Quadratic Voting Using Solidity

What is quadratic voting?

Quadratic voting is a fairly new voting idea that aims to revolutionize the way that voting takes place. In this day and age, voting has become such a complicated and sometimes unfair process. Quadratic voting is attempting to end this confusion with a very simple idea.

The idea behind quadratic voting is that participants are either given an allowance, or use their own assets to vote on certain propositions. Placing one vote on a proposition will cost you one unit of your allowance (assuming you were given 100 tokens to spend on votes, placing one vote would cost you 1 token). If you feel strongly about a specific proposition, you are welcome to place more votes at a price. The cost of a vote is represented as such: $cost\ of\ vote = (number\ of\ votes)^2$. So, if you wanted to place 3 votes on Proposition 2, the total cost would be 9 tokens. It is important to keep in mind that the more you spend on one proposition, the less votes you can place on others.

Quadratic has quite a few benefits, especially when implemented with blockchain technology. The main benefit is the ease of use. Another great benefit is that you get to place a higher weight on propositions you are truly passionate about. When quadratic voting is implemented on blockchain technology, it becomes very easy to incentivize voters. One simple example of incentivisation, is that when the voting is over, all of the tokens held in the contract will be distributed evenly among the voters, no matter how much you spent on votes. This means that it is possible to make money from voting, but you must participate to receive anything.

Solidity Implementation

At the beginning of the contract we define two different structs, voter and proposal.

Voter:

- toPorposal: to keep track of which proposal you voted for.
- votesSentTo: to keep track of the address you sent your votes to (if you decided to do so).
- votesReceived: the number of votes that you have received from other addresses (that you haven't paid for). This allows for a way to get people to vote for the same proposition as you, without that person having to pay quadratic prices.
- votesPlaced: keeps track of the number of votes that you have paid for.
- canVote: keeps track of whether you have the authority to vote or not.
- voted: keeps track of whether you voted or not. This is used to make sure that you don't vote more than once to get cheaper addtional votes.

Proposal:

- voteCount: keeps track of the amount of votes a certain proposal has

```
struct Voter
{
    // proposal voted for
    uint8 toProposal;
    // holds address you sent your votes to (if you choose to)
    address votesSentTo;
    // amount of votes that have been sent to you
    uint8 votesReceived;
    // ammunt of votes you placed (and paid for)
    uint256 votesPlaced;
    // have you been given authority to vote
    bool canVote;
    // have you voted
    bool voted;
}

struct Proposal {
    uint voteCount;
}
```

We can also create an Event, called VoteCast, which will log every time a vote is placed and can be used for a user interface.

EventCast:

- _voter: keeps track of the address of the person who voted,
- _proposal: keeps track of the proposal the person voted for.
- _voteCount: keeps track of how many votes the person placed.

```
event VoteCast(
  address _voter,
  uint8 _proposal,
  uint256 _voteCount
);
```

We will also need to declare the following state variables which will keep the smart contract up to date with the necessary information.

- *chairperson:* the address which deployed and therefore owns the smart contract. This person will be able to give other addresses the right to vote.
- *voters:* this is a mapping of addresses to the Voter objects (the struct we created earlier).
- *proposals:* this is an array of all the proposals that will be initiated in the constructor.

- *allVoters:* an array of all the addresses that have voted. We need to keep track of all the voters to be sure that we are able to evenly distribute the tokens to everyone.

```
//state variables

//creator of contract
address chairperson;
//mapping of addresses that have voted
mapping(address => Voter) voters;
//array of all the proposals
Proposal[] proposals;
//array of all voters to divide winnings evenly
address payable[] allVoters;
```

After creating the necessary building blocks, we can continue on with the implementation of quadratic voting. We'll start with the constructor of the contract which will take the number of proposals as a parameter. In the constructor we will set the length of the proposals array to the value of the parameter. We will also set the chairperson of the contract to be the sender (the sender of the function is the one who instantiates the smart contract). And finally, we will also set the chairpersons canVote variable to true.

```
constructor(uint8 _numProposals) public payable
{
    proposals.length = _numProposals;
    chairperson = msg.sender;
    voters[chairperson].canVote = true;
}
```

The first function we will create is the giveRightToVote function. This function simply allows the chairperson to give a specific address the ability to vote. It takes the address that the chairperson wants to permit to vote as a parameter. It first uses a require function to make sure that the caller of the function is the chairperson. It will also make sure that the address provided has not already voted. If these requirements are met, the toAddress paramter will be allowed to vote.

```
function giveRightToVote (address toVoter) public
{
    //make sure chairperson is giving the right
    require(msg.sender == chairperson,
            "Sender does not have the authority to give right to vote");

    //make sure voter hasn't already voted
    require(voters[toVoter].voted == false,
            "This address has already placed a vote");
    voters[toVoter].canVote = true;
}
```

The next function, sendVotesTo, is a big part of the theory of quadratic voting. Since voting on a proposal multiple times can quickly become expensive, a good way to get cheap votes is to form a following of people who want to vote for the same proposal as you. For example, you could buy 10 votes for proposition 1 for a total cost of 100 tokens. But a much cheaper way to do that is to get 9 other people to send you their vote of 1 token to you. Everyone involved only pays 1 token, but there is still 10 votes on the proposal.

In order to implement this, the function will take two parameters, the *to* address and the *numOfVotes*. We must first make sure that the sender has not already voted. After, we have to calculate the cost of the vote (based off of *numOfVotes)* and make sure that the value sent is high enough to buy those votes. Once we check those factors, we also have to make sure that you sending a vote to someone else doesn't loop back to yourself. For example if you send your vote to person 1 and person 1 sends his vote to you, your vote simply loops back to you. After checking that, we must change the variables *votesSentTo* and *voted.* If the person you sent your vote to has already voted, simply add the number of votes to the proposal they voted for. If not, then add the number of votes to their *votesReceived* variable. At the end, we will add the sender to the allVoters array to make sure they get compensated for participating.

```
//give your vote to someone else
//allows you to send your vote to someone else, so that they receive more votes,
//but you pay for it
function sendVotesTo(address to, uint8 numOfVotes) public payable
{
    require(voters[msg.sender].voted == false,
            "Sender has already placed a vote");

    uint256 costOfVote = numOfVotes ** 2;

    require(msg.value >= costOfVote,
            "Value sent is not high enough to send votes");

    //check that you are not sending a vote to yourself
    while (voters[to].votesSentTo != address(0) && voters[to].votesSentTo != msg.sender)
        to = voters[to].votesSentTo;
    require(to != msg.sender,
            "Sender attempted to delegate vote to himself");

    voters[msg.sender].votesSentTo = to;
    voters[msg.sender].voted = true;

    //check if person sent to already voted
    if(voters[to].voted == true)
    {
        proposals[voters[to].toProposal].voteCount += numOfVotes;
    }
    else
    {
        voters[to].votesReceived += numOfVotes;
    }

    allVoters.push(msg.sender);
}
```

The next function, vote, simply allows an address to place its vote for a proposal. It takes two parameters, *toProposal* and *numOfVotes.* We first must make sure that the sender has not already placed its vote and that they have sent enough tokens in order to pay for the amount of votes specified, as well as checking that the proposal they entered is a valid proposal. If everything checks out, we will add the *numOfVotes* and the voters *votesReceived* to the specified proposals *voteCount*. We will also change the voters *voted* and *toProposal* attributes and add the voter to the *allVoters* array to make sure they will be compensated for particiating. Finally, we will log the vote by using the emit keyword.

```
function vote(uint8 toProposal, uint8 numOfVotes) public payable
{
    //make sure sender has not already voted
    require(voters[msg.sender].voted == false,
            "Sender has already placed a vote");

    //make sure sent value is high enough
    uint256 costOfVote = numOfVotes ** 2;
    require(msg.value >= costOfVote,
            "Value sent is not high enough to send votes");

    //make sure proposal is in bounds
    require(toProposal < proposals.length,
            "Not a valid proposal number");

    Voter storage voter = voters[msg.sender];

    proposals[toProposal].voteCount += numOfVotes + voter.votesReceived;
    voter.votesPlaced += numOfVotes;
    voter.toProposal = toProposal;
    voter.voted = true;

    allVoters.push(msg.sender);

    emit VoteCast(msg.sender, toProposal, numOfVotes + voter.votesReceived);
}
```

The last function we will be implementing is the winningProposal function. This function simply calculates and returns the winning proposal and divides the all the tokens received evenly between all the participating voters. It uses a for loop to go through all the voters and transfers their portion of the tokens to them.

```
function winningProposal() public returns (uint8 _winningProposal) {

    uint256 winningVoteCount = 0;
    for (uint8 prop = 0; prop < proposals.length; prop++)
    {
        if (proposals[prop].voteCount > winningVoteCount)
        {
            winningVoteCount = proposals[prop].voteCount;
            _winningProposal = prop;
        }
    }

    //divide winnings evenly
    uint256 amountPerVoter = address(this).balance / allVoters.length;
    for(uint8 voter = 0; voter < allVoters.length; voter++)
    {
        allVoters[voter].transfer(amountPerVoter);
    }
}
```