UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

A project report submitted for the award of

Supervisor: Timothy Norman
Examiner: Julian Rathke

Multi Agent Systems on the BlockChain

by Jake Labelle

May 1, 2018

Contents

Chapter 1

Abstract

Ethereum is a cryptocurrency, which allows users to upload smart contracts to the network. Other users can then send transactions to the smart contract. These transactions are validated by miners in return for a miners fee; a certain amount of Ethereum. The miners then run the transaction and update the storage of the smart contract.

Multi-agent systems model how a number of agents, a computer system capable of autonomous action to achieve goals, interact. Each agent will have a social ability, that is each agent will have goals, but can only achieve these goals by interacting with other agents. In deciding how the budget will be split, Agents will need to be able to use Coopearation, as no agent is allowed to choose the budget by himself. Agents will also need Negotiation, as agents will have to convince other agents to support a budget.

My project will be a smart contract builder, that allows a founder to create a decentralised sports club. This will use the Ethereum 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 beforehand 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 blockchain without a terminal.

Chapter 2

Project Problem and Goals

2.1 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 year 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 negative is that currently many systems use the budget negotiation algorithm; sum all the representatives budgets 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.

Also, the current systems run on private servers and databases, and therefore users cannot trust that the owners of the systems are not manipulating the data. Addionaly the servers and databases require an upfront cost.

2.2 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.

Chapter 3

Background Literature

My background reading could be broadly separated into three fields; multi agent systems, Ethereum and smart contract design.

3.1 Multi Agent Systems

There are a number of voting systems each with different desirable properties (Wooldridge, 2009). First off, there was a distinction between social welfare and social choice functions. They both took in a list of preferences from agents, however in social welfare it outputs a list of preferences, whereas social choice outputted a single option. In my system, I will require a social welfare function, as I need to select a number of representatives. For my program, voting preferences are important so that the representatives are representative of all the voters wants. Second order Copeland rating satisfies this. This is where each of the options has a pairwise election with all other options. It then adds the sum of all defeated opponents to see what option is the best. This algorithm satisfies the Condorcet winner condition where w should be ranked first if it would defeat each other option in a pairwise election. This algorithm is Pareto optimal which means that if every one votes $w1 > w2$ then the output should be $w1 > w2$. Lastly, it also mentions the Gibbard-Satterwaite theorem which states that all voting systems are vulnerable to strategic manipulation. For my system, this is made worse by that fact that all transactions are transparent on a blockchain, making the manipulation even easier. However, second order Copeland rating is NP-complete to manipulate which makes it difficult to manipulate.

Also from the book, there was a description on how to run a negotiation algorithm.

The negotiation set, will be an array of sinks to values, where the values add up to the collected money. The negotiation algorithm is where Agent 1 makes a proposal. If it is rejected, then Agent 2 makes a proposal, then 1 etc. However, if no deal is reached after a certain number of rounds, then the players receive a conflict deal. In the book, two assumptions are made; agents seek to maximise utility and disagreement is the worst outcome. However, in my system, disagreement is not always the worst outcome. Another important point is that time is valuable and that agents are impatient. This means a deal w at round t1 is better than deal w at round t2 if $w1 < w2$. The algorithm should create a Nash equilibrium, this means that no agent can increase his utility by changing actions.

Lastly, reading continued with the coalition section of the book. My system would be a weighted voting game. This is where each agent has a weight, we have a quota and a coalition C is winning if the sum of the weights exceeds the quota. A coalition is said to be stable if every agent cannot get a higher utility by defecting. Coalitions can be formed by self interested rational agents who are trying to maximize their own utility (Chalkiadakis, 2012). In particular, dynamic coalition form where coalitions can form and fall apart. One approach is a Markov process, where agents explore sub optimal coalition formation actions and at each stage there is a small probability that they try to move to a new coalition. A player switches coalitions if their expected utility, in the best available coalition, exceeds their current utility. The player also demands as much as they can from the coalition.

3.2   Ethereum

The first modern cryptocurrency was Bitcoin. At its core, Bitcoin is a state transition machine. It has a ledger of account values. It then receives a transaction e.g send 50 Bitcoin from Alice to Bob. If Alice has 50 Bitcoin, the next state is the same ledger but 50 Bitcoin subtracted from Alice and 50 more in Bob's account. Or else it transitions to a error. Also, Alice is able to verify she is sending it from her account by signing the transaction with her private key.

The next problem Bitcoin had to solve, was how to method to reach consensus, to make sure users agree with the set of transactions. To solve, this Bitcoin created a blockchain. This is a linked list of blocks. Each block contains a list of transactions sent in that block and the previous hash of last block. The previous hash is computationally intensive problem to solve, and so miners race to solve it and can put the next block on the blockchain. The system rewards the miners with Bitcoin to incentivise more computation to secure the network.

The hashing algorithm must be difficult to prevent a Sybil attack. This is where a peer to peer network is comprised, by creating a large number of pseudonymous identities. The reward is given to prevent a 50% attack. This is where one person or group controls 50% of the mining power. This would allow them double spend and send the same bitcoin multiple times. Smart Contracts allow you to run scripts on the blockchain. Bitcoin can make basic smart contracts but has major limitations. First, it is not turning complete, for example loops cannot be implemented. Another problem is a lack of state, Bitcoin is either in or not in a users account.

To solve this problem Ethereum was created (Buterin, 2018). This functions the same way as Bitcoin except the blockchain has a built in Turing complete programming language (solidity). Also, transactions can now be sent with more data, this allows you to send parameters to functions. Furthermore transactions have a STARTGAS field. This is how much gas the user is willing to spend on the transactions, and the transactions will fail if it goes over the limit. Lastly, transactions have a GASPRICE field, which is how much the Ethereum the user is willing to pay per transactions. Miners will prioritise transactions with higher gas prices, as they will receive more Ethereum for their work.

3.3   Smart Contract Design

Firstly, I found the benefits of using smart contracts. They are useful whenever you need applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference (Foundation, 2018).

I also did research on writing smart contracts, and found the Truffle development environment. I then read through the Pet Shop tutorial (Truffle, 2017) . It showed how to set up the environment, create new contracts with functions and variables, and how to migrate and deploy the contracts. Once the contracts are deployed on Truffle's test network, it gives a number of addresses with Ethereum on them in order to test your contracts. These addresses can then interact with the functions on those contracts. Once you create the contract and deploy them, an interface can be made for the contract with web3. On my system, you will be able to register and vote on the interface, so that users do not have to use the command line. In order to use your Ethereum on a website, downloading the extension Metamask allows you to set your default account, and when you click register on the interface, it will send funds from that account.

Also, I researched how other applications on the Ethereum network had reduced

their gas cost: one such app was a distributed social network. The majority of the gas consumption on smart contracts come from storing data. To reduce the amount of data stored, he used IPFS and Stateless Smart Contracts (Childs-Maidment, 2016). IPFS take your key-value pair and store it as a hash, and then when access is needed to your key-value pair, you send the hash to IPFS, and it will return the key-value pair. IPFS is also decentralised so it does not compromise the trustless nature of smart contracts. Stateless Smart contracts have two parts; dumb functions and filters. A dumb function will take the parameters required but will do nothing. Then when you want to access the data on your interface, since all transactions are public, using InputDataDecoder, find what parameters were passed to the contracts functions.

However, there are trade-offs with this approach. The first being tampering; the person who controls the interface could lie about what was sent, although users could validate the transactions themselves which reduces the problem. It could also be a positive, as anyone can build a filter which could add extra data to improve user experience e.g media, metadata. Another problem is that smart contracts do not have access to all the transactions and will not be able to see your data. Lastly, if your contract has no events, it must process all transactions to find your data, but this can be mitigated by firing off random empty events. In my system, I will use a mix of standard and stateless functions. The voting and registering will be standard, but for functions such as messaging, gas can be saved by making it stateless.

This was a smart contract Merriam (2017) had put on the blockchain that allowed you to pay someone to run functions after a certain number of blocks. This would be useful, as I need time limits on voting, submitting a budget, term limits and registration and I need to run a function periodically to check that the time limits had not been reached yet.

I also needed to research common vulnerabilities in Smart Contracts. One common attack is Reentrancy (Grincalaitis, 2017). This is where users can interrupt the normal running of the function. Therefore, in my system before any actions occur, I will set the conditions to false. Another attack is under and overflows, for uint256 any number above 2**256 will be set to 0 and 0 - 1 will be 2**256. Lastly, any function which is not changing the state of the application should be labelled as constant, to save gas. In order to work out how much each transaction will cost, I will use EthGasStation (2018) to find out median gas price and CoinMarketCap (2018) to work out the cost of Ethereum in USD. I will use coinmarketcap over trading sites like Coinbase as coinmarketcap uses a weighted average and is therefore more reliable.

Chapter 4

Final Design

4.1 How Registration Fee Collected

One of my goals is 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	Cannot unless given the money by the budget.
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 happened.	Yes
Able to prove you're the one who registered	Yes	Yes
Transparency (all users can see in and outflows)	No	Yes

The benefits of the Clubs and Charites Account is that it costs zero 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 considerable time and effort. Also, if one of the signatories is unavailable, then the club would be unable to access the funds. Compared with the Smart Contract, this is done automatically. Another problem is that although the registration is legally non-repudiable, a member would have to take the club to small claims court to enforce it. Lastly, the account is viewable by the signatories, therefore not all members can not confirm how much money has been gathered and where it has spent. However, with 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 instructions used) used down. Also, you must be careful to monitor the Gas Price (cost miners are charging 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 registration.

4.2   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.

4.3   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 solved this problem.

	Borda Count	Second Order Copeland	Dodgsons method
Pareto efficient	No	Yes	Yes
Independence of Irrelevant Alternatives	Yes	No	No
Non dictatorship	Yes	Yes	Yes
Condorcet Condition	Yes	Yes	Yes
Complexity to work out	k	k^2	NP-Hard
Complexity to Manipulate	k	NP-Complete	NP-Hard

Borda Count is the cheapest to run, but due to the ease of getting the other users votes, a linear scale to manipulate the outcome is not acceptable. Dodgsons method was made to be hard to manipulate. However, to do this it is a complicated algorithm and would cost alot of gas to run. Second order copeland time complexity is quadratic, although this is bad as gas cost will rise exponentially, it is worth it to stop other users being able to manipulate the result easily.

4.4 How to Start and End Voting

Voting should start a term length after the last voting had ended and the voting should end after a preset time after the voting has started. Both term length and voting length are set when the club contract is made. However, Smart Contract functions cannot run for a long time, and so to check if enough time has past, someone will have to call the function after a set time. I have two options for calling the time sensitive 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. If it is, then users could send the function as quickly as possible to mitigate the problem. If the club has a small number of users, the first option might be better, as there is a greater chance all the users may forget to end the voting period.

4.5   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 pairwise 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 it can have a large impact on the cost of the contract, if users wish to change it, a super majority should be acquired.

4.6   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.

4.7   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 widely distributed and each manager will have more time to focus on a sink with less responsibilities. The disadvantage is that communication 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 method for submitting sinks. I have identified three

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 for. 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 likely to be abused, as representatives are likely to have a greater attachment to the club. The last option should prevent this 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 the second option.

4.8   Budget Negotiation Algorithm

The first and easiest idea for deciding the budget was to take each representative's budget, sum each one and divide by number of representatives. This would be cheap to run and sounds fair. The problem that occurs is 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 to 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.

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 fewer 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 it is a better utility then all offers. With the algorithm as follows;

```
Agent[] Agents;
Sink[] Sinks;
```

```
Coalitions[] Coalitions;
init Coalitions;
round = 0;
while(winningCoalition = false) {
        player = Agents.get(round % Agents.size());
        //its advantageous to be in a bigger coalition
        currentUtility =
    utility(player,player.coalition.budget,player.coalition.size);
        currentCoalition = myCoalition;
        for(Coalition c: Coaltions) {
                proposal = negotiate(player,c);
                if(currentUtiltiy < proposal)
                        currentCoalition = c;
                        currentUtiltiy = proposal;
                }
        }
        join current coalition, leave old one
        round +=1
}
```

4.8.1 Calculating Utility

The next step was to create a function which took a representative's budget , their current coalitions budget, and the current size of the coalition and returned the representative's current utility. I wanted the utility to capture that users are happier the more their budget overlaps with their final budget. 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 follows, with a being the representative and b being the coalition.

$Utility = (\sum_{n=0}^{k} max(a.budget[n].b.budget[n]))/quota+1-a.weight-b.totalWeight$

Although representatives joining coalitions reduces the number of infinite loops, any of them are unacceptable in my algorithm. To solve this I propose 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 in 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 a more informed decision. However, to work out the single negotiation would scale factorial, and therefore unusable with a medium number of representatives whereas the second method is a constant effect on the number of instructions and therefore scales much better. For this reason, I will use the second method, with the new utility function as follows.

$Overlap = (\sum_{n=0}^{k} max(a.budget[n].b.budget[n]))$

$Utility = ((coalitionSizeFactor * Overlap/(quota + 1 - b.totalWeight) + ((1 - coalitionSizeFactor) * Overlap)))$

I also need to set the coalition size factor when the algorithm is started, and need to increase it each round.

4.8.2 Alternating Offers Game

The next decision was how representatives would negotiate with coalitions. I used the model of the alternating offers game with impatient agents. I will therefore use back propagation to work out what the optimal starting offer is for both representative a and coalition b . First I work out $\bar{a}, \bar{b}, \underline{a}, \underline{b}$, these are in order, the max utlity a can get from joining b, the least utlity a can get from joining b, the current utility that a has in its current coalition, and the current utility of b.

$$\bar{a} = utiltiy(a.budget, a.budget, b.totalWeight + a.weight)$$
$$\bar{b} = utility(a.budget, b.budget, b.totalWeight + a.weight)$$
$$\underline{a} = utility(a.budget, a.currentCoalition.budget, a.currentCoaltion.totalWeight)$$
$$\underline{b} = utility(b.budget, b.budget, b.totalWeight)$$
$$\bar{ab} = \bar{a} + \bar{b}$$

The next step is to now check if $\bar{ab} < \underline{a} + \underline{b}$, if so, then we can stop the negotiation as there is no way for both the representative and the coalition to benefit. We also need the variables x and N, which are the cost to run one turn of negotiation, and the number of turns that are run. With this information we can generate this tree. We will set $N = 4$ in order to make it easier to display.

A

y |

B

z

$$\bar{a}b - y, y \qquad A \qquad \underline{a}, \underline{b}$$

g

$$z, \bar{a}b - z - x \qquad B \qquad \underline{a}, \underline{b}$$

f

$$\bar{a}b - g - 2x, g \qquad A \qquad \underline{a}, \underline{b}$$

$$f, \bar{a}b - f - 3x \qquad \underline{a}, \underline{b}$$
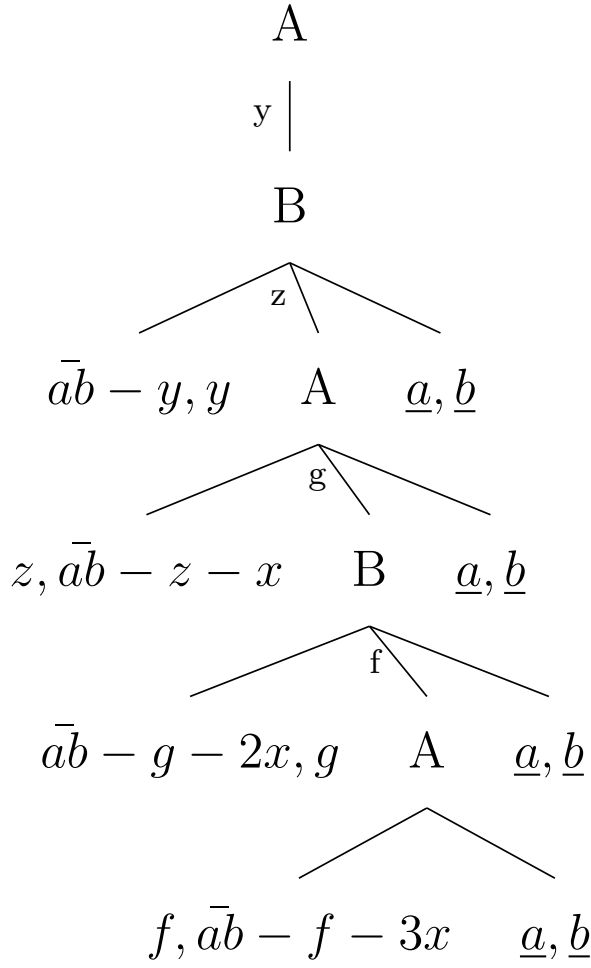
From this we can find n, that is the turn in which one of the agents can offer the other agent a score higher than the conflict deal for the both of them, $\underline{a}, \underline{b}$. We can rearrange $\bar{a}b - (n-1)x - \underline{a} - \underline{b} - 2 > 0$ to $(\bar{a}b - \underline{a} - \underline{b} - 2)/x = n$. If $n > N$, then we can set $n = N$, as it cannot go over that many turns.

The next step is to calculate the free utility. That is how much extra utility, that the agent can take for himself, on the turn that the offer is better than the conflict deal. This is $freeUtility = \bar{a,}b - (n-1)x - \underline{a} - \underline{b} - 2$. Now there are two cases, if n is even then b takes the free utility, and if it is odd then a takes it. Then for the remaining turns, the offeror takes $x - 1$ extra utility, and the offeree gets 1 extra utility. Therefore I get this function, for the optimal utility for each agent.

If n is even

$$aUtility = \underline{b} + 1 + freeUtility + (((n/2) - 1) * x) + 1;$$

$$bUtility = \bar{a} + \bar{b} - aUtility;$$

If n is odd

$$bUtility = \underline{a} + 1 + freeUtility + ((((n+1)/2) - 1) * x);$$
$$aUtility = \bar{a} + \bar{b} - bUtility;$$

The last flaw I needed to fix was that the algorithm could give a utility higher than $\bar{a}$, which should be impossible. Therfore, if either aUtility or bUtility is over $\bar{a}$, reduce it to $\bar{a}$ and set the other one to $\bar{b}$.

4.9   Club contract Website

The design of this will be very minimalistic. This is because anyone 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.

Chapter 5

Implementation

5.1 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 it 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 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.

5.2 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. For example, 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. Which I imported into my other contracts.

5.3   Registration Club

This contract needs to take a registration cost when it is created, and then allow users to pay that much Ethereum to register. It subsequently needs to store that user in an array, as a Struct containing the user's name and address

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 mapping which takes a registered user index and returns their current suggestion that user has for the new registration cost. When a new user is 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 iterate through all the submissions and check if a super majority of the same submissions has been reached.

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() for free as it is constant, and therfore does not need to be uploaded to the blockchain, and seeing if the return is a value different than the current registration cost.

5.4   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. Thus greatly reducing the average number of pairwise comparisons. Also, since storage is expensive on smart contract I will store the candidates as a array of registeredUsers indexs instead of the User struct.

Another problem was getting the current time for start and ending voting. This is because 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 for very little impact (starting and ending the voting period a few second earlier or later).

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.

5.4.1 Voting Algorithm

I also had to implement the countVotes function. This took all the votes and ranked them in order of their second order Copeland score. 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, leads to another problem, as the solidity language only allows 7 items of data on the stack. To solve this, I reused counter variables and split the count vote into separate functions.

In my algorithm, I had a inner loop and a outer for loop, which incremented i and p respectively, I then compared candidate[i] and candidate[p]. To save gas, I initialised the p counter as $1 + i$. This made it so that 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.

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 particular 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 indexs, ranked by second order Copeland scores. I used quicksort 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 whereby 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 next function to implement was the pairwise comparison function. To save gas, I decided 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. 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.

5.4.2   Weight Algorithm

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.

5.5   Budget Club

This smart contract had to allow representatives to submit sinks, and preferred budgets. It then had to run my decide budget algorithm. Ultimately, after a final budget is reached, the contract must send out the money according to the budget. I first implemented submitting budgets and sinks . 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. Also, users submitted budgets, will sum up to 100 and be a percentage of the money collected. This is in case more money is received after some budgets have been submitted.

The next code I implemented was the end budget submit. This can be done after a certain amount of time after voting had ended. I also checked that the list of

representatives was not empty to make sure that users did not run this function after it had already been run once.

5.5.1 Budget Algorithm

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 utility has been scaled up by 100 times, and it is now between 0 and 10,000. 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 implementation decision was to determine 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 reasons. Firstly, 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 needed this often, and finding out how much the preferred budget overlaps with the budget is expensive.

5.5.1.1 Negotiation

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 achieve 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 started 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 were 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 agents budget was from the other agent's budget.

5.6 Sending Out Money

Lastly, I had to implement the function which sent out the money to the owner of each sink. I decided to separate this function from my decide budget function. This was because a user might want to subsidise the club, as so users might want to wait before they send out the money. This had two problems; First, anyone can call the send out function, and so a malicious or ignorant user might send it out any way. The other problem is that this means I have to store the final budget in storage which is expensive in gas.

5.7 Club Contract Website

The first step was to create the html for the website. I created 5 pages; registration, home, settings, vote and representative, with a navigation bar at the top of each page allowing you to access each one. Registration allowed you to register. Home showed all registered users and allowed you to apply to be a candidate. Settings showed all the settings of the Smart Contract and allowed you to start and end voting and budget submit. Vote showed all candidates and allowed you to select them sequentially and submit the vote. Representative allowed representatives to submit sinks, view all submitted sinks and finally submit their preferred budget. The next step was to create the javascript. In order to interact with the Smart Contract you can use a library called web3.js. To use this library you must provide it with the address of your deployed contract and its ABI. Truffle provides a library called truffle-contract.js. This library allows you to automatically use the deployed address and ABI of the contract which you have migrated. This is useful during

development as the ABI will change every time the contract changes.

Once I connected to the Smart Contact, I then had to write functions which called the various functions on my contract. One important characteristic of the web3 library is that smart contract return values are returned as promises. A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that its not resolved. This is useful as this is non-blocking, and while the Smart Contract runs the function, the user can do other functions on the website.

Another method that I needed to implement was what account the users would be sending transactions from. I decided to do this using Metamask. Metamask is a web plugin which allows you to sign into your Ethereum account and then inject that account into the javsscript, using their library.

Chapter 6

Testing

6.1 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 tests were 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. A further advantage, is that because it allows me to run all the tests constantly, if I make a change to a latter part of the code, that might modify the results the earlier functions, I can make sure that it still passes the test. The cost, was that writing these tests, took a fair amount of time at the start of each module. However, the tests have to be written at some point in 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 (Truffle, 2018). This allows you to create programs in JavaScript that allow you to make assertion tests. For example, here is my registration unit tests;

```
var BudgetClub = artifacts.require("BudgetClub");
contract('BudgetClub', function(accounts) {
  it("User registration", function() {
    return BudgetClub.deployed().then(function(instance) {
      return instance.register("jake",{value: 600,from:
    accounts[0]}).then(function() {
        return instance.getRegisteredUsersLength({from:
    accounts[0]}).then(function(userLength) {
```

```
        assert.equal(userLength, 1, "User added");
        return instance.register("hard",{value: 600,from:
accounts[0]}).then(function() {
        return instance.getRegisteredUsersLength({from:
accounts[0]}).then(function(userLength) {
            assert.equal(userLength, 1, "User address added twice");
            return instance.register("jake",{value: 2,from:
accounts[0]}).then(function() {
            return instance.getRegisteredUsersLength({from:
accounts[0]}).then(function(userLength) {
                assert.equal(userLength, 1, "User hasnt send enough gas");
            });
        });
    });
    });
    });
    });
});
});
```

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.

6.2   Scaling

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 (EthGasStation, 2018), 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. I will not run functions which are not impacted by any parameters, as this does not change how well my program scales.

6.2.1   Change Registration Cost

This has one parameter; the number of registered users. I will graph the worst case situation, this is where no registration cost has a supermajority and therefore

does not return a value until looping through all the costs suggestions. This gave me this data;
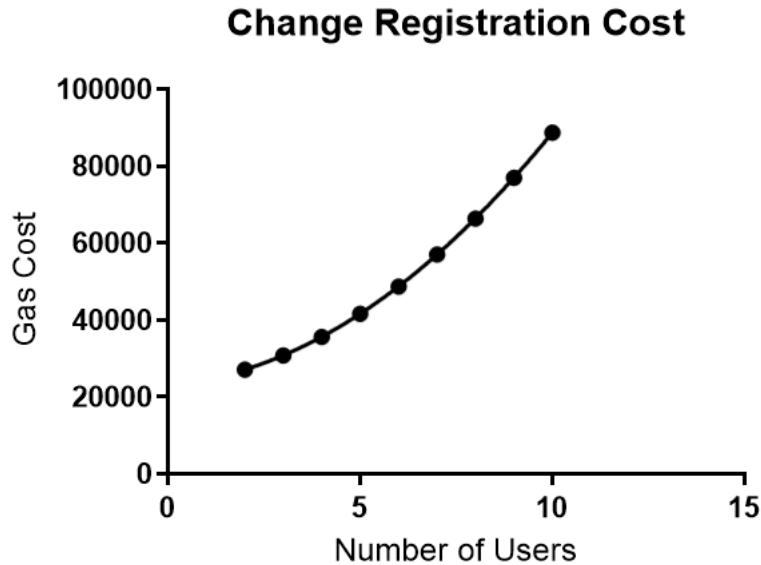
I preformed a quadratic regression. This gave me the formula $23235 + 809.9x + 573.6x^2$ with x being the number of users.. From this I could see that although the algorithm scales exponentially, the exponential coefficient are relatively low, and the majority of the gas cost comes from the constant coefficient unless there is a very large number of users.

6.2.2   End Vote

This has two parameters; number of votes and number of candidates. I will assume that each vote gives a full ranking of candidates. This gave me this data.;
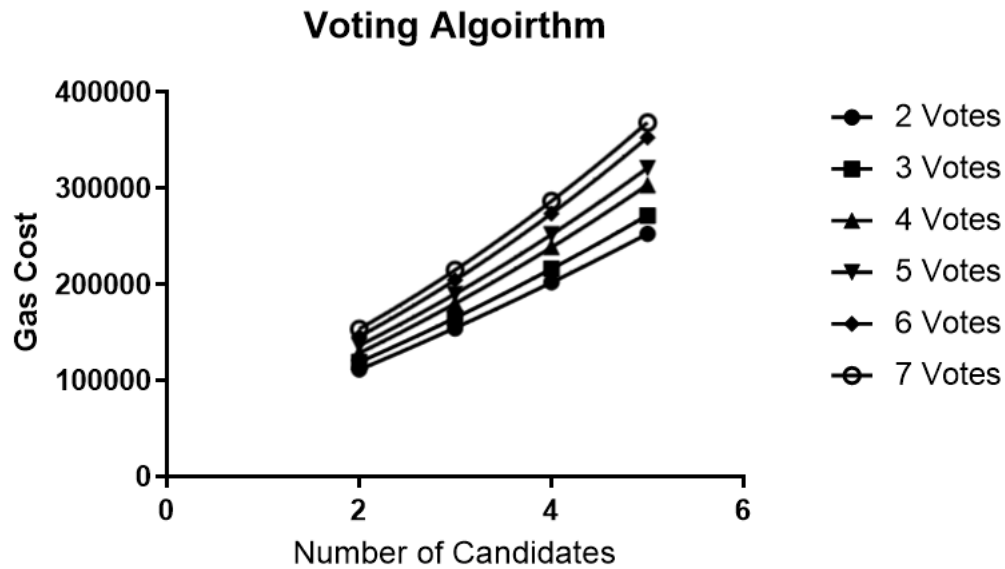
I then performed quadratic regression on each number of votes. This gave me these formulas with x being the number of candidates.

$$2VotesGasCost = 34949 + 34583x + 1785x^2$$
$$3VotesGasCost = 41603 + 33925x + 2415x^2$$
$$4VotesGasCost = 45277 + 34952x + 3329x^2$$
$$5VotesGasCost = 52040 + 34431x + 3868x^2$$
$$6VotesGasCost = 59087 + 33107x + 5122x^2$$
$$7VotesGasCost = 61946 + 34583x + 5139x^2$$

From this we can see that both increasing number of votes and number of candidates increases gas cost. However, we can also see that when there are more candidates, increasing the number of votes, increases the gas cost by more. We can also see that increasing number of candidates has a much greater impact than increasing number of votes.

This should scale well, as there should be considerably more votes than candidates. However there is no limit on the number of candidates and so the number of candidates could be the number of registered users. One way this could be solved is by removing all candidates at the end who have received 0 votes. This should stop users who are applying to representatives, to increase the gas cost of the voting algorithm. A problem with this solution is that it will increase the base cost of the function.

6.2.3 Decide Budget

This has five parameters number of representatives; number of sinks, coalition size factor, coalition size factor increase, and quota. For all my scaling tests, I will set quota to a majority of the total weights and collation size factor increase to one. The first scaling test was how number of representatives affected gas cost. I set number of sinks to two and coalition size factor to 20. This gave me this data;
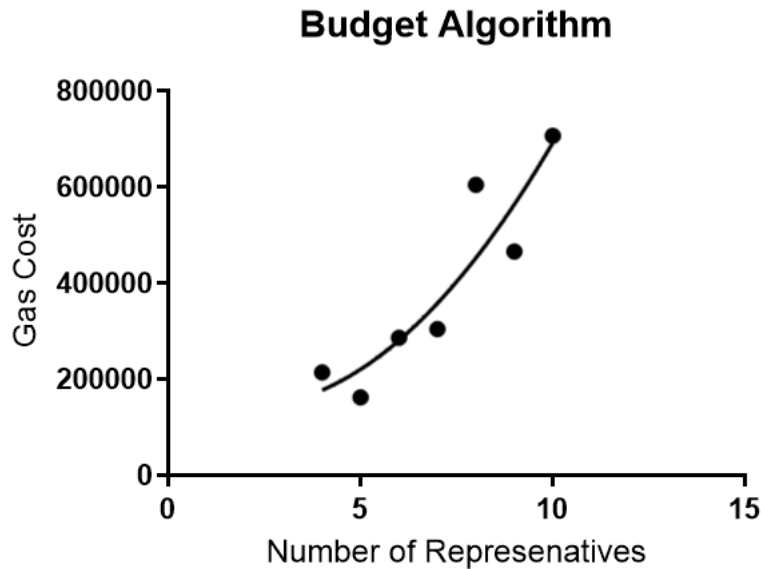
I performed quadratic regression on the data again giving me this formula. With x being the number of representatives. $GasCost = 17544933592x + 8521x^2$

From this I could see that my budget algorithm would not scale well, even with a moderate number of representatives. For example, 15 representatives would cost 1528044. Another pattern I noticed was that if n is number of representatives and n is odd, then it costs less gas that $n1$. This can be explained, by my scaling test, setting the quota to be the majority of representatives, and so n and $n1$ would need the same quota, but n had more options to form coalitions with.

The next test was to see how the number of sinks affected the gas cost. I also created two series; one with 5 representatives and one with 7, to test if that affected the impact of increasing sinks. The data was as follows;
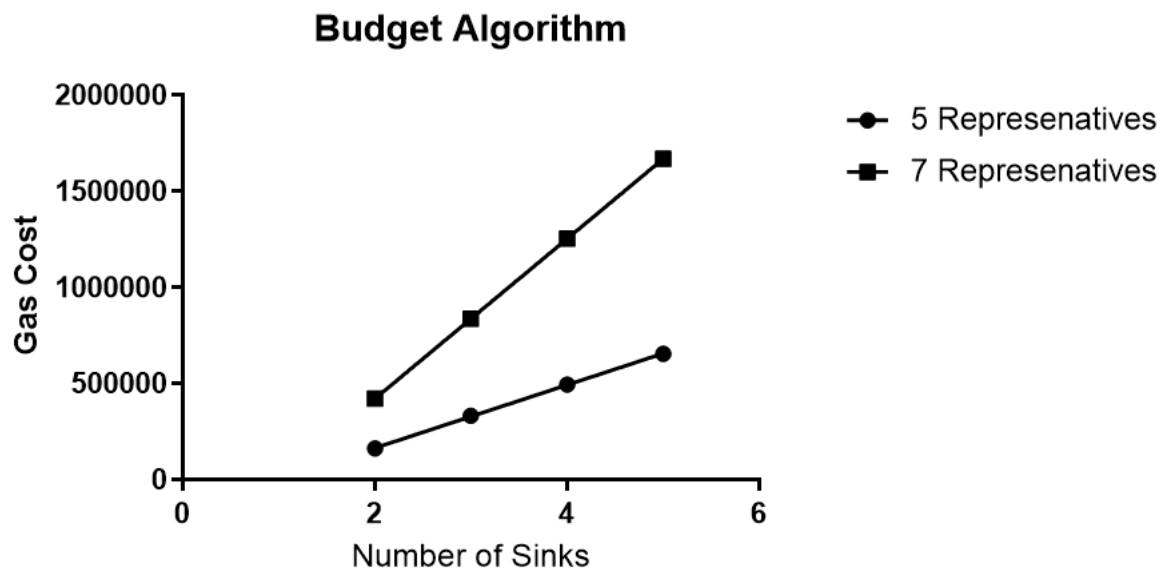
I then performed linear regression, giving me these formulas. With x being the number of sinks;

$$5 Representatives Gas Cost = -162873 + 163827x$$
$$7 Representatives Gas Cost = -162873 + 163827x$$

From this I could see that increasing the number of sinks linearly increased the gas cost. However, when there is more representatives, the slope is much steeper. The last test was how coalition size factor affected the gas cost. Again, I created two series; one with 5 representatives and one with 7. This gave me this data;

Budget Algorithm
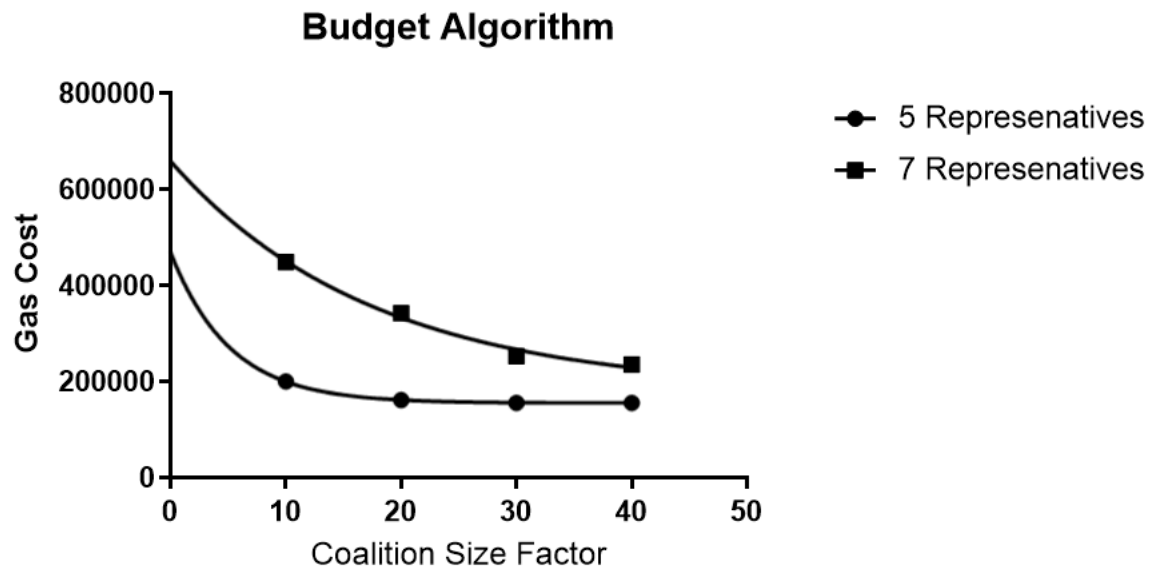
Then I performed a exponential decay regression. This created these functions. With x being the number of representatives;

$$5RepresentativesGasCost = (469581 - 155443) * e^{(-0.19x)} + 155443$$
$$7RepresentativesGasCost = (658793 - 180286) * e^{(-0.05705x)} + 180286$$

From this I can see that that increasing coalition size factor decreases the gas cost. The plateau is reached when coalition size factor is so big, that agents always join the coalition with the biggest size. Therefore, coalitions can not be made any faster.

The coalition size factor should be chosen to keep the gas cost down to manageable level but care must be taken to not make it too large, that that larger coalitions are able to take all the utility in the negotiations. For example with 5 representatives, with a 30 starting coalition size factor, each agent just joins the largest coalition. Another point to notice, is that budget negotiations with less representatives reach a plateau from increasing coalition size factor much quicker. From this we can see that the more representatives, the larger we should start the coalition size factor, but with a small number we risk making negotiations too one sided.

6.3   Fairness

6.3.1   Voting Algorithm

To measure my voting algorithm fairness, I created a google forms questionnaire. It asked users to rank crisp flavours from 1 to 5. It then used all the previous answered questions and the current users ranking with my voting algorithm and showed the user the current result of the voting algorithm. It then asked the user to rate the result Very Fair / Fair / Dont Know / Unfair / Very Unfair. To allow me to return a result to the first user, I also created 5 dummy votes.

The reason I choose to collect crisp flavour votes is that it is non-sensitive data, that users would be able to quickly be able to answer, and also because its a subject the users know about, and the user will care about the result and therefore be able to put how happy he is with the result.

I then sent the google forms link to the computer science group chat. 12 users answered my survey. The modal result was Fair and the median result was 4.2 (Very Unfair being 1 and Very Fair being 5). I will evaluate my voting algorithm as being fair enough, however 12 users is a small sample size and so more research will need to be done.

6.3.2   Budget Algorithm

To measure my budget algorithm fairness, I also created a google forms questionnaire. This showed users all the budgets of the representatives and showed the result of my algorithm. It then asked users to rate the result with the same Likert scale I used with my voting algorithm.

Again, I then sent the google forms link to the computer science group chat. 9 users answered my survey. The modal result was Fair and the median result was 3.8. Although most users choose fair, a few users thought the result was unfair. I need to do more research to change my algorithm, ensuring more users believe the result to be fair and to find out what is currently wrong with my algorithm.

Chapter 7

Conclusions

The System produced solves the current problem with clubs, where all the money is collected and spent by one or a few people. This problem is made worse at university clubs, where is its likely that a first year student has never seen the club before and therefore has no trust in the club or its leadership.

My solution is a smart contract which is run on the Ethereum network. Users can register, vote and decide the budget by sending transactions to the contract. Users do not have to trust the integrity of the smart contract, due to the Ethereum miners who validate every transaction.

One problem added, was that due to the complete transparency, malicious users could use the information of other users' votes and budget to manipulate the result of a election or final budget. I had to choose between simple algorithms that could be easily manipulated and cheaply run or algorithms that were hard to manipulate but cost more to run. I choose the second option, but tried to reduce the gas price as much as possible.

My system can scale well for users registering and voting. Although number of representatives and candidates must be kept low or the gas price starts to become too high. It can also be mitigated by increasing the coalition size factor, but this makes the negotiation pointless. However, for the last couple of months the gas to USD value has been trending down and so in the future may allow for more representatives.

Throughout the year other turing complete Cryptocurrencies were created, such as EOS, that claimed to be able to handle a much larger number of transactions then Ethereum. This meant that the cost to run programs on their network was much cheaper. The problem is that distributed systems are easy to scale when there are so few users. Ethereum used to be as cheap as EOS. Another problem

is that EOS has a lot less computing power securing the network with mining, and so a 50% is a lot easier. Lastly Ethereum already has a large number of distributed applications on the network, and will likely stay the main blockchain for distributed applications due to the network effect. I have also written a simple html and javascript interface for my smart contract. This allows users to interact with the smart contract without a command line.

Chapter 8

Future Work

One problem with my current system, is the html interface, although functional and more accessible than the command line, is still very basic and is not easy to use.

Another part to add to the system, is the ability to raise and distribute money during the year with a subset of users e.g some players in the team are going to a tournament. Now all money is raised at the start of the year by everyone and then distributed to managers.

Another addition would be a way to monitor the managers of the sinks. Instead of the all the money being sent to the managers, managers would have to start a vote (maybe only among representatives), and ask for a certain amount of money for a particular task. At the moment managers could get the money at the start of the year and just leave with the funds.

Another problem is that users are all cleared at the end of the term, and then they can register again for the next year. This is a problem, as gas is wasted deleting then storing their address and name again. This could be solved with a reregister function.

Also, if the second order Copeland algorithm or my budgeting algorithm can be verified in less time than it takes to solve, then the system can scale much better and use a lot less gas. The users can read the input, solve it on their computers, and upload the correct result to the smart contract. The smart contract can then verify the result and if it passes, set the result. However, I do not know if this is possible for either of the algorithms.

I also planned to secure my contract, although I tested the program to try and find bugs, I should use a formal method for system modeling, to prove users cannot reach problematic states.

Chapter 9

Appendix

9.1 Project Brief

9.1.1 Problem

When a club wants to raise money at the start of a year, they charge an upfront registration fee. The collected money is deposited in a bank account and then used to subsidize/pay for items the club needs. The problem arising from this system, is that the club members cannot see the allocation of the funds. They will be unaware of where the money has been spent and who benefits from the payment. This becomes more critical when a club has various divisions, which may benefit from varying payments. I.e. the top A team could be subsidised, whereas the bottom teams may pay more, (training facility costs etc) Also the financial decisions are made by the bank account holder or a small select committee.

9.1.2 Goal

My solution to this would be a smart contract builder in which a club manager could create a club smart contract. This would allow him to choose the initial registration cost, voting system and other similar parameters. The manager would then share the contracts public key with all prospective members. The club smart contract would allow users to send Ethereum to the contract, to become registered for a year. This gives the registered user privileges within the club, such as voting on budget proposals and electing committee members. There would also be side
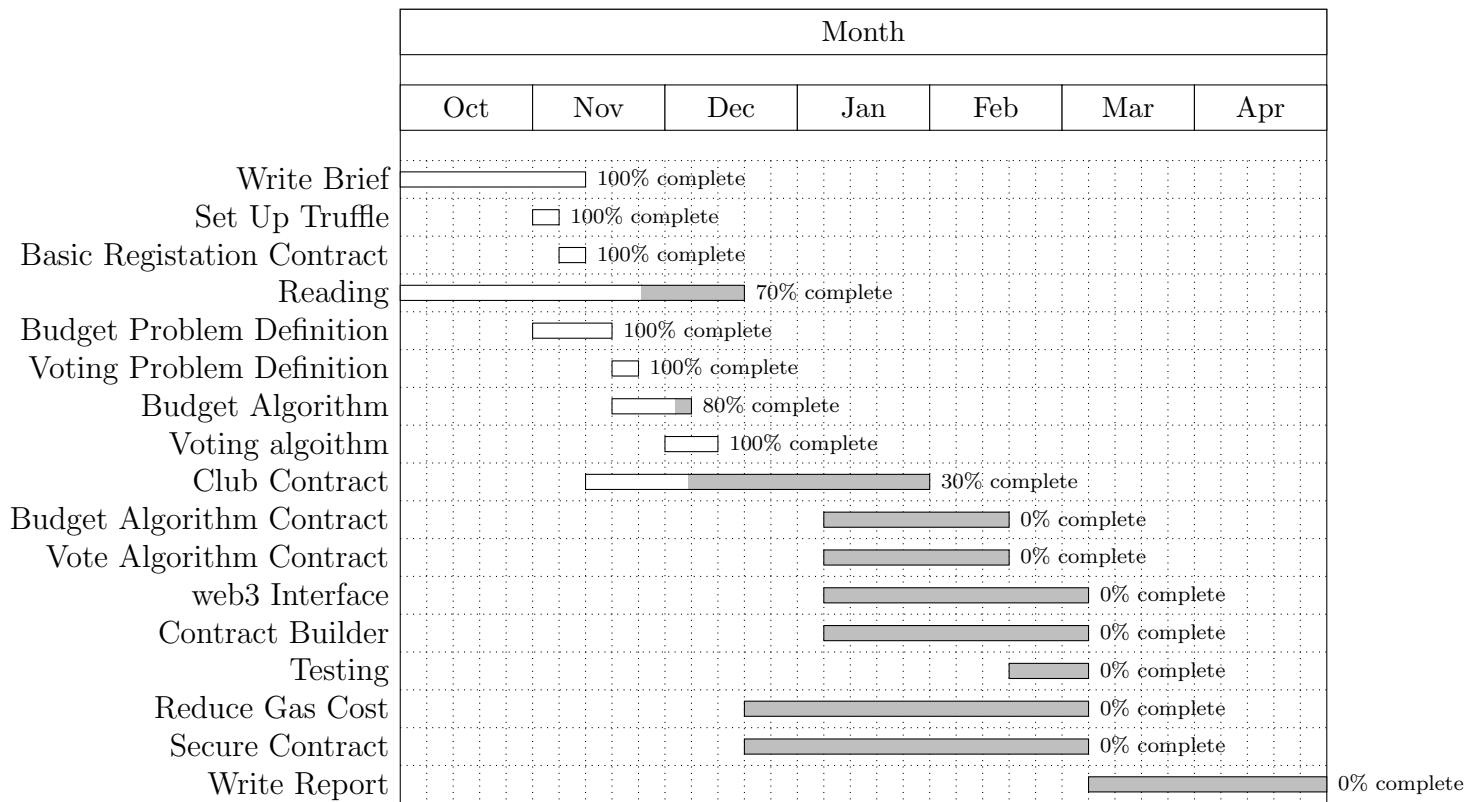
groups to allow club members to pay for something which does not necessarily involve the whole club. The side group can also ask for subsidizing from the main club, but this would be subject to a vote. Another difficulty we could address is that by writing in the smart contract that all money left over at the end of the year will be rebated to the members. Knowing this, members may be more willing to pay a larger registration fee initially. In order for the users to interact with the smart contract, I will also create a front end website. Users can access by entering in the clubs smart contract address and by having the extension MetaMask installed. Another goal of my smart contract will be security. No one should be able to access the funds gathered, and the funds should only be sent to addresses which are approved beneficiaries. Also, only members who have registered, should have the right to vote on where the money should be spent. Another goal will be to keep the gas price down (the cost to run smart contracts), this can be done by only storing and writing necessary data.

9.1.3 Scale

This smart contract would be useful for any group of people who want to pool money together and need a way of democratic way of choosing how the funds are allocated and need a trustless platform. However, I will focus my project on a sports club which has a hundred or so members. I will also implement a number of voting and negotiating algorithms, which can be set either when the contract is created or with a super majority. I will produce a simple smart contract builder, that will allow you to set a few parameters. I will also create a front end written in HTML that uses web3, an Ethereum API to interact with the blockchain. I will also assume the user is using MetaMask, a chrome extension which allows the user to set their Ethereum account while in a browser.

9.2 Gantt chart

9.2.1 Planned Schedule

	Month						
	Oct	Nov	Dec	Jan	Feb	Mar	Apr

Write Brief — 100% complete

Set Up Truffle — 100% complete

Basic Registation Contract — 100% complete

Reading — 70% complete

Budget Problem Definition — 100% complete

Voting Problem Definition — 100% complete

Budget Algorithm — 80% complete

Voting algoithm — 100% complete

Club Contract — 30% complete

Budget Algorithm Contract — 0% complete

Vote Algorithm Contract — 0% complete

web3 Interface — 0% complete

Contract Builder — 0% complete

Testing — 0% complete

Reduce Gas Cost — 0% complete

Secure Contract — 0% complete

Write Report — 0% complete

9.2.2 Actual Progress

			Month			
Oct	Nov	Dec	Jan	Feb	Mar	Apr

Write Brief — 100% complete
Set Up Truffle — 100% complete
Basic Registation Contract — 100% complete
Reading — 100% complete
Budget Problem Definition — 100% complete
Voting Problem Definition — 100% complete
Budget Algorithm — 100% complete
Voting algoithm — 100% complete
Write Registration Tests — 100% complete
Registration Club Contract — 100% complete
Write Voting Tests — 100% complete
Write Budget Tests — 100% complete
Budget Algorithm Contract — 100% complete
Vote Algorithm Contract — 100% complete
web3 Interface — 100% complete
Contract Builder — 80% complete
Reduce Gas Cost — 100% complete
Secure Contract — 50% complete
Write Report — 100% complete

9.3 Archive Table Of Contents

```
/
├── build
├── contracts
├── migrations
├── node modules
├── test
└── src
    ├── css
    ├── fonts
    └── js
```
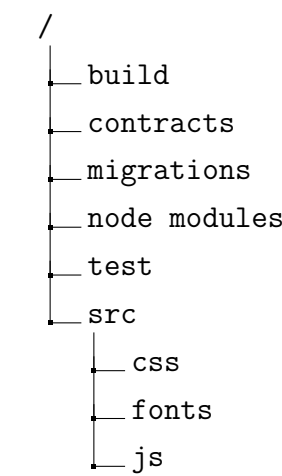
Bibliography

Vitalik Buterin. Ethereum white paper, 2018.

Georgios Chalkiadakis. Computational aspects of cooperative game theory, 2012.

James Childs-Maidment. Stateless smart contracts, 2016.

CoinMarketCap. Cryptocurrency market capitalizations, 2018.

EthGasStation. Gas price, 2018.

Ethereum Foundation. Build unstoppable applications, 2018.

Merunas Grincalaitis. The ultimate guide to audit a smart contract + most dangerous attacks in solidity, 2017.

Piper Merriam. Etherum alarm clock, 2017.

Truffle. Pet shop tutorial, 2017.

Truffle. Javascipt tests, 2018.

Michael Wooldridge. An introduction to multiagent systems, 2009.