Liquid Democracy

ASResearch[1]

1 Problem Description

Suppose there are n voters, each voter has a certain amount of voting power $a[i], i = 1, 2, ..., n$.

The liquid democracy refers that, each voter (called delegator) can delegate all his voting power to another voter (called delegatee), and his degegatee can further delegate all those voting power to another delegatee. Whenever a delegatee votes to a candidate, by default, all his delegators' (including multi-level) voting powers are also cast to that candidate.

We use a (direct) graph $G = (V, E)$ to represent the delegate relationship among voters, where each note represents a voter, and a direct edge (u, v) represents that voter v delegates to voter u. Since by default there is no loop in G, thus G is a forest (multiple trees). For convenience, in the case where there are more than one connect branch, we add a virtual node that is pointed by the root of each branch. So G is transferred to tree T, as Figure 1.

That is, there are 12 voters, each node's (voter) parent represents its delegatee. As an example, now we suppose $a[i] = i$. At the beginning, nobody votes. When voter 1 votes for candidate A (as the first voter), A obtains $1 + 2 + ... + 12 = 78$ votes.

Meanwhile liquid democracy allows voters to change their votes when they are unsatisfied with the voting results of delegatees (including multilevel). Correspondingly, their delegatees' voting powers decrease.

As Figure 1, 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.

[1]https://asresearch.io/

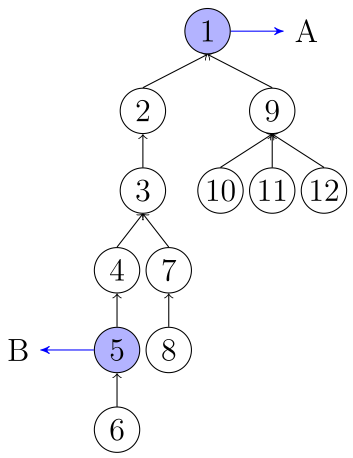

Figure 1: Tree T. We ignore the virtual node with index 0 here.

Suppose further, voter 3 (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.

1.1 Goal

Input $n < 10000000, a[i], T$. Each time input a voter and a candidate that he votes, output the current voting state of all candidates (with time complexity $O(\log n)$).

Example:

input	output
1 A	A 66 B 0 C 0
5 B	A 55 B 11 C 0
3 C	A 33 B 11 C 22

2 Algorithm

2.1 Overview

Our algorithm fully solves the liquid democracy problem. Compare to other algorithms discussed in the forum[2], it has the following feature:

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

- Our algorithm supports the tree with any structure, from a chain to star graph, without any restriction to max-depth.
- Our algorithm supports the realtime display of voting state (all candidates' votes), with on-chain time complexity $O(\log n)$
- Our algorithm requires on-chain space complexity $O(n)$ and off-chain time-complexity $O(n)$.

The core of the liquid democracy problem is the state transition when a voter votes. We record "lost voting power" for any node, representing the total voting power that its child nodes have cast, initialing 0. One of the key point is that, when a voter votes, his effective votes is his total voting power minus his lost voting power, and the voting operation **only affects the lost voting power of nodes on the path from its direct parent to its nearest parent that has already cast a vote (we call nearest voted parent).** Actually, the lost voting power of nodes on the path should increases by the amount of the voters effective votes. We use a data structure called interval tree to update.

Our algorithm consists of the following three parts.

- Initiation: including computing the total voting power of all nodes, getting the nodes' numbers in the pre-order sequence and so on. Require time complexity $O(n)$
- For each voting operation, finding the voter's nearest voted parent.
- For each voting operation, maintain nodes' lost voting powers.

We have the following variables:

- T: The liquid democracy tree, regard as input.
- n: Number of nodes
- $node.index$: Index of the node in the pre-order sequence.
- $node.address$: The address of each node.
- $b[n]$: Mapping from index to node.
- $nearestparent[n]$: Nearest voted parent of the nodes, with the index in the pre-order sequence.

- *node.endpoint*: Right endpoint of node's interval in the pre-order sequence.
- *node.votingpower*: Nodes total voting power (including its delegators').
- *node.candidate*: Recording the candidate that the voter votes.
- $v[]$: Recording the votes of candidates.

2.2 Initiation

It is not allowed to do on-chain initiation due to the $O(n)$ complexity, we can realize it through merkel root. The initiation process is done by the vote creator.

We first assign a index in the pre-order sequence for each node in the initiation part. Note that the pre-order sequence has the following property

1. A node always has index smaller than that of its child nodes.
2. For each nodes, all its child notes continuously appear after it in the pre-order sequence.
3. If a node's nearest voted parent is x, all its child nodes' nearest voted parent is x or a node with index larger than x.

```
Procedure $Preorder(Node\ root)$;
$n \leftarrow n + 1$;
$m \leftarrow m + 1$ $root.leftbracket \leftarrow m$//For bracket sequence;
$root.index \leftarrow n$;
$b[n] = root$;
for $node$ in $root$'s direct child do
    $Preorder(node)$
$root.endpoint \leftarrow n$;
$m \leftarrow m + 1$;
$root.rightbracket \leftarrow m$//For bracket sequence;
```

The preorder function is to get each node's index and corresponding interval for child nodes, in the preorder sequence.

The next step is to create a Merkel tree, and each leaf node is computed like this:

$$hash(node.address, node.endpoint, node.index).$$

The vote creator need to pass the Merkel root as parameter when creating the vote. And the Merkel root is stored on-chain.

2.3 Vote

For each voter, they need to get their information, like *endpoint* and *index*. They can achieve this by either contact the vote creator (through a web page) or do the initiation process themselves. Each voter needs to provide their information and also a Merkel proof when casting their vote. And the contract determines whether the provided information is correct since the Merkel root is already on-chain.

We then introduce the main algorithm, Algorithm 1. Note this algorithm is on-chain. To run this algorithm, the contract need to initiate several arrays (`mapping` in Solidity). Although the arrays' sizes are $O(n)$, they can be initiate with 0, which is the default value on Solidity.

Also note that $RootHash$ is already stored on-chain.

2.4 Finding Nearest Parent

In this subsection we realize the function that finding the a node's nearest parent.

Interval tree[3], is fits for operations that aim to a continuous interval. We construct an interval tree (Figure 2) with respect to the pre-order sequence and record each node's nearest voted parent. Each time a voter votes, we update the interval that represent the voter's child nodes in the interval tree.

We add a new global variable here.

- $lazy1[2n]$: Lazy-tag in the first interval-tree[4]

[3]`https://en.wikipedia.org/wiki/Interval_tree`

[4]Lazy-tag is a normal operation for interval tree: when we update an interval, we can not update all the leave nodes in the interval (otherwise the time complexity is $O(n)$). Instead, we first leave the updating information to an intermediate node, that known as lazy, when next time we need to find a node included in the interval, we then sink down the lazy-tag.

Algorithm 1: Vote

```
Input: $node$: voter
Input: $proof$: voter's Merkel proof
$h \leftarrow hash(node.address, node.index, node.endpoint)$;
if not check(RootHash, proof, h) then
    return;
$update2(node.leftbracket, node.leftbracket, 1, 2n, 1, 0)$//Find the value of node's leftbracket;
$update2(node.rightbracket, node.rightbracket, 1, 2n, 1, 0)$//Find the value of node's rightbracket;
$int\ t = node.votingpower - lazy2[node.leftbracket] + lazy2[node.rightbracket]$;
$C[node.candidate] + = t$;
$update1(node.index, node.index, 1, n, 1, 0)$
//Find the the node's nearest parent;
$Node\ parent = b[nearestparent[node.nunmber]]$;
$C[parent.candndate] - = t$;
$update1(node.index + 1, node.endpoint, 1, n, 1, node.index)$
//Update the first interval tree;
$update2(parent.leftbracket, node.leftbracket - 1, 1, 2n, 1, t)$
//Update the second interval tree;
Output $C[]$
```

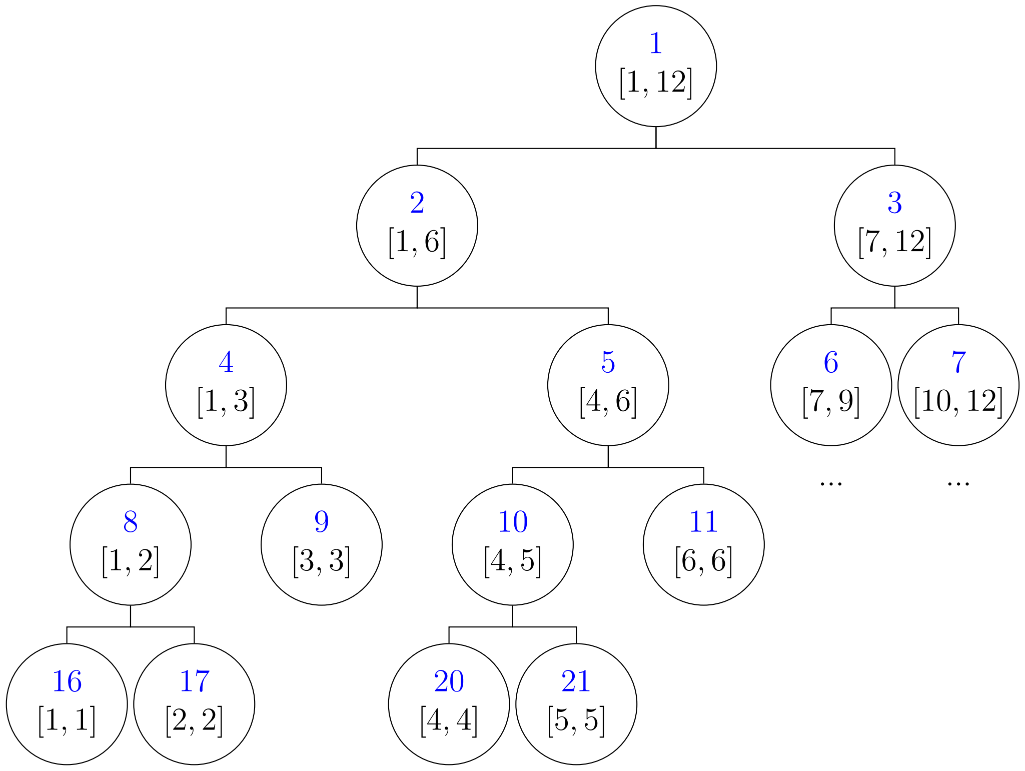

Figure 2: Interval tree, where the blue number in each node is the index of the node on the interval tree and the interval in each node represents the interval of the preorder sequence

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

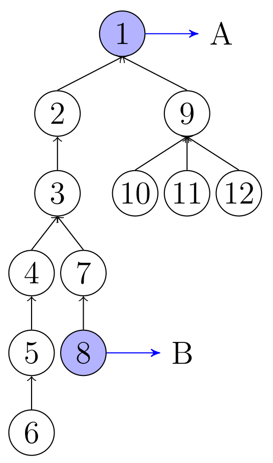

Figure 3: Updating a path

We use *update*1 to finding and update the interval tree. No extra code is needed for building the interval tree as we use $k, 2k, 2k+1$ to represent a node and its two children for a node in the interval tree.

The inherent property of interval tree guarantees that, there are at most two intervals in each depth that are recursively processed, so the time complexity for each updating is $O(\log n)$.

2.5 Updating 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 3, if voter 8 votes after voter 1 votes, then path $7, 3, 2, 1$ should be updated. However it is not a continuous interval in the pre-order sequence. We use the bracket sequence to handle this problem. A bracket sequence is to record each node twice in the pre-order execution, one for enter and one for exit, called left bracket and right bracket respectively. For Figure 3, the bracket sequence is

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

Let array $s[2n]$ record a value of the bracket sequence. When vote 8 votes, we let $s[1-10]+=8$, that is, the interval form node 1's left bracket to the node before node 8's left bracket.

The lost voting power of a node u is $s[u.leftbracket] - s[u.rightbracket]$.

To see why, it is not hard to find that, if a node does not occur on the path we need update, it occurs twice in the interval (1-10) of the bracket sequence, say node 4,5,6. If a node is to be update, it occurs only once in the interval (1-10) of the bracket sequence, say node 1,2,3. So from maintaining the array s we can maintain each node's lost voting power. Since now the operation is an interval again, we can use another interval tree to handle it.

We add the following variables:

- $lazy2[4n]$, the lazy-tag of the second interval tree.
- $node.leftbracket, node.rightbracket$

```
procedure $update2(int\ L, int\ R, int\ l, int\ r, int\ k, int\ v)$
//$L, R$ are the interval for updating, and $l, r$ are node's interval,$k$ is
 the index of the node on interval tree, $v$ is the value for updating
 the interval.
if $L = l\ and\ R = r$ then
    $lazy2[k] += v$
else
    $int\ m \leftarrow (l + r)/2$;
    $lazy2[2k] += lazy2[k]$;
    $lazy2[2k + 1] += lazy2[k]$;
    $lazy2[k] \leftarrow 0$ //sink down the lazy-tag;
    if $L \leq m$ then
        $update2(L, \min\{m, R\}, l, m, 2k, v)$
    if $R > m$ then
        $update2(\max\{m + 1, L\}, R, m + 1, r, 2k + 1, v)$
```