parent ← b[nearestparent[node.index]]
11: C[parent.candndate] ← C[parent.candndate] $-t$
12: FAUNVP(node.index +1,node.endpoint,1,$n$,1,node.index)
13: FAULVP(parent.left,node.left,1,$2n$,1,$t$)
```

Step D) When a voter casts a direct voting, he sends a voting message which contains (`data`, `proof`, `node.candidate`), where `data` is the initialization data of himself and `proof` is the Merkle path from the leaf node (`hash(data)`) to the Merkle root in the Merkle tree (constructed in step c).

Step E) Upon receiving a voting message, the voting contract first obtains the sender's Ethereum address to check if it matches with `node.address` in `data`. If it matches, the contract recovers a root according to `data` and `proof`, and checks if the result matches to the Merkle root stored in the contract. If matches, the contract begins to process the voting message, otherwise returns an "error" response.

Step F) Algorithm 2 shows the main procedure for processing a voting message, which consists of the following instructions:

- Compute the voter's lost voting power (line 5 and line 6).
- Define t to be the voter's total voting power minus lost voting power, which represents his actual votes. The ballots of the candidate he votes increase by t.
- Find the voter's nearest voted parent (line 9), whose candidate's ballots decrease by t.
- Update other voters' nearest voted parent (only the voter's children are affected) (line 12).
- Update the lost voting powers of the nodes on the path from the voter to his nearest voted parent (line 13).

Algorithm 3: Procedure of `FAUNVP`(L, R, l, r, k, v), which is to find and update the node's nearest voted parent

```
Input: $[L, R]$, which is the interval to be updated
Input: $[l, r]$, which is the current interval of the interval tree node
Input: $k$, which is the index of interval tree node
Input: $v$, which is the value for updating.
if $L = l$ and $R = r$ then
    if $v$ >lazy_1[$k$] then
        lazy_1[$k$] ← $v$
    end if
    if $L = R$ then
        nearestparent[$L$] ← lazy_1[$k$] // Recursion ends
    end if
else
    $m \leftarrow \lfloor (l + r)/2 \rfloor$
    if lazy_1[$2k$] <lazy_1[$k$] then
        lazy_1[$2k$] ← lazy_1[$k$]
    end if
    if lazy_1[$2k + 1$] <lazy_1[$k$] then
        lazy_1[$2k + 1$] ← lazy_1[$k$] // pass down the lazy-tag
    end if
    if $L \leq m$ then
        FAUNVP($L$, min{$m$, $R$}, $l$, $m$, $2k$, $v$)
    end if
    if $R > m$ then
        FAUNVP(max{$m + 1$, $L$}, $R$, $m + 1$, $r$, $2k + 1$, $v$)
    end if
end if
```

So far, the overall procedure of algorithm is produced through these three periods. The variables used in the algorithm are described in Table 2. All variables are global and initially valued 0 unless otherwise stated. In the next two subsections we will introduce the two functions `FAUNVP()`, `FAULVP()`.

4.2 Find the Nearest Voted Parent

The function `FAUNVP()`, of which the procedure is shown in Algorithm 3, achieves the goal of finding the voter's nearest voted parent, by implementing the updating operation of the interval tree with respect to the preorder sequence. The lazy-tag (`lazy_1[]`) of the interval tree's leaf node records the preorder index of corresponding voter's nearest voted parent. The following observation is sufficient for the correctness:

Observation 1.

- A node's preorder index is always smaller than its children's.
- Preoroder indexes of a node's children are successive to the node's preorder index.
- When a voter votes, only his children's nearest voted parents need to be updated, which should be at least the voter's preorder index.

For a node (voter), indexes from `node.index`+1 to `node.endpoint` represents the preorder indexes of its children, whose nearest voted parent need to be updated. Since they form a successive interval, the interval tree is applicable.

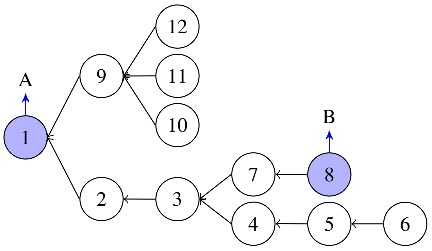

Fig. 2 Example of updating lost voting powers in a path.

4.3 Update the Lost Voting Power

When a voter votes, the lost voting powers of all the nodes on the path from the voter to its nearest voted parented should be updated. Take Fig. 2 for example, if voter 8 votes after voter 1 votes, the lost voting powers of the nodes on the path $7 \rightarrow 3 \rightarrow 2 \rightarrow 1$ should be updated. However it is not a successive interval in the preorder sequence.

We use the bracket sequence to handle this problem. A bracket sequence is to record each node twice in the pre-order traversal, one as entering and the other as exiting, called left bracket and right bracket respectively. For the tree in Fig. 2, the bracket sequence is

$$1, 2, 3, 4, 5, 6, 6, 5, 4, 7, 8, 8, 7, 3, 2, 9, 10, 10, 11, 11, 12, 12, 9, 1.$$

For a direct path from u to v in a tree, define the path's *bracket interval* to be indexed from `v.leftbracket` to `u.leftbreacket`. The following observation shows the property of the bracket interval:

Observation 2. Given a direct path, for any node not lying on the path, it either occurs twice or does not occur in the path's bracket interval. For any node lying on the path, it occurs exactly once in the path's bracket interval. Moreover, only the node's first appearance lies in the interval.

Take Fig. 2 as an example, suppose the path from 8 to 1 is to be updated, its bracket interval is $1, 2, 3, 4, 5, 6, 6, 5, 4, 7, 8$ (index 1-11). Nodes $4, 5, 6, 11, 12$ do not lie on the path, so they occur twice or do not occur in the interval. Nodes $1, 2, 3, 7, 8$ lie in the path, so they occur once in the interval.

We define an array `s[]` for recording the so-called "score" of the bracket sequence. Given a path where the nodes' lost voting powers need to add some value, we add that value to the scores of the path's bracket interval. Then for a node's lost voting power, we can compute it by

$$\texttt{s[node.left]} - \texttt{s[node.right]}$$

The reason is that, when we increase the score, only the nodes on the path increase their lost voting powers (only `s[node.left]` increases). For a node outside the path, the values of `s[node.left]` and `s[node.right]` either do not change or both increase by the same value, thus the lost voting power does not change.

We construct another interval tree with respect to the bracket sequence and maintain the score, recorded in the variable `lazy_2[]` of leaf nodes. The function `FAULVP()` gives the implementation, of which the procedure is shown in Algorithm 4.

Algorithm 4: Procedure of `FAULVP(`L, R, l, r, k, v`)`, which is to find and update the note's lost voting power

```
Input: [L, R], which is the interval to be updated
Input: [l, r], which is the current interval of the interval tree node
Input: k, which is the index of interval tree node
Input: v, which is the value for updating.
if $L = l$ and $R = r$ then
    lazy_2[$k$] ← lazy_2[$k$] +$v$
    if $L = R$ then
        $s[L]$ ←lazy_2[$k$]
    end if
else
    $m \leftarrow (l + r)/2$
    lazy_2[$2k$] ← lazy_2[$2k$] + lazy_2[$k$]
    lazy_2[$2k + 1$] ← lazy_2[$2k + 1$] + lazy_2[$k$]
    lazy_2[$k$] ← 0 // pass down the lazy-tag
    if $L \le m$ then
        FAULVP($L, \min\{m, R\}, l, m, 2k, v$)
    end if
    if $R > m$ then
        FAULVP($\max\{m + 1, L\}, R, m + 1, r, 2k + 1, v$)
    end if
end if
```

5. Theoretical Analysis

In this section we prove some properties of our algorithm. We first analyze our protocol for constructing the delegation graph.

Lemma 1. If a voter's delegating operation does not generate a cycle in the delegation graph (locally checked), the corresponding edge will never be deleted.

Proof. Assume by contradiction that the edge is deleted. By definition, there must be a cycle such that the edge is the latest, which means that the cycle is generated by the appearance of the edge, contradiction. □

Lemma 1 means that, if the voter follows the protocol, his delegation is garanteed to be retained, which is benefit for him. Otherwise if his delegation generates a cycle, it may be deleted. (It's also possible to be retained, if other voters further change their delegation so as to remove the cycle.)

Lemma 2. If a voter deviates from our mechanism, by building a delegating edge that generates a cycle, this edge will not cause other voter's delegating edge to be refused if they follows the protocol.

Proof. We call the edge built by the dishonest voter A. We prove that, if an edge B is refused with the existence of A, it will also be refused without the existence of A.

Since B is refused, it lies on a cycle which contains A. Since A also lies on another cycle, if A is deleted, B still lies

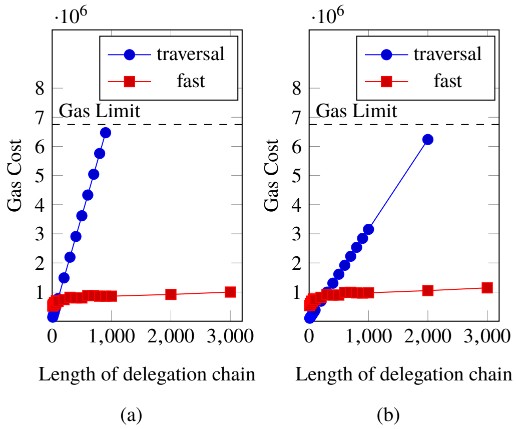

Fig. 3 Voting by delegation chain (a)*root*; (b)*leaf*.

on a cycle, and should be refused by the protocol. □

Lemma 2 shows that, even if a voter deviates from the protocol, other voters are not influenced. There are also sublinear-time algorithms that can judge whether a cycle is generated for a incoming delegating edge, which can be used in smart contract. However it is more complex and requires more gas fee for each delegating message. So still our protocol is recommended in practice.

Theorem 1. For each voting message in liquid voting problem, the voting status can be updated and displayed within $O(\log n)$ time complexity. Moreover, our algorithm can be deployed on the Ethereum mainnet and overcome the gas limitation, for the number of voters more than one million.

Theorem 1 is our main theorem, which is obvious according to the properties of the tools we used. Here we illustrate some issues. 1) Processing a voter's voting message does not rely on the initialization data of other voters that has not voted, since our algorithm only requires the data from the nearest voted parent. 2) The "mapping" structure in Solidity (the coding language of Ethereum smart contract) satisfies that, the storages are allocated only if they are assigned values. For example, the storage $lazy[3]$ can be allocated without the allocation of $lazy[1]$ and $lazy[2]$. While $lazy[1]$ and $lazy[2]$ still can be visit with a default value 0, which is just the requirement of our algorithm. 3) The time complexity of updating operation in interval tree is $O(\log n)$, because there are $O(\log n)$ levels in a interval tree and at max we will have to process 4 nodes in a level. For the part of Ethereum, we leave the proof in the experiment section.

6. Experiment

We compare our algorithm with traversal algorithm by recording the maximum consumed gas fee.

We conduct the evaluation on Ganache, which is a personal blockchain for Ethereum development that can be used to deploy contracts, and run tests. Our implementation can be found here †. As there is no standard implementation for the traversal algorithm, we implement it by ourselves. Specifically, the traversal only happens when casting a vote, instead of delegating.

Our comparison is from two aspects, 1) voting by delegation chain *root*, as illustrated in Fig. 3(a), and 2) voting by delegation chain *leaf*, as shown in Fig. 3(b). Notice that the gas limit is about 6,700,000 according to Ganache. Our evaluation shows that:

- the traversal algorithm performs better when the delegation chain is short, like smaller than 100;
- our algorithm significantly outperforms the traversal algorithm when the delegation chain is long enough;
- our algorithm can scale up with very limited gas increasing, while the traversal algorithm reaches the gas limit when the delegation chain grows up to 1,000.

When voting by a chain leaf by the traversal method, the leaf node can be found at the beginning. On the other hand, the root node will be found by traversing through all the nodes while voting by the chain root. Consequently, the gas cost of voting by delegation chain root (Fig. 3(a)) is much higher than by delegation chain leaf (Fig. 3(b)). Even so, the gas fees of both cases grow fast and reach the gas limit soon. The height of the chain is the highest among that of the normal graphs with the same number of nodes, so the cost will be highest when the delegation graph is chain-like. The result implies that the consumed gas fee remains at a very low level when the number of voters increases.

7. Conclusion

In this paper we study the liquid voting problem, where each voter can either directly vote to a candidate or delegate his voting power to a proxy. In order to make the liquid voting on blockchain practical without any restriction to voters, we propose a fast algorithm to guarantee that the gas fee of the voting transaction on blockchain through Ethereum smart contract does not exceed the gas limit. We outline the basic idea and overview of our algorithm and then describe the two key procedures. Finally, we evaluate our algorithm by comparing it with traversal algorithm. Experimental results show that our algorithm can be deployed on Ethereum and eliminate the restriction, which means it makes liquid voting on blockchain practical even for massive voters.

References

[1] C. Catalini and J.S. Gans, "Some simple economics of the blockchain," Report NBER Working Paper No.22952, NATIONAL BUREAU OF ECONOMIC RESEARCH, Cambridge, 2016.
[2] B. Cammaerts, "Revalidating participation: power and pre-figurative politics within contemporary leftwing movements," 2019.
[3] L. Carroll, The principles of parliamentary representation, Harrison and Sons, 1884.
[4] W.S. U'Ren, "Government by proxy now," New York Times, 1912.
[5] J.C. Miller, "A program for direct and proxy voting in the legislative

† `https://github.com/freeof123/liquid-voting/tree/master/ether-eval/contracts`

process," Public choice, vol.7, no.1, pp.107–113, 1969.

[6] A. Paulin, "Through liquid democracy to sustainable non-bureaucratic government," Proc. Int. Conf. for E-Democracy and Open Government, pp.205–217, 2014.

[7] S. Hardt and L.C. Lopes, "Google votes: a liquid democracy experiment on a corporate social network," 2015.

[8] J. Behrens, A. Kistner, A. Nitsche, and B. Swierczek, The principles of LiquidFeedback, Interaktive Demokratie e. V. Berlin, 2014.

[9] F. Hao, P.Y. Ryan, and P. Zieliński, "Anonymous voting by two-round public discussion," IET Information Security, vol.4, no.2, pp.62–67, 2010.

[10] C. Blum and C.I. Zuber, "Liquid democracy: Potentials, problems, and perspectives," Journal of Political Philosophy, vol.24, no.2, pp.162–182, 2016.

[11] R. Hanifatunnisa and B. Rahardjo, "Blockchain based e-voting recording system design," 2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA), pp.1–6, IEEE, 2017.

[12] F.P. Hjalmarsson, G.K. Hreiarsson, M. Hamdaqa, and G. Hjalmtysson, "Blockchain-based e-voting system," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp.983–986, IEEE, 2018.

[13] A. Kiayias and M. Yung, "Self-tallying elections and perfect ballot secrecy," International Workshop on Public Key Cryptography, pp.141–158, Springer, 2002.

[14] X. Yang, X. Yi, S. Nepal, and F. Han, "Decentralized voting: a self-tallying voting system using a smart contract on the ethereum blockchain," International Conference on Web Information Systems Engineering, pp.18–35, Springer, 2018.

[15] P. McCorry, S.F. Shahandashti, and F. Hao, "A smart contract for boardroom voting with maximum voter privacy," International Conference on Financial Cryptography and Data Security, pp.357–375, Springer, 2017.

[16] M. Schulze, "A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method," Social Choice and Welfare, vol.36, no.2, pp.267–303, 2011.

[17] G. Becker, "Merkle signature schemes, merkle trees and their cryptanalysis," Ruhr-University Bochum, Tech. Rep, 2008.

[18] K. Mulmuley, "Computational geometry," An Introduction Through Randomized Algorithms. Prentice-Hall, 1994.

Xiaoping Zhou received the B.S. degree in Computer Science and Technology and M.S. degrees in Software Engineering from the University of Huazhong Science and Technology (HUST), China, in 2009 and 2014, respectively. During 2014–2019, she worked in China as a backend development engineer. She is currently working toward the PhD degree in the University of Aizu, Japan. Her research interests focus on blockchain, security and privacy.

Peng Li received the BS degree from the Huazhong University of Science and Technology, China, in 2007, the MS and PhD degrees from the University of Aizu, Japan, in 2009 and 2012, respectively. He is currently an associate professor with the University of Aizu, Japan. His research interests include distributed systems, cloud/edge computing, machine learning systems, blockchain, as well as related security and privacy problems. He has published more than 100 technical papers on prestigious journals and conferences. He won the Young Author Award of IEEE Computer Society Japan Chapter in 2014. He won the Best Paper Award of IEEE TrustCom 2016. He supervised students to win the First Prize of IEEE ComSoc Student Competition in 2016. His serves as the guest editor of several international journal special issues and he is the editor of the IEICE Transactions on Communications and the IEEE Open Journal of the Computer Society.

Yulong Zeng obtained his PhD and Bachelor Degree at IIIS, Tsinghua University. His is a cofounder of YeeZ Technology. His research interests are algorithmic game theory, Economics and Computation and blockchain. He is a member of ASResearch.

Xuepeng Fan received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST)), in China. After working on a public blockchain project for two years, he founded YeeZ Technology at 2019. As the CEO of YeeZ Technology, his current interests include distributed systems, blockchain systems, and data privacy protection. He is a member of ASResearch.

Peng Liu received his B.S. and M.S. in Computer Science and Technology from Hangzhou Dianzi University in 2001 and 2004, and the Ph.D. degree in Computer Science and Technology from Zhejiang University in 2007, China. Currently he is an associate Professor at Hangzhou Dianzi University. His research interests include embedded system, wireless sensor networks and distributed computing, etc.

Toshiaki Miyazaki is Chairperson of the Board of Executives and President of the University of Aizu, Fukushima, Japan. His research interests are in reconfigurable hardware systems, adaptive networking technologies, and autonomous systems. He received the B.E. and M.E. degrees in applied electronic engineering from the University of Electro-Communications, Tokyo, Japan in 1981 and 1983, and Ph.D. degree in electronic engineering from Tokyo Institute of Technology in 1994. Before joining the University of Aizu, he has been worked for NTT for 22 years, and engaged in research on VLSI CAD systems, telecommunications-oriented FPGAs (Field Programmable Gate Arrays) and their applications, active networks, peer-to-peer communications, and ubiquitous network environments. He is a senior member of IEEE, IPSJ, and IEICE. He is a member of the Engineering Academy of Japan.