Abstract

My project will be a smart contract builder that allows a founder to create a decentralised sports club. This will use the blockchain to provide a trustless environment to collect and allocate money. It will also use multi agent system algorithms to speed up and rationalise the method of choosing representatives and deciding budgets. By combining multi agent systems and blockchain, users will be able to agree to the algorithms before hand and trust that they will be carried out accurately. The project will also look at the cost of running the algorithms and make sure it can scale with the number of representatives and users.

The project will also come with an example web interface that allows users to interact with the block chain without a terminal

Problem

Most university clubs collect registration fees to finance their activities for the year. The current solution for collecting the fees, is either with cash or online. The problem for both systems is that the money is collected by one entity, and cash has the added negative that the payment could be repudiated. At the annual general meeting (AGM) , members vote for representatives and for managers. A problem arises when the AGM is held before the start of semester one, and this means that many members and first years students, who are a large proportion of the club members, are unable to vote.

After representatives and managers are chosen, they will decide on several items where budget can be spent and how the money is allocated to those items. The disadvantage with this, is that deciding where the money should go, can take a long time. Another problem with the negotiation phase is that quiet individuals may be bullied into making concessions by someone more confident. Another problem is that currently a lot of systems use the budget negotiation algorithm of take all the representative’s budget and divide by the number of representatives. This is very easy to manipulate by overexaggerating one expenditure, in order to make that expenditure larger in the final budget. Lastly, the final budget may not be known by the entirety of the club, making it unaccountable and opaque.

Another problem is that current systems run on private servers and databases, and therefore users can not trust that the owners of the systems are not manipulating the data. Also the servers and databases require a upfront cost.

Goals

For the collection of the fees, I want my solution to have certain characteristics. First off, all registrations are non-repudiable. This means that there is a method for confirming that the user has paid the fee. Also, we can check the authenticity of a registration. This means that a user can identify himself as the one who paid the registration. Another feature is transparency; everyone should be able to see how much money has been collected from fees. Lastly, until the budget has been decided no one should be able to access the money.

For the voting, the features are as follows; Firstly, all members who have paid their registration fees should have equal voting power and the ability to apply as a representative. Another feature is the ability for a user to submit a full ranking of representatives, to allow him a larger ability to customize his vote. We also want to give each representatives a weight, to take into account the fact that some of them will receive more votes than their counterparts and therefore should have a larger mandate. Lastly, the system should discourage tactical voting, as we want people to submit their votes that represent their opinion, and it not swayed by how other people are voting.

For the negotiation and distribution of the budget, the beneficial characteristics are thus; representatives can submit sources of expenditures and their preferred budget. During this pre-negotiation phase, all messages sent to each other are recorded and non-repudiable, and this should increase accountability, because representatives will be held to the claims they made. The system will then run an automated negotiation algorithm, which solves the problem of representatives’ personality affecting the outcome. The negotiation algorithm should be deterministic and therefore other users can check the integrity of the budget. The algorithm should make it difficult to manipulate the final budget. Lastly for the distribution of the budget, after the negotiation, the money should be sent automatically to the managers of the expenditures.

Final Design

How Registration Fee Collected

One of my goals was that the owner of the system does not have any extra control over the money and the data. I will compare the use of a Clubs and Charites Account (Santander) with using a Smart Contract on the Ethereum Blockchain.

|  |  |  |
| --- | --- | --- |
|  | Clubs and Charites Account | Smart Contract |
| Signatories To Access Money | 3 | Can’t unless given the money by the buget. |
| Max Turnover | 250,000 | Unlimited |
| Cost to Run and Withdraw | 0 | Gas Used per Transaction \* Gas Price (Currently 3) |
| Registration is non repudiable | Legally yes, but practically nothing to stop someone from denying registration happend. | Yes |
| Able to prove you’re the one who registered | Yes | Yes |
| Transparency | No | Yes |
| Accountability | No | Yes |

The benefits of the Clubs and Charites Account is that it costs 0 to run and set up, whereas it can be expensive to store data on the Blockchain. The problem is that only 3 members are required to access the account. This is made worse as the signatories would have to change annually which would require a lot of time and effort. Also if one of the signatories is busy, then the club would be unable to access the funds. Compared to the smart contract, which would do this automatically. Another problem is that although the registration is legally non repudiable, a member would have to take the club to small claims to enforce it. Lastly the account is viewable by the signatories, therefore a regular member can not confirm how much money has been gathered and where it has spent. However a smart contract all members can see how much money has been gathered and where it has been spent.

From this I decided that the best design was a smart contract, however care must be taken to keep the Gas (how many instruction used) used down. Also to monitor the Gas Price (cost miners are changing to run a instruction) which this year to date has fallen by 70%, and the price of Ethereum in GBP. Also Another problem to consider is that if a user loses his private key to his Ethereum account it is unrecoverable and you will lose the ability to prove you registered.

How Registered Users Are Stored

A feature of blockchain is all data stored on it is publicly visible. Therefore, thought must be put into how much data is stored about each user. At minimum we must store the registered users public address. However storing only the users address will make it so communication and accountability is hindered, as a public address is long and non memorable, and will make it hard to see who said and did what. I decided that storing the address and name was the best compromise. In the rare situation two users had the same name and there was argument on who did what, it would not be too much effort for one of them to digitally sign using their private address.

Voting Algorithm

The voting algorithm to choose representatives is a social welfare function which takes a list of ordered preferences and outputs a ordered preference. I compared a few algorithms to see which best bit my problem.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Borda Count | First Order Copeland | Second Order Copeland | Dodgson’s method |  |
| Pareto efficient |  | Yes | Yes | Yes |  |
| Independence of Irrelevant Alternatives |  | No | No | No |  |
| Non dictatorship |  |  | Yes | Yes |  |
| Condorcet Condition |  |  | Yes | Yes |  |
| Complexity to work out |  |  | K^2 | NP-Hard |  |
| Complexity to Manipulate |  |  | NP-Complete | NP-Hard |  |

How to start and end Voting

Voting should start a “term length” after the last voting had ended and the voting should end after “voting length” after the voting had started. Both term length and voting length are set when the club contract is made. However the smart contract is not constantly running, and therefore needs to have a function called. I have two options for calling the functions. First I could use “Ethereum Alarm Clock”. This allows my smart contract to pay someone to run the function at a certain block time. The other option is waiting for one of the users to run the start and end vote functions. The first option has the benefit that we can assume that the voting period will end at a more precise time. The latter has the benefit that it is cheaper.

I decided to go for the latter, as in my case a few extra minutes or hours of voting, should not be a significant problem. And if it is, then users could send the function as quickly as possible. If the club has a small number of users the first option might be better as there is a greater chance all the users forget to end the voting period.

How many Representatives

An important design decision, is how many representatives should be elected. The trade off is between the system being more representative and the system being cheaper to run. Because the voting algorithm compares each candidate in a pair wise election, the gas cost is exponential to the number of representatives. In my system the number of representatives will be set when the contract is created. Also due to the fact in can have a large impact on the cost of the contract, if users wish to change it, a super majority should be acquired.

How to choose their Weight

After the voting algorithm is complete it will output a list of ranked representatives. From this point the system will need to determine how much weight they have in the budget negotiations. One example system could be weighting the 1st ranked representative, number of representatives, the 2nd, number of representatives – 1 and so on. Another system could be weighting them all one. These are common methods and found in numerous other systems and therefore my system should make it easy to choose these two methods. However in order to make the system more flexible, my system should allow the creator to also create their own function.

How many Sinks and who can create them

Another decision is how many sinks (budget expenditures), should be allowed to be created. Again the system will allow you to choose the number at contract creation. The benefit of more sinks is that the money will more distributed more and each manager will have more time to focus on a sink with less responsibilities. The disadvantage is that communicated between the managers may be poor, and double spending could occur. Another problem is that due to economies of scale, money concentrated in fewer hands may be able to afford more items . lastly it will raise the gas price of the budget negotiation. However since it only affects the cost linearly, the system should only require a plurality to change the number of sinks.

Also the system must have a system for submitting sinks. I have identified 3 ways to do this. First, all users have the ability, second all representatives have the ability and last, each sink is proposed and then voted on. The first option has the benefit that the system is more interactive for more users. However it is prone to abuse, as being a user only requires paying the fee, and a malicious person could fill up the sinks with bogus sinks. The second option is less like to be abused, as representatives are likely to have a greater attachment to the club. The last option should prevent all abuse, however it is time consuming to have to vote on each sink, and could end up costing a lot of money to run each vote. Therefore I have decided on my second option.

When can Budget submit end

Budget Negotiation Algorithm

(stuff moved around in latex)

The first and easiest idea for deciding the budget was to take each representatives budget, sum each one and divide by number of representatives. This would be cheap to run and sounds fair. The problem occurs with that all budgets submitted are publicly viewable, and with this algorithm a representative could easily manipulate his budget to achieve the final budget he wanted.

A better algorithm would be for each representative to try and form coalitions with his peers, until the coalition reached a certain quota. This solves the problem of a representative submitting a extreme budget to manipulate the final budget, as the other representatives would just form a coalition without him.

The next step was to create a function which took a representative’s budget, their current coalition’s budget, and the current size of the coalition and returned the representative’s current utility. To make the utility easy to understand, the range of the function will be from 0 to 100. 100 utility would be when the coalition size is the quota and the coalition budget is the representatives preferred budget. The function I decided on was as followed.

Next I had to decide how coalitions would be formed. The first step would be to initialise each representative with his own coalition with his preferred budget. From this point I had to decide whether representatives would take turns joining coalitions or coalitions would take turns inviting representatives. The negatives of coalitions inviting representatives is that larger coalitions would have more chance to have a representative stolen by another coalition and therefore take longer and cost more to calculate. The advantage is that there would be more negotiations and that should result in a more balanced budget. One problem with both of these solutions is that it could lead to a infinite loop and never end, however the joining coalitions method resulted in a lot less of them. Therefore I will use that method. Representatives will take turns, negotiating with each coalition, they will then compare their utility by joining their coalition and choose the coalition which offers the best utility, or stay in the current coalition if its better utility then all offers.

However any infinite loops are unacceptable in the algorithm. To solve this I came up with two solutions. First, representatives could be impatient, this means that a budget in round n is better than a budget in round m if n < m. Another method is to make the coalition size less relevant is earlier turns and make it more important in later turns. The first method allows representatives to look forward to later rounds in order to make an more informed decision. However to work out the single negotiation would be factorial, and therefore unusable with a medium number of representatives. For this reason I will use the second method. With the new utility function as follows.

The next decision.

Club contract Website

The design of this will be very minimalistic. This is because any one can connect to the smart contract on the blockchain and therefore this website should be used as a template. Also the website functions need be easy to copy and therefore I need to split my html and javascript code.

The website will allow users to connect their account with metamask. It will then allow them to run all the public functions of the contract, with textboxes to enter parameters, get the public variables, and run the getters to access my structs. Lastly it will have a ability to message and read messages.

Implementation

Setting up Development Environment

In order to create and migrate the smart contract onto the blockchain I used a tool called Truffle. This had boilerplate code which had a .sol contract file and a initial\_migration.js and a deploy\_contracts.js. After writing code in the .sol file and making sure to deploy in in the deploy\_contracts.js, you could then run truffle compile to create .json build files which contains fields such as the abi. This could then be deployed to the “real” blockchain, however while in development, Truffle has a tool develop, which allows you to write to deploy the smart contract to a local network. It also provides several addresses preloaded with Ethereum, this allows me to test my program without having to pay any money.

Club Contract Inheritance

I decided to split my Club smart contract into three, RegistrationClub, VotingClub and BudgetClub. With BudgetClub inheriting from VotingClub and VotingClub inheriting from RegistrationClub. The first reason this was done was to increase readability. Another reason is that I want to make it easier for developers to adapt my smart contract to their situation. E.g A completely different way of choosing representatives, would mean that only VotingClub needs to be changed.

I also created a library called MyLib which will contain all my structs.

Registration Club

This contract needs to take a registration cost when its created, and then allow users to pay that much Ethereum to register. It then needs to store that user in a array as a Struct containing the users name and address. I also require getters to return the number of registered uses, a certain registered user and the balance in the smart contract. This was done with the following code

Solidity the smart contract coding languages uses many function modifiers. Payable allows users to send extra gas with the contract. Constant is an important function modifier which states that the function will not modify the storage (global variables). Because it does not modify the storage the function is not run on the blockchain, and therefore costs no gas to run.

One problem with this registration is that users could register multiple times, with different addresses. This problem could be solved with a one time code that is given to prospective members when they wish to join. However that comes with its own problems as it centralises the control of the club. In the end, I decided that if users wished to pay more registration fees, then it would be fair to allow them greater representation of where that money went.

I also needed to add the functionality to allow users to change the registration cost if a super-majority of users wish it. First I added a global mapping which takes a registered user index and returns their current suggestion that user has for the new registration cost. When a new user registered I will set their suggestion to the current registration cost. After that I needed the ability to submit a new registration cost suggestion, and another to check all the submissions and check if a super majority of the same submissions has been reached. This was done with the code as follows.

An important part of the code is that checkRegistrationSuggestions() is constant and public. This is important as running changeRegistrationCost() costs gas even if no changes are made to the registration cost. Therefore a user only wants to run changeRegistrationCost() when he is certain that it will change the registration cost. A user can do this by running checkRegistrationSuggestions() e for free and seeing if the return is a value different than the current registration cost.

Voting Club

The basic features this needs to implement is the ability to start the voting, each user to submit one list of ranked users, end voting and then rank each user by their second order Copeland score. After ranking their score they will be given a weight according to the algorithm submitted (or with a default one). It will also take three variables, the registration cost, the term length and the number of representatives.

One problem that was spotted was that second order Copeland does pairwise comparison on each agent. This means we could be comparing users who have no votes which could cost a lot of gas. To solve this problem I added another list of users “candidates”. Any user can become a candidate and users can only vote for candidates. This greatly reduces the average number of users I have doing pairwise comparison on. Also since storage is expensive on smart contract I will store the candidates as a array of registeredUsers index’s instead of the User struct. My apply method is as follow.

The startVote function is as follows.

The block.timestamp is insecure as a malicious actor with a large amount of computing power could create fake blocks with the wrong timestamp. However I decided this was not of huge concern, as it would cost a lot of energy and money and all the actor would be able to do is start and end the voting a few minutes or seconds early or late.

The Vote function is as follows.

The endVote function is as follows.

The countVotes function is as follows.

In order to save gas I implemented a number of strategies. One method is to only load data from storage once. This is why I used copelandScore.length instead of candidates.length. As copelandScore.length is stored in memory. Memory load costs 3 gas whereas Storage load costs 50.

This however lead to another problem, as the solidity language only allows 7 items of data on the stack. To solve this is reused counter variables and split the count vote into separate functions.

Another way I saved gas was by moving the initialising the p counter as 1 + i. This made it so my I only had to do n(n+1)/2 pairwise comparisons to calculate all the Copeland scores. This can be done because a defeat for one candidate increases the Copeland score of the other. If this was not done it would have taken n^2 pairwise comparison.

Also during the pairwise comparison I reduced gas cost by stopping counting votes when either of the candidate has been rated higher in over half the votes. We can do this because Copeland score does not care about the quality of pairwise victories, but the quantity. This does not reduce the worst case gas cost, however the majority of times it should have significant impact.

Another decision I made was to break the for loop as soon as either candidate i or candidate p’s address has been found. This is because in Copeland score we do not need to know how much more a candidate is ranked higher, only if it is. This again saved gas.

After finding the Copeland score of each user and an array for each candidate of the candidates they defeated. I worked out the second order Copeland of each candidate by summing the Copeland score of their defeated candidates.

One problem that I faced now was that I could not just simply sort the second order Copeland scores. This is because to save gas all my variables stored the candidate list user index instead of the users themselves. This meant if I just sorted the scores, I would lose who had what score. To solve this problem I sorted them, then compared the sorted and unsorted scores to create another array called ranked, which stored the list of candidate index’s, ranked by second order Copeland scores. I used quick sort to sort as it has the best average case time complexity O(n log n), it also has space complexity of log n. Also while testing I realised a error occurred where if multiple candidates had the same second order Copeland score, they would all be ranked to the first candidate in the list. To solve this I added; secondOrderCopeland[p] = copelandScore.length \* copelandScore.length;

The last function to implement was how to weight the candidates. I decided upon creation of the contract the creator can choose options, with developers being able to add more options. By default I added the two simplest options, each representative getting 1 weight, and each representative get decreasingly less weight. The function is as follows.

One other important feature of solidity is that storage variable creation costs 20,000, but resizing the array length to 0 costs 5000. This means that cost of the first run of the system will cost more than the usual amount.

Budget Club

This smart contract had to allow representatives to submit sinks, and preferred budgets. It then had to run my decide budget algorithm. Finally after a final budget is reached, the contract must send out the money according to the budget.

The code for submitting budgets and sinks is as follows.

Only representatives can do these actions. Also to stop representatives from abusing their ability and filling up the number of sinks with their own addresses, I created a variable numOfSinksAllowedToSubmit which is set to sinks / reps + 1 and tracked how many sinks each representatives had already submitted.

The next code to implement was the end budget submit, this can be done after a certain amount of time after voting had ended. With the code as follows.

Next I had to implement the decide budget algorithm. It is as follows.

Again I tried to put as many variables I was using from storage multiple times on the stack and memory variables to save gas cost.

One problem I faced was that there is no built in decimal types in solidity. This is why my calculate utility has been scaled up by 100 times. The larger the scale the more accurately I will be able to calculate utility. It also allows more granularity in coalition size factor and coalition size factor increase. Another problem caused by this is that when negotiating a budget between representative and coalition, the total budget would not sum up to 100. To solve this at the end of negotiating it would give each sink one more unit in the budget until it summed up to 100. This causes the problem that earlier sinks will benefit from this much more than later. However smart contracts are deterministic and there is no way to randomise the orders.

Another implantation decision was to decide the order in which representatives take turns. I decided to leave the order of highest weight to lowest as it was. This was for two reason. First it would cost to reorder it, and because letting the biggest representative form coalitions first should end the negotiation earlier and save more gas.

I also created another function calculate100utility. This returns the utility if the preferred budget and budget are the same. This saved gas as I need this often, and finding out how much the preferred budget overlaps with the budget is expensive.

The negotiation function returned the utility for each agent. After finding the negotiation which returned the best utility for the representative, I then had to create a function negoToBudget which took the utility and the representative and coalition budgets and returned the new coalition budget. To do this first, I took each utility and increased it to what it would be if the coalition size was the quota. I then took each of them away from how much the budgets overlap. This gave me three uints, shardedab which is how much of the budgets overlap, individuala which is how much of the negotiated budget is only in the representative preferred budget, and individualb which is how much of the negotiated budget is only the coalitions current budget.

After working out these 3 variables, the function then starting the creating the new budget. First, all the overlapping budget is wanted by both agents so the new budget has them as well. Then I had to decide how the individual a and b where spread. Due to the way I created my utility function, if two sinks are 20 units below the budget, it generated the same utility as one sink 40 units below the budget. I decided to spread the individual utility proportionally by how far away the other agent’s budget was from the agents.

Club Contract Website

Testing

Functionality

While creating the smart contract I used a test driven development. This meant that before the creation of each club contract, I would create a skeleton of the function and return values of the club contract and then create a series of test that prove that my code meets the functionality.

The benefits of this was that I could quickly check to see if my code was functioning correctly, this combined with the fact that I could see what test where failing, allowed me to quickly find bugs and get rid of them. Another benefit was that by writing tests, it allowed me to understand what features I need to add. Another benefit is that because it runs all the tests constantly, if I make a change it a latter part of the code that changes the earlier, I can make sure that it still passes the test. The cost was that writing these test took a fair amount of time at the start of each module. However the tests have to be written at some time, and by doing it first was able to save me time debugging.

Truffle provides an ability to run tests with the Truffle test command (http://truffleframework.com/docs/getting\_started/javascript-tests) . This allows you to create programs in JavaScript that allow you to make assertion tests. For example here is my registration unit tests.

All functionality tests were passed in the final smart contract, and therefore I will conclude that all the functionality that was described in final design has been implemented.

Evaluation

The two ways I will evaluate my system is how well it scales, and how fair users believe my algorithms to be.

Gas Cost

I will measure how well my system scales by measuring the gas cost of my functions, vs several parameters. To put the gas cost into perspective I will work out the USD value of one gas at the current time of writing. The median gas price is 5 gwei (<https://ethgasstation.info/>), 5 gwei is a unit of measurement for 0.000000005 Ethereum. At the current cost of $704 this means one gas costs $0.00000352. For example an average register function takes around 92819 gas to run which costs $0.32672288.

I will run each algorithm multiple times with random numbers and average the gas cost. Also I will not run functions which are not impacted by any parameters as this does not change how well my program scales.

Change Registration Cost

This has one parameter, the number of registered users.

End Vote

Decide Budget

Fairness