Liquid Democracy

ASResearch[1]

1 Introduction

Democracy has always been a widely concerned topic. Voting activity, 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. Meanwhile, various issues emerge due to the current voting system: low participation, black-box operation and bribery election. 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 power in every voting activity, 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 their vote. Those delegates can further proxy their votes to other people, creating a directed network graph (called delegate graph). Whenever delegate votes to a candidate (or a proposal to 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 themself or pick another delegate for the next vote. Applying liquid democracy system

[1]`https://asresearch.io/`

significantly reduces time costs of voters and increase 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 [The Principles of Parliamentary Representation. London: Harrison and Sons.], and followed by a seires of economists (William S. U'Ren 1912, Gordon Tullock, James C. Miller 1967, Martin Shubik 1970). Nowadays, many companies/parties also implement practical applications of liquid democracy, such as Google votes [], Pirate Parties (software: liquid feedback) [Liquid democracy: Potentials, problems, and perspectives], etc. However, they are still centralized organization, where black-box operation and statistical error are inevitable.

The introduction of blockchain and on-chain smart contract technology satisfies the openess and transparency requirement of voting system. Generally speaking, a blockchain is an 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 voting system can be deployed on the Ethereum mainnet through smart contract and invoked through on-chain transactions, where each voter can obtain the voting status and check the availability from etherscan[2] or other Ethereum nodes. The decentralization of blockchain gurantees that the voting system are impartially executed without any need of trusted third party, eradicating black-box operations. Recently, a series of literature studies the implementation of on-chain voting system[block chain based e-voting, record..][AAAI19algorithmic], some of them take liquid democracy into consideration.

However, the main challenge for realizing on-chain liquid democracy is the limitation of gas fee on Ethereum mainnet, which is remain open[3]: Executing a single instruction of smart contracts consumes a certain amount of gas, called gas fee, with the average about 10 thousand[4]. 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, which means that number of instructions can not exceed 10 million/10 thousand = 1000,

[2]etherscan.io

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

[4]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 cost 20000 gas.

otherwise the transaction will be refused.

The difficult lies in the real-time display of voting status, i.e. the votes of each candidate. When a voter cast a direct vote, two pieces of information are critical: i) the actual voting power that he exercises, which would be reduced since some of his delegators may already cast a direct vote, and ii) the change other candates' votes, since his vote also influences 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 millions of voters' activity are required).

Some discussions try to solve the challenge by add restrictions to the delegate graph, i.e., only allow the graph with the max-depth less than 1000. This modification essentially limits voters' behavior, which is lack of friendliness and do not catch the core of democracy. Other solutions abound but all unreasonable when meet with the gas fee limitation.

In this paper we propose an algorithm that reduce 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 each voting activity, each voter obtains the delegate graph by snapshotting the current height of Ethereum, then executes a $O(n)$ off-chain initialization to get his initiation data.

- Each voter is not allowed to change his delegate within the period of the voting activity, but he can directly vote to a candidate by send a voting massage, attached with his initiation data. The merkel tree method check 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 algorithm also enhances the off-chain liquid democracy, as the on-chain algorithm excluding the merkel root part is enough: if each voter can vote 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.

2 Related Work

1. block chain and e-voting.

2. about liquid demo: [liquid democracy journal]: total overview

[Liquid Democracy: Potentials, Problems, and Perspectives]

Arxiv:[Binary Voting with Delegable Proxy: An Analysis of Liquid Democracy]

To the best of our knowledge our paper is the first ...

[Google vote and liquid feedback] –3.about shifting from on-chain to off-chain

3 Preliminary

3.1 Problem Description

Suppose there are n voters and m candidates. We separate the liquid democracy problem into two period:

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

- **Voting activity period** Before voting activity period, we suppose that the delegate graph and each voters voting powers are snapshotted, which can be regard as input. We will show in the next section how the voting powers are determined and how to handle the case where there is a loop in the delegate graph and return a no-loop graph. Then after the voting activity 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. The on-chain voting status are updated for

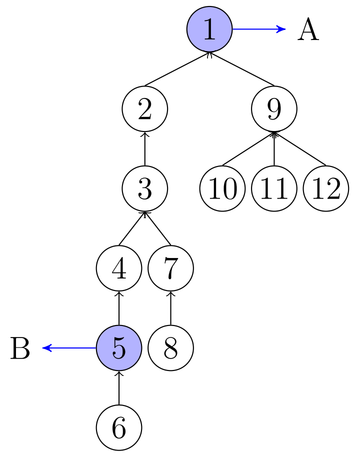

Figure 1: Tree T. We ignore the virtual node with index 0 here.

each voting message. For convenience, we assume that each voter can only vote once during each voting activity, while our algorithm also fit for the case where each voter can change his vote.

The following example illustrates how voting status changes upon receiving voting massages during voting activity period: a (direct) delegate graph $G = (V, E)$ is given, where each note represents a voter, and a direct edge (u, v) represents that voter v delegates to voter u. Since by assumption there is no loop 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 3.1. 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 become 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

3.2 Blockchain and Smart Contract

The smart contract of Ethereum supports Turing-complete programming language, which is deployed on the blockchain[]. Users invoke a smart contract by sending a transaction to the smart contract's address, with 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 executions of the smart contract are also bounded, that is the so-called "on-chain" time complexity ($O(n)$ is unsatisfactory). Ethereum clients obtain latest on-chain data through P2P network and implement smart contracts locally through Etheruem virtual machine (EVM). Thus, the space complexity of a smart contract is not issue, which only depends on Ethereum nodes' (clients) local storage.

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 nodes, which is reliable as long as the majority of nodes are honest.

3.3 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 of stores the hash value of data to be stored (see Figure x). The value of a 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

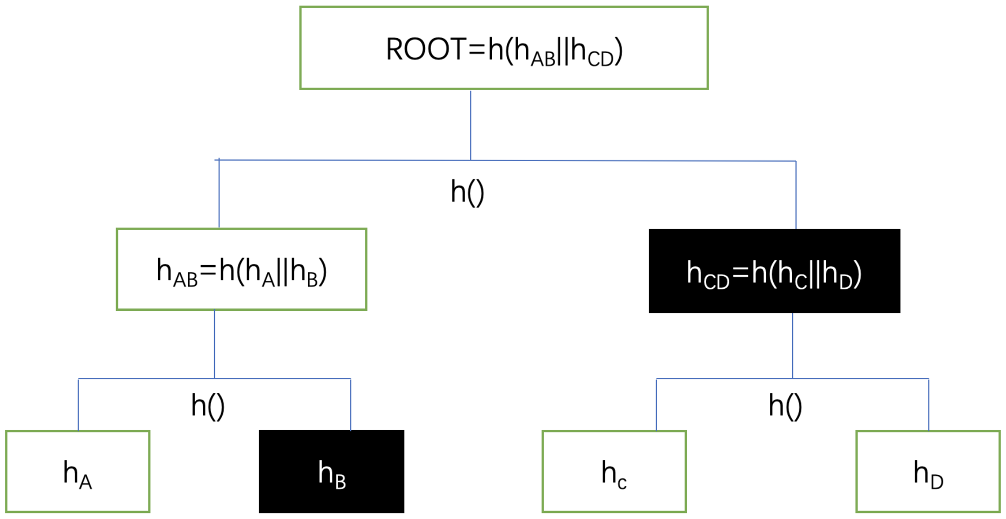

Figure 2: Merkel tree, where $h_{A/B/C/D}$ refers to the hash value of data $A/B/C/D$. The black nodes represent the Merkel path of data A

3.3)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 Merlek tree.

3.4 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 3.4 for the interval tree of 12 nodes in Figure 3.1. 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 3.3, does not exist, the space is

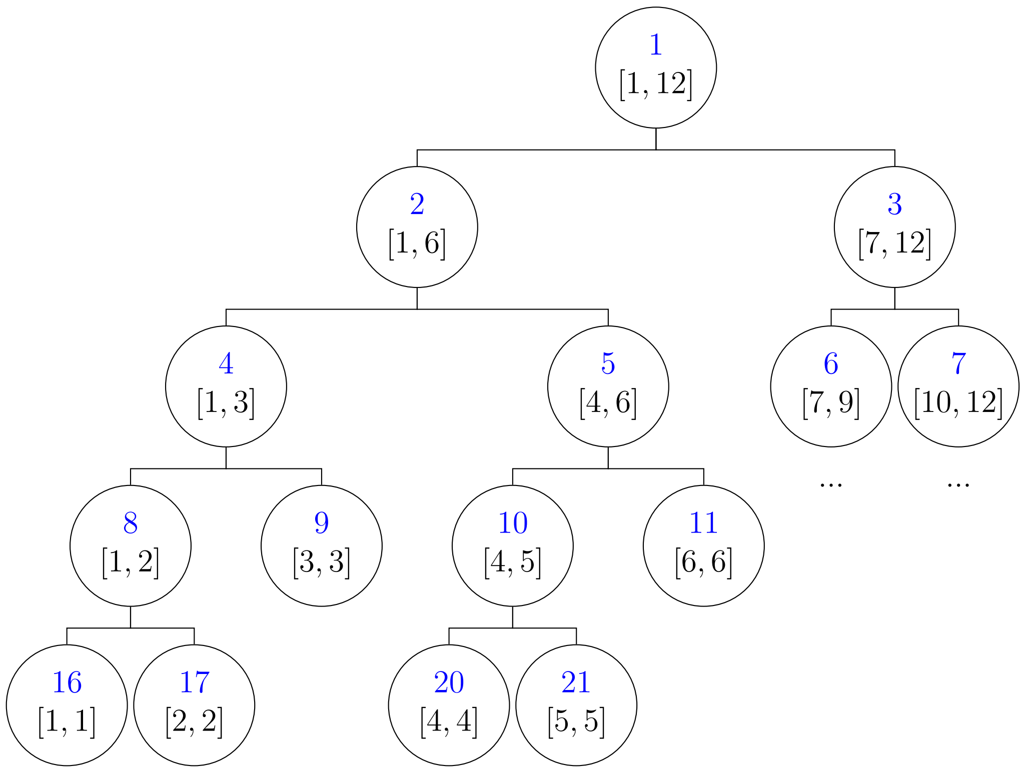

Figure 3: The interval tree of all nodes

still used.

Interval tree usually for maintaining a array where the operations are aiming at a continuous interval. The interval to update begins at the root node, then separates to one or two sub-intervals and recursively executed at the child notes, guaranteeing that the sub-intervals to be update belong to interval of the nodes. The recursion ends when the interval to be update equals to the interval of current nodes. 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 3.4).

Interval tree support find and update operation. For update operation, usually not every leaf node are updated since the update information may stop at an intermediate node, which is recorded as lazy-tag. Find operations need to be executed recursively starting from the root, and pass down all the lazy-tags until the leaf node is reached. For detial we refer [] to readers.