Implement Liquid Democracy on Ethereum: A Fast Algorithm for Realtime Self-tally Voting System

Xuepeng Fan [*1], Peng Li [†2], Yulong Zeng [‡1], and Xiaoping Zhou [§2]

[1]Autonomous System Research
[2]The University of Aizu

November 20, 2019

Abstract

We study the liquid democracy problem, where each voter can either directly vote to a candidate or delegate his voting power to a proxy. We consider the implementation of liquid democracy on the blockchain through Ethereum smart contract and to be compatible with the realtime self-tallying property, where the contract itself can record ballots and update voting status upon receiving each voting massage. A challenge comes due to the gas fee limitation of Ethereum mainnet, that the number of instruction for processing a voting massage can not exceed a certain amount, which restrict the application scenario with respect to algorithms whose time complexity is linear to the number of voters. We propose a fast algorithm to overcome the challenge, such that i) shifts the on-chain initialization to off-chain and ii) the on-chain complexity for processing each voting massage is $O(\log n)$, where n is the number of voters.

[*]xuepeng.fan@asresearch.io
[†]pengli@u-aizu.ac.jp
[‡]yulong.zeng@asresearch.io
[§]d8211110@u-aizu.ac.jp

1 Introduction

Democracy has always been a widely concerned topic. Voting, as a primary method for realizing democracy in modern society, is more and more common in practice, with applications ranging from the presidential election to community governance [9, 16]. Meanwhile, various issues emerge due to the current voting system, with low participation [3, 17] and black-box operation[1, 12][1] to be two of the most significant. Now, a group of technologist are looking for a new approach to reform the voting system, bring all people with voting rights closer to their representatives and holding elections in a public, verifiable way. In other words, voters should exercise their rights in every related voting, e.g. a policy issue or a piece of new legislation, in which the voting status should be publicly displayed and traceable. However, usually people do not have time for full participation, or they are not expert with respect to the area involved by the voting proposal, which resulted in a large number of voting powers not actually being exercised.

The concept liquid democracy (also known as proxy voting, delegate voting) is proposed to handle the participation issue, in which the core idea is that, each voter (also called delegator) can select a personal representative who has the authority to be a proxy (also called delegate) for his vote. Those delegates can further proxy their votes to other people, creating a directed network graph (called delegate graph). Whenever a delegate votes to a candidate (or a proposal, a policy), by default all his delegators' (including multi-level) voting powers are also cast to that candidate. If the delegator dislikes the way in which the proxy voted, they can either vote themselves, which dilutes the proxy's power, or pick another delegate for the next vote. Applying liquid democracy system significantly reduces time costs of voters and increases voting participation, which has been studied over a long period of time.

The early proposal about liquid democracy can be traced back to 1884 by Lewis Carroll [7], and followed by a number of economists [24, 19, 22]. Nowadays, many companies/parties also implement practical applications of liquid democracy, such as Google votes [11], Pirate Parties (software: liquid feedback) [2], etc. However, they are still centralized organization, where black-box operation and statistical error are inevitable.

[1] According to [12] Black Box Voting is: "Any voting system in which the mechanism for recording and/or tabulating the vote is hidden from the voter, and/or the mechanism lacks a tangible record of the vote cast"

The introduction of blockchain [21] and on-chain smart contract technology [25] satisfies the openess and transparency requirement of voting system. Generally speaking, a blockchain is a decentralized and immutable public ledger ensured by cryptography and P2P networking, storing data including transaction information which is observerble by any user. The smart contract is a pre-designed instruction set for storing and operating on-chain data, led by Ethereum. The source code of the voting system can be deployed on the Ethereum mainnet through smart contract and invoked through on-chain transactions. The decentralization of blockchain gurantees that the voting system are impartially executed without any need of trusted third party, eradicating black-box operations.

A fundamental requirement of on-chain voting systems is realtime self-tallying, which states that, for each incoming voting message, the contract itself can record ballots[2] and update the voting status (and display it). The self-tallying property skips the trouble to download the whole Ethereum data (and use it to off-chain compute the results, mainly for those who do not participate the voting), or to collect majority's signatures confirming a specific voting status.[3]

However, the main challenge for realizing on-chain liquid democracy is the limitation of gas fee on Ethereum mainnet, which is remain open[4]: executing a single instruction of smart contracts consumes a certain amount of gas, called gas fee, with the average about 10 thousand[5]. Whereas 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, 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 number of instructions can not exceed 10 million/10 thousand = 1000, otherwise the transaction will be refused.

The difficulty lies in the computation of two pieces of critical informa-

[2]We uses ballot to denote the number votes that a candidate receives.

[3]With the self-tallying property, users can visits the variables of a smart contract by simply send RPC requests to Ethereum full nodes. Otherwise, without the support of any centralized parties like *etherscan.io*, users need to download the full Ethereum in order to obtain historical visiting data of the smart contract (current size more than 3.4TB).

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

[5]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.

tion upon receiving a voting massage: i) the actual voting power that the voter exercises, which would be reduced since some of his delegators may already cast a direct vote, and ii) the change other candidates' ballots, since the voter's direct vote also reduces the actual voting power of his delegate. Naive algorithms usually compute the two values by traversing through the delegate graph, of which the on-chain time complexity is $O(n)$ for processing each voting message, where n is the number of voters. Especially, when the delegate graph is chain-like, they are undesirable due to the limitation of gas fee, since application scenarios are limited to less than one thousand voters (usually votings with millions of voters are required)

Some discussions try to solve the challenge by add restrictions to the delegate graph, i.e., only allow the delegate graph with the max-depth less than 100. This modification essentially limits voters' behavior, which is lack of user friendliness and does not catch the core of democracy. Moreover, this method is vulnerable under attack: suppose Bob wants to prevent Alice from delegating to anyone, he can create 98 accounts and form a delegate chain, with the top node of the chain delegating to Alice. (Creating new accounts is zero-cost in Ethereum) Anytime Alice delegates to a node, a 100-depth chain is generated, thus the delegation will be refused. Other solutions abound but all unreasonable when meet with the gas fee limitation.

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 democracy problem. Our algorithm does not add any restriction to voters: any voters can delegate arbitrary. Our algorithm's off-chain time complexity is also acceptable, only $O(n)$.

Our algorithm mainly depends on two techniques, the Merkel tree, an on-chain storage method, and the interval tree, a data structure. Our algorithm solves the liquid democracy problem with the following aspect:

- At the beginning of a voting, each voter obtains the delegate graph by snapshotting the current height of Ethereum, then executes a $O(n)$ off-chain initialization to get his initialization data.

- Each voter is not allowed to change his delegate within the period of a voting, but he can directly vote to a candidate by send a voting massage, attached with his initiation data. The Merkel tree method checks the correctness of the initiation data with $O(\log n)$ time complexity.

- Upon receiving a voting message, our algorithm requires $O(\log n)$ time

complexity for updating/displaying the voting status and storage, through the interval tree structure.

Our on-chain algorithm excluding the Merkel tree part also enhances the off-chain liquid democracy: if each voter votes once, then the time complexity is $O(n^2)$ for traversal algorithms, while our algorithm is $O(n \log n)$. We focus on the on-chain situation in the following of this paper.

1.1 Related Work

The *Liquid Democracy Journal*[6] collects many valuable literature about the liquid democracy problem, which begins at 2014 and almost information about latest progress can be found there. Blum and Zuber[4] give an overview liquid democracy, include the concepts, the history and problems. Recently, a few technical papers also are interested in liquid democracy. Anson et al. [14] analyze the problem that whether there exists a delegate voting that outperforms direct voting, for the situation where there are a correct candidate and an incorrect candidate. Brill and Talmon [6] study the case where a voter can delegate to several proxies and specify a partial order. They propose a way to overcome the complications of individual rational. Christoff and Grossi [8] analyze liquid democracy within the theory of binary aggregation, and consider the issues of individual rational and delegate cycle.

Recently, a series of literature studies the implementation of on-chain voting system [10, 13], some of which also refer to liquid democracy but not consider the self-tally requirement. The introduction of self-tally can be found in [15], 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.[26] introduce a self-tallying voting system by Ethereum smart contract, but do not consider the liquid democracy scenario. McCorry et al. [18] 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.

To the best of our knowledge, our paper is the first $O(\log n)$ algorithm solving the on-chain liquid democracy problem, while the algorithms in Google vote and liquid feedback work in following ways: Google vote's algorithm mainly bases on the work of Schulze's [23], which is a m^3 method for electing a winner, where m is the number of candidates. They also demonstrate

[6] https://liquid-democracy-journal.org/

that the system can implement liquid democracy on a social network in a scalable manner with a gradual learning curve. The basics of Liquid Feedback's algorithm comes from Harmonic Weight[7], Proportional Runoff[8] and Schulze method, whose proposes are to determine the weights of candidates. Though both algorithms can be applied to liquid democracy, the self-tallying requirement and gas fee limitation are not taken into consideration.

2 Preliminary

2.1 Problem Description

Suppose there are n voters, indexed by numbers $1, 2, ..., n$, and m candidates, indexed by capital letters $A, B, C, ...$ We separate the liquid democracy problem into three periods:

- **Spare period** During spare period, no voting is hold. Each voter can arbitrarily delegate, undelegate and change delegate, by sending a massage (transaction) to the blockchain, which is stored in the delegate contract. Each voter is allowed to appoint at most one delegate.

- **Prepare period** In prepare period, a specific voting is to be hold. The holder needs to deploy the voting contract and the delegate graph and each voters voting powers need to be constructed. In the following example we regard the delegate graph as input, while in the next section we will show how the voting powers are determined and how to handle the case where there is a cycle in the delegate graph.

- **Voting period** After the voting begins, each voter can directly vote to a candidate by sending a voting message, with all his delegators' voting powers also cast to that candidate (which may also reduce the vote of the candidate that his delegate votes). The on-chain voting status are updated for each voting message and need to be displayed. For convenience, we assume that each voter can only vote once during each voting activity, while our algorithm also fits for the case where each voter can change his vote. A voter's delegate is not allowed to change during voting period.

[7]https://en.wikipedia.org/wiki/Harmonic_mean
[8]http://www.magnetkern.de/prop-runoff/prop-runoff.html

It is notable that, although our algorithm does not allow voters to change their delegates during the voting period, they can accomplish the same thing by casting a direct vote. Changing delegate to a voter that has voted is equivalent to casting a direct vote and changing delegate to a voter that has not voted is rare in practice. Actually, a voter wants to change his delegate during the voting period usually because he dislikes the way his delegate votes, which means that he has a better candidate in mind and is better off to vote by himself, and he has the time to do so.

The following example abstracts how voting status changes upon receiving voting massages during voting period. Suppose a (direct, no-cycle) delegate graph $G = (V, E)$ is given, where each node represents a voter, and a direct edge (u, v) represents that voter v delegates to voter u. We will use terms "voter" and "node" interchangeably in the following of this paper. Since by assumption there is no cycle in G, thus G is a forest (multiple trees). For convenience, we add a virtual node (indexed 0) that is pointed by the root of each connected branch. So G is transferred to tree T, as Figure 1. That is, there are 12 voters. We further assume that each voter's voting power equals to his index.

- At the beginning, nobody votes. When voter 1 votes for candidate A (as the first voter), A obtains $1 + 2 + ... + 12 = 78$ votes.
- After voter 1 votes, suppose voter 5 (as the second voter) votes for candidate B. Then B obtains $6 + 5 = 11$ votes. A's vote decreases by 11, turning into 67.
- Further, voter 3 (as the third voter) votes for candidate C, then C obtains $3 + 4 + 7 + 8 = 22$ votes, A's vote becomes 45, and B's vote is still 11.

Goal: For each voting massage, display the votes of all candidates. (Suppose $m < 100$)

input	output
1 A	A 78 B 0 C 0
5 B	A 67 B 11 C 0
3 C	A 45 B 11 C 22

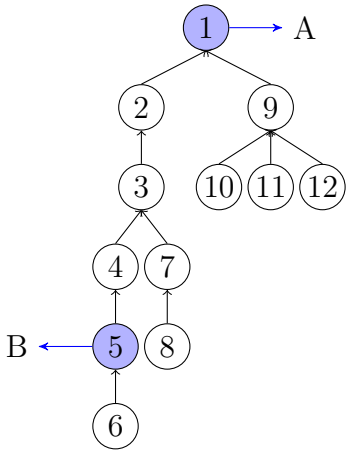

Figure 1: Tree T. We ignore the virtual node with index 0 here.

2.2 Blockchain and Smart Contract

The smart contract of Ethereum supports Turing-complete programming language, which is deployed on the blockchain [5]. Users invoke a smart contract by sending a transaction to the smart contract's address, which contains additional information including the gas fee of the transaction and other incoming parameters, which would further be included in a block. As shown in section 1, the gas fee of a valid transaction are limited by the fixed parameter block_gas_limit, so that the number of instructions of the smart contract are also bounded. That is the so-called "on-chain" time complexity. In this paper, we use "voting massage" to represent the transaction that invokes the voting contract, whose on-chain time complexity are required to be sub-linear to the number of voters. Ethereum clients obtain latest on-chain data through P2P network and implement smart contracts locally through the Etheruem virtual machine (EVM). Thus, the space limitation of a smart contract only depends on Ethereum nodes' (clients) local storage, which is not an issue in this paper.

It is important to distinguish on-chain smart contract and (open source) cloud computation. The later is still realized by centralized servers, which can not guarantee the codes are correctly executed, while in Ethereum, there is no "center" for executing smart contract: they are executed by every Ethereum node, which is reliable as long as the majority of nodes are honest.

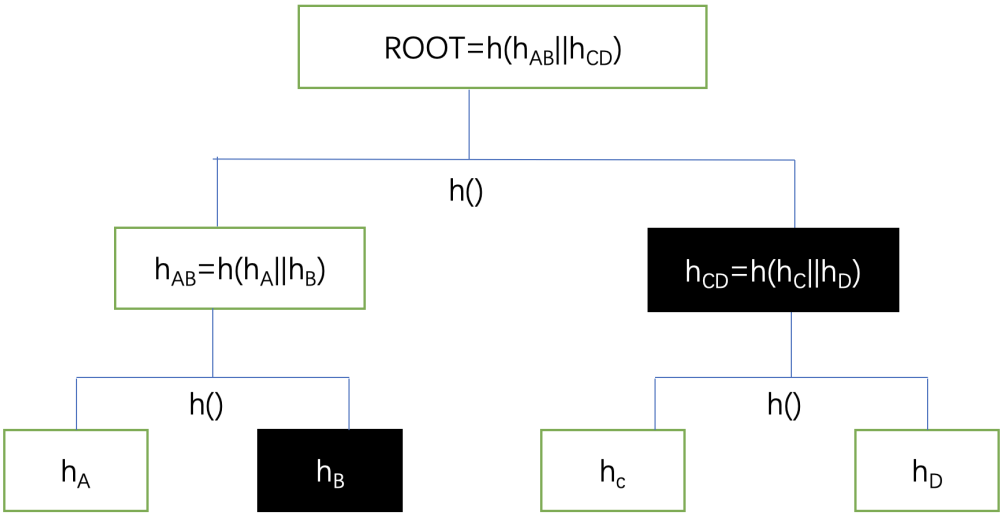

Figure 2: Merkel tree, where $h_{A/B/C/D}$ refers to the hash value of data $A/B/C/D$. The black nodes represent the Merkel path of data A.

2.3 Merkel Tree

The Merkel Tree is a common used method for store and verifying on-chain data. One of the key tools is the hash function, $h()$, which is (cryptographically) hard to find collisions and for inverse computation (In Ethereum, SHA3-256 is used). The Merkel tree is a full binary tree, where each leaf node stores the hash value of data to be stored (see Figure 2.3). The value of an intermediate node is the hash value of the combination of its two children.

The use of Merkel tree is that, the blockchain only need to store the root of the Merkel tree. In order to proof that a data (say data A in the Figure 2.3)belongs to the Merkel tree, A along with its Merkel path (also called Merkel Proof) are required:

Merkel path, which is defined to be a sequence of nodes in the Merkel tree that corresponds to brother of each node on path from A's leaf node to the root. For example, data A's Merkel path is (h_B, h_{CD}). A leaf node together with its correct Merkel path can recover the root of the Merkel tree (called Merkel root). The length of the Merkel path and the time complexity for recovering the Merkel root are all logarithmic to the number of leaf nodes of the Merkel tree. The one-wayness of the hash function guarantees that it

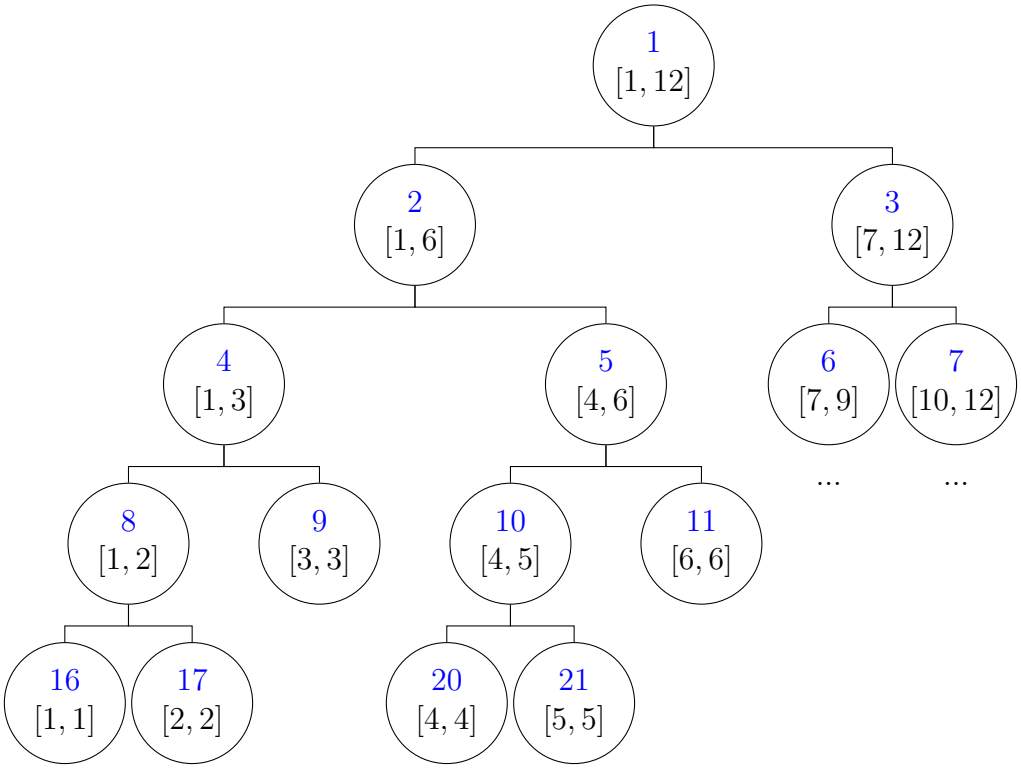

Figure 3: The interval tree of all nodes

is hard to construct a correct Merkel path for any data that does not belong to any leaf nodes of the Merkel tree.

2.4 Interval Tree

Interval tree is also a binary tree, where each node represents a interval and the interval of a parent node is uniformly distributed to its two child nodes, until the interval becomes a singular, to be a leaf node. See Figure 3 for the interval tree of the 12 nodes in Figure 1. The root are indexed 1 and for node k, its child nodes are indexed $2k, 2k + 1$ respectively. Note that, although some indexes, e.g. 18,19 in Figure 2.3, does not exist, the space is still used.

Interval tree is usually used for maintaining an array where the operations are aiming at a successive interval. Given the interval to be updated, the execution begins at the root node, then the interval are separated to one or

two sub-intervals. which are recursively executed at the child notes, guaranteeing that the sub-intervals to be update belongs to interval of current node. The recursion ends when the interval to be update equals to the interval of current node. For example, when interval [4,8] is to be update, beginning at the root, it separates to [4,6] and [7,8], which are executed at note 2 and 3 respectively. The recursion ends at node 5 and node 12 (which are omitted in Figure 3).

Interval tree supports find and update operation. For update operation, usually not every leaf node is updated since the update information may stop at an intermediate node, recorded as lazy-tag, which means that it is temporarily suspended and will be executed in subsequent operations. Find 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 [20], Chapter 10.1 to readers for more detail.

3 Algorithm

3.1 Overview

In order to real-time display the voting status, the core of our algorithm is to record and maintain each node's "lost voting power", initially valued zero. 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 democracy problem can be solved.

See again the example in Figure 1,

- After voter 1 votes for candidate A, all other voters' lost voting powers do not change.
- After voter 5 votes for candidate B, B's votes actually increase by $11 - 0 = 11$ (voter 5's total voting power minus lost voting power). Meanwhile, voter 2,3,4's lost voting powers all increase by 11. Lost voting powers of other voters *that have not voted yet* do not change (lost voting powers for voters that have voted is meaningless since each voter can only vote once).

- After voter 3 votes for candidate C, C's votes actually increase by $33 - 11 = 22$ (voter 3's total voting power minus lost voting power). Meanwhile, voter 2's lost voting power increase by 22. Lost voting powers of other voters *that have not voted yet* do not change.

It is not hard to get the following observation:

Observation 1. *When a voter votes, a node's lost voting power needs to be updated if and only if the node lines on the path from the voter to the voter's nearest parent node that has voted (we call it nearest voted parent for short, and voter 0 is regarded as voted. Note that in a tree, there is one and only one path from a node to one of his parent).*

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

So the main two goals of our algorithm are:

1. Find the voter's nearest voted parent.
2. Update lost voting powers of nodes on the path from the voter to the voter's nearest voted parent.

However, both 1 and 2 need to traverse the graph in traditional method, whose time complexity is $O(n)$ when the depth of the graph is high (say, chain like). Our solution is to use the interval tree to achieve $O(\log n)$, and use Merkel tree to for initialization.

The following two sequences are used, the preorder sequence and the bracket sequence.

- *Preorder sequence*, that is, traverse the tree in the order of $root \to leaf$ and record the nodes. In Figure 1, the preorder sequence is 1 2 3 ... 12, that is, each node's index equals to its index of the preorder sequence.
- *Bracket sequence*, that is, still traverse the tree in preorder but record the nodes when entering and exiting it respectively. In Figure 1, 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.

Our algorithm consists of the following three parts.

- In prepare period, each voter locally do an initialization of all voters' data, including their total voting powers, index of the preorder sequence, index of the bracket sequence and so on. The data are submit to the voting contract together with voters' voting massages, and are checked by the Merkel root (Section 3.3).
- For each voting massage, find the voter's nearest voted parent, based on the interval tree with respect to the preorder sequence (3.5).
- For each voting massage, update lost voting powers of nodes on the path from the voter to the voter's nearest voted parent, based on the interval tree with respect to the bracket sequence (3.6).

3.2 Notations

We have the following variables:

- T: The delegate 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.stake*: 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 (showed 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 votes 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.

All variables are global and initially valued 0 unless otherwise stated. For other intermediate variables we will illustrate in the following subsection.

3.3 Spare and Prepare Period

Spare Period

A perpetual smart contract, called the *delegate contract*, are established. It has two methods:

- $Delegate()$, voters call this method to appoint a delegate, or to undelegate (by delegate to an empty address). Before that, we recommend a protocol that each voter locally downloads the blockchain data and checks whether his delegate operation generates a cycle in the delegate graph. If so, the voter should change his delegate. The protocol can be integrated in the client.
- $Vote()$, a holder calls this method to start a voting, and deploys a new smart contract, called the *voting contract*, which will be introduced in the following. After that, the prepare period begins.

Prepare Period

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

delegate and stake. Then, each voter involved in the voting locally constructs the delegate tree T and gets all voters' voting powers according to the following rule.

- For each voter, get his last delegate operation from the snapshot and add the corresponding direct edge to the delegate graph. Then for all edges that are on a cycle, delete the latest edge. Then repeat the deletion until the delegate graph has no cycle[9]. After that, add an edge from each zero-outdegree node to the virtual node, resulting in the delegate tree T. Since the rule is deterministic, all voters obtain a same delegate tree. We show in next section that the construction rule is incentive compatible for voters,

- Usually in a decentralized authority organization (DAO), a voter's voting power equals to one of his stake on the blockchain (say, one kind of ERC-20 token). So all voters' voting powers can be obtained from the snapshot (*node.stake* in the notation). There are other off-chain method in practice to distribute voting powers, which is not critical of our paper as long as all voters can reach an agreement.

b) After the construction of T, we use $T.root$ to denote the root of T, i.e., the virtual node. Then, each voter locally call $Preorder(child)$, to obtain initialization data. (n and n_0 is initialed -1 to exclude the virtual node).

Intrinsically, a preorder traversal are executed, and all nodes' initialization data are computed simultaneously.

c) Each voter construct the Merkel root according the initialization data, where the information of each leaf node is *hash(node.adddress, node.power, node.index, node.endpoint, node.leftbracket, node.rightbracket)*, representing the initialization data of a voter. The leaf nodes are ordered according to *node.index* so that each voter's local Merkel tree are identical. The Merkel root are hard-coded in the voting contract. (If the voting holder makes a mistake of the Merkel root, every voter can choose to ignore the voting contract and remind the holder to deploy a new contract)

After step (c), the prepare period is over and the voting period begins.

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

Procedure *Preorder*(*Node root*);

$n \leftarrow n + 1$;
$n_0 \leftarrow n_0 + 1$;
$root.left \leftarrow n_0$;
$root.index \leftarrow n$;
$root.power \leftarrow root.stake$;
for *node in root's direct child* **do**
  $Preorder(node)$ $root.power \leftarrow root.power + node.power$;
$root.endpoint \leftarrow n$;
$n_0 \leftarrow n_0 + 1$;
$root.right \leftarrow n_0$;

3.4 Voting Period

In this subsection, we describe the voting contract to process each direct voting massage.

d) When a voter casts a direct vote, he sends a voting message which contains *(data, proof, node.candidate)*, where *data=(node.power, node.index, node.endpoint, node.leftbracket, node.rightbracket)* (here the node corresponds to the voter), the initialization data about himself.

e) Upon receiving a voting massage *(data, proof, node.candidate)*, the voting contract first obtains the sender's Ethereum address, to check if it matches with *node.address* in *data*. If it matches, then the contract recovers a root according to *data* and $proof$, and checks if the result matches to the Merkel root stored in the contract. If matches. the contract begins to process the voting massage, otherwise returns an "error" response.

f) The Algorithm 1 shows the main procedure for processing a voting massage, which consists of the following instructions:

- Compute the voter's lost voting power.
- Define t to be the voter's total voting power minus lost voting power, which represents his actual votes.
- Find the voter's nearest voted parent, and then update other voters' nearest voterd parent (only the voter's children are affected).

Algorithm 1: Vote: upon receiving a voting message

Input: *node*: voter

Input: $data, proof, node.candidate$

if *not check(RootHash, proof, data)* **then**
  return;

$b[node.index] = node$;

$update2(node.left, node.left, 1, 2n, 1, 0)$//Find the value of node's leftbracket;

$update2(node.right, node.right, 1, 2n, 1, 0)$//Find the value of node's rightbracket;

$int\ t = node.power - s[node.left] + s[node.right]$;

$C[node.candidate] += t$;

$update1(node.index, node.index, 1, n, 1, 0)$//Find the node's nearest parent;

$Node\ parent = b[nearestparent[node.index]]$;

$C[parent.candndate] -= t$;

$update1(node.index + 1, node.endpoint, 1, n, 1, node.index)$//Update nearest voted parents;

$update2(parent.left, node.left, 1, 2n, 1, t)$
//Update scores;

Output $C[]$

- The ballot of the candidate that the voter's nearest parent votes decreases by t.
- Update the lost voting powers of the nodes on the path from the voter to the voter's nearest voted parent.

So far, the total procedure of algorithm is produced. In the next two subsections we will introduce the two functions $update1(), update2()$

3.5 Find the Nearest Voted Parent

The function $update1()$ achieves the goal of finding the voter's nearest voted parent, by implementing the update operation of the interval tree with respect to the preorder sequence. The lazy-tag of the interval tree's leaf node records the preorder index of corresponding voter's nearest voted parent.

The following three observations are sufficient for the correctness:

Observation 2.

- *A node's preorder index is always larger 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.

- *update1(node.index+1, node.endpoint,1,n,1,node.index)* uses the voter's preorder index to do maximum-value update to (preorder indexes of) all its children's nearest voted parent (if the current value is less than the incoming parameter, then replace).
- *update1(node.index,node.index,1,n,1,0)* finds and records the node's nearest voted parent (recorded in $nearestparent[]$). The value used for update is zero since no additional updating is needed (just trigger the pass-down operation of lazy-tags). Note that if there is no voted parent then 0 is record, the default value.

```
procedure update1(int L, int R, int l, int r, int k, int v)
//[L, R] are the interval to be updated, [l, r] is the current interval of
  the interval tree node, k is the index of interval tree node and v is
  the value for updating.
if L = l and R = r then
    if v > lazy1[k] then
        lazy1[k] ← v
    if L = R then
        nearestparent[L] = lazy1[k]
    //Recursion ends when updating interval equals to current
    node's interval, and then updating the value of the interval
else
    int m ← (l + r)/2;
    if lazy1[2k] < lazy1[k] then
        lazy1[2k] ← lazy1[k]
    if lazy1[2k + 1] < lazy1[k] then
        lazy1[2k + 1] ← lazy1[k]//pass down the lazy-tag
    if L ≤ m then
        update1(L, min{m, R}, l, m, 2k, v)
    if R > m then
        update1(max{m + 1, L}, R, m + 1, r, 2k + 1, v)
```

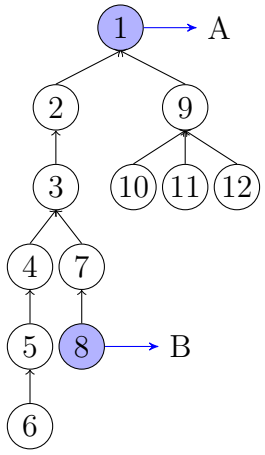

Figure 4: Updating a path

3.6 Update the Lost Voting Power

When a voter votes, all nodes on the path from the voter to its nearest voted parented should update their lost voting powers. See Figure 4, if voter 8 votes after voter 1 votes, then path $7 \to 3 \to 2 \to 1$ 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 for enter and one for exit, called left bracket and right bracket respectively. For Figure 4, 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 starts from u and ends at v, 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 3. *Given a direct path, for any node that does not lie on the path, it occur twice or does not occur in the path's bracket interval. For any node that lies on the path, it occur exactly once in the path's bracket interval. Moreover, only the node's first appearance lies in the interval.*

For 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 then 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 into the scores of the path's bracket interval. Then for a node's lost voting power, we can compute it by

$$s[node.left] - s[node.right]$$

The reason is that, when we increase the score, only the nodes on the path increase their lost voting powers (only the leftbracket increases). For a node outside the path, the scores either does not change or both its leftbraket and rightbracket 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 $update2()$ gives the implementation.

```
procedure update2(int L, int R, int l, int r, int k, int v)
if L = l and R = r then
    lazy2[k] ← lazy2[k] + v;
    if L = R then
        s[L] = lazy2[k]
else
    int m ← (l + r)/2;
    lazy2[2k] ← lazy2[2k] + lazy2[k];
    lazy2[2k + 1] ← lazy2[2k + 1] + lazy2[k];
    lazy2[k] ← 0 //pass down the lazy-tag;
    if L ≤ m then
        update2(L, min{m, R}, l, m, 2k, v)
    if R > m then
        update2(max{m + 1, L}, R, m + 1, r, 2k + 1, v)
```

- *update2(node.left,node.left,1,2n,1,0)* finds the score of the node's left-bracket (recorded in the array $s[]$). Similar for the rightbracket.

- *update2(parent.left, node.left,1,2n,1,t)* update the score of the bracket interval of the path from the voter to its nearest parent.

So far our algorithm is introduced. In the next section we prove some theorems.

4 Theorem

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

Lemma 1. *If a voter's delegate operation does not generate a cycle of the delegate graph (locally checked), then the corresponding edge will never be deleted.*

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

It means that, if the voter follows the protocol, then his delegation are garanteed to be retained, which is benefit for him. Otherwise if he deviates (his delegation generates a cycle), his delegation may be deleted. (It is also possible to be retained, if other voters further change their delegate and remove the cycle)

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

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

Since B is refused, it lies on a cycle which contains A. Since A also lies on another cycle, if A is delete, B still lies on a cycle, and should be refused according to the protocol. The lemma is proved. □

The lemma 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 delegate edge, which can be used in smart contract. However it is more complex and require more gas fee for each delegate massage. So still our protocol is recommended in practice.

Next, we introduce our main theorem:

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

The theorem is obvious according to the properties of the tools we used. Here we illustrate some issues.

- 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.
- 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 $lazy[2]$. Moreover, $lazy[1]$ and $lazy[2]$ still can be visit but always returned a default value 0, which is just the requirement of our algorithm.
- The time complexity of update operation in interval tree is $O(\log n)$, since at each level of the interval tree, at most two intervals are at the recursive state: after the interval to be updated is first time separated to two sub-intervals, there are at least one endpoints of the interval to be update are the same as the endpoints of the interval of current node. So at the next level, either there are only one interval to be executed, or there are two intervals, but one of them is identical to the interval of the next interval tree node, and the recursion ends.

For the part of Ethereum, we leave the proof in the experiment section.

5 Experiment

We compare between our algorithm and traversal algorithm on Ethereum test nest, recording the maximum consumed gas fee, from the following aspect:

- Random graph, random voting order.
- Chain, forward and backward voting order.
- Start graph (no delegation) any voting voter.

representing two extreme cases and one normal case. The number of voter ranges from 100 to 1000000.

TBA.

Acknowledgment

We thank YingLiu from Peking University for the advising the algorithm.

References

[1] Jonathan Bannet, David W Price, Algis Rudys, Justin Singer, and Dan S Wallach. Hack-a-vote: Security issues with electronic voting systems. *IEEE Security & Privacy*, 2(1):32–37, 2004.

[2] Jan Behrens, Axel Kistner, Andreas Nitsche, and Björn Swierczek. *The principles of LiquidFeedback*. Interaktive Demokratie e. V. Berlin, 2014.

[3] Sarah Birch. Full participation: A comparative study of compulsory voting. 2016.

[4] Christian Blum and Christina Isabel Zuber. Liquid democracy: Potentials, problems, and perspectives. *Journal of Political Philosophy*, 24(2):162–182, 2016.

[5] Thomas Bocek and Burkhard Stiller. Smart contracts–blockchains in the wings. In *Digital Marketplaces Unleashed*, pages 169–184. Springer, 2018.

[6] Markus Brill and Nimrod Talmon. Pairwise liquid democracy. In *IJCAI*, pages 137–143, 2018.

[7] Lewis Carroll. *The principles of parliamentary representation*. Harrison and Sons, 1884.

[8] Zoé Christoff and Davide Grossi. Binary voting with delegable proxy: An analysis of liquid democracy. *arXiv preprint arXiv:1707.08741*, 2017.

[9] Elisabeth R Gerber and Rebecca B Morton. Primary election systems and representation. *Journal of Law, Economics, & Organization*, pages 304–324, 1998.

[10] Rifa Hanifatunnisa and Budi Rahardjo. Blockchain based e-voting recording system design. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, pages 1–6. IEEE, 2017.

[11] Steve Hardt and Lia CR Lopes. Google votes: a liquid democracy experiment on a corporate social network. 2015.

[12] Bev Harris and David Allen. *Black box voting: Ballot tampering in the 21st century*. Talion Pub., 2004.

[13] Friorik P Hjalmarsson, Gunnlaugur K Hreiarsson, Mohammad Hamdaqa, and Gisli Hjalmtysson. Blockchain-based e-voting system. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 983–986. IEEE, 2018.

[14] Anson Kahng, Simon Mackenzie, and Ariel D Procaccia. Liquid democracy: An algorithmic perspective. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[15] Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In *International Workshop on Public Key Cryptography*, pages 141–158. Springer, 2002.

[16] Tadayoshi Kohno, Adam Stubblefield, Aviel D Rubin, and Dan S Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 27–40. IEEE, 2004.

[17] Thad Kousser and Megan Mullin. Does voting by mail increase participation? using matching to analyze a natural experiment. *Political Analysis*, 15(4):428–445, 2007.

[18] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375. Springer, 2017.

[19] James C Miller. A program for direct and proxy voting in the legislative process. *Public choice*, 7(1):107–113, 1969.

[20] Ketan Mulmuley. Computational geometry. *An Introduction Through Randomized Algorithms. Prentice-Hall*, 1994.

[21] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[22] Alois Paulin. Through liquid democracy to sustainable non-bureaucratic government. In *Proc. Int. Conf. for E-Democracy and Open Government*, pages 205–217, 2014.

[23] Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36(2):267–303, 2011.

[24] William S. U'Ren. Government by proxy now. *New York Times*, 1912.

[25] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[26] Xuechao Yang, Xun Yi, Surya Nepal, and Fengling Han. Decentralized voting: a self-tallying voting system using a smart contract on the ethereum blockchain. In *International Conference on Web Information Systems Engineering*, pages 18–35. Springer, 2018.